Design of a Vision System for an Autonomous Underwater Vehicle

Daniel Loung Huat Lim

November 1, 2004

Bachelor of Engineering Honours Thesis

Mobile Robotics Laboratory,
Centre for Intelligent Information Processing Systems,
School of Electrical, Electronic and Computer Engineering,
University of Western Australia

Supervisor: Associate Professor Thomas Bräunl

Letter of Transmittal

20 Dongala Way
Ferndale WA 6148

1st November 2004

The Dean
Faculty of Engineering, Computing and Mathematics
University of Western Australia
Crawley WA 6009

Dear Professor Mark Bush,

It is with great pride that I submit this thesis entitled *Design of a Vision System for an Autonomous Underwater Vehicle* to the University of Western Australia as partial fulfilment of the requirements for a degree of Bachelor of Engineering with Honours.

Yours Faithfully,

Daniel Loung Huat Lim

Abstract

The enormous potential of autonomous underwater vehicles, AUV's, in marine applications, has spawned a major escalation in the number of research projects involved with autonomous control of submersible vehicles. Hence, there has been a definite trend toward the use of vision systems for robust autonomous guidance.

The primary role of these vision systems, is to process and extract information from an image, to provide an AUV with navigational decisions. In underwater systems such as the ocean, dynamic lighting conditions and turbulent water can greatly affect the images captured. These anomalies must be addressed and accounted for when information is extracted from the images in order to improve accuracy.

The main goal of this project, was to create a vision system capable of carrying out visually guided tasks on an AUV, and provide the necessary functionality for pipeline following. The first vision system for the University of Western Australia's *Mako* AUV has been completed, and is able to capture and process images, and send navigational commands to the thruster controllers of the AUV. A number of historical computer techniques have also been applied in achieving autonomous pipeline following with the AUV. In particular, the Hough Transform is utilised in pipeline detection, and moments and binary morphology are employed for determining the pipeline location and orientation. An extended binary thresholding technique was also investigated, for use in highly irregular lighting conditions where traditional binary thresholding methods fail to produce a concise binary image. Lastly, performance measurements were taken which highlighted the need for updating the hardware of the AUV.

Acknowledgements

My time spent at the mobile robotics lab will always be a memorable experience. During the last year, I have spent the majority of my time here, undertaking my first major research project. I would like to thank Thomas Bräunl for allowing me to work on such an interesting project and Peter Kovesi, and his unit on computer vision for initially sparking my interests in image processing.

The year was very demanding and I could not have succeeded without the support of my fellow classmates. A big thank you to Elky, Elliot and Pat for keeping me in line during the year and managing to get me back on track whenever i lost the plot; and to Louis Gonzalez, Minh Nyugen and Evan Broadway for their help and guidance during this research project. The thank you also extends to everybody else who has managed to make the mobile robotics lab such a pleasant environment to work in.

To my Friday night friends, I would like to extend my appreciation for bringing me out and providing me with much needed "de-stressing" periods at the pub. Ox, Cookie, Brenno, Bolitho and Andy who all provide great company. A special thank you must also go to my girlfriend Claire, whose love and understanding has helped carry me through the year.

Lastly, a big thank you to my parents and family for their unconditional support throughout my studying life. Their love and care throughout this period has given me the momentum to see through each year.

Contents

List of Figures

List of Tables

Nomenclature

Acronym	Definition
AMI	Axis of minimum inertia
AUV	Autonomous underwater vehicle
CCL	Connected component labeling
CPU	Central processing unit
IPL	Image processing library
LoG	Laplacian of Gaussian
RGB	A colour image consisting of red green and blue pixels

Table 1: Table of acroynms.

Variable	**Definition**
$I(x, y)$	Continuous function that represents an image
$\bar{x}$	x co-ordinate of the centre of mass
$\bar{y}$	y co-ordinate of the centre of mass
ϑ	Angle of the axis of minimum inertia ($rads$)
$M(x, y)$	Continuous function that represents a mask
∇I	The edge gradient of an image
$\|\nabla I\|$	The magnitude of the edge gradient of an image
$\angle\nabla I$	The edge gradient of an image
ρ	Perpendicular distance from a straight line to the origin used in the Hough Transform ($pixels$)
θ	Angle offset from the x-axis in a clockwise direction used in the Hough Transform ($rads$)
$I[x, y]$	Discrete function that represents an image
ϕ	Angle offset from the vertical AUV axis within an image concering the axis of minimum inertia ($rads$)
φ	Angle offset from the vertical AUV axis within an image concerning the centre of mass ($rads$)
s	Displacement from the origin of the AUV axis within an image concerning the centre of mass ($pixels$)
$\beta\varphi$	Tolerance in the offset from the vertical AUV axis within an image concerning the centre of mass ($rads$)
βs	Tolerance in the displacement from the origin of the AUV axis within an image concerning the centre of mass ($pixels$)
$\beta\phi$	Tolerance in the offset from the vertical AUV axis within the image ($rads$)

TABLE 2: Table of variables in order of appearance.

Chapter 1

Introduction

The field of robotics encompasses a wide range of research topics. One of the biggest areas of focus, is the development of robotic vehicles and investigations into their possible applications. Historically, *remotely operated vehicles* or ROV's have been the dominant area of interest with researchers. Although they have been proved to be useful in aiding certain tasks, they still require human intervention for overall guidance of the vehicle. This has lead to a major increase into the research and development of robotic vehicles that adopt autonomous control technologies.

1.1 Autonomous Underwater Vehicles

The most basic definition of an autonomously controlled vehicle is one which does not require any human intervention to make decisions. Instead, the vehicle usually extracts information about its surrounding environment using a variety of sensors, and then uses this information to make navigational decisions. A relatively modern area of research, is the design and development of autonomous vehicles for submersible applications in the marine environment. These are commonly referred to as *Autonomous Underwater Vehicles*, or, AUV's.

The hidden potential of AUV's in marine applications has spawned a major escalation in the number of research projects involved with autonomous control of submersible vehicles. The ocean is the currently largest body of mass on Earth, but still remains the least explored. Hence, scientists have realised the potential of AUV's in furthering the exploration of the deep sea [1]. Other organisations, have focused on the potential of AUV's in surveying the ocean floor for valuable minerals, reducing the cost of exploration for mining organisations. AUV's have also been identified, as being especially useful in carrying out hazardous underwater tasks such as deep-sea object retrieval [2], military countermeasures [2] and detection

of undersea mines.

The list of applications of AUV's is still growing. As autonomous control technology improves, AUV's become more flexible in the possibilities of their application. Hence, there has been a definite trend toward more robust methods of autonomous navigation such as vision guided control.

1.2 Underwater Vision

Autonomously navigated vehicles generally require some way of *sensing* the surrounding environment in order to effectively guide the vehicle. Historically, autonomous navigation has been achieved by using sophisticated sensors with *dead reckoning* navigation. The major pitfall of this however, is that the quality of navigational decisions is highly dependant on accurate sensors, which are typically very expensive. As sensors tend to be constricted in the direction of their measurements, these navigation systems are often poor at reacting to spontaneous changes in the external environment. Hence, there is a need to create autonomous control systems, that are more adaptable to the dynamic marine environment.

At present, vision systems are one of the most commonly used forms of autonomous guidance in the field of robotics. The primary role of a vision system is to extract information from an image and use this information to guide a host system [3]. The popularity of vision based control has gained momentum particularly within the last decade, as computers capable of processing multiple frames a second become relatively inexpensive and the dream of real time image processing is realised. For long range sensing, sonar technology is the most appropriate. On the other hand, vision systems are invaluable in situations requiring accurate measurements at short ranges [4]. The most popular applications of vision systems in both land and marine tasks include obstacle avoidance [4], object recognition, maintenance and inspection [5] and object tracking [6].

Generally, image processing in the natural marine environment is a very complex task. In underwater systems such as the ocean, dynamic lighting conditions and turbulent water can greatly affect the images captured. These anomalies must be addressed and accounted for when information is extracted from the images in order to improve accuracy.

1.3 Project Scope

As discussed in Section 1.1, autonomous underwater vehicles have a range of applications in both commercial and non commercial industries. Of particular interest in this project is the implementation of an AUV vision system that is able to track and follow an underwater pipeline. This application is a very popular area of research for mining organisations, that posses pipelines on the ocean floor. However, pipeline following is a fairly difficult task to achieve in the natural marine environment, due to the frequent presence of noise in a sub surface system. Noise is commonly introduced in underwater images by sporadic marine growth and dynamic lighting conditions as shown in Figure 1.1.

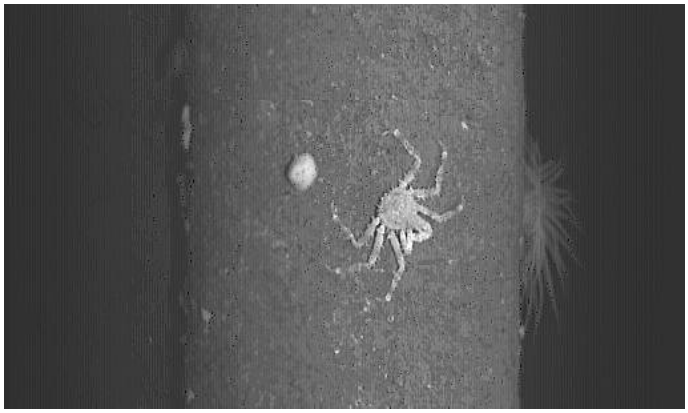


FIGURE 1.1: A commercial underwater pipeline [7].

The main goal of this project, is to create a vision system capable of carrying out visually guided tasks on an AUV. Before this can be achieved, the core tasks such as image capture and processing must first be implemented. The vision system is then programmed to perform underwater pipeline tracking in an artificial test environment, which must address the issues of detection, location and orientation of the underwater pipeline.

1.4 Test Environment

This thesis focuses on the application of the vision system in underwater track following. The majority of testing is performed in indoor environments for the experimental Chapters 5 to 6, where lighting conditions remain fairly static. However, an outdoor environment is used when conducting the experiments demonstrated in Chapter 8 so that binary image analysis can be conducted in situations where dynamic lighting causes problems with standard binary thresholding.

1.4.1 The Underwater Track

An underwater track is the object of interest in all images captured by the vision system. A simple track was created out of standard PVC piping and associated joiner pipes to allow easy construction of multiple track configurations. The properties of the PVC piping allow it to fill with water and sink to the bottom of the test environment. Consequently, the track will lie on the floor of the underwater test environment it is placed in and eliminates the need for anchorage of the pipeline. Throughout the remainder of this thesis, the terms, track and pipeline are used interchangeably, when referring to the underwater track.

1.5 Major Contributions

The major contribution of this project is the design and development of a vision system for the University of Western Australia's autonomous underwater vehicle, named the *Mako*. This entails the compilation of all the vision system hardware and the development of a software library that will form the basis of all vision processing tasks. The major contributions are summarised as follows;

- **Core Vision Software**
  This is comprised of the software structures that standardise how the images are kept in memory and core functions for manipulating images such as;
  - loading and saving,
  - conversion between colour scales and
  - manipulating pixels.

- **Image Processing Library**
  A library of image processing functions has been created that either manipulate images or extract data from them. This is comprised of a number of functions that perform the image processing techniques discussed in Chapter 2, such as;
  - Gaussian filtering,
  - edge detection,
  - connected component labeling and
  - morphological operations.

- **Vision System Hardware**
  The vision system encompasses all the equipment related to image capture, communication and overall interconnection of the vision system with the AUV. This equipment provides several functionalities including;
  - frame capture
  - remote access and
  - image processing.

1.6 Thesis Outline

Firstly, a brief overview of computer vision is presented, focusing on the techniques that are particularly useful to image processing in general and in the application of pipeline following.

In the earlier chapters of this thesis, the vision system is examined in terms of its implemented hardware and software. The system is broken down into a number of modules and the interconnections between these modules are described in detail. This then follows with a description of the software, including the classes created and the overall design rationale.

The experimental chapters deal with the application of the vision system in pipeline following and analyses the detection, location and orientation stages. Each section entails how specific computer vision techniques are used to accomplish a certain goal and the results are shown in sets of example images. Basic performance analysis of the software is then carried out to determine the validity of AUV navigation using real time image processing. Generally, the AUV requires the vision system to capture, process and extract information from images in a timely manner to be considered a feasible control mechanism.

Chapter 2

Background in Computer Vision

The major objective of vision is to extract information from an image or images of the surrounding environment for use in the guidance of a host system [3]. This definition not only applies to natural vision systems but also to artificial vision systems, especially in the field of autonomous robotics. Consequently, much research has been undertaken in the development of techniques that extract information from pictures. Since the dawn of computer vision related research in the late 1950's, a number of useful methods have emerged, that take the implicit information contained within an image and expose this information to the host system in an understandable manner [3]. A number of these techniques are implemented in the vision system software of the *Mako* AUV, for application in underwater pipeline following.

2.1 Binary Image Analysis

Analysis of binary images is one of the most simple and efficient ways to extract information from pictures. It is particularly useful when information needs to be acquired about an object's position and orientation within an image. Equation 2.1 [8] corresponds to the function of a binary image which is represented visually in Figure 2.1.

$$I(x,y) = \begin{cases} 1, & if\ (x,y) \in object \\ 0, & if\ (x,y) \notin object \end{cases} \tag{2.1}$$

2.1.1 Thresholding

Binary thresholding is one of the most common ways to extract an object of interest from an *rgb* or *grayscale* image. The thresholding process generally involves com-

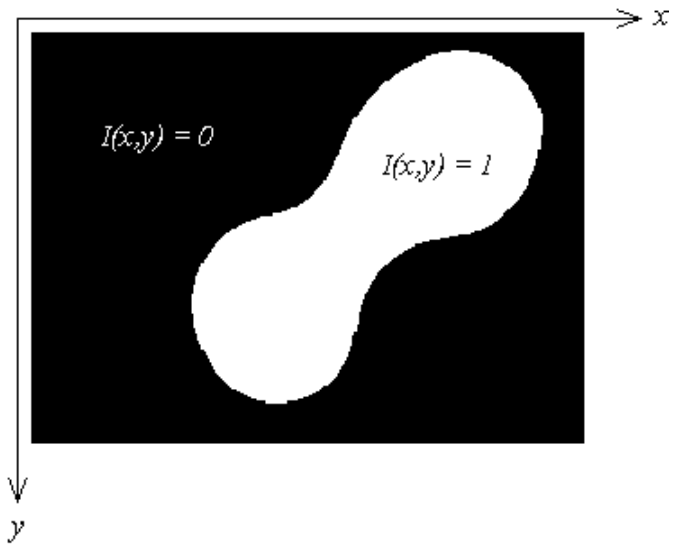


FIGURE 2.1: An example of a binary image.

paring a pixel against a certain value or values and turning *on* a pixel in the output image if it satisfies the criteria or turning *off* a pixel in the output image if it violates the criteria. The most common form of thresholding is simple grayscale thresholding where a single threshold value is specified. If a pixel intensity is greater than this threshold, then the outputted pixel is assigned a value of 1, else the outputted pixel is assigned a value of 0. Thus the binary image produced, consists entirely of pixels with intensity values of either a 1 or 0. This is a very simple and efficient method for extracting objects from an image. However, the degree of accuracy of this process is highly dependant upon the characteristics of the input image. Binary thresholding works best on images where the foreground pixels (object) are clearly distinguishable from the background pixels.

2.1.2 Moments

Binary images often contain implicit information about the objects held within termed *moments*. There are many techniques available for extracting this information, that can reveal data about the object such as its position, orientation and size. Of particular significance to the pipeline following application are the first and second moments of a binary image or the *centre of mass* and the *axis of minimum inertia* respectively.

2.1.2.1 Centre of Mass (Centroid)

It is often desirable to obtain positional information about an object in a binary image. This can be achieved by calculating the *centre of mass* or *centroid* of the object within a picture. The centre of mass is useful as it can give a general indication of the location of the object in relation to an origin, located in the top left corner of an image as shown in Figure 2.1. Equations 2.2 and 2.3 [8] give the $\bar{x}$ and $\bar{y}$ co-ordinates of the centroid location respectively, where $I(x, y)$ is the image function specifying the value of a pixel at a particular x, y location.

$$\bar{x} = \frac{\int\int x^2 I(x,y)dxdy}{\int\int I(x,y)dxdy} \tag{2.2}$$

$$\bar{y} = \frac{\int\int y^2 I(x,y)dxdy}{\int\int I(x,y)dxdy} \tag{2.3}$$

2.1.2.2 Axis of Minimum Inertia

Another useful calculation which can be performed, is to determine the *axis of minimum inertia* or *AMI* of an object. This property is used to represent the orientation of an object within a binary image. Equations 2.4 to 2.9 [8] are used to calculate the angle of the axis of minimum inertia (ϑ), where $\tilde{x} = x - \bar{x}$ and $\tilde{y} = y - \bar{y}$.

$$a = \int\int \tilde{x}^2 b(x,y)dxdy \tag{2.4}$$

$$b = \int\int \tilde{x}\tilde{y} b(x,y)dxdy \tag{2.5}$$

$$c = \int\int \tilde{y}^2 b(x,y)dxdy \tag{2.6}$$

$$sin\,(2\vartheta) = \frac{b^2}{\sqrt{b^2 - (a-c)^2}} \tag{2.7}$$

$$cos\,(2\vartheta) = \frac{b^2}{\sqrt{b^2 - (a-c)^2}} \tag{2.8}$$

$$\vartheta = arctan2\left(\frac{sin2\vartheta}{cos2\vartheta}\right) \tag{2.9}$$

2.2 Morphology

The binary thresholding process often produces an image that is less than perfect. Common problems associated with binary images are random speckles produced by noise, and gaps in an object usually created by inconsistent lighting. It is often desirable to process a binary image before analysis to remove these abnormalities. This is accomplished by applying *morphological operations* on the raw binary picture. After doing so, calculations of properties such as the *centroid* or *axis of minimum inertia* become more accurate, as the definition of the object within a binary image is improved. For example, speckles that are not evenly distributed around an object of interest will tend to pull the centre of mass location toward the direction of the noise, thereby degrading the accuracy of the calculation.

2.2.1 Structuring Element

Morphological operations require the specification of an image processing element termed a *structuring element* [8]. The properties of the structuring element determine the extent to which a binary image is enhanced. A structuring element can be specified to be of any size or shape and is only limited by the size of the image and the processing power of the platform the algorithms are performed on. The essential properties that must be specified are given below.

- **Shape**
  A structuring element can be given any size or shape, but the choice is usually governed by the properties of the object of interest.

- **Origin**
  The origin of a structuring element is the pixel that is placed on the object pixels of the binary image.

These properties must be defined in order to perform the actual morphological operations. The two most common types are called *erosion* and *dilation*. These two operations have been implemented in the vision system, and allow a structuring element of any size and shape to be specified. Figure 2.2 shows an example of a structuring element and the location of its origin, and Figure 2.3 shows an example image, upon which this structuring element will be applied in the subsequent descriptions of erosion and dilation.

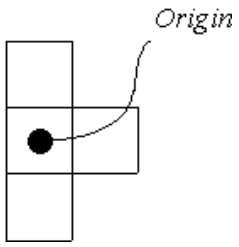


FIGURE 2.2: An example of a structuring element.

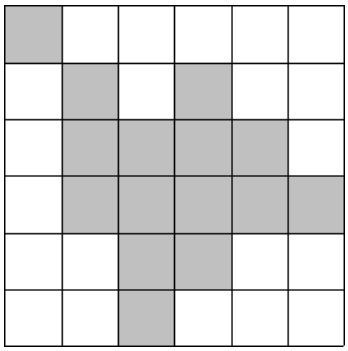


FIGURE 2.3: An example of a digital image.

2.2.2 Erosion

The process of erosion is normally used to remove speckles or salt and pepper noise from a binary image. It involves placing the origin of the structuring element over all pixels of the object within the binary image. The binary output image is produced by turning *on* a pixel in the output image, if every pixel of the structuring elements falls on an object pixel in the original image. This process is demonstrated in Figure 2.4 using the structuring element shown in Figure 2.2 and the example image in Figure 2.3.

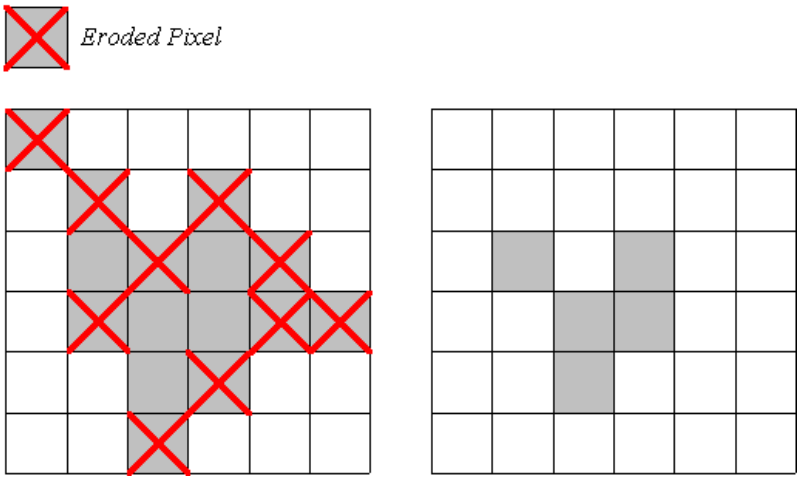


FIGURE 2.4: A demonstration of an erosion operation.

2.2.3 Dilation

The process of dilation is normally used to fill in gaps in an object in binary picture. Similar to erosion, the process involves placing the origin of the structuring element over all pixels of the object in the original image. However, dilation produces the output image differently from an erosion operation. The binary output image is produced by turning pixels *on* in all locations of the structuring element, when the origin of the structuring element is placed over all object pixels in the original picture. This process is demonstrated in Figure 2.5 using the structuring element shown in Figure 2.2 and the example image in Figure 2.3.

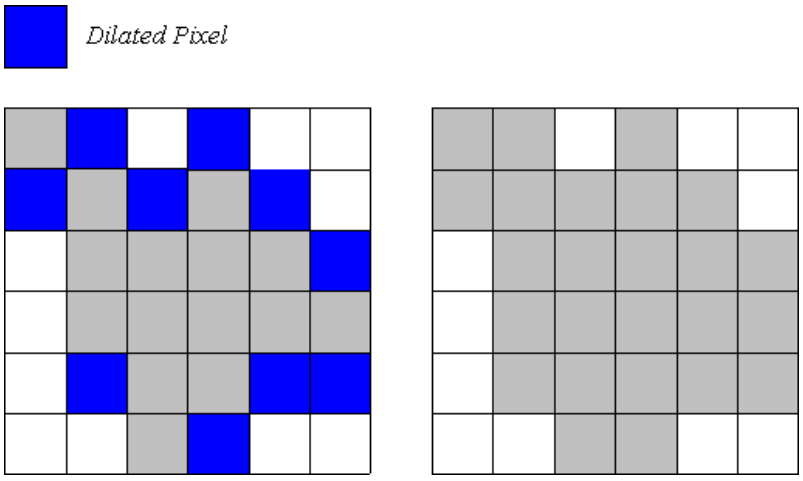


FIGURE 2.5: A demonstration of a dilation operation.

