2001

Video compression using wavelets and hierarchical motion estimation

Andrew Peter Byrne

Edith Cowan University

Recommended Citation

This Thesis is posted at Research Online.
https://ro.ecu.edu.au/theses_hons/343
Edith Cowan University

Copyright Warning

You may print or download ONE copy of this document for the purpose of your own research or study.

The University does not authorize you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site.

You are reminded of the following:

- Copyright owners are entitled to take legal action against persons who infringe their copyright.

- A reproduction of material that is protected by copyright may be a copyright infringement. Where the reproduction of such material is done without attribution of authorship, with false attribution of authorship or the authorship is treated in a derogatory manner, this may be a breach of the author’s moral rights contained in Part IX of the Copyright Act 1968 (Cth).

- Courts have the power to impose a wide range of civil and criminal sanctions for infringement of copyright, infringement of moral rights and other offences under the Copyright Act 1968 (Cth). Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.
Video Compression Using Wavelets and Hierarchical Motion Estimation

Andrew Peter Byrne

Bachelor of Engineering (Computer Systems)

Faculty of Communication, Health and Science

Edith Cowan University
Abstract

This thesis investigates the benefits and the significant compression that can be obtained from data that has been decomposed using a wavelet transform. A video compression algorithm was developed that employs the wavelet transform and a hierarchical motion estimation algorithm which itself utilises benefits of the wavelet transform.

Using MATLAB, a popular software tool for matrix based computation and analysis, several functions were developed which together formed the video compression algorithm. A variety of tests were conducted on a sample video sequence to ascertain the strengths and weaknesses of the techniques employed. The results, although not the same as the main comparison (MPEG-1), show that the wavelet transform does have a huge potential for application in the area of video compression. The quality of the output also has advantages over DCT (Discrete Cosine Transform) based compression algorithms.

One of the main outcomes is an awareness that the wavelet transform and the hierarchical motion estimation algorithm do provide significant compressibility, however the overheads need to be reduced as much as possible to ensure viability.

There is still plenty of room for improvement and enhancement of the results achieved in this thesis. The potential benefits of wavelet based video compression are endless. With this being the early stages of the Information Era, compression (in particular video) is becoming increasingly important on a daily basis.
I certify that this thesis does not, to the best of my knowledge and belief:

(i) incorporate without acknowledgment any material previously submitted for a degree or diploma in any institution of higher education;

(ii) contain any material previously published or written by another person except where due reference is made in the text; or

(iii) contain any defamatory material.

Signed: __________________________

Date: 6/2/2001
Acknowledgments

I would like to sincerely thank my supervisor, Associate Professor Abdesselam Bouzerdoum, for his support and guidance throughout the year. He was also generous enough to lend me several of his books and journals that greatly assisted me whilst researching and writing this thesis.

Thank you also to Dr Douglas Chai who provided me with a few excellent Internet references from which I was able to find many sources of valuable information.

Finally, I would like to acknowledge all the lecturers who have taught me over the past five years. This thesis is the culmination of everything I have learned, whether it is directly related to this topic, or other fields that have given me the ability to apply myself to this type of research.

Once again, thank you to everyone at Edith Cowan University.
# Table of Contents

**ABSTRACT** ..............................................................................................................................................2

**ACKNOWLEDGMENTS** .................................................................................................................................4

**INDEX OF FIGURES AND TABLES** .............................................................................................................7

**ABBREVIATIONS** ...........................................................................................................................................8

**CHAPTER 1 - AN INTRODUCTION TO IMAGE AND VIDEO COMPRESSION** .................................................9

1.1 **INTRODUCTION** ........................................................................................................................................9
1.2 **COMPRESSION** .........................................................................................................................................10
1.3 **A BRIEF HISTORICAL OVERVIEW** ........................................................................................................11
  1.3.1 **Image Compression** ..........................................................................................................................11
  1.3.2 **Video Compression** ..........................................................................................................................11
1.4 **TRANSFORM CODING** ...........................................................................................................................12
1.5 **THE WAVELET TRANSFORM** ..................................................................................................................13
1.6 **TECHNIQUES USED IN VIDEO COMPRESSION** .....................................................................................14
  1.6.1 **Motion Estimation** ...........................................................................................................................15
  1.6.2 **Motion Compensation** ......................................................................................................................15
1.7 **AN OUTLINE OF THIS THESIS AND ITS OBJECTIVES** ............................................................................16

**CHAPTER 2 - AN IN-DEPTH LOOK AT IMAGE AND VIDEO COMPRESSION TECHNIQUES** .......................18

2.1 **IMAGE COMPRESSION** ............................................................................................................................18
  2.1.1 **The Wavelet Transform** ....................................................................................................................18
    2.1.1.1 Octave Bandpass Filtering .......................................................................................................20
    2.1.1.2 Multiresolution Analysis ...........................................................................................................23
    2.1.1.3 The Haar Wavelet .......................................................................................................................24
  2.1.2 **Quantisation** .....................................................................................................................................24
  2.1.3 **Entropy Encoding Techniques** ........................................................................................................27
    2.1.3.1 Run Length Encoding ...............................................................................................................27
    2.1.3.2 Huffman Coding .......................................................................................................................28
    2.1.3.3 A Summary of the Encoding Techniques ..................................................................................30
2.2 **VIDEO COMPRESSION** ............................................................................................................................31
  2.2.1 **Motion Estimation** ..........................................................................................................................31
  2.2.2 **Motion Compensation** ....................................................................................................................34
2.3 **PUTTING IT ALL TOGETHER - THE CODEC** .....................................................................................35
  2.3.1 **Encoding** .........................................................................................................................................35
  2.3.2 **Decoding** .........................................................................................................................................36

**CHAPTER 3 - IMPLEMENTATION USING MATLAB** .......................................................................................37

3.1 **CHOOSING MATLAB** .............................................................................................................................37
3.2 **STRUCTURE OF THE ENCODING ALGORITHM** ...................................................................................37
3.3 **THE FUNCTIONS, PROCEDURES AND SCRIPTS** ....................................................................................40
  3.3.1 **initialise** .........................................................................................................................................40
  3.3.2 **convertimage** ....................................................................................................................................41
  3.3.3 **wavedec2 (internal function)** .........................................................................................................41
  3.3.4 **quantiser** .........................................................................................................................................41
  3.3.5 **quantise** ...........................................................................................................................................42
  3.3.6 **compress** ..........................................................................................................................................43
  3.3.7 **huffman_compress (external function)** ..........................................................................................44
  3.3.8 **motion_est** .......................................................................................................................................44
  3.3.9 **minblock** ..........................................................................................................................................44
  3.3.10 **minmae** .........................................................................................................................................45
  3.3.11 **decompress** ....................................................................................................................................46
Index of Figures and Tables

FIGURE 1.1 - (A) ORIGINAL IMAGE; (B) 20:1 COMPRESSION USING WAVELETS; ........................................ 14
(c) 176:1 JPEG (DCT) COMPRESSION (MAXIMUM POSSIBLE); IMAGES FROM (TOPIWALA, 1998) .... 14
FIGURE 2.1 - A CASCaded OCTAVE BANPASS FILTER; BASED ON A SIMILAR FIGURE FROM (BURGESS, 2000) ................................................................. 20
FIGURE 2.2 - A REPRESENTATION OF THE EFFECT OF THE WAVELET TRANSFORM .............................. 21
FIGURE 2.3 - A TWO-DIMENSIONAL OCTAVE-BAND DECOMPOSITION USING TWO-CHANNEL PERFECT RECONSTRUCTION FILTERS; BASED ON A SIMILAR FIGURE FROM (TOPIWALA, 1998) .................. 22
FIGURE 2.4 - A TWO-LEVEL SUBBAND DECOMPOSITION OF THE LENA IMAGE, WITH HISTOGRAMS OF THE SUBBANDS; IMAGE FROM (TOPIWALA, 1998) .................................................. 23
FIGURE 2.5 - THE ELEMENTARY HAAR SCALING FUNCTION AND WAVELET; IMAGE FROM (TOPIWALA, 1998) .................................................................................................................. 24
FIGURE 2.6 - A VISUAL EXAMPLE OF HOW VALUES ARE QUANTISED USING A UNIFORM STEPSIZE AND A DEADZONE ........................................................................................................ 26
TABLE 2.1 - PROBABILITIES OF VALUES IN A DATA SET ........................................................................ 29
FIGURE 2.7 - THE HUFFMAN CODEWORD GENERATION PROCESS ........................................................... 29
TABLE 2.2 - THE ASSIGNMENT OF HUFFMAN CODEWORDS TO VALUES .................................................. 30
FIGURE 2.8 - A GENERIC HIERARCHICAL MOTION VECTOR SEARCH STRATEGY; IMAGE FROM (BHASKARAN & KONSTANTINIDES, 1995) ................................................................................. 33
FIGURE 3.1 - FLOW DIAGRAM OF MAIN VIDEO COMPRESSION ALGORITHM ........................................ 39
TABLE 4.1 - PARAMETERS FOR TESTS IN SECTION 4.2.3 ........................................................................ 53
TABLE 4.2 - PSNR RESULTS FOR TESTS IN SECTION 4.2.3 ................................................................. 53
TABLE 4.3 - SUMMARY OF TEST RESULTS IN SECTION 4.2.3 .............................................................. 53
TABLE 4.4 - PARAMETERS FOR TESTS IN SECTION 4.2.4 ........................................................................ 54
TABLE 4.5 - PSNR RESULTS FOR TESTS IN SECTION 4.2.4 ................................................................. 54
TABLE 4.6 - SUMMARY OF TEST RESULTS IN SECTION 4.2.4 .............................................................. 54
TABLE 4.7 - PARAMETERS FOR TESTS IN SECTION 4.2.5 ........................................................................ 55
TABLE 4.8 - PSNR RESULTS FOR TESTS IN SECTION 4.2.5 ................................................................. 55
TABLE 4.9 - SUMMARY OF TEST RESULTS IN SECTION 4.2.5 .............................................................. 55
FIGURE A.1 - ORIGINAL UNCOMPRESSED FRAMES .............................................................................. 65
FIGURE A.2.1 - DECOMPRESSED FRAMES FROM TEST 1 IN SECTION 4.2.3 ........................................ 66
FIGURE A.2.2 - DECOMPRESSED FRAMES FROM TEST 2 IN SECTION 4.2.3 ........................................ 67
FIGURE A.2.3 - DECOMPRESSED FRAMES FROM TEST 3 IN SECTION 4.2.3 ........................................ 68
FIGURE A.3.1 - DECOMPRESSED FRAMES FROM TEST 1 IN SECTION 4.2.4 .......................................... 69
FIGURE A.3.2 - DECOMPRESSED FRAMES FROM TEST 2 IN SECTION 4.2.4 .......................................... 70
FIGURE A.3.3 - DECOMPRESSED FRAMES FROM TEST 3 IN SECTION 4.2.4 .......................................... 71
FIGURE A.3.4 - DECOMPRESSED FRAMES FROM TEST 4 IN SECTION 4.2.4 .......................................... 72
FIGURE A.4.1 - DECOMPRESSED FRAMES FROM TEST 1 IN SECTION 4.2.5 .......................................... 73
FIGURE A.4.2 - DECOMPRESSED FRAMES FROM TEST 2 IN SECTION 4.2.5 .......................................... 74
FIGURE A.4.3 - DECOMPRESSED FRAMES FROM TEST 3 IN SECTION 4.2.5 .......................................... 75
FIGURE A.4.4 - DECOMPRESSED FRAMES FROM TEST 4 IN SECTION 4.2.5 .......................................... 76
Abbreviations

bpp - bits per pixel
dB - decibels
C - current frame
DCT - discrete cosine transform
fps - frames per second
JPEG - Joint Photographic Experts Group
kB - kilobytes
M - width of macroblock
MAE - mean absolute error
MB - megabytes
Mbps - megabits per second
MPEG- - Moving Pictures Expert Group
MSE - mean squared error
N - height of macroblock
p - search parameter
PSNR - peak signal-to-noise ratio
R - reference frame
RGB - red green blue
RLE - run length encoding
Chapter 1 - An Introduction to Image and Video Compression

