Digital Image Transformation and Compression

A Thesis submitted for examination for the Degree of Master of Engineering (Electrical)

By

E. Lenc

B. Eng (Elec.)

Victoria University of Technology

Department of Electrical and Electronic Engineering

Victoria University of Technology

P.O. Box 14428, MCMC,

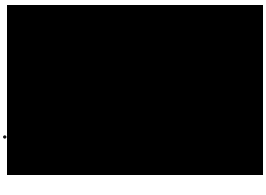
Melbourne, Victoria 8001,

Australia

August 1996

Statement of Originality

I, Emil Lenc, hereby declare that this submission is my own work and that, to the best of my knowledge, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of a university or other institute of higher learning, except where due acknowledgment is made in the text.



.....

Emil Lenc

Table of Contents

List of Figures

List of Tables

Abstract

Compression algorithms have tended to cater only for high compression ratios at reasonable levels of quality. Little work has been done to find optimal compression methods for high quality images where no visual distortion is essential. The need for such algorithms is great, particularly for satellite, medical and motion picture imaging. In these situations any degradation in image quality is unacceptable, yet the resolutions of the images introduce extremely high storage costs. Hence the need for a very low distortion image compression algorithm.

An algorithm is developed to find a suitable compromise between hardware and software implementation. The hardware provides raw processing speed whereas the software provides algorithm flexibility. The algorithm is also optimised for the compression of high quality images with no visible distortion in the reconstructed image.

The final algorithm consists of a Discrete Cosine Transform (DCT), quantiser, run-length coder and a statistical coder. The DCT is performed in hardware using the SGS-Thomson STV3200 Discrete Cosine Transform. The quantiser is specially optimised for use with high quality images. It utilises a non-uniform quantiser and is based on a series of lookup tables to increase the rate of computation. The run-length coder is also optimised for the characteristics exhibited by high-quality images. The statistical coder is an adaptive version of the Huffman coder. The coder is fast, efficient, and produced results comparable to the much slower arithmetic coder.

Test results of the new compression algorithm are compared with those using both the lossy and lossless Joint Photographic Experts Group (JPEG) techniques. The lossy JPEG algorithm is based on the DCT whereas the lossless algorithm is based on a Differential Pulse Code Modulation (DPCM) algorithm. The comparison shows that for most high quality images the new algorithm compressed them to a greater degree than the two standard methods. It is also shown that, if execution speed is not critical, the final result can be improved further by using an arithmetic statistical coder rather than the Huffman coder.

Acknowledgments

I wish to express my thanks to both of my supervisors, Alec Simcock and Ann Pleasants for their help, guidance and patience. Special thanks to Alec for his curry nights that gave me the spice of life to carry on. His special friendship and dedication were also a great inspiration.

Special thanks also should go to my family, in particular to Dong, his help and experience in the field of image compression was invaluable.

Finally, I owe special thanks to my wife Wendy for her untiring support and faith in me, without which I would not have had the will to continue. I love you very much!

Abbreviations

CISC	Complex Instruction Set Computer.
DCT	Discrete Cosine Transform.
DPCM	Differential Pulse Code Modulation.
DSP	Digital Signal Processing.
EPLD	Electrically Programmable Logic Device.
HVS	Human Visual System.
IDCT	Inverse Discrete Cosine Transform.
JPEG	Joint Photographic Experts Group.
KLT	Kahrunen Loeve Transform.
MOS	Mean Opinion Score
MPEG	Moving Picture Experts Group.
MSE	Mean Square Error.
OptIC	An acronym for the new algorithm described in this thesis.
PCX	A common run-length coding technique for compressing images on an IBM PC.
PPM	A common image format which provides no compression.
RGB	Red, Green, Blue colour coding model.
RISC	Reduced Instruction Set Computer.
YIQ	Intensity (Y) and Chrominance (I and Q) colour model.

1. Introduction

The need for image compression techniques has been apparent for many years now. This need has lead to the design of many algorithms and many different implementations of these algorithms. The more common of these are described in chapter 2. Unfortunately, most of these algorithms concentrate on increasing the compression factor rather than maintaining high-fidelity. They are generally aimed for video or television quality images where some losses in quality can be tolerated.

During the literature survey no documentary evidence was found of research specifically aimed at high compression rates for high quality images. The emphasis here is to prevent the introduction of visible distortions into the image whilst still trying to optimise the compression factor. This form of compression would be extremely useful in applications where such levels of quality are a necessity e.g. medical imaging, satellite imaging and cinema quality motion pictures. The images in these applications are often of extremely high resolution and so require large storage requirements. They also demand high quality and no distortions are acceptable as life/death or profit/loss decisions may depend on very fine data contained within them. However, data storage comes at a cost and any increase in compression factor can produce a proportional decrease in storage cost.

This thesis sets out to define a high quality, high speed compression algorithm OptIC (Optimised Image Compressor) that can be utilised in applications where such quality

is a necessity. Particular emphasis is placed on optimising the compression factor whilst maintaining visible image quality.

Chapter 3 defines the aims of the project, the basic principles of the OptIC algorithm required to achieve these aims and how the algorithm will be tested to ensure that the aims have been fulfilled. Chapters 4 to 7 explain the functional components of the algorithm in further detail. Each of the functional components are described in detail. They are then tested as part of a stepwise refinement and conclusions made at the end of each chapter describing the effectiveness of the implementation of the OptIC algorithm to that point. Chapter 8 takes the entire algorithm and compares it with currently existing algorithms to measure its overall performance.

A final conclusion and discussion of the advantages and disadvantages may be found in chapter 9. This chapter also introduces some research avenues that may be pursued in future.

2. Research Background

2.1 Introduction

Digital image compression provides a means by which the storage requirements of a digitised image may be reduced with little or no reduction in quality. This is done by removing redundancies which may occur within the image. Consider a typical motion picture frame with a resolution of 6000 by 4000 pixels (picture elements) each with a colour resolution of 24 bits allowing 16 million possible colours. A single frame of this image requires 576 Mbits of storage. If this image was compressed by half, the total storage costs would be halved. This would also save costs if the image was to be transmitted in the compressed form as it could be transmitted in half the time of the original.

A number of factors allow images to be compressed to a greater degree than other forms of data, such as text or binary code. Firstly, image data is two-dimensional providing correlation in two directions; this allows us to predict adjacent pixel intensities with greater accuracy. In motion picture sequences, the data may be considered to be three-dimensional thus providing correlation in three directions and so further improving the prediction of adjacent pixels. Secondly, the restored image data does not need to be exactly the same as the original image since the human eye can not perceive certain levels of detail or changes in intensity.

An image compression algorithm is generally, though not always, composed of two basic components: a predictive function and a symbol coding function. The predictive function attempts to reduce the entropy of the data to be compressed; see chapter 3 for the definition of entropy in this context. The predictor does not normally perform any compression, it only maps the data into a different form that is more readily compressed by the symbol coding function. The predictor can be either *lossless* or *lossy* depending upon whether or not it introduces errors after reconstruction of the data.

The symbol coding function generally takes the form of an entropy or statistical coder and is the component that performs the compression. These coders are *lossless* and so do not introduce any further error into the data after reconstruction. They are treated in more detail in chapter 7.

Both *lossless* and *lossy* techniques exist to compress motion pictures and colour images.

2.2 Lossless Compression Techniques

The term *lossless* implies that the coding method used is entirely circular or reversible, i.e. the compression-decompression procedure returns the image bit for bit to its original state. A *lossless* technique either has no predictor function or has a predictor function that is *lossless*. In general these algorithms will compress images by factors in the range of two to three [THE89, QUI93].

It is possible to have a *lossy* algorithm method which may appear to be *lossless*. In this situation the predictive component of the algorithm only introduces errors in those areas which would not be obvious or noticed because of imperfections in the eye.

2.2.1 Run-Length Coding

Run-length coding [HUA74, THI92] is most effective in coding data which contains large strings of the same value. It replaces large symbol strings with a shorter run-length code. That code contains an identifier code, the length of the string and the data contained within the string. The predictor of the run-length coder assumes that adjacent pixels will most probably contain the same value. For this reason, it is generally used for compressing cartoon-style images containing large patches of uniform colouring. It is also useful for compressing bi-level images such as FAX images as they often contain large areas of white space.

The compression achieved with run-length coding is not very good when it is used to code gray-scale or colour images, particularly if the images contain noisy data or constantly varying shades and colours. The *PCX* image format is a commonly used image compression format that incorporates run-length coding.

2.2.2 Statistical Coding

Statistical or entropy coding [THI92] is useful for compressing data with large quantities of a particular data value. This is achieved by assigning shorter codes for those data values which are statistically more probable and longer codes for those data

values which are less probable. Though it is possible, a statistical coder is not particularly effective when used on its own, generally providing only about 20% compression. To be effective, it is generally performed after a predictive function.

An inherent problem of this form of coding is the need for statistical data about the input data. Fixed coders assume a particular set of statistics for the input data but perform poorly when the input data statistics deviate from this. Adaptive coders generate the statistical data on the run and so adjust to varying statistical trends. Both of these coders perform poorly when the input data is extremely noisy or random.

2.2.3 Differential Pulse Code Modulation (DPCM)

In DPCM [COR90, EKS84] the predictive stage relies on the high degree of correlation that occurs between neighbouring pixels in an image. The prediction algorithm attempts to predict the value of the next pixel by taking into account the values of previous pixels. The difference of the predicted and actual pixel value is then stored as this is typically smaller in magnitude than the actual pixel value itself.

Improved prediction may be obtained by taking into account more pixels on the current or previous line of the image to predict the value of the next pixel with greater accuracy. Once again, only the difference value would be stored.

The final stage of the DPCM algorithm is the coding section. Here an entropy coding method is used to perform the actual compression. This technique, on average, compresses images by a factor of about two.

2.3 Lossy Compression Techniques

In order to obtain higher compression ratios for images, it is necessary to use *lossy* compression techniques. *Lossy* techniques are not circular, i.e. the compression-decompression procedure will produce distortions in the reconstructed image. This implies that the predictor block is present and that it introduces errors into the reconstructed image.