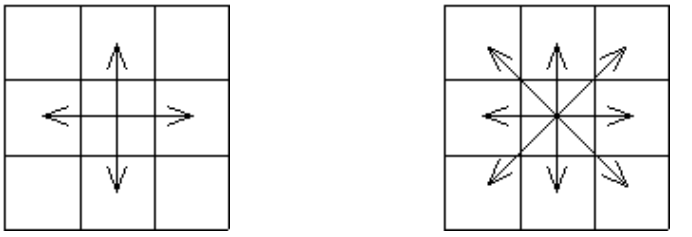
2.2.4 Opening and Closing

The erosion and dilation operations can be combined into one step to perform operations known as *opening* and *closing*. Opening of a binary image involves an erosion operation followed by a dilation operation using the same structuring elements. This will tend to remove speckles from a binary image but will counter the effects of the eroding the pixels of the object. Closing of a binary image involves a dilation operation followed by an erosion operation. Generally, closing tends to fill in thin gaps in an object but will counter the effects of the object growing in size from the dilation operation. The opening and closing operations are generally only performed once with the same structuring element, as any repetitions of these operations result in the same binary output image and are a waste of computation time [9].

2.3 Connected Component Labeling

Upon the attainment of a binary image, it is often desirable to somehow segregate individual objects within the image. This process of identifying and giving each object a unique label is most commonly known as *connected component labeling* (CCL) [10]. Generally, the degree of connection between pixels differs according to the implementation. The type of connectivity of the labeling algorithm is referred to as the *n-connectedness*. The most common and simple object connectivities used are 4-connectedness and 8-connectedness which are shown in Figures 2.6(a) and 2.6(b) respectively. However, researchers have found these types of object connectivity to be either too conservative or too excessive as they violate the *Jordan Curve Theorem* [8].

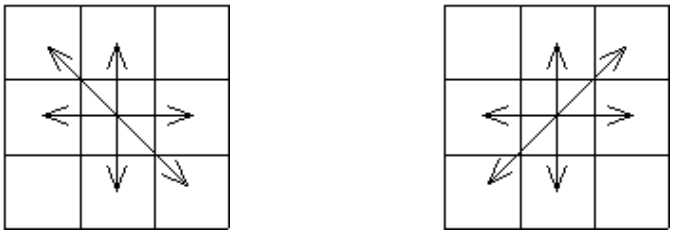
> The Jordan Curve Theorem states that; *A simple closed curve separates the plane into two simply connected components.* [8]



(a) 4-connectedness (b) 8-connectedness

FIGURE 2.6: Various object connectivity types.

The ideal connectivity to be used would be a hexagonal grid with 6-connectedness [8]. Due to the location of image pixels this is impossible. However, often a compromise is made and 6-connectedness is achieved by skewing the image pixels. This is demonstrated in figures 2.7(a) and 2.7(b) for different skewed schemes.



(a) Left-skewed (b) Right-skewed

FIGURE 2.7: 6-connectedness schemes.

The most common algorithm used to label components is the sequential scan. This basically involves iterating over the image from column to column, row to row, and assigning each pixel a label. Each time, the neighbouring pixels are checked for a label according to the connectedness scheme used. In some cases a problem may arise where there are neighbouring pixels with differing labels assigned to them. In this case a particular pixel can assign any one of those labels to itself, but must record the equivalence of this label to the alternative. This gives rise to a structure called an *equivalence table* that records any equivalent labels encountered in the sequential scan. It is then necessary to parse through this equivalence table and create unique *equivalence classes*. All pixels in the labeled image are then re-assigned a label according to the equivalence class to which their current label belongs.

2.4 Filtering

Due to the dynamic nature of natural lighting in the marine environment, images are often corrupted by noise. Noise introduces irregular intensity values of pixels in random locations. It is usually desirable to reduce the effects of these pixel anomalies when inspecting images visually and highly desirable when processing images in an automated environment. This is most often done by convolving an image with a structure commonly referred to as a *mask*, refer to Appendix B. There is an infinite number of possibilities when it comes to designing filter masks. However, a common property of these filters is that the masks are normalised such that the sum of the weights on every pixel of the mask add up to one. This is done to avoid scaling the brightness of the image. It is also common for masks to be of an odd square size so that there is a distinct, symmetric, centre pixel to assign the new intensity value calculated.

2.4.1 Averaging Filter

Convolving an image with an averaging filter is one way to reduce the occurrence of noisy pixels in an image. The basic principle used in averaging filters is to make a particular pixel intensity similar to the pixel intensities of its neighbours. The averaging filter gives equal weighting to each of the pixels considered to be in the neighbourhood of the pixel in focus. Figure 2.8 shows a typical 3×3 averaging filter.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

FIGURE 2.8: A typical 3×3 averaging filter.

2.4.2 Median Filter

Averaging filters tend to blur the boundaries between the objects in an image and the background. The median filter is useful for removing outlier pixel intensities whilst preserving edges in an image. Instead of using a mask consisting of with weights, the median filter takes the median of all neighbouring intensity values and assigns this value to the pixel in focus.

2.4.3 Gaussian Filter

The *Gaussian* filter is probably the most common type of filter, and is often used as a preprocessing step to the edge detection process. The Gaussian filter consists of a mask made up of weighted pixel values. The weights are calculated according to the equation of a Gaussian distribution shown in Equation 2.10, where $M(x, y)$ represents the function of the Mask. However, the equation is normally reduced to Equation 2.11 by removing the scaling factor of $\frac{1}{\sqrt{2\pi\sigma^2}}$, as the weights of the pixels must be scaled to add up to one anyway. Figure 2.9 shows the coordinate system of the mask used in calculation of the weighted pixel values. In a Gaussian filter, the centre pixels are given the most weighting with decreasing weight given to neighbouring pixels of increasing distance, such as the example shown in Figure 2.10.

$$M(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.10}$$

$$M(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.11}$$

2.5 Edge Detection

Edges within an image usually represent the boundaries between objects and the background. When viewing a scene, a great deal of information can be conveyed

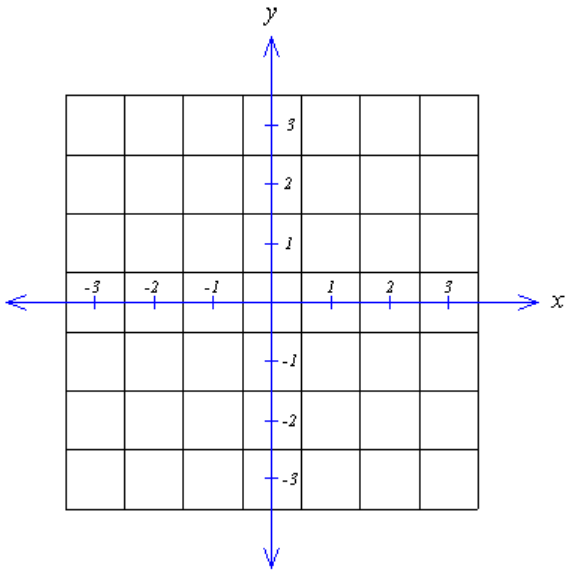


FIGURE 2.9: Coordinate System of a 7×7 Gaussian mask.

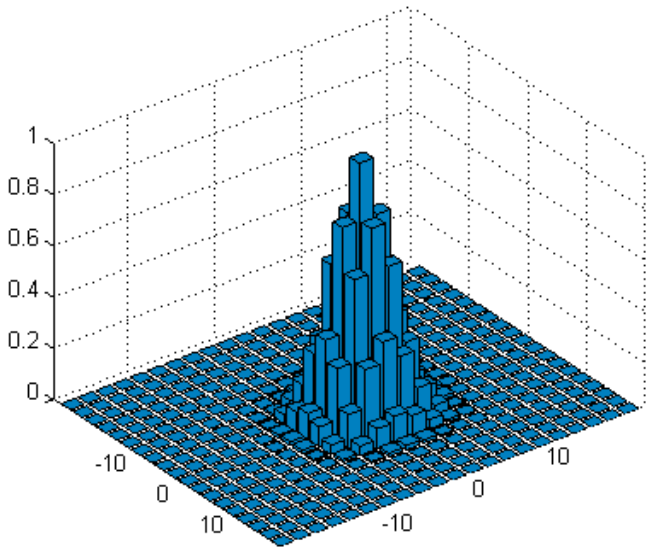


FIGURE 2.10: A 21×21 Gaussian Filter, ($\sigma = 1.4$).

to the observer through a simple line drawing [8]. Hence, it is evident that edges contain ample information of the particular image and the process of edge detection is an extremely important part of computer vision.

To the natural eye, the process of detecting edges in a picture is a fairly simple process. In computer vision however, edge detection is still a cumbersome and complex process. In image processing, edges are defined as step differences in the intensity of adjacent pixels. In computer terms, a digital image consists of an array of pixel intensity values. In a colour or *rgb* image, this array consists of pixel intensity values for each of the red, green and blue colour channels, ranging from 0 to 255. In a *grayscale* image, this array consists only of the pixel intensity values for the white channel. For grayscale images, edges occur at points in the picture where neighbouring pixels have the largest differences in their intensity values. Therefore, the general purpose of an edge detection algorithm is to locate the regions where peaks of these pixel intensity differences occur.

2.5.1 Finding Edges

An edge has been defined as any region within an image where the values of intensity of neighbouring pixels differ greatly. This intuitively presents two alternative methods of finding edges. Suppose the derivative is taken of the image, i.e. the difference in intensity values between a pixel and its neighbours is calculated for all pixels of the image. Finding edges then involves searching for the locations of local peaks within this derivative image. The second method involves taking the second derivative of the original image. Finding edges now involves searching for the locations of *zero-crossings* or where the second derivative values cross any of the horizontal axes. The most common forms of first derivative based edge detection involve taking the gradients of the image in the x and y directions. This can be represented by Equation 2.12 [8], where ∇I represents the gradient of the image function $I(x, y)$.

$$\nabla I = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right) \tag{2.12}$$

The edge magnitude and direction of a particular pixel can then be calculated using Equations 2.13 and 2.14 [8]respectively.

$$|\nabla I| = \sqrt{\left(\frac{\partial I}{\partial x} \right)^2 + \left(\frac{\partial I}{\partial y} \right)^2} \tag{2.13}$$

$$\angle \nabla I = \arctan \frac{\left(\frac{\partial I}{\partial x}\right)}{\left(\frac{\partial I}{\partial y}\right)} \tag{2.14}$$

2.5.2 Edge Detection via Classical Methods

Historically, a large part of computer vision research has been dedicated to the investigation of edge detection schemes. Of the many schemes available, the more commonly known edge detectors are the *Canny* and *Marr-Hildreth* operators [11]. The two schemes differ mainly in their implementations of finding an edge.

2.5.2.1 Marr-Hildreth Operator

The *Marr-Hildreth* operator transpired from research undertaken by Marr and Hildreth [11]. Their research involved the use of the *Laplacian of Gaussian* (LoG) to detect edges within an image. It is important to note that the filtering of the image is done in the frequency domain instead of the spatial domain and is based upon finding zero-crossings in the second derivative of the image. Once the image has been convolved with the LoG filter, a search for zero-crossings is performed on the resultant output to find edges. The most common way of doing so, is to threshold the resultant output using a threshold value of zero. The output of this process will be a binary image, where the pixels that are turned *on* are those that have value less than zero. An operation must then be performed to determine the boundary locations between pixels that are turned *on* and those that are *off*.

2.5.2.2 Canny Operator

The Canny Operator is a product of research aimed at designing the most optimal edge detector. The process used here differs from the Marr-Hildreth operator in that the first derivative of a Gaussian filter is used to detect edges within an image. The edge detection process of the Canny operator consists of a number of steps [11];

1. convolution with a Gaussian filter,
2. finding the edge derivatives in orthogonal directions,
3. calculating the edge directions and magnitudes and
4. performing non-maximal suppression.

The major accomplishments of the Canny operator was the introduction of *thresholding hysteresis*. This allowed two threshold values to be specified when performing non-maximal suppression on edges. The magnitude of an edge is first compared with the higher threshold to determine it's validity as an edge. If the edge magnitude is greater than the upper threshold, then the pixel is classified as a *definite edge*. Otherwise, it is classified as a *possible edge* and checked against the lower threshold. If the edge magnitude exceeds the lower threshold and is connected to a neighbouring pixel that has been classified as a *definite edge*, then the pixel is now classified as being a *definite edge*, else the pixel is classified as being a *non edge*. This type of thresholding usually produces edge maps that contain less discontinuities in the edges.

2.5.3 Edge Detection via Neural Networks

The advent of *Neural Networks* has lead to many applications in regards to robot control. The technology can also be applied to the edge detection detection problem in the computer vision field. This form of edge detection can potentially offer greater performance and results over conventional edge detection algorithms. It also removes the need to reproduce a suitable edge detection algorithm in code. Although Neural Networks have the potential to produce better results and simplify the edge detection process, it is extremely difficult to compose a proper training set to the Neural Network to achieve these results.

Generally, the Neural Network is trained using a method called *back-propagation* [12]. This involves creating a set of training patterns and calculating the weights between each of the nodes in the Neural Network. In this particular case, the set of training outputs are likely to be binary edge maps and the inputs to the Neural Network would be the derivative images that the binary edge maps are obtained from. To input a two dimensional image into a Neural Network first requires the image to be *flattened*. This involves the creation of a one dimensional vector from the two dimensional image matrix. This is usually achieved by simply appending consecutive rows of an image together to form a one dimensional vector.

2.6 The Hough Transform

Feature extraction is commonly used for the detection of objects in a picture. One of the most easily and commonly extracted features of an image are straight lines. Visually, this is a relatively easy task for the human eye but is a fairly complex

process to perform in software. This process is known as the *Hough Transform* and is in widespread use for the extraction of straight lines from images. The route taken by the Hough Transform is to collect votes for every possible line equation that passes through an edge point in an image. The number of line equations to be tested is highly dependent on the degree of precision required.

In order to test a finite set of straight lines that gives a fair weighting to all possibilities, it is necessary to first derive a suitable line equation. The most obvious equation to use would be the simple linear equation $y = mx+b$. However, considering that $-\infty < m < \infty$ and $-\infty < b < \infty$, it would be very difficult to test a set lines that give fair weighting to all possibilities [13].

In the Hough Transform, straight lines are represented by Equation 2.15 [14]. Representing straight lines in this form, allows better parameterisation of the line equation, compared to the traditional linear equation [14]. In Equation 2.15, ρ and θ correspond to the perpendicular distance to the origin from the straight line and the angle offset from the x-axis respectively. It is important to note that in computer images, the origin is located at the top left hand corner of the image.

$$\rho = x \sin\theta + y \cos\theta \tag{2.15}$$

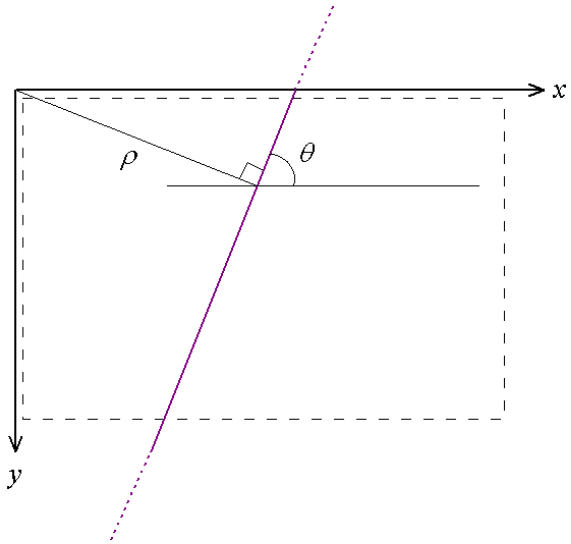


FIGURE 2.11: Hough line representation in a digital image.

Figure 2.11 demonstrates the depiction of Hough lines in an image. The diagram shows the co-ordinate axis of the image and also gives visual meaning to the ρ and θ parameters.

The ρ and θ parameter spaces are then quantised into a suitable number of intervals to produce an accumulator array [14]. Each cell in the array corresponds

to a specific line equation, and the cell is used to accumulate votes for that particular line. Therefore, each accumulator cell is incremented for every edge point that lies on the corresponding line equation. The result of this process is a two dimensional histogram from which the most significant straight lines can be extracted. Figure 2.12 shows a typical Hough Transform, which is a graphical representation of the accumulator array.

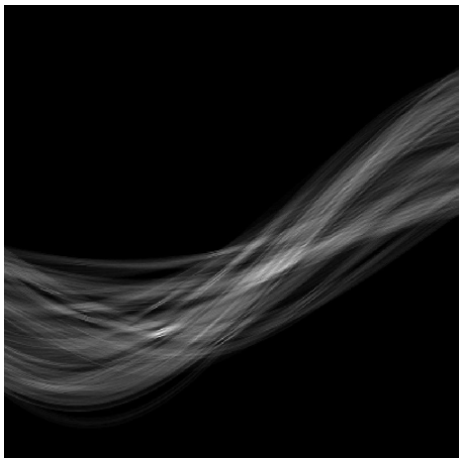


FIGURE 2.12: A visual representation of an accumulator array.

Chapter 3

Vision System

The vision system of the autonomous underwater vehicle is comprised of a number of hardware and software modules. The development of the vision system involved linking these modules through hardware and software to create a vision system capable of carrying out visually guided assignments.

3.1 System Goals

The vision based tasks that the AUV must be able to accomplish will vary depending on the application. Thus, the vision system must be easily adaptable to incorporate new functionality for the ever changing task requirements. To obtain the desired functionality, there should be minimal coupling between each module outlined in Figure 3.1. I.e. modules should be able to be modified without requiring dramatic alteration to any of the other modules. The following properties were a major influence on the design of the AUV's vision system.

- The vision system should be created out of relatively inexpensive hardware, as the role of an engineer generally entails building efficient, cost-effective solutions.
- The design of the vision system should be modular such that there is minimal coupling between the modules. This is a necessary requirement due to ever changing functional requirements. The AUV will often be undergoing hardware upgrades, add-ons and software updates. Thus, it is highly desirable to allow a module to be altered without affecting any existing modules.
- The vision system software should be re-usable and re-programmable as the visually guided tasks will often change. Image processing algorithms should be able to be reused where possible to reduce programming time.

3.2 System Overview

The vision system is comprised of a number of hardware components used for both processing and communications. Figure 3.1 shows the major modules of the vision system in terms of hardware modules and their interconnections. Figure 3.2 shows the major software modules.

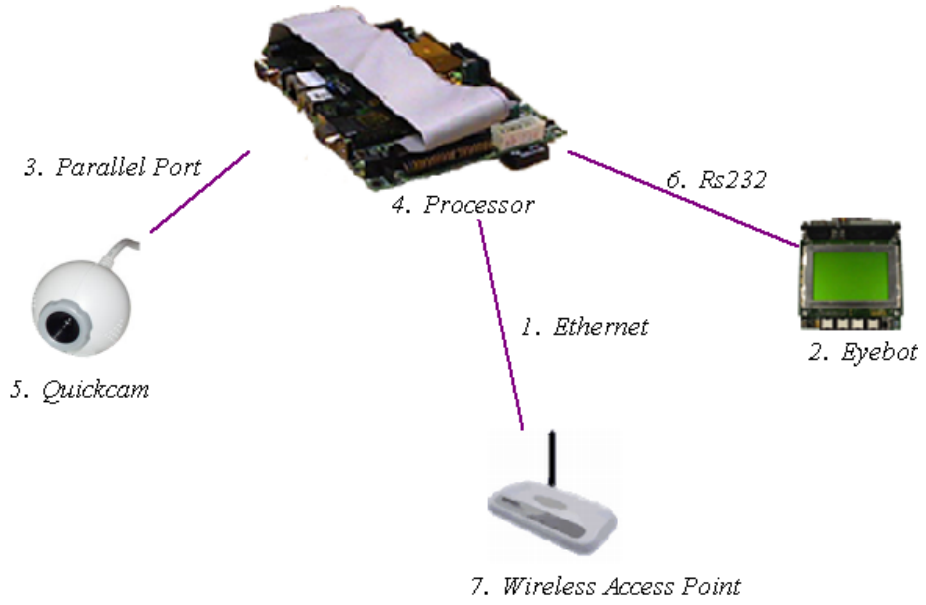


FIGURE 3.1: An overview of the hardware components.

For more detailed specifications of any hardware or software described, refer to Appendix A.

3.3 Processor

All image processing is performed on a commercially available microprocessor, (refer to Appendix A for specifications). Currently, this processor is completely dedicated to image processing as this is generally a computationally intensive procedure. For example, a simple convolution operation requires a computation to be performed on every pixel in the image, whereas other functions may require several iterations over each image pixel. Hence, there is a definite need for a dedicated microprocessor for image processing. The use of a commercially available processor not only offers performance advantages but also provides flexibility when introducing hardware additions.

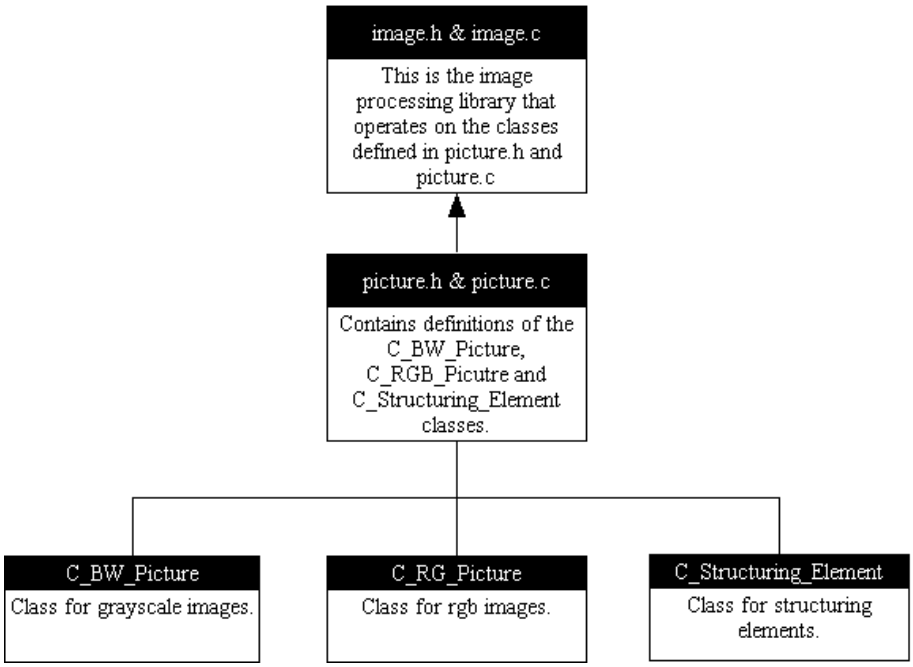


FIGURE 3.2: An overview of the software components.