1.1 Introduction

Since the future seems destined to revolve around information, and the real-time access to that information, society is going to demand more and more to be delivered in shorter time spans. This information is diverging from the conventional textual form, into rich, colourful images and video sequences.

With the advent of the Internet, there has been an explosion in the number of people around the world who have taken to it as one of their major forms of communication. Where previously most people would be happy with sending text based emails, it is now very common to see emails with voice and video integrated into them. Even some telephone calls and video conferencing are now transmitted over the Internet.

The current problem is with the amount of information contained in images and video sequences, as opposed to simple text based data. The old saying that a picture is worth one thousand words is quite true in this case, if not an understatement.

One might think that a simple solution would be to increase the capacity of the communications links that join the millions of networks of the Internet together. Despite recent advances in this field, the expansion in the numbers of people getting online and their demand for more and more information, has meant that
finding another way to deliver more data to more people, faster, has been the top priority. This is where the idea of compression enters the equation.

1.2 Compression

Compression is not new, however due to the increasing requirements of the Internet, more efficient and faster compression algorithms are required. Previously, some compressed video sequences were of such poor quality, that most people hardly bothered to utilise them. This is the number one requirement for compression algorithms at the moment – compress the data at a better rate, yet maintain a high level of quality.

In general, compression involves a number of operations to achieve the final output. It can be lossless, where the original data can be reconstructed with 100% accuracy from the compressed data, or it can be lossy, where there is some degree of error in the compressed image. The vast majority of compression algorithms devised today suffer from some degree of loss. It is not possible to obtain the desired result of higher compression and better quality without creating a less than perfectly reconstructed piece of data. The aim therefore, is to be able to remove as much redundant information as possible from the data.

Taking the compression of Images as an example – it is possible to remove irrelevant data from the image such as colours or patterns beyond the capability of the human eyes that will be viewing the compressed image. This means that
as far as human eyes are concerned, the image is perfect, yet it has still been possible to apply compression.

1.3 A Brief Historical Overview

1.3.1 Image Compression

In terms of image compression, one of the most popular algorithms is the JPEG standard. It was developed between 1988 and 1990 with the draft international standard released in 1991, followed by the final international standard in 1992. According to (Bhaskaran & Konstantinides, 1995) the requirements for JPEG as it was being devised included the following:

- Be as close as possible to the state of the art in image compression.
- Allow applications (or a user) to tradeoff easily between desired compression and image quality.
- Work independently of the image type. That is, the method should not be restricted by the type of image source, image content, colour spaces, dimensions, pixel resolution, etc.
- Have modest computational complexity that would allow software-only implementations even on low-end computers. Low-complexity hardware implementations should also be feasible.
- Allow both sequential (single scan) and progressive coding (multiple scans).
- Offer the option for hierarchical encoding, in which a low-resolution version of the image can be accessed without a need to decompress the image at full resolution. (pp. 129-130)

1.3.2 Video Compression

The most common standard for video compression is the MPEG-x range of standards (such as MPEG-1 and MPEG-2). These standards define the way to compress and encode a video sequence using a combination of techniques. Without going into too much detail (as these concepts will be discussed later),
the MPEG-x standards compress images in a similar way to the JPEG standard for images. MPEG also makes use of motion estimation and motion compensation algorithms to reduce the amount of data that needs to be encoded. It also uses prediction algorithms and encodes the prediction errors, in order to further reduce the volume of data.

### 1.4 Transform Coding

Both the image and video compression algorithms above employ a technique known as Transform Coding, in particular, the Discrete Cosine Transform.

Transform-based coding algorithms have three major stages:

1. **The transform** – which changes the data into a form more likely to result in better compression rates
2. **Quantisation** – which rounds the data to a value from a fixed set of values
3. **Entropy coding** – which encodes the data into its final form

The first step doesn’t usually result in a smaller amount of data, but is important in ensuring the following steps yield high compression. The second step is where loss is introduced in the form of rounding. The purpose of quantisation is to reduce the number of possible values to which each point in the data can be assigned. In other words, rather than infinite possibilities, a predefined number of possibilities will exist. The third step removes any redundancy in the data and encodes it in a suitable form for storage and/or transmission.
1.5 The Wavelet Transform

There is another transform that has been around for quite some time, but has only become more popular in the past decade. The Wavelet Transform has several properties that make it very suitable for image and video compression. It is just one of many ways to rearrange data into another more compressible form. The current indication is that wavelet transforms yield much better compression ratios, with less noticeable error in the output, than compression algorithms employing non-wavelet transform techniques (Topiwala, 1998). This is aligned with the goal to compress data more, yet retain a high level of quality. Figure 1.1 shows a comparison between DCT and Wavelet compression.
Figure 1.1 – (a) Original image; (b) 200:1 compression using wavelets; (c) 176:1 JPEG (DCT) compression (maximum possible); Images from (Topiwala, 1998)

1.6 Techniques Used in Video Compression

A simple definition of video is a stream of images displayed fast enough in succession that the human eye processes them as fluent motion. Rates of over 10 images (or frames as they are referred to) per second are required in order for the eye to view the images as a continuous stream instead of snap shots. The most common rates are generally 15, 24 or 30 frames per second.
Putting this into perspective, in one second of video there is usually very little motion (of course this depends on the type of images in the sequence). Therefore, dividing each second into 15 frames usually results in images that are not very different from one another.

1.6.1 Motion Estimation

Using the above information, the concept of motion estimation takes form. By comparing two sequential images and defining how the objects within them have moved, a significant amount of compression can be achieved.

The image is usually divided into equal sized blocks, and each block in the second image is matched up with the best matching block from the first image. A set of motion vectors is obtained from these calculations.

1.6.2 Motion Compensation

The motion vectors can be used to compensate for the motion from one frame to the next. This is achieved by generating the next image from blocks in the previous image, as defined by the motion vectors. The result is never usually 100% accurate, but is a reasonably accurate prediction.

There are many ways to implement the motion estimation and motion compensation algorithms. Some use more processing power to get the motion vectors refined enough such that the compensated second image is virtually
identical to the actual. Others use an average motion estimation algorithm to obtain an approximate second image, and then encode the difference between the compensated and actual images into the video stream with the motion vectors.

1.7 An Outline of this Thesis and its Objectives

The purpose of this thesis is to design and implement a video compression algorithm that utilises the wavelet transform as its most fundamental component. Hierarchical motion estimation and compensation algorithms will also be designed and implemented and their impact analysed.

Since the use of wavelet transforms in this field is still not as widespread as the discrete cosine transform, the results will aim to prove that they are worthy of the attention that the DCT has received in the past decade.

The primary objectives of this thesis are:

- To investigate the wavelet transform and its use in image and video compression, and
- To implement a hierarchical motion estimation and compensation algorithm, and
- To combine the wavelet transform, a quantiser, and encoder to obtain a video compression encoding and decoding scheme.
The rest of this thesis is organised as follows:

Chapter 2 provides in-depth information about the theory behind all the techniques being used and investigated in this thesis. It gives an insight into the three phases of image compression (transform, quantisation, encoding) and also discusses the theory behind motion estimation and compensation.

Chapter 3 explains the implementation of the video compression algorithm in MATLAB, and provides information about how each function and procedure works towards achieving the final result.

Chapter 4 contains the numerical results of each test that the video compression algorithm and its variants generated. These results are analysed and discussed, and conclusions drawn from them.

Chapter 5 outlines the overall achievement of the work completed, and suggests areas of further refinement in future work.

The Appendices contain important information including all the original and decompressed frames from the video used in the tests, as well as the full printouts of the MATLAB functions, procedures and scripts which were used to achieve the results.
2.1 Image Compression

2.1.1 The Wavelet Transform

A common way to analyse signals is to represent them as a weighted sum of a number of simple basis functions: (the following formulae are from (Hilton, Jawerth, & Sengupta, 1994))

\[ f(x) = \sum c_i \Psi_i(x) \]

where

\( c_i \) are the coefficients (also known as weights) and \( \Psi_i(x) \) are the basis functions.

The basis functions are fixed, and hence it is the coefficients that carry all the information about the signal. Choosing translations of the impulse function for the bases only gives information about the signal in the time domain. Likewise, sinusoidal bases reveal information about the frequency domain of the signal only.

Since the application is compression, neither of the above are acceptable. Information about the signal's behaviour in both domains is required.
Unfortunately, there is a trade-off between the two as demonstrated by the Heisenberg inequality:

\[ \lambda \Delta \omega \geq \frac{1}{2} \]

That is, increased resolution in time gives decreased frequency resolution, and vice versa. In image compression however, low frequencies are usually not localised in time, and higher frequencies are usually very concentrated in time. This leads to a compromise whereby the basis functions form a set with each member providing support for different finite widths. That is, some support a wide region of the signal to resolve low frequency details accurately, whilst others support a narrow region of the signal to resolve high frequency (and therefore time) details accurately.

The basis functions are further simplified to being scaled and translated versions of one prototype function, known as the mother wavelet. By choosing a scaling factor of integer powers of 2, the end result is in the same form as a cascaded octave bandpass filter. This is shown by the formula

\[ \Psi(2^v x) \]

where \( v \) is an integer in a finite range.