2.3.1 Subsampling

Subsampling [GRU92, QUI93] is a very but simple form of image compression. The predictor simply reduces the horizontal and vertical resolutions upon compression. For example a compression ratio of 4:1 can be achieved by reducing the horizontal and vertical resolutions by a factor of two. The decompression process would then need to expand each pixel into a 2x2 pixel block containing the same colour and intensity of the sub-sampled pixel. The distortion in this case becomes apparent even at relatively low rates of compression, as the pixel expansion tends to produce a blocking effect.

Sophisticated subsampling techniques attempt to interpolate between the pixels of the compressed image. This, in effect, is a form of low pass filtering and tends to soften the appearance of the image.

2.3.2 Transform Coding

Transform coding techniques are typically more computationally intensive than other compression methods. This technique requires two steps : transformation and coding [EKS84, AMO89, BAR88, KOU89, RAB89].

The transform forms the predictor function of the algorithm and operates on a two dimensional block of data to produce an array in which most of the image information is stored in as few elements of the array as possible. The transformed data represents the levels of the frequencies in the two-dimensional space within a block. The block size of the transform ranges between 4x4 and 16x16. Block sizes smaller than this do not produce good results and larger block sizes become computationally difficult to calculate.

The coding stage of this method requires the selection, quantisation and storage of the transformed data. The most significant values, i.e. those values which hold the majority of the image information are quantised the least so as to keep their values intact. The resulting data is then coded using similar coding techniques to those mentioned in section 2.2.3 for predictive coding. Note that the quantisation stage causes the distortions which may be visible after decompression.

The compression ratios obtained from transform coding are much greater than those obtained from predictive coding. This is especially so where there is little correlation between neighbouring pixels in an image. The commonly used orthogonal or unitary

transforms for digital image coding, such as DCT, only work well if the inter-pixel correlation is high [CLA85]. In general there is a trade-off between the compression ratio and the fidelity of the resulting image.

The Kahrunen Loeve Transform, Discrete Cosine Transform and the Fractal Transform are three of the most commonly used transforms.

2.3.2.1 Kahrunen Loeve Transform (KLT)

The KLT [RAO90, STA88] is the optimum transform for image coding. It is, however, computationally slow as no fast algorithms exist. There are a number of related transforms which have been designed to provide a compromise between image quality, compression ratio and computational complexity. The most common of these is the Discrete Cosine Transform (DCT).

2.3.2.2 Discrete Cosine Transform (DCT)

The DCT [RAO90] is most frequently used for image compression. Its popularity is due to its being a very close approximation of the KLT. There are also quite a large number of fast algorithms available for evaluating the transform and its inverse. A number of hardware implementations of the DCT have also become available. For these reasons the DCT is also used in the OptIC (Optimised Image Compression) algorithm.

2.3.2.3 Fractal Transform

The Fractal Transform [BAR88, SKA94, WOO94] is the most recent of the three transforms discussed here. The transform itself can produce extremely complex yet natural looking images with only a small number of coefficients. Unfortunately the process of obtaining the coefficients required to produce a particular image is not a simple process. Barnsley [BAR88] invented the fractal transform and has since brought out a number of software packages which utilise this transform in image compression.

The fractal transform is an asymmetric algorithm, i.e. it takes a great deal longer to compress an image than it does to decompress it. The compression time is about 48 minutes on a 33Mhz 80486-based machine for a 640x400 pixel 24 bit colour image [DET92]. The decompression time for the same image is performed in the order of a few seconds.

One problem with the fractal transform is the lack of any quantitative quality measurements. There are often claims of extremely large compression ratios (75:1) but no mention of the quality of the restored image [SAU94]. Also the algorithm is registered by Barnsley as a trademark thus leaving little room for experimentation.

2.4 Interframe Compression Techniques

2.4.1 Introduction

Interframe compression [EKS84, QUI93] is useful for sequences of images or motion picture images. In most sequences of images there is a high level of redundancy in the information between two consecutive frames, particularly when the background of the image is constant and only the foreground varies. Interframe compression works to reduce this redundancy. To allow motion to commence from various points in the image sequence without decoding the entire image sequence, reference frames are taken periodically. These reference frames form the points between which or from which the compression will take place. The use of reference frames also prevents continuous degradation in the image quality in a lossy algorithm which would occur if only one reference image was taken and all subsequent images were based on this.

2.4.2 Predictive

In predictive coding [GRU92] for interframe compression, the difference between reference frames is taken and later used to re-create the second reference frame from the first. For images where the background is constant, the differencing will produce a large number of zeros which are easily compressed. The predictive algorithm is normally a lossless coder.

2.4.3 Interpolative

The interpolative method [GRU92, QUI93], also known as average prediction or forward and backward prediction, calculates the current frame based on differences between the last and the next reference frame. So, for example, only every second or third frame could be kept and the intermediate frames would then later have to be predicted. As this algorithm requires the average of frame information, it is a lossy one.

2.4.4 Motion Prediction

Motion prediction [GRU92, QUI93] attempts to isolate moving objects and track their movements across subsequent frames. More advanced algorithms may also determine if the object has rotated or changed in scale to provide improved results in compression. In general, these algorithms are rather complex and difficult to implement. Also, the coding algorithm is often more complicated than the decoding algorithm. The motion predictor often makes approximations in order to simplify the coding procedure; this leads to losses.

2.5 Colour Space Transformation

2.5.1 Introduction

Colour space transformations provide methods of reducing redundancy in colour images. These in general make use of characteristics of the Human Visual System (HVS) which indicates that the human eye cannot perceive colours as well as it can

perceive intensities. The human eye is also more sensitive to certain colours than to others. For this reason the common RGB (Red, Green, Blue) colour model is more often transformed to the YIQ model where Y is the intensity component and I and Q are the chrominance or colour components. The YIQ format more accurately models the eye's capabilities. The transformation is shown in (2.1).

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.522 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.1}$$

2.5.2 Quantisation

Quantisation of the colour space [QUI93] simply reduces the precision of the colour carrying information. This will reduce the total number of colours that may be represented in the image. In general, before the quantisation is performed, the colour space is transformed to the YIQ format as described in section 2.5.1. By doing so the colour set will still contain those colours which are most clearly identified by the human eye and reduce redundancy. In this case the chrominance (IQ) is normally quantised more than the intensity (Y).

2.5.3 Subsampling

Subsampling in the colour space [QUI93] averages the colours in a block of pixels and in effect reduces the resolution of the colour image whilst leaving the resolution of the intensity image intact.

2.6 *Common Compression Standards*

2.6.1 JPEG (Joint Photographic Experts Group)

The JPEG algorithm [QUI93] defines methods for coding still picture images using both lossless and lossy techniques. The lossless technique is based on predictive coding as described in section 2.2.3. The lossy technique is based on the discrete cosine transform (DCT) and a combination of run-length and statistical coding. The DCT is an 8x8 DCT and is performed on each of the three colour channels.

The algorithm has been implemented on a variety of systems and is available in software and hardware versions. The implementations range from low cost, low performance systems to high cost, high performance systems.

Although this algorithm is capable of producing high quality reproductions of images after compression it is not optimised for this purpose. Instead, it is aimed at the average consumer market where high rates of compression rates are preferred to high quality.

Feature	JPEG	OptIC
DCT Block Size	8x8.	16x16.
Coefficient Ordering	Zig-Zag.	Proportional to the coefficients probability of being zero.
Block Ordering	Left to right.	Alternating left to right then right to left.
Grouping	Coefficients in same block grouped.	Like coefficients grouped.
Compression Stage	A run-length coder stage only.	A run-length and statistical coder stage.

Table 2.1 A comparison of the JPEG algorithm with OptIC.

As the OptIC algorithm is also based on the DCT it is useful to see how it differs from the JPEG algorithm. The major differences are briefly highlighted in Table 2.1.

2.6.2 MPEG (Moving Picture Experts Group)

The MPEG algorithm [FAI95, QUI93, THI92] is an extension of the lossy techniques outlined in the JPEG algorithm. It defines a standard for coding moving pictures with a sound track. The MPEG does not precisely detail the procedure for compressing video images as does JPEG; it merely specifies the format and data rate of the output bitstream as well as a set of compression techniques that can achieve varying degrees of quality. MPEG uses JPEG for the intraframes together with combinations of both predictive and interpolated motion compensation and sub-band coding for the audio.

As the MPEG standard only specifies the output format, most of the products currently on the market only perform MPEG decompression. Only three real-time high quality encoders existed by July 1995 [GIL95]. As with JPEG, MPEG is already available in both software and hardware implementations. Recently (July 1995), both SGS-Thomson Microelectronics and Zatek introduced MPEG decoders for decoding both video and audio.

The output quality of the MPEG algorithm is relatively poor as it is optimised for high speed and high levels of compression rather than for quality. For this reason it is not feasible for use in high quality motion picture image compression.

3. Outline of Research

3.1 Introduction

An outline of the research aims is to be presented together with a proposed algorithm structure to realise these aims. A test platform and a set of test methods are also defined. These provide a means to determine whether or not the set aims have been achieved.

3.2 Research Aims

3.2.1 General Aims

The aim of this research is to compress motion picture quality images using an optimal combination of hardware and software to minimise costs and maximise performance. Most compression systems are either fully software based and require extremely high performance computers to provide any reasonable performance [GRU92, KOU89, CHA87] or they are fully hardware based and suffer from high costs and inflexibility [LEO93, TSA89, ART88, STA88, REZ87].

3.2.2 Specific Aims

- To transform digitised images into a format more suitable for manipulation using computers, and then to reduce the amount of information required to represent the original optical impression by at least a factor of 2, whilst maintaining fidelity of the original image.

- To store and retrieve an image, and to reconstitute the original optical image from the stored information.

- To produce compression and reconstitution algorithms that are adaptable for use in processing motion picture quality images (approx. 6000x4000x24 bits) in the order of five seconds with a minimum reduction of image quality. Initially, smaller images will be used (256x256x8 bits or 512x512x8 bits); all results will then be extrapolated for the full resolution.

- To investigate various transformation and compression techniques.

- To choose and optimise a compression / transform technique for use in a hardware / software implementation.

3.3 Basic Structure of OptIC Algorithm

The basic structure of the OptIC (Optimised Image Compression) algorithm is shown in Fig. 3-1. It can be seen from this diagram that the compression and decompression components of the OptIC algorithm are each composed of six distinct functions.

