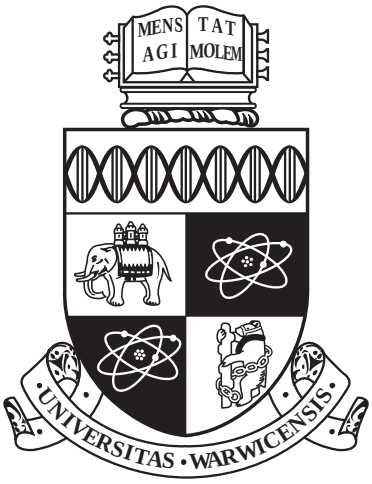

A Look At Data Compression With a Specific Focus on Image Compression

by

Thomas Preece

M469 Scholarly Report

Submitted to The University of Warwick

Mathematics Institute

April, 2013

THE UNIVERSITY OF
WARWICK

Contents

1 Introduction

In this paper I aim to talk about the methods used to compress different types of data such as text, audio, images and video and why they call for different methods with a brief description of how these methods actually work. I further aim to explore in greater detail image compression with a specific look at the JPEG and JPEG2000 image compression standards and the mathematics that underpins them.

Compression is very important in modern day life and you use it everyday without even realising. Every time you turn on a computer, most of the graphics it displays to you are stored as compressed files. When you use the internet, almost all of the data you retrieve is compressed. When you use your mobile phone, your voice is compressed before it is sent to the person on the other end. Even when you use a digital camera, the photos you take are compressed before they are stored on its memory card. Compression is needed in all of the above applications because each application has a bottleneck in either storage or bandwidth that compression helps overcome. For example the internet and phones have to send data over cables such as your telephone wire which have a maximum amount of data (called the bandwidth) that can be sent every second and the other applications have storage that has a maximum capacity. As increasing storage capacity and/or bandwidth is expensive we need to maximize the effectiveness of the storage and bandwidth by making our data smaller, in other words compressing the data. Just to give you an example, without compression a typical HD video would require a internet speed of around 500Mbps[1], the maximum line speed we have in the UK is 100Mbps with the average being 8Mbps making it impossible to watch without compression. Also the amount of data used today on the internet would stop the internet from functioning as the whole network would be at maximum capacity without compression and this would cause the internet to be incredibly slow.

I chose this topic mainly because I find it very interesting that we can use mathematics to make a piece of data smaller without losing anything from the content. A concept that seems very counter intuitive and made me want to find out how this is achieved. I also chose it as I love technology and because of how vital compression now is in so many of the things we take for granted nowadays.

2 Preliminaries

In this chapter we define some of the basic definitions and concepts we will use later on in this paper.

[1] $256 = 2^8$ colours per pixel, 1920x1080 pixels per frame and 30 frames per second. So we need $1920x1080x8x30$ bits per second

2.1 Basic Definitions

Definition 1. All modern day computers work in units called bits with a single bit being either a zero or a one. A byte is a collection of 8 bits. As computers only work in zeros and ones we say it uses a binary numeral system.

Computers represent all data in this binary numeral system, for example the letter a is represented as 01100001 in binary under the ASCII scheme. Each bit can have two values, zero or one so for n bits we can have 2^n possible combinations. Therefore if we had a list of 10 items we would like to represent in binary we would need at least 4 bits because $n = 4$ is the smallest number such that $2^n \geq 10$.

Definition 2. Compression is the science of taking an object represented in bits and using models and statistics to reduce the overall number of bits required to represent the same or similar object. We achieve compression via two routines, compression and uncompression. Compression is where we take our original object, which we will call the input data or source and shrink the overall number of bits to create a new object called the compressed data. Uncompression is where we feed in our compressed data and reverse to process to obtain our original object or a very close approximation of the object, which we call the output data or sink. Where routine or algorithm simply means a set of instructions that are performed in order.

Definition 3. Lossless compression is where the output data is exactly the same as the input data. Lossy compression is where we allow loss of data going from our input data to our output data.

Definition 4. A pixel is a single coloured square. Together a finite number of pixels make up an image.

Again these pixels can be represented in binary. For example if your image is made up of 10 pixels by 10 pixels, more commonly written 10x10 pixels, and each pixel can be one of 256 colours then we need 8 bits = 1 byte for each colour pixel as $2^8 = 256$ and as we have 100 pixels, we would join each byte representing each pixel together in a string. Hence to represent our image we would need a 100 bytes string.

Definition 5. We define a greyscale image as an image where all the pixels have a colour that is a shade of grey.

In the image compression algorithms that follow in this paper, we will apply them first to greyscale images, this allows us to understand the algorithm and then we can apply the algorithm to colour images using the same technique as greyscale images but applying it 3 times to the different components of the image. The three components are usually red, green and blue but there is other colour systems that we can use.

Definition 6. When working with data we often find it easier to think of the data as symbols as opposed to zeros and ones. We say that we encode a symbol when we convert the symbol to its binary representation (or binary code). We say that we decode a binary

number when we turn it back into its symbol.

2.2 Uniquely Decodable Codes

We get the following from [3]. Before we apply compression to a set of N symbols, the symbols are stored in a fixed-length encoding, that simply means that each symbol from our set is stored as a unique combination of zeros and ones of length $\lceil log_2 N \rceil$. So assume we had symbols a_0, a_1, a_2, as we have N=3 we must use a length 2 binary representation for each symbol. We can choose any of the combinations for any symbol but they all have the same length, eg. $a_0 = 00, a_1 = 01, a_2 = 10$. A common form of compression is to use smaller length binary representations for more common occurring symbols. For example say we had a sequence $a_0a_0a_0a_1a_0a_2$ then we could get compression by setting $a_0 = 1, a_1 = 11, a_2 = 10$ because then we only need 7 binary digits instead of 12. However the problem comes when we try to decode the sequence, with the above sequence we get 11111110 but then the first symbol could be a_0 or a_1 and we could decode the sequence as $a_1a_1a_1a_2$ or $a_0a_0a_0a_1a_0a_2$. We need a way to tell if we can uniquely decode any combination of encoded symbols.

Suppose we have a binary representations of a and b where a is of length k and b is of length n, $k < n$, then if the first k bits of b is the same as a we call a a prefix of b. The last $n - k$ bits of b are called the dangling suffix. Prefix codes are a subclass of uniquely decodable codes in which for any two elements in the binary representation of symbols of our set, neither of those two elements is a prefix of the other. It can be proved that for any uniquely decodable code there is a prefix code which has the same length so from now on we will only use prefix codes.

2.3 Comparing Compression Algorithms

The definitions in this subsection are constructed from [3], [5] and [4]. So far we have defined some basic concepts that allow us to talk about compression algorithms but how do we go about comparing different compression algorithms? Well the effectiveness of each algorithm depends on 3 factors: size of compressed data, distortion in compressed data and time an algorithm takes to run. We define distortion as changes in the data introduced by a compression algorithm that we can detect.

2.3.1 Information Theory

We first look at lossless compression. We note that because it is lossless we don't have any distortion so we look at if there is a limit to compression size. Our intuition tells us

that there must be a limit to the compression size otherwise we could represent everything with a single bit, which is clearly impossible.
Definition 7. We define the self-information of a symbol a as $I = -log_2 P(a)$ where $P(a)$ is the probability of a occurring in the source.

This makes intuitive sense as the symbol a contains more information if $P(a)$ is smaller because it is less expected to occur within the sequence of the source. We now define average self-information per symbol, which we call entropy.
Definition 8. We define Entropy as $H = -\sum_{i=0}^{n} p_i log_2(p_i)$ for a source with symbols $a_0, a_1, a_2, ..., a_n$ with probability of occurrence of each symbol $p_0, p_1, p_2, ..., p_n$ respectively.
Theorem 1. ***(Shannon's noiseless coding theorem)*** *For any binary prefix coded source with an average length of the source L and Entropy of source H we have* $L \geq H$

The previous theorem comes from a paper written by Shannon entitled 'A Mathematical Theory of Communication' [11]. This is an important result as it tells us that we can not lossless code our source to get a bits per symbol lower than entropy H. In other words we have a lower limit to how much we can losslessly compress a source.

2.3.2 Rate Distortion Theory

We next look at lossy compression. We have seen that lossless compression has a fixed limit on how much we can reduce the size of the data, with lossy compression we have no such limit. We could for example take an image and transmit a single pixel of it and it technically would be lossy compression and would reduce the size of the data by a huge amount but we introduce a huge amount of distortion. The idea behind lossy compression is that we want to reduce the size of the data as much as possible with minimal distortion for the particular application we are using the compression algorithm for.

Our biggest problem with testing compression algorithms for distortion is that with most applications, most of the compression comes from exploiting weaknesses in our imperfect senses such as eyes and ears. We will discuss this exploitation more in section 3 but as we are exploiting our own senses the only true way to tell if a compression algorithm introduces distortion is to get a large group of humans to compare a large set of output images from compression algorithms. The results can be erroneous due to human error and hence we have to have a huge sample size and this can be expensive. As this method is unrealistic in almost all situations there have been formulas developed to test compression algorithms that can be performed by computers. We briefly cover the main 3 formulae used for testing compression in images. The following definition is modified from [2].
Definition 9. We let $f(x, y)$ be the value of the input image at pixel in row x, column y . Similarly we let $g(x, y)$ be the value of the output image at pixel in row x, column y. We define $e(x, y) = f(x, y) - g(x, y)$ and then the Mean Square Error (MSE) is defined as $E_{ms} = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} e(x, y)^2$. We also define the Peak Signal to Noise Ratio (PSNR)

as $PSNR = 10\log_{10}\left(\frac{(2^A-1)^2}{E_{ms}}\right)$, where A is the number of bits required to represent each pixel. The lower the MSE is, the smaller the distortion. Similarly the higher the PSNR is, the smaller the distortion. When we are dealing with color images we simply sum up the errors over all pixels and all 3 components and then divide by $3MN$ instead of MN.

The following is obtained from [10]. The two above methods for computing distortion are not perfect and there exist examples of images that give poor MSE and PSNR numbers but are given good rating when humans are used to compare. No computational method can be perfect due to complexity of our eyes and the fact we still don't fully understand them[2], therefore we now introduce a complementary formula that looks at the structure of the image instead of individual pixel values and tries to improve upon the faults in the previous formula. By using multiple methods we can get a much better idea about how much distortion an algorithm introduces.

Definition 10. We define the structural similarity index measure (SSIM) for two windows x and y where x is an N by N window of the input image and y is an N by N window of the output image as

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$

where μ_x is the average of x, σ_x^2 is the variance of x, σ_{xy} is the covariance of x and y, $c_1 = (k_1 L)^2$, $c_2 = (k_2 L)^2$, L is the dynamic range of pixel values (that is $2^{\text{bits per pixel}} - 1$), and k_1, k_2 are typically chosen to be 0.01, 0.03 respectively. We also note that this is only applied to the luminance values of each pixel. We will discuss luminance more in subsection 5.4.3.

2.3.3 Complexity

So far we have shown that low distortion and low compressed size make a good compression algorithm. The final factor to consider is how long it takes to run the compression and uncompression algorithms, the efficiency of the algorithm. For example it's no good having 90% compression if it takes 2 years to compress a file, it would be much better to go with an algorithm that has 80% compression and only takes an hour.

3 Modelling

The following chapter is adapted from [5]. There are many different types of data but we can categorise all of them into the following four categories, text (for example digital books, scientific reports, etc), sound (for example music, podcasts, internet calls, etc), images and video. Compression is all about removing redundancies in data. Each different data type has different redundancies so different data types call for different methods to get optimum

compression. We can however often use many compression methods on all the types of data but they will be much less efficient then algorithms designed specifically for that data type. We now look over the main data types and briefly discuss what redundancies the data contains.

3.1 Text Data

We first look at text data, for this data we must use a lossless compression algorithm. This is due to the fact that the output data needs to be identical to the input. Take for example the phrase 'Do not enter', if we use a lossy compression we might end up with the phrase 'Do enter' which has a completely changed meaning, also for scientific data if the lossy algorithm changes some of the numbers or data slightly it might lead to a different hypothesis. The redundancies in text data often come from the fact that there is a lot of repeated words in any text and there is a high percentage of commonly used words and letters, this allows us to use a technique that groups together commonly used letters and assigns a single binary code to it. As the single binary code will be shorter than the sum of individual codes for each letter we obtain less data but we can still reconstruct the original exactly (provided we use prefix binary codes). We could also use statistics to assign smaller binary codes to the most common letters. You can see some lossless compression algorithms in chapter 4.

3.2 Sound Data

Another interesting data type is sound. This is commonly used over internet calls and on some phone lines. The interesting thing about sound is it is an analogue signal so to digitize it, we must take samples of the signal at different points and enough samples so that we cannot tell the different between the analogue and digital versions. The compression of this data relies on the fact that the human ear is nowhere near perfect and can only detect frequencies between 16-20Hz to 20000-22000Hz depending on person and age. Due to this limit we know that any sampling rate over 44000Hz does not produce any improvements in the playback of the sound and also removes frequencies over 220000Hz from our signal. We can further compress voice data because the vocal range is much smaller than 22000Hz so we can sample at a much lower rate, such as the telephone system that samples at 8000Hz. For each sample we also have to decide to what precision we store the amplitude, this is often 8 or 16 bits per sample, where 8 bits produces more compression but a worse reconstruction. We also note that for different frequencies the ear has a different minimum amplitude required for it to detect that frequency so we can achieve compression by deleting all points with amplitude below the threshold for the frequency at that point. We can also achieve compression through frequency masking and temporal masking. Frequency masking is where a sound that is above the threshold is masked by

another sound with nearby frequency because the second sound raises the threshold for the first sound and hence we can delete the first sound because we can no longer hear it. Temporal masking is where a high amplitude sound is preceded or followed in time by a lower amplitude sound of nearby or equal frequency, and if the time interval between the two sounds is short then the low amplitude sound will be inaudible and so it can be deleted.

3.3 Image Data

For images we rely on the fact that pixels in images tend to have neighbouring pixels that are highly correlated to the value of that pixel. We also use the fact that our eyes are not perfect and we find it hard to distinguish between very similar shades of a colour. We shall study much more about image compression and particular algorithms later on in this paper.

3.4 Video Data

Videos are just a sequence of images with sound so we can compress the sound separately using the methods mentioned in section 3.2. We then have the sequences of images left and we compress them by removing the same redundancies as we found for images above in section 3.3 and further compress by removing redundancies between each frame. We could compress the sequence of images by noting that the difference between frames in videos is very small so a good compression technique would be to only send a full frame every 60 frames and the differences between each consecutive frame. We could also use a compression where we look for large blocks of pixels that move position in consecutive frames and therefore we would only have to send the first frame, the size and position of where the block was and now is, and the changed pixels outside the moving blocks we have encoded.