The translation is chosen to make the basis function cover the entire signal, by considering all the integral shifts of \( \Psi \):

\[ \Psi(2^v x - k), k \in \mathbb{Z} \]
By putting all this together, the wavelet decomposition of the signal is given by

\[ f(x) = \sum_{r \text{time}} \sum_{t \text{time}} c_{r,t} \psi_{r,t}(x) \]

where

\[ \psi_{r,t}(x) = 2^{r/2} \psi(2^r x - t) \]

The factor in front of the mother wavelet is required to make the bases orthonormal. The coefficients are calculated from the wavelet transform which is the inner product of \( f(x) \) with the basis functions.

### 2.1.1.1 Octave Bandpass Filtering

Since wavelet transformation can be thought of and implemented as octave bandpass filtering, this is usually the easiest method for studying and applying the wavelet transform to signals.

![A cascaded octave bandpass filter](image)

*Figure 2.1 – A cascaded octave bandpass filter; Based on a similar figure from (Burgess, 2000)*
The output of each filter is downsampled by a factor of two in order to keep the original amount of data. (When the filtered data is synthesised to obtain the original data, the values have zeroes inserted between them to prepare for being recombined)

Figure 2.2 shows exactly which frequencies are being filtered by the above filter sequence.

![Diagram](Image)

*Figure 2.2 – A representation of the effect of the wavelet transform*

The lower frequency can be further split for as many decomposition levels as deemed necessary in the application. Image compression usually goes as far as 4 to 5 levels of decomposition.
Since images are two dimensional, the above process is applied to both the horizontal and vertical data, in exactly the same way as for a one-dimensional signal.

A common visual representation of the decomposition of a two-dimensional signal is shown in Figure 2.3.

![Figure 2.3 - A two-dimensional octave-band decomposition using two-channel perfect reconstruction filters; based on a similar figure from (Topiwala, 1998)](image)

The "LL" quadrant is the one that is further decomposed in subsequent levels.

Figure 2.4 shows an example of the above decomposition, as well as histograms showing the range of coefficients remaining after the transformation and their frequency.
2.1.1.2 Multiresolution Analysis

This idea of dividing an image into octaves of frequency is referred to as a multiresolution technique. One of the other advantages of this, especially when the final bitstream is going to be transmitted over a communication link, is that the higher frequency bands do not have to be transmitted if there is not enough bandwidth available. The only loss in not transmitting the higher frequencies is that of resolution. The more frequency bands that are transmitted the higher the resolution of the final image.

Another benefit of this is in the application of generating preview images whereby the entire image does not need to be decoded in order to get an acceptable representation of the image.
2.1.1.3 The Haar Wavelet

One of the most basic type of wavelets is the Haar wavelet. It is considered elementary and is shown in Figure 2.5 below.

![The elementary Haar scaling function and wavelet](image)

Figure 2.5 – The elementary Haar scaling function and wavelet; Image from *(Topiwala, 1998)*

This wavelet type will be used throughout the thesis, in order to keep the gathering and analysis of results standard.

2.1.2 Quantisation

Once an image has been decomposed using a wavelet transform into a number of subbands, the remaining information needs to be quantised. This process maps the infinite range of values that may exist in the transformed data, into a finite set of values which can be more compactly represented.

For example, if the values range from 0 through to 1000 and are not all integers, there are an infinite number of values that can exist in this range. Rounding is the most basic form of quantisation, where once rounded, the number of values
becomes finite. The problem with this form of quantisation is that it may not be sufficient to represent the data in its most compact form.

The values of the data after wavelet transformation consist of a large number of zeros, as well as a number of high valued coefficients, and a smaller proportion of medium ranged coefficients. A suitable technique for quantising this type of data is known as uniform quantisation, with the additional use of a deadzone.

The deadzone is a region around the origin in which all values are quantised to zero. Due to the nature of a wavelet transformed image, these values can be quantised to zero without any significant loss in the visual quality of the output. In most cases, these small values tend to represent only noise and are hence insignificant. The deadzone is usually twice the width of the step size, but can be any value below which all values should be quantised to zero.

The remainder of the values are quantised after a step size is chosen. The value of the step size determines the total number of bits that the values will be represented by once quantised. The trade-off is between the number of bits, and the quality of the output.
Another important aspect to consider is whether the same step size and deadzone should be used in every subband, or selected independently based on analysis of the data.

The lowest subband always contains the majority of the information of the original image, so care must be taken to quantise this subband with minimal loss. The other subbands contain much less information, however they carry valuable information that defines edges in the image, and enhances the overall resolution and quality of the image.

Based on this information, the deadzone in the higher frequency subbands can be made significantly larger.

An algorithm could be implemented that calculates the step size and deadzone for each subband based on a specified target bit rate. Usually, a target bit rate of 0.5 bits per pixel is used for 8 bit grayscale images. This effectively produces
a 16:1 compression ratio. The algorithm also has to take into account the compression achieved by the entropy encoding that will be used after quantisation.

2.1.3 Entropy Encoding Techniques

After quantisation, there is still a significant amount of compression that can be achieved by removing the redundancy in the data. Due to the large proportion of zero values, an encoding technique that can represent these values in the smallest possible manner is required.

Two examples of encoding techniques that are suitable for image compression are run length encoding and huffman coding.

2.1.3.1 Run Length Encoding

Run length encoding works by converting long strings of the same value into a smaller amount of information including:

- A marker representing the start of a run
- The number of occurrences of the value
- The value itself

Since run length encoding would provide best results when there are large runs of zero values, it is very effective in compressing wavelet transformed images.
A modified form of run length encoding is one whereby the data is transformed into two sets of information. The first is an array of all the non-zero values from the data, and the second is the index of each of these non-zero values. This provides for less overheads than standard RLE, and is very suitable for wavelet transformed data due to the large overall proportion of zeros.

Since the index array would not contain any duplicate values, a way to improve the results of further compression on it would be to represent each value in the array as the difference between itself and the previous value. For example, the original index \{1 2 4 7 8 9\} would be represented as the difference index \{1 1 2 3 1 1\}.

As can be seen, the first value is the same as the original, but the rest are obtained by sequentially adding their value to the previous values. There are now duplicate values that can be further compressed using techniques such as Huffman Coding.

### 2.1.3.2 Huffman Coding

Huffman coding is a popular technique that works by examining the data and assigning codewords to values according to the number of times they occur. The most frequently occurring values are assigned codewords of a smaller length (in bits) than the less frequently occurring values.
A codeword is a string of bits that, when concatenated to the other codewords in the final coded bitstream, will not produce any ambiguity.

Huffman coding is best explained using an example.

<table>
<thead>
<tr>
<th>Value</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.7</td>
</tr>
<tr>
<td>2</td>
<td>0.05</td>
</tr>
<tr>
<td>3</td>
<td>0.15</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
</tr>
</tbody>
</table>

As can be seen from Table 2.1, there are four possible values in a set of quantised data that have different probabilities of occurring in the data. A tree is created by taking the two values with the lowest probabilities (in brackets) and building the structure as shown below in Figure 2.7.

![Figure 2.7 - The Huffman codeword generation process](image)

The codewords are created by reading the binary values (bold) in the tree from bottom to top. The codewords for this example are shown in Table 2.2.
Table 2.2 – The assignment of huffman codewords to values

<table>
<thead>
<tr>
<th>Value</th>
<th>Probability</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.7</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.05</td>
<td>000</td>
</tr>
<tr>
<td>3</td>
<td>0.15</td>
<td>01</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
<td>001</td>
</tr>
</tbody>
</table>

Before huffman coding, the data would have required 3 bits to encode the values. Therefore the average number of bits per symbol is 3.

After huffman coding, the average number of bits per symbol is:

\[(0.7 \times 1) + (0.05 \times 3) + (0.15 \times 2) + (0.1 \times 3) = 1.45\]

This is a saving of over 50%. Of course the dictionary that maps the codewords to the actual values must be transmitted along with the data stream, however when there is a significant amount of data being transmitted with the same codewords, its impact is minimal.

2.1.3.3 A Summary of the Encoding Techniques

The modified technique of run length encoding is suitable for the encoding of wavelet transformed images due to the large percentage of zero values. The encoding would then be followed by huffman coding, to further compress the data, especially the index array.

Although the lowest frequency subband would not benefit from RLE as much as the higher frequency subbands due to the relative proportions of zeros, RLE is used on the entire image as a whole, followed by Huffman coding. However
there is potential to perform the RLE only on the higher frequency subbands, in order to reduce overheads.

2.2 Video Compression

2.2.1 Motion Estimation

It may sound simple, but determining where each pixel or block has moved when comparing one image to the next, is quite complicated.

Take for example an image of size 320 pixels wide by 240 pixels high. Dividing this into blocks of 8x8 (MxN) pixels, a total of 40x30 = 1200 blocks is obtained.

If one also considers the fact that each block in the current image (C) is not necessarily being compared with the blocks defined in the reference image (R), but rather with every possible 8x8 block of pixels located anywhere in the image, the computation involved is too large to be worthwhile.

In motion estimation, a parameter known as the search parameter is defined. It is denoted in formulae as \( p \). This value defines a region of pixels around the top-left corner of the block that will be searched for a best match. The higher the rate of motion between each frame, the larger this search parameter must be. If not defined according to the rate of motion, the best match will not necessarily be close to the actual block.
Now that we know which regions of the first image we are looking at to find the best match for each block in the second image, a way of determining which block is the best match must be defined. A common method is known as Minimum Mean Absolute Error or MAE, which is defined by (Bhaskaran & Konstantinides, 1995) as follows:

$$\text{MAE}(i, j) = \frac{1}{MN} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} |C(x+k, y+l) - R(x+i+k, y+j+l)|$$

where

$$-p \leq i \leq p \text{ and } -p \leq j \leq p$$

An alternative known as the Minimum Mean Squared Error (MSE) can also be used. The MSE performs the same operation as the MAE, except it squares the difference before the sum is calculated. It adds to the computation required, but may produce slightly better results.

For each block, the number of calculations required is very large. The MAE would need to be calculated for the square of $(2p + 1)$ blocks. Considering each MAE calculation contains an absolute value and subtraction, performed for each pixel $(MxN = 8x8 = 64)$, and then summed, it is easy to see why the number of calculations required is substantial.

Clearly this is not the most efficient way to produce the motion vectors. A technique known as hierarchical motion estimation, described in (Bhaskaran & Konstantinides, 1995) is becoming increasingly popular. This works on the assumption that if a lower resolution version of an image is put through the
motion estimation algorithm above, an approximate set of motion vectors can be obtained. These motion vectors can then be applied to a higher resolution version of the image, and then the search parameter can be lowered to 1 or 2 pixels when processing this higher resolution in order to refine the motion vectors. This continues until the full resolution image is processed.