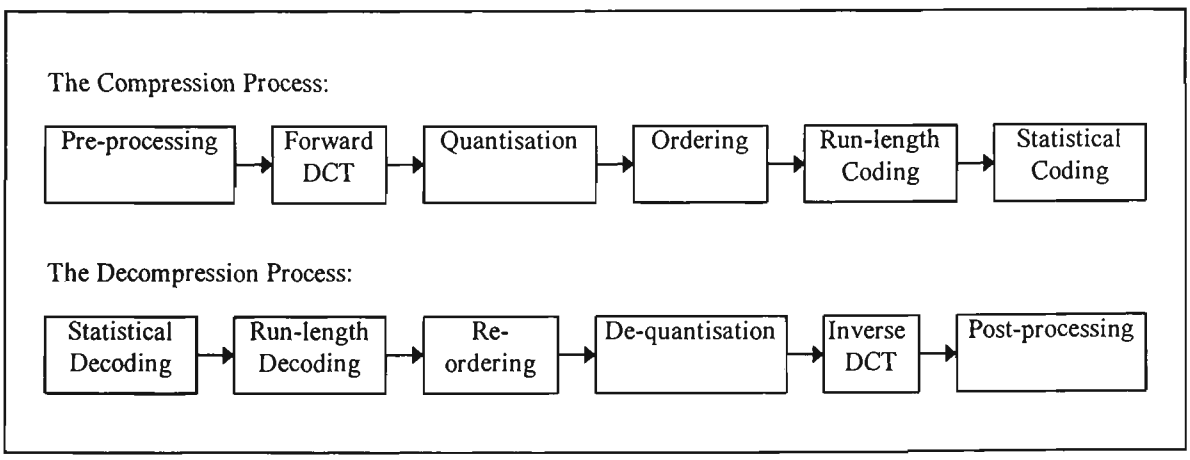


Fig. 3-1 The basic structure of the OptIC compression and decompression algorithm.

3.3.1 The Compression Process

The pre-processing block conditions the input image data so that it is in a format readily accepted by the forward DCT function. The pre-processing also adds a bias to the image data so that the average of the image data lies approximately about the zero value. This block does not add any error to the data nor does it in any way affect the entropy of the original image data.

The forward DCT function transforms the image data and outputs an array of DCT coefficients. The number of coefficients output is equivalent to the number of pixels in the input image. As these coefficients are twice the precision of the pixel data in the original image, this block actually doubles the storage requirement of the image. It does, however, improve the entropy of the image and so the image can be compressed to a greater extent than the original image by a statistical coder. The DCT process introduces an extremely small amount of error to the image data and so forms the first lossy function.

Once the image has been transformed it is then quantised to reduce the precision of the coefficients which do not have a great affect on our perception of the image. This improves the entropy yet again but in the process also increases the error introduced into the image data and as such, it forms the second lossy function. The errors introduced here, though much greater than those introduced in the DCT process, are still not visible to the human eye.

The effect of quantisation on the image tends to reduce a number of coefficients to zero. By ordering the coefficients produced by the DCT, it is possible to increase the chance of generating large lengths of zero values. These can then be effectively compressed by the run-length coder.

The run-length coder function replaces repetitions of a like value with one or two smaller values. This effectively is the first compression function in the algorithm and is completely lossless. As the run-length coder looks at the relationship between neighbouring values, it can produce better results when used in conjunction with a statistical coder than those obtained by using the latter alone.

The run-length coded data is finally passed through a statistical coder which forms the second and final compression function in the algorithm. The statistical coder is also a lossless coder which looks at the statistics of the data and replaces frequently occurring data symbols with short symbols and less frequently occurring data symbols with longer symbols.

3.3.2 The Decompression Process

The decompression process is basically the exact reverse of the compression process. The compressed image data is first statistically decoded to produce the run-length coded data that is then passed through the run-length decoder to produce the ordered and quantised DCT coefficients. These coefficients are re-ordered back to their original positions and then de-quantised to approximate the original coefficient values that were obtained after the forward DCT process. As there were errors involved in

the original quantisation function, the values of the coefficients that are output from the de-quantiser are only an approximation of those that were originally produced by the forward DCT process.

Once the coefficients have been restored they are passed through the inverse DCT. This function will add further error though in this process it is very minute. The output of the inverse DCT is finally passed through the post-processing function that removes the bias that was introduced by the pre-processing function. The final output will be, apart from the introduced errors, the reconstructed version of the original image. The accumulated errors obtained throughout the entire compression and decompression process are not visible to the human eye.

3.4 Testing Platform

All of the tests were performed on an IBM compatible computer based on the 80486SX processor running at a clock speed of 33 MHz (this does not include a floating point processor). The computer was equipped with 8 Mb of memory, 410 Mb Hard disk and 256 Kb of cache memory. The hard disk was controlled by a standard VESA local bus IDE hard disk controller card. A Cirrus Logic VL-VGA-24 graphics card fitted with 1 Mb of VRAM was used. It is common practice to use a Sony Grade 1 monitor for subjective evaluation exercises; but as one was not available, an NEC Multisync 4D monitor in True colour mode (24 bit per pixel or 16 million colours) was used as it was the best available alternative.

The software for the OptIC algorithm was developed using a Borland C++ Compiler version 4.0 for Microsoft Windows. The algorithm was executed under Microsoft Windows version 3.1.

3.5 Test Procedure

Where applicable four important tests were performed after the introduction of each function to measure progress and to provide a progressive indication of the performance of the compression algorithm. These four tests are : measuring the error introduced by a lossy function, the timing benchmarks with the function included, the entropy of the data after introduction of a function and the size of the output data after introduction of a compression function.

3.5.1 Error Measurements

A number of error measurement techniques exist, the most common of these is the Mean Square Error (MSE) as defined in (3.1) [RAO90] for an *NxM* image where $x(m,n)$ is an element in the original image and $\hat{x}(m,n)$ is an element in the reconstructed image.

$$MSE = \frac{1}{MN}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}\left\{x(m,n)-\hat{x}(m,n)\right\}^2 \qquad (3.1)$$

This form, though very commonly used, does not incorporate the Human Visual System (HVS) and so does not provide a real indication of how much of the error the human eye can visualise.

A number of attempts have been made to define a method of measuring error which would be perceived by the human eye [HOS86, MIY85] but these methods are not entirely accurate and it has not been proved that they apply for all types of images. As such they would not form a reliable method of testing the image quality. Instead a form of Mean Opinion Score (MOS) was devised. Images were subjected to increasing levels of compression and the image qualities verified by individual inspection at very close distances from the monitor screen. The image inspection was performed by a group of ten randomly chosen university students, three staff members, five family members and myself. The group consisted of a wide spread of ages and technical ability, all individuals had to decide whether or not they could perceive any differences between the original image and the reconstructed image. The image on the display could be quickly swapped between original and reconstructed image so that any distortions may be easily observed. In order to satisfy the aims all the individuals had to agree that all of the images contained no visible distortions after reconstruction. A note was made (for both the JPEG and OptIC algorithms) of the greatest compression which each image could tolerate with no observer identifying errors. This represents a subjective mean opinion score of the algorithm's capability to compress without introducing visible errors (please see section 8.3 and Tables 8.3 and 8.4 for a description of the results).

3.5.2 Timing Benchmarks

In all of the test cases the test was executed ten times and timed using a stop-watch. The result was then divided by ten giving a result that is at least accurate to the nearest tenth of a second.

3.5.3 Entropy Measurements

The measurement of entropy gives an indication of how much further an image may be compressed if processed with an ideal statistical coder. It is thus an important test in determining the effectiveness of processes that only manipulate data but do not actually compress it, such as, the DCT and the quantisation processes. The entropy of an image can be defined by (3.2) where N is the number of symbols and $p(i)$ is the probability of the i^{th} symbol.

$$Entropy = -\sum_{i=1}^{N} p(i) \log_2 p(i) \tag{3.2}$$

The entropy value is the average number of bits required to code each of the possible symbols. This implies that a reduction in entropy would result in a smaller output data size if the data was to be passed through a statistical coder.

3.5.4 Output Data Size

The output data size of the image after processing is simply the size of the data when output to a file on the hard disk.

3.6 The Image Test Set

3.6.1 Standard Images

Fifteen standard images were used to form the standard image test set. These are images which are commonly used for testing image compression algorithms and provide a base from which the algorithm may be compared with other existing algorithms. The images were originally stored as colour images that were converted to intensity images and green component images as these were the components generally used by other compression techniques for comparison. The statistical characteristics for the 512x512x8 and the 256x256x8 bit pixel images are tabulated in Tables 3.1, and 3.2 respectively. A hard copy of each of the intensity images, both original and reconstructed, can be found in Appendix A.

Image	Image Type	average	variance	minimum	maximum	entropy
airplane	Intensity	179.132	2161.472	16	231	6.705764
airplane	Green	177.856	2687.813	0	234	6.805543
baboon	Intensity	129.694	1789.110	0	231	7.358139
baboon	Green	128.863	2282.638	0	236	7.475280
lena	Intensity	124.108	2298.224	25	245	7.447764
lena	Green	99.056	2796.058	1	248	7.595153
peppers	Intensity	120.464	2909.980	0	228	7.594303
peppers	Green	115.581	5629.022	0	237	7.518362
sailboat	Intensity	125.284	4297.523	2	239	7.485789
sailboat	Green	124.305	6026.763	0	249	7.646107
splash	Intensity	103.284	2662.172	9	242	7.258475
splash	Green	70.521	3638.671	0	247	6.916109
tiffany	Intensity	211.345	862.075	0	255	6.600483
tiffany	Green	208.631	1125.752	0	255	6.689978

Table 3.1 Statistical characteristics of the 512x512x8 bit standard images.

Image	Image Type	average	variance	minimum	maximum	entropy
beans1	Intensity	176.000	1469.183	32	204	5.724703
beans1	Green	180.638	2021.443	19	212	5.700033
beans2	Intensity	167.403	1849.728	24	207	6.242539
beans2	Green	170.869	2562.536	9	213	6.231839
couple	Intensity	33.424	1000.177	1	244	6.427370
couple	Green	30.122	989.483	0	254	6.064014
girl1	Intensity	58.872	1579.841	1	234	7.053766
girl1	Green	139.960	857.937	30	255	5.405798
girl2	Intensity	139.691	880.510	27	255	5.607469
girl2	Green	139.960	857.937	30	255	5.405798
girl3	Intensity	111.114	2471.922	14	250	7.261685
girl3	Green	99.275	2871.558	0	254	7.325684
house	Intensity	138.066	2128.970	16	240	6.504477
house	Green	133.004	3145.283	0	246	6.551604
tree	Intensity	129.115	4554.317	0	237	7.314443
tree	Green	124.906	5854.357	0	238	7.418101

Table 3.2 Statistical characteristics of the 256x256x8 bit standard images.