4 Lossless Compression

In this section we cover two basic lossless compression methods: Run Length Coding which is used in the BMP and JPEG image file formats[8] and Huffman Coding which is also used in JPEG as well as many text and audio formats[3].

4.1 Run Length Encoding (RLE)

We use run length encoding to compress sources where there is a lot of consecutive repetition of symbols. When our sequence of symbols repeats 3 or more of the same symbol we

group them together by sending a unique repeat symbol, followed by the repeated symbol and finally by the number of repeats [5][8].

4.2 Huffman Coding

The algorithm uses basic probabilities and prefix codes to compress data. It is described as follows:
Step 1 - Order symbols $a_0, a_1, a_2, ..., a_n$ in order of probability, highest probability at top.
Step 2 - Choose the bottom two symbols a_i, a_j in the list, remove a_i and a_j from list and create a new symbol $a_{i,j}$ with probability $P(a_{i,j}) = P(a_i) + P(a_j)$ and add $a_{i,j}$ to list. Now sort symbol list again in order of probability, if there are multiple symbols with probability $P(a_{i,j})$ then put $a_{i,j}$ above all those symbols. Set $B(a_{i_k})$ to be $B(a_{i_k})$ with an added 0 to the left for each i_k where $i = i_1, i_2, ..., i_m$, $i_k \in \{0, ..., n\}$. Set $B(a_{j_k})$ to be $B(a_{j_k})$ with an added 1 to the left for each j_k where $j = j_1, j_2, ..., j_l$, $j_k \in \{0, ..., n\}$.
Step 3 - Repeat Step 2 until you have only 1 element in the list
Step 4 - Now $B(a_k)$ is the binary code for a_k $k = 0, ..., n$, and they make up a prefix code that compresses the any source using the symbols.

5 The Discrete Cosine Transformation

In this chapter we are going to look at the Discrete Cosine Transformation (DCT) and how it is used in JPEG compression. We shall start by discussing the Discrete Fourier Transform and from here we will derive the DCT. The DCT is an example of a subband coding compression, that is a transform that breaks a signal up into frequency bands. By breaking our image data up into frequency bands we compact the data into fewer entries due to the fact that pixels in images are highly correlated and we can discard or change higher frequency bands with little loss to overall image quality.

5.1 Discrete Fourier Transform

Definition 11. [7] [3] A discrete set of points $\{x_\tau : \tau = 0, ..., N-1\}$ is transformed into an N periodic sequence X_v by

$$X_v = \frac{1}{\sqrt{N}} \sum_{\tau=0}^{N-1} x_\tau e^{-i2\pi(v/N)\tau}$$

where N is number of samples. This is called the Discrete Fourier Transform (DFT).

Definition 12. [7] [3] A discrete set of points $\{X_v : v = 0, ..., N-1\}$ is transformed into an N periodic sequence x_τ by

$$x_\tau = \frac{1}{\sqrt{N}} \sum_{v=0}^{N-1} X_v e^{i2\pi(v/N)\tau}$$

where N is number of samples. This is called the Inverse Discrete Fourier Transform (IDFT).

Theorem 2. [7] *The IDCT is the inverse transformation of the DCT and vice versa.*

Proof. This proof is taken from [7]

We note that

$$\sum_{v=0}^{N-1} e^{-i2\pi(v/N)\tau} e^{i2\pi(v/N)\tau'} \quad = \quad \sum_{v=0}^{N-1} e^{-i2\pi(v/N)(\tau-\tau')} \quad = \quad \begin{cases} N & if\ \tau = \tau' \\ 0 & otherwise \end{cases}$$

hence

$$\begin{aligned} x_\tau = \frac{1}{\sqrt{N}} \sum_{v=0}^{N-1} X_v e^{i2\pi(v/N)\tau} &= \frac{1}{\sqrt{N}} \sum_{v=0}^{N-1} (\frac{1}{\sqrt{N}} \sum_{\tau'=0}^{N-1} x_{\tau'} e^{-i2\pi(v/N)\tau'}) e^{i2\pi(v/N)\tau} \\ &= \frac{1}{N} \sum_{\tau'=0}^{N-1} \sum_{v=0}^{N-1} x_{\tau'} e^{-i2\pi(v/N)(\tau'-\tau)} = \frac{1}{N} \sum_{\tau'=0}^{N-1} x_{\tau'} \sum_{v=0}^{N-1} e^{-i2\pi(v/N)(\tau'-\tau)} = \frac{1}{N} x_\tau N \end{aligned}$$

and similarly for X_v □

Definition 13. [7] A function $f(\tau)$ is even if $f(-\tau) = f(\tau)$ and odd if $f(-\tau) = -f(\tau)$.

Lemma 1. [7] The sum for an N point periodic odd function f is 0 i.e. $\sum_{i=0}^{N-1} f(i) = 0$

Proof. $f(j) = -f(-j) = -f(N-j)$ and if N is even $f(N/2) = -f(-N/2) = -f(N - N/2) = -f(N/2)$ hence $f(N/2) = 0$ and also $f(0) = -f(0)$ so $f(0) = 0$ □

Lemma 2. *An odd function multiplied with an even function is odd. An even function multiplied with an even function is even*

Proof. Let f be even, g odd then $f(x)g(x) = f(-x)(-g(-x))$. Let f and g be even then, $f(x)g(x) = f(-x)g(-x)$ hence result. □

Theorem 3. [7] *An even real function has an even real Discrete Fourier Transform*

Proof. Details of proof worked out from [13] and [7]

$$X_v = \frac{1}{\sqrt{N}} \sum_{\tau=0}^{N-1} x_\tau e^{-i2\pi(v/N)\tau} = \frac{1}{\sqrt{N}} \sum_{\tau=0}^{N-1} x_\tau [\cos(2\pi(v/N)\tau) + i\sin(2\pi(v/N)\tau)]$$
$$= \frac{1}{\sqrt{N}} \sum_{\tau=0}^{N-1} x_\tau \cos(2\pi(v/N)\tau) + \frac{i}{\sqrt{N}} \sum_{\tau=0}^{N-1} x_\tau \sin(2\pi(v/N)\tau)$$

As x_τ is even and $\sin(2\pi(v/N)\tau)$ is an N point periodic odd function, $\sum_{\tau=0}^{N-1} x_\tau \sin(2\pi(v/N)\tau) = 0$ by lemma 1 and 2. So $X_v = \frac{1}{\sqrt{N}} \sum_{\tau=0}^{N-1} x_\tau \cos(2\pi(v/N)\tau)$. Hence as x_τ and $\cos(2\pi(v/N)\tau)$ are even and real, X_v is even and real. □

5.2 Discrete Cosine Transformation (DCT)

5.2.1 Deriving the DCT

Definition 14. [13] We define the Discrete Cosine Transformation (DCT) also know as DCT-II on a discrete set of points $\{x_\tau : \tau = 0, ..., N-1\}$, as

$$X_v = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} x_i \cos(\frac{\pi(i+1/2)v}{N})$$

The reason we call this the DCT-II is because it is version 2 of 8 different variants of the DCT. We get 8 as each one is a different way to extend our N point sequence to a periodic sequence. We will later show that the DCT is derived from the DFT. As the sequence of points we put into the DCT can be such that it is not periodic we must extend it before we can use those points with the DFT. We assume we have a sequence of N points. From theorem 3 we see that our extension must also be even around zero, if it is odd around zero then we end up with the Discrete Sine Transformation (DST) which we will briefly discuss later in subsection 5.2.3. We also note that it could be even around zero by having the first point in our sequence at 0 or our first point at 1/2 as is shown in the left two graphs in figure 1. We have dealt with the extension at the left side of our N points but we don't have any restrictions on the extension at the right side of our N points so we could extend the right side via an even extension or odd extension as show in the two right graphs in figure 1 and each of those extensions could be around the Nth point or half way between the Nth and (N+1)th point. So there is 4 choices at the right boundary and 2 at the left boundary hence we have 8 different variants of the DCT.[13]

Theorem 4. [13] *The Discrete Cosine Transformation on N points is equivalent to a 4N point Discrete Fourier Transformation*

Proof. We take as our N points $\{y_m : m = 0, ..., N-1\}$ and define our 4N points

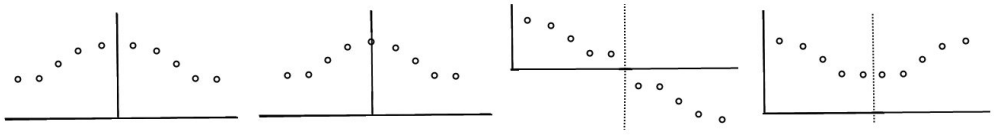

Figure 1: From left to right - extension around point half way between point and next one, extension around point, odd extension, even extension

$\{x_m : m = 0, ..., 4N-1\}$ by $x_{2a} = 0, x_{2a+1} = y_a$ for $0 \leq a < N$, $x_{2N} = 0$, $x_{4N-a} = x_a$ for $0 < a < 2N$. Then we have a real even function and the DFT reduces to $X_\tau = \frac{1}{\sqrt{4N}} \sum_{v=0}^{4N-1} x_v \cos(2\pi(v/4N)\tau)$. When we remove the zero values we get $X_\tau = \frac{1}{2\sqrt{N}} \sum_{v=0}^{2N-1} x_{(2v+1)} \cos(\pi((2v+1)/2N)\tau)$, the values repeat after $v = N-1$ so we can take half the values and multiply by 2. Hence we get

$$X_\tau = \frac{2}{2\sqrt{N}} \sum_{v=0}^{N-1} x_{(2v+1)} \cos(\pi((v+1/2)/N)\tau) = \frac{1}{\sqrt{N}} \sum_{v=0}^{N-1} y_v \cos(\pi((v+1/2)/N)\tau)$$

□

This is an important theorem because it allows us to use a modified version of the Fast Fourier Transform (FFT) to calculate the DCT. This allows us to calculate the DCT in a much quicker time, making the compression algorithm much faster. The transform is sometimes changed to make it orthogonal by multiplying the first term by $\frac{1}{\sqrt{2}}$ and all terms by $\sqrt{2}$, this breaks the direct correspondence with the DFT but we can still use the FFT. We also note that this transform is for a 1 dimensional data set. As images are 2-dimensional we simply apply it once to the rows of pixels and then we apply it again to the columns of the resulting matrix.

5.2.2 Why The DCT Is Used For Image Compression

We can think of the basis of cosine waves of increasing frequency as a coordinate system. Then the DCT basically takes our data from Euclidean coordinates to our new cosine coordinate system. In an image when we take a row or column, the values slowly change over the course of that row or column so when we use the DCT we end up getting most of the content of the image in the first few coordinates as the first few coordinates are for slow frequency cosine waves, those representing slow changes in the data. We also note that there is very little change between individual pixels so the last coordinates will be small or zero, due to there being little to none fast change in the data. This is called energy compaction as we take several values and compact them mostly into a few values.

This works well for compression as we now have a lot of small values with a few zeros

and then a few important values. Depending on the level of compression we can discard certain amounts of the small values. These values represent high frequency changes in the data, and as our eyes are not as sensitive at detecting high frequency changes in pixels values we can easily discard them without noticeable difference in the reconstructed image. Therefore we have less values required to represent our image hence we obtain compression.

5.2.3 DCT vs DST vs DFT

The following is adapted from [3]. The DCT is evenly extended at 0, causing the data to be smooth at the boundaries, whereas the DFT and DST both have sharp discontinuities in the values at the boundaries. When we have sharp discontinuities at the boundaries we have to have large values for the high frequency components to compensate. Then as we only have sharp discontinuities at the boundaries, we have to use the other frequency components to compensate for the high frequency components elsewhere in the data. This causes us to have high frequency components that we can't get rid of and hence we have more values required to represent our image. Therefore the DCT is the best option out of the three for image compression.

5.3 Fast Fourier Transform (FFT)

The following section is adapted from [13], [19], [20]. We put the $N = 2^p$ point DFT into matrix form F, and then we apply a matrix P that moves the even columns before the odd columns by post multiplying F with P. For example if we had matrix with columns c_0, c_1, c_2, c_3 then post multiplying with P would result in the matrix with columns c_0, c_2, c_1, c_3. We then note that the top left quarter of FP is the matrix for the DFT on 2^{p-1} points, we call it f. We also note that the bottom left matrix is also f. We define the diagonal matrix D as

$$D = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & w & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & w^{2^{p-1}-1} \end{bmatrix}$$

then the top right quarter of FP is Df and the bottom right quarter is $-Df$. So we have the following factorisation

$$F = \begin{bmatrix} f & Df \\ f & -Df \end{bmatrix} P^{-1} = \begin{bmatrix} I_{2^{p-1}} & D \\ I_{2^{p-1}} & -D \end{bmatrix} \begin{bmatrix} f & 0_{2^{p-1}} \\ 0_{2^{p-1}} & f \end{bmatrix} P^{-1}$$

Original DCT	FFT Method	Improvement	% Improvement
0.7988985682	0.0096011616	0.7892974067	98.7982001767
1.0030654822	0.0078033701	0.9952621121	99.2220477919
0.8640956181	0.0067587858	0.8573368322	99.2178196859
1.0219844602	0.0072313252	1.014753135	99.2924231734

Figure 2: Table of computation speed for different DCT methods

We can then do the same for f and keep iterating until we get the 2 point DFT. The following is an example of the FFT on a $N = 2^2$ point DFT. We let $w = e^{2\pi i/4}$ then

$$F\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & w^2 & w & w^3 \\ 1 & w^4 & w^2 & w^6 \\ 1 & w^6 & w^3 & w^9 \end{bmatrix}\begin{bmatrix} x_0 \\ x_2 \\ x_1 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & w \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -w \end{bmatrix}\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & w^2 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & w^2 \end{bmatrix}\begin{bmatrix} x_0 \\ x_2 \\ x_1 \\ x_3 \end{bmatrix}$$

We have gone from needing N^2 multiplications and $N(N-1)$ additions to $2N \log N$ multiplications and $N \log N$ additions, hence we have a big computational saving for sufficiently large N.

In appendix B listing 1 is a Matlab program I have created to calculate the speed difference between the original DCT and the 4N FFT version of the DCT and as figure 2 shows, we get a very large increase in efficiency by using the FFT.

5.4 JPEG Algorithm

This section and all subsections excluding 5.4.4 are adapted from [3] and [5]. JPEG is a compression that is used extensively, it is used by around 70% of all websites on the internet[2], by most cameras and when an image is stored on your computer it is mostly likely in JPEG format. The compression algorithm has the following steps for a greyscale image:

1. Split the image into 8x8 blocks of pixels, if the image isn't exactly a multiple of 8 in width or height we pad any block with size less then 8x8 with zeros making it up to size 8x8. We will chuck away these extra pixels when we decode the image.
2. On each 8x8 block of pixels we level shift (subtract $2^{BitsPerPixel-1}$) the values and then apply the 2 dimensional DCT.
3. We then quantize the obtained values using quantization tables
4. We encode the quantized values using differential coding for the DC coefficients and we code the AC coefficients by zigzagging them and using a mixture of run length

[2]http://w3techs.com/technologies/details/im-jpeg/all/all

encoding and Huffman codes.

