

# Image Processing Modules Implemented, Tested and Embedded in GIS tools

David Hng

*This report is submitted as partial fulfilment  
of the requirements for the Honours Programme of the  
School of Computer Science and Software Engineering,  
The University of Western Australia,  
2007*

# Abstract

A Geographic Information System (GIS) is a software suite which provides a unified system for managing spatially referenced data. These systems deliver information to an end user through powerful visualisation and analysis tools which can be used to answer complex spatial queries. Current GIS software, such as ArcGIS, include a well developed suite of vector analysis tools which can be applied to geographic data, but these tools have only a limited ability to analyse raster data. Image processing algorithms provide an effective way to enhance raster data or to detect features of interest in raster data.

The aim of this project is to develop an Image Processing Framework where user-developed image processing modules can be embedded within a desktop GIS tool, namely ArcGIS. This framework allows a GIS user to apply image processing techniques to rasters to extract relevant features. These image processing techniques are packaged into individual modules which are loaded into the framework using a plug-in architecture. This system can also be extended by writing additional plug-in modules.

A novel software methodology was proposed for this project. The software development process was viewed as a means of acquiring knowledge about the systems that needed to interact, and codifying this knowledge. In order to do this, a special test harness was created which unifies concepts from example code, unit testing, and documentation. This test harness allows a developer to create Code Experiments: short segments of code which demonstrate a particular aspect of the ArcGIS API. These segments of code are used to document and demonstrate the interactions between the Image Processing Framework and the ArcGIS API.

This project successfully demonstrates linear feature detection from aeromagnetic datasets of the Yilgarn Craton in Western Australia.

**Keywords:** Geographic Information Systems, Image Processing, Software Engineering, Example Code, Documentation, Unit Testing

**CR Categories:** D.2.4, F.3.1, D.2.7, I.4.6, J.2

# Acknowledgements

I would especially like to thank my supervisors Eun-Jung Holden and Rachel Cardell-Oliver for their support and guidance throughout the year. Undertaking a significant amount of work such as a final year engineering dissertation is no easy task and I thank them for their patience.

I would also like to thank Professor Mike Dentith for his input on aeromagnetic data analysis.

The aeromagnetic data of the Yilgarn Craton used in this dissertation is property of Fugro Airborne Surveys Pty Ltd and I would like to thank them for their permission for use of this data.

Last, but not least, I would also like to thank my family and close friends for providing support and advice throughout the year.

# Contents

<b>Abstract</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 The Aim . . . . .	2
<b>2 System Overview</b>	<b>4</b>
<b>3 Image Processing Framework</b>	<b>6</b>
3.1 Plug-in system . . . . .	6
3.2 Background: Linear Feature Detection and Extraction . . . . .	6
3.2.1 Hough Transform . . . . .	7
3.2.2 Radon Transform . . . . .	8
3.2.3 Comparison . . . . .	9
3.2.4 Optimisations for Geological Data . . . . .	10
3.3 Method . . . . .	14
3.3.1 Proposed Approach . . . . .	14
3.3.2 Modules Implemented . . . . .	14
<b>4 Unifying Example Code, Unit Tests and Documentation</b>	<b>21</b>
4.1 Problem . . . . .	21
4.2 Example Code . . . . .	22
4.3 Background: Addressing Problems in Example Code . . . . .	22
4.3.1 Verification Problem . . . . .	23
4.3.2 Storage Problem . . . . .	26
4.3.3 Useful Documentation . . . . .	27
4.4 The Process of Code Experiments . . . . .	29

4.5	Method . . . . .	30
4.5.1	Tools for Experiment Support . . . . .	30
4.5.2	Reusing Existing Testing Tools . . . . .	32
4.5.3	A Custom Test Harness . . . . .	33
4.6	Experiment Environment: ArcMapUnit . . . . .	34
<b>5</b>	<b>Results and Evaluation</b>	<b>37</b>
5.1	Code Metrics . . . . .	37
5.2	Linear feature extraction process . . . . .	38
5.2.1	Demonstration . . . . .	38
5.2.2	Performance Improvements . . . . .	39
5.3	Code Experiments . . . . .	43
5.3.1	Verification . . . . .	43
5.3.2	Storage . . . . .	43
5.3.3	Maintenance . . . . .	43
5.3.4	Discussion . . . . .	44
5.3.5	Experiments created . . . . .	45
<b>6</b>	<b>Summary and Future Development</b>	<b>47</b>
6.1	Contributions . . . . .	47
6.2	Future Work . . . . .	48
<b>A</b>	<b>Original Honours Proposal</b>	<b>49</b>
	<b>Bibliography</b>	<b>54</b>

# List of Figures

2.1	Interactions between subsystems . . . . .	4
2.2	Use cases for the image processing framework . . . . .	5
3.1	Image Processing Framework User Interface . . . . .	7
3.2	The Radon Transform . . . . .	8
3.3	Biases in the Radon Space . . . . .	11
3.4	Sobel Convolution Masks . . . . .	15
3.5	Nearest neighbour approximation to a line. . . . .	16
3.6	Effects of using the Mean Gradient technique. . . . .	17
3.7	Effects of Windowing . . . . .	18
3.8	Effects of Non-Maximal Suppression . . . . .	19
4.1	An example Test Fixture in C# using NUnit . . . . .	25
4.2	Standard xUnit Test Cycle . . . . .	28
4.3	ArcMapUnit User Interface . . . . .	31
4.4	ArcMap hosting the NUnit application within its process . . . . .	32
4.5	NUnit hosting the ArcMap application within its process . . . . .	33
4.6	ArcMapUnit: Executing experiments across a process boundary . . . . .	34
5.1	Aeromagnetic data of the Yilgarn Craton in Western Australia. . . . .	38
5.2	Sobel edge detection applied to the original raster. . . . .	39
5.3	Results of image processing as a vector layer in ArcMap. . . . .	40
5.4	Radon Transform Performance . . . . .	42
5.5	An experiment using verification. . . . .	44

# List of Tables

5.1	Code Metrics for software authored in this project. . . . .	37
5.2	Performance Analysis of a typical user workflow . . . . .	41
5.3	Range of input images . . . . .	42

## CHAPTER 1

# Introduction

A Geographic Information System (GIS) is a system that manages and visualises data that is spatially referenced to the earth [21]. It allows the integration of various geoscientific information from multiple sources by providing an environment that can create and perform interactive queries and visualise the results of these operations. Geoscientific data is obtained by sampling phenomena in the real world and is either stored in raster or vector format. For example, geologists generate geological maps that illustrate the distribution of geological entities such as rock types which are then digitised into groups of shapes (vectors) or stored as gridded data (rasters).

The choice of datasets and the form of data analysis performed depends on the intended application of the results. Environmental studies, for example, would benefit from satellite imagery from which a user can extract information on the distribution of water and vegetation. The mining industry on the other hand, analyses geological maps and geophysical data to target locations for exploration of a certain type of mineral. However, the analysis of raster data, regardless of the application, benefits from image processing to enhance or recognise certain features. For example, simple colour detection is useful in identifying areas of vegetation from satellite images. The detection of long linear features in aeromagnetic images can also provide crucial information in identifying a fault or a dyke, which are important indicators for many forms of mineralisation. A fault is a planar fracture in the Earth's crust resulting from shear motion, causing one side of the Earth's crust to be displaced relative to the other and faults are known to be used as conduits for gold mineralising fluids [17]. A dyke is an intrusion of magma into the Earth's crust and is an important indicator for the precipitation of minerals such as uranium [29].

There are two major GIS tools that are commercially available: ArcGIS [10] and MapInfo [24]. These software tools provide sophisticated functionality to view and analyse data spatially, but only provide limited image processing functionality. Thus to perform image processing of raster data, users are required to export data from the GIS tool, load it into another application to process the

data, for example ER Mapper [11], and then import the results back into the GIS tool so that the processed raster data can be integrated with other datasets in the GIS environment. Such processes require careful handling of data to ensure that precision and spatial referencing is not lost through the use of multiple software applications and is further complicated if data conversion is required between application specific data formats. A preferred approach would be to perform image processing operations directly within the existing GIS environment.

## 1.1 The Aim

The aim of this project is to provide a framework where image processing modules can be tested and embedded within an existing GIS environment, specifically ArcGIS. This framework allows image processing modules to be used as ‘plug-ins’ to ArcGIS, adding new functionality to the system.

In developing the image processing framework, a number of considerations need to be met:

- The image processing framework must interact with ArcGIS which is closed-source, proprietary software with limited documentation.
- The utility of the image processing framework needs to be demonstrated.
- The image processing framework must be extensible: other programmers should be able to write image processing modules which can be plugged in to the framework.

Interactions between software systems are complex and lead to uncertainty in the software development process as there may be multiple perceived ways of achieving a task, but only some are actually correct and yield the desired result. This is a problem of information asymmetry: the software developer does not know how to employ their available tools optimally to achieve their desired outcomes.

Documentation is the traditional software asset which is used to overcome this information asymmetry. However, the documentation provided with the ArcGIS Application Programming Interface (API) is not sufficiently detailed to allow integration of an image processing framework into the software. To mitigate this problem, a process deemed *Code Experiments* was investigated and developed. Code Experiments are a hybrid of concepts from example code, unit testing and documentation. Supported by a specialised test harness, Code Experiments

provide a means of acquiring and capturing knowledge regarding how to use the ArcGIS API. The Code Experiment process was used to extensively test and demonstrate portions of the ArcGIS API relevant to the image processing framework, thereby reducing uncertainty in this project's software development.

To allow other developers to extend the image processing framework, a plugin system was designed and incorporated into the software. This design for the image processing framework is based on the Adapter and Factory design patterns [15], and additional image processing modules can be built by implementing a series of software interfaces.

The image processing modules implemented for this project automatically detect linear features within aeromagnetic data, which correspond to faults and dykes, for the application of targeting minerals. In aeromagnetic data, these geological entities appear as dominant linear anomalies, which have a distinct contrast in magnetic intensity from their surroundings. A well known line detection technique, namely either the Radon or the Hough transform [7], is commonly used for the detection of these linear anomalies in geophysical data. However, there are problems associated with the traditional use of the Radon and Hough transforms in that they detect dominant lines without considering the bias towards features that are centred in the image, and that thick linear features can be recognised as multiple lines. In this project, the technique developed by Zhang *et al.* [36] has been extended to deal with scale and line thickness of linear anomalies using an effective non-maximal suppression and a sliding window for analysis.

For this project the following three software components were designed and implemented:

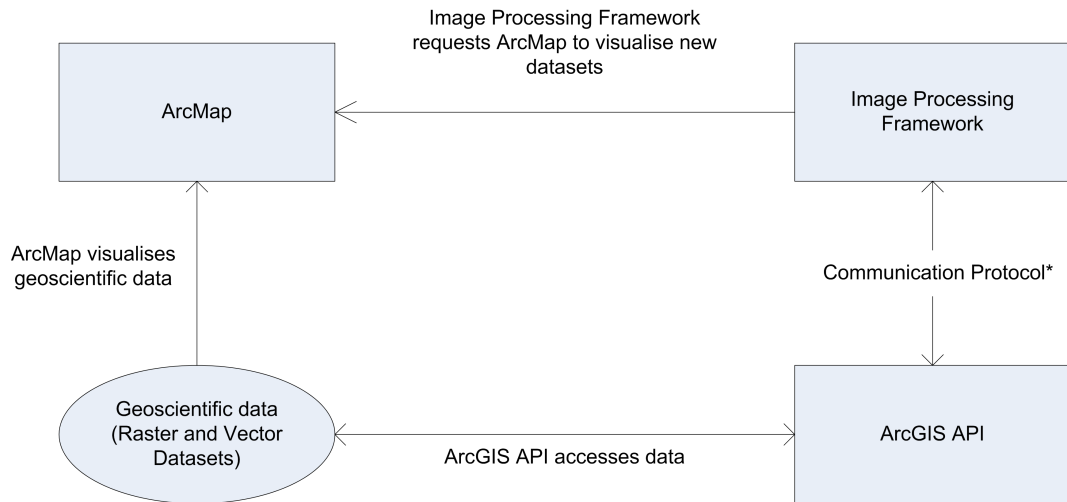
- An *Image Processing Framework* which embeds image processing modules within ArcGIS.
- Example *Image Processing Modules* that find linear features within aeromagnetic datasets.
- *ArcMapUnit*, a test harness for running 'Code Experiments' which were used to support the development of the Image Processing Framework.

In this dissertation, Chapter 2 provides an overview of the software systems developed in this project and highlights their relations. Chapter 3 discusses the Image Processing Framework which was developed for integration with a GIS tool, examining previous work in the area of linear feature detection. Chapter 4 details the software engineering problems encountered in developing the image processing framework and describes the concept of Code Experiments. Chapter 5 reports the results of the systems developed and algorithms investigated.

## CHAPTER 2

# System Overview

The software in this project can be broken down into a number of independent subsystems. The general relationship between these subsystems can be seen in Figure 2.1 below.



\* Image Processing Framework communicates with the ArcGIS API to read rasters and to create new vector and raster datasets. This interaction is documented and tested using code experiments.

Figure 2.1: Interactions between subsystems

ArcMap is part of the ArcGIS suite of software tools and is a visualisation portal for maps and spatial data. ArcMap is developed by Environmental Systems Research Institute (ESRI), and is part of the ArcGIS Desktop suite of GIS tools. It provides basic GIS functions for a standalone desktop user: rasters and vectors stored in a local database can be loaded and visualised.

The Image Processing Framework is a software component which was developed in this project and interacts with a raster source to produce raster or vector

outputs. The Image Processing Framework provides a Graphical User Interface (GUI) for the manipulation of raster data and also hosts image processing modules. At the architecture level, the Image Processing Modules follow a Pipe and Filter pattern [5]. Each image processing module takes an input raster as its source data, and produces a raster or vector layer as its output. The Image Processing Framework and Image Processing Modules were written using C# and the Microsoft .NET Framework.