In order to obtain the lower resolution images, a wavelet transform is performed on the image at various levels.

![Figure 2.8 – A generic hierarchical motion vector search strategy; Image from (Bhaskaran & Konstantinides, 1995)]
2.2.2 Motion Compensation

Once the motion vectors have been computed using the motion estimation algorithm, they need to be applied to the image they represent. Due to factors such as the limited search range and the concept of block processing, the predicted image will not be exactly the same as it is actually supposed to be.

A choice needs to be made as to whether it is acceptable to simply use the motion vectors to generate subsequent frames, or whether extra information will be transmitted along with the motion vectors.

This process is known as motion compensation. The primary aspect of motion compensation is to apply the motion vectors to generate the new frame. Once this new frame is calculated, the difference between the predicted frame and actual frame can be computed. This difference, known as the prediction error, can then be transmitted along with the motion vectors to help reconstruct the frame with less error.

Due to the motion compensation, the prediction error should have less entropy, and therefore is able to be compressed more, as opposed to simply taking the difference between two frames in the first place (without motion estimation and compensation).

The savings may not be large when looking at individual frames, however video sequences consist of many thousands of frames. All the small savings in size
add up to make the overall compression more effective. Test results further in this thesis show comparisons between using simple differencing, and using motion compensation plus differencing.

2.3 Putting it All Together – The Codec

The sections above have all described ways to process an image sequence in order to obtain significant compression. These techniques must all be combined in the correct order, with enough extra information saved to allow for the data stream to be decompressed at the receiving end.

Therefore, a coding algorithm must be devised, which puts everything together in a suitable order. This algorithm usually also defines a decoding procedure, which in most cases is simply the reverse of the encoding procedure.

2.3.1 Encoding

The creation of an encoding algorithm really goes hand in hand with the decoding algorithm. Data must be encoded in such a way that it is able to be simply and efficiently decoded. In most cases, the encoding has no real limitations in terms of speed and processing power, however the decoder usually requires the ability to decode the data in realtime.

The basic methodology to follow with encoding, is to put any header information first (this lets the decoder know the various parameters that were used when encoding). After the header, is the data that the decoder will use as input.
For example, the bitrate and stepsize information that is needed to perform the
dequantisation step on the data would be found in the header, whereas the
quantised values would be transmitted immediately after the header. The basic
idea is that the information is organised in such a way that the decoder knows
what each section represents. Without devising a proper encoding algorithm,
all the decoder sees is millions of bits of data that don't make sense.

2.3.2 Decoding

Since decoding is the reverse of encoding, it simply needs to use the
information it receives to reconstruct the original frames of the video.

The first task would be to undo any entropy coding that was performed on the
frames. This could include Huffman Coding and RLE. After that, the
information should consist of the original quantised values, as the entropy
coding is a lossless step.

Knowing the quantisation stepsize, the dequantised values are easily
calculated. The next step is to reverse the wavelet transformation process and
the decoded/decompressed frame should be the final result.

The final step is a motion compensation procedure that utilises motion vectors
to calculate subsequent frames, and adds the reconstructed difference frames
to the predicted ones. Motion compensation is the reverse of the motion
estimation procedure in the encoding algorithm.
Chapter 3 - Implementation Using MATLAB

3.1 Choosing MATLAB

MATLAB was chosen as the best method to implement the video compression codec from an analysis point of view. Its extended use of matrices and arrays makes it perfect for image processing, since an image can be represented as a matrix of pixels.

There are of course limitations, such as speed, however the aim of this thesis was not to fully implement a new codec, but to investigate the benefits of using wavelet transformations and hierarchical motion estimation in video compression.

3.2 Structure of the Encoding Algorithm

A number of procedures and functions needed to be written to implement each phase of the coding and decoding algorithms. The use of functional blocks rather than a start to finish program, made the design and implementation much easier.

The algorithm implemented in this thesis consists of the following major steps:

- Wavelet transform
- Quantisation with varying bits and deadzone values
- A modified Run Length Encoding routine
- Huffman Coding
• Hierarchical motion estimation

The flow diagram in Figure 3.1 shows each of the procedures and functions and how they interact to get the final compressed video sequence. Those marked internal are functions provided internally by MATLAB but were important enough to mention as a separate step in the flow. Those marked external are functions not written by myself, but obtained via the Internet as freely distributable files.

The functions that have a larger box with a thicker border protruding from them, are those which contain within them calls to other sub-functions.

As can be seen, the decoding functions are built into the coding functions to provide better quality output. Rather than predict each frame based on its predecessor and then calculate the difference, each frame is predicted using the decoded version of the preceding frame. This prevents the quantisation errors (the part of the compression which is lossy) from successively degrading the quality of subsequent frames.
main

Notes:
- Dashed line 1 represents a function that is executed on all but the first iteration.
- Dashed line 2 represents an order of execution to be followed on the final iteration.

Figure 3.1 – Flow diagram of main video compression algorithm
3.3 The Functions, Procedures and Scripts

The following sections contain information about how each of the major functions, procedures and scripts that form the complete video compression algorithm work. An overview of the order of tasks within each script is also shown.

3.3.1 initialise

This script defines a number of variables and loads uncompressed video data into memory.

1. Define path variable
2. Define global variables
3. Load the images from a precompiled .mat file
4. Define the number of frames to process
5. Define the search parameter
6. Define the macroblock size
7. Define the number of levels in the wavelet transform
8. Define the type of wavelet to use
9. Define the number of bits for each frame, and within each frame define the number of bits for each level
10. Define the deadzone for each frame, and within each frame define the deadzone for each level
3.3.2 convertimage

This is a function that takes one frame and performs a 2 dimensional discrete wavelet transform on it, returning level 0, 1 and 2 as three separate images. The diagonal, horizontal, and vertical components are discarded.

1. Obtain the size of the image
2. Convert it to the double datatype and define this as the level 0 image
3. Perform a 2D Discrete Wavelet Transform on the level 0 image using the Haar wavelet type, and define the approximation coefficients as the level 1 image
4. Perform a 2D Discrete Wavelet Transform on the level 1 image using the Haar wavelet type, and define the approximation coefficients as the level 2 image

3.3.3 wavedec2 (internal function)

This internal function is used to obtain the wavelet transform of the frame according to the number of levels and wavelet type specified in the initialisation script.

3.3.4 quantiser

This function calls a lower level function (quantise.m) to perform quantisation on each level of the wavelet transformed image, and returns the quantised data along with the stepsize and minimum value matrices.

1. Obtain the size of the bookkeeping matrix S (S defines how the transformed data is arranged with respect to the original image)
2. Initialise the variables that will be returned as output from the function

3. Step through each level of the transformed image, and call the quantise function for each of the levels

3.3.5 quantise

This is the low level function that performs the quantisation on the values passed to it by the quantiser function above. It takes the input and divides it up depending on the number of bits allocated, returning the quantised values along with the stepsize used and the minimum value of the data. It also applies a deadzone to the data, that is used to make more of the data quantise to zero to improve compression.

1. Obtain the maximum and minimum values of the data and calculate the stepsize based on their difference and the number of bits

2. Initialise the variable to hold the quantised data to zeros of the correct length

3. Step through from the minimum value up to the maximum value, using the calculated stepsize, and find all values in the current quantisation level. Each of these values is associated with a quantised value.

4. Due to rounding of floating point numbers, specifically quantise all occurrences of the lowest value to 1, and all occurrences of the maximum value to the highest number dependent on the number of bits specified

5. Apply the deadzone to all values in the specified region

6. Convert the quantised data to uint8 (unsigned 8 bit integer) to ensure it is represented as this format and not as floating point numbers (Note, the
number of bits will never exceed 8 anyway, so this is an appropriate
datatype for all situations)

3.3.6 compress

The purpose of this function is twofold. The first task it performs is to separate
the incoming data into two single dimension matrices of the same length. The
first contains all the non-zero values from the data, and the second contains the
index of each non-zero value. This is similar to run length encoding but is less
complicated and more suited to the type of data remaining after a wavelet
transform. Another conversion is made on the index data, converting each
value to the difference between it and its previous value. This will allow it to be
compressed further in the next task which is the huffman coding of the index
and main data.

1. Obtain the index and non-zero value matrices from the input data
2. Convert each value in the index matrix to be the difference between itself
   and the previous value
3. Call the huffman_compress function to compress the index data
4. Call the huffman_compress function to compress the main data
5. Define a lengths matrix which contains three values – the length of the non-
   zero value matrix, the length of the index matrix, and the length of the
   original input data
3.3.7 huffman_compress (external function)

This function was obtained from David Cary's website (http://www.rdrop.com/~cary/program/image_processing/). It performs Huffman Coding on the input data, returning both a compressed form of the data, and a codebook matrix.

3.3.8 motion_est

This is the main function that implements the hierarchical motion estimation algorithm. It ensures the correct two frames are in memory (the current and reference) and then calls the lower level routines to actually perform the motion estimation.

1. Change the previous “current” frame to the new “reference” frame
2. Load the new “current frame” using the convertimage function
3. Obtain the size of the image
4. Call the minblock function for each of the three levels of images, starting with the lowest, through to the original full size image
5. Calculate the final motion vectors using the intermediate motion vectors obtained from each of the calls to minblock

3.3.9 minblock

This function is called from motion_est and divides the image into blocks based upon the block size specified in the initialise procedure. For each block, it determines the starting position for the search based on which level it is
processing. It then calls the lower level function minmae to obtain the motion vector for each block.

1. Obtain the size of the image
2. Initialise the motion vector matrices to the correct size, filled with zero values
3. Step through each block in the image and calculate the location of the top-left corner of the block based on which level is being processed. If it is not the lowest level (2), modify the starting search position according to the motion vectors already calculated.
4. Call the minmae function to obtain the motion vector for each block
5. Store the motion vectors in the motion vector matrices

**3.3.10 minmae**

This was the first function I wrote, and everything else was written around it. It uses the search parameter and starting point to calculate the Mean Absolute Error between the block in the current frame, and a range of nearby blocks in the reference frame. The minimum is then found and a vector defining its relative position is returned to the minblock function.

1. Initialise the MAE matrix to the correct size, filled with zero values
2. Step through each region as defined by the search parameter and calculate the MAE
3. Find the minimum MAE value and return it as the output from the function
   (Note, if the minimum is at the origin, this is returned, however if there are multiple minimums of the same value, only the first is returned)
### 3.3.11 decompress

As expected, this function does the opposite of the compress function. The Huffman Coded values and codebooks are input to this function, which calls huffman_uncompress to reverse the procedure. The index is then recalculated to its original values, as opposed to each value being the difference of the one before it. Finally, it creates the original matrix by filling in the gaps between the non-zero values with zeros.