5. The sequence of bits obtained in step 4 is now our compressed image.

We decompress the image by

1. Decode the steam of bits into quantized values
2. Reverse the quantization process using the quantization tables
3. Apply the IDCT to get back a similar version of the original image.

We now explain the quantization and coding steps of the above algorithm in more detail below.

5.4.1 Quantization

Up until this point in the compression algorithm (step 3 above) we haven't lost any data as the DCT is reversible (excluding any rounding errors that have occurred) so this step is where most of the compression comes from. We have a fixed 8x8 block of values 'A' called a quantization matrix and for each value of our 8x8 block of pixels we round to the nearest multiple of the corresponding element in A. More precisely for a 8x8 block of pixels P, and an 8x8 block quantization matrix A, we get a new matrix Q such that $Q_{ij} = \left\lfloor \frac{P_{ij}}{A_{ij}} \right\rfloor$. When we reverse the quantization we apply the reverse operation $P'_{ij} = Q_{ij}A_{ij}$. This is an important step as we more heavily quantize, that is use higher value A_{ij}'s for the values towards the bottom right of our matrix P. This is because those values are high frequency coefficients and hence they are less important to reconstruct a similar image than the values towards the top left.

5.4.2 Coding

For each 8x8 block we define the top left value as the DC coefficient and the remaining values as AC coefficients, where the DC coefficient is light red and the AC coefficients are light yellow in figure 3. We encode the DC coefficients by using differential coding on all the 8x8 blocks of pixels and then using a modified Huffman coding on the values we obtain. More specifically, we assume our image consists of ordered 8x8 blocks of pixels $A^{(1)}, A^{(2)}, ..., A^{(n)}$. Then for the DC coefficient in $A^{(1)}$ we would send the code for the value $A^{(1)}_{0,0}$ which consists of the prefix code for the category that $A^{(1)}_{0,0}$ is in (that is k if in $[-2^k+1, -2^{k-1}] \cup [2^{k-1}, 2^k-1]$ where $k > 0$, or 0 if $A^{(1)}_{0,0} = 0$) followed by k bits representing which number $A^{(1)}_{0,0}$ is in within the category range $[-2^k + 1, -2^{k-1}] \cup [2^{k-1}, 2^k - 1]$. We generate the prefix codes for the categories by applying Huffman coding to the categories. Then for the DC coefficient in $A^{(j)}$ we do the same as we did for $A^{(1)}$ but send the code for the value $A^{(j)}_{0,0} - A^{(j-1)}_{0,0}$. The encoding of AC coefficients is more complicated and we

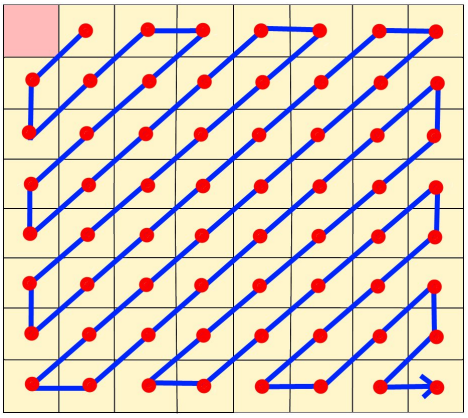

Figure 3: Coding order of an 8x8 block of pixels

first order the coefficients in a zigzag order as shown in figure 3, going from the top left to bottom right. The coefficients are then converted to a modified version of run length encoding where for each non zero coefficient we have a symbol (a,b) followed by a binary sequence c. Where a is the number of zeros preceeding this coefficient, b is the category of the coefficient as explained above and c is a binary sequence representing which number the coefficient is in the category range. If we have a run of more than 15 zeros we simply use the symbol (15,0) to represent 15 zeros. We then use Huffman coding described in section 4.2 on these (a,b) symbols to get a prefix code for them. Then we encode by sending the prefix code for (a,b) followed by c. If we encounter a point where the remaining entries are all zero we send the (0,0) symbol, encode the symbols and move to next 8x8 block.

5.4.3 Colour Images

If we want to apply the algorithm to colour images we could simply repeat the above algorithm 3 times on the red, green and blue components of each pixel. But the problem with that is the 3 components still have some correlation between them so instead we often break up the pixels into YCrCb components, that is Y represents luminance and Cr,Cb are the chrominance. Where luminance is the lightness of each pixel, in other words the value if it was a black and white image, and chrominance represents the colour part of the image. This allows us to get better compression as we can more heavily quantize the chrominance matrices as the eyes are less sensitive to those components. The exact conversion matrices between RGB and YCrCb are used in the Matlab code below and can be found in appendix A.1 for those interested.

5.4.4 Example

I have made Matlab code that runs through the JPEG algorithm in listing 2 of appendix B. This algorithm doesn't calculate the exact sequence of bits that the compressed JPEG

Figure 4: Left:Uncompressed Image, Right:JPEG Compressed Image

Uncompressed Size	JPEG Size	JPEG Savings	JPEG SSIM
9437184	656909	0.9303914176	0.9111949149
9437184	531438	0.9436868032	0.8943573813
9437184	393837	0.9582675298	0.9396635995
9437184	491737	0.9478936725	0.9150246166

Figure 5: Comparison of JPEG compressed images and their uncompressed counterparts. Full results can be found in appendix A.3.

image would be stored as but it does produce the size of that sequence of bits and the image we would get when we uncompress the compressed JPEG image. Figure 4 shows an original image (image 5 in full results contained in appendix A.3) and the compressed version obtained using the Matlab code. It's important to note that the original takes up 9437184 bits as opposed to the compressed version that only takes up 871970 bits. That is a saving of over 90%, with what you can see is very little loss of quality.

We now use listings 3, 4 and 5 in appendix B to compare file sizes of JPEG compression against uncompressed images. Running the script on our image set [33] and [34] we obtain the results in figure 5, as we can see each image saves at least 90% of the original size and they are all close to a SSIM of 1 so the compressed JPEGs are all fairly good representations of the original data.

6 Wavelets

Wavelets let us analyse data in both the time and frequency domain simultaneously [4] as opposed to the DCT which just analyses data in the frequency domain. We first define multiresolution analysis and some related theorems before moving on to how to derive the Daubechies wavelets and how to use wavelets in compression.

6.1 Multiresolution Analysis

We first start with some very basic definitions, which should already be familiar to the reader. Then we state without proof some theorems that will be used later on, the proofs of which can be found in [16].

Definition 15. $\delta_{kl} = \begin{cases} 0 & k \neq l \\ 1 & k = l \end{cases}$

Definition 16. $L^2(\mathbb{R})$ is the space of functions that are square integrable. In other words $L^2(\mathbb{R}) = \{f : \mathbb{R} \to \mathbb{R}; \int |f(t)|^2 dt < \infty\}$. The inner product on this space is $\langle f, g\rangle = \int f(t)\overline{g(t)}dt$.

Theorem 5. *Suppose V is an inner product space and V_0 is an N-dimensional subspace with orthonormal basis $\{e_1, e_2, ..., e_N\}$. The orthogonal projection v_0 of a vector $v \in V$ onto V_0 is given by*

$$v_0 = \sum_{j=1}^{N} \alpha_j e_j \quad with\ \alpha_j = \langle v, e_j\rangle$$

Theorem 6. *Let V be a complex inner product space with an orthonormal basis $\{u_k\}_{k=1}^{\infty}$, then if $f \in V, g \in V$ with the expansions*

$$f = \sum_{k=1}^{\infty} a_k u_k \qquad g = \sum_{k=1}^{\infty} b_k u_k$$

then we get

$$\langle f, g\rangle = \sum_{k=1}^{\infty} a_k \overline{b}_k$$

We now define a multiresolution analysis which is a collection of spaces V_j that are called approximation spaces. Any function in $L^2(\mathbb{R})$ can be approximated by a function in V_j but as j gets larger the approximations get better.

Definition 17. [16] Let V_j, $j \in \mathbb{Z}$ be a sequence of subspaces of functions in $L^2(\mathbb{R})$. The collection of spaces $\{V_j : j \in \mathbb{Z}\}$ is called a multiresolution analysis (MRA) with scaling function ϕ if the following hold:

1. $V_j \subset V_{j+1}$
2. $\overline{\cup V_j} = L^2(\mathbb{R})$
3. $\cap V_j = \{0\}$
4. The function $f(x)$ belongs to V_j if and only if the function $f(2^{-j}x)$ belongs to V_0
5. The function ϕ belongs to V_0 and the set $\{\phi(x-k), k \in \mathbb{Z}\}$ is an orthonormal basis for V_0

It is also worth noting that a different choice of ϕ may result in a different MRA.

Theorem 7. [16] *Suppose $\{V_j : j \in \mathbb{Z}\}$ is a MRA with scaling function ϕ. Then for any $j \in \mathbb{Z}$, the set of functions*

$$\{\phi_{jk}(x) = 2^{j/2}\phi(2^j x - k) : k \in \mathbb{Z}\}$$

is an orthonormal basis for V_j

Proof. Taken from [16]. Take any function $f(x) \in V_j$ then by property 4 above $f(2^{-j}x) \in V_0$ by property 5 above $f(2^{-j}x)$ is a linear combination of $\{\phi(x-k), k \in \mathbb{Z}\}$. We replace x by $2^j x$ and then $f(x)$ is a linear combination of $\{\phi(2^j x-k), k \in \mathbb{Z}\}$, hence $\{\phi_{jk}(x) : k \in \mathbb{Z}\}$ is a basis. We now show its orthonormal by making a change of variables $y = 2^j x$

$$2^j \int \phi(2^j x - k)\overline{\phi(2^j x - l)}dx = 2^j \int \phi(y - k)\overline{\phi(y - l)}\frac{dy}{2^j} = \delta_{kl}$$

as property 5 above says that $\{\phi(x - k), k \in \mathbb{Z}\}$ is an orthonormal basis. □

We now prove that there is a relation between the basis for V_j and V_{j+1}.
Theorem 8. [16] *Suppose $\{V_j : j \in \mathbb{Z}\}$ is a MRA with scaling function ϕ. The the following scaling relation holds:*

$$\phi(x) = \sum_{k\in\mathbb{Z}} p_k \phi(2x - k), \text{ where } p_k = 2\int_{-\infty}^{\infty} \phi(x)\overline{\phi(2x - k)}dx$$

Moreover, we also have

$$\phi(2^{j-1}x - l) = \sum_{k\in\mathbb{Z}} p_{k-2l}\phi(2^j x - k) \tag{1}$$

Proof. Taken from [16]. We use theorem 5 and note that $\phi(x) = \sum \tilde{p_k}\phi_{1k}(x)$ must hold for some $\tilde{p_k}$ as $\phi(x)$ belongs to $V_1 \supset V_0$. Then by theorem 5 we have $\tilde{p_k} = \langle \phi, \phi_{1k} \rangle = 2^{1/2}\int_{-\infty}^{\infty} \phi(x)\overline{\phi(2x - k)}dx$. We let $p_k = 2^{1/2}\tilde{p_k}$, then we get the first equation above. We get then second equation by replacing x by $2^{j-1}x - l$, and then changing the index of the sum $\phi(2^{j-1}x - l) = \sum_{k\in\mathbb{Z}} p_k\phi(2^j x - 2l - k) = \sum_{m\in\mathbb{Z}} p_{m-2l}\phi(2^j x - m)$. □

We call the above relation between $\phi(x)$ and $\phi(2x - k)$ the scaling relation. Also the formulation in p_k will become very useful as later on you will see that we can perform the wavelet transform knowing only the $p_k's$. This is very important as most of the useful wavelets cannot be explicitly written. We now derive some properties of these p_k.
Theorem 9. [16] *Suppose $\{V_j : j \in \mathbb{Z}\}$ is a MRA with scaling function ϕ. Then provided the scaling relation can be integrated term wise, the following hold:*

1. $\sum_{k\in\mathbb{Z}} p_{k-2l}\overline{p_k} = 2\delta_{l0}$

2. $\sum_{k\in\mathbb{Z}}|p_k| = 2$
3. $\sum_{k\in\mathbb{Z}} p_k = 2$
4. $\sum_{k\in\mathbb{Z}} p_{2k} = 1$ *and* $\sum_{k\in\mathbb{Z}} p_{2k+1} = 1$

Proof. For 1 we use theorem 6 on $\phi(\frac{x}{2} - l) = \sum p_{k-2l}\phi(x-k)$ and $\phi(\frac{x}{2}) = \sum p_k\phi(x-k)$. Hence $\sum p_{k-2l}\overline{p_k} = \left\langle \phi(\frac{x}{2}), \phi(\frac{x}{2} - l)\right\rangle = 2\int \phi(y-l)\phi(y)dy = 2\delta_{l0}$ by change of variables and orthonormal basis of V_0. The proof of 2, 3 and part of 4 is taken from [16]. 2 is yielded by setting $l = 0$ in 1. For part 3 we integrate both sides of the formula in theorem 8 and make a change of variables $t = 2x - k$ and $dx = dt/2$ to get

$$\int_{-\infty}^{\infty} \phi(x)dx = \frac{1}{2}\sum_{k\in\mathbb{Z}} p_k \int_{-\infty}^{\infty} \phi(t)dt$$

As $\int \phi(x)dx$ cannot be zero, otherwise we wouldn't be able to approximate functions with $\int f(t)dt \neq 0$, we can simplify the above equation to $1 = \frac{1}{2}\sum_{k\in\mathbb{Z}} p_k$. To obtain the forth equation, we use 1 and replace l by $-l$ and sum over l and then split up the odd and even terms.

$$2 = \sum_{l\in\mathbb{Z}}\sum_{k\in\mathbb{Z}} p_{k+2l}\overline{p}_k = \sum_{k\in\mathbb{Z}}(\sum_{l\in\mathbb{Z}} p_{2k+2l})\overline{p}_{2k} + \sum_{k\in\mathbb{Z}}(\sum_{l\in\mathbb{Z}} p_{2k+1+2l})\overline{p}_{2k+1}$$

we then replace l with $l - k$ and get

$$2 = \sum_{k\in\mathbb{Z}}\overline{p}_{2k}\sum_{l\in\mathbb{Z}} p_{2l} + \sum_{k\in\mathbb{Z}}\overline{p}_{2k+1}\sum_{l\in\mathbb{Z}} p_{2l+1} = \left|\sum_{k\in\mathbb{Z}} p_{2k}\right|^2 + \left|\sum_{k\in\mathbb{Z}} p_{2k+1}\right|^2$$