ArcMap is used to visualise and select which rasters are imported into the Image Processing Framework. The Image Processing Framework reads the selected rasters and allows image processing modules to work on the raster data. The modules subsequently produce output which is either in raster or vector form and the output data is exported from the Image Processing Framework and loaded into ArcMap where it can be visualised.

The process of data exchange between ArcMap and the Image Processing Framework is documented and tested using *ArcMapUnit*, a specialised test harness developed in this project. ArcMapUnit allows short segments of code to be written which experiment with the ArcGIS API, and are called *Code Experiments*. This facility was used to create experiments which demonstrated the necessary interactions between the Image Processing Framework and ArcMap: reading rasters through the ArcGIS API to obtain source data, and saving raster and vector data produced by the Image Processing Framework.

The use cases for the system are shown in Figure 2.2. A geoscientist will be able to process datasets by invoking the Image Processing Framework from within ArcMap. They will be able to select a raster layer which is part of the current map to be analysed. The Image Processing Framework will extract the data from this raster and allow the geoscientist to run image processing modules on the dataset. The result of this processing is a new dataset which can then be immediately visualised on the current map.

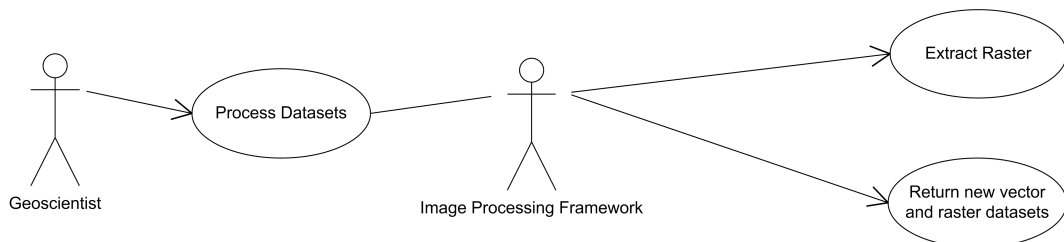


Figure 2.2: Use cases for the image processing framework

## CHAPTER 3

# Image Processing Framework

This chapter provides a brief overview of the plug-in system and then discusses the background theory and algorithms used in linear feature extraction and their application to geoscientific data. It then explains the proposed linear feature detection algorithm that adapts and improves the existing techniques. A short description follows of the other processing modules that were implemented.

### 3.1 Plug-in system

The image processing framework consists of a plug-in system and a set of useful image processing modules that were designed and implemented for the processing of geoscientific data. The plug-in system manages modules and discovers them at runtime using Reflection and dynamic Assembly loading, feature of the Microsoft .NET Framework. Thus, additional image processing modules can be added to the system by copying new modules into the image processing framework's directory.

A screenshot of the image processing framework user interface can be seen in Figure 3.1.

### 3.2 Background: Linear Feature Detection and Extraction

In geoscientific data, linear features often indicate areas of interest. These features are automatically detected using image processing techniques known as the Hough [13] and Radon [33] Transforms. These two algorithms detect lines and other parameterised shapes, and can effectively be applied to geoscientific datasets through several optimisations.

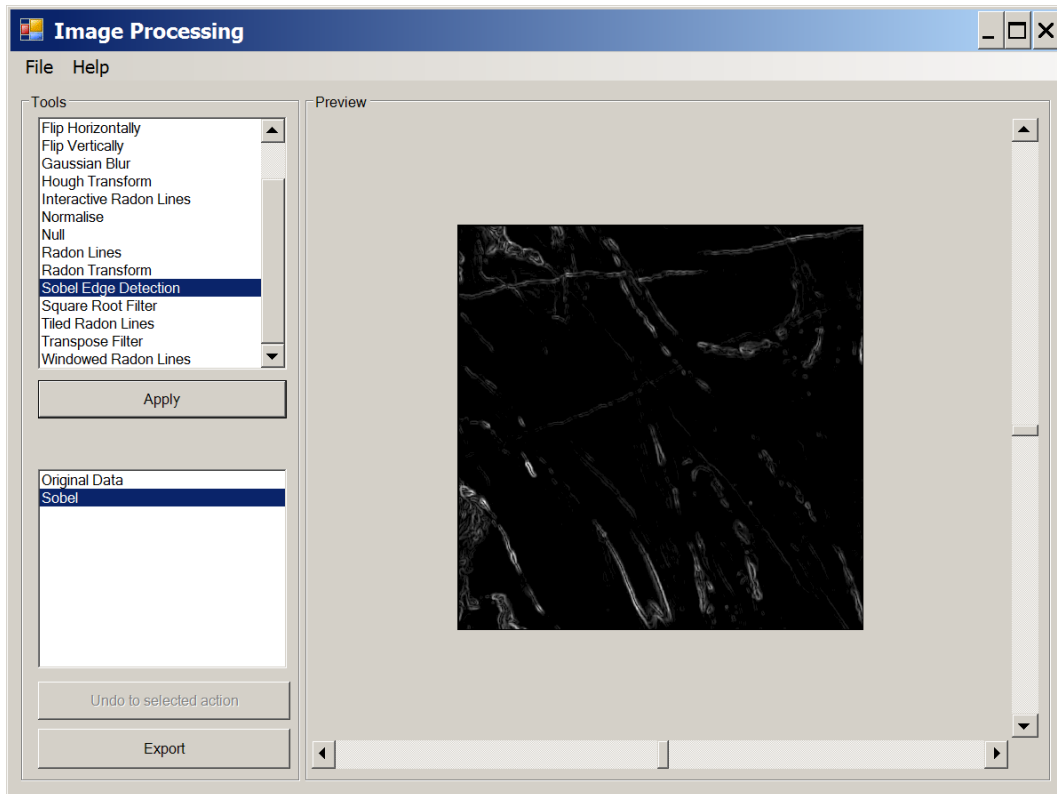


Figure 3.1: Image Processing Framework User Interface

### 3.2.1 Hough Transform

The Hough Transform is an algorithm that detects groups of pixels which reside on the same characteristic line. The transform was originally developed using a Cartesian coordinate system but was subsequently modified for a polar coordinate system. A line can be represented in its normal form with the following equation [7]:

$$x \cos \theta + y \sin \theta = \rho$$

This equation describes a line passing through the point  $(x, y)$  where  $\rho$  is the perpendicular distance from the origin to the line, and  $\theta$  is the angle between the perpendicular distance line to the x axis. Given an input image, the Hough Transform generates a Hough accumulator that stores ‘votes’ for lines that are represented by discretely sampled parameters,  $\rho$  and  $\theta$ . The transform iterates over each pixel in the image, and if the pixel has a foreground pixel value, it computes all possible lines that pass through the pixel and increments the votes within the accumulator for the computed lines.

The method for quantifying each ‘vote’ depends on characteristics of the input image and the type of feature that is being identified, but most often the existence of a foreground pixel in the input array is weighted as a single vote. In the Hough accumulator, more ‘votes’ at a particular coordinate are a stronger indication of a linear feature with specified  $\rho$  and  $\theta$  parameters within the image.

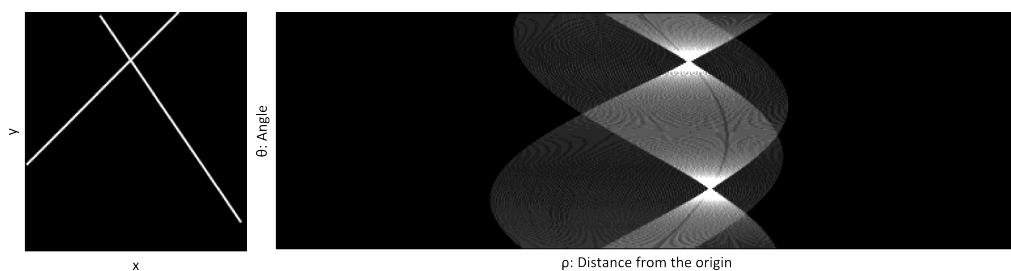
### 3.2.2 Radon Transform

The 2-dimensional Radon Transform is “a mapping of 2D data defined over a rectangular set of coordinates  $(x, y)$  onto a domain defined by the slope and intercept  $(\rho, \tau)$  of lines present in the data” [9]. In the polar coordinate system, the Radon Transform is a function of slope and distance and is expressed as:

$$F(\rho, \theta) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) \delta(\rho - x \cos \theta - y \sin \theta) dx dy$$

In this equation,  $\delta$  is the delta function and restricts the integrand such that only points residing on the line  $\rho = x \cos \theta + y \sin \theta$  are integrated. Thus, the expression calculates the line integral under the data bounded by the input image and stores it in the Radon space at a coordinate  $(\rho, \theta)$ .

The Radon Transform is useful as a line detector because the greatest responses (largest integral values) coincide with the parameters for dominant lines. Figure 3.2 shows an example of the Radon Transform. Each point in the input array has an impact on the regions of a skewed sinusoidal curve traced in the Radon space as shown in the figure. Points coinciding on the same line overlap and maximise the value of the Radon space at the points which most likely represent a line. The two brightest points (whitest areas) in the Radon space coincide with the parameters for the two apparent lines in the original raster.



(a) Raster containing two lines.

(b) Radon Transform of raster.

Figure 3.2: The Radon Transform

The Radon Transform has many applications in geoscience including the identification and enhancement of linear features in images. The Cartesian form of the Radon Transform is also known as the ‘slant-stack’ [31], and is used in seismology as an aid in solving scattering problems. The Radon Transform can also be extended into three or more dimensions for identification of planes and other surfaces.

### 3.2.3 Comparison

The Hough and Radon Transforms are similar in many ways, and with some manipulation it can be seen that the Hough Transform is a discrete form of the Radon Transform [34]. However, the two transforms were initially developed to solve very different problems and the similarities in the algorithms were not noticed until the algorithms were generalised. The major differences between the original Hough Transform and the Radon Transform are as follows.

The original forms of the two transforms differ in their coordinate systems. The Hough Transform was initially developed using a Cartesian coordinate system. The original transform operated by mapping an array into a Hough accumulator, parameterised by a gradient and an intercept. However, the use of a gradient value leads to problems in detecting lines which were parallel to the vertical axis, due to the gradient of these lines being infinite. As an added problem, a gradient could range from negative infinity to infinity and required careful sampling in discretisation. These problems were solved by applying the transform to a parameterisation of a line in polar coordinates. This is the same parameterisation used in the Radon Transform.

The two transforms differ in their domain of operation. The Hough Transform’s domain is discrete data, whereas the Radon Transform operates on continuous data. This gives the Radon Transform a potential advantage as its specification is more general. If the Radon Transform is applied to discrete data, the model of discretisation is not specified and the implementation can select a model appropriate for the data (e.g. sub-pixel oversampling). In comparison, the Hough Transform embeds its discrete model directly in its specification, making it less adaptable.

The differences between the two transforms can be seen when considering a grey-value source. The Radon Transform is able to work directly on these arrays as the integral of each line in the input raster is computable. In comparison, the Hough Transform does not have defined behaviour when the input raster is not binary. There are two general approaches that can be taken in this situation when using a Hough Transform: convert the input raster into a binary array and

execute the Hough Transform as per normal, or modify the Hough Transform to handle a spectrum of input values. The former involves thresholding the grey-value source to yield a binary raster, and from here the Hough Transform is applied as per normal. The latter alters the Hough Transform so each ‘vote’ is weighted depending on the intensity of the array value. This causes ‘brighter’ (higher intensity valued) lines to be detected more strongly than their ‘darker’ (low intensity valued) counterparts. This change brings the Hough Transform closer to that of its Radon counterpart.

For the purposes of this dissertation, the output and nature of the Radon and Hough Transforms are similar enough that geological optimisations of either transform are considered to be applicable to linear feature detection.

### 3.2.4 Optimisations for Geological Data

The Hough and Radon Transforms are frequently used for geoscientific applications. Fitton and Cox [13] used the Hough Transform to identify linear features on land satellite images. Zhang, et. al [36] use the Radon Transform to find linear anomalies in aeromagnetic datasets. Sykes and Das [33] used the Radon Transform to remove linear artefacts from datasets. These geoscience applications extend and optimise the Radon and Hough Transforms using the following techniques.

#### 3.2.4.1 Normalisation

The standard Hough and Radon Transforms are biased towards long lines in the centre of the area under analysis. These features span a larger number of pixels in the input image than features which are close to corners and hence have a higher maximum number of possible votes. This causes longer features to be overrepresented and shorter features to be under-represented in the Hough accumulator and Radon space.

Fitton and Cox [13] suggest a method to reduce the effect of this problem of the Hough Transform: each Hough accumulator value is divided by its maximum possible value. This reduces the bias against corner lines as the Hough accumulator values now represent a percentage of its maximum value. This technique is possible because all elements in the input array have an equal weight and the maximum value of a cell in the Hough accumulator can be determined. However, this solution introduces a new side effect: the normalisation will over-enhance short features in the corners of the area under analysis. As a result, Fitton and Cox suggest ‘softening’ the normalisation to ensure that long lines are appro-

priately emphasised. A suggested expression for obtaining a suitable normalised value of each Hough coordinate is [13]:

$$f_n(\rho, \theta) = \frac{f(\rho, \theta)}{f_{max}(\rho, \theta)^{1/(k+1)}} \times F_{max}^{-k/(k+1)}$$

Where  $F_{max}$  is the maximum value of  $f_{max}(\rho, \theta)$ , and  $k$  is a user-defined softening parameter. The optimal value of  $k$  will depend on the characteristics of the input image and the size of the area under analysis.

The Radon Transform suffers from the same bias problem as the Hough Transform. For a rectangular input space, a line crossing from one corner of the raster to the diagonally opposite corner has a much greater potential line integral than a line crossing the corner of the image due to the increased number of points that are integrated. This effect can be seen in Figure 3.3: line A will have a lower maximum Radon space value than line B, because there are fewer points in the source raster to integrate across.