3.4 Camera

The vision capture module encompasses the frame capture device and the software that interfaces the device to the microprocessor. A standard, colour web-camera is used as the frame capture hardware and is interfaced to the operating system through a software package, that reads data bytes directly from the parallel port and places the image in memory.

3.4.1 Equipment

A simple parallel port *quickcam* or web camera provides the frame capture ability for the vision system. The use of a web camera allows software based frame grabbing which is relatively cheaper than harder based frame grabbers at the sacrifice of speed. A parallel port camera was chosen over a faster USB camera due to lack of reliable USB camera drivers for the Linux operating system installed on the microprocessor. Although the older parallel port cameras are slower than their USB counterparts, there are some advantages to be gained from their use. Firstly, there are a number of free, open-source drivers and software available for parallel port cameras, whilst most of the available USB drivers are still in their experimental stages. Secondly, a USB camera would require installation of the *Video4Linux* [15] package and recompilation of the operating system on the microprocessor. Due to

the limited processing power of the microprocessor, smaller images are captured by the camera so the speed of the parallel port device is less of an issue.

3.4.2 Drivers

The *quickcam* is interfaced to processor via dedicated software rather than generic drivers. The freely available and open source *cqcam* [16] software bundle is used to capture frames from the parallel port camera. The relevant code fragments relating to the actual capture of frames has been extracted from this software bundle and used in the vision system. The software is capable of capturing images at a maximum size of 320×240 pixels. However, in this implementation, images are captured at a resolution of 160×120 pixels to reduce the processing time of an image and allow faster frame capture.

3.5 Module Interconnection

The ultimate goal of the vision system is to make navigational decisions, and communicate these to the thruster controllers. To achieve this, requires the interconnection of the image processor to the Eyebot micro-controller. A wireless connection to a remote computer is also desirable to aid in data acquisition and remote control.

3.5.1 Eyebot Communication

The actual AUV electromechanical system is controlled by the Eyebot micro-controller. The Eyebot is currently programmed to receive navigational commands via rs232 serial port communication and can be controlled remotely via a Bluetooth module connected to the serial port. The vision microprocessor is connected to the second serial port of the Eyebot. This is used as the communications medium for the vision system to send navigational commands to control the AUV. Modifications have been made to *adrport* source code freely available from [17] to allow serial communications in the vision system software.

3.5.2 Wireless Access

It is highly desirable to implement wireless communication with the vision system, to allow for gathering of important experimental data and allow remote compilation and execution of programs. Currently, wireless communication with the AUV is only needed when the AUV has emerged from the water. Hence, there are two obvious

alternatives to provide wireless communication, namely Bluetooth technology or wireless Ethernet. The choice of communications technology was primarily governed by;

- hardware cost and
- data rates.

In terms of actual cost, both technologies are relatively similar in pricing. However when comparing the cost against the data rates provided by each technology, wireless ethernet has a clear advantage. Bluetooth technology does however, have an advantage, in terms of the range of communication. Unfortunately, this form of Bluetooth technology comes at a higher cost than a standard USB port Bluetooth module that has a similar communication range to wireless ethernet. The remote communication system used is not appropriate for commercial or industrial usage where communication with the AUV must be enabled over longer ranges and whilst the AUV is submerged. It is however, suitable for use in small testing environment for wireless surface communication.

Chapter 4

Vision System Software Design

The image processing software forms the major part of the overall vision system. Coinciding with the goals stated in Chapter 3, Section 3.1, the software is designed with re-usability and re-programmability in mind. Due to the degree of precision required when using vision based navigation, the general aim is to reduce the time spent coding, and increase the time spent determining the ideal parameter values of the various processing functions. For detailed descriptions of the image processing library, see Appendix D.

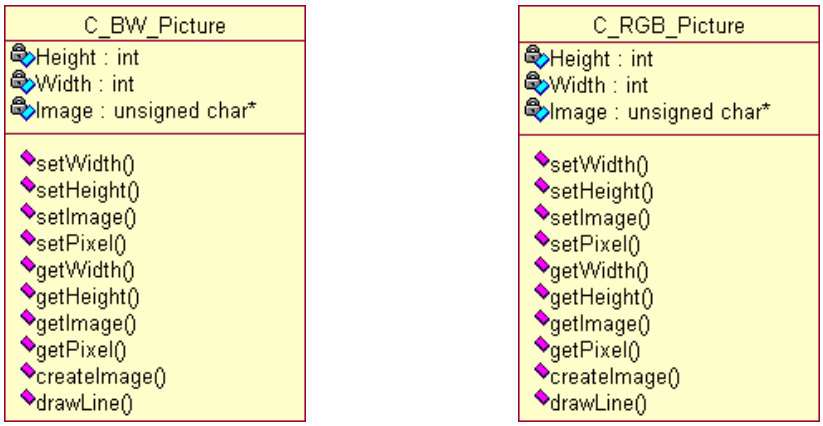
4.1 Structures

A number of classes were created to standardise any inputs that are processed by the image processing library. The creation of these classes is to decouple the vision processing library from any reliance upon the way specific camera hardware stores images in memory. Thus, the software is independant of the camera hardware and allows the camera hardware to be updated without causing any problems to the vision system software.

4.1.1 Image

The *C_BW_Picture* and *C_RGB_Picture* classes (refer to Appendix D), represent grayscale and rgb, colour images respectively. The definition of these structures allows the image processing library to remain independant of the vision capture process. Functions in these classes, allows access to the pixels of an image similar to a matrix, using the *setPixel()* and *getPixel()* functions. A *drawLine()* function has also been implemented under each of these classes, as it is often needed for experimental purposes. Class diagrams of these structures are given in Figures

4.1(a) and 4.1(b).



(a) Grayscale images. (b) RGB images.

FIGURE 4.1: UML Diagrams: Image classes.

4.1.2 Structuring Element

The *C_Structuring_Element* class (refer to Appendix D), is used to create a structuring element of any size and shape. This is important in the use of morphological operations on images, where a structuring element often needs to be tailored to a specific application. The structuring element class is similar to that of a binary mask and when creating the structuring element, its rectangular size and origin location is specified as in Figure 2.2. Pixels of the mask are then either turned *on* or *off*, similar to a binary image, to define the shape of the structuring element. The structuring element may also be input into a connected component labeling function to specify the connectivity used in the labeling algorithm during run time. The class diagram of this structure is given in Figure 4.2.

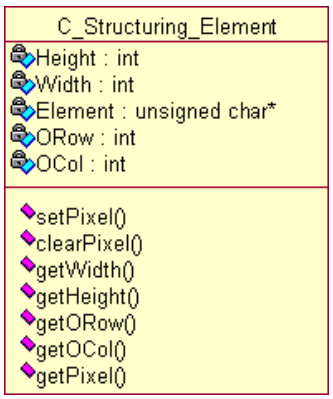


FIGURE 4.2: UML Diagram: Structuring element class.

4.2 Image Processing Library

The image processing library, IPL, predominantly contains implementations of various computer vision algorithms. Figure 4.3 provides an overview of the vision capture and processing flow. Upon capturing an image, a custom wrapper must first be written that places the captured image into a C_BW_Picture or C_RGB_Picture object. The IPL is then able to perform image processing on these objects and generate navigational decisions to be sent to the thruster controllers.

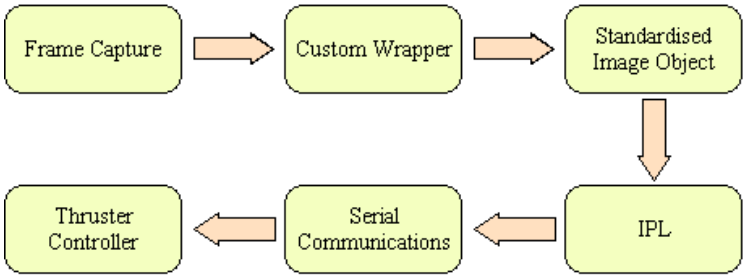


FIGURE 4.3: Image capture and processing flow.

4.2.1 Software Language

The IPL is written purely in the standard *c* programming language, to allow the library to be used on any processing platform that can compile *c* programs. Hence, there are no operating system specific calls made in the library, allowing it to be compiled on virtually any computer. It is hoped that the cross platform compatibility of the software, will see it used extensively in a number of robotic vision applications other than the AUV vision system, such as image based door detection in the *autonomous wheelchair project* at the University of Western Australia [18].

4.2.2 Design Influence

The vision system software is essentially a *toolbox* for image processing. All functions are designed to support code re-use such that they can used in *pre* or *post* processing in other image functions. Consequently, all functions allow the design time specification of the parameters that govern the extent of the image processing performed. For example, the implemented erosion operation, takes as its inputs an image (*C_BW_Picture* or *C_RGB_Picture*) and a structuring element (C_*Structuring_Element*) of any size or shape and outputs the processed image.

Chapter 5

Track Detection

Artificial objects normally have significant features that can distinguish them from the natural scenario, even in an image corrupted with noise [5]. Hence, vision systems are often used for robust detection and tracking of objects. There are a variety of techniques available to perform object detection. Most commonly, this is performed by taking a feature of an object that is invariant to rotation, scaling and translation [19] and matching these quantities against a set of known patterns to positively identify an object. In track following, the vision system must be able to recognise an artificial pipeline structure within an image. This will be helpful in scenarios where an AUV must initially search for the track, or has wandered off from the track and must detect it to re-commence following. In industrial applications such as underwater pipeline and cable following, this functionality is extremely significant, as the vision system may often lose sight of the pipeline or cable due to sediments on the ocean floor and the occurrence of natural marine growth.

5.1 Application of the Hough Transform

The *Hough Transform*, described in Chapter 2, Section 2.6, is a useful algorithm for finding shapes or lines in a computer image. Generally, the technique can be used to find any geometric shape that can be represented by a set of parameters which have a finite set of values [13]. In the application of underwater track following, the Hough Transform is used to extract the most significant straight lines from a computer image and use the properties of these lines to recognise the underwater track.

There are a number of reasonable assumptions that enable the Hough Transform to be used as a suitable detection method for an underwater pipeline.

1. "The roll and pitch of the AUV is passively controlled and hence considered to be negligible [20]",
2. The vision capture device is aimed directly down along on the z-axis,
3. The underwater track lies on a horizontal plane and
4. The underwater track is composed of straight line segments.

Given these assumptions, a feature of an underwater track that is invariant to rotation and scaling can be defined;

> *The underwater track is detected within an image, if there exists at least one set of semi-parallel lines.*

The process of line extraction involves the use of a number of classical computer vision techniques. Parameter specifications are required at each step, to determine the extent of processing done at each stage of the line extraction process. However, the optimum values of these parameters are highly dependant on the current conditions of the external environment. Therefore, there is no unique set of parameter values that satisfies every scenario. An aim of this chapter, is to investigate the affects of these parameters on the overall line extraction process. Table 5.1 shows the parameters required by each step of the line extraction process.

Step	*Parameters*
Gaussian Filtering	(σ) Standard Deviation, ($N \times N$) Filter Size
Edge Detection	(T_U) Upper Threshold, (T_L) Lower Threshold
Hough Transform	(ρ) Intervals, (θ) Intervals
Line Extraction	(T_A) Accumulator Threshold

TABLE 5.1: Steps in the Line Extraction Process

5.2 Gaussian Filter Parameters

Convolution of an image with a Gaussian filter before performing edge detection is a very important preprocessing step. The aim of applying this sort of filtering is to remove noisy pixels from an input image. This can greatly improve the quality of the edge map obtained after edge detection, as edges are no longer produced around the noisy pixel regions.

The parameters that control the degree of smoothing of a Gaussian filter on an input image are namely the standard deviation (σ) and size of the filter ($N \times N$).

The ideal values of these parameters are highly dependant on the degree of noise in an image and also on the current lighting conditions. Consequently, there are no universal values for these parameters that will always give the desired amount of smoothing. There are possibly ways to automatically determine the optimum parameter values, however, this is outside the scope of this project.

Figures 5.2 and 5.3 demonstrate the affects of standard deviation and filter sizes on the resultant edge strengths of the image, as shown in Figure 5.1. Generally, increases in standard deviation lead to a higher degree of blurring or thicker edges, due to more influence of neighbouring pixels. The same can be said for increasing the size of a filter, although the resultant blurring is not as severe as introduced by increasing the standard deviation.

FIGURE 5.1: A noisy test image of the underwater pipeline.

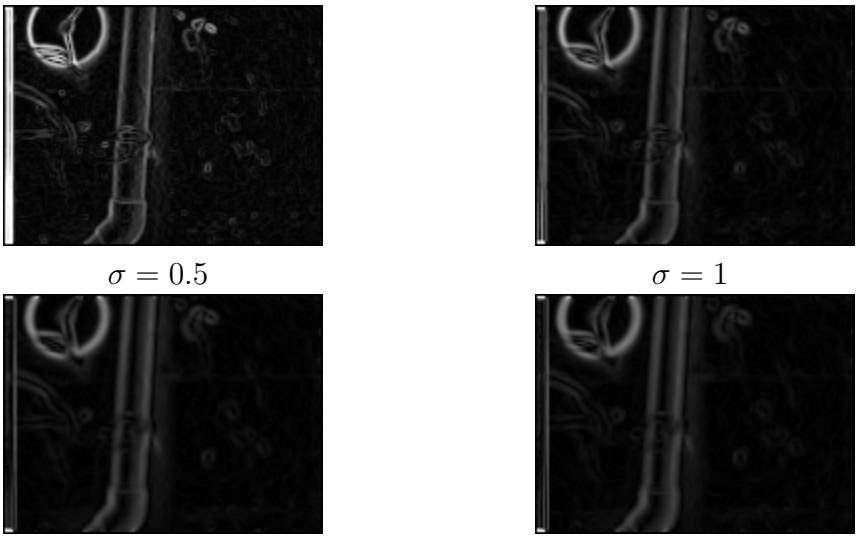


FIGURE 5.2: Effects of standard deviation (σ) on edge strengths using 5×5 filter.

The extent of Gaussian filtering has a direct effect on the performance of edge detection. The raw gradient images in Figures 5.2 and 5.3 show that as blurring increases, thicker edges and lower magnitudes tend to arise. This results in less

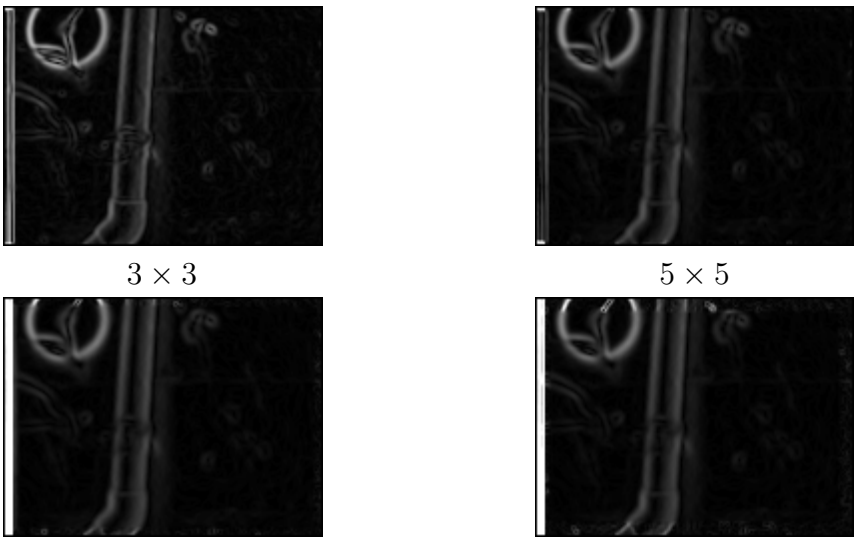


FIGURE 5.3: Effects of filter size ($N \times N$) on edge strengths with $\sigma = 1$.

edges being detected and a less defined edge map being produced. This is expected, as the filtering process will reduce the intensity differences between adjacent pixels, thus reducing the likelihood of finding edges according to their definition in Chapter 2, Section 2.5. A 5×5 Gaussian filter of standard deviation $\sigma = 0.5$, is appropriate for removing most noisy pixels from the image whilst preserving edges. Any filters with $\sigma > 1.5$, generally results in edges being lost, due to increases in edge thickness, but decreases in edge strengths.

5.3 Edge Detection Implementation

The edge detection process used is a variation of the Canny Operator described in Chapter 2, Section 2.5.2. The algorithm implemented is similar to the tutorial on Canny edge detection presented in [21]. The edge detection algorithm approximates the Canny Operator by convolving the input image with the first derivative of a Gaussian filter. This is a two step process which first involves convolution of the image with a Gaussian filter, and then taking the difference between adjacent pixel intensities. The second step is achieved by convolving the image with *Sobel Masks* [22], shown in Figure 5.4.

Upon completion of this step, Equations 2.13 and 2.14 are used to calculate the magnitude and direction of the edge point. The direction of each edge point is then quantised into a minimal set of directions [21], coinciding with the discrete nature of digital images. Most commonly, the quantised set is made up of four edge

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

(a) x direction

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

(b) y direction

FIGURE 5.4: Sobel masks (a) calculates $\frac{\partial I}{\partial x}$ (b) calculates $\frac{\partial I}{\partial y}$

directions, corresponding to the locations of neighbouring pixels. Figure 5.5 shows the quantisation levels commonly used.

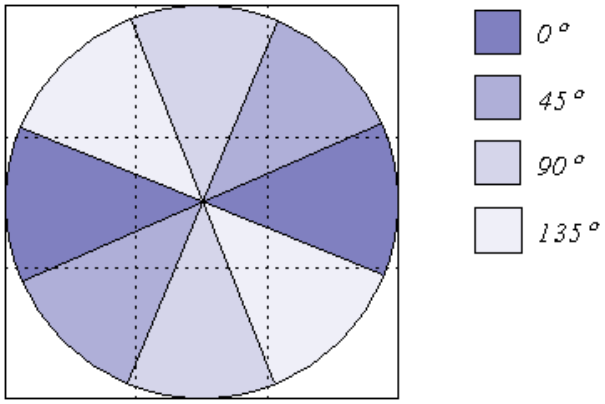


FIGURE 5.5: Quantised edge directions in a 3×3 pixel region.

Finally, the process of non-maximal suppression is applied to the gradient image to obtain an edge map consisting of thinned lines. The algorithm employs the thresholding hysteresis used in the Canny Edge Detector to improve the continuity of edges in the image.

The application of the AUV in underwater track following is tested in an artificial environment. Therefore, it is clear that the underwater track will have very distinct edges against the background pixels. This enables a higher upper threshold, T_U, to be used and still detect the edges of the pipeline whilst providing the advantage of eliminating false edges. The lower threshold can be set relatively low, as the edge pixel must be connected to an edge point that satisfies the upper threshold in order to be classified as an edge point.

5.4 Line Extraction from the Hough Transform

The Hough Transform is subsequently used, to accumulate votes for a set of straight line equations in the edge map. The process of extracting the most significant lines in this accumulator can sometimes be a cumbersome process. This involves searching the accumulator for cells that have a vote count exceeding a specified threshold value. At first glance, the most obvious solution would be to simply iterate through the accumulator array and take any line equation whose votes exceed the specified threshold. However, this does not produce adequate results as it produces line multiplicities. This occurs when a number of similar lines appear, that correspond to a single edge. Figure 5.7 shows the results obtained using this type of line extraction on the pipeline image in Figure 5.6, with subsequently increasing thresholds.

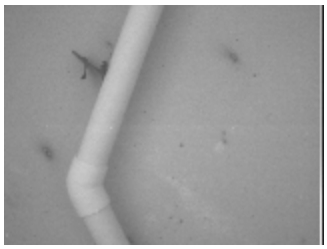


FIGURE 5.6: A test image of a turn in the underwater pipeline.

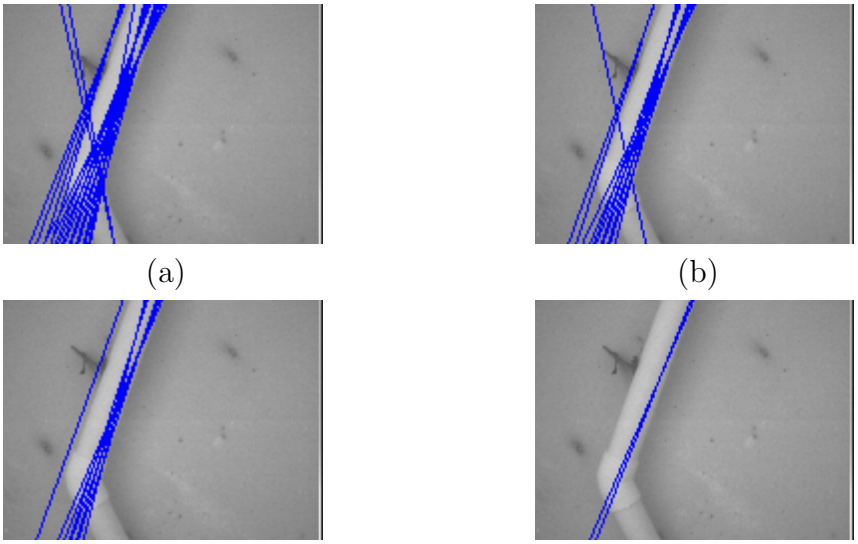


FIGURE 5.7: Line extractions from Figure 5.6 at different thresholds.

Figure 5.7 demonstrates the problems associated with this type of line extraction. In almost every image there are multiple lines being extracted for the same edge.

This is expected, as there is a deficiency in the edge detection process where edges can be of a greater width than one pixel. This can also be caused by using an excessive number of intervals for the ρ and θ parameter space, as small variations between line equations will usually result in them having a similar number of votes. There are two disadvantages in using these lines for positive track identification. Firstly, it increases the number of comparisons needed when having to search for parallelism between pairs of lines. Secondly, the multiple lines themselves may be considered as being parallel and may give false identification of the underwater track. Generally, the only way to remove these line multiplicities using this method of line extraction, is to increase the threshold value. This is still not a solution to the problem as an increase in threshold may remove the lines of the opposite edge of the pipeline as shown in Figure 5.7(d).