This final bit is my own work. We finally let $E = \sum_{k\in\mathbb{Z}} p_{2k}$ and $O = \sum_{k\in\mathbb{Z}} p_{2k+1}$. Then we have $\Re(E) + \Re(O) = 2$, $\Im(E) + \Im(O) = 0$ and $\Re(E)^2 + \Im(E)^2 + \Re(O)^2 + \Im(O)^2 = 2$. Then we get $\Re(E)^2 + \Re(O)^2 = 2 - 2\Im(E)^2$. We then see that $\Re(E) = \Re(O) = 1$ and $\Im(E) = \Im(O) = 0$ is the only solution by looking at the graph in figure 6 and seeing that when $\Im(E) > 0$, $\Re(E)^2 + \Re(O)^2 = 2 - 2\Im(E)^2$ doesn't touch $\Re(E) + \Re(O) = 2$. □

Theorem 10. [16] *Suppose $\{V_j : j \in \mathbb{Z}\}$ is a MRA with scaling function $\phi(x) = \sum_{k\in\mathbb{Z}} p_k\phi(2x-k)$ with p_k as above. Let W_j be the span of $\{\psi(2^j x - k) : k \in \mathbb{Z}\}$ with*

$$\psi(x) = \sum_{k\in\mathbb{Z}} (-1)^k \overline{p_{1-k}}\phi(2x-k)$$

Then $W_j \subset V_{j+1}$ is the orthogonal complement of V_j in V_{j+1}. Furthermore $\{\psi_{jk}(x) := 2^{j/2}\psi(2^j x - k) : k \in \mathbb{Z}\}$ is an orthonormal basis for the W_j. Moreover we have

$$\psi(2^{j-1}x - l) = \sum_{k\in\mathbb{Z}} (-1)^k \overline{p_{1-k+2l}}\phi(2^j x - k) \tag{2}$$

Proof. We prove the theorem for $j = 0$ and then use the scaling condition to get result for

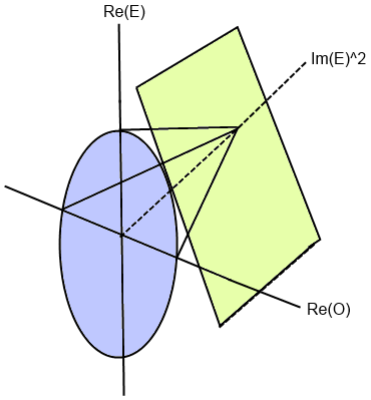

Figure 6: Shows the graph of $\Re(E)+\Re(O)=2$(yellow) and $\Re(E)^2+\Re(O)^2=2$(blue)

$j>0$. Using theorem 6 and

$$\psi_{0m}=\sum_{k\in\mathbb{Z}}(-1)^k\overline{p_{1-k}}\phi(2x-2m-k)=\frac{1}{\sqrt{2}}\sum_{k\in\mathbb{Z}}(-1)^{k-2m}\overline{p_{1-k+2m}}\sqrt{2}\phi(2x-k)$$

we get $\langle\psi_{0m},\psi_{0l}\rangle=\frac{1}{2}\sum_{k\in\mathbb{Z}}\overline{p_{1-k+2m}}p_{1-k+2l}$. The remainder of this proof is taken from [16]. We now make the change of index $k'=1-k+2m$ and use part 1 of theorem 9 to get

$$\langle\psi_{0m},\psi_{0l}\rangle=\frac{1}{2}\sum_{k'\in\mathbb{Z}}\overline{p_{k'}}p_{k'+2l-2m}=\delta_{m-l,0}=\delta_{m,l}$$

so $\{\psi_{jk}(x):=2^{j/2}\psi(2^jx-k):k\in\mathbb{Z}\}$ is orthonormal. By using the scaling relation and changing the index we get $\phi_{0l}=\frac{1}{\sqrt{2}}\sum_{k\in\mathbb{Z}}p_{k-2l}\sqrt{2}\phi(2x-k)$ and by theorem 6 we get

$$\langle\phi_{0l},\psi_{0m}\rangle=\frac{1}{2}\sum_{k\in\mathbb{Z}}(-1)^kp_{1-k+2m}p_{k-2l}=0$$

as the terms $k=l+m-j$ and $k=l+m+j+1$ cancel for all $j\geq 0$. Therefore ψ_{0k} is orthogonal to V_0 for all $k\in\mathbb{Z}$. We lastly want to show that V_1 is spanned by translates of ϕ and translates of ψ, as this will show that any element of W_0 is a combination of translates of ψ. So we must show that for every j we have

$$\phi(2x-j)=\sum_k a_k\phi(x-k)+b_k\psi(x-k)$$

If the a_k exist they will be given by $\int\phi(2y-j)\overline{\phi(y-k)}dy=\int\sum_l\phi(2y-j)\overline{p_l}\overline{\phi(2y-2k-l)}dy$ by the scaling function. As $\{\phi_{1k}(x)=2^{1/2}\phi(2x-k):k\in\mathbb{Z}\}$ is orthonormal, we have $a_k=\frac{1}{2}\overline{p_{j-2k}}$. Similarly $b_k=\frac{1}{2}(-1)^jp_{1-j+2k}$ by using the formula for ψ stated in the theorem. By substituting in the a_k's and b_k's and using the scaling relation and the formula

for ψ in the theorem we get

$$\phi(2x-j) = \frac{1}{2}\sum_{k,l}((-1)^{j+l}p_{1-j+2k}\overline{p_{1-l}} + \overline{p_{j-2k}}p_l)\phi(2x-2k-l)$$

and must show that when $l = j-2k$ we have

$$\sum p_{1-j+2k}\overline{p_{1-j+2k}} + \overline{p_{j-2k}}p_{j-2k} = 2 \tag{3}$$

and when $l = j-2k+t$ for $t \neq 0$ we have

$$\sum_k (-1)^t p_{1-j+2k}\overline{p_{1-j+2k-t}} + \sum_k \overline{p_{j-2k}}p_{j-2k+t} = 0 \tag{4}$$

By using $\gamma = 1-j+2k$ and $\gamma = j-2k$ we get $\sum_\gamma \overline{p_\gamma}p_\gamma$ for equation 3 which is 2 by theorem 9. We handle equation 4 differently for odd and even values of t. If t is odd say $t = 2s+1$ we can use the change of variables $k' = s+j-k$ to get

$$\sum_{k'} -\overline{p_{j-2k'}}p_{j-2k'+t} + \sum_k \overline{p_{j-2k}}p_{j-2k+t} = 0$$

If t is even say $t = 2s$, then we use $k' = -k+j+s$ to get

$$\sum_k p_{1-j+2k}\overline{p_{1-j+2k-t}} + \sum_{k'} p_{-j+2k'}\overline{p_{-j+2k'-t}}$$

using $\gamma = 1-j+2k$ for the first sum above and $\gamma = -j+2k'$ for the second sum we get $\sum_\gamma p_\gamma \overline{p_{\gamma-2s}}$ which is zero by part 1 of theorem 9. We obtain the last formula in the theorem by replacing x by $2^{j-1}x - l$ and adjusting the index. □

By the above theorem we have $V_j = W_{j-1} \oplus V_{j-1}$ and we can repeatedly apply this to obtain the next theorem. This is important because it allows us to use MRA for any function in $L^2(\mathbb{R})$ to decompose the function into a set of functions from the various W_j's. We also note that the function ψ above is called a wavelet, it may also be called the mother wavelet and the scaling function ϕ called the father wavelet.

Theorem 11. [16] *Suppose $\{V_j : j \in \mathbb{Z}\}$ is an MRA with scaling function ϕ. Let W_j be the orthogonal complement of V_j in V_{j+1}. Then*

$$L^2(\mathbb{R}) = ... \oplus W_{-1} \oplus W_0 \oplus W_1 \oplus ...$$

In particular, each $f \in L^2(\mathbb{R})$ can be uniquely expressed as a sum $\sum_{k=-\infty}^{\infty} w_k$ with $w_k \in W_k$ and where the $w_k's$ are mutually orthogonal. Equivalently, the set of all wavelets, $\{\psi_{jk}(x)\}_{j,k\in\mathbb{Z}}$, is an orthonormal basis for $L^2(\mathbb{R})$.

We now group together everything we have proved into a theorem we can use to decompose

and recompose a signal.

Theorem 12. [16] *Suppose that we have a signal f that is already in one of the approximation spaces V_j. Then we can represent f in the following two ways $f = \sum_{k\in\mathbb{Z}}\langle f, \phi_{j,k}\rangle\phi_{j,k}$ and $f = \sum_{k\in\mathbb{Z}}\langle f, \phi_{j-1,k}\rangle\phi_{j-1,k} + \sum_{k\in\mathbb{Z}}\langle f, \psi_{j-1,k}\rangle\psi_{j-1,k}$ and the decomposition formula and reconstruction formula are as follows:*

$$Decomposition: \begin{cases} \langle f, \phi_{j-1,l}\rangle = 2^{-1/2}\sum_{k\in\mathbb{Z}}\overline{p_{k-2l}}\,\langle f, \phi_{j,k}\rangle \\ \langle f, \psi_{j-1,l}\rangle = 2^{-1/2}\sum_{k\in\mathbb{Z}}(-1)^k p_{1-k+2l}\,\langle f, \psi_{j,k}\rangle \end{cases}$$

$$Reconstruction: \ \langle f, \phi_{jk}\rangle = 2^{-1/2}\sum_{l\in\mathbb{Z}} p_{k-2l}\langle f, \phi_{j-1,l}\rangle + 2^{-1/2}\sum_{l\in\mathbb{Z}}(-1)^k\overline{p_{1-k+2l}}\langle f, \psi_{j-1,l}\rangle$$

Let $a_k^{j-1} = 2^{(j-1)/2}\langle f, \phi_{j-1,k}\rangle$ and $b_k^{j-1} = 2^{(j-1)/2}\langle f, \psi_{j-1,k}\rangle$ then the following are equivalent:

$$Decomposition: \begin{cases} a_l^{j-1} = 2^{-1}\sum_{k\in\mathbb{Z}}\overline{p_{k-2l}}a_k^j \\ b_l^{j-1} = 2^{-1}\sum_{k\in\mathbb{Z}}(-1)^k p_{1-k+2l}a_k^j \end{cases}$$

$$Reconstruction: \ a_k^j = \sum_{l\in\mathbb{Z}} p_{k-2l}a_l^{j-1} + \sum_{l\in\mathbb{Z}}(-1)^k\overline{p_{1-k+2l}}b_l^{j-1}$$

Proof. To obtain the decomposition formula we first adjust equations 1 and 2 to get $\phi_{j-1,l} = 2^{-1/2}\sum_k p_{k-2l}\phi_{j,k}$ and $\psi_{j-1,l} = 2^{-1/2}\sum_k(-1)^k\overline{p_{1-k+2l}}\phi_{j,k}$. Then applying the inner product with f to both of these formula yields the required formula. We know $V_j = V_{j-1}\oplus W_{j-1}$ so $\phi_{j,k} = \sum_l\langle\phi_{j,k}, \phi_{j-1,l}\rangle\,\phi_{j-1,l} + \sum_l\langle\phi_{j,k}, \psi_{j-1,l}\rangle\,\psi_{j-1,l}$, again applying the above two formula for $\phi_{j-1,l}$ and $\psi_{j-1,l}$ and using the orthonormality of $\phi_{j,k}$ we get the required reconstruction formula. □

So we now have formula to decompose a set of data and reconstruct it again, but where do we get our initial a_k^j from? In its current form the a_k^j are hard to calculate, especially when the mother and father wavelet cannot explicitly be written. We find an easier way to obtain the a_k^j in the next section.

6.2 Decomposition Algorithm

This section is based on [16]. There are 3 steps in decomposing a signal: initialization, iteration and termination. We explain each one in further detail below.

6.2.1 Initialization

The first step involves choosing $f_i \in V_i$ to best approximate f. We approximate f in V_j by using the orthogonal projection of f onto V_j. That is by theorem 5 we have

$$f_j(x) = \sum_{k\in\mathbb{Z}} a_k^j \phi(2^j x - k), \text{ where } a_k^j = 2^j \int_{-\infty}^{\infty} f(x)\overline{\phi(2^j x - k)}dx \tag{5}$$

Definition 18. [16] A function has compact support if the function is identically zero outside a finite region.

We now prove a theorem that allows us to obtain the a_k^i's directly from the function and provided we know $\int \overline{\phi(x)}dx = 1$, we do not require any complicated evaluation of ϕ.
Theorem 13. [16] *Suppose $\{V_j : j \in \mathbb{Z}\}$ is an MRA with scaling function ϕ that is compactly supported. If $f \in L^2(\mathbb{R})$ is continuous, then, for j sufficiently large we have*

$$a_k^j = 2^j \int_{-\infty}^{\infty} f(x)\overline{\phi(2^j x - k)}dx \approx m f(k/2^j)$$

where $m = \int \overline{\phi(x)}dx$

Proof. Taken from [16]. Since ϕ is compactly supported it is nonzero on an interval $|t| \leq M$. Thus the interval of integration of the integral for a_k^j in equation 5 is $\{x : \left|2^j x - k\right| \leq M\}$. We change variables in integral from x to $t = 2^j x - k$ to obtain

$$a_k^j = \int_{-M}^{M} f(2^{-j}t + 2^{-j}k)\overline{\phi(t)}dt$$

When j is large $2^{-j}t + 2^{-j}k \approx 2^{-j}k$ for $t \in [-M, M]$. As f is uniformly continuous on any finite interval we have $f(2^{-j}t + 2^{-j}k) \approx f(2^{-j}k)$ so we obtain

$$a_k^j \approx f(k/2^j)\int_{-M}^{M} \overline{\phi(t)}dt = f(k/2^j)\int_{-\infty}^{\infty} \overline{\phi(t)}dt = f(k/2^j)m$$

□

6.2.2 Iteration

The second step is where we decompose f_i into a sum of lower-level approximation parts. We use a modified version of the deconstruction formula in theorem 12 to do this.
Definition 19. [16] We first define the down sample operator D on a vector $x = (...x_{-1}, x_0, x_1...)$ as $Dx = (...x_{-2}, x_0, x_2...)$ and we define the up sample operator U on x as $Ux = (...x_{-1}, 0, x_0, 0, x_1, 0...)$. We also define convolution on two sequences $x = (...x_{-1}, x_0, x_1...)$ and $y = (...y_{-1}, y_0, y_1...)$

as $(x * y)_l = \sum_{k\in\mathbb{Z}} x_k y_{l-k}$.