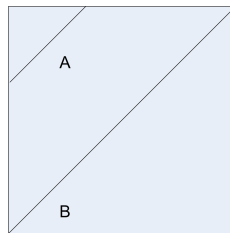


Figure 3.3: Biases in the Radon Space

Zhang, et. al. [36] describe an adjustment to the Radon space that reduces this effect. Similar to the method described by Fitton and Cox, their ‘Mean Gradient’ technique involves adjusting the values in the Radon space by normalising to their maximum value. However, the approach used is slightly different as it is not possible to determine the maximum value of a cell in the Radon space. Instead, the Radon space values are divided by the result of a ‘unit’ transform: the Radon Transform of an array containing all ‘1’ values, of the same dimensions as the input array. This reduces the bias towards lines in the centre of the image. The Mean Gradient is calculated with the following equation:

$$MG(\rho, \theta) = \begin{cases} \frac{R_g(\rho, \theta)}{R_n(\rho, \theta)} & R_n(\rho, \theta) \geq 1 \\ 0 & R_n(\rho, \theta) < 1 \end{cases}$$

In the equation,  $R_g$  is the Radon Transform of the original raster and  $R_n$  is the Radon Transform of an array with all array elements set to ‘1’, of the same

dimensions as the original raster. From here, linear features are extracted out of the results of the mean gradient array transformation. This method provides normalisation and reduces the bias of extracting linear features in the centre of the raster. The same side effect seen in Fitton and Cox’s technique is also seen, as small linear features can be overemphasised by this technique, and a ‘softening’ modification to the Mean Gradient equation is also suggested by Zhang, et. al.

A simpler form of normalisation is presented by Karnieli, *et al.* [22]. They suggest that a normalisation should be applied to the Hough accumulator to allow for consistent thresholding. Their normalisation scales Hough accumulator values according to minimum and maximum ‘vote’ counts in the entire Hough accumulator:

$$z_{\rho\theta} = \frac{a_{\rho\theta} - a_{min}}{a_{max} - a_{min}}$$

where  $a_{min}$  and  $a_{max}$  are the minimum and maximum ‘vote’ count in the entire Hough accumulator respectively.  $z$  values are then searched and ranked from highest to lowest, indicating dominant linear features.

#### 3.2.4.2 Non-maximal suppression

The standard process in selecting potential linear features to extract is to select the dominant peaks in the Hough or Radon space by sorting the transform coordinates by value and then iterating from highest to lowest.

An irregularity in the standard Hough Transform is that several dominant peaks in the Hough accumulator may be detected for a single dominant feature which is thick or noisy. Fitton and Cox [13] suggest using the following process to address this issue:

1. Sort the accumulator and record the location of the highest peak. Extract a line equation from this peak.
2. Use the extracted equation to model a line. Iterate across this line, subtracting the ‘votes’ in the accumulator that the line infers.
3. Check if the peak value found in step 1 has been reduced to a particular user-defined threshold.
4. If the threshold has not been reached, increase the width of the modelled line and reperform the subtraction process until the threshold is reached.
5. Repeat for each subsequent highest peak.

This process reduces the problem of a maximum in the Hough accumulator masking other linear features which are also significant. The drawback of this method however is that the cut-off value is heavily dependent on the image being analysed.

The Radon Transform also has similar problems: a dominant linear feature may mask other linear features which are also nearby in the Radon space. A solution to this is suggested by Fam, et. al [12]. Rather than iterating Radon space coordinates from highest value to lowest, each coordinate is instead asserted to be a local maxima. Coordinates implying lines that are not local maxima are assumed to be due to noise and are discarded. The suggested technique involves applying a dilation operator in two dimensions to the Radon space, and identifying points which did not move due to the dilation. These points are inferred as local maxima and extracted as linear features.

### 3.2.4.3 Windowing

Karnieli *et al.* [22] present an adaptation of the Hough Transform which can detect line segments of varying sizes and intensities. It constructs a neighbourhood for each pixel by selecting a rectangle of surrounding pixels, and applying a modified Hough Transform to the neighbourhood. Significant linear features are then obtained from the Hough accumulators line segments, which are merged to cover the dimensions of the input image.

The algorithm works by summing the values along a line using the expression:

$$\sum (|(g_i - g_0)| \leq Threshold)$$

The symbols  $g_0$  and  $g_i$  represent the intensity of the pixel in the centre of the neighbourhood and the intensity level of another pixel on the potential line respectively. Thus, this algorithm detects lines passing through the centre of the neighbourhood, which are of a similar intensity to the centre pixel. If a line passes the threshold test on the first run, its corresponding Hough accumulator coordinate in the accumulator is increased by the intensity of the pixel.

The second run of the algorithm detects black lines. This process is similar to that of the first run, except the pixel values are now inverted in contrast to give black pixels high values and white pixels low values. These results are also combined into the accumulator and the Hough accumulator is then normalised across the maximum and minimum values in the accumulator and thresholded to obtain dominant linear features.

An issue left unresolved is how line segments from individual neighbourhoods should be merged to reconstruct linear features across the original image dimen-

sions. The authors of the paper note that using their algorithm, dark and bright lines are mixed in the reconstruction process. This may cause artefacts to be detected as significant linear features and reduces the algorithm's utility.

## 3.3 Method

### 3.3.1 Proposed Approach

As an example of image processing tools embedded into ArcMap, a linear feature detection tool was implemented and tested as part of this project. A variety of tools were developed to perform sub-functions of linear feature detection such as the Radon Transform, Hough Transform, edge detection and other related processing tools.

### 3.3.2 Modules Implemented

The extracting of linear features from geoscientific rasters is a multi-stage process. An example of a process to detect linear features is as follows:

1. Select aeromagnetic source rasters.
2. Apply Sobel Edge detection to generate a gradient magnitude raster.
3. Apply the Hough or Radon Transform to the data to detect linear features.
4. Extract the linear features from the transform space.
5. Optimise the process to improve feature extraction in a geological setting.

A brief overview of the modules created to aid in linear feature extraction for the Image Processing Framework is given below.

#### 3.3.2.1 Sobel Edge Detection

Linear features in gravity and aeromagnetic data are often difficult to distinguish in grey-level rasters as there may not be anything immediately remarkable about the data describing the linear feature to distinguish it from its surroundings. However, it has been shown that the gradient extrema of anomalies can be used to approximate the edges of the source bodies [36]. As such, a module to compute

the gradient at each point in the raster was created to take advantage of this finding.

Although the Sobel operator [8] is usually associated with the detection of edges in an image, it is used in this project to generate a new raster which describes the magnitude of the gradient implied by the original raster. The magnitude of the gradient is calculated by convolving the original raster with a 3x3 detector mask, approximating the following equation:

$$\nabla f(x, y) = \text{mag}(\nabla \mathbf{f}) = \sqrt{G_x^2 + G_y^2}$$

$\nabla \mathbf{f}(x, y)$  is the gradient function in a Cartesian coordinate system, a vector. The magnitude of the gradient for each point in the raster is calculated, and stored in the corresponding gradient raster. The approximations to  $G_x$  and  $G_y$  are shown in Figure 3.4 as the horizontal Sobel and vertical Sobel masks respectively.

$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 & 1 \\ 2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$
(a) Horizontal Sobel Mask.	(b) Vertical Sobel Mask.

Figure 3.4: Sobel Convolution Masks

### 3.3.2.2 Hough Transform

The initial Hough Transform code used in the Hough Transform module was ported from the C code in Parker’s ‘Algorithms for Image Processing and Computer Vision’ [30]. Several optimisations were applied including skipping input array elements with a zero value and precalculation of some values which are used in the calculation of line equations. The code was also modified to work with either a standard image coordinate system ((0,0) in the top left corner extending positively in the right and down directions) or a Cartesian coordinate system. This modification allows the Hough Transform module to work on standard images (pngs, jpegs, tiffs) as well as geoscientific rasters, facilitating easier testing.

### 3.3.2.3 Radon Transform

As previously mentioned in 3.2.2, the Radon Transform operates in a continuous domain, but the raster and image data is read as an array. To approximate

the continuous form of the Radon Transform, a nearest neighbour approach was used to select input array elements which belong to a particular line. This is the simplest approximation to a line in discrete form.

The effects of the nearest neighbour approximation can be seen in Figure 3.5 which shows a line overlaid on an array, and Figure 3.5(b) shows a nearest neighbour approximation to this line. Notice that it is difficult to infer the original line from the approximation. If the nearest neighbour approximation causes too great a loss of precision, techniques such as interpolation and oversampling can be used to obtain a better representation of the original line. This would be important if an inverse Radon Transform was necessary. However, in the extraction of linear features, the nearest neighbour operation is sufficient because the process is one way and there is no need to reflect back upon the original line.

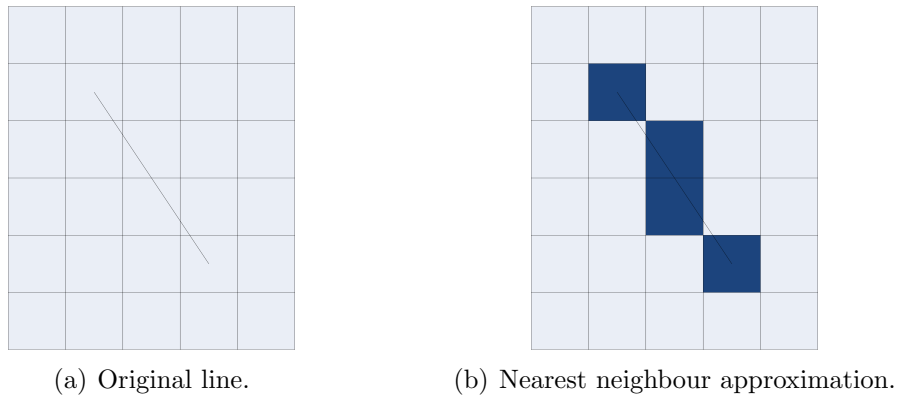


Figure 3.5: Nearest neighbour approximation to a line.

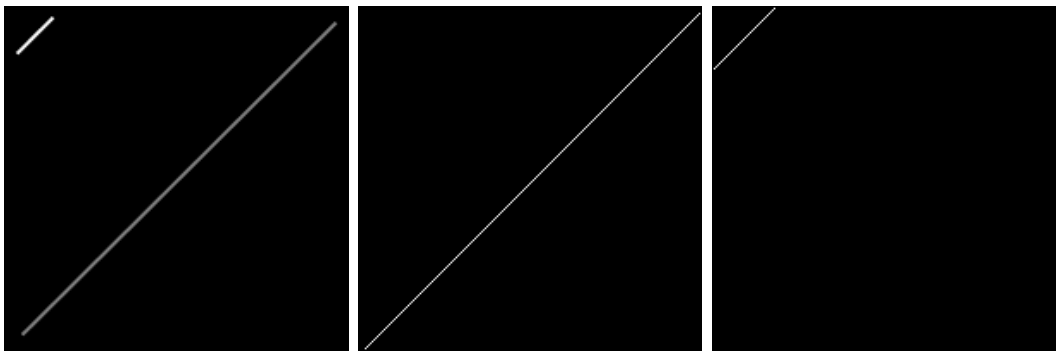
Two aspects of optimisation are considered. One aspect speeds up the implementation of the Radon Transform, by caching frequently used operations and multi-threading. This is reported later in Chapter 5. The other aspect is optimisation of the Radon algorithm to deal with its limits as discussed in Section 3.2.4. These optimisations include normalisation, windowing and non-maximal suppression.

### 3.3.2.4 Normalisation

After obtaining the Radon space or the Hough accumulator of the input raster, the transform space is searched for points of maximum intensity. These points are selected by thresholding the Radon space with a user specified threshold. The thresholded points are then used to imply lines in the input raster. This is

performed by intersecting the extracted line with the four edge sides of the input raster, and using the intersection points as starting and ending points of the line.

Normalisation, windowing and non-maximal suppression techniques are implemented as part of the Radon line extraction module. The Mean Gradient technique described by Zhang et. al. [36] which was previously explained in section 3.2.4.1 reduces the problem of lines in the centre of the raster dominating the Radon Space. The effects of the Mean Gradient technique on dominant lines can be seen in Figure 3.6.



(a) Raster containing two lines. (b) Most dominant line without mean gradient. (c) Most dominant line with mean gradient.

Figure 3.6: Effects of using the Mean Gradient technique.

Figure 3.6(a) shows a raster containing two lines. The line in the top left corner of the raster is of greater per unit intensity than the line spanning the widest portion of the raster. As seen in Figure 3.6(b), the most dominant line detected by the normal Radon Transform is the longer line through the centre. If the Mean Gradient technique is applied, the most dominant line becomes the shorter, more intense line and this is seen in Figure 3.6(c). This technique allows shorter lines to be pushed up in rank prior depending on their brightness and can result in more reliable feature extraction.

### 3.3.2.5 Tiling and Windowing

To detect linear features of various lengths, windowing can be incorporated to the Radon line extraction module and this technique was previously discussed in more detail in section 3.2.4.3. Windowing and tiling both involve segregating the input array into many smaller arrays and executing the linear feature extraction process on each individual array. However, tiling assumes no overlap between the

smaller arrays, while windowing overlaps the smaller arrays at a specified sliding interval.

The results of the extraction process are then collated. As this process works with smaller arrays, it increases possibility of extracting lines which would have otherwise been masked by other linear features and noise. An example of this is seen in Figure 3.7: the input raster is a synthetic noisy input raster containing two lines, with the left line being more dominant in length and intensity. Figure 3.7(b) shows the result of extraction without windowing. Figure 3.7(c) shows the result of extraction with the same parameters as Figure 3.7(b), but with windowing enabled.