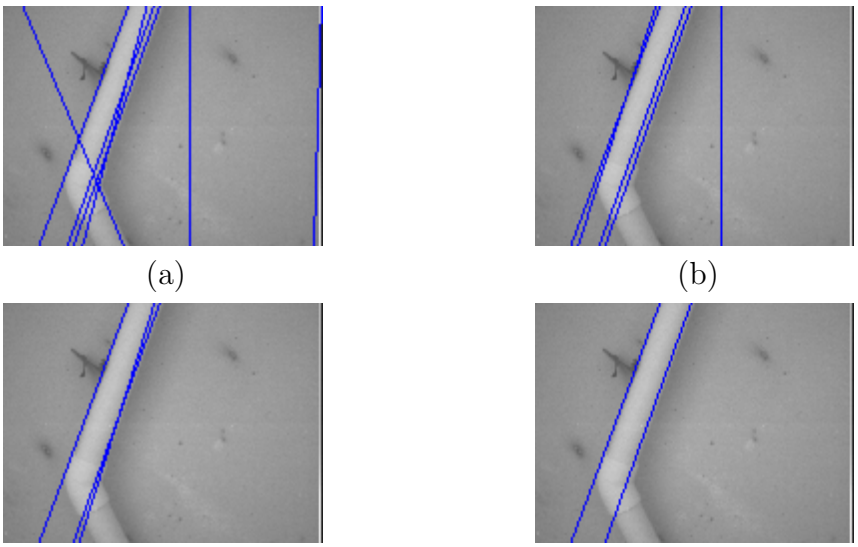
5.5 Improved Line Extraction

It is evident that the threshold value must be low enough to capture a fair range of line equations. However, it is highly desirable to remove the case where there are several lines of small variation which correspond to one particular edge length. This presents a need to group lines together that represent the same edge, which can be achieved with some additional processing.

Firstly, the Hough accumulator is thresholded using a lower value, to produce a mask of cells consisting of a larger range of lines. The resultant output is a binary image, containing distinct objects that represent the groups of lines that exceed the threshold value. Each group normally relates to one particular straight edge, where similar line equations satisfy the threshold value.

The objective now is to find the line within each group with the highest number of votes. This can be done via connected component labeling, described in Chapter 2, Section 2.3. Each distinct group in the binary image is given a label number, which is used to separate out a set of similar lines in the accumulator array, and find the equation with the highest number of votes within that group.

This additional processing gives a definite reduction in line multiplicities as demonstrated in Figure 5.8. Although the problem has been somewhat reduced, there are still cases where line multiplicities can occur. These will appear when a particular line group is separated into two distinct objects after thresholding the accumulator array.



(a) (b) (c) (d)

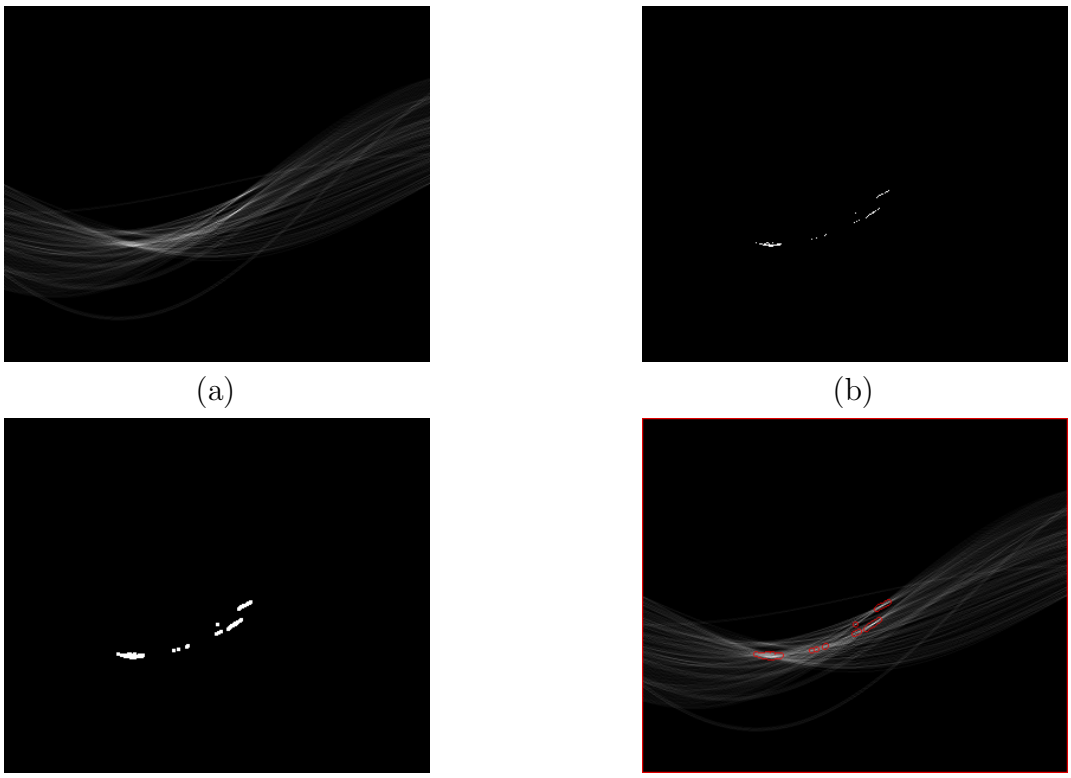
FIGURE 5.8: Line extractions from Figure 5.6 using improved line extraction.

5.5.1 Additional Morphology

A technique that will further improve the line extraction process, is to introduce morphological operations on the binary mask used to segregate line groups. With the use of a suitable structuring element, a dilation operation can be applied to the binary mask to join line collections that are relatively close together. This will further reduce the occurrence of line multiplicities. The use of morphology almost eliminates all occurrences of line multiplicities and results in a better line extraction process. Hence it improves the accuracy of the detection process. The overall process is demonstrated in Figure 5.9, with the resultant line extractions shown in Figure 5.10.

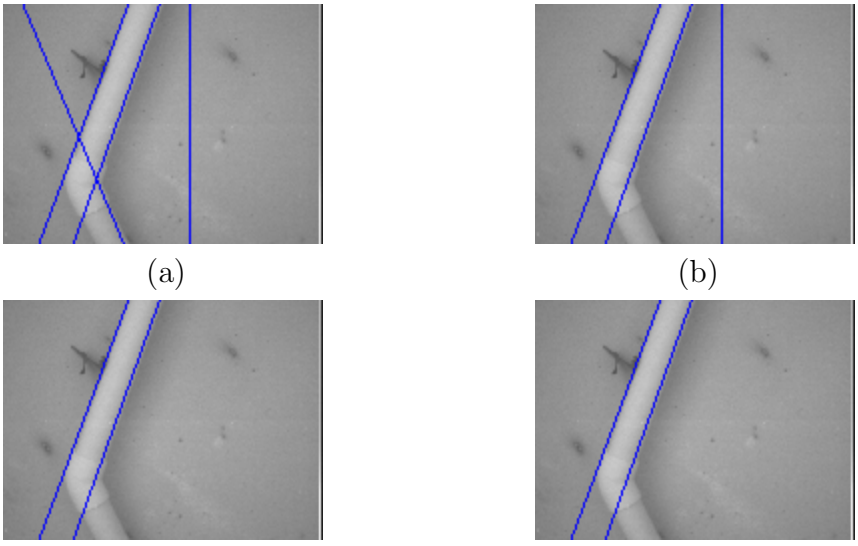
5.6 Finding Straight Lines of the Track

Once the most significant straight lines have been extracted from the image, it is now necessary to find those equations that actually correspond to the edges of the pipeline. To do so, the set of equations of must be parsed to find pairs of semi-parallel lines. Parallel pairs are obtained by first searching the set of lines for equations with similar θ values. These lines are then separated into pairs corresponding to their parallelism. Each pair of equations is then further analysed, to determine if the pair of equations actually corresponds to the pipeline. This is done by further comparing the ρ values of each equation. The difference in ρ values should be limited to a range,



(a) (b) (c) (d)

FIGURE 5.9: The line extraction process. (a) Original votes accumulator. (b) Accumulator filter mask. (c) Mask processed with morphology. (d) Resultant line groups.



(a) (b) (c) (d)

FIGURE 5.10: Line extractions from Figure 5.6 with additional morphology.

typical of the pipeline. Equations 5.1 and 5.2 represent the conditions placed on the differences in these pairs of equations, where β_θ and β_ρ are the acceptable differences in θ and ρ values respectively.

$$\Delta\theta = |\theta_1 - \theta_2| \leq \beta_\theta \tag{5.1}$$

$$\Delta\rho = |\rho_1 - \rho_2| \leq \beta_\rho \tag{5.2}$$

Figure 5.11(a), displays the results obtained when performing line extraction on the image in Figure 5.6. Figure 5.11(b) shows the results when line pairs that satisfy Equations 5.1 and 5.2 are extracted from this set of the lines. It is noticeable that equations were not produced for the bottom part of the track. If these need to be retained, then a greater β_θ tolerance must be specified. In general, these line pairs are only tested for their existence, so, if at least one pair of parallel lines does exist than the track has been detected and thus pipeline following can commence.

(a) (b)

FIGURE 5.11: (a)Line extraction using the Hough Transform. (b)Extraction of parallel line pairs.

5.7 Performance Analysis

For analysis of the track detection process, the performance issues are examined. Performance issues relating to the actual successful detection of the pipeline and the speed at which frames are processed are examined.

5.7.1 Detection Performance

To simulate the detection of the track, a range of test images were applied to the pipeline detection algorithm to gather data on the success rate of this process. Figure 5.12 shows some examples from a batch of 96 pipeline images and the lines detected

during the track detection process. In these images, only the pairs of parallel lines have been plotted if they have been found in the set of equations produced by taking the Hough Transform of an image. To perform pipeline detection, the θ parameter space was quantised into 360 levels, to attain accuracy to the nearest 0.5 *rads*, and the ρ parameter space was quantised into 200 levels, to attain accuracy to the nearest pixel unit.

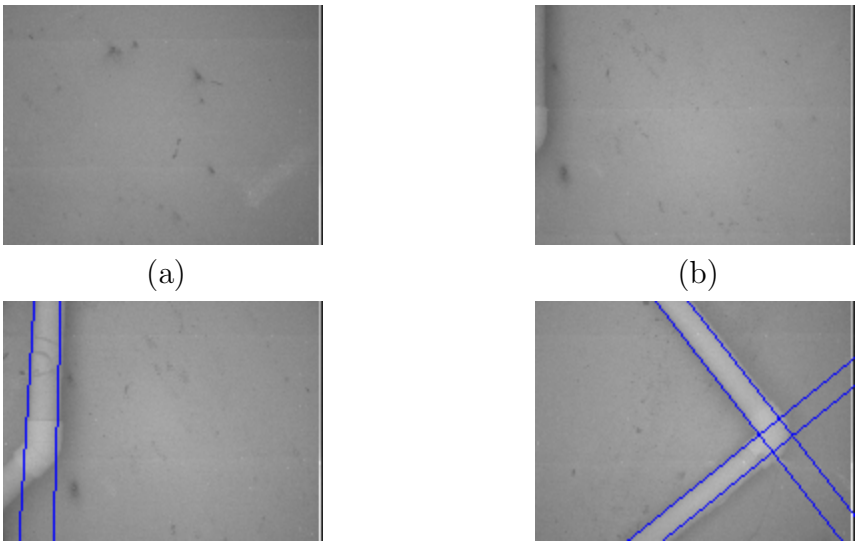


FIGURE 5.12: Detection of the underwater pipeline using the Hough Transform.

In each case, the same parameters were used for Gaussian filtering, edge detection and the line extraction process. Table 5.2 displays some performance results calculated for the detection algorithm using a batch of 96 test images. A description of the detection measures are;

- **Track is detected correctly**
  This is measured by taking the total number of correct detections and non-detections as a proportion of the total number of test images.

- **Track is falsely detected**
  This is measured by taking the total number of incorrect detections as a proportion of the total number of test image, when either;

  - the pipeline is not in the image
  - the pipeline is detected but not on its appropriate edges

- **Track is not detected**
  This is measured by taking the total number of non detections when the track is in the image as a proportion of the total number of images that contain the pipeline.

These results can be improved upon, if the β_θ and β_ρ tolerances are increased. By increasing these values, the probability of the track not being detected will be reduced, although it may result in an increased number of false detections.

Detection Measure	Performance
Track is detected correctly	77%
Track is falsely detected	3%
Track is is not detected	21%

TABLE 5.2: Performance measures for the track detection algorithm.

5.7.2 Computational Performance

The line extraction process is a complex process. In terms of the processing time required, the detection algorithm is reasonably expensive. Thus, the computational performance of the track detection algorithm is of particular interest, as there is a need to make sure pipeline detection occurs in real time.

Due to the speed limitations of the AUV, the line extraction process can occur as slowly as one frame per second without the AUV wandering off the track. An investigation follows into the selection of the ρ and θ parameter spaces that give reasonable precision in line equations whilst hopefully maintaining the specified performance limit. Generally, the size of the parameter spaces chosen will yield similar results in both surface and underwater applications. Therefore the parameters should generate similar results for the same application even with dynamic environmental conditions.

Table 5.3 shows the timing results obtained, when performing the Hough Transform and line extraction processes over 10 cycles. The results were then used to calculate the frequency of this process.

When measuring the actual detection performance, it was determined that the a Hough parameter space of 200 ρ intervals and 360 θ intervals yielded a good range of results in terms of line extraction. However, the performance results in Table 5.3 indicate that this precision cannot be achieved using the current track detection algorithm and vision system hardware. Even for a relatively small parameter space

ρ **intervals**	θ **intervals**	**Trial 1 (s)**	**Trial 1 (s)**	**Trial 1 (s)**	**Mean**
100	100	17	17	17	17.0s, 0.6 Hz
200	200	27	27	26	26.7s, 0.4 Hz
300	300	35	34	34	34.3s, 0.3 Hz
400	400	50	50	50	50.0s, 0.2 Hz

TABLE 5.3: Measurement of computational performance for track detection over 10 cycles.

of 100 ρ intervals and 100 θ intervals, processing cannot be achieved at a rate better than one frame per second.

The track detection algorithm was then performed on an alternative microprocessor, to determine if the software contains major inefficiencies. The specifications of the hardware are as follows;

- 1.6GHz Celeron Processor
- 256Mb RAM
- CYGWIN, Windows XP O/S

The results shown in Table 5.4 were calculated by measuring the execution time of the track detection algorithm over 100 cycles. The increase in the number of cycles used was necessary to obtain a longer execution time, in order to compensate for the increase in performance.

ρ **intervals**	θ **intervals**	**Trial 1 (s)**	**Trial 1 (s)**	**Trial 1 (s)**	**Mean**
100	100	12	11	11	11.3s, 8.9 Hz
200	200	15	14	15	14.7s, 6.8 Hz
300	300	18	19	17	18.0s, 5.6 Hz
400	400	29	29	30	29.3s, 3.4 Hz

TABLE 5.4: Measurement of computational performance for track detection over 100 cycles on an alternative micro-processor.

The results in Table 5.4, indicate that the track detection algorithm requires a faster processor, to be executed in a timely manner. With the 1.6Ghz Celeron processor, multiple frames per second can be processed during the track detection algorithm. The results also show that the track detection algorithm, using a parameter space of 200 ρ intervals and 360 θ intervals, still results in multiple frames per second being processed. Hence, it is highly desirable to update the microprocessor of the vision system.

5.8 Summary

It is evident that the line extraction process requires an appropriate choice of the accumulator threshold value. Specifying the threshold too low, results in an excessive amount of extracted lines, whilst a threshold too high may not yield enough extracted lines. This poses a particular problem when several line equations of small variations in ρ and θ parameters are exposed when they correspond to the same straight edge. Figures 5.7 and 5.8 demonstrated this problem, when a poor line extraction algorithm is used.

The extraction process can be greatly improved by using techniques such as connected component labeling and morphological operations. By processing the binary mask used to extract lines, it allows a larger number of intervals to be used for the ρ and θ parameter spaces, without introducing line multiplicities. However, these operations require additional computations which reduce the performance of the line extraction process. Optionally, it may be more viable to improve the edge detection algorithm such that it gives thinned edges of no more than one pixel widths.

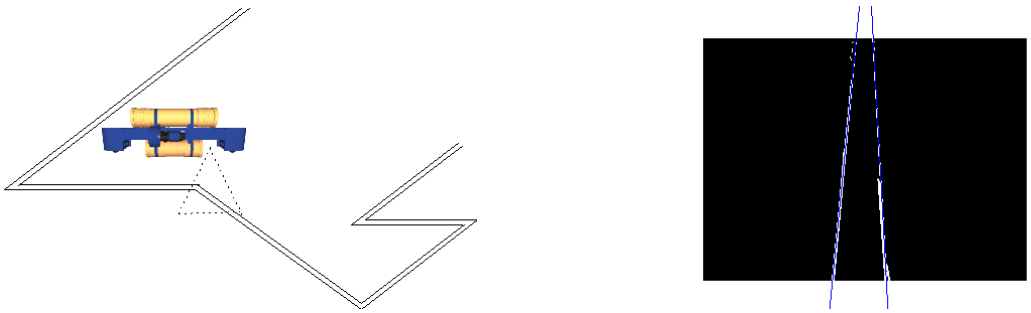
Computational performance analysis suggests the the track detection algorithm cannot be performed at the desired rate of one frame per second, on the current microprocessor hardware. Hence improvements must be made, either to the detection algorithm or the vision system hardware in order to achieve a faster processing rate.

Overall, this type of track detection is fairly robust in the artificial test environment. In most of the images taken, the underwater pipeline will have a distinct edge boundary against the background of the image. In industrial applications however, such as underwater cable tracking, this type of pipeline identification may not be as effective. Nevertheless, straight line segments are normally a rare property of natural underwater objects. So, even if the pipeline or track is covered in sediment or marine growth, the minimal amount of pipeline that is exposed could still be enough for robust identification using straight lines.

5.9 Extensions

The application of the the Hough Transform in track detection has relied on the assumption that the underwater track lies a horizontal plane. I.e. the track does not ascend or descend in the vertical directions. This procedure can be extended, to cater for tracks that do not lie on a horizontal plane. This can be achieved by modifying the feature that is detected.

In this application, the track detector searches for parallelism between lines to identify the track. If a track is vertically angled, this feature will not be produced due to the intrinsic perspective properties of images. An alternative detection feature, may be to search for any pair of lines that do not meet or intersect in the image. These lines could then be used to make the AUV follow the track at a constant distance according to the parallelism of the lines.



(a) (b)

FIGURE 5.13: Examples of an non-uniform horizontal track. (a) AUV field of view. (b) Example line extraction.

Consider the track example given in Figure 5.13(a) and an example of the possible edge map obtained by the vision system in Figure 5.13(b). According to the parallelism of the lines, the AUV could be instructed to adjust its pitch angle until the Hough lines become roughly parallel. This would give the AUV the ability to maintain a parallel distance from the underwater track.

Chapter 6

Track Location

Upon detection of the underwater track, the goal of the vision system is now to guide the AUV in autonomously following the pipeline. To do so, it is necessary to find some method of relating the position of the pipeline to the AUV. This information is needed so that the AUV can effectively keep the underwater track within its field of view. The most simplistic method used for determining an object's position in an image, is by determining the location of the centre of mass or centroid of the object, as discussed in Chapter 2.

The fixed location of the frame capture device, allows the geometric centre of an image to be defined as the relative position of the AUV. Thus, when determining the location of objects in a picture, it is desirable to use a co-ordinate axis with the origin defined at the centre of the image, as shown in Figure 6.1. However, the calculation of the centroid using Equations 6.1 and 6.2, yields a result based upon the co-ordinate axis shown in Figure 2.1. It is therefore necessary to transform the centre of mass location to a position relative to the AUV co-ordinate axis, so that they may be used for guidance.

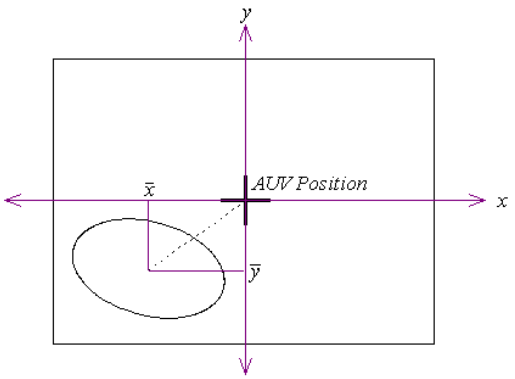


FIGURE 6.1: Object co-ordinates relative to the AUV position.

The calculation of the centroid requires a number of steps to yield the result and also improve accuracy. These include;

1. binary thresholding,
2. processing the binary image with morphological operators,
3. labeling connected components and extracting the object of interest (optional) and
4. calculating the centre of mass or centroid.

The labeling of connected components is an optional step in this process. This is only required when segregation of particular objects from a binary image is needed. In the track following application, connected component labeling is not used, as the test environment is artificial and generally there are no objects within the image of a size and colour that is characteristic of an object. In natural environments however, numerous objects may appear in the binary image that cannot be appropriately removed via morphological operations. Hence, this labeling process is only needed when the results of morphology are inadequate.

Once all preprocessing has been completed, the vision system calculates the centre of mass of the pipeline in relation to the AUV axis. These co-ordinates can then be input to a control algorithm, discussed in Chapter 9, for positioning the AUV over the track. The general aim is to keep the centre of mass location within a small region about the centre of the image, in order to keep the pipeline in the optimum field of view.

6.1 Centre of Mass Equations

Equations 2.2 to 2.3, given in Chapter 2, calculate the centroid co-ordinates $\bar{x}$ and $\bar{y}$, according to a continuous two dimensional function. However, a digital image is represented by a two dimensional discrete function, $I[x, y]$. Thus, the corresponding discrete equations for calculation of the centroid are given by Equations 6.1 and 6.2.

$$\bar{x} = \frac{\sum\sum x^2 I[x, y]}{\sum\sum I[x, y]} \tag{6.1}$$

$$\bar{y} = \frac{\sum\sum y^2 I[x, y]}{\sum\sum I[x, y]} \tag{6.2}$$

6.2 Morphological Operations

In most cases, the binary thresholding process does not produce a *perfect* binary image, where only pixels that belong to the object are turned *on* and all background pixels are turned *off*. This means that the accuracy of the centre of mass location could be severely corrupted by noise in the binary image. For this reason, morphological operations are often adopted to process the binary image before any calculations take place.