3.6.2 Supplementary Images

A number of supplementary images were also generated to support the test set. The first of these images, *testpatt*, was generated by a C program to test the compression algorithms ability to compress various frequencies of alternating low and high intensity.

Image	Image Type	average	variance	minimum	maximum	entropy
testpatt	intensity	112.026	16016.361	0	255	0.989348
wendy1	intensity	155.573	7239.374	1	255	6.822758
wendy2	intensity	65.610	3881.104	1	255	7.096842
wendy2	green	68.916	4113.347	0	255	6.544705
wendy3	intensity	109.056	4813.107	0	253	7.821364
wendy3	green	107.081	5252.605	0	255	6.820176

Table 3.3 Statistical characteristics of the 512x512x8 bit supplementary images.

The remaining images were obtained from photographs digitised using a high resolution Hewlett Packard colour image scanner. All of the images were 512x512x8 bits, and their characteristics are shown in Table 3.3. A hard copy of each intensity image, both original and reconstructed, can be found in Appendix A.

3.6.3 Image Data Format

All of the images are stored in format known as the PPM format. This format is used because the data is stored in a raw format with no compression. This format is also quite simple and so it is very easy to read.

The PPM file consists of a header which identifies the image format and also provides information about the width, height and the maximum intensity of the pixels. Following the header is the raw image data where each byte represents a pixel intensity and the pixels are stored consecutively on a line by line basis.

The image header format is shown in Fig. 3-2 where LF is the code for a line-feed (decimal 10) and the width, height and maximum intensity are ISO-coded values. An ISO-coded value is simply the required number stored with a separate character for each digit in that value. For example, the number 134 when stored in an ISO-coded format would become the characters {"1", "3", "4"}.

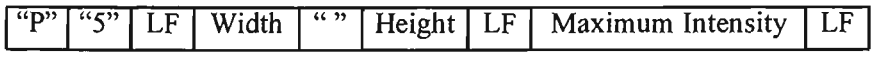


Fig. 3-2 PPM Image header format.

The software to read and write a PPM format image may be found in Appendix D. The C functions for the read and write are *LoadImage* (page D-11) and *SaveDecompressed* (page D-13) respectively.

4. The Discrete Cosine Transform

4.1 Introduction

The original Discrete Cosine Transform (DCT) is based on the Fast Fourier Transform (FFT) [RAO90, WAN84]. Since its discovery in 1974 [AHM74], its use has become widespread in digital signal processing (DSP), in particular for image processing. There are a number of reasons for its popularity, the DCT is real, separable, orthogonal and it approaches the statistically optimal transform, KLT [KAR47, LOE60]. The DCT does not suffer the computational problems involved in generating the KLT, as it is not dependent on signal statistics. Furthermore there are a large number of fast algorithms available to evaluate the DCT [LEE84, WAN84, SUE86, CHA87, KOU89]; a number of these have also been implemented in hardware for extremely high speed processing [JUT87, REZ87, ART88, QUI93, LEO93]. As the DCT is a separable transform, it is also possible to extend all the algorithms to multiple dimensions.

The DCT forms the heart of the image compression algorithm described in this thesis. The transform does not actually compress the data. It could instead increase the size of the data since the resolution of the output coefficients is generally greater than that of the input data. It does, however, have the useful property of reducing the entropy of the input data. The entropy of the data gives an indication to the extent of which an image may be compressed. By reducing the entropy the image is made more readily

compressible. This is achieved by transferring the majority of the input vector information or energy into small number of elements of the transformed output vector.

This chapter will begin by introducing the various forms of the DCT, describing how they can be used for image compression and discussing the various factors that can affect the level of compression obtained. Software and hardware implementations of the DCT are described. Both of the implementations will be optimised for high quality images, tested and compared in detail.

4.2 The One-Dimensional DCT

The family of one-dimensional forward and inverse DCTs as classified by Wang [WAN84] can be defined as shown in (4.1) to (4.4).

DCT-I

$$\left[C_{N+1}^{I}\right]_{mn} = \left(\frac{2}{N}\right)^{\frac{1}{2}}\left[k_m k_n \cos\left(\frac{mn\pi}{N}\right)\right] \quad m,n = 0,1,\cdots,N \tag{4.1}$$

DCT-II

$$\left[C_{N}^{II}\right]_{mn} = \left(\frac{2}{N}\right)^{\frac{1}{2}}\left[k_m \cos\left(\frac{m\left(n+\frac{1}{2}\right)\pi}{N}\right)\right] \quad m,n = 0,1,\cdots,N-1 \tag{4.2}$$

DCT-III

$$\left[C_{N}^{III}\right]_{mn} = \left(\frac{2}{N}\right)^{\frac{1}{2}}\left[k_n \cos\left(\frac{m\left(n+\frac{1}{2}\right)\pi}{N}\right)\right] \quad m,n = 0,1,\cdots,N-1 \tag{4.3}$$

DCT-IV

$$\left[C_{N}^{IV}\right]_{mn} = \left(\frac{2}{N}\right)^{\frac{1}{2}}\left[\cos\left\{\frac{\left(m+\frac{1}{2}\right)\left(n+\frac{1}{2}\right)\pi}{N}\right\}\right] \quad m,n = 0,1,\cdots,N-1 \tag{4.4}$$

where

$$k_j = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } j = 0, N \\ 1 & \text{for } j \neq 0, N \end{cases}$$

The relationship between the four classifications of the DCT can be summarised by (4.5) to (4.8) below.

DCT-I:

$$\left[C_{N+1}^{I}\right]^{-1} = \left[C_{N+1}^{I}\right]^{T} = \left[C_{N+1}^{I}\right. \tag{4.5}$$

DCT-II:

$$\left[C_{N+1}^{I}\right]^{-1} = \left[C_{N+1}^{I}\right]^{T} = \left[C_{N+1}^{I}\right. \tag{4.6}$$

DCT-III:

$$\left[C_{N+1}^{I}\right]^{-1} = \left[C_{N+1}^{I}\right]^{T} = \left[C_{N+1}^{I}\right. \tag{4.7}$$

DCT-IV:

$$\left[C_{N+1}^{I}\right]^{-1} = \left[C_{N+1}^{I}\right]^{T} = \left[C_{N+1}^{I}\right. \tag{4.8}$$

DCT-II is the discrete cosine transform first reported by Ahmed, Natarajan, and Rao [AHM74]. DCT-III is simply the transpose of DCT-II. DCT-IV is the shifted version of DCT-I. Note that only DCT-I and DCT-IV are capable of involution. DCT-II is the most commonly used of the four forms of DCT. As most software and hardware algorithms are based on this form, the remainder of this chapter focuses on this rather than the other three forms.

Using (4.2) it is possible to define the forward and inverse DCT as shown in (4.9) and (4.10) respectively.

Forward DCT-II :

$$X^{C(2)}(m) = \left(\frac{2}{N}\right)^{1/2} k_m \sum_{n=0}^{N-1} x(n) \cos\left[\frac{(2n+1)m\pi}{2N}\right] \quad m = 0, \cdots, N-1 \qquad (4.9)$$

Inverse DCT-II :

$$x(n) = \left(\frac{2}{N}\right)^{1/2} \sum_{m=0}^{N-1} k_m X^{C(2)}(m) \cos\left[\frac{(2n+1)m\pi}{2N}\right] \quad n = 0, \cdots, N-1 \qquad (4.10)$$

where

$$k_p = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } p = 0, N \\ 1 & \text{for } p \neq 0, N \end{cases}$$

The diagram shown in Fig. 4-1 shows the basis functions for DCT-II with N=16. A basis function is closely related to harmonics in a Fourier series. Any waveform can be created by summing different levels of each of the basis functions, just as any waveform can be created by summing the various harmonics of a Fourier series. The basis functions in the diagram were constructed by individually setting each of the coefficients in the DCT-II input vector to 127 with all other coefficients forced to zero. An IDCT-II is performed on this input vector and the respective results plotted. This was performed 16 times, once for each coefficient in the input vector. Note that the waveforms represent the intensities of the image and not the spectral qualities of the light emerging from the image. For this reason the peaks in the waveforms represent bright areas and the troughs represent dark areas.