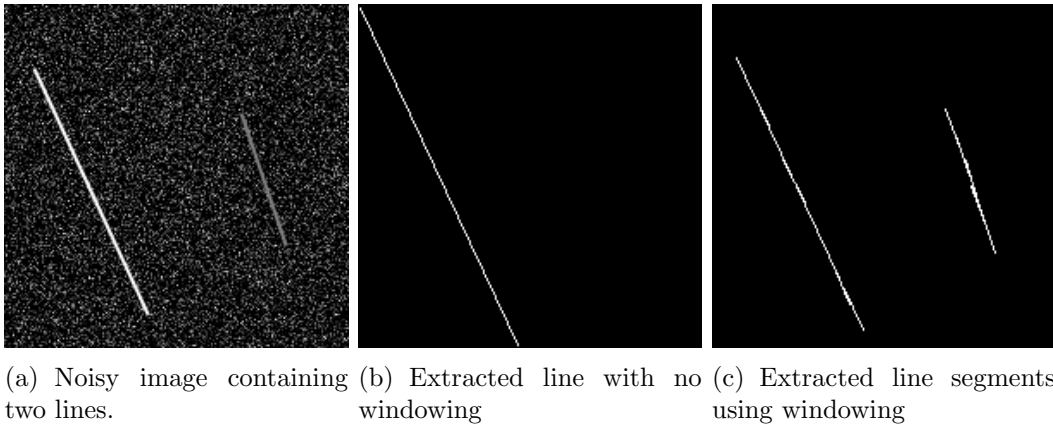


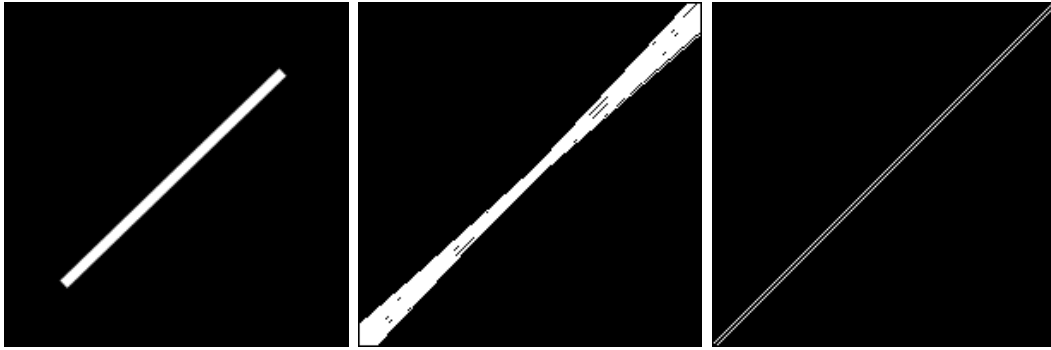
Figure 3.7: Effects of Windowing

The windowing algorithm in Figure 3.7(c) uses a window size of 50x50 pixels (a quarter of the original raster in both directions), and slides from left to right with an overlap of 75%. The process isolates the left linear feature from the right, extracting them independently. This yields better extraction results at the cost of increased computation.

### 3.3.2.6 Non-Maximal Suppression

Non-maximal suppression is also implemented in the Radon line module. The effects of non-maximal suppression are shown in Figure 3.8. Figure 3.8(a) is the original raster, and in real world data this may represent a feature which was blurred or smeared in data acquisition. Figure 3.8(b) shows all detected lines in the Radon space without any additional processing. The majority of these lines arise due to the nature of the transform: if the original axis of the line is taken and rotated slightly, the line integral across this skewed line is still relatively high.

Non-maximal suppression addresses this problem by removing potential lines in the Radon space which are not local maxima. Figure 3.8(c) illustrates the effect of non-maximal suppression: only two parallel lines are extracted.



(a) Thin rectangle in raster. (b) All detected line equations without non-maximal suppression (c) All detected line equations with non-maximal suppression

Figure 3.8: Effects of Non-Maximal Suppression

The non-maximal suppression technique developed in this project adapts the concept from Fam’s dilation technique [12], but instead of using a dilation operator a local search takes place. For each potential maxima in the Radon space, its immediate neighbours are inspected and their values are compared to the element being processed. If the number of neighbouring pixels with a Radon space value less than or equal to the current element exceeds a threshold, this element is a local maxima. If the threshold is the number of neighbours - 1, then the technique detects only a single local maxima. This threshold gives further control over the non-maximal suppression process: increasing the threshold selects Radon coordinates which are more likely to be true lines.

This method is preferential over dilation for two reasons. Firstly, it has more flexibility than dilation and the threshold value can be adjusted due to the nature of the data. For example, if the original raster contained many lines with only a slight angle of deviation, a user may wish to lower the threshold to detect more of these lines. They would then manually inspect the output and delete lines which were not correct. Secondly, the dilation and match operation is computationally expensive.

### 3.3.2.7 Other Modules

In addition to the Sobel and Radon line extraction, a number of other modules were also built. These provide basic image processing functions including normalisation, thresholding, horizontal and vertical mirroring and Gaussian smoothing.

## CHAPTER 4

# Unifying Example Code, Unit Tests and Documentation

Real world problems are mostly complex in nature, thus software providing solutions to these problems also requires complex designs. These designs contain abstractions and patterns which must be communicated effectively to software developers seeking to use these systems and this is usually provided in the form of documentation. However, not all software comes with sufficient documentation and developers must seek out other methods to improve their knowledge of these systems. This chapter outlines the strategy used in this project for obtaining and capturing this knowledge, deemed *Code Experiments*. It also provides an overview of *ArcMapUnit*, a specialised test harness that was developed to provide tool support for Code Experiments.

## 4.1 Problem

The Image Processing Framework created for this project must interact with the ArcGIS API to achieve high level tasks such as drawing a line or polygon on a map. However, the documentation provided for the ArcGIS API is class level documentation. This means it only describes the fundamental building blocks of the software: the classes and their methods, fields and properties. It does not describe how classes interact with each other to achieve high level tasks. This knowledge is just as important to the user as the low level description of individual classes, as it provides information on how to assemble these building blocks into functional software. Thus, it is necessary to discover and capture this knowledge, and document these relationships in an effective and useful way. A possible solution to the problem of understanding complex software systems comes in the form of API example code.

## 4.2 Example Code

Example code is a short segment of code which can demonstrate a single facet of high-level functionality by showing how to successfully coordinate low-level software components. A developer can read these examples and follow the same process to achieve their desired tasks. However, there are storage, verification and maintenance issues to consider.

**Storage Problem:** Example code is often stored in a *non-executable state*. For example, code found in a textbook or in help files cannot be executed without first transcribing it into a source code file and compiling it. This reduces the utility of example code as there is an intermediate step where the code must be transferred from one medium into another. As a result it would be advantageous to store example code in a more appropriate medium to facilitate reuse [1].

**Verification Problem:** The value of non-executable example code is reduced because there is no mechanism for ensuring that the code actually achieves the intentions of its author. This problem can be addressed by providing test assets for the example code which demonstrate that an example is correct.

**Maintenance Problem:** Example code is difficult to maintain as it is not kept in an executable form and there are no mechanisms to verify it. This often leads to a loss of synchronisation between the example code and the libraries it describes [1]. The syntax and intentions of API functions may change over time but the example code is not updated to reflect these changes, thus it loses its usefulness.

In this project, a process called *Code Experiments* was created to address these problems. Code Experiments provide a solution to the storage, verification and maintenance problems of example code by adding verification tools and providing tool support in the form of a specialised test harness. Code Experiments facilitate the acquisition and capture of knowledge of the ArcGIS API with the aim of creating packaged experiment artefacts which demonstrate high level ArcGIS tasks.

## 4.3 Background: Addressing Problems in Example Code

The following sections review current literature and current software tools which were used to formulate the Code Experiment process. Firstly, the verification problem is addressed using the findings of Hoffman, *et al.* [19] who show that unit tests can be used to document APIs. Secondly, the storage problem is discussed by investigating tool support and an overview of existing unit testing

tools. Finally, the important features of useful documentation are reviewed as proposed by Forward and Lethbridge [14].

### 4.3.1 Verification Problem

#### 4.3.1.1 Unit Testing

A Unit Test is a procedure that exercises a unit of code in isolation and compares the result with an expected output. The motivation for unit testing comes as a dual from decomposing large software systems: large software systems are broken down into a number of modules to reduce the complexity of any one part of the system. Unit tests achieve the same purpose: instead of testing the entire application, they focus on testing the small building blocks of a program [26].

#### 4.3.1.2 Documentation using Tests

Traditionally, Unit Tests are written to verify software modules within a larger system. However they also have other uses and Hoffman, *et al.* [19] describe a use for Unit Tests in documenting APIs.

In general, there is a spectrum of documentation types with prose specified at one extreme, and formal specifications at the other. Each of these has its advantages and disadvantages: Prose can provide a good overview of a system but may not provide the precision that is required by readers of the documentation. Formal specifications are very precise but may be difficult to understand and do not exploit domain knowledge which the reader may have of the system. Additionally, neither technique provides mechanisms to check that code and documentation are consistent.

As a compromise, Hoffman *et al.* present a system where documentation is given by the generation of test cases and supporting prose which describes the test cases. This technique is described as the ‘FAQ approach’ [18] and has four main benefits over traditional prose documentation:

- Precise (though partial) documentation.
- Guaranteed consistency of code and documentation.
- Good fault detection.
- Helpful examples of use.

Precise (though partial) documentation is achieved because each test states its inputs and expected outputs. This defines the scope of the test by showing the boundaries over which the test operates. Although it is not feasible to cover all possible inputs, inputs can be chosen to demonstrate cases which are of the greatest utility to the reader. These cases may cover the most common functionality or special cases.

There is guaranteed consistency between code and documentation, because it is easy to detect when inconsistencies exist: the reader can verify the test case by simply executing it. Any inconsistencies between the code and documentation become immediately apparent as they will show up as failed tests.

Test cases provide good fault detection as they verify the expected output of the code against its actual output. At any time, test cases can be re-run to determine if the behaviour of the underlying libraries has changed such that different outputs are produced. If there is a mismatch between the expected output and the actual output, the test case fails. As a result, these tests provide a form of quality assurance of the example code.

Helpful examples of use are provided as each test case contains complete, executable examples of code. Developer utility is maximised because each test case is in the same programming language that the reading developer is using. Thus, there is no need to unnecessarily translate the contents of the test case into another medium before the developer can use the knowledge.

In addition to these ideas, Jeffries *et al.* [20] build on unit tests as documentation of a system and describe how tests can be used to specify the scope of inputs which the system has been designed to handle. These specific unit tests provide code examples which have been explicitly marked to fail with the current software. These provide a formal statement about the boundaries of the system and describe the legal and illegal input ranges for code. If it is necessary to make changes to the codebase at a later point in time, these unit tests can provide feedback to the programmer if the system boundaries have changed. This helps to improve the quality of software being developed.

#### 4.3.1.3 Verifying the Output from Tests

The xUnit family of testing tools also includes a comprehensive set of methods for asserting the status of variables. NUnit [27] is a member of the xUnit family which provides tool support for unit testing in the Microsoft .NET Framework. An example of NUnit tests can be seen in Figure 4.1.

In the example, there are two implementations of the interface `IAdder`, `GoodAd-`

---

```

using System;
using NUnit.Framework;

[TestFixture]
public class AdderTests
{
    [Test]
    public void TestGoodAdder()
    {
        IAdder goodAdder = new GoodAdder();
        TestAdder(goodAdder); // test passes
    }

    [Test]
    public void TestBadAdder()
    {
        IAdder badAdder = new BadAdder();
        TestAdder(badAdder); // test fails
    }

    private void TestAdder(IAdder adder)
    {
        int a = 1;
        int b = 2;

        int c = adder.Add(a, b);

        Assert.AreEqual(3, c);
    }
}

public interface IAdder
{
    int Add(int a, int b);
}

public class BadAdder : IAdder
{
    public int Add(int a, int b)
    {
        return a; // this is deliberately wrong
    }
}

public class GoodAdder : IAdder
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}

```

---

Figure 4.1: An example Test Fixture in C# using NUnit

der and BadAdder. GoodAdder has a correct implementation of Add, whereas BadAdder's implementation is wrong. Both classes are verified using the private method TestAdder, which uses the NUnit Assert class to ensure that the actual output is equal to the expected output of the test.

The Assert class in NUnit provides the crucial verification stage in a unit test. Assertions check the output of the test against predefined values and signal to the test harness if verification fails. In the given example, the test TestBadAdder fails because the actual output from the BadAdder class did not match the expected output. NUnit's Assert class also contains other verification methods including checking for equality, inequality, reference equality, and Boolean expressions. This provides a systematic way of verifying the result of the test.

## 4.3.2 Storage Problem

### 4.3.2.1 Existing Unit Testing Tools

One of the most popular families of unit testing tools is the xUnit family. Their origins lie in Kent Beck's original SmallTalk unit testing program, SUnit, and have been made popular by featuring in Extreme Programming [3] and Test-Driven Development [4]. These tools provide support for executing unit tests by providing a test host, a graphical user interface (GUI), and a comprehensive assertion library for verifying the output of tests.

### 4.3.2.2 A Standard Process for Executing Unit Tests

In most cases, a unit test does not execute in isolation. There are dependencies at many levels which need to be set up before a unit test can be executed. In order to provide the various hierarchies of dependencies required by a unit test, the xUnit suite of testing tools introduces the concept of a *Fixture*: A single configuration with predictable behaviour. A fixture is a consistent environment in which a test executes that provides the necessary dependencies which are required to validate a segment of code. The xUnit suite of testing tools also provides a system to set up these fixtures called the standard test execution cycle. This system was first described by Kent Beck in his paper "Simple Smalltalk Testing: With Patterns" [2] and is shown in Figure 4.2.

The establishment of the base test environment is denoted as *TestFixture-SetUp*. This phase of the test cycle is used to load any dependencies and initialise classes that are necessary for a test to run.

A *Test Case* is a stimulus that is to be verified, and the stimulus may have local dependencies. To ensure that all tests have a consistent execution environment, the xUnit frameworks provide a mechanism for setting up local dependencies which are renewed in between Test Cases. These mechanisms form the *SetUp* and *TearDown* stages of the test cycle. *SetUp* is executed prior to running each test and ensures that all dependencies needed for the test are available. *TearDown* is executed after the running of each test and ensures that consumed dependencies have been reset.

The *Test* phase of the cycle executes the unit test. Throughout this stage, the actual output of the code under test is verified against the expected values. If at least one actual output does not match its expected dual, an error is raised with the test harness.

Finally, the xUnit framework also provides a means of shutting down the Fixture cleanly. The *TestFixtureTearDown* stage allows a programmer to clean up and free resources which are used in setting up the Fixture. This phase runs regardless of whether the tests succeed or fail.