6.3 Structuring Element Determination

The benefits produced from binary morphology are dependant upon a suitable choice in the structuring element. The structuring element selection can influence the performance of morphology, not only in terms of visual performance, but also computational performance. For example, a larger structuring element will tend to result in a longer computation time for the operation. When performing morphology on binary images, a number of factors must be taken into consideration;

1. the size of speckles or salt and pepper noise in the image,
2. the shape and size of the object,
3. existence of gaps in the object and
4. performance requirements.

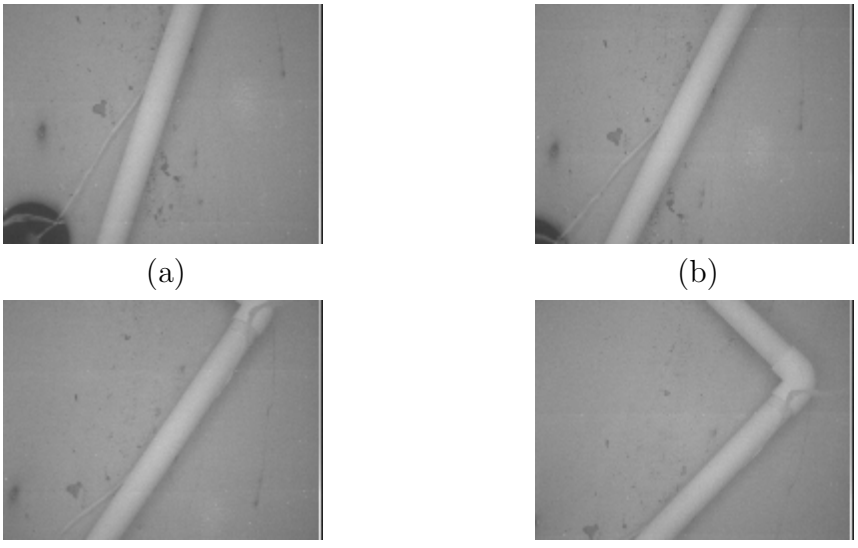
An opening operation, described in Chapter 2, Section 2.2.4, is used to remove random noisy pixels from the raw binary image. To accomplish this completely, the structuring element must be of a size greater than the largest noisy object, but of a size less than the pipeline object in the image. Specifying a structuring element larger than the track object, may result in the pipeline being eroded away making the centre of mass undeterminable. The closing operation is a useful follow on process, that can fill in any gaps within the track object. Hence in closing, the choice of structuring element is mainly governed by the size of the gaps in the track. Closing usually takes place after opening so that noisy objects do not grow in size.

The structuring element choice is also influenced by the actual application in which the morphology is applied. The AUV requires real time guidance from the visual system and hence, morphological processing should not significantly delay the calculation of the centre of mass. To reduce the computation time of the morphological operations, the structuring element used in preprocessing does not exceed a

square size of 3×3. This means that noise cannot be completely removed from a binary image if it is larger than a 3×3 square. Consequently, this places more emphasis on choosing an appropriate threshold value when obtaining the initial binary image. The second influence on the determination of a structuring element is the geometric properties of the underwater track, in particular its straight edges. The resultant structuring element chosen is a simple 3×3 square, with the origin defined as the centre pixel.

6.4 Morphological Operation Performance

The aim of preprocessing a binary image with morphological operations is to enhance the accuracy of the centre of mass calculation of the pipeline. The strange lighting conditions of the underwater environment often contribute to noisy pixels in the binary image as shown in image sequence in Figure 6.2. When this noise is concentrated in a certain area of the picture, the centre of mass location is pulled toward the noise and may result in the AUV wandering off the track. Sections 6.4.1 and 6.4.2, demonstrate the importance of morphological operations on the accuracy of the centre of mass calculations. The centroid is calculated for the image sequence shown in Figure 6.2 and the centroid locations are then plotted on the images using the vision system software library.



(a) (b) (c) (d)

FIGURE 6.2: Image sequence of an underwater pipeline.

6.4.1 Raw Centroid Calculation

The calculation of the centroid was first performed without preprocessing the binary image using morphology. Figure 6.3 shows the locations of the centre of mass in the sequence of pictures given in Figure 6.2. In the images with high concentrations of noise in a single area, it is evident that the location of the centre of mass is greatly affected. Hence, there is a need for morphological operations to reduce the amount of noise in an image and improve the calculation accuracy.

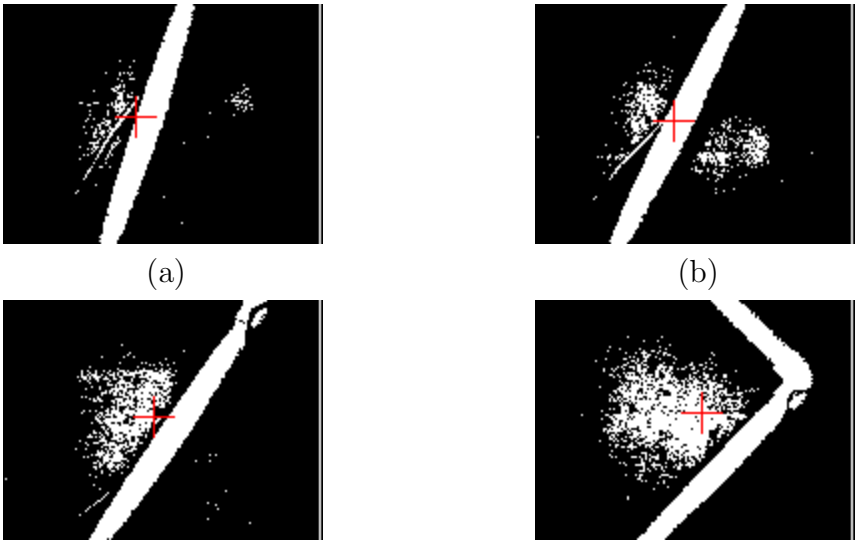
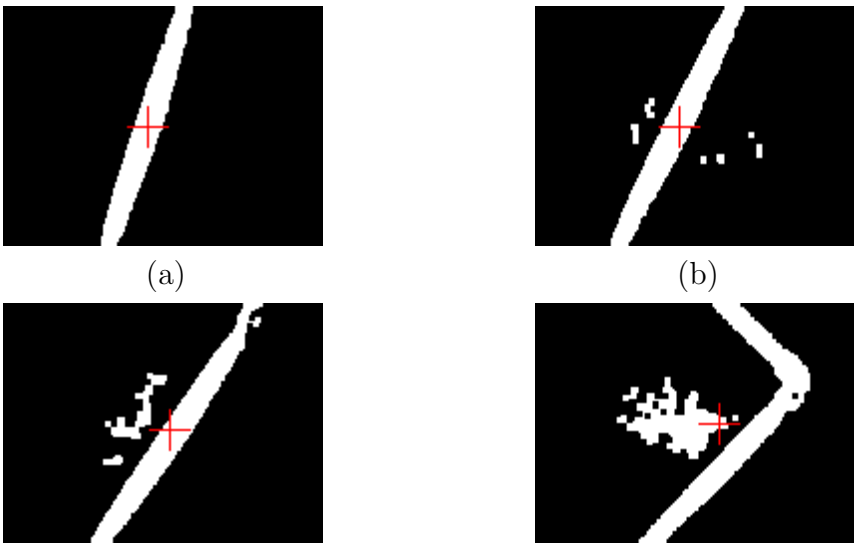


FIGURE 6.3: Image sequence of raw centre of mass calculations.

6.4.2 Processed Centroid Calculation

Figure 6.4 shows the centre of mass locations obtained after performing morphological operations on the image sequence in Figure 6.2 with a 3×3 square structuring element. An opening operation is first performed to remove the spread of the speckles predominantly to the left of the pipeline. This group of pixels is largely responsible for pulling the centre of mass location to the left, in the raw calculations in Figure 6.3. A closing operation is then performed to close up any gaps in the pipeline. The usefulness of the closing operation is demonstrated in Figures 6.3(d) and 6.4(d), where the majority of the gap near the corner of the pipeline has been filled. It is apparent that the morphological operations greatly improve the accuracy of the centre of mass calculations when comparing Figures 6.3 and 6.4. Thus, there is less likelihood that the AUV will wander off the pipeline.



(a) (b) (c) (d)

FIGURE 6.4: Image sequence of processed centre of mass calculations.

6.5 Computational Performance

The calculation of the location of the pipeline must occur in a timely fashion so that the AUV does not wander off the track. The actual calculation of the centroid is a fairly quick and efficient process. The morphological operations however, make up the bulk of the processing time required for the track location algorithm. The overall process has been analysed to determine the performance of the pipeline location algorithm.

Table 6.1 shows the timing results obtained when performing the thresholding, morphological operations and the centre of mass calculations over 10 cycles, using different sized structuring elements. The results were then used to calculate the frequency of this process.

$N \times N$ **size**	**Trial 1 (s)**	**Trial 2 (s)**	**Trial 3 (s)**	**Mean**
3×3	3	3	4	3.3s, 3.0 Hz
5×5	6	6	6	6.0s, 1.7 Hz
7×7	8	9	9	8.7s, 1.1 Hz
9×9	9	9	9	9.0s, 1.1 Hz

TABLE 6.1: Measurement of computational performance for track location over 10 cycles.

The results show that for a small structuring element, the track location algorithm can be performed in real time and thus provide on time navigational decisions to the thruster controllers.

6.6 Summary

The results in Figure 6.4 show the importance of the morphological operations as a preprocessing step, to increase the accuracy of moment calculations. There are significant improvements in the location of the centre of mass compared with the raw calculations shown in Figure 6.3. It was also confirmed that when using a small 3×3 square structuring, the morphological operations do not severely delay the calculations of the centre of mass. However, anything larger than 10×10 structuring element is likely to result in the frame processing rate dropping below one per second.

The location of the track is not sufficient enough information for the AUV to efficiently follow the pipeline as the vision system still has no knowledge of the direction of the underwater track. Although it may be possible to follow the track based purely on the centroid property, this is still an inefficient method, as the AUV may be facing in the wrong direction after aligning the centroid. This presents a need for the vision system to be able to determine the direction of the pipeline for use in navigation.

Chapter 7

Track Orientation

Once the AUV has aligned the location of the centre of mass within the central region of the image, it should now orientate itself in the direction of the pipeline. The AUV should subsequently start moving in the direction of the track and commence autonomous following. To achieve this, the vision system must obtain information regarding the orientation of the pipeline. The vision system will then need to instruct the AUV to orientate itself in the calculated track direction and proceed to move forwards.

Using the axis of minimum inertia property, discussed in Chapter 2, the track direction can be determined as an angle, offset from the x-axis in a clockwise direction. However, this quantity must first be manipulated, in order to relate this angle to the AUV's orientation in the image plane. The fixed camera position allows the AUV's orientation to be represented by a vertical axis in the centre of the image. Figure 7.1 displays the angle of the AMI (ϑ) on the image plane, and shows the desired angle (ϕ), as an angle offset from the vertical axis. The desired angle is calculated by using Equation 7.1.

$$\phi = -\frac{\pi}{2} + \vartheta, \quad -\frac{\pi}{2} \leq \phi \leq \frac{\pi}{2} \tag{7.1}$$

There is a general constraint that must be adhered to such that the axis of minimum inertia remains a good representation of the track direction. The AUV must maintain a certain distance from the track such that the edges of the object appear in the image. This is because if the object fills the whole image, the axis of minimum inertia will simply be a horizontal line passing through the centre of the image.

This property of binary images is also extremely useful in determining a heading direction when the AUV approaches corners or turns in the track. Generally, the

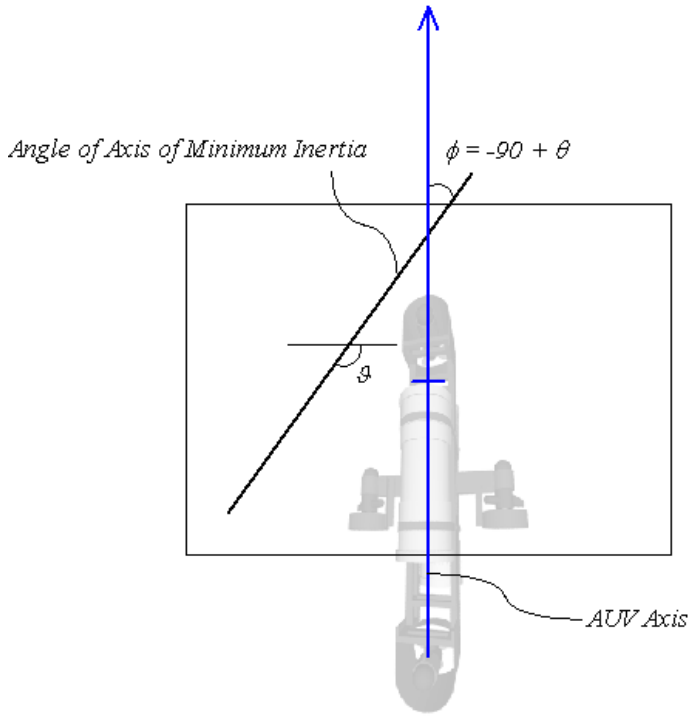
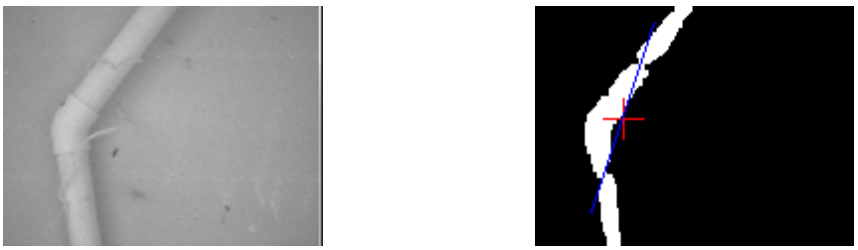


FIGURE 7.1: Angle of the axis of minimum inertia relative to the AUV axis.

axis of minimum is inertia is a straight line in which the pixels are distributed evenly on both sides of the line and spread out along the axis. The inverse of this is the axis of maximum inertia, which has a similar definition, but in this case the pixels are spread away from the axis. When the vision system obtains an adequate binary image of a corner or turn in the track, the calculated AMI angle is an excellent representation of the turn direction in every case. Figure 7.2 demonstrates the usefulness of this property in determining a turn direction for the AUV.



(a) (b)

FIGURE 7.2: A turning point of an underwater pipeline (a) Raw image (b) AMI calculation.

A number of steps are carried out prior to calculation of the axis of minimum inertia. Some are required whilst others are simply useful in improving the accuracy

of the calculation. These steps include;

1. binary thresholding,
2. processing the binary image with morphological operations,
3. labeling connected components and extracting the object of interest (optional),
4. calculating the centre of mass or centroid and
5. calculating the angle of the axis of minimum inertia.

The orientation calculation is generally performed as a follow on process from the centroid or centre of mass calculation, as the AMI equations require the centre of mass co-ordinates of the object.

7.1 Axis of Minimum Inertia Equations

For similar reasons as given in Chapter 6, Section 6.1, the Equations 2.4 to 2.6, given in Chapter 2 must first be manipulated so that they can be used for a discrete image function $I[x, y]$. Thus, the corresponding discrete equations for calculation of the parameters a, b and c are given by Equations 7.2 to 7.4.

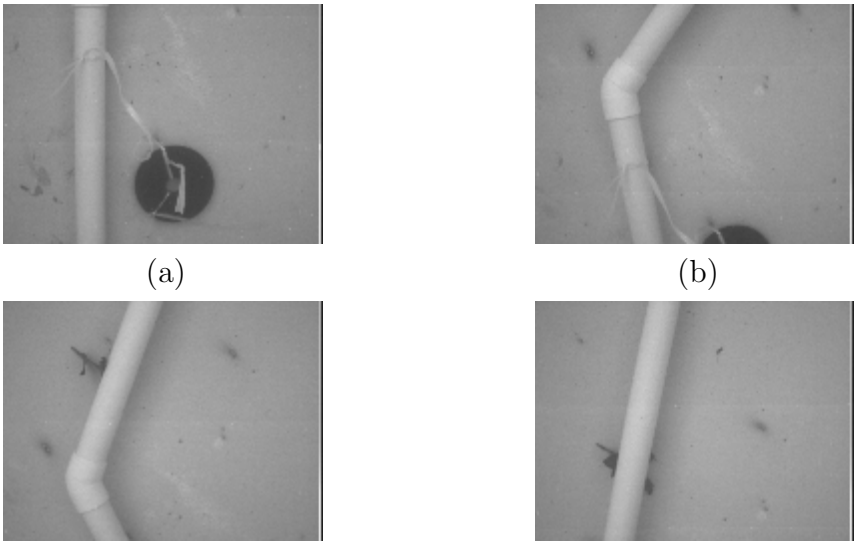
$$a = \sum \sum \tilde{x}^2 I[x, y] \tag{7.2}$$

$$b = \sum \sum \tilde{x}\tilde{y} I[x, y] \tag{7.3}$$

$$c = \sum \sum \tilde{y}^2 I[x, y] \tag{7.4}$$

7.2 Morphological Operation Performance

The AMI is a moments calculation that requires the use of a binary image. Hence, the calculation can benefit from the use of morphological operations in preprocessing the image. Calculations of the AMI are performed on the image sequence given in Figure 7.3 both with and without using morphology. A 3×3 square structuring element is used for the opening closing operations for the same reasons given in Chapter 6, Section 6.3.

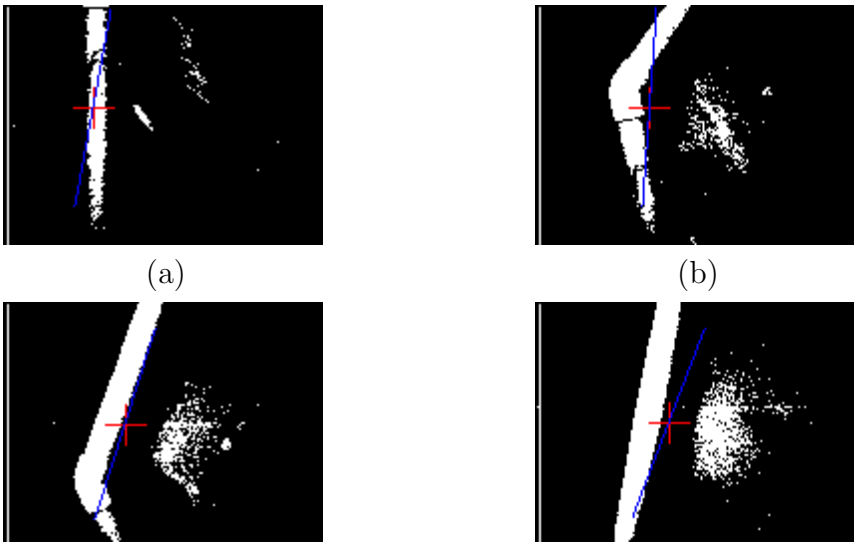


(a) (b) (c) (d)

FIGURE 7.3: Image sequence of an underwater pipeline.

7.2.1 Raw AMI Calculation

Calculation of the axis of minimum inertia were first performed without any morphological preprocessing. Figure 7.3 and the corresponding binary images in Figure 7.4, display the inaccuracies present when the binary does not undergo noise reduction.



(a) (b) (c) (d)

FIGURE 7.4: Image sequence of raw AMI calculations.

7.2.2 Processed AMI Calculation

The calculation of the AMI was then performed after processing the binary image with morphological operations. The results shown in Figure 7.5 demonstrate that the removal of noise using morphological operations has greatly improved the influence of the underwater pipeline on the AMI calculation. Consequently, this reduces the probability of the AUV heading away from the track and in the direction of the noise.

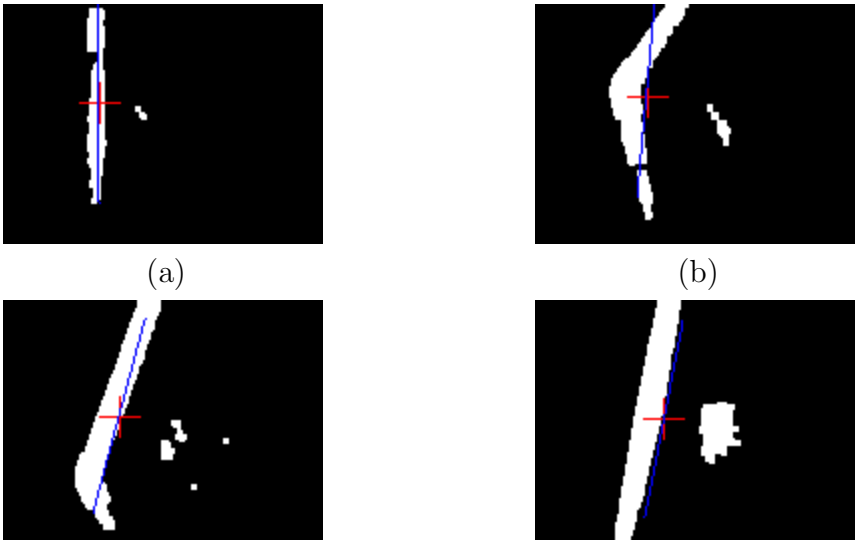


FIGURE 7.5: Image sequence of processed AMI calculations.

7.3 Computational Performance

The orientation of the track must be determined in real time so that the AUV can align itself with the pipeline direction and maintain the track in its field of view. The computation times measured, consist of the calculation time of the centre of mass and the the time taken to calculate the AMI.

Table 7.1 shows the timing results obtained when measuring the performance of the pipeline orientation algorithm over 10 cycles, using different sized structuring elements. The results were then used to calculate the frequency of this process.

The frame processing rates recorded in Table 7.1 are similar to those measured for the centroid calculations in Chapter 6. This is because the morphological operations are performed with the same structuring elements used in the previous chapter, and morphology takes up the majority of the processing time. As the AMI calculation

$N \times N$ **size**	**Trial 1 (s)**	**Trial 2 (s)**	**Trial 3 (s)**	**Mean**
3×3	4	4	4	4.0s, 2.5 Hz
5×5	7	7	7	7.0s, 1.4 Hz
7×7	11	11	11	11.0s, 0.9 Hz
9×9	12	11	12	11.7s, 0.8 Hz

TABLE 7.1: Measurement of computational performance for track orientation.

also involves the determination of the centre of mass, it indicates that the overall track location and orientation calculations can exceed the desired rate of one frame a second. Thus track following is achievable using the current vision system hardware.