We next define two filters, the high pass filter h_k and the low pass filter l_k

$$h_k := \frac{1}{2}(-1)^k p_{k+1} \qquad l_k := \frac{1}{2}\overline{p_{-k}}$$

and using the above definitions we next define $H(x) = h * x$ and $L(x) = l * x$, then the deconstruction formulae in theorem 12 becomes

$$\begin{cases} a^{j-1} = DL(a^j) \\ b^{j-1} = DH(a^j) \end{cases}$$

where D is the down sample operator defined in definition 19. We may repeat this procedure on $a^{j-1}, a^{j-2}, a^{j-3}, ...$

6.2.3 Termination

The final step is when we decide to stop, which depends on the application. Most of the time we will carry on until we don't have enough coefficients left to perform another step but other applications such as removing noise from signals may stop before then.

6.3 Reconstruction Algorithm

This section is also based on [16]. For reconstruction we use modified versions of the reconstruction formula in theorem 12. Again we first define two filters, the high pass filter $\tilde{h_k}$ and the low pass filter $\tilde{l_k}$

$$\tilde{h_k} := \overline{p_{1-k}}(-1)^k \qquad \tilde{l_k} := p_k$$

We next define $\tilde{L}(x) = \tilde{l} * x$, and $\tilde{H}(x) = \tilde{h} * x$ then the reconstruction formula becomes

$$a^j = \tilde{L}(Ua^{j-1}) + \tilde{H}(Ub^{j-1})$$

where U is the up sample operator defined in definition 19. By applying this multiple times we can get back to our original data set.

6.4 Example of Simple Wavelet

The Haar Wavelet is the pair ϕ, ψ defined as

$$\psi(x) = \phi(2x) - \phi(2x-1) \qquad \phi(x) = \begin{cases} 1 & if\ 0 \le x \le 1 \\ 0 & elsewhere \end{cases}$$

where ψ is the mother wavelet and ϕ is the farther wavelet. We see that $p_0 = 1, p_1 = 1$ and one can check that these values of p satisfy theorem 9.

6.5 Daubechies Wavelets

6.5.1 Theory

We now have the tools to take a set of data and deconstruct and reconstruct it, but we still need a way to obtain the coefficients p_i. These could in theory be obtained directly from our mother and father wavelet but for almost all wavelets except the basic Haar wavelet this is far too difficult and impractical. A first course in wavelets with Fourier analysis [16] develops the following theorem continuing from the theory above, we simply state the result and the full derivation can be found in the book. Using this theorem allows us to construct the p_i easily.

Theorem 14. [16] *Suppose* $P(z) = \frac{1}{2}\sum_k p_k z^k$ *is a polynomial that satisfies the following conditions:*

$P(1) = 1$

$|P(z)|^2 + |P(-z)|^2 = 1$ *for* $|z| = 1$

$|P(e^{it})| > 0$ *for* $|t| \leq \pi/2$

Let $\phi_0(x)$ *be the Haar scaling function and let* $\phi_n = \sum_k p_k \phi_{n-1}(2x - k)$ *for* $n \geq 1$. *Then, the sequence* ϕ_n *converges pointwise in* L^2 *to a function* ϕ *which satisfies the normalization condition, orthonormality condition and the scaling equation.*

6.5.2 Construction

Using theorem 14 we can construct many of the Daubechies Wavelets. As an example we construct the 6-tap Daubechies wavelet by modifying the method for obtaining the 4-tap Daubechies wavelet in [16].

For a given polynomial $P(z)$, we let $p(\xi) = P(e^{-i\xi})$. We also let $1 = (cos^2(\xi/2) + sin^2(\xi/2))^5$, we raise it to the power 5 as we want the $5 + 1 = 6$ tap Daubechies wavelet. We then expand the brackets and use the identities $cos(u) = sin(u + \pi/2)$ and $sin(u) = -cos(u + \pi/2)$ as well the identity $|p(\xi)|^2 + |p(\xi + \pi)|^2 = 1$ from the hypothesis of theorem 14 and we get

$$|p(\xi)|^2 = cos^{10}(\xi/2) + 5cos^8(\xi/2)sin^2(\xi/2) + 10cos^6(\xi/2)sin^4(\xi/2)$$

In	211	211	213	213	214	215	216	215	213	216
LC	14.13		211.13		212.98		214.82		214.93	200.51
HC	52.75		0.50		-0.09		0.68		-1.62	-53.73
Out	211	211	213	213	214	215	216	215	213	216

Figure 7: Applying the Daubechies Wavelet. 'In' is the input data set, 'LC' are the coefficients obtained from the low filter, 'HC' is the coefficients obtained from high filter and 'Out' is the reconstruction

We let $A = \sqrt{10}$, $B = \sqrt{5 + 2\sqrt{10}}$ and rearrange to get

$$|p(\xi)|^2 = cos^6(\xi/2)|cos^2(\xi/2) - Asin^2(\xi/2) + iBcos(\xi/2)sin(\xi/2)|^2$$

$$p(\xi) = cos^3(\xi/2)(cos^2(\xi/2) - Asin^2(\xi/2) + iBcos(\xi/2)sin(\xi/2))\alpha(\xi)$$

where $|\alpha(\xi)| = 1$. We now let $p_0 = \frac{1+A+B}{16}$, $p_1 = \frac{5+2A+3B}{16}$, $p_2 = \frac{10+A+2B}{16}$, $p_3 = \frac{10+A-2B}{16}$, $p_4 = \frac{5+4A-3B}{16}$ and $p_5 = \frac{1+A-B}{16}$. Then by using $cos(\xi/2) = \frac{e^{i\xi/2}+e^{-i\xi/2}}{2}$ and $sin(\xi/2) = \frac{e^{i\xi/2}-e^{i\xi/2}}{2i}$ we get

$$p(\xi) \quad = \quad \frac{\alpha(\xi)}{2}(p_0e^{5i\xi/2} \ + \ p_1e^{3i\xi/2} \ + \ p_2e^{i\xi/2} \ + \ p_3e^{-i\xi/2} \ + \ p_4e^{-3i\xi/2} \ + \ p_5e^{-5i\xi/2})$$

We set $\alpha(\xi) = e^{-5i\xi/2}$ so that we get only powers of $e^{-i\xi}$ in $p(\xi)$. Hence we get $P(z) = \frac{1}{2}(p_0z + p_1z^2 + p_2z^3 + p_3z^4 + p_4z^5)$ which concurs with the results listed in [16] for the 6-tap Daubechies Wavelet and satisfies theorem 14.

6.5.3 Example

We now use listing 6 in appendix B. The listing is a Matlab program that puts the above theory to test. In this program we use the Daubechies 4 tap and the methods in sections 6.2 and 6.3 to get the results in figure 7. As you can see, the high coefficients with the exception of the edge coefficients are very small and could be encoded as zero with only minimal change in the original data hence causing compression. The boundary coefficients are very high due to this method extending the data by zeroing it outside our initial data set. This is very similar to what we saw in subsection 5.2.3 with the discontinuities at the boundaries. We could again apply this algorithm to the coefficients from the low filter and further zero out some coefficients in the high coefficients that are obtained. The problem with this wavelet though is that we end up with more coefficients from the filters than the number of original data points making it quite poor for compression. Orthogonal wavelets are not ideal for image compression but are useful for other applications such as signal analysis and noise reduction so we next discuss biorthogonal wavelets.

6.6 Biorthogonal Wavelets

The following paragraph is adapted from [3]. Orthogonal wavelets are not used for image compression in practice, this is due to the fact that the number of outputs is greater than the number of inputs so we have to encode more values if we compress the image, something that is extremely undesirable. We have a few options to fix this. Firstly we could make our N point data set periodic of period N, which would result in a periodic output of period N, hence we would only have to transmit N values. This is perfect but we get large compensation values at the boundaries which have to be encoded and will make for poor compression. Our next option is to symmetrically extend the N point data so we end up with a 2N periodic sequence, which results in a 2N periodic output, which is not desirable due to us having twice as many outputs as inputs but we get around this problem by using a symmetric filter, causing the output to be symmetric and we only have to encode N points. The problem is that the only symmetric filter that is orthogonal is the Haar transform which has poor compression so we must relax the condition of orthogonality and use biorthogonal wavelets.

Biorthogonal basically means that instead of our wavelet ϕ being orthogonal with its translates, it is orthogonal with another wavelet $\hat{\phi}$ and its translates. Much of the material mirrors that of orthogonal wavelets [15] so I will simply state the end result. Biorthogonal wavelets have two sets of filters the analysis filters h_i, l_i and the synthesis filters $\tilde{h}_i, \tilde{l}_i$. They are related to each other by the biorthogonality condition, $\tilde{h_k} = (-1)^k l_{1-k}$ and $h_k = (-1)^k \tilde{l}_{1-k}$. We end up with the following formula, taken from [14]

$$Decomposition : \begin{cases} a_l^{j-1} = \sum_{k\in\mathbb{Z}} l_{k-2l} a_k^j \\ b_l^{j-1} = \sum_{k\in\mathbb{Z}} h_{k-2l} a_k^j \end{cases}$$

$$Reconstruction : \ a_k^j = \sum_{l\in\mathbb{Z}} \tilde{l}_{l-2k} a_l^{j-1} + \sum_{l\in\mathbb{Z}} \tilde{h}_{l-2k} b_l^{j-1}$$

6.6.1 Example

We now use Matlab program listings 7, 8 and 9 in appendix B that I have created to compute the coefficients of the biorthogonal (9,7) wavelet. The results are shown in figure 8. See how we now have exactly the same number of coefficients as original data and that we don't have large boundary values in the high filter. These two points together make biorthogonal wavelets much more efficient and useful in image compression.

6.6.2 Why Wavelets Are Used In Data Compression

The reason for using wavelets in data compression is much the same as the reasons for using the DCT. Wavelets allow us to decompose a signal or set of data into subbands.

Input	1	2	3	4	5	6	7	8	9	10
Low Coefficients	1.33		3.07		5		6.95		9.06	
High Coefficients	0.2500		-4.149e-15		-6.4948e-15		-0.1825		0.8651	
Output	1	2	3	4	5	6	7	8	9	10

Figure 8: Computing the (9,7) wavelet transform of a data set. The low coefficients are also called the low subband, similarly high coefficients are called the high subband.

The low subband contains most of the content of the data and the high subband contains very small numbers that we can quite easily quantize to get compression, as we saw in example 6.6.1.

6.7 JPEG 2000

We now briefly describe the JPEG 2000 compression algorithm, the main implementation of wavelets into image compression. We will only briefly cover the coding area as the full details of the algorithm are long and beyond the scope of this paper.

6.7.1 Compressing An Image

Step 1 - We transform our image into Luminance and Chrominance representation, more commonly written YCrCb. This helps us get better compression as discussed in subsection 5.4.3**.**

Step 2 - Next we perform the DWT to our image. As described in example 6.5.3 we can keep applying the DWT to the results from the previous DWT to get multiple levels or subbands as they are more commonly called. The JPEG2000 standard allows for up to 32 subbands[4] but a JPEG2000 algorithm could use less. We perform this transform to each of the Y Cr and Cb components. JPEG 2000 has two different biorthogonal wavelets. The Le Gall (5,3) spline filter for lossless compression and (9,7) biorthogonal spline filter for lossy compression. Exact values for the (9,7) filter are used in the examples below and can be found in appendix A.2.

Step 3 - We now apply quantization, this provides some of the compression and we only do this step if we wish to get lossy compression. There are two methods that we can apply, the standard is to use a fixed quantization step. The other method uses the following formula

$$q_b(i,j) = sign(y_b(i,j)) \left\lfloor \frac{|y_b(i,j)|}{\Delta_b} \right\rfloor$$

where Δ_b is quantization value for subband b.

Step 4 - The last step in compressing the image is entropy encoding, this uses several methods that I won't discuss here such as fractional bit-plane coding, binary arithmetic

Figure 9: Left: JPEG image compressed to 11.6KB, Right: JPEG2000 image compressed to 11.2KB

coding and tag tree coding. Compression is also obtained here by throwing away the least important bit-planes.

6.7.2 Uncompressing An Image

As we have seen with the JPEG algorithm, uncompressing the image is simply a matter of reversing the steps for compression so I won't describe it in more detail.

6.7.3 Comparison of JPEG and JPEG 2000

We again use program listing 5 and its associated functions also listed in appendix B. The results of running our program on our test set of images [33] and [34] are in the table in figure 10. As you can see the size of both compressions is very similar for each image, yet JPEG2000 images are all of better quality. We can see this by the fact that the SSIM value is higher, EMS lower and PSNR higher for every JPEG2000 image. We therefore conclude that at similar sizes JPEG2000 produces just as good an image as JPEG does.

We also take a look at JPEG and JPEG2000 at very low sizes. Figure 9 shows an image compressed by JPEG to a size of around 11.6KB and JPEG2000 to a size of 11.2KB. While both images are of very poor quality, the JPEG image looks a lot worse with many block based artifacts (most noticeable on the window shutters), hence JPEG2000 wins against JPEG for high compression ratios. These results concur with [31] and [32].

7 Conclusion

Initially we compared the different types of data and discussed why a different model was required to get the best compression for each data type. We then briefly discussed lossless compression and went into some more detail about run length encoding and Huffman

Size		SSIM		EMS		PSNR	
JPEG	JP2000	JPEG	JP2000	JPEG	JP2000	JPEG	JP2000
656909	654368	0.9112	0.9584	26.8269	11.1888	3.7077	7.5056
531438	535536	0.8944	0.9558	12.7068	3.9182	6.9531	12.0625
393837	411392	0.9397	0.9748	8.6126	2.0214	8.6421	14.9369
491737	538568	0.9150	0.9638	11.8960	3.5749	7.2394	12.4608

Figure 10: Comparison of JPEG with JPEG2000 compression on a test set of images of uncompressed size 9437184 bits. Full results can be found in appendix A.3.

coding as they are used in JPEG compression and also in other image and text compression algorithms.

We then derived the Discrete Cosine Transformation from the Discrete Fourier Transform. We then used the algorithm on a large test set of images. The algorithm produced compressed images with more than 90% saving in space for every test image and the output was very close to the original according to the SSIM formula and observing the images by eye.

We next proved a lot of theory on multiresolution analysis and wavelets and then implemented this theory to get the wavelet transform, which is used in the newer JPEG2000 compression standard. We then went on to show how JPEG2000 produces a better quality image then JPEG when compressed to a similar size, and also showed how JPEG produces poor images when the target size of compressed image is very small.

For those who wish to carry on reading about image compression, a new standard called HEVC-MSP (High Efficiency Video Coding - Main Still Picture) which is also called H.265 would be a good place to start. It has been shown to be much more efficient than JPEG and JPEG2000 [22] and is hoped to be the successor of JPEG. A good introductory text is [23].

So over the course of this text you should now be familiar with how images are compressed in most applications where images are used today and you should now have an understanding of the mathematics that underpins these compression algorithms. You should also have a brief knowledge of what factors compression algorithms exploit to reduce size of data for the other data types such as text, audio and video. Finally you should know why we need compression and how important it is as well as seeing how effective it can be.