An example demonstrating the standard test execution cycle for a database test is listed below.

1. **TestFixtureSetUp:** Create a connection to the database.
2. **SetUp:** Create testing tables in the database.
3. **Test:** Insert some rows into a table. Execute a query to verify that the data has been stored correctly.
4. **TearDown:** Drop testing tables from the database.
5. **TestFixtureTearDown:** Close database connections.

### 4.3.3 Useful Documentation

#### 4.3.3.1 Important Features of Documentation

An important aspect of software maintenance is documentation: it serves to describe the intentions and implementation of a system. Forward and Lethbridge [14] performed a survey across industry to investigate the perceived relevance of software documentation and to determine which documentation tools and technologies are used. It was found that documentation is an important tool for communication and ‘lightweight, everyday documentation’ is the most useful.

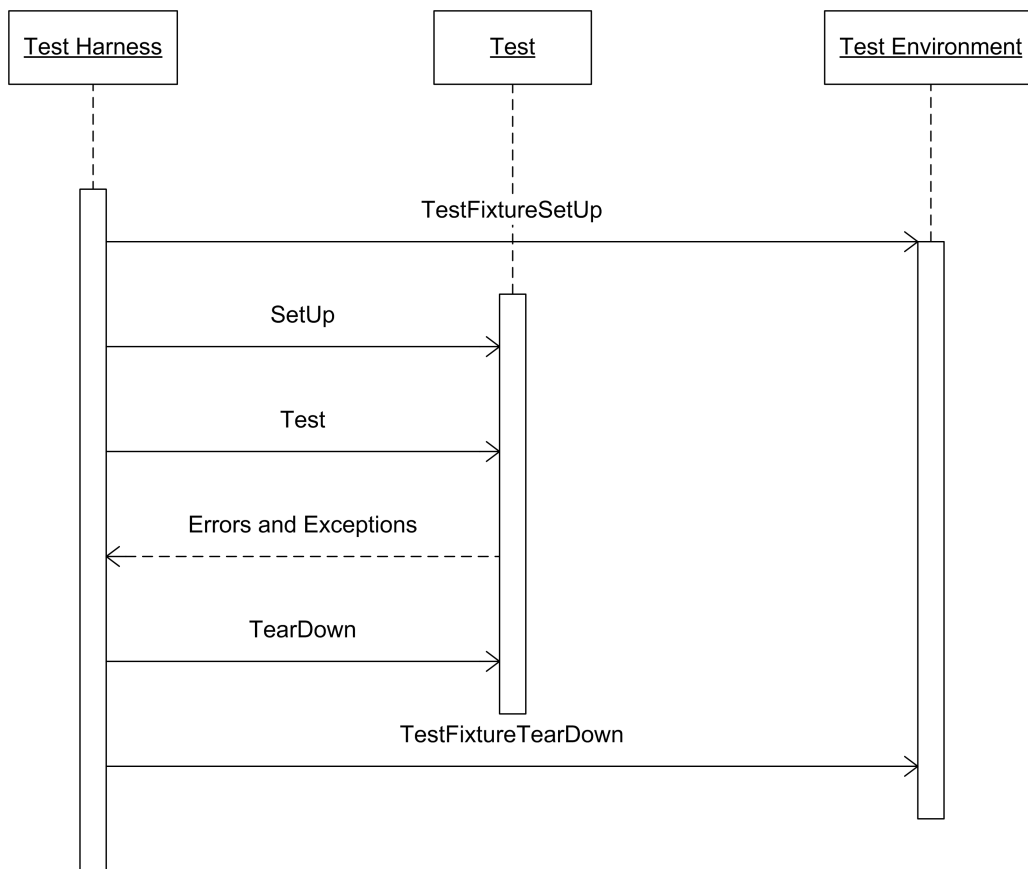


Figure 4.2: Standard xUnit Test Cycle

Key features of this form of documentation are:

- Content creation over maintenance: Technologies which are easy to use and support the creation of information as opposed to its maintenance.
- Ideas over accuracy: Documentation is a communication medium and facilitates the communication of ideas and provides prompt feedback.
- Simplified features: Documentation tools should not constrain users from communicating effectively.
- Automated archiving: Documents are unlikely to be discarded and due to this information overload becomes a problem. Users must be able to archive documentation to reduce the total number of visible documents without having to discard documents.

- Reader feedback: Feedback allows the author of the document to produce more useful content.

The survey also noted that that test code may also contain useful data. Overall, the majority of participants agreed that a lot of information can be extracted directly from source code and that automated testing provides resources that serve as useful documentation. These findings were incorporated into the software developed in this project by creating an environment in which this information can be easily understood and used as a documentation resource.

## 4.4 The Process of Code Experiments

A *Code Experiment* unifies concepts from example code, unit testing, and documentation and it consists of a short segment of code which demonstrates a high level task (example code), a segment of code which verifies that task (integrated unit tests), and any associated comments and remarks (documentation). The process described here is applied to ArcGIS in this project, but is applicable to other software systems.

**Example Code:** Code Experiments provide a foundation for the acquisition of new knowledge. A developer can use the code experiment environment to provide a point of entry into the ArcGIS environment. The standard xUnit test cycle described in Section 4.3.2.2 has been adapted and ensures that the experimental environment is consistent and reproducible, giving a programmer the opportunity to try out various functions of the ArcGIS API. The codifying process captures the knowledge that they have acquired and stores it in an executable form as example code. This is different to the process described by Hoffman *et al.* [18] as the author of the experiment is not the API author, and the objects under test are black-boxes where the underlying implementation is unknown.

**Integrated Unit Tests:** The above process is similar to writing example code. However, Code Experiments add further value by providing a facility for verifying example code. The NUnit assertion tools discussed in Section 4.3.1 have been adapted to the Code Experiment environment and can be placed in the example code. These statements compare the output of the system against an expected output and are powerful because they allow a developer to verify their understanding of the API functions against the concrete API specification. Thus code experiments provide a means to align the specification as understood by the developer with the specification hidden behind the API.

**Documentation:** Finally, once a developer has a good understanding of the API specification, the experiment is documented with metadata and comments.

This allows Code Experiments to be used as a knowledge sharing device where other developers can read experiments and executing them. This helps other developers improve their knowledge and understanding of the ArcGIS API. These experiments are now also a form of documentation.

## 4.5 Method

*ArcMapUnit* is the environment that was created for Code Experiments in this project. It provides a functional test harness in which experiments can be run. ArcMapUnit provides tool support for acquiring and capturing knowledge of a previously unknown software system. Experiments can be written to demonstrate high level functionality of the ArcGIS API, showing the temporal and dependent relationships between classes. This provides a form of documentation for the ArcGIS API which has the added ability of being executed and verified.

A screenshot of the GUI for ArcMapUnit can be seen in Figure 4.3.

In this section, the development of ArcMapUnit is described, outlining the problems encountered and software solutions that were developed.

### 4.5.1 Tools for Experiment Support

In order to provide a solution to the maintenance problem of example code, Code Experiments are supported by a specialised test harness in which the code experiments execute. The test harness provides a graphical user interface for controlling experiment execution and supports the verification framework used in experiments. Existing tools were reviewed to determine if they could be extended to support the Code Experiment environment.

The following requirements were considered when evaluating an existing tool:

1. The test harness would need to initialise the ArcMap environment and provide access to its objects.
2. The test code would behave as if it was called directly from ArcMap.
3. The test harness would provide a consistent environment for executing tests.

The test environment must be initialised and set up in a way that mimics a minimal environment for the tests to run. Any dependencies of the classes and instances under test need to be available and respond in the same way as they

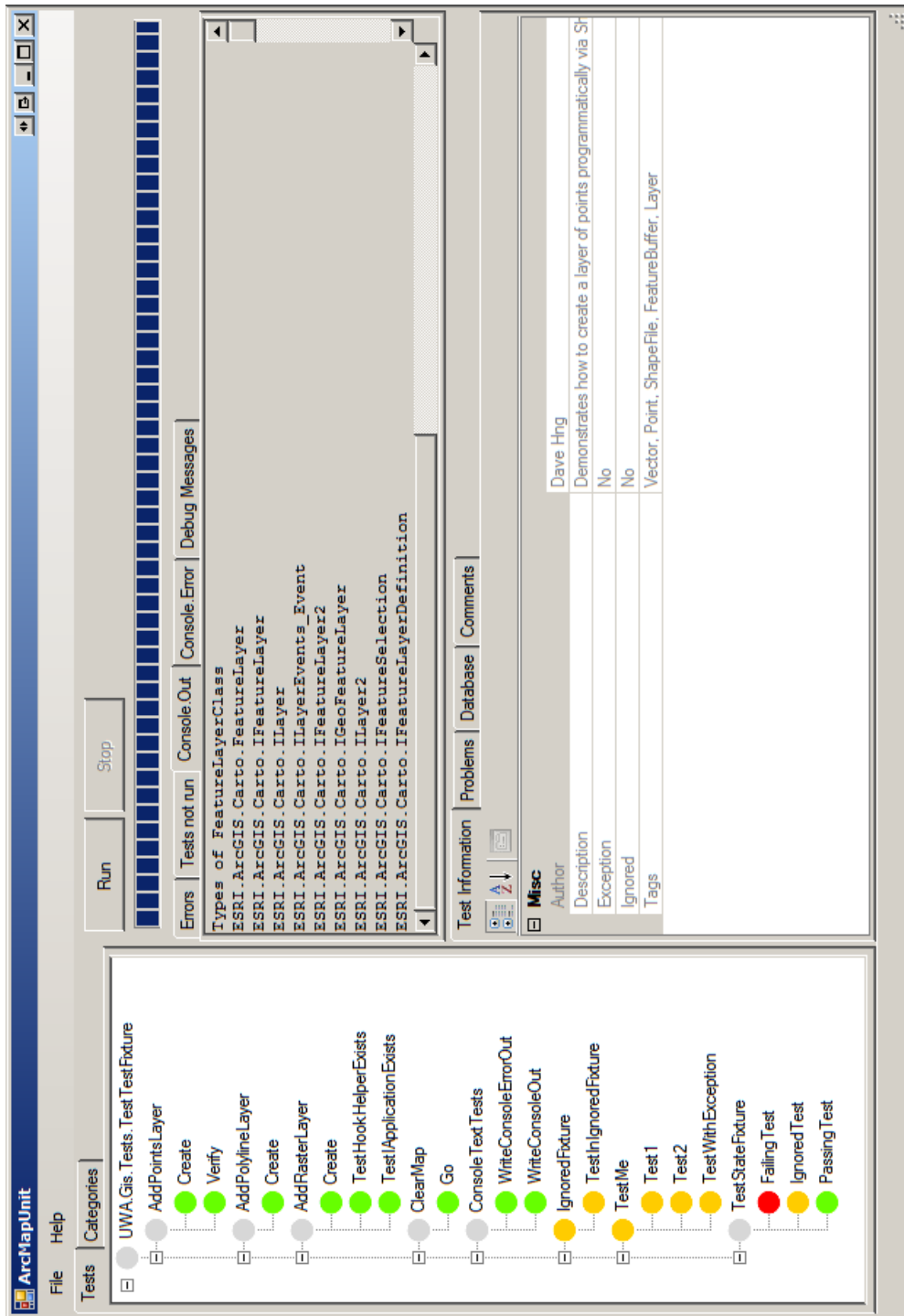


Figure 4.3: ArcMapUnit User Interface

would in the production environment. This means that all global variables and control objects should be available to the code under test. This ensures that tests accurately represent the same cases as a class or instance would encounter in the real world. The test environment must also be consistent: tests should not interfere with each other. If tests were not able to execute in a mutually exclusive manner, tests may fail due to residual data remaining from other tests.

#### 4.5.2 Reusing Existing Testing Tools

NUnit [27], a member of the xUnit testing family, was investigated as a test harness for supporting Code Experiments. NUnit provides much of the functionality that is required to execute experiments and is also open source, and could be modified to suit many additional needs.

In order to provide Code Experiments with access to the ArcMap environment, NUnit and ArcMap must share the same process space: ArcMap initialises global variables in the ArcGIS API which experiments need to execute. In order to achieve this, two strategies were considered: load NUnit then load ArcMap from within NUnit, or load ArcMap then load NUnit from within ArcMap.

The strategy of loading NUnit from within ArcMap caused memory leaks and can be seen in Figure 4.4. When executing Code Experiments, certain operations in the ArcGIS API would not fully clean up resources which they acquired, leading to problems with resetting the state of ArcMap for subsequent tests to run as shown in Figure 4.4. This undermines the third requirement of providing a consistent environment for executing tests and thus it was decided that this application structure was not usable.

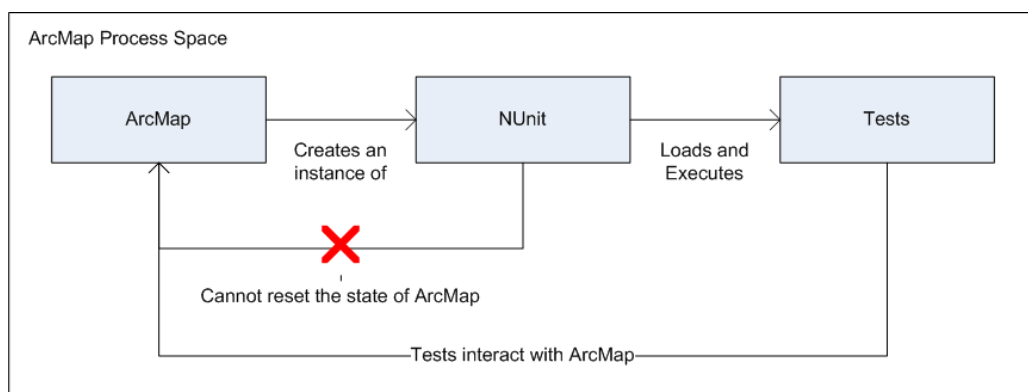


Figure 4.4: ArcMap hosting the NUnit application within its process

An alternative was to invert the process structure and host ArcMap from within NUnit. This structure can be seen in Figure 4.5. Here, NUnit controls ArcMap by creating a new instance of the ArcMap application during the *TestFixtureSetUp* phase of the xUnit test cycle. This resolved the memory and residues problems but hosting ArcMap from within NUnit yielded other problems. ArcGIS objects created from within NUnit were not initialised properly and behaved inconsistently when used. This problem with object initialisation breaks our second requirement: test code would not behave in the same way as if it was called from ArcMap. As a result, this application structure could not be used to support the Code Experiment environment either.