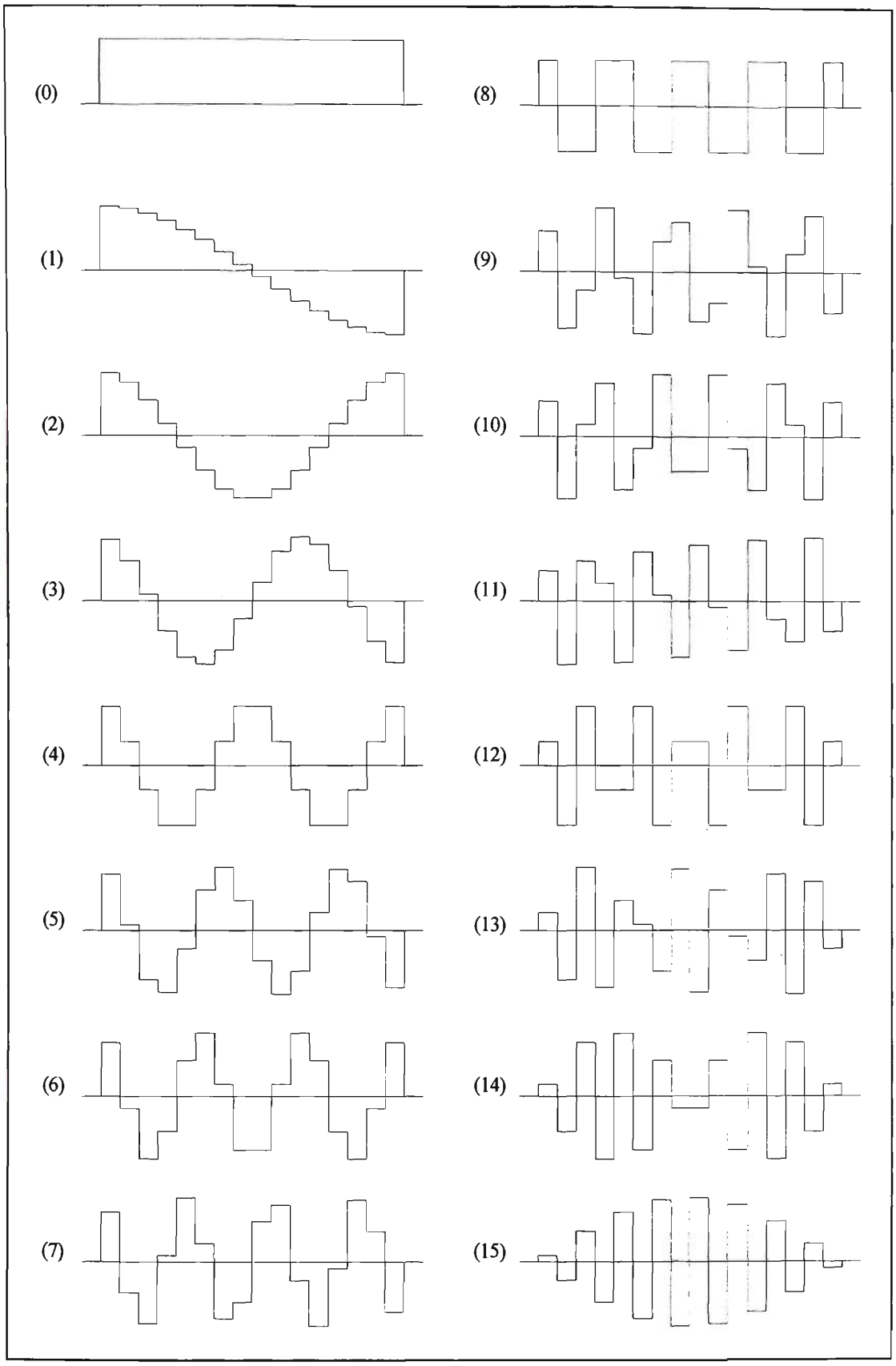


Fig. 4-1 Basis functions for DCT-II, N = 16 [RAO90].

4.3 The Two-Dimensional DCT

The DCT can also be performed in two dimensions. This is particularly useful when dealing with digitised images. Since the DCT is separable, the two-dimensional transform can be implemented by a series of one-dimensional transforms.

4.3.1 The Two-Dimensional DCT-II

Let g be an MxN input matrix and G its two-dimensional DCT-II. The uv^{th}-element of G is given by (4.11) below.

$$G_{uv} = \frac{2c(u)c(v)}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} g_{mn} \cos\left[\frac{(2m+1)u\pi}{2M}\right] \cos\left[\frac{(2n+1)v\pi}{2N}\right] \qquad (4.11)$$

where $u = 0, \cdots, M-1$ and $v = 0, \cdots, N-1$, and

$$c(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } k = 0 \\ 1 & \text{if } k \neq 0 \end{cases}$$

4.3.2 The Two-Dimensional IDCT-II

Similarly, the mn^{th}-element of g is given by (4.12) below.

$$g_{mn} = \frac{2}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} c(u)c(v)G_{uv} \cos\left[\frac{(2m+1)u\pi}{2M}\right] \cos\left[\frac{(2n+1)v\pi}{2N}\right] \qquad (4.12)$$

where $m = 0, \cdots, M-1$ and $n = 0, \cdots, N-1$.

4.3.3 Basis Functions of the Two-Dimensional DCT

As with the one-dimensional DCT, the two-dimensional DCT coefficients provide the building blocks necessary to reconstitute any two-dimensional waveform. In Fig. 4-2, the effect of several of the two-dimensional coefficients on the final waveform is shown. Once again, the coefficient of interest was set to 127 and all other coefficients were forced to zero. A two dimensional IDCT-II is then performed and the results plotted in three dimensional space.

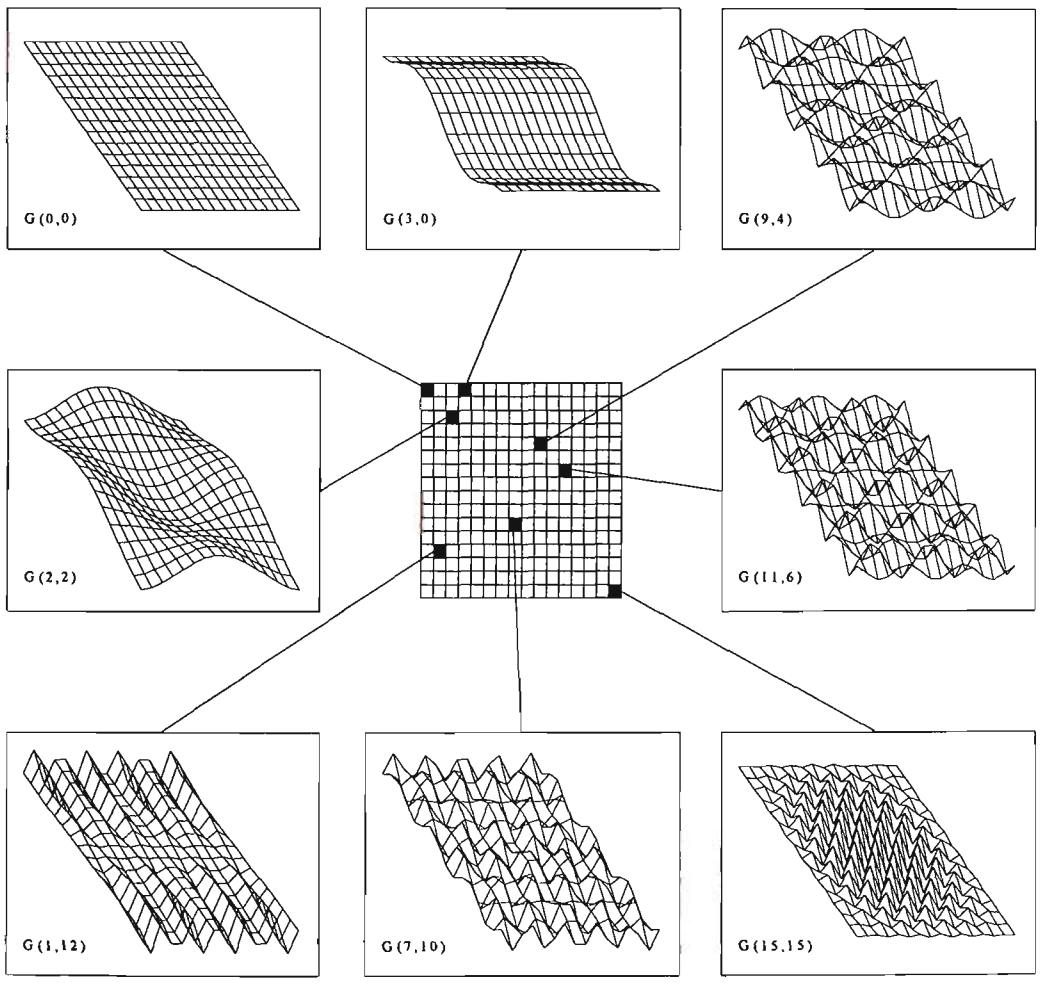


Fig. 4-2 Basis functions for the 2D-DCT-II, N = 16.

4.4 Factors Affecting Compression After Transformation

There are two controllable factors relating to the DCT which can affect the eventual level of compression possible after transformation. The first is the DCT block size, which has a direct relation to the DCT itself. The second is the level of quantisation of the coefficients after transformation. This is indirectly related to the DCT in that adequate knowledge is required about the DCT coefficients in order to provide low error quantisation. The following is an analysis of these two factors to determine their optimum settings for the compression of high quality images.

4.4.1 The DCT Block Size

Increasing the block size of the DCT will in most cases improve the entropy of the final result after transformation. Unfortunately errors in rounding will generally increase with larger transformations because of increasingly more complex calculations. The block sizes are limited to 4, 8 and 16 for reasons of computational efficiency in the software implementation and due to available block sizes in the hardware DCT transform device. In order to make an accurate comparison of the various size transforms a plot of the entropy versus the MSE of the images after restoration must be made. The different points in the plot are generated by scaling the DCT coefficients from 12 bit values down to one bit values in one bit decrements, doing so reduces the range of the coefficients by a factor of two in each step. As this will reduce the number of symbols, the overall entropy will as a consequence also be reduced. In each step the entropy of the transformed image and the MSE of the restored image were measured.

The graphs in Fig. 4-3 and Fig. 4-4 show plots of the entropy of the transformed image with respect to the MSE of the restored image for the DCT sizes 4x4, 8x8 and 16x16 for the images *Tiffany.Y* and *Testpatt.Y* respectively. It should be noted that most of the points in the graphs are contained below the MSE value of ten, for this reason their markers have been removed for the sake of clarity.

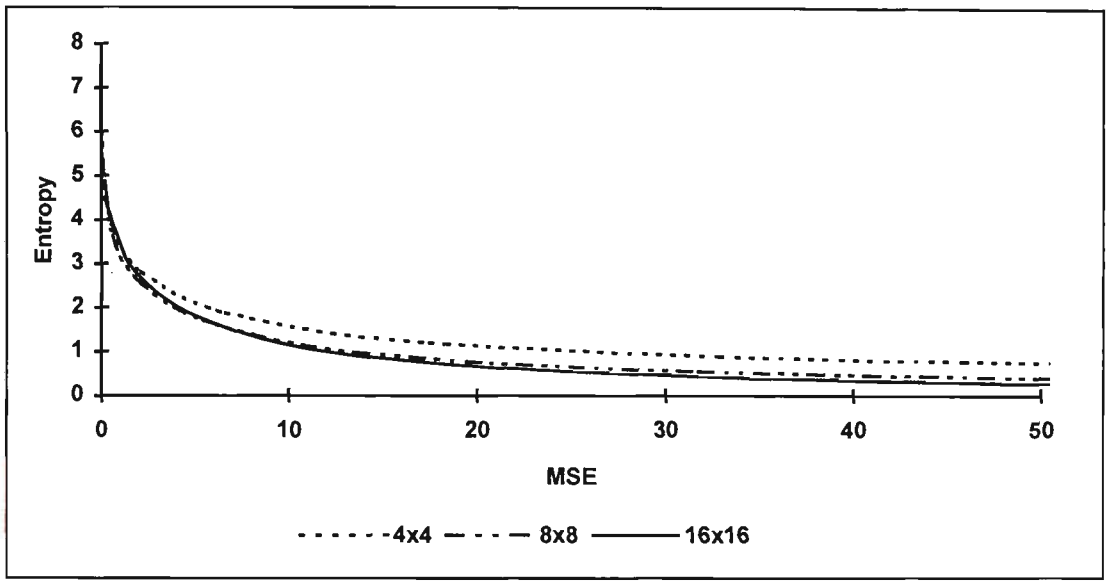


Fig. 4-3 A comparison of 4x4, 8x8 and 16x16 Discrete Cosine Transforms on Tiffany.Y.