A First Appendix

A.1 Colour Transforms

The following two colour transformations are taken from [4]. We convert between RGB and YCrCb by using the following matrices (Called ICT, not reversible, used for lossy coding):

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299000 & 0.587000 & 0.114000 \\ -0.168736 & -0.331264 & 0.500000 \\ 0.500000 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.0 & 0.0 & 1.402000 \\ 1.0 & -0.344136 & -0.714136 \\ 1.0 & 1.772000 & 0.0 \end{bmatrix} \begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix}$$

We convert between RGB and YUV using the following matrices (Called RCT, reversible, used for lossless coding):

$$Y_r = \left\lfloor \frac{R + 2G + B}{4} \right\rfloor, \quad U_r = B - G, \quad V_r = R - G$$

$$G = Y_r - \left\lfloor \frac{U_r + V_r}{4} \right\rfloor, \quad R = V_r + G, \quad B = U_r + G$$

A.2 Biorthogonal Wavelet Filter Coefficients

The following is taken from [4]. First we state the coefficients for the (9,7) biorthogonal filter. This filter can only be used in lossy compression.
9-tap low-pass analysis filter:

$h_4 = h_{-4} = 0.026748757410810$

$h_3 = h_{-3} = -0.016864118442875$

$h_2 = h_{-2} = -0.078223266528988$

$h_1 = h_{-1} = 0.266864118442872$

$h_0 = 0.602949018236358$
7-tap high-pass analysis filter:

$g_3 = g_{-3} = 0.0912717631142495$

$g_2 = g_{-2} = -0.057543526228500$

$g_1 = g_{-1} = -0.591271763114247$

$g_0 = 1.115087052456994$

Now we state the coefficients for the (5,3) biorthogonal filter. This filter can be used in lossless compression.

5-tap low-pass analysis filter:

$h_2 = h_{-2} = -1/8$

$h_1 = h_{-1} = 1/4$

$h_0 = 3/4$

3-tap high-pass analysis filter:

$g_1 = g_{-1} = -1/2$

$g_0 = 1$

A.3 Full Results of Examples

Here is the full output of the test run on our sample images using the Matlab program listing 5 in appendix B. The shortened versions are contained in the main text in figure 5 and figure 10.

Image #	Uncompressed_Size	Size		Savings	
		JPEG	JP2000	JPEG	JP2000
1	9437184	656909	654368	0.9304	0.9307
2	9437184	531438	535536	0.9437	0.9433
3	9437184	393837	411392	0.9583	0.9564
4	9437184	491737	538568	0.9479	0.9429
5	9437184	871970	694440	0.9076	0.9264
6	9437184	583668	574648	0.9382	0.9391
7	9437184	540606	449984	0.9427	0.9523
8	9437184	830625	741208	0.9120	0.9215
9	9437184	374882	496064	0.9603	0.9474
10	9437184	449606	516456	0.9524	0.9453
11	9437184	574013	558048	0.9392	0.9409
12	9437184	379226	460128	0.9598	0.9512
13	9437184	907092	806728	0.9039	0.9145
14	9437184	701971	645896	0.9256	0.9316
15	9437184	453126	491280	0.9520	0.9479
16	9437184	451900	480656	0.9521	0.9491
17	9437184	426411	517968	0.9548	0.9451
18	9437184	741370	729376	0.9214	0.9227
19	9437184	509885	586832	0.9460	0.9378
20	9437184	398313	463000	0.9578	0.9509
21	9437184	557263	584440	0.9410	0.9381
22	9437184	659933	623856	0.9301	0.9339
23	9437184	480989	436888	0.9490	0.9537
24	9437184	681239	646032	0.9278	0.9315

Image #	SSIM		EMS		PSNR	
	JPEG	JP2000	JPEG	JP2000	JPEG	JP2000
1	0.9112	0.9584	26.8269	11.1888	3.7077	7.5056
2	0.8944	0.9558	12.7068	3.9182	6.9531	12.0625
3	0.9397	0.9748	8.6126	2.0214	8.6421	14.9369
4	0.9150	0.9638	11.8961	3.5749	7.2394	12.4608
5	0.9400	0.9707	25.1701	10.3112	3.9846	7.8603
6	0.9197	0.9652	19.7752	6.1459	5.0322	10.1076
7	0.9585	0.9804	9.4866	2.6581	8.2223	13.7476
8	0.9300	0.9629	27.4045	9.9160	3.6152	8.0301
9	0.9319	0.9621	9.3388	2.2527	8.2905	14.4663
10	0.9296	0.9663	9.9535	2.5466	8.0136	13.9338
11	0.9128	0.9601	16.7082	5.3804	5.7641	10.6853
12	0.9206	0.9662	9.3725	2.5522	8.2749	13.9243
13	0.8971	0.9525	37.5424	16.7739	2.2482	5.7471
14	0.9143	0.9573	20.3998	7.8379	4.8971	9.0514
15	0.9226	0.9640	11.9049	3.5784	7.2362	12.4565
16	0.9217	0.9702	12.8715	3.5366	6.8971	12.5076
17	0.9355	0.9717	11.0046	2.9743	7.5777	13.2595
18	0.9138	0.9581	21.4458	7.0329	4.6800	9.5221
19	0.9169	0.9655	15.8536	4.1653	5.9921	11.7969
20	0.9443	0.9821	11.3309	2.5897	7.4508	13.8610
21	0.9305	0.9622	17.6623	4.6401	5.5229	11.3281
22	0.9101	0.9585	16.7174	5.5728	5.7617	10.5326
23	0.9489	0.9742	7.1229	1.7214	9.4668	15.6346
24	0.9259	0.9615	21.1956	8.3674	4.7309	8.7675

B Second Appendix

All of the following program listing can be found at http://thomaspreece.com/thesis/matlab/

Listing 1: FFT vs DCT matlab code

```
clear;

for p=1:10
    %Randomly generate 1000 sets of 8 values
    X=rand(1000,8);
    %Set N as the size of 1 set, so will hold value 8.
    N=size(X,2);
    M=size(X,1);
```

```
9
10      %Loop through our 1000 sets and extend them to 4N (4*8) size so we ...
            can use
11      %them with the FFT
12      for k=1:M
13          for i=1:N
14              X2(k,2*i-1)=0;
15              X2(k,2*i)=X(k,i);
16          end
17          X2(k,N+1)=0;
18          for i=2:N*2
19              X2(k,4*N-i+2)=X2(k,i);
20          end
21      end
22
23      %-------------------------------- Time FFT Method ...
            --------------------------------
24
25      %Start Timer
26      ticFFT=tic;
27      for k=1:M
28          %Perform the fast Fourier transform to each set of data
29          Y2=real(fft(X2(k,:))*(1/sqrt(4*N)));
30          %Remove the symmetric repeating entries
31          Y2Cut(k,:)=Y2(:,1:N);
32      end
33      %Save time taken to calculate
34      FFT(p)=toc(ticFFT);
35
36      %-------------------------------- Time DCT Method ...
            --------------------------------
37
38      %Start Timer
39      ticDCT=tic;
40      for k=1:M
41          %Calculate the DCT for each of the N points
42          for i=0:N-1
43              sum=0;
44              for j=0:N-1
45                  sum=sum+X(k,j+1)*cos((pi*(j+0.5)*(i))/N);
46              end
47              Y3(k,i+1)=(1/sqrt(N))*sum;
48          end
49      end
50      %Save time taken to calculate
51      DCT(p)=toc(ticDCT);
52  end
53  Difference=DCT-FFT;
54  PercentageDifference=Difference./DCT;
```

```
PercentageDifference=PercentageDifference.*100;

T=table(transpose(DCT), transpose(FFT), transpose(Difference), ...
    transpose(PercentageDifference), 'VariableNames', {'DCT', 'FFT', ...
    'FFT_Speed_Improvement', 'FFT_Percentage_Speed_Improvement'});

writetable(T,'FFTvsDCT.csv')
```