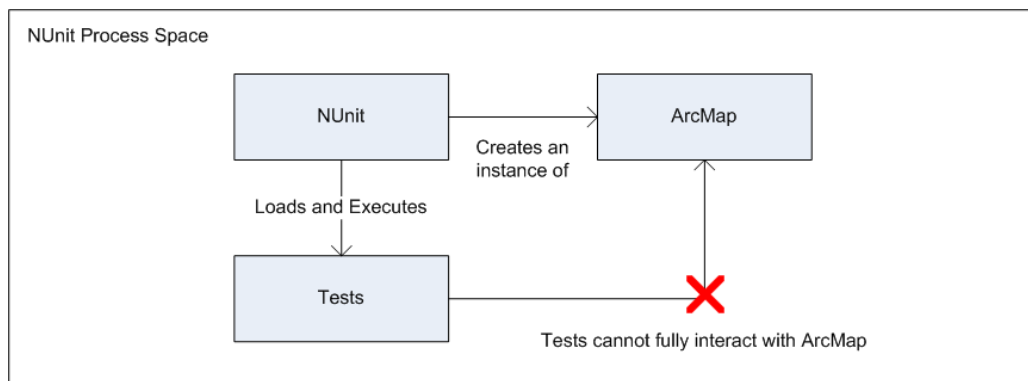


Figure 4.5: NUnit hosting the ArcMap application within its process

### 4.5.3 A Custom Test Harness

To alleviate the problems that were encountered when using NUnit as the test harness, a new strategy was formulated for this project. The design was modified to segregate the experimental environment into three parts, a test runner, a controller, and messaging system over which the two components can communicate. Although this strategy is complicated and increased the complexity of the experimental environment significantly, its main advantage is that experiments could be executed within the ArcGIS / ArcMap environment. This architecture is shown in Figure 4.6.

The Controller sets up the environment by managing the start-up of ArcMap. When a test fixture is created a new copy of ArcMap is launched and on completion of all tests the fixture is torn down and ArcMap closed. This ensures that no residues exist in the experiment environment when a new fixture starts up.

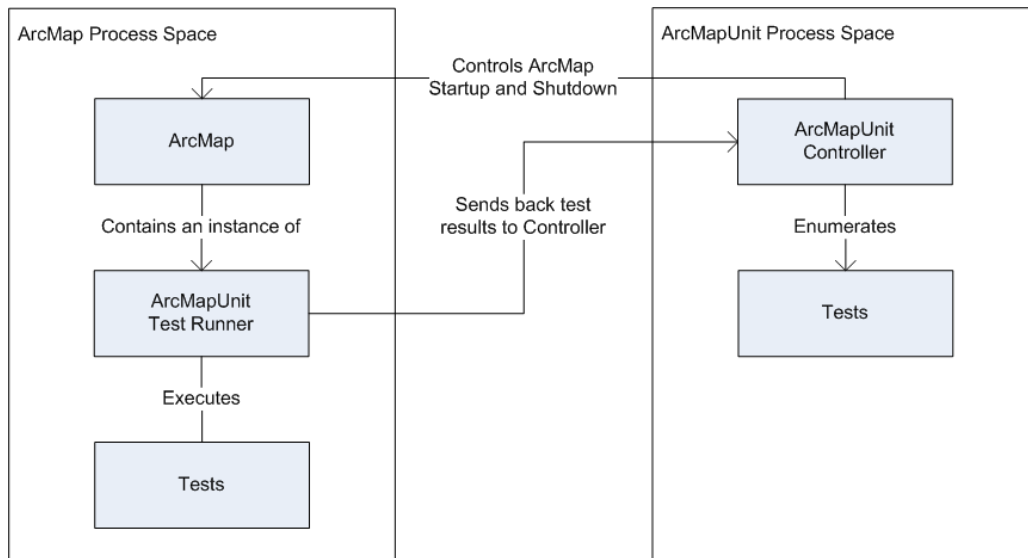


Figure 4.6: ArcMapUnit: Executing experiments across a process boundary

The Controller also acts as the role of the coordinator in the system, directing the overall test process.

The Test Launcher is a component that resides within ArcMap. Its purpose is to load experiments into ArcMap and execute them when the Controller instructs it to do so. As the launcher itself runs within ArcMap, it has full access to the ArcMap environment and its objects. This allows the experiments to behave as if they were running directly within ArcMap.

A messaging system was also designed using a publish and subscribe architecture to enable communication between the test runner and the Controller. The messaging system must also be able to work across process boundaries as the test runner and Controller exist in completely separate process spaces. This was achieved using .NET Remoting; the .NET Remote Method Invocation system. Communication between processes takes place across TCP/IP and messages are passed between applications in a binary format native to the .NET framework.

## 4.6 Experiment Environment: ArcMapUnit

To reduce the learning curve when using ArcMapUnit, the tool has been built to be as similar to NUnit as possible. Developers already familiar with NUnit will be able to use ArcMapUnit quite intuitively as most of the visual cues and

terminology are shared. The assertion code is also the same as in NUnit and developers do not need to learn another API to write code experiments. ArcMapUnit also borrows from the xUnit family of tools by providing a console which experiment code can write to. This can be used to provide additional messages about the state of the test to a user running the experiment.

The ability to select which test fixtures will run is useful in multiple situations. A user of the Controller can either execute all experiments at once, execute just one group of experiments, or execute a single experiment. This gives the user fine grained control over which tests are run which is particularly useful when writing a code experiment for the first time as it may take several iterations of compiling, executing and later modifying the experiment to reach a desired outcome. The ability to run a single test instead of the entire suite reduces the cycle time to obtain feedback from the experiment. This feature is also useful for a developer seeking to improve their knowledge of a system by reading through experiments. Again, they have the ability to select which experiments will execute. This reduces the amount of irrelevant information fed back to the developer.

ArcMapUnit also includes a number of features which are not found in similar tools such as NUnit. These features provide further tool support for executing code experiments:

- ArcMapUnit includes a logging database which records the past history of experiment runs.
- Experiments can be tagged with comments, and these comments are displayed in the test harness.
- Experiments execute outside of the ArcMapUnit process space.
- Experiments can be tagged with keywords, and can be grouped by these keywords.

ArcMapUnit uses a lightweight database, SQLite, to log the history of all tests. This history provides traceability within the test harness: if a developer encounters a code experiment that fails, they can examine the history to see if the code experiment had succeeded in the past. This gives the developer further indications about what may have changed in between the last successful run of the experiment and the current run. For example, the date of last success may indicate where to begin looking in a source code version control system for changes that caused the experiment to fail. This logging facility fulfils the Automated archiving feature suggested in Forward and Lethbridge [14].

The ability to run experiments outside the Controller is unique to ArcMapUnit. Since experiments do not run within the ArcMapUnit process space, experiments can run within complex environments that are only accessible by executing another program. Therefore, experiments can execute from within environments which are black boxes, allowing a developer to acquire and capture knowledge of the black box. This adds value to the test harness and improves its usefulness in developing and documenting unfamiliar software systems.

A feature that ArcMapUnit provides that NUnit lacks is an ability to provide descriptive data for fixtures and tests which is integrated into the test harness. ArcMapUnit has additional syntax similar to JavaDoc [32] which can be written in experiments to indicate the author of the experiment and provides an experiment description that can be viewed from the Controller. This improves the cohesion of experiment related metadata, as it provides the opportunity for an experiment author to capture additional knowledge and places it in a location which is easily found by other developers.

An additional feature is the ability to associate keywords with an experiment. For example, the experiments ‘CreateRasterLayer’, ‘CreatePolylineLayer’, ‘EnumerateLayers’ and ‘OpenRasterLayer’ may all be tagged with the keyword ‘layer’. A user can choose to sort the experiments loaded in to ArcMapUnit by keyword, such that all experiments with the keyword ‘layer’ are displayed together in the experiment selection tree of the GUI. This cataloguing system assists users examining experiments to further their knowledge of a system as they can see groupings of experiments which demonstrate related functionality.

In summary, this chapter has described the motivation behind Code Experiments and outlined the functionality of the ArcMapUnit tool. The environment provided by ArcMapUnit was used to create experiments that ensured the high level functions required by the image processing framework could be enacted through the ArcGIS API. These experiments captured the knowledge required to reproduce high level functions in an executable, verified form.

## CHAPTER 5

# Results and Evaluation

This chapter is divided into three main sections. Firstly, a brief outline of the code metrics for the systems that were built in this project is presented. Secondly, the linear feature extraction tools developed in this project are demonstrated and benchmarked. Thirdly, an evaluation and discussion of the Code Experiments developed in this project is presented.

### 5.1 Code Metrics

A significant amount of code was written for this project and a breakdown of the size of the system represented by lines of code and the number of classes can be seen in Table 5.1. These results were obtained using SourceMonitor from Campwood Software [6].

<b>Module</b>	<b>Lines of Code</b>	<b>Classes</b>	<b>Average Methods per Class</b>
Image Processing Framework	3983	62	4.40
Image Processing Modules	4608	70	4.22
ArcMapUnit Base Libraries	4527	92	3.57
ArcMapUnit Test Runner	1132	9	7.00
ArcMapUnit GUI	2786	25	5.84
ArcMapUnit Unit Tests	1331	23	3.87
ArcMapUnit Console	324	2	12.50
Code Experiments	818	11	3.73
Total	19509	294	n/a

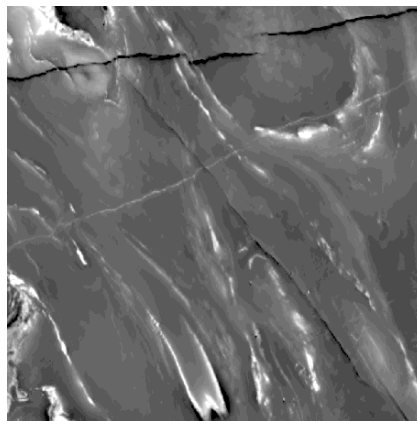
Table 5.1: Code Metrics for software authored in this project.

## 5.2 Linear feature extraction process

### 5.2.1 Demonstration

To demonstrate the linear feature extraction process, image processing modules are embedded into ArcMap using the proposed framework. These modules are then tested by running them against an aeromagnetic dataset of the Yilgarn Craton in Western Australia to extract linear features. A rendering of the original dataset can be seen in Figure 5.1. The data is then processed using the following steps:

- Applying Sobel edge detection to obtain the gradient map of the image.
- Thresholding the gradient map to remove noise.
- Applying the Windowed Radon Lines processing module.



This image has been normalised for printing.

Figure 5.1: Aeromagnetic data of the Yilgarn Craton in Western Australia.

The output from Sobel edge detection is seen in Figure 5.2. The edge strength indicates the magnitude of the sum of horizontal and vertical image gradients in this figure. The Radon transform is performed by applying normalisation, windowing and non-maximal suppression. The resulting output from the image processing modules can be seen in Figure 5.3. The white line segments in the figure have been extracted from the Sobel output. For this experiment, the parameters for the Windowed Radon Lines image processing module were as follows: global Radon space threshold of 800, non-maximal suppression sensitivity of 5, window overlap of 20%, window size of 40x40.

The results demonstrate that the implemented algorithm successfully identifies linear segments from an aeromagnetic image, which are represented as a vector layer within ArcMap. Figure 5.3 shows the line segments that are identified, and clearly specifies magnetic anomalies within the aeromagnetic grid. As a result, information useful to a geoscientist has been extracted from the aeromagnetic grid and is presented in a form which is easy to visualise, as highlighted in Figure 5.3.



This image has been normalised for printing.

Figure 5.2: Sobel edge detection applied to the original raster.

## 5.2.2 Performance Improvements

The image processing modules are used in a desktop environment, and thus it is crucial that the modules perform their processing efficiently to prevent degradation of the end user experience. As some of the algorithms used in this project are computationally expensive and incur a significant amount of processing time, performance experiments were run to identify bottlenecks in the image processing modules, and the results are used to speed up processing.

The experiments were run using the linear feature detection process on a computer with the following specifications:

- Intel Pentium Core Duo T2400 Processor
- 2GB 667MHz DDR2 SDRAM
- nVidia Quadro 110M Video Adapter

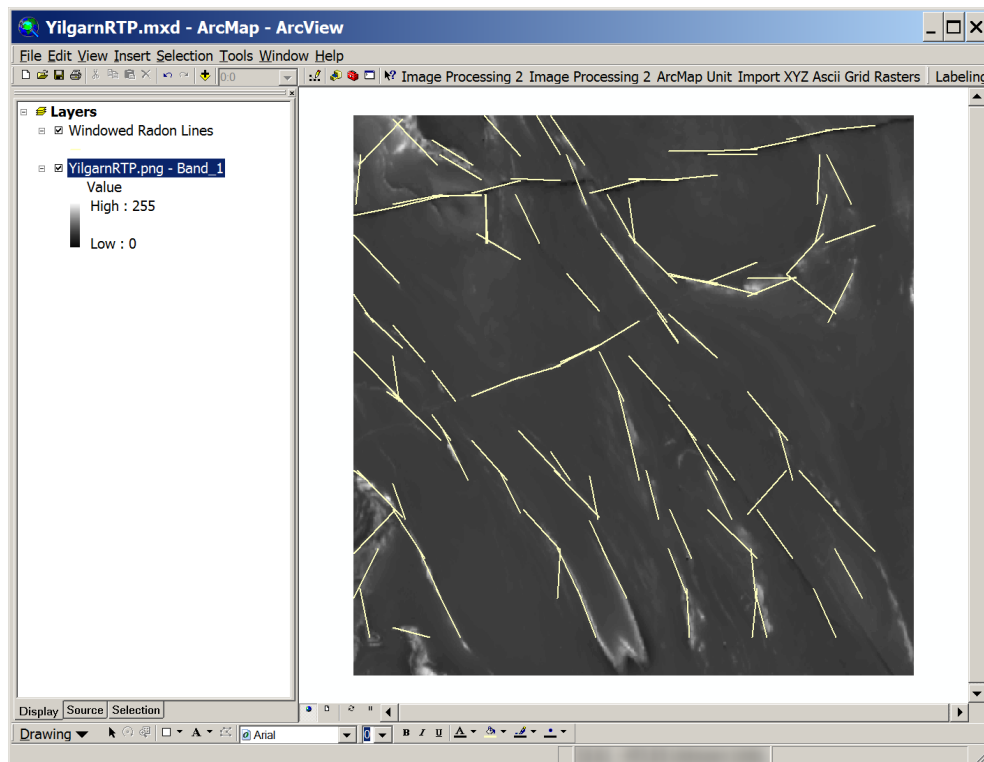


Figure 5.3: Results of image processing as a vector layer in ArcMap.