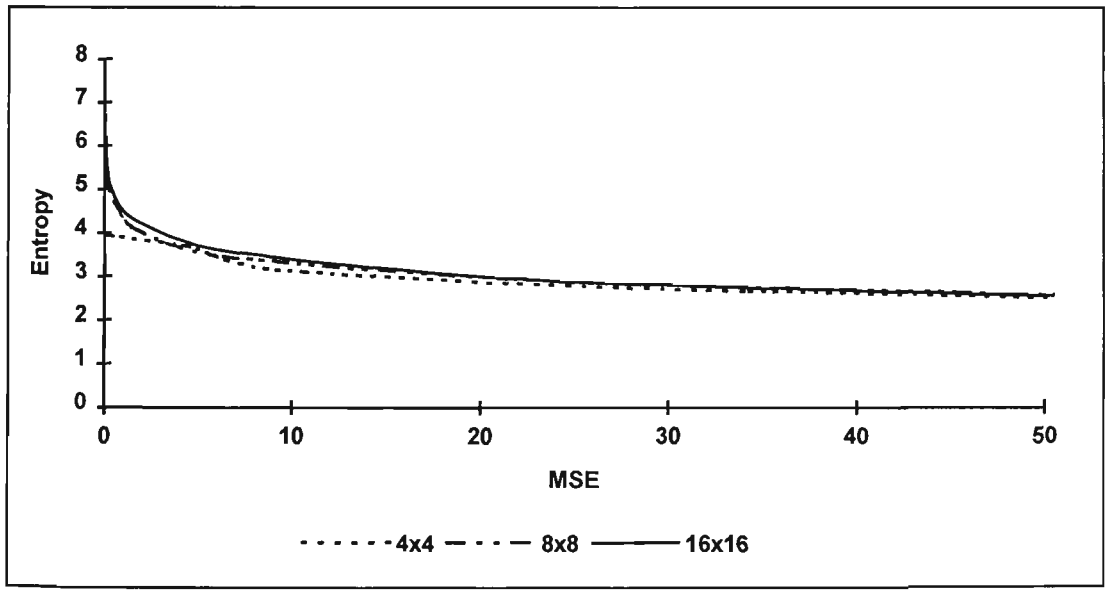


Fig. 4-4 A comparison of 4x4, 8x8 and 16x16 Discrete Cosine Transforms on Testpatt.Y.

All the other images in the test set were also examined and exhibited characteristics similar to that of those shown for *Tiffany.Y*. The graph shown in Fig. 4-3 for *Tiffany.Y* is a typical result that is obtained for all but one of the images in the test set. Note that for an MSE of less than two, it is advantageous to use a smaller block size for the DCT. This is due to the larger errors involved in generating the more arithmetically complex algorithms required for the larger block size DCTs. From the graphs it is, however, noticeable that the effect of the different block sizes on the graphs quickly reduce as the block size increases above 8x8. The algorithms dealt with in this project will be generating MSEs greater than two but at levels which are not visible to the human eye. At this level of error it is desirable to perform the DCT with the largest possible block size.

The results for *Testpatt.Y* shown in Fig. 4-4 show values which are inconsistent with those obtained from the other images in the test set. *Testpatt.Y* produced improved entropy results with a smaller block size. This particular image is artificially generated using software and so its characteristics are not normally found in natural images. It contains adjacent pixel values of minimum and maximum intensity only and so there are a large number of sharp transitions which are difficult for the larger block size transforms to reproduce without added error. It should be noted that the error effects are still minor and not perceivable at low error rate coding levels.

From Fig. 4-3 and Fig. 4-4, it can also be noted that the improvements in entropy are not increased significantly when the block size is increased to a size greater than 8x8.

Secondly, the largest block size is restricted by the device used for the hardware implementation and the calculation time for the software implementation. The hardware transform device limits the block size to 16x16 and the software implementation becomes impractically slow for block sizes greater than 16x16. A block size of 16x16 is the most suitable choice for the hardware and software implementation.

4.4.2 Quantisation of DCT Coefficients

To further improve the compression after the DCT the coefficients of the output transform are quantised. The coefficients should not be quantised equally or indiscriminately. Each coefficient plays a different role in building up the original image. The extent to which a particular coefficient is quantised depends on three factors: the visual importance of that coefficient, the amount of error introduced to the entire image by quantising that coefficient, and the improvement in entropy gained by quantising that coefficient.

4.4.2.1 Visual Importance of the Coefficients

The Human Visual System (HVS) [NGA86, TZO84] has flaws which allows certain forms of error in the reconstructed image to be, in effect, invisible. By the same token some errors become clearly visible if the flaws in the HVS make them stand out.

The HVS is most sensitive to the DC level of the image [TZO84], i.e. the average intensity level of the image. This is directly related to the DCT coefficient (0,0). Inappropriate quantisation of this coefficient will produce a great deal of blocking,

that is, the different blocks of the reconstructed image will have slightly altered average intensities giving a mosaic-like appearance. For this reason care must be taken to avoid unnecessary quantisation of this coefficient.

The HVS is least sensitive to high frequency changes in intensity. This relates to coefficients furthest from the coefficient (0,0), for example (15,15), (14,15) and (15,14). These coefficients can be quantised to reasonably high levels without any really noticeable effects. Extremely high quantisation of these coefficients tends to filter the image resulting in the reconstructed image appearing rather soft and perhaps unclear.

4.4.2.2 Errors Introduced Through Quantisation

Different coefficients of the DCT have differing sensitivities to quantisation and the resultant error after restoration. For example, in Fig. 4-2, even if G(0,0) and G(15,15) were quantised by the same amount, G(0,0) would introduce greater errors than G(15,15) as it carries more information about the image. The graph in Fig. 4-5 shows the average MSE for all the images in the test set where a given coefficient has been quantised by a factor of 256, that is, the eight least significant bits were removed. The graph was produced by normalising the results of each image and then taking the average for each of the coefficients.

From the graph in Fig. 4-5 it can be seen that the most sensitive coefficients are those close to the origin, along the zero X axis and along the zero Y axis. These are the low frequency coefficients. The most sensitive of all the coefficients is the DC coefficient.

Where the block size is 16, those coefficients that are not on the axis and are more than 3 units in distance from the origin are reasonably insensitive to quantisation. This seems to match quite accurately with the HVS. Further information on the quantisation of the DCT coefficients can be found in Chapter 5.

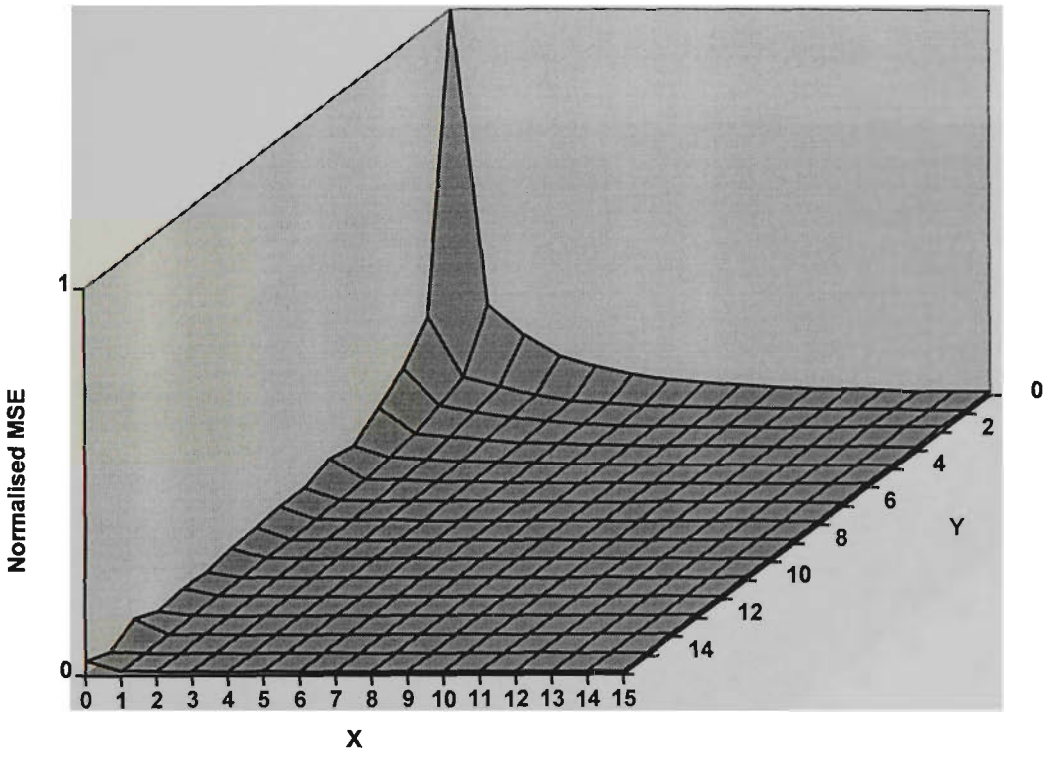


Fig. 4-5 Normalised MSE showing coefficient sensitivity to quantisation

4.4.2.3 Entropy Improvements Through Quantisation

The graph shown in Fig. 4-6 shows the normalised sum of the entropy of all the intensity images after quantising each coefficient individually by a factor of 256. The vertical axis has been exaggerated to highlight any trends, however small they may be. From this figure it can be seen that the entropy improves more after quantisation of the coefficients closer to the origin than with those further from the origin. It is also

interesting to note that the improvements gained are very minor. For example, only a 0.6% improvement is gained in quantising the DC coefficient as opposed to quantising the coefficient at (15,15). This is quite advantageous for the compression system as the coefficients closer to the DC coefficient are quite sensitive to quantisation as discussed in section 4.4.2.2.