7.4 Summary

The accuracy of binary image analysis is highly dependant upon the quality of a binary picture. Chapters 6 and 7 have demonstrated how easily noise in an image can corrupt the calculation of the centre of mass location and the axis of minimum inertia. Morphological operations seem to offer great improvements in terms of image quality and object clarity. However, in some cases it is almost impossible to remove the noise as demonstrated in Figures 7.5.

If more accuracy is needed, then connected component labeling can be used to separate out the pipeline. For example, in the binary image in Figure 7.5(d), there are two distinct objects that will be given a unique label. The larger object in this labeled image is the most likely to correspond to the underwater track, so by measuring each object's area it is possible to obtain a binary image containing only the pipeline.

The performance results shown in Table 7.1, also indicate that pipeline following is achievable using the current hardware setup of the vision system, as following only requires the calculation of the track location and orientation.

Chapter 8

Extended Binary Thresholding

Binary thresholding, described in Chapter 2, Section 2.1.1, gives relatively good results when performing pipeline location and orientation calculations in an indoor test environment. The artificial lighting conditions remain fairly constant, allowing most object pixels within an image to be distinguishable from the background pixels. In an outdoor environment however, natural lighting makes it extremely difficult to extract the object of interest through simple binary thresholding. Figure 8.1 shows an image of the pipeline taken in an outdoor environment with severe lighting noise in the picture.

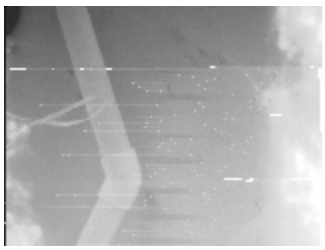


FIGURE 8.1: Pipeline image in outdoor conditions.

Figure 8.2 shows the results obtained when attempting binary thresholding on the pipeline image, where T_U and T_L are the upper and lower thresholds respectively. The process was first attempted with a loose range of threshold values and later trialled with more constrictive ones. The properties of the pipeline image however, indicate that binary thresholding is inadequate, no matter what threshold values are specified.

Using the binary images shown in Figure 8.2 to calculate the pipeline location and orientation yields unpredictable results. The centre of mass cannot be determined accurately, as the track cannot be separated from the background pixels effectively. Hence, an alternative method of binary thresholding is needed, that produces a more

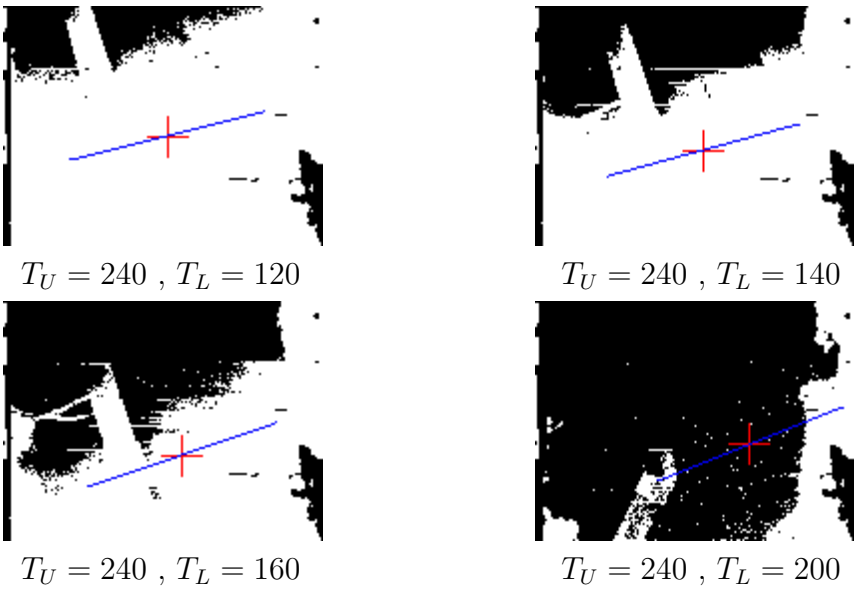


FIGURE 8.2: Binary thresholding in an outdoor environment.

precise binary image. I.e. one which provides a clear separation between the object and the background pixels.

8.1 Binary Thresholding using Edge Detection

Edge detection, described in Chapter 2, Section 2.5, provides an alternative method of binary thresholding. In this case, the binary image produced is an edge map, where the thresholding is performed based on the edge strengths. Although, the dynamic lighting conditions of the outdoor environment produces brights spots in the captured image, these spots tend to diminish smoothly. This means that edges are not significantly produced around these bright spots. Figure 8.3 is an example of a binary edge map, obtained from using an edge detector on the image in Figure 8.1. The figure also shows the corresponding calculations of the centre of mass and the AMI.

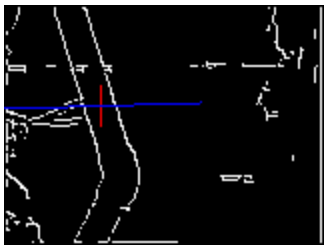


FIGURE 8.3: Binary edge map of significant edges

In Figure 8.3, the pipeline is much more defined. Thus, the resultant calculation of the centre of mass location has an improvement in accuracy when using a binary edge map rather than a binary thresholded image. Figure 8.3 clearly shows that the pipeline has the most predominant influence on the location of the centre of mass. However, due to the presence of other edges the same cannot be said for the AMI calculation. It is still difficult to determine the general orientation of the pipeline with the binary edge map. Hence, additional processing is still needed.

8.2 Binary Thresholding using Hough Transform

Improving the calculation of the AMI, requires removing the presence of edges that are not associated with the pipeline. This is achievable, if there is a way to determine which edges are of interest, and which edges should be discarded. By using the Hough Transform, it is still possible to detect the straight lines of the track even in an image degraded by noise. This is because the underwater pipeline will produce the most significant straight lines in the image, as other edges are created from uneven lighting conditions or natural objects that lack geometry. The resultant set of straight lines can then be used to filter the binary edge map, such that only edge pixels that lie on a straight line are kept whilst all others are discarded. It is important that straight lines are extracted for all parts of the pipeline, so that turns in the pipeline are not lost. It is important to retain these turns as they impact the calculation of the AMI.

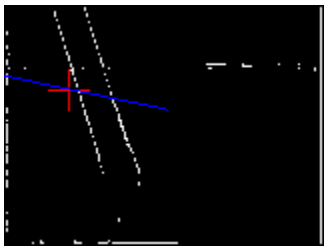


FIGURE 8.4: Binary image analysis with an edge map filtered with a Hough Transform.

Figure 8.4 shows the results of this process. In this image, edge pixels that do not lie along a significant straight line have been removed. This has slightly improved the calculation of the AMI as the orientation angle now tends closer to the actual track direction. However, there is still the presence of edges that do not lie on the pipeline, influencing the orientation angle of the AMI. The next logical step is to only retain edge pixels that actually lie on the underwater track. To do so, the set of

straight lines needs to be searched for any pairs of line equations that are relatively parallel. Any edge pixels that do not lie on a line equation of a pair of parallel lines is removed from the image.

Figure 8.5 shows an example of a binary edge map when the edge pixels have been filtered against a set of parallel lines corresponding to the pipeline. Although, this method does remove most edges that do not correspond to the pipeline, it is a little too pedantic in its removal of non pipeline edge pixels. The image shows that some of the pipeline edges have been lost in this process. Generally, better results are achieved when an allowance is made when determining if an edge pixel lies on a line equation. For example, when using Equation 2.15 to calculate the values of ρ, a tolerance can be introduced between the calculated value of ρ and the actual ρ value of the line equation. In doing so, the quality of the binary edge map is greatly increased and the subsequent moments calculations are improved, as shown in Figure 8.6.

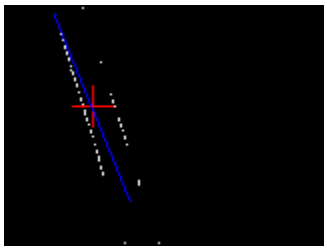


FIGURE 8.5: Straight line filtered binary edge map.

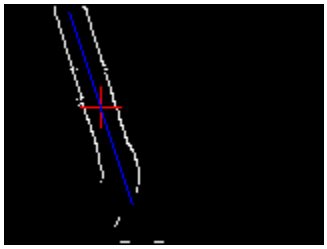


FIGURE 8.6: The final binary edge map.

8.3 Summary

It has been shown that by filtering binary edge maps with a Hough Transform, it is possible to obtain a clearer and better representation of the underwater track in a binary image. This has proved effective, in cases where natural lighting has introduced pixels of the same intensity as the pipeline and simple binary thresholding

fails to produce a clear separation of the object from the background pixels. In theory however, this can be applied to any instance when there exists pixels of same intensity of the pipeline and binary thresholding becomes inappropriate.

In a more realistic application of pipeline following, it cannot be guaranteed that the image will only contain the pipeline and no other objects of similar colour. For example , in an oceanic environment, corals and rocks of similar colour may be visible in the image, possibly covering the underwater pipeline. By using edge detection, it is possible to obtain a binary edge map that will outline every object within the image. The image can then be filtered against a set of straight line equations that are relatively parallel, so that only the pixels that lie on these straight lines are kept whilst others are removed. The resultant output will be a binary edge map containing only the outline of the pipeline.

In some cases, tolerance needs to be introduced when determining if an edge pixel satisfies a certain line equation. In doing so, more edge pixels corresponding to the pipeline will be retained in the final binary edge map, resulting in a more accurate calculation of the moments. This method of producing a binary image is more robust than traditional binary thresholding, even in the presence of uneven lighting and similarly coloured objects. Hence, it is more suited to a real application of pipeline following in a natural marine environment.

Chapter 9

Track Following Algorithm

The vision system must now provide some form of control to the AUV, based on the location and orientation information extracted from the image of the pipeline. This section outlines a control algorithm based on the centre of mass and the AMI calculations performed in Chapter 6 and 7 respectively. The algorithm is suitable for underwater tracks that lie completely on a lateral plane and thus do not ascend or descend in the vertical directions.

9.1 Desired Control Algorithm

The overview of the desired control algorithm for pipeline following is shown in Figure 9.1. The process first begins with track detection, where the AUV must initially search for the underwater pipeline. When the feature, described in Chapter 5, Section 5.1, has been detected within the image, the vision system assumes that the underwater track is in the field of view of the vision system. Hence, the track detection process should cease and the vision system should start aligning the pipeline in the optimum viewpoint of the camera. This involves maneuvering the AUV such that the centre of mass location moves into the geometric centre of the image.

9.1.1 Track Detection Algorithm

The track detection algorithm simply involves the AUV travelling forward until the underwater pipeline has been detected within the image. Detection occurs when at least one pair of semi-parallel lines that are suitably spaced apart are extracted from the captured image. Figure 9.2 outlines an algorithm suitable to perform the track detection process.

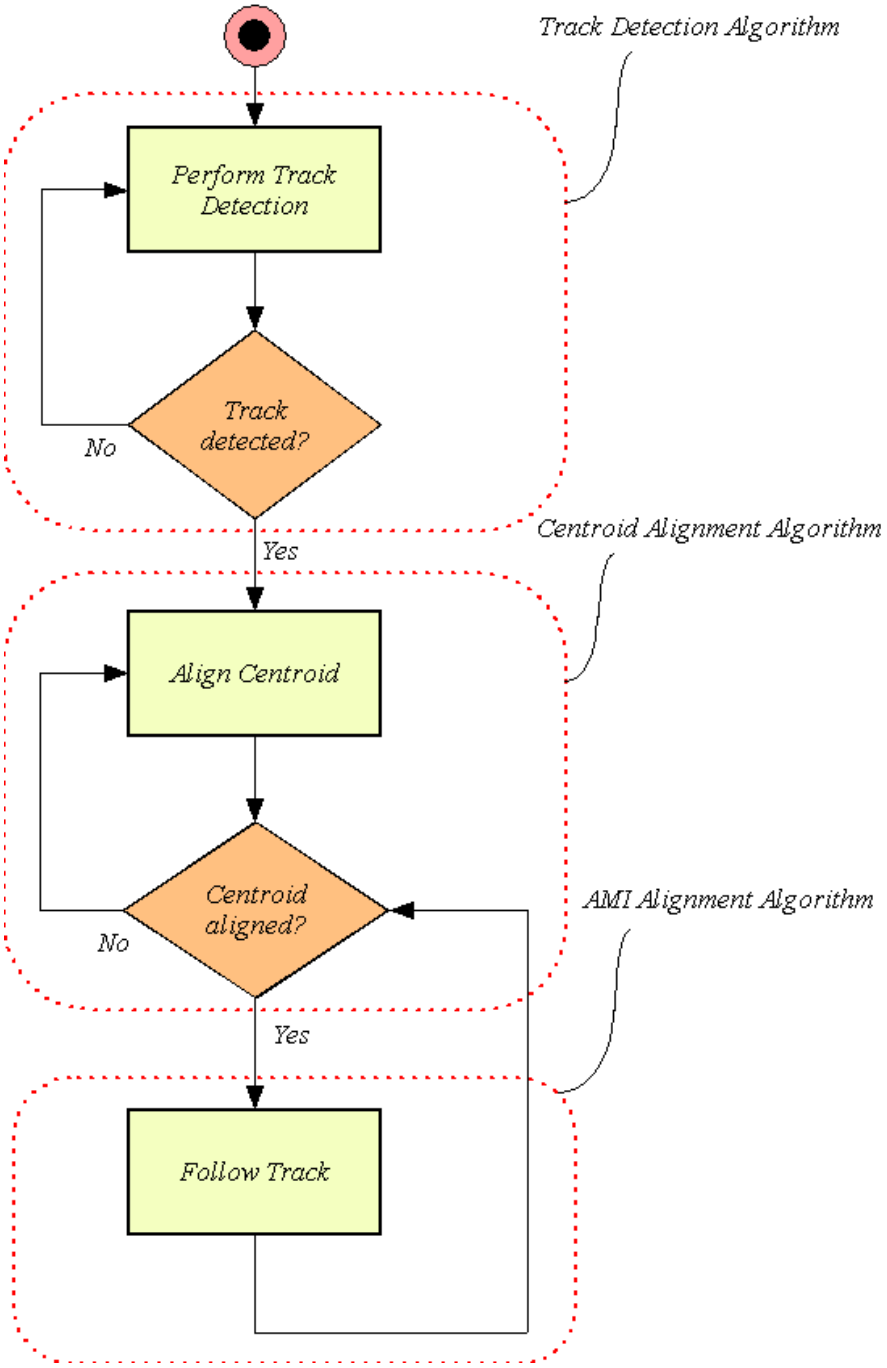


FIGURE 9.1: Flow Diagram: Desired AUV control algorithm.

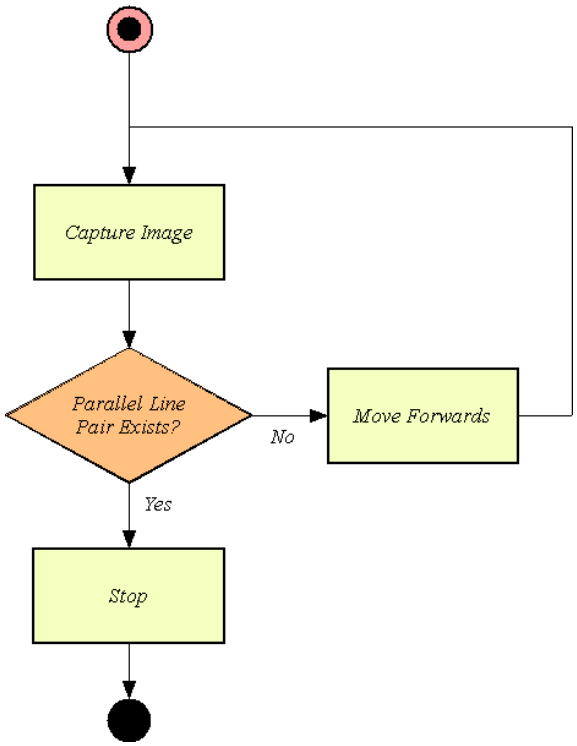


FIGURE 9.2: Flow Diagram: Track detection algorithm.

9.1.2 Centre of Mass Alignment Algorithm

Before the centre of mass can be aligned in the geometric centre of the image, the co-ordinates must first be modified such that they are relative to AUV axis, as shown in Figure 6.1. Equations 9.1 and 9.2 are used to calculate the polar co-ordinates, φ and s of the centre of mass, defined as the angle the AUV must rotate and the displacement it must travel respectively. Basically, this means the AUV should rotate until the x co-ordinate of the centre of mass tends to zero or is within some region of y-axis. The vehicle should then move forwards or backwards depending on the sign of the displacement.

$$\varphi = \begin{cases} -90 - \arctan\left(\frac{\bar{y}}{\bar{x}}\right), & \text{if } \arctan\left(\frac{\bar{y}}{\bar{x}}\right) \leq 0 \\ 90 - \arctan\left(\frac{\bar{y}}{\bar{x}}\right), & \text{if } \arctan\left(\frac{\bar{y}}{\bar{x}}\right) > 0 \end{cases} \tag{9.1}$$

$$s = \begin{cases} -\sqrt{\bar{x}^2 + \bar{y}^2}, & \text{if } \bar{y} \leq 0 \\ \sqrt{\bar{x}^2 + \bar{y}^2}, & \text{if } \bar{y} > 0 \end{cases} \tag{9.2}$$

The centre of mass alignment algorithm is outlined Figure 9.3, where β_φ and β_s are the tolerances accepted in relation to the geometric centre of the image. Once the co-ordinates of the centre of mass are in a suitable location, the centroid alignment

algorithm ceases and the AUV will commence the AMI alignment algorithm to perform track following.

9.1.3 AMI Alignment Algorithm

Once the track has been appropriately aligned in the field of view, the AUV should now orientate itself along the direction of the underwater pipeline. By using the AMI property of the images captured by the vision system, pipeline following should be more efficiently executed by the AUV. Generally, track following will be much smoother if the AUV simply travels in the direction of the pipeline, instead of continuously attempting to align the centre of mass. Equations 7.1 given in Chapter 7 are first used to calculate the angle offset, φ, from the vertical AUV axis.

The AMI alignment algorithm is outlined in Figure 9.4, where β_ϕ is the tolerance accepted in the offset from the vertical AUV axis. Subsequently, the vehicle should turn left if $\phi < 0$ and turn right if $\phi > 0$. The magnitude of ϕ can then be input to a control technique such as PID or fuzzy logic, in order to devise the motor speeds necessary, to reduce this offset to zero. Generally, a higher magnitude of ϕ should produce a higher turning speed and a lower magnitude should govern a lower turning speed.

The AUV should ideally be able to follow the track based solely on the calculated pipeline orientation. However, the vehicle is subject to drifting either from slow response in the thrusters or from the turbulent water. To compensate, the vision system should continually check the location of the centre of mass of the pipeline. Once the co-ordinates move outside a tolerable region, the AUV should cease track following and attempt to re-align the centroid of the pipeline within the middle of the image. This is done by using the centre of mass alignment algorithm discussed previously. In summary, the sole purpose of the AMI alignment algorithm is to ensure that the AUV is always orientated along the direction of the underwater pipeline and thus follows it correctly.

9.2 Summary

A track following algorithm has been devised based on the information pertaining to the underwater track's location and orientation. To accomplish this task, the AUV must provide certain maneuverability which as yet is incomplete. The aim of this chapter was to outline a suitable track following algorithm that can be implemented when the AUV control functionality is complete. To successfully use this algorithm,

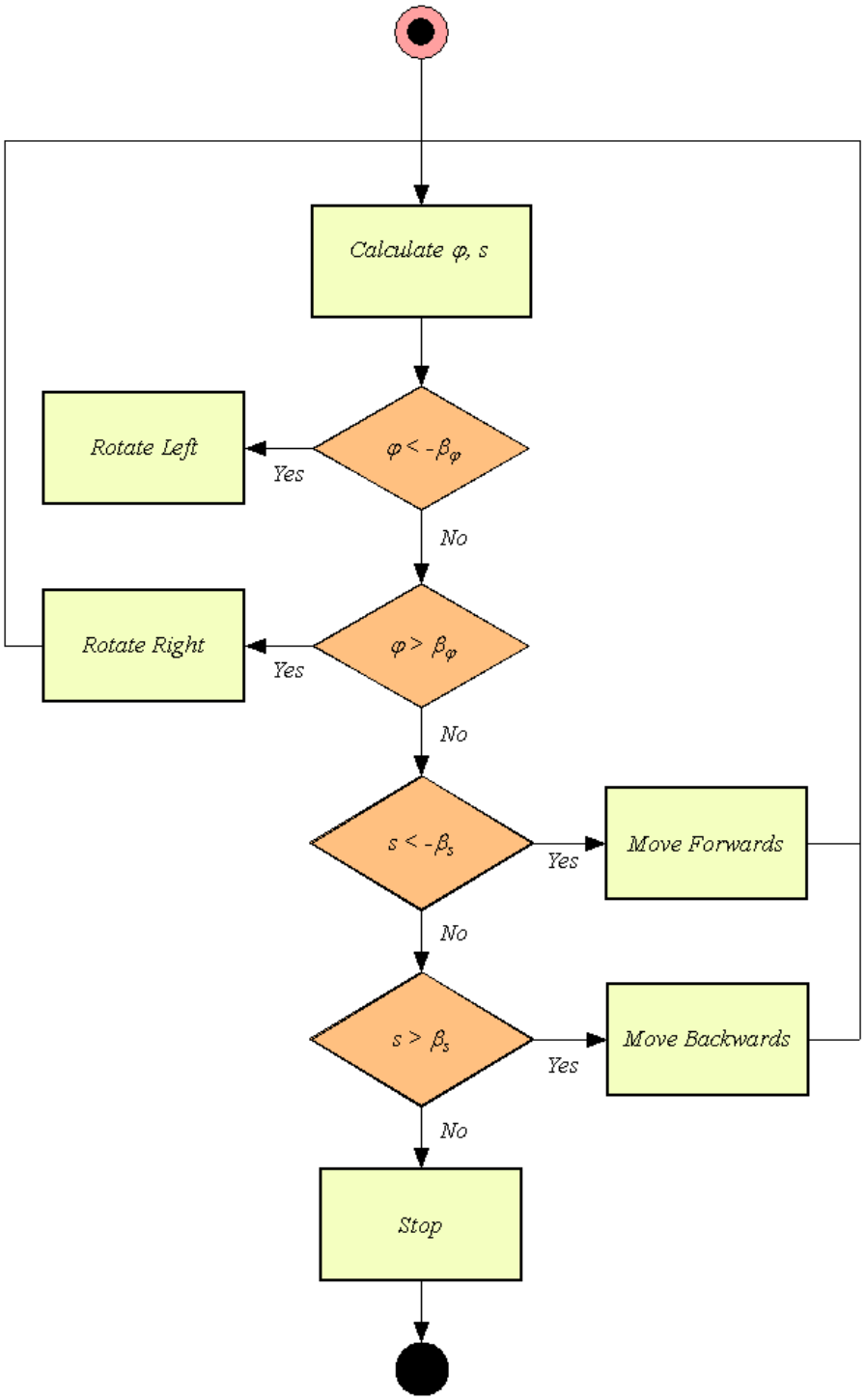


FIGURE 9.3: Flow Diagram: Centroid alignment algorithm.