1. Call the huffman_uncompress function to reverse the Huffman Coding that was originally performed on the data
2. Recalculate the index
3. Create the original matrix using the MATLAB internal sparse function
4. Ensure any zeros at the end are added to make the original matrix the correct size

### 3.3.12 huffman_uncompress (external function)

This function was obtained as part of the huffman_compress package, and reverses the Huffman Coding procedure that huffman_compress.m performs.

### 3.3.13 dequantiser

Similar to the decompress function, dequantiser does the reverse of the quantiser function. It takes the input from the decompress function and the bookkeeping information such as the stepsize, bits and minvalue figures and returns the dequantised matrix after processing each level with the dequantise function.
1. Obtain the size of the bookkeeping matrix S (S defines how the transformed data is arranged with respect to the original image)

2. Initialise the dequantised variable to the current size

3. Step through each level of the decompressed image, and call the dequantise function for each of the levels

### 3.3.14 dequantise

This function is very simple, and simply takes a quantised input variable, plus the corresponding stepsize, bits and minvalue figures, and returns the dequantised data.

1. Initialise the dequantised variable to the correct size

2. Set all zero values to zero

3. Step through each other value from 1 to the maximum value according to the number of bits, and set the dequantised value based on the stepsize and minvalue variables

### 3.3.15 waverec2 (internal function)

This internal function is used to reverse the wavelet transform performed by the wavedec2.m function. It takes the transformed data, bookkeeping matrix, and wavelet type as inputs, and returns the original image.

### 3.3.16 PSNR (external function)

This function was obtained from the Mathworks FTP site (ftp://ftp.mathworks.com/pub/contrib/v5/image/PSNR.m) and is used to
calculate the PSNR of the two images passed into it. The input must be a figure between 0 and 1, so it is used in my program to calculate the PSNR of a decompressed frame and the original frame, after dividing each value by 255 (Since their values range from 0 to 255 as they are 8-bit grayscale images).

### 3.3.17 predict

This function implements the motion compensation part of the codec. It takes the reference frame and applies the motion vectors to it. The result is compared to the actual current frame, and the difference returned as the output of the function. Note, this can be used where the reference frame is the original, or in the case of my codec, the reference frame is the decompressed and dequantised frame. This improves the quality of the prediction as it eliminates cascading quantisation error noise.

1. Initialise with zeros the variable to hold the predicted frame
2. Step through each block and use the motion vectors to assign it to the corresponding block from the reference frame
3. Take the difference between the predicted and actual current frame and return it as output

### 3.3.18 huff

This procedure simply performs Huffman Coding on the motion vectors using the `huffman_compress` function. Since some values are negative, it also needs to translate each value up so that the minimum becomes 0. These minimum values are stored along with the encoded motion vectors.
1. Obtain the minimum value of the motion vectors

2. Using huffman_compress, compress the motion vectors after subtracting the minimum values
Chapter 4 - Analysis and Discussion of the Test Results

4.1 Overview of Test Data and Procedures

The video sequence used for testing was extracted from an MPEG video on the Windows 95 CDROM called Goodtime.avi.

One second was extracted, in which a car enters from the left and moves across to the right, with people moving in the background. I considered it appropriate to test a motion estimation based video compression algorithm on.

The format of each frame is RGB (each pixel has a red, green and blue component) however they were converted into YCbCr format (Y is the luminance value, and Cb and Cr are the colour or chrominance values) and only the Y component was used. This is effectively a grayscale version of the frame, with pixels taking on a value between 0 and 255.

Each of the 15 frames is 320 pixels wide by 240 pixels high. Although extracted from an already compressed video sequence, it is suitable enough to obtain sufficient results. The compression however may not be as much as desired, due to much of the redundancy already being removed by the original MPEG encoding. It is however the effectiveness of the method that is being investigated in this thesis.
The frames as extracted from the MPEG video sequence are shown in Appendix A, Figure A.1.

4.2 Test Results

All tests use a variety of parameters, however the following were kept constant for each:

- \( p \) (search parameter) = 16
- Wavelet type = Haar
- Wavelet levels = 4

The quality of the output is measured by the peak signal-to-noise ratio (PSNR). This gives a reading in decibels (dB) that rates the quality of a decompressed frame against the original. The larger the PSNR, the better a match they are.

4.2.1 Uncompressed frame sequence

There are 76,800 pixels in each frame, and being an 8-bit grayscale image, this equates to 76,800 bytes per frame. Thus the total size of one second of uncompressed video is 1,125kB \((1kB = 1024 \text{ bytes})\). This is larger than one megabyte \((1\text{MB} = 1024\text{kB})\) and obviously not suitable for transmission or convenient storage.
4.2.2 Uncompressed First Frame Followed by Difference Frames

Considering that the minimum value in an 8-bit grayscale image is 0 (binary 0000 0000), and the maximum is 255 (binary 1111 1111), the difference between two frames can vary between -255 and +255. This requires 9 bits to represent (1 bit for the sign and 8 bits for the range 0 to 255). Therefore, although at first it may seem that sending only the difference between one frame and the next would be more efficient, it's actually worse.

The first second of video would require one frame of 76,800 bytes followed by fourteen frames of 86,400 bytes. Thus the first second would be 1256.25kB and subsequent seconds would be 1265.625kB.
4.2.3 Wavelet Compressed Frame Sequence

Taking the wavelet transform of each frame leads to a large number of zero valued coefficients being generated. Several tests were done using the entire video compression algorithm already discussed, but with the motion estimation and compensation parts left out. Also, the full frame was transformed as opposed to the difference frame (see the next section).

<table>
<thead>
<tr>
<th>Table 4.1 - Parameters for tests in section 4.2.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>Deadzone</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4.2 - PSNR results for tests in section 4.2.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>PSNR (dB)</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4.3 - Summary of test results in section 4.2.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>Average PSNR (dB)</td>
</tr>
<tr>
<td>Total size (bytes)</td>
</tr>
<tr>
<td>Average size per frame (bytes)</td>
</tr>
</tbody>
</table>
4.2.4 Wavelet Compressed Initial Frame Followed by Wavelet Compressed Difference Frames

Similar to the previous set of tests, these perform the video compression without motion estimation. However, all frames besides the first are encoded as the difference between them and the previous frame. As in the main codec, the difference is between the decompressed/dequantised version of the previous frame. This stops quantisation noise from propagating through the frames.

<table>
<thead>
<tr>
<th>Table 4.4 – Parameters for tests in section 4.2.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
</tr>
<tr>
<td>[8 7 6 5 4]</td>
</tr>
</tbody>
</table>

| Table 4.5 – PSNR results for tests in section 4.2.4 |
|-----------------|----------------|----------------|----------------|----------------|
| Frame | Test 1 | Test 2 | Test 3 | Test 4 |
| 1     | 35.84  | 35.84  | 35.84  | 28.22 |
| 2     | 33.90  | 33.25  | 31.85  | 29.37 |
| 3     | 33.30  | 29.80  | 30.05  | 26.86 |
| 4     | 35.72  | 30.01  | 28.70  | 29.90 |
| 5     | 33.95  | 32.23  | 29.51  | 30.21 |
| 6     | 34.55  | 31.81  | 29.76  | 29.83 |
| 7     | 30.86  | 30.46  | 29.35  | 30.37 |
| 8     | 34.22  | 27.39  | 29.19  | 29.46 |
| 9     | 31.06  | 29.48  | 29.79  | 28.46 |
| 10    | 30.85  | 28.26  | 29.56  | 26.91 |
| 11    | 30.64  | 26.84  | 29.85  | 27.16 |
| 12    | 32.84  | 29.45  | 25.84  | 28.78 |
| 13    | 28.27  | 29.92  | 23.47  | 24.60 |
| 14    | 31.76  | 32.11  | 26.12  | 29.72 |
| 15    | 32.63  | 31.54  | 29.24  | 29.05 |

| Table 4.6 – Summary of test results in section 4.2.4 |
|-------------------------------|----------------|----------------|----------------|----------------|
| Average PSNR (dB)            | 32.69          | 30.56          | 29.21          | 28.59          |
| Total size (bytes)           | 215,928        | 152,528        | 145,976        | 142,632        |
| Average size per frame (bytes) | 14,395        | 10,169         | 9,732          | 9,509          |

54
4.2.5 The Full Encoding Algorithm with Motion Estimation and Compensation

The following tests were conducted with the same parameters as the previous test. They were implemented using the full video compression encoding routine covered in section 3.2.

Table 4.7 - Parameters for tests in section 4.2.5

<table>
<thead>
<tr>
<th>Bits</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>[8 7 6 5 4]</td>
<td>[8 7 6 5 4]</td>
<td>[8 7 6 5 4]</td>
<td>[5 5 5 5 5]</td>
<td></td>
</tr>
<tr>
<td>2-15: [5 5 5 5 5]</td>
<td>2-15: [5 5 5 5 5]</td>
<td>2-15: [5 5 5 5 5]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deadzone</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1 2 4 8 16]</td>
<td>[1 2 4 8 16]</td>
<td>[1 2 4 8 16]</td>
<td>[30 30 30 30 30]</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.8 - PSNR results for tests in section 4.2.5

<table>
<thead>
<tr>
<th>Frame</th>
<th>Test 1 (dB)</th>
<th>Test 2 (dB)</th>
<th>Test 3 (dB)</th>
<th>Test 4 (dB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35.84</td>
<td>35.84</td>
<td>35.84</td>
<td>28.22</td>
</tr>
<tr>
<td>2</td>
<td>34.95</td>
<td>32.07</td>
<td>30.85</td>
<td>30.00</td>
</tr>
<tr>
<td>3</td>
<td>33.57</td>
<td>31.77</td>
<td>29.90</td>
<td>29.96</td>
</tr>
<tr>
<td>4</td>
<td>30.41</td>
<td>31.04</td>
<td>29.71</td>
<td>29.14</td>
</tr>
<tr>
<td>5</td>
<td>33.25</td>
<td>29.66</td>
<td>28.83</td>
<td>29.00</td>
</tr>
<tr>
<td>6</td>
<td>32.25</td>
<td>29.17</td>
<td>28.98</td>
<td>28.29</td>
</tr>
<tr>
<td>7</td>
<td>30.15</td>
<td>30.23</td>
<td>27.81</td>
<td>28.66</td>
</tr>
<tr>
<td>8</td>
<td>31.45</td>
<td>29.67</td>
<td>27.75</td>
<td>27.79</td>
</tr>
<tr>
<td>9</td>
<td>28.90</td>
<td>28.43</td>
<td>27.93</td>
<td>27.19</td>
</tr>
<tr>
<td>10</td>
<td>30.69</td>
<td>29.67</td>
<td>27.86</td>
<td>27.53</td>
</tr>
<tr>
<td>11</td>
<td>29.28</td>
<td>28.00</td>
<td>27.44</td>
<td>27.47</td>
</tr>
<tr>
<td>12</td>
<td>30.50</td>
<td>27.47</td>
<td>27.26</td>
<td>27.09</td>
</tr>
<tr>
<td>13</td>
<td>31.68</td>
<td>29.63</td>
<td>27.15</td>
<td>26.98</td>
</tr>
<tr>
<td>14</td>
<td>31.27</td>
<td>28.22</td>
<td>27.20</td>
<td>26.15</td>
</tr>
<tr>
<td>15</td>
<td>31.66</td>
<td>29.09</td>
<td>26.91</td>
<td>26.86</td>
</tr>
</tbody>
</table>