- Windows XP Build 2600.xpsp\_sp2.qfe.070227-230
- Microsoft .NET Framework version 2.0.50727
- Microsoft Visual C# Compiler version 8.00.50727.42
- Microsoft CLR Native Image Generator version 2.0.50727.832

The program binaries produced for these experiments were built using Visual Studio 2005 Service Pack 1 in release mode. The .NET Assemblies produced through compilation were processed using the CLR Native Image Generator to generate native x86 binaries. This measure removes the overhead incurred in Just-In-Time compilation which would have obscured the results.

The following workflow was selected and profiled for performance:

1. Load an aeromagnetic raster into the Image Processing Framework.
2. Apply Sobel Edge detection to generate a gradient magnitude raster.

Raster Size	Sobel (ms)	Sobel (%)	Radon (ms)	Radon (%)	Others (ms)	Others (%)
Small	70	4.00%	1670	96.00%	0	0.00%
Medium	150	4.66%	3080	95.03%	10	0.31%
Large	320	3.96%	7720	95.92%	10	0.12%
Very Large	561	3.79%	14235	96.08%	20	0.13%
Average		4.10%		95.76%		0.14%
Stdev		0.00383		0.00488		0.00128
Stdev %		0.0934%		0.00510%		0.898%

Figures indicate the execution time or percentage of execution time spent in each part of the profiled workflow.

Grid dimensions: Small: 100x100, Medium: 200x200, Large, 500x500, Very Large: 1000x1000

Table 5.2: Performance Analysis of a typical user workflow

3. Apply the Radon Transform to the data to generate the Radon space.
4. Identify local maxima in the Radon space.
5. Extract linear features from these local maxima.
6. Render the resulting linear features.

Experiments were conducted to analyse the proportion of time taken to perform components of linear feature detection for varying image sizes. The results are shown in Table 5.2.

The major bottleneck in the system was the Radon transform which consumed approximately 96% of the execution time. This was to be expected as the Radon transform is the most computationally expensive algorithm employed, and hence the Radon transform was targeted for optimisation.

Two optimisations were made: the first involves precalculating and caching the result of some frequently accessed mathematical functions. The second involves multi-threading the Radon transform algorithm to take advantage of computers with two processors.

To measure the change in performance due to these optimisations, the Radon transform was applied to a set of rasters with increasing size. The dimensions of these rasters are shown in Table 5.3. These grids were obtained by resampling the image in Figure 5.2. The set of grids was selected to provide a real world operating range for the Radon transform. A chart of the results from the experiment can be seen in Figure 5.4.

The first optimisation generated the greatest increase in performance, reducing algorithm run time to approximately a third. The second optimisation was

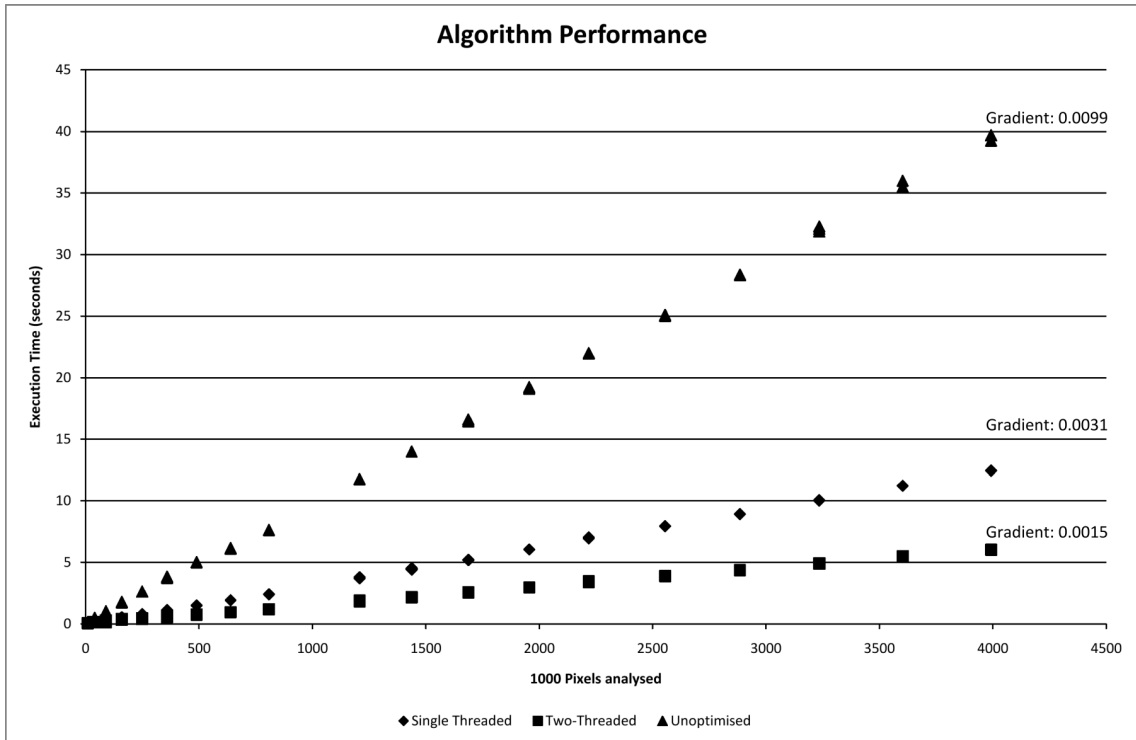


Figure 5.4: Radon Transform Performance

made by multi-threading the algorithm, as the algorithm's summations could be divided into mutually exclusive partitions. Multi-threading successfully reduced the execution time again by half. This is a significant performance improvement which will reduce the waiting time that end user's experience, improving the utility of the modules.

	Grid sizes
Sub-sampled	100x100, 200x200, 300x299, 400x399, 500x499, 600x699, 700x699, 800x799, 900x898
Super-sampled	1100x1098, 1200x1198, 1300x1298, 1400x1397, 1500x1479, 1600x1597, 1700x1697, 1800x1797, 1900x1896, 2000x1996

Table 5.3: Range of input images

## 5.3 Code Experiments

The framework designed in this project provides a solution to the three problems of verification, storage and maintenance which were discussed in section 4.2.

### 5.3.1 Verification

The verification problem was tackled by providing tool support which allows actual outputs from example code to be verified against a set of expected outputs. This provides a mechanism for ensuring that example code actually achieves the intentions of its author. Thus, code experiments capture knowledge in the form of code, and provide a means of verifying that the knowledge is correct.

An example of verification can be seen in Figure 5.5. This experiment demonstrates an unusual choice in naming: The method *OpenFromFile* does more than a developer would expect: It is used to create new raster workspaces as well as open existing ones. This behaviour is not documented in the ArcGIS API, yet is important from a developer's perspective. The experiment demonstrates this unexpected behaviour and verifies it by asserting that the ArcGIS libraries acknowledge that a raster workspace has been created in the specified directory.

### 5.3.2 Storage

The problem of executability was addressed by the creation of a specialised test harness. This tool, called *ArcMapUnit* is tasked with setting up the environment for Code Experiments, cataloguing experiments, and executing code experiments. The experimental environment is set up by controlling the execution of and communicating with ArcMap, the desktop component of ArcGIS. This ensures that experiments run in a consistent environment. The experiments can be run by on any computer that has ArcMapUnit installed. By cataloguing past Code Experiments, ArcMapUnit allows the example code to be stored in an executable state, and thus preserves its utility as a resource for future developers.

### 5.3.3 Maintenance

A useful product of using Code Experiments is that it creates knowledge regression test assets. For example, if a new revision of the ArcGIS API was released, Code Experiments could be re-run to detect any discrepancies between the behaviour of the old and new systems. Code Experiment failures would imply

```

[Test]
[Description("Demonstrates that a raster workspace is created using the OpenFromFile method.")]
[Tags("Raster", "Dataset")]
[Author("Dave Hng")]
public void CreateRasterWorkspace()
{
    // create a temporary directory and get its path
    TemporaryDirectory temporaryDirectory =
        new TemporaryDirectory();

    string directoryPath = temporaryDirectory.Path;

    // check that our temporary directory has been created
    Assert.IsTrue(Directory.Exists(directoryPath));

    // initialise workspace factory class
    RasterWorkspaceFactory rasterWorkspaceFactory =
        new RasterWorkspaceFactoryClass();
    IWorkspaceFactory workspaceFactory =
        COM.QueryInterface<IWorkspaceFactory>(rasterWorkspaceFactory);

    // create a new workspace
    IWorkspace workspace =
        workspaceFactory.OpenFromFile(directoryPath, 0);

    // verify that the workspace has been created
    Assert.IsTrue(workspaceFactory.IsWorkspace(directoryPath));
}

```

Figure 5.5: An experiment using verification.

changes in the new version of the ArcGIS API, indicating that the knowledge captured in the experiment, and hence the knowledge of the developer, need to be updated. This adds value to the software development process as any changes that need to be made are immediately visible and reduces the maintenance overhead required to restore the functionality of the system, as well as maintain the utility of the example code.

### 5.3.4 Discussion

Code Experiments can effectively document the interactions between systems, but only if the experiment code is readable. This can be difficult to achieve: the intentions of the programmer may not be immediately obvious from reading their code. A programmer writing Code Experiments must be careful to select appropriate abstractions for the reader and to be more explicit in their coding technique. This leads to developers being encouraged to write easily understood and readable code. This change in mindset promotes good programming practice by improving the readability and quality of code and is a highly desirable outcome

from the use of Code Experiments.

There are some situations where Code Experiments will not be able to provide a comprehensive, documented example of how to use a system. Two of these problems relate to verification of the expected outputs from the example code.

One example is when code executes outside of the scope of the experimental environment. The assertion framework used in this project assumes that there is a single point of entry into the code where all variables are initialised, and a single point of exit from the example code which is within the scope of the test. This latter assumption is not always true.

If the point of exit from the example code is outside of the scope of the experiment environment, it is not possible to verify the code's output with an expected output. An example of this is when an object is garbage collected. In the .NET Framework, there is special provision for code to execute when an object is garbage collected and this is called finalisation. As the experimental environment does not have full control over object garbage collection, code which makes use of finalisation cannot be verified.

A second verification problem is when the output of an experiment is graphical. Some of the experiments written in this project produce output which results in additional graphics, for example, a line segment, being overlaid on a geological map. Without a sophisticated image processing tool it is difficult to analyse the output image to confirm that the feature is now visible on the map. Considerations would also need to be made for differing screen resolutions, colour quality, font sizes and other user definable settings. Instead, the experiment checks other available evidence that indicates that the feature has been drawn. This is not ideal as it does not directly check that the actual output meets the experiment's specification, but it provides a sufficient compromise such that some verification can take place.

### 5.3.5 Experiments created

A subset of the experiments created in developing the Image Processing Tool are listed below.

**Create Raster Layer:** This experiment programmatically creates a raster on the local disk containing a gradient, composes it in a raster dataset and then adds the raster dataset to the currently open map. The experiment is verified by checking for the existence of a raster layer on the current map.

**Create Polygon Layer:** This experiment programmatically creates a shapefile which contains an enclosed polygon that is spatially referenced, and adds it

to the current map. The experiment highlights the captured knowledge that enclosed polygons must be constructed by successively adding points in a clockwise direction. This experiment is verified by enumerating the current map for vector layers and ensuring that one of the currently loaded layers contains the polygon that was created.

**Create Point Layer:** This experiment demonstrates creation of a layer containing the capital cities of Australia, spatially referenced by latitude and longitude using the WGS1984 coordinate system. Each capital city is represented by a point on the map. The experiment is verified by enumerating all points on the map and checking that each capital city exists and is positioned correctly by latitude and longitude.

**Get the Spatial Reference of a Raster:** Provides an example of how to enumerate through a raster's properties and find its spatial reference. This experiment does not have a verification test as it simply shows how to access a property of an object which is difficult to find.

**Set the Spatial Reference of a Raster:** An experiment which demonstrates how to set the spatial reference on a raster layer. Similar to the previous experiment, there is no verification test as its purpose is to highlight how spatial references are stored.

**Read Raster Data to Array:** Scans an existing raster and obtains a 2D array of double floating point types containing the raster's data. This experiment demonstrates the requirement to use both `IPixelBlock` and `IPixelBlock3` interfaces to obtain the raster data in a format that the .NET framework can understand. This experiment does not have a verification test as it is simply an example of how to read raster data.

**Save a Raster:** Shows that writing array data to a raster only affects the data in memory, and that a special interface must be `QueryInterfaced` to obtain a method which allows a developer to save a raster to disk.

These experiments capture knowledge that would be useful to any developer working on the Image Processing Framework in the ArcMap environment by providing a short explanation about how to achieve some commonly executed tasks.