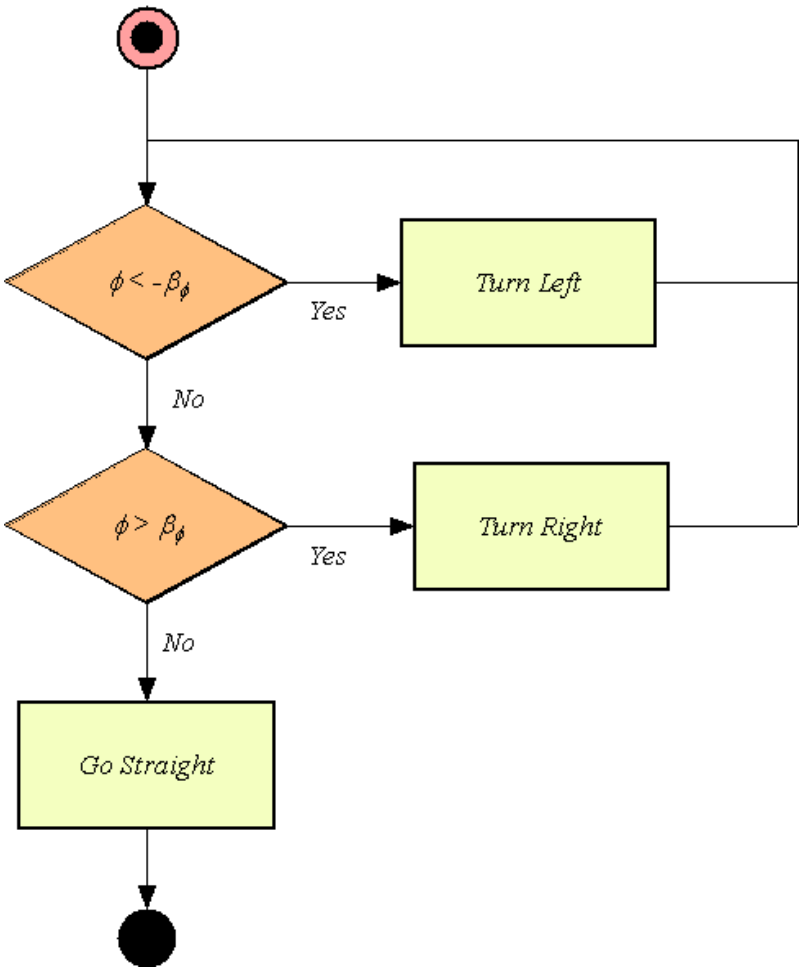


FIGURE 9.4: Flow Diagram: AMI alignment algorithm.

the AUV must provide controls to rotate on the spot, move in the forward and reverse directions in a straight line, and be able to stop and hold a stationary position. The magnitudes of ϑ, s and ϕ can later be used in some form of advanced control techniques, such as PID or fuzzy logic, to maneuver the AUV more accurately when following the pipeline.

Chapter 10

Conclusion

The implementation of a vision system for the University of Western Australia's autonomous underwater vehicle, named the *Mako*, has been presented in this thesis, with the major contributions outlined in Chapter 1, Section 1.5. The completion of the vision system has involved the compilation and interconnection of a range of hardware components and the design and development of an image processing library. The hardware encompasses the functionality of frame capture, image processing and communication with the AUV's thruster controller. Image processing is then achieved using the vision system software in which a number of core functions are provided that accomplish most of the vision processing techniques discussed in Chapter 2. An investigation was then carried out, into the application of the vision system in underwater track or pipeline following, of which several useful image processing techniques were highlighted.

10.1 Track Following Application

A number of techniques were investigated to achieve pipeline following with the implemented vision system. Using a number of historical computer vision techniques, methods were devised to detect the underwater pipeline and determine its location and orientation. However, trialling the system in an outdoor environment also highlighted the need for a more robust form of binary thresholding. This section discusses the major outcomes exposed from these investigations.

10.1.1 Track Detection

Detection of the underwater pipeline is performed by using the Hough Transform to determine the most significant straight lines in an image. The geometric nature

of the underwater track allows these lines to be used in identifying the presence of the track. To do so, the array of line equations is searched for at least one pair of semi-parallel lines that are appropriately spaced apart. Generally, the most difficult stage of the detection process is the extraction of the most significant lines from the accumulator array. In order to minimise the occurrence of line multiplicities, where a number of lines are extracted that correspond to the same straight edge, connected component labeling and morphological operations are introduced. These operations are performed on the filter mask obtained by thresholding the accumulator array against a specified number of votes. In doing so, similar line equations are grouped together and then only the most significant line within each line group is extracted, thus reducing the occurrence of line multiplicities.

This is generally the most computationally expensive process performed during pipeline following. The performance measurements obtained indicate that this type of track detection is not feasible due to the poor computational performance of the algorithm. Improvements must be made to either, improve the efficiency of the detection algorithm, or improve the performance of the vision system hardware.

10.1.2 Track Location

The location of the underwater track relative to the AUV needs to be determined so that the AUV can maintain the pipeline in its field of view. The centre of mass property of binary images can be used to calculate the co-ordinates of the underwater track within the binary image. To do this accurately, requires an adequate binary image, which can be defined as one that shows a clear separation between pixels belonging to the object (the pipeline), and pixels belonging to the background. Perfect binary images are often hard to produce and thus morphological operations are very useful in cleaning up the binary image before any moments calculations are performed. The results in Chapter 6 show a genuine improvement in the centre of mass calculation of the underwater pipeline when preprocessing binary images using morphology.

10.1.3 Track Orientation

Although the centre of mass can be used to position the underwater track in the centre of the field of view of the AUV, the vision system still has no knowledge on the direction of the track. The geometric properties of the underwater pipeline allow the axis of minimum inertia to be used to determine an orientation heading for the track. Once again the use of morphological operation is imperative in improving

the accuracy of these calculations. Chapter 7 has demonstrated that the AMI is particularly useful in the presence of turns in the underwater pipeline, as the angle of the AMI coincides with the general direction the AUV must head in to follow the track in a straight line.

10.1.4 Extended Binary Thresholding

When attempting to capture pipeline images in an outdoor setting, the natural lighting introduced severe noise in the resultant picture, presenting a need for a more robust method of binary thresholding. The most simple form of binary thresholding which involved retaining pixels that exceeded a certain intensity value proved inappropriate as bright spots in the image tended to be of the same pixel intensity as the underwater track. Consequently, the resultant binary images could not be used to accurately calculate the moments needed for determining the location and orientation of the pipeline.

A new form of thresholding was introduced that makes use of the Hough Transform. Binary image analysis is now performed on an edge map instead of a binary thresholded image. The results produced tend to provide an improvement in the centre of mass calculations but the AMI calculations were still affected by the presence of edges not related to the pipeline. To remedy this, edge pixels are filtered against pairs of equations that correspond to the straight parallel lines of the track edges. However, this proved too restrictive on the retaining of edge pixels, hence, tolerance must be introduced when determining if an edge pixel lies upon a certain equation. The resultant edge map provides great improvements in the centre of mass and AMI calculations compared to the original binary image.

10.1.5 Track Following Algorithm

A track following algorithm has been outlined in Chapter 9 that utilises the centre of mass and AMI calculations in guiding the AUV to follow the track autonomously. However, successful navigation is highly dependant upon precise control of the AUV which has is currently not completed. An attempt to follow the pipeline using a simplified control algorithm did not yield any results as currently the AUV cannot be maneuvered precisely resulting in it wondering off the track almost instantaneously.

10.2 Recommendations and Future Work

A basic vision system has been created that allows the capturing and processing of images. Much work still needs to be done to improve on both the hardware and software components of the system.

10.2.1 Camera Hardware

A current limitation of the vision system is the poor performance of the frame capture device. This limitation can prevent the image processing from occurring in real time. It is recommended that the *video4linux* [15] package be installed on the operating system of the processor. This will allow the use of a *usb* web camera that offers much higher frame capture rates thus removing the processing bottleneck of the camera hardware.

10.2.2 Vision Microprocessor

It was demonstrated in Chapter 5, that the desired processing rate of one frame per second could not be achieved using the track detection algorithm. It is recommended that the microprocessor of the vision system be updated in order to provide better processing performance in the vision based algorithms.

10.2.3 Track Following Control Logic

When proper control of the AUV has been implemented, it is advisable to investigate the use of various control techniques such as PID and fuzzy logic to perform track following. This will involve using the outputted quantities from the techniques described in this thesis as inputs to a control system.

10.2.4 Object Recognition with Neural Networks

Pipeline following is just one of the many commercial applications AUV's are particularly useful for. This research deals explicitly with the detection of pipeline structures using a specific computer vision technique, namely the *Hough Transform*. However, in order to make an AUV commercially viable, it is desirable to create an adaptable object detection system that is able to recognise several different objects. This can be achieved by finding features of objects that are invariant to object rotation, translation and scaling [19]. A neural network can then be trained to actually recognise objects according to their features.

10.2.5 Improve Edge Detection Algorithm

In some cases, the binary edge maps produced by the implemented edge detection algorithm contains thick edges. This can affect the track detection process, in that it will introduce line multiplicities when extracting straight lines from the image. Although, various techniques have been implemented to reduce the occurrence of this happening, an alternative is to improve the edge detection algorithm, in particular the process of non-maximal suppression. In the implemented algorithm, a simplified version of this process is used. Improvements can be made by implementing the original non-maximal suppression algorithm, described in Chapter 2.

10.2.6 Testing Platform

It is advisable to create an improved testing platform that enables real time visual feedback from the vision system. A brief investigation has shown that file transfer can be achieved through the use of sockets. This will allow the vision system to continuously send images over the wireless network to a remote computer. Implementing the test platform will provide an excellent aid in the determination of visual processing parameters required by the system.

Appendix A

Specifications

A.1 Hardware

This section gives the specifications of the major hardware used for the vision system.

A.1.1 Connectix Quickcam

The Connectix Quickcam provides the frame capture ability for the vision system.

- CCD: charge coupled device
- Frame Size: 320×240, 160×120, 80×60

A.1.2 Wizard Processor

Specifications for the image microprocessor were obtained from [23].

FIGURE A.1: Wizard PC [20].

- 233MHz Processor

- 32Mb RAM
- 5Gb storage
- 2 x RS232 ports
- 1 x Parallel port, supports SPP/EPP/ECP mode
- 2 x USB ports, USB 1.0 compliant
- 10/100 Mbps ethernet
- Username: root
- Password: temp

A.1.3 Eyebot

The Eyebot controller is responsible for the control of the thrusters but does not have the computational power for intensive image processing. Hence, the Eyebot is interfaced to the Wizard PC through an RS232 serial port.

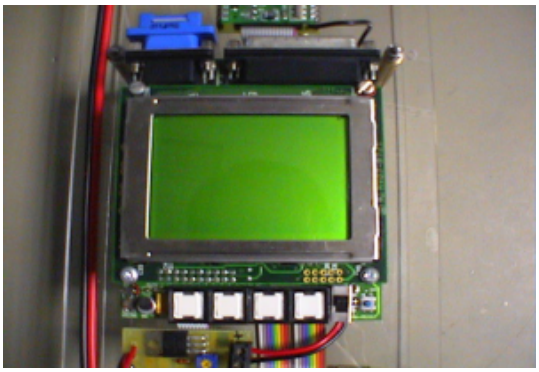


FIGURE A.2: Eyebot micro-controller [20].

The specifications obtained from [24] are as follows;

- 25MHz 32bit Controller (Motorola 68332)
- 1MB RAM
- 512KB storage
- 2 x Serial ports
- 1 x Parallel port

A.1.4 Minitar Wireless Access Point

The access point is used to allow remote communication with the image microprocessor. There was no need for reconfiguration of the network settings on the Wizard PC for use with the access point.

FIGURE A.3: Minitar wireless access point.

The specifications are as follows;

- IEEE 802.11b Compatible
- Power: 12VDC, 0.5A
- IP: 192.168.2.1
- Username: admin
- Password: 1234

A.2 Software

This section gives the specifications any software used for the vision system that has been taken from others.

A.2.1 cqcam

The cqcam software package is used to provide software frame grabbing capability with the Connectix Quickcam hardware [16].

Features;

- Automatic brightness and colour adjustment

- 24bpp and 32pp colour mode
- Frame Size: 320×240, 160×120, 80×60

A.2.2 adrport

Modifications have been made to the serial port code obtained from [17]. The serial port code is compatible with the Linux operating system installed on the vision micro-processor. Should the operating system be changed, alternative serial port libraries must be used.

Appendix B

Image Processing Fundamentals

This appendix addresses the fundamentals of computer vision. The ways in which images are stored digitally, the process of convolution and the use of masks in the filtering process are explained here.

B.1 Digital Images

The bulk of this thesis concentrates on the processing of digital images. Consequently, it may be helpful to understand how a picture is actually stored in memory. Physically, images tend to be stored in a one dimensional array in memory. Conceptually, it is easier to visualise this as a two dimensional matrix that consists of a certain number of rows and columns.

B.1.1 RGB

RGB is an acronym for red, green and blue. This type of image is commonly used for storing colour pictures in a digital format. Each cell of the two dimensional matrix is broken down into three sub cells that each store an intensity value for each of the corresponding red, green and blue colour channels. This concept is illustrated in Figure B.1.

B.1.2 Grayscale

Grayscale images are used to represent black and white pictures. In a grayscale image, every cell of the two dimensional matrix stores on intensity value. Hence, grayscale images not only take up less memory than their RGB counterparts, but processing of grayscale images is generally less complex and computationally faster. Figure B.2 gives an example of a digital grayscale image.

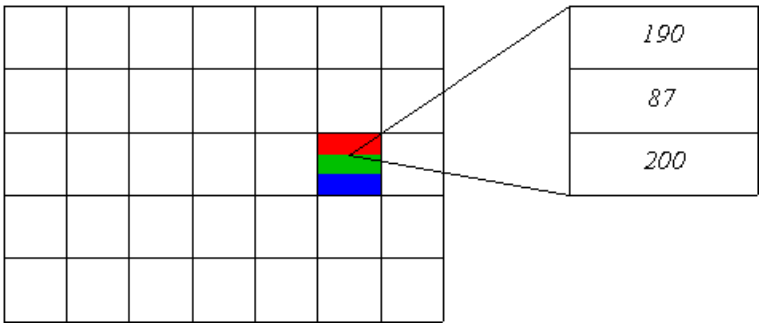


FIGURE B.1: An RGB colour image.

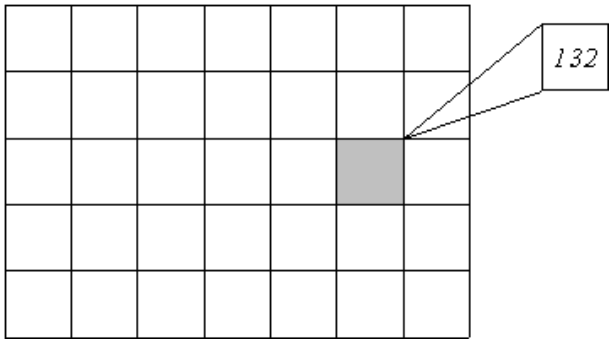


FIGURE B.2: A grayscale image.

B.2 Masks

Masks are commonly used to filter images and perform some type of transformation of image pixels according to the neighbouring pixels. Masks are most commonly presented as a two dimensional matrix with a defined origin. Generally, the values of each pixel in the mask are used to multiply the value of the image pixel it is placed over. A typical example of a mask is a 3×3 averaging filter shown in Figure 2.8 in Chapter 2.

B.3 Convolution

Convolution of an image with a mask refers to the process of taking a filter mask and moving the origin its origin over every pixel of the image. At each step, the values of the mask are multiplied with the corresponding image pixel and then the sum of all output values is placed in the pixel the origin is placed over. Figure B.3 demonstrates a typical convolution process with the averaging filter in Figure 2.8 in Chapter 2.

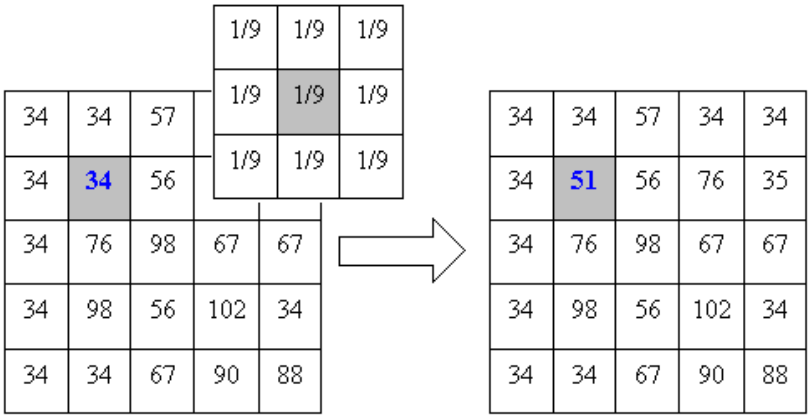


FIGURE B.3: A simple convolution process.

Appendix C

Vision System Access

The vision system is able to be access remotely so that the internal electronics of the AUV does not need to be removed when testing. A wireless access point device provides the wireless network access to the vision microprocessor. For configuration information, see Appendix A, Section A.1.

C.1 Wireless Hardware

To communicate with the access point, a wireless PCI desktop, or PCMIA laptop card is required. The TCP/IP settings of the wireless card must then be configured to communicate with the vision microprocessor.

- IP address: 130.95.149.x
- Subnet mask: 255.255.255.0
- Default gateway: 130.95.149.1

The vision microprocessor has the following TCP/IP settings;

- IP address: 130.95.149.84
- Subnet mask: 255.255.255.0

C.2 Problems

During testing of the wireless access, it was noticed that when using certain cables that were plugged into the serial port, occasionally interfered with wireless communications to the vision microprocessor. This problem could not be resolved, so

in cases where the vision system cannot be contacted, the serial cable should be removed.

Appendix D

Image Processing Library

The majority of this project was dedicated to the development of software for the vision system. This appendix outlines the structures and functions currently provided by the software.

D.1 picture.h

This code contains definitions of classes used in the image processing library file.

- C_BW_Picture - used for black and white images.
- C_RGB_Picture - used for colour images.
- C_Structuring_Element - used for definition of a structuring element for morphological operations.

D.1.1 C_BW_Picture

```
/*********************************************************
 *                                                       *
 *     Width   =  Image Array: Width                     *
 *     Height  =  Image Array: Height                    *
 *     Image   =  Image Array                            *
 *                                                       *
 *     Use setImage(), getImage() to set or get a        *
 *     a pointer to the image array or createImage() to  *
 *     create memory for the image.  Use setPixel(),     *
 *     getPixel() to modify or observe pixel intensity   *
 *                                                       *
 *********************************************************/
```

```
class C_BW_Picture
{
  int Width;
  int Height;
  unsigned char *Image;

  public: C_BW_Picture(void);
  public: void setWidth(int aWidth);
  public: void setHeight(int aHeight);
  public: void setImage(unsigned char *aImage);
  public: void setPixel(int Row, int Col, int Value);
  public: int getWidth(void);
  public: int getHeight(void);
  public: unsigned char *getImage(void);
  public: int getPixel(int Row, int Col);
  public: void createImage(void);
  public: void drawLine(int x1, int x2, int y1, int y2, int Gray);
};
```

D.1.2 C_RGB_Picture

```
/*********************************************************
 *                                                       *
 *     Width   =  Image Array: Width                     *
 *     Height  =  Image Array: Height                    *
 *     Image   =  Image Array                            *
 *                                                       *
 *     Use setImage(), getImage() to set or get a        *
 *     a pointer to the image array or createImage() to  *
 *     create memory for the image.  Use setPixel(),     *
 *     getPixel() to modify or observe pixel intensity   *
 *                                                       *
 *********************************************************/

class C_RGB_Picture
{
  int Width;
  int Height;
  unsigned char *Image;

  public: C_RGB_Picture(void);
  public: void setWidth(int aWidth);
  public: void setHeight(int aHeight);
  public: void setImage(unsigned char *aImage);
  public: void setPixel(int Row, int Col, int Channel, int Value);
```

```
  public: int getWidth(void);
  public: int getHeight(void);
  public: unsigned char *getImage(void);
  public: int getPixel(int Row, int Col, int Channel);
  public: void createImage(void);
  public: void drawLine(int x1, int x2, int y1, int y1,
                        int Red, int Green, int Blue);
};
```

D.1.3 C_Structuring_Element

```
/*********************************************************
 *                                                       *
 *     Width   =  Element Array: Width                   *
 *     Height  =  Element Array: Height                  *
 *     ORow    =  Origin Location: Row                   *
 *     OCol    =  Origin Location: Column                *
 *     Element =  Structuring Element Array              *
 *                                                       *
 *     To use this allocate an array of the smallest     *
 *     possible size to contain the structuring          *
 *     element.  Use setPixel() to allocate pixel as     *
 *     part of the structuring element.                  *
 *                                                       *
 *********************************************************/

class C_Structuring_Element
{
  int Width;
  int Height;
  int ORow;
  int OCol;
  unsigned char *Element;

  public: C_Structuring_Element(int Width, int Height, int ORow, int OCol);
  public: void setPixel(int Row, int Col);
  public: void clearPixel(int Row, int Col);
  public: int getWidth(void);
  public: int getHeight(void);
  public: int getORow(void);
  public: int getOCol(void);
  public: int getPixel(int Row, int Col);
};
```