Table 4.9 - Summary of test results in section 4.2.5

<table>
<thead>
<tr>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average PSNR (dB)</td>
<td>31.72</td>
<td>30.00</td>
<td>28.76</td>
</tr>
<tr>
<td>Frames (bytes)</td>
<td>243,512</td>
<td>172,520</td>
<td>138,920</td>
</tr>
<tr>
<td>Motion vectors (bytes)</td>
<td>19,816</td>
<td>19,816</td>
<td>19,816</td>
</tr>
<tr>
<td>Total size (bytes)</td>
<td>263,328</td>
<td>192,336</td>
<td>158,736</td>
</tr>
<tr>
<td>Average size per frame (bytes)</td>
<td>17,555</td>
<td>12,822</td>
<td>10,582</td>
</tr>
</tbody>
</table>
4.3 Analysis and Discussion

4.3.1 Comparing the Two Non-Motion Estimation and Compensation Tests

The use of wavelet compressed difference frames gave better compression than the wavelet compressed full frames, however there was a slightly lower average PSNR value. When comparing figures A.2.1 and A.3.1 (test 1), A.2.2 and A.3.2 (test 2), A.2.3 and A.3.3/A.3.4 (tests 3 and 4), little difference can be seen.

The main reasons for the better compression rate is due to the increased number of zero, or close to zero values that exist when taking the difference of one frame to the next. The zeros ensure that the RLE works more efficiently, and the smaller range of values in the difference frames ensure that the huffman coding requires less codewords to process the quantised values.

4.3.2 Comparing the Two Difference Encoded Algorithms

This set of comparisons will be made between the wavelet compressed difference frames, and the full encoding algorithm including motion estimation and compensation.

Unfortunately, the results are not as hoped. All four tests show that the motion estimation and compensation actually reduce the compression rate, with
average frame sizes being approximately 800 bytes larger, with a slightly less (but insignificant) PSNR value.

However, a closer look shows that the main reason is due to the overhead of the motion vectors themselves. In tests 1 and 2, the simple wavelet compressed difference frame algorithm still performed better. However, in tests 3 and 4, the total size of the simple algorithm was larger than the size of the encoded frames in the full algorithm. Since the motion vectors add 19,816 bytes per second to the file size (approximately 1415 bytes per frame, considering there are only 14 sets of motion vectors), this blows out the average size per frame to become larger for the full algorithm than the simple one.

The factors influencing this could be any or all of the following:

- Block size (M x N) too big
- Search parameter (p) too small
- Inefficient encoding of motion vectors
- Insufficient motion to warrant the overhead that the motion vectors have

A smaller block size would improve the quality of the output significantly in cases where motion is not purely translational. That is, in cases where there may be stretching or rotation instead of a chessboard type motion. A larger the block size leads to coarser output from the motion compensation, which can make the difference values not significantly better than the non-motion compensated difference values.
If the motion is outside the range of the search parameter, then the motion vectors will not provide the best matching block. A search parameter of 16 for instance will search a region of 33 by 33 pixels around the origin. If most of the motion is greater than 33 pixels, this search parameter would not be sufficient.

In order to use motion estimation and compensation, the overheads it generates (such as encoding time and the size of the motion vectors and other parameters) must not take away more than the gain achieved by implementing the motion estimation in the first place. If extra compression of 500 bytes is achieved but the overhead of the motion vectors amounts to 1000 bytes, clearly the overall effect is detrimental to the objective.

Following on from the previous idea, if the motion is not significant enough, there may be no benefit in using motion estimation and compensation as the difference values are already minimal.

This does not mean that using motion estimation and compensation should be avoided, as the difference in file size (ignoring the motion vector component) is still significant compared with the output from the non-motion estimation algorithm. In test 4, this equates to approximately 8kB per second. Considering a 5 minute video, the saving of over 2MB is definitely significant.
There needs to be more work done in the area of efficiently encoding the extra overheads that motion estimation and compensation generate. A thresholding mechanism could also be implemented which would determine on a frame by frame basis whether or not to use motion estimation and compensation. This would possibly generate an end result that outperforms the results from the simple and full algorithms. The overhead of the motion vector would be avoided when it will lower the compression rate, and included when the compression is higher when using it.

Without losing sight of the purpose of this research, the use of Wavelet based compression in video compression is definitely beneficial. If anything can be seen from the above results, it is that the more zero values that can be created in the data, the better the compression will be. Since the wavelet transform converts the data into a large proportion of zeros (or values which can be safely quantised to zero), it aids significantly in this task.

4.3.3 Rough Comparison with MPEG-1

The following example is taken from p164 of (Bhaskaran & Konstantinides, 1995).

The image size in the example is 352x240 pixels with 12bpp (4bit RGB or 12 bit grayscale). A frame rate of 30fps was used and the compression achieved was 1.15Mbps.
Using the best result obtained in the tests above, Test 4 using no motion estimation would yield a rate of 9509 bytes/frame x 8 bits/byte x 15 frames/second = 1.14Mbps. This is virtually the same as MPEG-1, however this is based on an original image slightly smaller (320x240 pixels) and only 8bpp and 15fps.

In summary, using this algorithm with the same video sequence in the MPEG-1 test, would produce a bits per second (bps) rate 2 to 3 times larger. Clearly, more needs to be done to make it as good, if not better.

However, considering the number of years and people who worked on the MPEG-1 standard, the purpose of my research was not to develop a better result, but to evaluate the benefits of the use of wavelet transforms in the process.
Chapter 5 - Further Work and Conclusions

The analysis of the test results shows that a wavelet transform based video compression algorithm, including the use of hierarchical motion estimation and compensation, is a viable alternative to the current DCT based algorithms such as MPEG-1.

Although these test results do not achieve the same compression as MPEG-1, they do indicate, provided more work is performed in this area, that a wavelet transform based algorithm can achieve the same, if not better results. The key point is the potential of the wavelet transform to convert as many coefficients to zero as possible, without causing as much loss in the quality of the output.

DCT based compression achieves higher rates by using a coarser quantisation step. Although this does yield high compression, the quality of the output drops dramatically as the compression rate is increased.

Wavelet transform based compression can produce more zero values and hence higher compression rates, at the transform stage itself – before any quantisation is even applied.

As discussed, further work needs to be performed in order to realise the full potential of these relatively new applications of the wavelet transform. There is even research being conducted into a three dimensional wavelet transform, and
how it can benefit video compression. More information on this field can be found in Chapter 24 of (Topiwala, 1998)

Some of the improvements that need to be investigated include the following:

- Tests on other video sequences, preferably ones which have not yet had any compression performed on them (unlike the one used in this thesis)
- Rewriting of the code into a language such as C++, which will improve the speed of the motion estimation
  - Note: It took approximately 1 hour on a Sun Sparcstation 20 to encode one second of video
- More precise comparisons between wavelet based video compression and other algorithms such as MPEG-1
- Enhancements to the quantisation algorithm to work out what the best number of bits and deadzone are, rather than manually guessing
  - It would have to calculate this based on the target bpp (bits per pixel) rate and an acceptable PSNR value
  - This was beyond the scope of this research, as the computation and work involved could easily be the subject of an entire thesis itself
The content of this thesis is a starting point for future Engineering Honours students who are interested in the field of image and video compression. The framework has been created, upon which further research and work can be added.

As time goes by there is always going to be a need for faster, more effective, and better quality video compression algorithms. The use of wavelet transforms will absolutely be a major factor in achieving this goal.
Bibliography


Note: Some references were not used in-text however they provided background information when conducting research.
Appendix A – Video Sequences

Figure A.1 – Original uncompressed frames
Figure A.2.1 – Decompressed frames from test 1 in section 4.2.3
Figure A.2.2 – Decompressed frames from test 2 in section 4.2.3
Figure A.2.3 – Decompressed frames from test 3 in section 4.2.3
Figure A.3.1 – Decompressed frames from test 1 in section 4.2.4
Figure A.3.2 – Decompressed frames from test 2 in section 4.2.4
Figure A.3.3 – Decompressed frames from test 3 in section 4.2.4
Figure A.3.4 – Decompressed frames from test 4 in section 4.2.4
Figure A.4.1 – Decompressed frames from test 1 in section 4.2.5
Figure A.4.2 – Decompressed frames from test 2 in section 4.2.5
Figure A.4.3 – Decompressed frames from test 3 in section 4.2.5
Figure A.4.4 – Decompressed frames from test 4 in section 4.2.5
compress.m

% This function takes a vector (1D) and does the following
% 1. Creates a vector containing the non-zero values of the data
% 2. Creates an index vector containing the location of the non-zero values
% 3. Converts the index vector so that each entry is the difference between it and the previous
% 4. Applies Huffman coding to the resultant vectors

function [main_data,main_kraft,index_data,index_kraft,lengths] = compress(decompressed);

% Remove the zero valued components and create indices
[nothing,temp_index,compressed] = find(decompressed);
new_index(2:length(temp_index)) = temp_index(2:length(temp_index)) -
       temp_index(1:((length(temp_index)-1)));
new_index(1) = temp_index(1);
[index_data,index_kraft] = huffman_compress(double(new_index));
[main_data,main_kraft] = huffman_compress(double(compressed));
lengths(1) = length(compressed);
lengths(2) = length(new_index);
lengths(3) = length(decompressed);

convertimage.m

function [image,image1,image2] = convertimage(myimage);

[Y,X] = size(myimage);

% Fix the type problem coming from the images in the video sequence
%tempimage = zeros(Y,X);
%tempimage(1:Y,1:X) = myimage(1:Y,1:X);
%image = tempimage;

image = double(myimage);