## CHAPTER 6

# Summary and Future Development

## 6.1 Contributions

This project has produced an image processing framework which is embedded in a commercial GIS tool, ArcGIS. This framework facilitates the integration of image processing modules into ArcGIS, adding new functionality to the system. In addition to this, this project has also investigated and produced a useful knowledge capture system which will assist developers who seek to extend the image processing system.

The potential benefits of image processing in a GIS environment have been demonstrated in this project. Image processing modules have been built which demonstrate the value of the Radon and Hough transforms in extracting linear features from raster datasets. This has been accomplished by extending current research in optimising the Hough and Radon transforms for geoscientific data by adding normalisation, non-maximal suppression and sliding window techniques.

Assistance to developers seeking to author image processing modules or to extend the image processing framework has also been provided. By building on the concept that software is a medium for the codification of knowledge, and creating an environment in which example code can be created and refactored into *Code Experiments*. The Code Experiment process adds a test stage that verifies the outputs of the example code and provides a specialised test harness which allows Code Experiments to be executed. These two features add value by providing a means to verify, systematically execute and maintain example code.

This project provided a platform in which image processing modules can be embedded within GIS tools, and the results from processing rendered natively in an integrated geological analysis environment. It is envisioned that the framework developed in this project will be used as a foundation for the further investigation of the applicability of image processing within GIS.

## 6.2 Future Work

The applicability of image processing to geoscientific datasets within a GIS environment is a relatively new research area and there are many further topics which follow on from this project and extend the work already done.

Additional image processing modules could be researched and implemented. Techniques such as skeletonisation and contrast equalisation could be used to pre-process raster data prior to applying the Hough or Radon transforms. This would improve the robustness of the analysis technique used in this dissertation and allow it to be applied to other geoscientific datasets.

Another image processing problem which has not been addressed in this project is how to assemble the line segments which have been extracted from the linear extraction process. The linear features extracted in this project may potentially find small segments of longer features which were partially extracted. Thus, research into techniques for reassembling line segments is necessary in order to fully examine the structure of the linear feature.

The image processing framework could be extended to function in a larger GIS environment. This project has targeted ArcMap, a desktop GIS tool. The image processing framework could be extended to work with larger GIS software packages as well, such as ArcSDE which hosts raster and vector data in a dedicated spatial database. These systems allow multiple users to work and interact with a common dataset, improving the spread of relevant information in a company wide enterprise.

The software engineering concepts in this project could also be further researched. The effectiveness of Code Experiments could be compared against other forms of knowledge media such as traditional documentation and raw example code in an experimental setting to determine which was the most effective for developers furthering their knowledge of an existing system.

The experimental environment created in this project could also be backported to unit testing tools such as NUnit. NUnit has an established user base and could be used as the experiment environment for applications that do not have complex initialisation issues. Refactoring existing example code into code experiments adds value to the example code as it becomes verifiable, executable, and more maintainable. This will improve the utility of example code for users, and reduce maintenance costs for the example authors.

## APPENDIX A

# Original Honours Proposal

**Title:** Automated image analysis of Geographic Raster Data

**Author:** David Hng

**Supervisors:** Dr. Eun-Jung Holden, Dr. Rachel Cardell-Oliver

## Abstract

The development of Geographic Information Systems (GIS) has led to improved ability of geoscientists to view and analyse geographic information. Current GIS software such as ArcGIS and MapInfo include a well developed suite of vector analysis tools which can be applied to geographic data, but have a limited ability to analyse raster data. This thesis aims to produce a framework for automatic image analysis of geographic raster data within ArcGIS and to use this framework to identify linear anomalies in geophysical datasets. In addition to this, a system for producing executable documentation will be investigated and designed. This will allow for the production of test cases that document the image processing framework and its interaction with ArcGIS.

## Background

### Geological Information Systems

A Geographic Information System (GIS) is a means by which geographic data is aggregated, processed and transformed into geographic information. These systems provide a unified architecture and a standardised approach to managing geographic data [23]. GIS is used in many industries including health, cartography, criminology and mining for a variety of different purposes. This project focuses on the applications of GIS in mineral exploration.

There are several dominant software packages which provide GISes:

- ArcGIS from Environmental Systems Research Institute (ESRI)
- MapInfo Professional and MapXtreme from MapInfo
- MapGuide from Autodesk

Each of these companies produce a proprietary GIS with associated tools which can be customised to meet a user's needs. This project will focus on ESRI's ArcGIS software package which is widely used by major mining companies and geoscientists.

Due to increases in desktop computer processing power, GIS analysis tools have become more available for end users without the need for expensive hardware clusters to perform computational tasks [25]. It is now possible for a single computer to perform a relatively complex set of operations on geographical data to gather evidence about a particular set of features. These queries assist in providing decision support for users of the software.

GIS software segregates data into two broad categories: Raster data and Vector data. Raster data consists of rows and columns of cells where each cell stores a single value. Aerial photography of terrain under examination as possible sites for exploration are an example of raster data. Vector data is described as geometries: points, lines and polygons. Land ownership maps are an example of vector data.

In the field of mineral exploration, GIS is used to identify geological features which may indicate the presence of mineral deposits. A user may analyse a combination of aerial terrain, magnetic and radiometric data to search for major shear zones. These shear zones are conduits for mineralising fluids and suggest possible areas for further investigation. Analysis of these geological images by computer software can aid in the identification of these features.

Current GIS software provides rudimentary support for the automatic analysis of raster data such as image addition and contrast adjustments. To perform a more complicated analysis the raster data is exported from the GIS software into another analysis package (such as 'ERmapper' and Matlab) where the raw image data is processed. The results of the computation are then imported back into the GIS software and visualised alongside existing data. This process is not optimal: users must have a good understanding of both GIS and image processing software and commonly used actions cannot be scripted and replayed. This project seeks to address these issues.

## Executable Documentation

This project will also focus on software engineering practices, specifically providing executable documentation for the system that will be developed. It is likely that the software produced from this project will be used in further research in applying computer vision to GIS and so there is a need to provide both end-user documentation and documentation for developers who seek to extend the system.

Although prose is the most common form of documentation, it is not necessarily the most ideal method when providing documentation for developers. In most scenarios, example code with a short worded explanation is of greater value. This example and code approach has been used previously in a Java environment by Hoffman and Stropper [18], where documentation was written in a Frequently Asked Questions (FAQ) style. Test cases are selected that are interesting to people writing code. The test cases may detail commonly used classes and modules, they may demonstrate a notable piece of functionality, or may demonstrate a special case that would otherwise cause confusion. The test cases add value to the documentation of the system by providing concrete examples in code form that are immediately executable.

Hoffman and Stropper outline a number of benefits of executable documentation [18]. Executable documentation is precise, although partial: examples of valid input and output are given, but only for selected code segments. The documentation is consistent: it is accurate and contains no errors. It can be verified by programmers through execution, noting any errors that occur. The approach provides a form of quality assurance about the code, similar to that provided by unit testing [4].

Another application for executable documentation is to aid in sharing knowledge. If a programmer is writing code for an unfamiliar Application Programming Interface (API), they can write test cases in documentation to verify their understanding of the API. As the programmer increases their knowledge of how to use the API and the best practices associated with it, this knowledge is distilled in the test cases that they write. Tests as documentation provide a great utility to programmers as a means of capturing useful information in a form which can be easily understood.

## Aim

This project ultimately aims to develop a new framework for automatic image analysis within ArcGIS. This framework will allow a user to compose multiple

filters and transformations to detect and enhance features in geographic raster data. It will also include an architecture which will allow for additional filters to be added to a scripted sequence that is executed on the raster data.

In addition to this, an executable documentation system will be built and used to develop test cases for parts of the ArcGIS API used in the image processing framework. It will also be used to document the image processing framework itself.

The framework will then be used to create image filters which will be able to automatically identify linear anomalies in geophysical datasets (specifically aeromagnetic datasets). This area builds on previous research into the use of the Radon [33] or Hough [13] transformations to identify linear anomalies within raster datasets.

The end result of the project will be a module which will allow for the further application of image analysis within Geographic Information Systems, together with a set of test cases which will document the functionality of the framework.

## Project Timeline

<b>Task</b>	<b>Approximate Dates</b>
Research and Preparation of Proposal	February - March
Conduct literature review	March
Conduct risk analysis and feasibility	March
Implement basic image analysis filters	April - May
Implement executable documentation framework	May
Implement image analysis module in ArcGIS	June - July
Implement remainder of image analysis filters	July - August
Compile results for draft dissertation	August - September

## Software and Hardware Requirements

This project will require the use of Environmental Systems Research Institute's (ESRI) ArcGIS software suite as software library for the provision of geological raster data. The ArcGIS interface module will be written using C# and run on the Microsoft .NET framework and communicate with ArcGIS through COM. Individual image processing modules may also be written in C or C++ for performance reasons.

The executable documentation framework will also be written in C# and run on the Microsoft .NET framework.

# Bibliography

- [1] ARMOUR, P. *The Laws of Software Process: A New Model for the Production and Management of Software*. Auerbach Publications, 2004.
- [2] BECK, K. Simple Smalltalk Testing: With Patterns. *Smalltalk Report 4*, 2 (1994).
- [3] BECK, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.
- [4] BECK, K. *Test-Driven Development: By Example*. Addison-Wesley, 2006.
- [5] BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc. New York, NY, USA, 1996.
- [6] CAMPWOOD SOFTWARE. SourceMonitor Version 2.2. Available: <http://www.campwoodsw.com/sourcemonitor.html> [Online]. Accessed on October 10, 2007.
- [7] DUDA, R., AND HART, P. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM* 15, 1 (1972), 11–15.
- [8] DUDA, R., HART, P., ET AL. *Pattern classification and scene analysis*. Wiley New York, 1973.
- [9] DURRANI, T., AND BISSET, D. The Radon transform and its properties. *Geophysics* 49 (1984), 1180.
- [10] ENVIRONMENTAL SYSTEMS RESEARCH INSTITUTE. ArcGIS: The Complete Geographic Information System. Available: <http://www.esri.com/software/arcgis/> [Online]. Accessed on October 10, 2007.
- [11] ER MAPPER. ER Mapper Professional. Available: <http://www.ermapper.com/ProductView.aspx?t=27> [Online]. Accessed on October 10, 2007.
- [12] FAM, C., HOLDEN, E., DENTITH, M., AND KOVESI, P. Towards the Automated Mapping Linear Anomalies within Aeromagnetic Datasets. Tech. rep., iVEC, 2007.

- [13] FITTON, N., AND COX, S. Optimising the application of the Hough transform for automatic feature extraction from geoscientific images. *Computers and Geosciences* 24, 10 (1998), 933–951.
- [14] FORWARD, A., AND LETHBRIDGE, T. The relevance of software documentation, tools and technologies: a survey. *Proceedings of the 2002 ACM symposium on Document engineering* (2002), 26–33.
- [15] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.
- [16] GONZALEZ, R., WOODS, R., AND EDDINS, S. *Digital image processing using MATLAB*. Prentice Hall, 2004.
- [17] HENKEL, H., AND GUZMAN, M. Magnetic features of fracture zones. *Geoplotation* 15, 3 (1977), 173–181.
- [18] HOFFMAN, D., AND STROOPER, P. API documentation with executable examples. *The Journal of Systems & Software* 66, 2 (2003), 143–156.
- [19] HOFFMAN, D., STROOPER, P., AND WILKIN, S. Tool support for executable documentation of Java class hierarchies. *Software Testing, Verification and Reliability* 15, 4 (2005), 235–256.
- [20] JEFFRIES, R., ANDERSON, A., AND HENDRICKSON, C. *Extreme Programming Installed*. Addison-Wesley Professional, 2000.
- [21] JONES, C. *Geographical information systems and computer cartography*. Longman Harlow, England, 1997.
- [22] KARNEILI, A., MEISELS, A., FISHER, L., AND ARKIN, Y. Automatic extraction and evaluation of geological linear features from digital remote sensing data using a hough transform. *Photogrammetric engineering and remote sensing* 62, 5 (1996), 525–531.
- [23] LONGLEY, P. *Geographical Information Systems and Science*. Wiley, 2005.
- [24] MAPINFO. MapInfo Professional. Available: <http://extranet.mapinfo.com/products/Overview.cfm?productid=1044&productcategoryid=1> [Online]. Accessed on October 10, 2007.
- [25] MORAIN, S., AND S. L. BAROS, E. *Raster Imagery in Geographic Information Systems*. Onword Press, 1996.

- [26] MYERS, G. *The Art of Software Testing*. John Wiley and Sons, 2004.
- [27] NEWKIRK, J., TWO, M., VORONTSOV, A., CRAIG, P., AND POOLE, C. NUnit. Available: <http://www.nunit.org> [Online]. Accessed on October 10, 2007.
- [28] NEWKIRK, J., AND VORONTSOV, A. *Test-Driven Development in Microsoft. Net*. Microsoft Press Redmond, WA, USA, 2004.
- [29] PARASNIS, D. *Principles of Applied Geophysics*. Kluwer Academic Publishers, 1997.
- [30] PARKER, J. *Algorithms for Image Processing and Computer Vision*. John Wiley & Sons, Inc. New York, NY, USA, 1996.
- [31] PAWLOWSKI, R. Short Note Use of the slant stack for geologic or geophysical map lineament analysis. *Geophysics* 62, 6 (1997).
- [32] SUN MICROSYSTEMS. Javadoc. Available: <http://java.sun.com/j2se/javadoc/> [Online]. Accessed on October 10, 2007.
- [33] SYKES, M., AND DAS, U. Directional filtering for linear feature enhancement in geophysical maps. *Geophysics* 65 (2000), 1758.
- [34] VAN GINKEL, M., HENDRIKS, C., AND VAN VLIET, L. A short introduction to the Radon and Hough transforms and how they relate to each other. Tech. rep., Technical Report QI-2004-01, Quantitative Imaging Group, Delft University of Technology, 2004.
- [35] WATT, A., AND WATT, M. *Advanced animation and rendering techniques*. ACM Press New York, NY, USA, 1991.
- [36] ZHANG, L., WU, J., HAO, T., AND WANG, J. Automatic lineament extraction from potential-field images using the Radon transform and gradient calculation. *Geophysics* 71 (2006), J31.