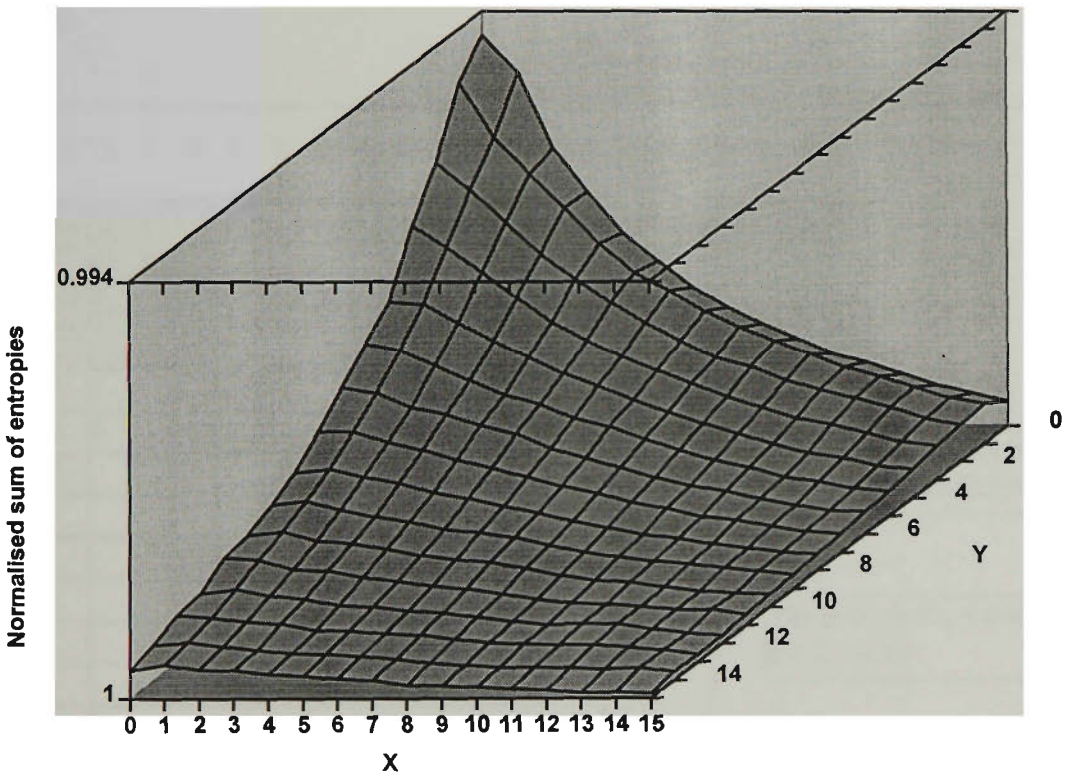


Fig. 4-6 Normalised sum of the entropies of all intensity images showing the effect of coefficient quantisation

4.5 Pre-Processing

The pre-processing stage is performed on the image before it can be transformed. It involves two very basic steps : Input Data Ordering and Biasing.

4.5.1 Input Data Ordering

The image data itself is a large two dimensional array of pixels. It is stored in memory as a one dimensional block with each new line of the image stored consecutively after the previous line. The DCT, however, expects the data to appear in a smaller two dimensional blocks, the size of which is determined by the block size of the DCT. For this reason input data ordering is required. This conversion process is also known as blockup.

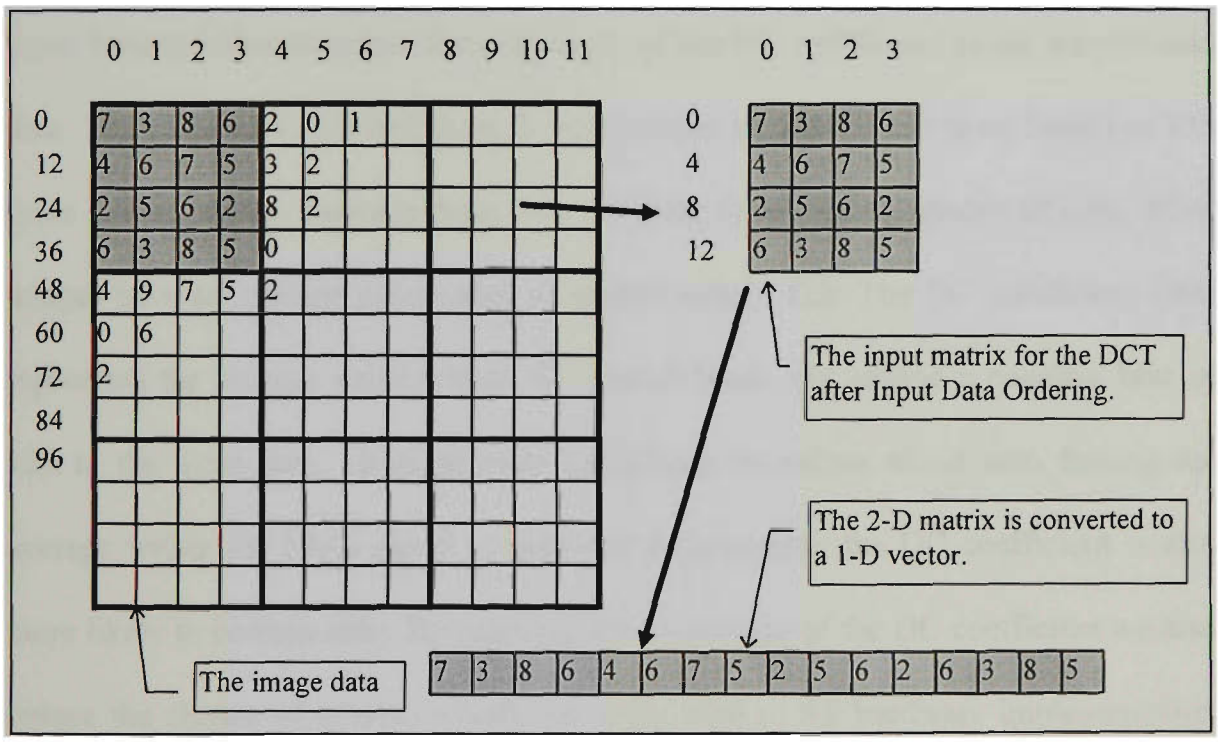


Fig. 4-7 Illustration of the data ordering procedure.

Fig. 4-7 shows the image data for a 12x12 pixel image. The pixels of the image are stored line by line in a large 144 pixel array. In order to perform a DCT transform with a block size of 4 on this image it is necessary to first split the image into 4x4 blocks as shown by the thick borders. The values inside the boxes indicate some

sample data, whereas the values outside of the arrays indicate the memory location that the data is stored in. Note that since the size of the DCT block is smaller than that of the image many pixels must be skipped to get to the next line of the DCT block. This explains the need for input data ordering. For the hardware algorithm the two-dimensional block must be converted to a one-dimensional vector before it can be transformed.

4.5.2 Biasing

Input biasing helps to reduce the magnitude of the DC coefficient in the transformed data. The input data obtained from the test images ranges from 0 (pure black) to 255 (pure white), values between these two extremes form various shades of gray. Most images have an average pixel value of approximately 128. The DC coefficient itself represents the average value within the current block. By adding a negative bias of 128 to the input data, it is possible to balance its values about zero forcing the average within the block closer to zero and subsequently the DC coefficient is also more likely to become zero. By reducing the magnitude of the DC coefficient we also reduce the chance of internal overflows in the case of the hardware implementation which may occur because of its limited internal resolution.

The effect of the biasing can be seen more clearly in the diagram Fig. 4-8. Note the lower level of peak magnitudes and that the waveform varies about zero on the Y axis.

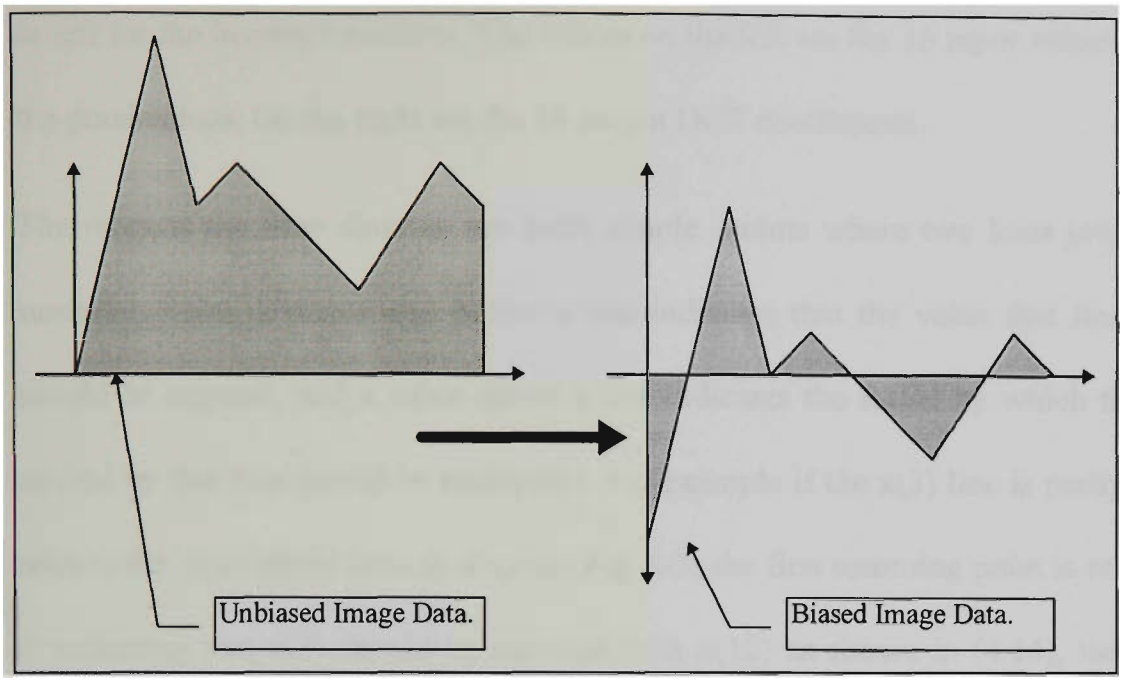


Fig. 4-8 Image data biasing

4.6 The Software Implementation

4.6.1 The Forward Transform

The forward transform is based on B.G. Lee's algorithm [RAO90] for the 1-Dimensional (1-D) DCT. This algorithm is one of the fastest available algorithms and is relatively simple to implement. Its flow diagram is shown in Fig. 4-9 where each coefficient *C(m)* is defined in (4.13) as:

$$C(m) = \frac{1}{2\cos\left(\frac{m\pi}{32}\right)} \qquad (4.13)$$

A flow diagram is often used to describe the implementation of transform algorithms. The flow can be interpreted from left to right for the forward transform and from right

to left for the inverse transform. The values on the left are the 16 input values, that is, the pixel values. On the right are the 16 output DCT coefficients.