% 2D discrete wavelet transform of image (level 1)
[CA,CH,CV,CD] = dwt2(image,'haar');
image1 = CA;

% 2D discrete wavelet transform of image1 (level 2)
[CA,CH,CV,CD] = dwt2(image1,'haar');
image2 = CA;
decompress.m

function decompressed = decompress(main_data, main_kraft, index_data, index_kraft, lengths)
    compressed = huffman_uncompress(main_data, main_kraft, lengths(1));
    new_index = huffman_uncompress(index_data, index_kraft, lengths(2));
    nothing = ones(1, length(new_index));
    new_index = double(new_index);
    compressed = double(compressed);
    for y = 2:length(new_index),
        % new_index(2:length(new_index)) = new_index(2:length(new_index)) +
        % new_index(1:length(new_index)-1);
        new_index(y) = new_index(y) + new_index(y-1);
    end;
    decompressed = sparse(nothing, new_index, compressed);
    decompressed = full(decompressed);
    if length(decompressed) ~= lengths(3)
        decompressed(length(decompressed)+1:lengths(3)) = 0;
    end;

dequantise.m

% dequantiser for positive values
function [new] = dequantise(old, stepsize, bits, minvalue);
new = zeros(1, length(old));

    new(find(old == 0)) = 0;
    for i = 1:1:((2^bits)-1),
        new(find(old == i)) = ((i-0.5) * stepsize) + minvalue;
    end;
dequantiser.m

function [dequantised] = dequantiser(C,S,stepsize,bits,minvalue);

[y,x] = size(S);

if nargin < 5
    % need to abort because we don't have the correct number of arguments
elseif (length(bits) ~= (y-1))
    % need to abort because the bits variable doesn't have the correct number of entries
else
    dequantised = zeros(1,S(y,1)*S(y,2));

    for level = 1:y-1,
        if level == 1,
            [dequantised(1:S(level,1)*S(level,2))] = dequantise(C(1:S(level,1)*S(level,2)),
            stepsize(level),bits(level),minvalue(level));
        else
            [dequantised(1+(S(level,1)*S(level,2)) : S(level,1)*S(level,2)*4)] =
            dequantise(C(1+(S(level,1)*S(level,2)) : S(level,1)*S(level,2)*4),
            stepsize(level),bits(level),minvalue(level));
        end;
    end;
end;

huff.m

% HUFF
%
% Compress the motion vectors with huffman coding

i_min(index-1) = min(min(i_block{index-1}));
j_min(index-1) = min(min(j_block{index-1}));
[i_data{index-1},i_kraft{index-1}] = huffman_compress(i_block{index-1} - i_min{index-1});
[j_data{index-1},j_kraft{index-1}] = huffman_compress(j_block{index-1} - j_min{index-1});
initialise.m

% INITIALISE
%
% Set up paths and global variables in preparation for the rest of the procedures
%
% set path
path(path,’/home/abyine/Motion’);
path(path,’/home/abyine’);
path(path,’/home/abyine/video’);
path(path,’/home/abyine/huffman’);
path(path,’/home/abyine/huffman/other’);
%
% define global variables
global i_block j_block index
%
% load the image file from disk
load 'myimage.mat';
%
% Parameters
%
% number of frames to process
frames = 15;
%
% motion estimation
% search parameter
p = 16;
%
% macroblock size
M = 8;
N = 8;
%
% wavelet transform
% number of levels
wavelet_levels = 4;
%
% type of wavelet
wavelet_type = 'haar';
%
% quantisation
% number of bits per level - should have (wavelet_levels + 1) entries
bits{1} = [8 7 6 5 4];
for i = 2:frames,
    bits{i} = [5 5 5 5 5];
end;
%
% deadzone values - should have (wavelet_levels + 1) entries
deadzone{1} = [1 2 4 8 16];
for i = 2:frames,
    deadzone{i} = [30 30 30 30 30];
end;
main.m

% MAIN
%
% Call all the relevant procedures using the specified parameters

% Initialise
initialise;

% Process the first frame
disp('Processing the first frame...');

% load the first frame
disp('Loading');
[R,R1,R2] = convertimage(myimage{1}(:, :, 1));

% wavelet decomposition of first frame
disp('Wavelet decomposition');
[CXC,S] = wavedec2(R,wavelet_levels,wavelet_type);

% quantise the first frame
disp('Quantise');
[quantised,stepsize{1},minvalue{1}] = quantiser(CXC,S,bits{1},deadzone{1});

% perform modified RLE and huffman coding on the first frame
disp('Coding');
[main_data{1},main_kraft{1},index_data{1},index_kraft{1},lengths{1}] = compress(quantised);

% Process the remaining frames
for index = 2:frames,
    sprintf('Processing frame %d',index)
    motion_est;

    % decompress the previous frame (undo huffman coding and modified RLE)
disp('Decompressing previous frame');
decompressed = decompress(main_data{index-1},main_kraft{index-1},index_data{index-1},index_kraft{index-1},lengths{1});

    % dequantise the previous frame
disp('Dequantise previous frame');
dequantised = dequantiser(decompressed,S,stepsize{index-1},bits{index-1},minvalue{index-1});

    % Reconstruct previous frame
    if index == 2
        frame{index-1} = waverec2(dequantised,S,wavelet_type);
    else
        frame{index-1} = waverec2(dequantised,S,wavelet_type) + prediction;
    end;

    % Give PSNR value
    sprintf('PSNR for frame %d is',index-1)
    PSNR(double(uint8(frame{index-1}))/255,double(myimage{index-1}(:, :, 1))/255)
% Predict the C image and return the difference between the actual and predicted images
    disp('Motion compensation');
    [difference,prediction] = predict(Y,X,M,N,C,frame{index-1});

% wavelet decomposition of difference frame
    disp('Wavelet decomposition');
    [CXC,S] = wavedec2(difference, wavelet_levels, wavelet_type);

% quantize the difference frame
    disp('Quantise');
    [quantised,stepsize{index},minvalue{index}] = quantiser(CXC,S,bits{index},deadzone{index});

% perform modified RLE and Huffman coding on the difference frame
    disp('Coding');
    [main_data{index},main_kraft{index},index_data{index},index_kraft{index},lengths{index}] = compress(quantised);

% Perform Huffman coding on the motion vectors
    disp('Motion vector coding');
    huff;
end;

% decompress the previous frame (undo Huffman coding and modified RLE)
    disp('Decompressing previous frame');
    decompressed = decompress(main_data{index},main_kraft{index},index_data{index},index_kraft{index},lengths{index});

% dequantize the previous frame
    disp('Dequantise previous frame');
    dequantised = dequantiser(decompressed,S,stepsize{index},bits{index},minvalue{index});
    frame{index} = waverec2(dequantised,S,wavelet_type) + prediction;

% Give PSNR value
    sprintf('PSNR for frame %d is ',index)
    PSNR(double(uint8(frame{index}))/255,double(myimage{index}(:,:,1))/255)
    disp('Saving data...');
% save the frames in a file
    save 'frames.mat' main_data main_kraft index_data index_kraft lengths stepsize bits minvalue S;

% save the motion vectors in a file
    shape = uint8(size(i_block{1}));
    save 'motion_vectors.mat' i_data i_kraft j_data j_kraft i_min j_min shape;

% save decoded frames for printing
    save 'decoded.mat' frame;
    disp('Operation complete.');
function [min_i_block, min_j_block] = minblock(p,M,N,C,R,U,V,level)

[Y,X] = size(C);

% initialise matrices of locations to zero
min_i_block = zeros(Y/N,X/M);
min_j_block = zeros(Y/N,X/M);

for x = 1:M:(X-M+1),
    for y = 1:N:(Y-N+1),
        position_x = ((x-1)/M)+1;
        position_y = ((y-1)/N)+1;
        if (level == 2)
            x2 = round(x+U(position_y,position_x));
            y2 = round(y+V(position_y,position_x));
        else
            x2 = round(x+(2*U(position_y,position_x)));
            y2 = round(y+(2*V(position_y,position_x)));
        end;
        [i,j] = minmae(p,x2,y2,M,N,C,R,X,Y);
        min_i_block(position_y,position_x) = i;
        min_j_block(position_y,position_x) = j;
    end;
end;
function [min_i, min_j] = minmae(p,x,y,M,N,C,R,X,Y);

% initialise the matrix holding the MAE values
mae_matrix = zeros(2*p + 1,2*p + 1);

% loop through each i and j coordinate and calculate the MAE
for i = -p:p,
    for j = -p:p,
        % check boundary conditions and only calculate value if within the boundary
        if ((y+N-1 <= Y) & (x+M-1 <= X) & (y > 0) & (x > 0) & (y+j > 0)
            & (x+i > 0) & (y+j+N-1 <= Y) & (x+i+M-1 <= X))
            mae_matrix(j+p+1,i+p+1) = sum(sum(abs(C(y:y+N-1,x:x+M-1) -
            R(y+j:y+j+N-1,x+i:x+i+M-1))));
        else
            mae_matrix(j+p+1,i+p+1) = 100000;
        end;
    end;
end;

% determine the location of the first minimum in the matrix
[a1,b1] = min(mae_matrix);
[a2,b2] = min(a1);
min_mae = a2;

% If the centre value (i.e. i and j both equal zero) is equal to the minimum, then use this instead
if (mae_matrix(p+1,p+1) == min_mae)
    min_i = 0;
    min_j = 0;
else
    min_i = (b2 - (p+1));
    min_j = (b1(b2) - (p+1));
end;
motion_est.m

% load frames
% If this is the second frame, we already have the first frame in memory
disp('Loading');

if index ~= 2
    % The previous C image becomes the new R image
    R = C;
    R1 = C1;
    R2 = C2;
end;

% Load second frame
[C,C1,C2] = convertimage(myimage{index}{:,:1});

% perform calculations
[Y,X] = size(C);

disp('Motion estimation phase 1');
[i_block_2,j_block_2] = minblock(p/4,M/4,N/4,C2,R2,zeros(Y,N,X,M),zeros(Y,N,X,M),2);
disp('Motion estimation phase 2');
[i_block_1,j_block_1] = minblock(1,M/2,N/2,C1,R1,i_block_2,j_block_2,1);
disp('Motion estimation phase 3');
[i_block_0,j_block_0] =
    minblock(1,M,N,C,R,(2*i_block_2)+i_block_1,(2*j_block_2)+j_block_1,0);

% final motion estimation vectors
i_block{index-1} = (4*i_block_2) + (2*i_block_1) + i_block_0;

j_block{index-1} = (4*j_block_2) + (2*j_block_1) + j_block_0;
predict.m

function [difference, newC] = predict(Y,X,M,N,C,R);

global i_block j_block index

% use motion estimation vectors to predict new image
newC = zeros(Y,X);

for x = 1:X/M,
    for y = 1:Y/N,
        y_index = 1 + (y*N) : N;
        x_index = 1 + (x*M) : M;

        if (y_index+j_block(index-1)(y,x) > 0) & (y_index+j_block(index-1)(y,x)+ (N-1) <= Y) & (x_index+i_block(index-1)(y,x) > 0) &
            (x_index+i_block(index-1)(y,x)+ (M-1) <= X)
            newC(y_index:y_index+(N-1),x_index:x_index+(M-1)) =
                R(y_index+j_block(index-1)(y,x):y_index+j_block(index-1)(y,x)+(N-1),x_index+i_block(index-1)(y,x):x_index+i_block(index-1)(y,x)+(M-1));
        end;
    end;
end;

% difference
difference = double(C - newC);
function PSNR(A,B)

% PURPOSE: To find the PSNR (peak signal-to-noise ratio) between two
% intensity images A and B, each having values in the
% interval [0,1]. The answer is in decibels (dB).