D.2 image.h

This code provides a number of image processing functions, some of which are described in Chapter 2.

```
/*********************************************************
 *                                                       *
 *     Filename: image.h                                 *
 *                                                       *
 *     Image Processing Library IPL v1.0                 *
 *                                                       *
 *     Original Author: Daniel Lim                       *
 *     Date: 21 March 2004                               *
 *                                                       *
 *     Description: IPL Header File                      *
 *                                                       *
 *********************************************************/

#include <stdlib.h>
#include "picture.h"

/*********************************************************
 *                                                       *
 *     typedef struct Lines                              *
 *                                                       *
 *     Size      =  Number of line equations             *
 *     Equation  =  2D array containing rho and theta    *
 *                  parameters, where;                   *
 *                  Equation[Size][0] = rho              *
 *                  Equation[Size][1] = theta            *
 *                                                       *
 *     Structure for storing line equations produced     *
 *     by the Hough Transform                            *
 *                                                       *
 *********************************************************/

typedef struct
{
int Size;
double **Equations;
}
Lines;

/*********************************************************
```

```
 *                                                          *
 *     readppm()                                            *
 *     Reads in a ppm image and places it in a              *
 *     C_RGB_Picture object                                 *
 *                                                          *
 *     INPUTS:                                              *
 *                                                          *
 *     aFilename  =  Filename of ppm image                  *
 *                                                          *
 *                                                          *
 *     OUTPUTS:                                             *
 *                                                          *
 *     C_RGB_Picture  =  Colour image object                *
 *                                                          *
 ************************************************************/

extern C_RGB_Picture readppm(const char *aFilename);

/************************************************************
 *                                                          *
 *     readpgm()                                            *
 *     Reads in a pgm image and places it in a              *
 *     C_BW_Picture object                                  *
 *                                                          *
 *     INPUTS:                                              *
 *     aFilename  =  Filename of pgm image                  *
 *                                                          *
 *     OUTPUTS:                                             *
 *     C_BW_Picture  =  Colour image object                 *
 *                                                          *
 ************************************************************/

extern C_BW_Picture readpgm(const char *aFilename);

/************************************************************
 *                                                          *
 *     writeppm()                                           *
 *     Writes an C_RGB_Picture object to a ppm file         *
 *                                                          *
 *     INPUTS:                                              *
 *     aFilename  =  Filename of ppm image no ext.          *
 *     aPicture   =  Colour image to output                 *
 *     aComment   =  Comment to place in ppm                *
 *                                                          *
 *     OUTPUTS:                                             *
```

```
 *     None                                                  *
 *                                                           *
 ************************************************************/

extern void writeppm(const char *aFilename, C_RGB_Picture aPicture,
                     const char *aComment);

/************************************************************
 *                                                           *
 *     writepgm()                                            *
 *     Writes an C_BW_Picture object to a pgm file           *
 *                                                           *
 *     INPUTS:                                               *
 *     aFilename  =  Filename of pgm image no ext.           *
 *     aPicture   =  grayscale image to output               *
 *     aComment   =  Comment to place in pgm                 *
 *                                                           *
 *     OUTPUTS:                                              *
 *     None                                                  *
 *                                                           *
 ************************************************************/

extern void writepgm(const char *aFilename, C_BW_Picture aPicture,
                     const char *aComment);

/************************************************************
 *                                                           *
 *     ones()                                                *
 *     Fills a rectangular area in a binary image with       *
 *     ones                                                  *
 *                                                           *
 *     INPUTS:                                               *
 *     Row1      =  Top left corner (row)                    *
 *     Row2      =  Bottom right corner (row)                *
 *     Col1      =  Top left corner (column)                 *
 *     Col2      =  Bottom right corner (column)             *
 *     aPicture  =  Binary image                             *
 *                                                           *
 *     OUTPUTS:                                              *
 *     None                                                  *
 *                                                           *
 ************************************************************/

extern void ones(int Row1, int Row2, int Col1, int Col2,
                 C_BW_Picture aPicture);
```

```
/*********************************************************
 *                                                       *
 *     zeros()                                           *
 *     Fills a rectangular area in a binary image with   *
 *     zeros                                             *
 *                                                       *
 *     INPUTS:                                           *
 *     Row1      =  Top left corner (row)                *
 *     Row2      =  Bottom right corner (row)            *
 *     Col1      =  Top left corner (column)             *
 *     Col2      =  Bottom right corner (column)         *
 *     aPicture  =  Binary image                         *
 *                                                       *
 *     OUTPUTS:                                          *
 *     None                                              *
 *                                                       *
 *********************************************************/

extern void zeros(int Row1, int Row2, int Col1, int Col2,
                  C_BW_Picture aPicture);

/*********************************************************
 *                                                       *
 *     rgb2gray()                                        *
 *     Converts RGB image to a grayscale image           *
 *                                                       *
 *     INPUTS:                                           *
 *     aPicture  =  RGB image to convert                 *
 *                                                       *
 *     OUTPUTS:                                          *
 *     C_BW_Picture  =  Grayscale output object          *
 *                                                       *
 *********************************************************/

extern C_BW_Picture rgb2gray(C_RGB_Picture aPicture);

/*********************************************************
 *                                                       *
 *     gray2rgb()                                        *
 *     Converts grayscale image to an RGB image          *
 *                                                       *
 *     INPUTS:                                           *
 *     aPicture  =  Grayscale image to convert           *
 *                                                       *
```

```
 *     OUTPUTS:                                                *
 *     C_RGB_Picture  =  RGB output object                     *
 *                                                             *
 ***************************************************************/

extern C_RGB_Picture gray2rgb(C_BW_Picture aPicture);

/***************************************************************
 *                                                             *
 *     bin2gray()                                              *
 *     Converts a binary image to a grayscale image            *
 *                                                             *
 *     INPUTS:                                                 *
 *     aPicture  =  Binary image to convert                    *
 *                                                             *
 *     OUTPUTS:                                                *
 *     C_BW_Picture  =  Grayscale output object                *
 *                                                             *
 ***************************************************************/

extern C_BW_Picture bin2gray(C_BW_Picture aPicture);

/***************************************************************
 *                                                             *
 *     label2gray()                                            *
 *     Converts a labeled image to a grayscale image           *
 *                                                             *
 *     INPUTS:                                                 *
 *     aPicture  =  Labeled image to convert                   *
 *                                                             *
 *     OUTPUTS:                                                *
 *     C_BW_Picture  =  Grayscale output object                *
 *                                                             *
 ***************************************************************/

extern C_BW_Picture label2gray(int ObjectCount, C_BW_Picture aPicture);

/***************************************************************
 *                                                             *
 *     invert()                                                *
 *     Inverts a binary iamge                                  *
 *                                                             *
 *     INPUTS:                                                 *
 *     aPicture  =  Binary image to invert                     *
 *                                                             *
```

```
 *     OUTPUTS:                                                *
 *     C_BW_Picture  =  Binary output object                   *
 *                                                             *
 ************************************************************/

extern C_BW_Picture invert(C_BW_Picture aPicture);

/************************************************************
 *                                                           *
 *     rgb_channelthreshold()                                *
 *     Binary threshold a colour image over a                *
 *     particular colour channel using a value or range      *
 *                                                           *
 *     INPUTS:                                               *
 *     Value     =  Threshold value                          *
 *     Value1    =  Lower threshold range                    *
 *     Value2    =  Upper threshold range                    *
 *     Channel   =  Colour channel                           *
 *     aPicture  =  Colour image                             *
 *                                                           *
 *     OUTPUTS:                                              *
 *     C_BW_Picture  =  Binary output object                 *
 *                                                           *
 ************************************************************/

extern C_BW_Picture rgb_channelthreshold(int Value, int Channel,
                                         C_RGB_Picture aPicture);

extern C_BW_Picture rgb_channelthreshold(int Value1, int Value2,
                                         int Channel,
                                         C_RGB_Picture aPicture);

/************************************************************
 *                                                           *
 *     rgb_threshold()                                       *
 *     Binary threshold a colour image over all              *
 *     colour channels                                       *
 *                                                           *
 *     INPUTS:                                               *
 *     aRed      =  Red threshold                            *
 *     aGreen    =  Green threshold                          *
 *     aBlue     =  Blue treshold                            *
 *     Channel   =  Channel of interest                      *
 *     aPicture  =  Colour image                             *
 *                                                           *
```

```
 *     OUTPUTS:                                              *
 *     C_BW_Picture  =  Binary output object                 *
 *                                                           *
 ***********************************************************/

extern C_BW_Picture rgb_threshold(int aRed, int aGreen, int aBlue,
                                  int Channel, C_RGB_Picture aPicture);

/***********************************************************
 *                                                         *
 *     bw_channelthreshold()                               *
 *     Binary threshold a grayscale image using a          *
 *     value or range                                      *
 *                                                         *
 *     INPUTS:                                             *
 *     Value     =  Threshold value                        *
 *     Value1    =  Lower threshold range                  *
 *     Value2    =  Upper threshold range                  *
 *     aPicture  =  Colour image                           *
 *                                                         *
 *     OUTPUTS:                                            *
 *     C_BW_Picture  =  Binary output object               *
 *                                                         *
 ***********************************************************/

extern C_BW_Picture bw_threshold(int Value,
                                 C_BW_Picture aPicture);

extern C_BW_Picture bw_threshold(int Value1, int Value2,
                                 C_BW_Picture aPicture);

 /***********************************************************
 *                                                         *
 *     median()                                            *
 *     Perform median filtering of a grayscale image       *
 *     INPUTS:                                             *
 *     iSize     =  N x N size of the median filter        *
 *     aPicture  =  Grayscale image                        *
 *                                                         *
 *     OUTPUTS:                                            *
 *     C_BW_Picture  =  Grayscale output object            *
 *                                                         *
 ***********************************************************/

extern C_BW_Picture median(C_BW_Picture aPicture);
```

```
/*********************************************************
 *                                                       *
 *     average()                                         *
 *     Perform average filtering of a grayscale image    *
 *                                                       *
 *     INPUTS:                                           *
 *     iSize     =  N x N size of the averaging filter   *
 *     aPicture  =  Grayscale image                      *
 *                                                       *
 *     OUTPUTS:                                          *
 *     C_BW_Picture  =  Grayscale output object          *
 *                                                       *
 *********************************************************/

extern C_BW_Picture average(C_BW_Picture aPicture);

/*********************************************************
 *                                                       *
 *     gaussian()                                        *
 *     Perform Gaussian filtering of a grayscale image   *
 *                                                       *
 *     INPUTS:                                           *
 *     sDev      =  Standard deviation of Gaussian       *
 *     iSize     =  N x N size of the Gaussian filter    *
 *     aPicture  =  Grayscale image                      *
 *                                                       *
 *     OUTPUTS:                                          *
 *     C_BW_Picture  =  Grayscale output object          *
 *                                                       *
 *********************************************************/

extern C_BW_Picture gaussian(double sDev, int iSize,
                             C_BW_Picture aPicture);

/*********************************************************
 *                                                       *
 *     erode()                                           *
 *     Performs erosion on a binary image                *
 *                                                       *
 *     INPUTS:                                           *
 *     aElement  =  Structuring element                  *
 *     aPicture  =  Binary image                         *
 *                                                       *
 *     OUTPUTS:                                          *
```

```
 *     C_BW_Picture  =  Binary output object             *
 *                                                       *
 *******************************************************/

extern C_BW_Picture erode(C_Structuring_Element aElement,
                          C_BW_Picture aPicture);

/*******************************************************
 *                                                     *
 *     dilate()                                        *
 *     Performs dilation on a binary image             *
 *                                                     *
 *     INPUTS:                                         *
 *     aElement  =  Structuring element                *
 *     aPicture  =  Binary image                       *
 *                                                     *
 *     OUTPUTS:                                        *
 *     C_BW_Picture  =  Binary output object           *
 *                                                     *
 *******************************************************/

extern C_BW_Picture dilate(C_Structuring_Element aElement,
                           C_BW_Picture aPicture);

/*******************************************************
 *                                                     *
 *     open()                                          *
 *     Performs an opening of a binary image           *
 *                                                     *
 *     INPUTS:                                         *
 *     aElement  =  Structuring element                *
 *     aPicture  =  Binary image                       *
 *                                                     *
 *     OUTPUTS:                                        *
 *     C_BW_Picture  =  Binary output object           *
 *                                                     *
 *******************************************************/

extern C_BW_Picture open(C_Structuring_Element aElement,
                         C_BW_Picture aPicture);

/*******************************************************
 *                                                     *
 *     close()                                         *
 *     Performs a closing of a binary image            *
```

```
 *     INPUTS:                                                *
 *     aElement  =  Structuring element                       *
 *     aPicture  =  Binary image                              *
 *                                                            *
 *     OUTPUTS:                                               *
 *     C_BW_Picture  =  Binary output object                  *
 *                                                            *
 ************************************************************/

extern C_BW_Picture close(C_Structuring_Element aElement,
                          C_BW_Picture aPicture);

/************************************************************
 *                                                          *
 *     canny()                                              *
 *     Performs canny edge detection on a grayscale         *
 *     image                                                *
 *                                                          *
 *     INPUTS:                                              *
 *     sDev        =  Standard deviation of Gaussian        *
 *     iSize       =  N x N size of the Gaussian filter     *
 *     ThreshHigh  =  Upper edge threshold                  *
 *     ThreshLow   =  Lower edge threshold                  *
 *     aPicture    =  Grayscale image                       *
 *                                                          *
 *     OUTPUTS:                                             *
 *     C_BW_Picture  =  Binary edge map output object       *
 *                                                          *
 ************************************************************/

extern C_BW_Picture canny(double sDev, int iSize, int ThreshHigh,
                          int ThreshLow, C_BW_Picture aPicture);

/************************************************************
 *                                                          *
 *     houghlines()                                         *
 *     Extracts most significant straight lines from        *
 *     image                                                *
 *                                                          *
 *     INPUTS:                                              *
 *     HoughLines      =  Equation storage structure        *
 *     SE              =  Structuring Element               *
 *     ThreshHough     =  Accumulator threshold             *
 *     rhoIntervals    =  No. rho intervals                 *
 *     thetaIntervals  =  No. theta intervals               *
```

```
 *     sDev            =  Standard deviation of          *
 *                        Gaussian                       *
 *     iSize           =  N x N size of the Gaussian     *
 *                        filter                         *
 *     ThreshHigh      =  Upper edge threshold           *
 *     ThreshLow       =  Lower edge threshold           *
 *     aPicture        =  Grayscale image                *
 *                                                       *
 *     OUTPUTS:                                          *
 *     None                                              *
 *                                                       *
 *******************************************************/

 extern void houghlines(Lines *HoughLines, C_Structuring_Element SE,
                        int TreshHough, int rhoIntervals, int thetaIntervals,
                        double sDev, int iSize, int ThreshHigh, int ThreshLow,
                        C_BW_Picture aPicture);

/*******************************************************
 *     DO NOT USE THIS! CONTAINS BUGS                  *
 *     sequential_label()                              *
 *     Uses sequential algorithm to label binary image *
 *                                                     *
 *     INPUTS:                                         *
 *     ObjectCount  =  Stores count of distinct objects*
 *     aPicture     =  Binary image to be labeled      *
 *                                                     *
 *     OUTPUTS:                                        *
 *     C_BW_Picture  =  Labeled image output object    *
 *                                                     *
 *******************************************************/

extern C_BW_Picture sequential_label(int *ObjectCount, C_BW_Picture aPicture);

/*******************************************************
 *                                                     *
 *     recursive_label()                               *
 *     Uses recursive algorithm to label binary image  *
 *                                                     *
 *     INPUTS:                                         *
 *     ObjectCount  =  Stores count of distinct objects*
 *     aPicture     =  Binary image to be labeled      *
 *                                                     *
 *     OUTPUTS:                                        *
 *     C_BW_Picture  =  Labeled image output object    *
```

```
 *                                                          *
 ************************************************************/

extern C_BW_Picture recursive_label(int *ObjectCount, C_BW_Picture aPicture);

/************************************************************
 *                                                          *
 *     labelNeighbours()                                    *
 *     Recursive function for recursive labeling            *
 *                                                          *
 *     INPUTS:                                              *
 *     Row         =  Row of pixel to be labeled            *
 *     Col         =  Column of pixel to be labeled         *
 *     inPicture   =  Binary image                          *
 *     outPicture  =  Labeled image                         *
 *                                                          *
 *     OUTPUTS:                                             *
 *     None                                                 *
 *                                                          *
 ************************************************************/

extern void labelNeighbours(int Row, int Col, int Label,
                            C_BW_Picture inPicture,
                            C_BW_Picture outPicture);

/************************************************************
 *                                                          *
 *     getobject()                                          *
 *     Produces binary image of a distinct object           *
 *     within a labeled image                               *
 *                                                          *
 *     INPUTS:                                              *
 *     ObjectLabel  =  Label no. of object of interest      *
 *     aPicture     =  Labeled image                        *
 *                                                          *
 *     OUTPUTS:                                             *
 *     C_BW_Picture  =  Binary image output object          *
 *                                                          *
 ************************************************************/

extern C_BW_Picture getobject(int ObjectLabel, C_BW_Picture aPicture);

/************************************************************
 *                                                          *
 *     centroid()                                           *
```

```
 *     Calculates the centre of mass of a binary image   *
 *                                                       *
 *     INPUTS:                                           *
 *     meanRow   =  Stores centre of mass y co-ordinate  *
 *     meanCol   =  Stores centre of mass x co-ordinate  *
 *     aPicture  =  Binary image                         *
 *                                                       *
 *     OUTPUTS:                                          *
 *     None                                              *
 *                                                       *
 *******************************************************/

extern void centroid(double *ameanRow, double *ameanCol, C_BW_Picture aPicture);

/*******************************************************
 *                                                     *
 *     orientation()                                   *
 *     Calculates the angle of the axis of minimum     *
 *     inertia                                         *
 *                                                     *
 *     INPUTS:                                         *
 *     meanRow   =  Centre of mass y co-ordinate       *
 *     meanCol   =  Centre of mass x co-ordinate       *
 *     aPicture  =  Binary image                       *
 *                                                     *
 *     OUTPUTS:                                        *
 *     double  =  Angle (radians) of the AMI           *
 *                                                     *
 *******************************************************/

extern double orientation(double meanRow, double meanCol, C_BW_Picture aPicture);

/*******************************************************
 *                                                     *
 *     area()                                          *
 *     Calculates the area of object pixels in a binary  *
 *     image                                           *
 *                                                     *
 *     INPUTS:                                         *
 *     aPicture  =  Binary image                       *
 *                                                     *
 *     OUTPUTS:                                        *
 *     int  =  area                                    *
 *                                                     *
 *******************************************************/
```

```
extern int area(C_BW_Picture aPicture);
```

References

[1] H.H. Wang, S.M. Rock, and M.J. Lees. Experiments in automatic retrieval of underwater objects with an auv. In *OCEANS '95. MTS/IEEE. 'Challenges of Our Changing Global Environment'. Conference Proceedings.*, volume 1, pages 366–373, 1995.

[2] H.H. Pien, D.E. Gustafson, and W.F. Bonnice. An auv vision system for target detection and precise positioning. In *Autonomous Underwater Vehicle Technology, 1994. AUV '94., Proceedings of the 1994 Symposium on*, pages 36–43, 1994.

[3] T.P. Pridmore and W.M.M. Hales. Understanding images: an approach to the university teaching of computer vision. *Engineering Science and Education Journal*, 4:161–166, 1995.

[4] J. Santos-Victor and J. Sentieiro. The role of vision for underwater vehicles. In *Autonomous Underwater Vehicle Technology, 1994. AUV '94., Proceedings of the 1994 Symposium on*, pages 28–25, 1994.

[5] A. Ortiz, M. Simo, and G. Oliver. Image sequence analysis for real-time underwater cable tracking. In *Applications of Computer Vision, 2000, Fifth IEEE Workshop on*, pages 230–236, 2000.

[6] A. Balasuriya and T. Ura. Vision-based underwater cable detection and following using auvs. In *Oceans '02 MTS/IEEE*, volume 3, pages 1582–1587, 2002.

[7] Norsk Elektro Optikk AS. Underwater pipeline inspection. [Online], 2004. Available: `http://www.neo.no/research/pipeline/xplisit.html`.

[8] R. Owens. Discrete binary images. [Online], 1997. Available: `http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT2/node3.html`.

[9] M.M. Skolnick, R.H. Brown, C. Bhagvati, and B.R. Wolf. Morphological algorithms for centroid normalization in relational matching. In *Circuits and Systems, 1989., IEEE International Symposium on*, volume 2, pages 987–990, 1989.

[10] V. Khanna, P. Gupta, and C.J. Hwang. Finding connected components in digital images. In *Information Technology: Coding and Computing, 2001. Proceedings. International Conference on*, pages 652–656, 2001.

[11] C.J. Vienna and E.A. De Souza. Combining marr and canny operators to detect edges. In *Microwave and Optoelectronics Conference, 2003. IMOC 2003. Proceedings of the 2003 SBMO/IEEE MTT-S International*, volume 2, pages 695–699, 2003.

[12] P.J. Terry and D. Vu. Edge detection using neural networks. In *Signals, Systems and Computers, 1993. 1993 Conference Record of The Twenty-Seventh Asilomar Conference on*, volume 3, pages 391–395, 1993.

[13] R. Owens. The hough transform. [Online], 1997. Available: `http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT6/node3.html#SECTION00030000000000000000`.

[14] K. Alexiev. Implementation of hough transform as track detector. In *Information Fusion, 2000. FUSION 2000. Proceedings of the Third International Conference on*, volume 2, pages THC4/11–THC4/16, 2000.

[15] G. Knorr. Video4linux. [Online]. Available: `http://linux.bytesex.org/v4l2/`.

[16] P. Reynolds. The cqcam home page. [Online]. Available: `http://www.cs.duke.edu/~reynolds/cqcam/`.

[17] ONTRACK Control Systems. Using linux to send commands to adr interfaces. [Online], 2002. Available: `http://www.ontrak.net/linux.htm`.

[18] E. Rosse. Design and control of an autonomous underwater vehicle. Final Year Project Thesis. School of Electrical, Electronic and Computer Engineering, The University of Western Australia, WA, Australia, 2004.

[19] X. Yuan, C. Qiu, R. Chen, Z. Hu, and P. Liu. Vision system research for autonomous underwater vehicle. In *Intelligent Processing Systems, 1997. ICIPS '97. 1997 IEEE International Conference on*, volume 2, pages 1465–1469, 1997.

[20] L. Gonzalez. Design, modelling and control of an autonomous underwater vehicle. Final Year Project Thesis. School of Electrical, Electronic and Computer Engineering, The University of Western Australia, WA, Australia, 2004.

[21] B. Green. Canny edge detection tutorial. [Online], 2002. Available: `http://www.pages.drexel.edu/~weg22/can_tut.html`.

[22] N. Kanopoulos, N. Vasanthavada, and R.L. Baker. Design of an image edge detection filter using the sobel operator. In *Solid-State Circuits, IEEE Journal of*, volume 23, pages 358–367, 1998.

[23] C. Patrick. Wafer-5820-r3. [Online], 2000. Available: `http://dagobah.ucc.asn.au/wizard/cp.yi.org-mirror/wafer-5820.pdf`.

[24] T. Bräunl. Eyebot mk3 features. [Online], 2004. Available: `http://robotics.ee.uwa.edu.au/eyebot/index.html`.