Listing 2: An matlab function to calculate results of JPEG compression algorithm

```
function [ Output,TotalBits,UncompressedBits ] = JPEG( ...
    Input,BitsPerPixel )
%JPEG Applies the JPEG method to Input image.
%   Applies the JPEG method to compress an image and works out the required
%   storage size as well as returning the image that would be obtained by
%   uncompressing the image. It does not however calculate the actual bit
%   sequence (the compressed image) that the JPEG method would return,
%   just the size of that bit sequence.

%Luminance and Chrominance values obtained from Annex K of JPEG ...
    specification, tables K.1,K.2,K.5 K.6
%Quantization Values
LuminanceQ=[16,11,10,16,24,40,51,61; 12,12,14,19,26,58,60,55; ...
    14,13,16,24,40,57,69,56; 14,17,22,29,51,87,80,62; ...
    18,22,37,56,68,109,103,77; 24,35,55,64,81,104,113,92; ...
    49,64,78,87,103,121,120,101; 72,92,95,98,112,100,103,99];
ChrominanceQ=[17,18,24,47,99,99,99,99; 18,21,26,66,99,99,99,99; ...
    24,26,56,99,99,99,99,99; 47,66,99,99,99,99,99,99; ...
    99,99,99,99,99,99,99,99; 99,99,99,99,99,99,99,99; ...
    99,99,99,99,99,99,99,99; 99,99,99,99,99,99,99,99];

%Bits required for each category of DC difference
LuminanceDC=[2,3,3,3,3,3,4,5,6,7,8,9];
ChrominanceDC=[2,2,2,3,4,5,6,7,8,9,10,11];

%Bits required for each Run Length Huffman encoding of Luminance AC
%coefficients
LuminanceAC(1,:)=[4,2,2,3,4,5,7,8,10,16,16];
LuminanceAC(2,:)=[0,4,5,7,9,11,16,16,16,16,16];
LuminanceAC(3,:)=[0,5,8,10,12,16,16,16,16,16,16];
LuminanceAC(4,:)=[0,6,9,12,16,16,16,16,16,16,16];
LuminanceAC(5,:)=[0,6,10,16,16,16,16,16,16,16,16];
LuminanceAC(6,:)=[0,7,11,16,16,16,16,16,16,16,16];
LuminanceAC(7,:)=[0,7,12,16,16,16,16,16,16,16,16];
LuminanceAC(8,:)=[0,8,12,16,16,16,16,16,16,16,16];
LuminanceAC(9,:)=[0,9,15,16,16,16,16,16,16,16,16];
LuminanceAC(10,:)=[0,9,16,16,16,16,16,16,16,16,16];
LuminanceAC(11,:)=[0,9,16,16,16,16,16,16,16,16,16];
```

```
LuminanceAC(12,:)=[0,10,16,16,16,16,16,16,16,16,16];
LuminanceAC(13,:)=[0,10,16,16,16,16,16,16,16,16,16];
LuminanceAC(14,:)=[0,11,16,16,16,16,16,16,16,16,16];
LuminanceAC(15,:)=[0,16,16,16,16,16,16,16,16,16,16];
LuminanceAC(16,:)=[11,16,16,16,16,16,16,16,16,16,16];

%Bits required for each Run Length Huffman encoding of Chrominance AC
%coefficients
ChrominanceAC(1,:)=[2,2,3,4,5,5,6,7,9,10,12];
ChrominanceAC(2,:)=[0,4,6,8,9,11,12,16,16,16,16];
ChrominanceAC(3,:)=[0,5,8,10,12,15,16,16,16,16,16];
ChrominanceAC(4,:)=[0,5,8,10,12,16,16,16,16,16,16];
ChrominanceAC(5,:)=[0,6,9,16,16,16,16,16,16,16,16];
ChrominanceAC(6,:)=[0,6,10,16,16,16,16,16,16,16,16];
ChrominanceAC(7,:)=[0,7,11,16,16,16,16,16,16,16,16];
ChrominanceAC(8,:)=[0,7,11,16,16,16,16,16,16,16,16];
ChrominanceAC(9,:)=[0,8,16,16,16,16,16,16,16,16,16];
ChrominanceAC(10,:)=[0,9,16,16,16,16,16,16,16,16,16];
ChrominanceAC(11,:)=[0,9,16,16,16,16,16,16,16,16,16];
ChrominanceAC(12,:)=[0,9,16,16,16,16,16,16,16,16,16];
ChrominanceAC(13,:)=[0,9,16,16,16,16,16,16,16,16,16];
ChrominanceAC(14,:)=[0,11,16,16,16,16,16,16,16,16,16];
ChrominanceAC(15,:)=[0,14,16,16,16,16,16,16,16,16,16];
ChrominanceAC(16,:)=[10,15,16,16,16,16,16,16,16,16,16];

Input=double(Input);

%Dimensions of Image must be exactly divisible by 8
if mod(size(Input,1),8) ~= 0 || mod(size(Input,2),8) ~= 0
    errordlg('Dimensions of image not divisible by 8')
    return
end

%Preallocate arrays for speed
Y=double(zeros(size(Input,1),size(Input,2)));
Cb=double(zeros(size(Input,1),size(Input,2)));
Cr=double(zeros(size(Input,1),size(Input,2)));

%Set the bit counters to zero
YBits=0;
CrBits=0;
CbBits=0;

%Convert Image from RBG to YCrCb
for i=1:size(Input,1)
    for j=1:size(Input,2)
        Y(i,j)=0.299000*Input(i,j,1) + 0.587000*Input(i,j,2) + ...
            0.114000*Input(i,j,3);
```

```
78          Cb(i,j)=(-0.168736*Input(i,j,1)) - (0.331264*Input(i,j,2)) + ...
                (0.500000*Input(i,j,3));
79          Cr(i,j)=(0.500000*Input(i,j,1)) - (0.418688*Input(i,j,2)) - ...
                (0.081312*Input(i,j,3));
80      end
81  end
82
83  %Level shift Y by 128 and Cr Cb by 64 (assuming that input image is 8-bit)
84  Y=Y-(2^BitsPerPixel)/2;
85  Cr=Cr-(2^BitsPerPixel)/4;
86  Cb=Cb-(2^BitsPerPixel)/4;
87
88  OldY=0;
89  OldCr=0;
90  OldCb=0;
91
92  %Loop through each 8x8 block and apply DCT, quantization, calculate bits
93  %required to store using JPEG method described earlier in report, reverse
94  %quantization and reverse DCT
95  for i = 1:(size(Input,2)/8)
96      for j = 1:(size(Input,1)/8)
97          %Get 8x8 subblock
98          YSub = Y(8*(j-1)+1:8*j,8*(i-1)+1:8*i);
99          CrSub = Cr(8*(j-1)+1:8*j,8*(i-1)+1:8*i);
100         CbSub = Cb(8*(j-1)+1:8*j,8*(i-1)+1:8*i);
101
102         %Use matlabs builtin command dct2 to perform the DCT faster
103         DCTY = dct2(YSub);
104         DCTCr = dct2(CrSub);
105         DCTCb = dct2(CbSub);
106
107         %preallocate arrays for speed
108         DCTQY = zeros(8,8);
109         DCTQCr = zeros(8,8);
110         DCTQCb = zeros(8,8);
111         DCTUQY = zeros(8,8);
112         DCTUQCr = zeros(8,8);
113         DCTUQCb = zeros(8,8);
114
115         %Apply quantization
116         for m=1:8
117             for n=1:8
118                 DCTQY(m,n)=round(DCTY(m,n)/LuminanceQ(m,n));
119                 DCTQCr(m,n)=round(DCTCr(m,n)/ChrominanceQ(m,n));
120                 DCTQCb(m,n)=round(DCTCb(m,n)/ChrominanceQ(m,n));
121             end
122         end
123
124         %Calculate difference between this luminance coefficient and last
```

```
125         Diff=DCTQY(1,1)-OldY;
126         %Calculate the DC coefficient category
127         if Diff==0
128             Category=0;
129         else
130             Category=floor(log2(abs(Diff)))+1;
131         end
132         %Add the required bits for that category to the total for Y
133         YBits=YBits+LuminanceDC(Category+1)+Category;
134         %Set last coefficient to be the one used in this loop
135         OldY=DCTQY(1,1);
136 
137         %Calculate difference between this Chrominance coefficient and last
138         Diff=DCTQCr(1,1)-OldCr;
139         %Calculate the DC coefficient category
140         if Diff==0
141             Category=0;
142         else
143             Category=floor(log2(abs(Diff)))+1;
144         end
145         %Add the required bits for that category to the total for Cr
146         CrBits=CrBits+ChrominanceDC(Category+1)+Category;
147         %Set last coefficient to be the one used in this loop
148         OldCr=DCTQCr(1,1);
149 
150         %Calculate difference between this Chrominance coefficient and last
151         Diff=DCTQCb(1,1)-OldCb;
152         %Calculate the DC coefficient category
153         if Diff==0
154             Category=0;
155         else
156             Category=floor(log2(abs(Diff)))+1;
157         end
158         %Add the required bits for that category to the total for Cr
159         CbBits=CbBits+ChrominanceDC(Category+1)+Category;
160         %Set last coefficient to be the one used in this loop
161         OldCb=DCTQCb(1,1);
162 
163 
164 %http://www.mathworks.com/matlabcentral/fileexchange/15317-zigzag-scan
165         %is used to calculate the zigzag scan of AC coefficients
166         Yzigzag=zigzag(DCTQY);
167         Crzigzag=zigzag(DCTQCr);
168         Cbzigzag=zigzag(DCTQCb);
169 
170         zeroCount=0;
171         for m=2:64
172             if Yzigzag(m)==0
173                 %Element is zero so add 1 onto the zero count
```

```
174                 zeroCount=zeroCount+1;
175             else
176                 if zeroCount>15
177                     %Send the marker for 15 zeros (ZRL) zeroCount/15 times
178                     YBits=YBits+floor(zeroCount/15)*LuminanceAC(16,1);
179                     zeroCount=zeroCount-floor(zeroCount/15)*15;
180                 end
181                 %Calculate the AC coefficient category
182                 Category = floor(log2(abs(Yzigzag(m))))+1;
183                 %Add the required bits for that category and zero count to
184                 %the total for Y
185                 YBits=YBits+LuminanceAC(zeroCount+1,Category+1)+Category;
186                 zeroCount=0;
187             end
188         end
189         if zeroCount>0
190             %Use End of Block (EoB) code and add bits required for code to
191             %total for Y
192             YBits=YBits+LuminanceAC(1,1);
193         end
194
195         for m=2:64
196             if Crzigzag(m)==0
197                 %Element is zero so add 1 onto the zero count
198                 zeroCount=zeroCount+1;
199             else
200                 if zeroCount>15
201                     %Send the marker for 15 zeros (ZRL) zeroCount/15 times
202                     CrBits=CrBits+floor(zeroCount/15)*ChrominanceAC(16,1);
203                     zeroCount=zeroCount-floor(zeroCount/15)*15;
204                 end
205                 %Calculate the AC coefficient category
206                 Category = floor(log2(abs(Crzigzag(m))))+1;
207                 %Add the required bits for that category and zero count to
208                 %the total for Cr
209                 CrBits=CrBits+ChrominanceAC(zeroCount+1,Category+1)+Category;
210                 zeroCount=0;
211             end
212         end
213         if zeroCount>0
214             %Use End of Block (EoB) code and add bits required for code to
215             %total for Cr
216             CrBits=CrBits+ChrominanceAC(1,1);
217         end
218
219
220         for m=2:64
221             if Cbzigzag(m)==0
222                 %Element is zero so add 1 onto the zero count
```

```
223                 zeroCount=zeroCount+1;
224             else
225                 if zeroCount>15
226                     %Send the marker for 15 zeros (ZRL) zeroCount/15 times
227                     CbBits=CbBits+floor(zeroCount/15)*ChrominanceAC(16,1);
228                     zeroCount=zeroCount-floor(zeroCount/15)*15;
229                 end
230                 %Calculate the AC coefficient category
231                 Category = floor(log2(abs(Cbzigzag(m))))+1;
232                 %Add the required bits for that category and zero count to
233                 %the total for Cb
234                 CbBits=CbBits+ChrominanceAC(zeroCount+1,Category+1)+Category;
235                 zeroCount=0;
236             end
237         end
238         if zeroCount>0
239             %Use End of Block (EoB) code and add bits required for code to
240             %total for Cb
241             CbBits=CbBits+ChrominanceAC(1,1);
242         end
243
244
245         %Reverse Quantization
246         for m=1:8
247             for n=1:8
248                 DCTUQY(m,n)=DCTQY(m,n)*LuminanceQ(m,n);
249                 DCTUQCr(m,n)=DCTQCr(m,n)*ChrominanceQ(m,n);
250                 DCTUQCb(m,n)=DCTQCb(m,n)*ChrominanceQ(m,n);
251             end
252         end
253
254         %Reverse DCT
255         IDCTY=idct2(DCTUQY);
256         IDCTCr=idct2(DCTUQCr);
257         IDCTCb=idct2(DCTUQCb);
258
259         %Put Output of 8x8 block into correct place in full image
260         OutputY(8*(j-1)+1:8*j,8*(i-1)+1:8*i) = IDCTY;
261         OutputCr(8*(j-1)+1:8*j,8*(i-1)+1:8*i) = IDCTCr;
262         OutputCb(8*(j-1)+1:8*j,8*(i-1)+1:8*i) = IDCTCb;
263     end
264 end
265
266 %Reverse Level shift
267 OutputY=OutputY+(2^BitsPerPixel)/2;
268 OutputCr=OutputCr+(2^BitsPerPixel)/4;
269 OutputCb=OutputCb+(2^BitsPerPixel)/4;
270
271 %Preallocate array for speed
```

```
272 Output=double(zeros(size(Input,1),size(Input,2),size(Input,3)));
273
274 %Reverse colour transformation, takes YCrCb to RGB
275 for i=1:size(Input,1)
276     for j=1:size(Input,2)
277         Output(i,j,1)=1*OutputY(i,j) + 1.402000*OutputCr(i,j);
278         Output(i,j,2)=1*OutputY(i,j) - 0.344136*OutputCb(i,j) - ...
                0.714136*OutputCr(i,j);
279         Output(i,j,3)=1*OutputY(i,j) + 1.772000*OutputCb(i,j);
280     end
281 end
282
283 %Correct data type
284 Output=uint8(Output);
285
286 %Calculate total bits required to store compressed image and uncompressed
287 %image
288 TotalBits=YBits+CrBits+CbBits;
289 UncompressedBits=8*size(Input,1)*size(Input,2)*size(Input,3);
290
291 end
```

Listing 3: Calculates the SSIM of two images

```
1  function [ Out ] = AverageSSIM(Input,Output,BitsPerPixel)
2  %AverageSSIM Calculates the SSIM over 8x8 subblocks and averages the values
3  %   Calculates the SSIM over 8x8 blocks with overlap of block column with
4  %   the next being 50% and similarly for block rows.
5  %   Input, Output must be X by Y by 3 arrays of RGB colours. BitsPerPixel
6  %   contains the number of bits used per pixel in Input and Output.
7
8      %Preallocate Luminance arrays IY,OY
9      IY=zeros(size(Input,1),size(Input,2));
10     OY=zeros(size(Output,1),size(Output,2));
11
12     %Use conversion formula to calculate the Luminance of each pixel in ...
           Input
13     for i=1:size(Input,1)
14         for j=1:size(Input,2)
15             IY(i,j)=0.299000*Input(i,j,1) + 0.587000*Input(i,j,2) + ...
                   0.114000*Input(i,j,3);
16         end
17     end
18
19     %Use conversion formula to calculate the Luminance of each pixel in ...
           Output
20     for i=1:size(Output,1)
21         for j=1:size(Output,2)
```

```
22              OY(i,j)=0.299000*Output(i,j,1) + 0.587000*Output(i,j,2) + ...
                    0.114000*Output(i,j,3);
23          end
24      end
25
26      %Preallocate array to store SSIM values
27      S=zeros((floor(size(Input,1)/4)-1),(floor(size(Input,2)/4)-1));
28
29      %Runs through all the blocks and uses SSIM function to calculate SSIM
30      for i=1:(floor(size(Input,1)/4)-1)
31          for j=1:(floor(size(Input,2)/4)-1)
32              A = IY((i-1)*4+1:(i-1)*4+8 , (j-1)*4+1:(j-1)*4+8);
33              A = double(transpose(A(:)));
34              B = OY((i-1)*4+1:(i-1)*4+8 , (j-1)*4+1:(j-1)*4+8);
35              B = double(transpose(B(:)));
36              S(i,j)=SSIM(A,B,BitsPerPixel);
37          end
38      end
39
40      %Find the average of all 8x8 blocks calculated
41      Out = mean(S(:));
42
43  end
44
45  function [ Out ] = SSIM( A,B,BitsPerPixel )
46  %SSIM Calculates SSIM on A and B
47  %   Uses the SSIM formula to calculate the SSIM of A and B
48
49      if size(A,1)~=1 || size(A,1)~=size(B,1) || size(A,2)~=size(B,2)
50          errordlg('Non Matching Image')
51          return
52      end
53
54      %Average of A
55      UA=mean(A(:));
56      %Average of B
57      UB=mean(B(:));
58
59      %Array containing CoVariance and Variance
60      CoVar = cov(A,B);
61
62      %Constants used in formula, using k1=0.01, k2=0.03
63      c1=(0.01*((2^BitsPerPixel)-1))^2;
64      c2=(0.03*((2^BitsPerPixel)-1))^2;
65
66      %Return the SSIM
67      Out = ( (2*UA*UB+c1)*(2*CoVar(1,2)+c2) )/( ...
            (UA^2+UB^2+c1)*(CoVar(1,1)+CoVar(2,2)+c2) );
68
```

```
69 end
```

Listing 4: Calculates the MSE, PSNR

```
1  function [ EMS,PSNR ] = MSE( Input,Output,BitsPerPixel )
2  %MSE Returns the MSE and PSNR of two images
3  %   Returns the mean square error EMS and the Peak Signal to Noise PSNR of
4  %   two images Input and Output with BitsPerPixel number of bits per pixel
5
6  %Throw error if image sizes are incorrect
7  if size(Input,1)~=size(Output,1) || size(Input,2)~=size(Output,2) || ...
       size(Input,3)~=size(Output,3)
8      errordlg('Non Matching Image')
9      return
10 end
11
12 %Get image sizes and store them for later use
13 N=size(Input,1);
14 M=size(Input,2);
15 Z=size(Input,3);
16
17 sumElts=0; %removed double(0)
18
19 %Loop through each pixel in the Red,Green and blue channels and add the
20 %error to the sum
21 for z=1:Z
22     for n=1:N
23        for m=1:M
24            sumElts = sumElts + double((Input(n,m,z)-Output(n,m,z))^2);
25        end
26     end
27 end
28
29 %Calculate the Mean square error and the peak signal to noise
30 EMS=double(sumElts)/(M*N*Z);
31 PSNR = 10*log10(((BitsPerPixel^2)-1)/EMS);
32
33 end
```

Listing 5: Produces figures on JPEG and JPEG 2000 compression on a test set of images

```
1  clear;
2  %Image set is obtained from http://r0k.us/graphics/kodak/. A mirror of the
3  %images can be obtained from http://thomaspreece.com/thesis/.
4
5  %This file compares JPEG compression and JPEG2000 compression. Due to the
6  %complexity of implementing a decent JPEG2000 compression algorithm in
```

```
%matlab I have omitted it and used a separate program to convert our
%original PNG files into JPEG2000 files and then converted those back to
%PNG format. Due to PNG being lossless the input of our modified PNG files
%is exactly the same as if we had imported the JPEG2000 images directly
%allowing us to compare them as if we were reading the JPEG2000 image
%files.

%Preallocate arrays for speed
UncompressedBits=zeros(1,24);
Saving_JPEG=zeros(1,24);
SSIM_JPEG=zeros(1,24);
EMS_JPEG=zeros(1,24);
PSNR_JPEG=zeros(1,24);
TotalBits_JPEG=zeros(1,24);

Saving_JP2=zeros(1,24);
SSIM_JP2=zeros(1,24);
EMS_JP2=zeros(1,24);
PSNR_JP2=zeros(1,24);
TotalBits_JP2=zeros(1,24);

%Loop through our 24 image set
for i=1:24
    if i<10
        FileNumber=strcat('0',int2str(i))
    else
        FileNumber=int2str(i)
    end

    %Load image
    Input = imread( strcat( ...
        'C:\Users\Tom\Dropbox\Matlab\PictureSet\kodim', FileNumber, ...
        '.png' ) );
    Input=double(Input);
    BitsPerPixel=8;

    %Use the JPEG algorithm
    [Output,TotalBits_JPEG(i),UncompressedBits(i)]=JPEG(Input,BitsPerPixel);

    %Calculate saving gained from compression.
    Saving_JPEG(i)=(UncompressedBits(i) - TotalBits_JPEG(i)) / ...
        UncompressedBits(i);
    SSIM_JPEG(i)=AverageSSIM(Input,Output,BitsPerPixel);
    [EMS_JPEG(i),PSNR_JPEG(i)]=MSE(Input,Output,BitsPerPixel);

    JP2Output = imread( strcat ( ...
        'C:\Users\Tom\Dropbox\Matlab\PictureSet\kodim', FileNumber, ...
        '_Modified.png' ) );