% SYNOPSIS: PSNR(A,B)

% DESCRIPTION: The following is quoted from "Fractal Image
% Compression", by Yuval Fisher et al., (Springer Verlag, 1995),
% section 2.4, "Pixelized Data".

"...PSNR is used to measure the difference between two
images. It is defined as

\[
\text{PSNR} = 20 \log_{10}(\frac{b}{\text{rms}})
\]

where \( b \) is the largest possible value of the signal
(typically 255 or 1), and \( \text{rms} \) is the root mean square
difference between two images. The PSNR is given in
decibel units (dB), which measure the ratio of the peak
signal and the difference between two images. An
increase of 20 dB corresponds to a ten-fold decrease in the rms
difference between two images.

There are many versions of signal-to-noise ratios, but
the PSNR is very common in image processing, probably
because it gives better-sounding numbers than other
measures."

% EXAMPLE 1: load clown
% A = ind2gray(X,map); % Convert to an intensity image in
% [0,1]  
% B = 0.95 * A; % Make B close to, but different
% from, A.
% PSNR(A,B) % ---> "PSNR = +33.49 dB"

% EXAMPLE 2: A = rand(256); % A is a random 256 X 256 matrix in [0,1].
% B = 0.9 * A; % Make B close to, but different from, A.
% PSNR(A,B) % ---> "PSNR = +24.76 dB (approx)"

if A == B
    error('Images are identical: PSNR has infinite value')
end

max2_A = max(max(A));
max2_B = max(max(B));
min2_A = min(min(A));
min2_B = min(min(B));

if max2_A > 1 | max2_B > 1 | min2_A < 0 | min2_B < 0
    error('input matrices must have values in the interval [0,1]')
end
\[
\text{error} = A - B;
\]
\[
\text{decibels} = 20 \times \log_{10}(1/(\sqrt{\text{mean(\text{mean(error.^2)})}}));
\]
\[
\text{disp(sprintf('PSNR = %.5.2f dB',\text{decibels}))}
\]

---

**quantise.m**

% quantiser

function [new, stepsize, minvalue] = quantise(old, bits, deadzone);

    maxvalue = max(old);
    minvalue = min(old(find(old)));
    stepsize = (maxvalue - minvalue) / (2^bits - 1);

    new = zeros(1, length(old));

    for i = minvalue:stepsize:maxvalue,
        new(find(old <= i & old > (i - stepsize))) = ((i - minvalue) / stepsize);
    end;

    new(find(old == minvalue)) = 1;
    new(find(old == maxvalue)) = (2^bits - 1);
    new(find(old < deadzone & old > (0 - deadzone))) = 0;

    new = uint8(new);
quantiser.m

function [quantised, stepsize, minvalue] = quantiser(C, S, bits, deadzone);
[y,x] = size(S);

if nargin < 4
    % need to abort because we don't have the correct number of arguments
elseif (length(bits) == (y-1))
    % need to abort because the bits variable doesn't have the correct number of entries
elseif (length(deadzone) == (y-1))
    % need to abort because the deadzone variable doesn't have the correct number of entries
else

    quantised = zeros(1,S(y,1)*S(y,2));
    stepsize = zeros(1,y-1);
    minvalue = zeros(1,y-1);

    for level = 1:y-1,
        if level == 1,
            [quantised(1:S(level,1)*S(level,2)),stepsize(level),minvalue(level)] = quantise(C(1:S(level,1)*S(level,2)),bits(level),deadzone(level));
        else
            [quantised(1+(S(level,1)*S(level,2))):S(level,1)*S(level,2)*4),stepsize (level),minvalue(level)] = quantise(C(1+(S(level,1)*S(level,2))):S(level,1)*S(level,2)*4,level),deadzone(level));
        end;
    end;

end;
function [compressed_data, Kraft_vector, encodedlength4] = huffman_compress( raw_data )
% [compressed_data, Kraft_vector] = huffman_compress( raw_data )
% compresses a list of non-negative integers (raw data)
% into a uint8 list of compressed data.
% Usage:
% plaintext = double(imread('cameraman.tif'));
% plaintext = [5 5 5 5 4; 4 0 0 1 5];
% shape = size(plaintext);
% [compressed_data, bitlengths] = huffman_compress(plaintext,:);
% to recover, only need shape, compressed_data, bitlengths.
% recovered_data = huffman_uncompress(compressed_data, bitlengths, prod(shape));
% recovered_matrix = reshape(recovered_data, shape);
% if( isequal( plaintext, recovered_matrix ) ),
% disp('it worked')
% else,
% error(' I messed up somewhere ... ')
% end;
%
% The "bitlengths" array could be compressed further --
% -- see huff03.m from
% http://www.ux.his.no/-karlsk/
% .
% The form
% [compressed_data, Kraft_vector, encodedlength] = huffman_compress( raw_data )
% also returns the length of the compressed data, in bits, where
% length(compressed_data) == ceil(encodedlength4/8)
% See also IMWRITE_COMPRESSED, HUFFMAN_UNCOMPRESS.
%
% Change log:
% 1999-06-26:DAV: David Cary <d.cary@ieee.org> started.
%
% huff03.m in huffman.zip
% at
% http://www.ux.his.no/-karlsk/proj98/huffman.zip
% mirrored
% http://www.mathworks.com/ftp/miscv5.shtml
% written by
% Karl Skretting
% Hogskolen in Stavanger (Stavanger University), Norway
% karl.skretting@tn.his.no  Homepage: http://www.ux.his.no/-karlsk/
% does exactly the same thing as
% huffman_compress.m
% Like all Huffman compressors,
% huff03.m will get the same compressed data length
% as huffman_compress,
% but huff03.m packs the Kraft_vector
% much more tightly than huffman_compress.m does,
% and pre-pends it to the compressed data.
% (With David Cary's test data,
% packing that Kraft_vector down to zero bits
% would only improve my compression by 0.031 bpp,
% which is relatively insignificant).
% On the other hand, huffman_compress.m is much faster
% (a factor of 10 in my tests).

% Force raw_data into a 1D single list
raw_data = raw_data(:);
if (min(raw_data) < 0)
    error('Sorry, support for negative numbers not yet implemented.')
end;

histogram = histo(raw_data);
% ASSERT ( isequal ( length (histogram) , 1+max (raw_data) ) )
disp('getting Kraft vector')
drawnow
Kraft_vector = unsorted_kraft ( histogram ) ;
disp('generating code book')
drawnow

% ASSERT ( isequal ( encodedlengthl , huffmanlength (plaintext) ) )
encodedlengthl = Kraft_vector ' * histogram;
encodedlength2 = huffmanlength (raw_data);
if ( isequal ( encodedlengthl , encodedlength2 ) ) ,
else,
    encodedlengthl
    encodedlength2
    Kraft_vector
    error('Sorry, internal error')
end;
max_length = max(Kraft_vector);

% If we could guarantee that code lengths
% would never be more than 15,
% then we could pack 2 values 0..15
% into each byte (0 indicates "never happens").
if ( max_length < 15 )
else,
    disp('maximum code length:')
    max_length
end;

% ASSERT ( max_length < 53 )
if ( max_length < 32 )
else,
    max_length
    disp('... it *might* work up to')
    BITMAX
    disp('bits.')
    error('Sorry, code lengths greater than 31 bits not yet implemented.')
end;

% Get the code words:
% compactcodeP(Kraft_vector) gives the bogus code "-1" to items that
% never occur.
codebook = compactcodeP(Kraft_vector);
if (0)
print_bitstrings( codebook )
end;

disp('compressing data')
drawnow
% Finally, actually compress the data.
compressed_bits = prefix_encode( codebook, raw_data );
disp('packing data')
drawnow

encodedlength4 = length(compressed_bits);
% ASSERT( isequal( encodedlength3, encodedlength1 )
if( isequal( encodedlength1, encodedlength4 ) ),
else,
   encodedlength1
   encodedlength4
   print_bitstrings( codebook )
   print_without_spaces( compressed_bits )
   error('Sorry, internal error')
end;

% Pack everything into uint8 vectors.

% minor compression of code lengths
Kraft_vector = uint8(Kraft_vector);

% add '0' bits to round up to a multiple of 8
bytes = ceil(encodedlength4/8);
stuff_bits = 8*bytes - encodedlength4;
if(0 == stuff_bits)
else,
   compressed_bits(8*bytes) = 0;
end;

bits = reshape(compressed_bits, 8, bytes);

% pack first bit into least-significant-bit
% of first byte (little-endian)
compressed_data = uint8(...
   1*bits(1,:) +
   2*bits(2,:) + ...
   4*bits(3,:) + ...
   8*bits(4,:) + ...
   16*bits(5,:) + ...
   32*bits(6,:) + ...
   64*bits(7,:) + ...
   128*bits(8,:) ...
);

% end huffman_compress.m

92
function [recovered_data] = huffman_uncompress(compressed_data, bitlengths, the_length)
% [recovered_data] = huffman_uncompress(...
%   compressed_data, bitlengths, the_length)
% The recovered data will be a vector of length the_length.
% Uncompresses data packed using huffman_compress.
% See also IMREAD_COMPRESSED, HUFFMAN_COMPRESS.

% Change log:
% 1999-06-26: DAV: David Cary <d.cary@ieee.org> started.

% the bitlength for each code
codebook = compactcodeP(bitlengths);

bits = zeros(8, length(compressed_data));
for bit = 1:8;
    bits(bit,:) = bitget(compressed_data, bit);
end;
compressed_bits = bits(:,);

recovered_data = prefix_decode(codebook, compressed_bits, the_length);

% end huffman_uncompress.m