```

```
51      s = dir( strcat( 'C:\Users\Tom\Dropbox\Matlab\PictureSet\kodim', ...
            FileNumber, '.jp2' ) );
52      TotalBits_JP2(i)=s.bytes*8;
53
54      Saving_JP2(i)=(UncompressedBits(i)-TotalBits_JP2(i))/UncompressedBits(i);
55      SSIM_JP2(i)=AverageSSIM(Input,JP2Output,BitsPerPixel);
56      [EMS_JP2(i),PSNR_JP2(i)]=MSE(Input,JP2Output,BitsPerPixel);
57
58  end
59
60  T=table(transpose(UncompressedBits), transpose(TotalBits_JPEG), ...
        transpose(TotalBits_JP2), transpose(Saving_JPEG), ...
        transpose(Saving_JP2), 'VariableNames', {'Uncompressed_Size', ...
        'JPEG_Size', 'JP200_Size', 'JPEG_Savings', 'JP2000_Savings'});
61  writetable(T,'Sizes.csv')
62
63  T2=table(transpose(UncompressedBits), transpose(Saving_JPEG), ...
        transpose(Saving_JP2), transpose(SSIM_JPEG), transpose(SSIM_JP2), ...
        transpose(EMS_JPEG), transpose(EMS_JP2), transpose(PSNR_JPEG), ...
        transpose(PSNR_JP2), 'VariableNames', {'Uncompressed_Size', ...
        'JPEG_Savings', 'JP2000_Savings', 'JPEG_SSIM', 'JP2000_SSIM', ...
        'JPEG_EMS', 'JP2000_EMS', 'JPEG_PSNR', 'JP2000_PSNR'});
64  writetable(T2,'Quality.csv')
65
66  return
```

Listing 6: Using Daubechies Wavelets to decompose and reconstruct a set of data

```
1   clear;
2
3   %Set the Polynomial Coordinates
4   global P;
5   %Daubechies 6-tap
6   %P=[0.47046721,1.14111692,0.650365,-0.19093442,-0.12083221,0.0498175];
7
8   %Daubechies 4-tap
9   P=[0.6830127,1.1830127,0.3169873,-0.1830127]; %Polynomial Values
10
11  %Daubechies 2-tap/Haar Wavelet
12  %P=[1,1];
13
14  %Input our data
15  X=[211,211,213,213,214,215,216,215,213,216];
16
17  %preallocate arrays for speed
18  h=zeros(1,size(P,2));
19  l=zeros(1,size(P,2));
20
```

```
21 q=size(X,2);
22 n=size(P,2);
23
24 %Calculate deconstruction filters from polynomial coefficients
25 for k=1:n
26     h(k)=0.5*P(k)*(-1)^k;
27 end
28 for k=1:n
29     l(k)=0.5*P(n-k+1);
30 end
31
32 %Apply the convolution method of deconstruction
33 w=conv(l,X);
34 w2=conv(h,X);
35
36 %Down sample our coefficients
37 Lcoeff2=w(:,2:2:end);
38 Hcoeff2=w2(:,2:2:end);
39
40 Hcoeff2(2)=0;
41 Hcoeff2(3)=0;
42 Hcoeff2(4)=0;
43 Hcoeff2(5)=0;
44
45 %Up sample our coefficients
46 for k=1:size(Lcoeff2,2)
47     ULcoeff(2*k-1)=Lcoeff2(k);
48     ULcoeff(2*k)=0;
49 end
50
51 for k=1:size(Hcoeff2,2)
52     UHcoeff(2*k-1)=Hcoeff2(k);
53     UHcoeff(2*k)=0;
54 end
55
56 %Construct reconstruction filters
57 for k=1:n
58     hT(k)=P(n-k+1)*((-1)^(n-k+1));
59 end
60 for k=1:n
61     lT(k)=P(k);
62 end
63
64 %Apply reconstruction method using convolution
65 X2=conv(lT,ULcoeff)+conv(hT,UHcoeff);
66
67 %Strip out the unnecessary data to get original data
68 X2=X2(n-2+1:n-2+q);
69
```

```
70 return
```

Listing 7: Computes the (9,7) Wavelet transform of a data set

```
1 %Simple script to show off wavelet transform
2 X=[1,2,3,4,5,6,7,8,9,10];
3
4 %Compute the Wavelet transform on X
5 [w,w2]=Filter97(X);
6
7 %Reverse wavelet transform to get back original values
8 X2=InverseFilter97(w,w2);
```

Listing 8: Computes the (9,7) Wavelet transform

```
1 function [ Z,Z2 ] = Filter97( X )
2 %Filter97 Applies the CDF9/7 wavelet transformation
3 %   Applies the wavelet transformation using the the 9/7
4 %   Cohen-Daubechies-Feauveau biorthogonal wavelet. The algorithm is very
5 %   inefficient but shows the main principles discussed in the paper.
6
7 %Low pass analysis filter
8 h(1) = 0.026748757411;
9 h(2) = -0.016864118443;
10 h(3) = -0.078223266529;
11 h(4) = 0.266864118443;
12 h(5) = 0.602949018236;
13 h(6) = h(4);
14 h(7) = h(3);
15 h(8) = h(2);
16 h(9) = h(1);
17
18 %high pass analysis filter
19 g(1) = 0.0912717631142495;
20 g(2) = -0.057543526228500;
21 g(3) = -0.591271763114247;
22 g(4) = 1.115087052456994;
23 g(5) = g(3);
24 g(6) = g(2);
25 g(7) = g(1);
26
27 %lowpass synthesis filter
28 h2(1)=-g(1);
29 h2(2)=g(2);
30 h2(3)=-g(3);
31 h2(4)=g(4);
32 h2(5)=-g(3);
```

```
33 h2(6)=g(2);
34 h2(7)=-g(1);
35
36 %highpass synthesis filter
37 g2(1)=h(1);
38 g2(2)=-h(2);
39 g2(3)=h(3);
40 g2(4)=-h(4);
41 g2(5)=h(5);
42 g2(6)=-h(4);
43 g2(7)=h(3);
44 g2(8)=-h(2);
45 g2(9)=h(1);
46
47
48 %Deconstruction using the lowpass analysis filter
49 for l=0:size(X,2)-1
50     sum=double(0);
51     for k=-4:4
52        sum=sum+h(k+5)*double(SymX(X,l-k));
53     end
54     w(l+1)=double(sum);
55 end
56
57 %Deconstruction using the highpass analysis filter
58 for l=0:size(X,2)-1
59     sum=double(0);
60     for k=-3:3
61        sum=sum+g(k+4)*double(SymX(X,l-k));
62     end
63     w2(l+1)=double(sum);
64 end
65
66 %Down sample signal
67 Z=w(1:2:end);
68 Z2=w2(2:2:end);
69
70
71 end
72
73 function [ y ] = SymX( X,n )
74 %SymX Symmetric extension of data set X
75 %   Returns value n of the symmetric extension of the data set X where the
76 %   borders are extended around the whole point. Also X starts from 0
77 %   instead of the usual matlab 1.
78 X=wextend('1','symw',X,size(X,2)-2,'r');
79 M=size(X,2);
80
81 N=mod(n,M);
```

```
82
83 y=X(N+1);
84
85 end
```

Listing 9: Computes the inverse (9,7) Wavelet transform

```
1  function [ Y ] = InverseFilter97( w,w2 )
2  %InverseFilter97 Applies the inverse CDF9/7 wavelet transformation
3  %   Applies the inverse wavelet transformation using the the 9/7
4  %   Cohen-Daubechies-Feauveau biorthogonal wavelet. The algorithm is very
5  %   inefficient but shows the main principles discussed in the paper.
6
7  %Low pass analysis filter
8  h(1) = 0.026748757411;
9  h(2) = -0.016864118443;
10 h(3) = -0.078223266529;
11 h(4) = 0.266864118443;
12 h(5) = 0.602949018236;
13 h(6) = h(4);
14 h(7) = h(3);
15 h(8) = h(2);
16 h(9) = h(1);
17
18
19 %high pass analysis filter
20 g(1) = 0.091271763114249 5;
21 g(2) = -0.057543526228500;
22 g(3) = -0.591271763114247;
23 g(4) = 1.115087052456994;
24 g(5) = g(3);
25 g(6) = g(2);
26 g(7) = g(1);
27
28 %lowpass synthesis filter
29 h2(1)=-g(1);
30 h2(2)=g(2);
31 h2(3)=-g(3);
32 h2(4)=g(4);
33 h2(5)=-g(3);
34 h2(6)=g(2);
35 h2(7)=-g(1);
36
37 %highpass synthesis filter
38 g2(1)=h(1);
39 g2(2)=-h(2);
40 g2(3)=h(3);
41 g2(4)=-h(4);
```

```
42  g2(5)=h(5);
43  g2(6)=-h(4);
44  g2(7)=h(3);
45  g2(8)=-h(2);
46  g2(9)=h(1);
47
48  %Upscaling the coefficients
49  for k=1:size(w,2)
50      Uw(2*k-1)=w(k);
51      Uw(2*k)=0;
52  end
53
54  for k=1:size(w2,2)
55      Uw2(2*k)=w2(k);
56      Uw2(2*k-1)=0;
57  end
58
59
60  %Reconstruction using the synthesis filters
61  for k=0:size(w,2)+size(w2,2)-1
62      sum1=0;
63      sum2=0;
64      %lowpass synthesis filter
65      for l=-3:3
66          sum1=sum1+h2(l+4)*SymX(Uw,k-l);
67      end
68      %highpass synthesis filter
69      for l=-4:4
70          sum2=sum2+g2(l+5)*SymX(Uw2,k-l);
71      end
72      %Return reconstruction
73      Y(k+1)=sum1+sum2;
74  end
75
76  end
77
78  function [ y ] = SymX( X,n )
79  %SymX Symmetric extension of data set X
80  %   Returns value n of the symmetric extension of the data set X where the
81  %   borders are extended around the whole point. Also X starts from 0
82  %   instead of the usual matlab 1.
83  X=wextend('1','symw',X,size(X,2)-2,'r');
84  M=size(X,2);
85
86  N=mod(n,M);
87
88  y=X(N+1);
89
90  end
```

Bibliography

[1] Jaideva C. Goswami, Andrew K. Chan, *Fundamentals of wavelets : theory, algorithms, and applications,* Wiley, Hoboken, 2011

[2] Yun Q. Shi, Huifang Sun, *Image and video compression for multimedia engineering : fundamentals, algorithms, and standards*, CRC Press, Boca Raton, 2008

[3] Khalid Sayood, *Introduction to data compression*, Morgan Kaufmann, Waltham, 2012

[4] Tinku Acharya, Ping-Sing Tsai, *JPEG2000 standard for image compression : concepts, algorithms and VLSI architectures*, Wiley-Interscience, Hoboken, 2005

[5] David Salomon, *Data compression : the complete reference*, Springer, New York, 2004

[6] Timothy Sauer, *Numerical analysis*, Pearson, Boston, 2012

[7] Ronald N. Bracewell, *The Fourier transform and its applications,* McGraw-Hill, New York, 1986

[8] John Miano, *Compressed image file formats : JPEG, PNG, GIF, XBM, BMP*, Addison-Wesley, Harlow, 1999

[9] Ian H. Witten, Alistair Moffat, Timothy C. Bell, *Managing gigabytes : compressing and indexing documents and images*, Morgan Kaufmann Publishers, San Francisco, 1999

[10] Zhou Wang, Alan Conrad Bovik, Hamid Rahim Sheikh, Eero P. Simoncelli, *Image Quality Assessment: From Error Visibility to Structural Similarity*, IEEE Transactions on Image Processing Vol. 13 no. 4 (2004), 600-612

[11] C. E. Shannon, *A Mathematical Theory of Communication*, The Bell System Technical Journal Vol 27 (1948), 379-423,623-656

[12] Julius Smith, *Mathematics of the Discrete Fourier Transform (DFT) with Audio Applications*, https://ccrma.stanford.edu/~jos/mdft/

[13] K. R. Rao, D. N. Kim, J. J. Hwang, *Fast Fourier Transform: Algorithms and Applications*, Springer, New York, 2010

[14] Steven B. Damelin, Willard Miller, Jr., *The mathematics of signal processing*, Cambridge, New York, 2012

[15] S. Allen Broughton, Kurt Bryan, *Discrete Fourier analysis and wavelets : applications to signal and image processing*, Wiley, Hoboken, N.J., 2009

[16] Albert Boggess, Francis J. Narcowich, *A first course in wavelets with Fourier analysis*, John Wiley & Sons, Hoboken, N.J., 2009

[17] Jon Kleinberg, Éva Tardos, *Algorithm design*, Pearson, Harlow, Essex, 2014

[18] Sanjeev Arora, Boaz Barak, *Computational complexity : a modern approach*, Cambridge University Press, New York, 2009

[19] E. Oran Brigham, *The fast Fourier transform and its applications*, Prentice Hall, Englewood Cliffs, N.J, 1988

[20] http://math.berkeley.edu/~berlek/classes/CLASS.110/LECTURES/FFT

[21] CCITT, *T.81: Information technology - Digital compression and coding of continuous-tone still images - Requirements and guidelines*

[22] Mozilla Corporation, *Lossy Compressed Image Formats Study*, https://people.mozilla.org/~josh/lossy_compressed_image_study_october_2013/

[23] Benjamin Bross, *Overview of the HEVC Video Coding Standard*, Next Generation Mobile Broadcasting (2013), 291-318

[24] Jaideva C. Goswami, Andrew K. Chan, *Fundamentals of wavelets : theory, algorithms, and applications*, Wiley, Hoboken, N.J, 2011

[25] N. Ahmed, T. Natarajan, K. R. Rao, *Discrete Consine Transform*, IEEE Transactions on Computers Vol 23 no. 1 (1974), 90-93

[26] Robert G. Gallager, *Variations on a Theme by Huffman*, IEEE Transactions on Information Theory Vol 24 no. 6 (1978), 668-674

[27] Luigi Atzori, Jaime Delgado, Daniele Giusto, *Mobile Multimedia Communications*, Springer, New York, 2012

[28] E. Feig, Shemuel Winograd, *Fast algorithms for the discrete cosine transform*, IEEE Transactions on Signal Processing Vol. 40 no. 9 (1992), 2174-2193

[29] Jin Li, *Image Compression: The Mathematics of JPEG 2000*, Modern Signal Processing Vol 46 (2003),

[30] Eugenio Hernández, Guido Weiss, *A first course on wavelets*, CRC Press, Boca Raton, 1996

[31] D. Santa-Cruz, T. Ebrahimi, J. Askelöf, M. Larsson and C. A. Christopoulos, *JPEG 2000 still image coding versus other standards*, http://www.jpeg.org/public/wg1n1816.pdf

[32] Farzad Ebrahimi, Matthieu Chamik, Stefan Winkler, *JPEG vs. JPEG2000: An objective comparison of image encoding quality*, http://www.researchgate.net/publication/215482742_JPEG_vs._JPEG2000_An_objective_comparison_of_image_encoding_quality

[33] http://r0k.us/graphics/kodak/

[34] http://thomaspreece.com/thesis/