The rules of the flow diagram are quite simple. Points where two lines join form a summing point, a minus sign below a line indicates that the value that line carries should be negated, and a value above a line indicates the factor by which the value carried by that line should be multiplied. For example if the x(3) line is partly traced, refer to the highlighted lines in diagram Fig. 4-9, the first summing point is reached at *P* indicating that x(3) should be summed with x(12) as shown in (4.14), the second point *Q* indicates that *P* should then be summed with the x(4) line, as in (4.15), which itself is now the sum of x(4) and x(11). The value at *Q* is then negated and summed with the value carried by the line x(0) which itself is the sum of x(0), x(15), x(7) and x(8). The result of this summation is then scaled by the factor C(4) giving a result at point R as given by (4.16). This process continues until the end of the line is reached at which point the coefficient X(4) would have been fully calculated.

$$P = x(3) + x(12) \tag{4.14}$$

$$\begin{aligned} Q &= P + x(4) + x(11) \\ &= x(3) + x(12) + x(4) + x(11) \end{aligned} \tag{4.15}$$

$$\begin{aligned} R &= C(4) * (x(0) + x(15) + x(7) + x(8) - Q) \\ &= C(4) * (x(0) + x(15) + x(7) + x(8) - x(3) - x(12) - x(4) - x(11)) \end{aligned} \tag{4.16}$$

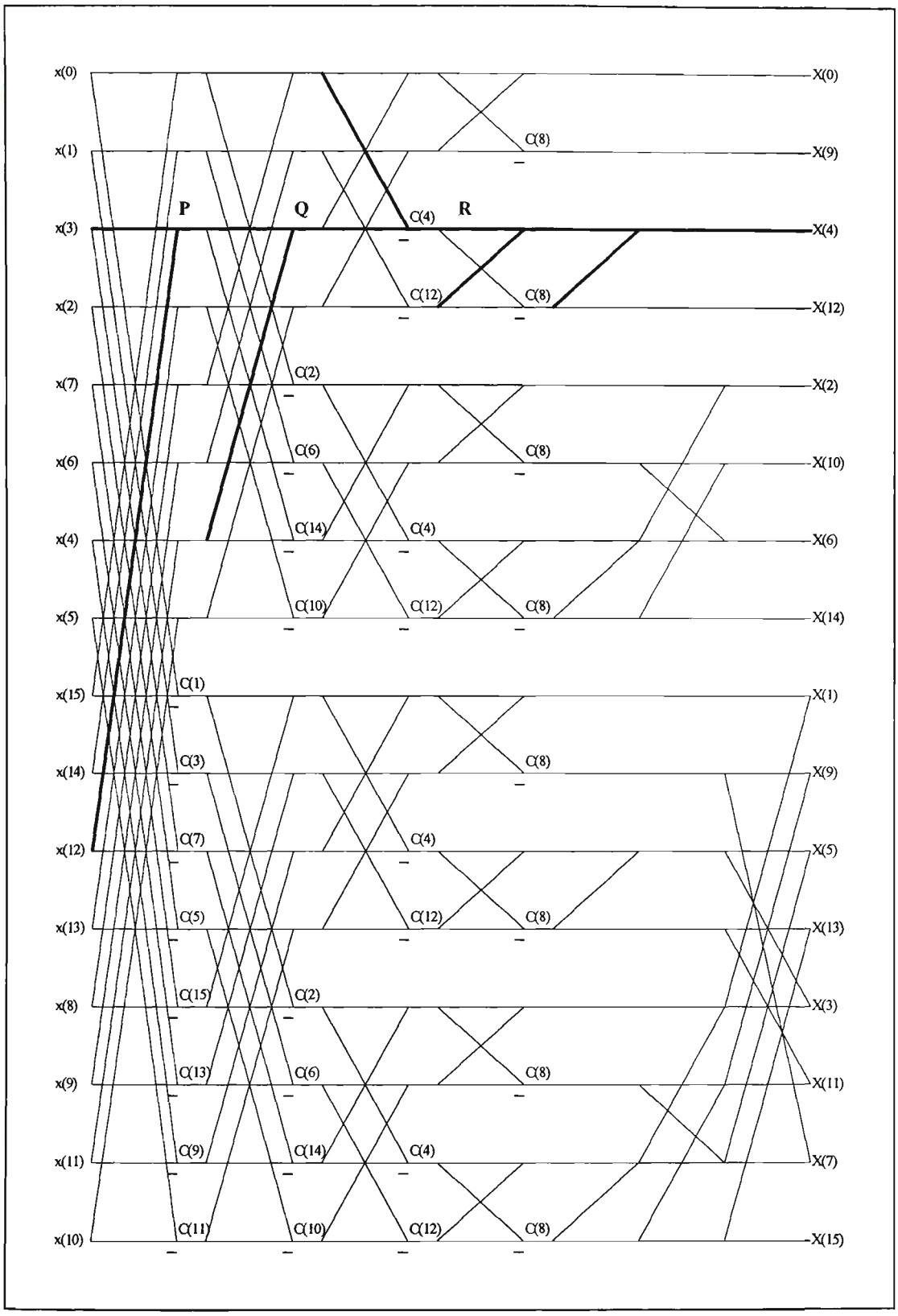


Fig. 4-9 Flowgraph for B.G.Lee's DCT-II algorithm[RAO90].

The software algorithm follows the flow diagram but has output scaling adjustments so that it generates outputs equivalent to those of the SGS-Thomson device used in

the hardware implementation. See section 4.7 for more details on the hardware algorithm. The software also clips the output coefficients so that they do not exceed the maximum possible values for a 12 bit signed integer. Again this is to provide compatibility with that of the SGS-Thomson device.

The software is contained in two files, which may be found in Appendix B. The first listing is the header file *dct.h* and the second is the algorithm source file *dct.c*. As with the hardware device, the software algorithm will limit the block size to one of the following : 4x4, 4x8, 8x4, 8x8, 8x16, 16x8 or 16x16. The forward DCT function prototype is defined in the C programming language as follows :

fdct (intu16 xdct, intu16 ydct, ints16 I[16][16], ints16 O[16][16])

The inputs *xdct* and *ydct* specify the block size to be used for the forward transform. The input (I) and output (O) arrays are 16x16 arrays regardless of the block size. Even though both the input and output are integer values, the internal calculations are performed using floating point arithmetic and so it is advisable that a microprocessor with a floating point co-processor is used for higher performance during calculations (e.g. 80486DX, 80387 etc). The 80486SX used in the test system contained no floating point processor, as such the software algorithm performed rather slowly but nonetheless provided a base upon which simulations could be performed.

4.6.2 The Inverse Transform

The inverse transform follows the same flow diagram as that for the forward transform, but the diagram is read from right to left. The inverse DCT function prototype is defined in the C programming language as follows :

ifdct (intu16 xdct, intu16 ydct, ints16 I[16][16], ints16 O[16][16])

The inputs *xdct* and *ydct* specify the block size to be used for the inverse transform. The input (*I*) and output (*O*) arrays are 16x16 arrays regardless of the block size. As with the forward transform all of the calculations use floating point arithmetic internally.

As there are rounding errors in the forward and inverse transformation cycle it is possible that the output of the inverse transform can fall out of the valid range for a pixel (-128 to 127). For this reason an additional check is made here to ensure that the restored pixel value is within the valid range. If it is not then its value is clipped to the appropriate extreme. This also helps keep the output within the 8 bit restriction of a byte.

4.7 The Hardware Implementation

4.7.1 The SGS-Thomson STV3200

The hardware implementation of the DCT is based upon the SGS-Thomson STV3200 Discrete Cosine Transform. The device itself is capable of transforming up to 15 million pixels per second. It can perform 4x4, 4x8, 8x4, 8x8, 8x16, 16x8 or 16x16

forward or inverse two-dimensional DCTs. The input data is 8-bit two's complement signed integers and the output coefficients are 12-bit two's complement signed integers. Two's complement values are capable of representing both negative and positive numbers, the most significant bit is used to represent the sign and the remaining 11 bits the magnitude. Internally the device implements the forward transform as shown in (4.17).

$$F(u,v) = Round\left[\frac{32}{NM}c^2(u)c^2(v)\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}D(i,j)\cos\left\{\frac{(2i+1)u\pi}{2M}\right\}\cos\left\{\frac{(2j+1)v\pi}{2N}\right\}\right] \tag{4.17}$$

$$\text{where } c^2(u) = \begin{cases} \frac{1}{2} & \text{if } u = 0 \\ 1 & \text{if } u \neq 0 \end{cases}$$

The inverse transform is internally implemented by (4.18).

$$D(i,j) = Round\left[\frac{1}{8}\sum_{u=0}^{N-1}\sum_{v=0}^{M-1}\cos\left\{\frac{(2i+1)u\pi}{2M}\right\}\cos\left\{\frac{(2j+1)v\pi}{2N}\right\}\right] \tag{4.18}$$

4.7.2 The IBM Hardware DCT Interface Description

Using the SGS-Thomson STV 3200, an inexpensive add-on board for an IBM PC was designed and built to allow the DCT and IDCT to be performed in hardware. The schematic diagram for the DCT hardware is shown in Appendix C. The heart of the system is the STV3200CP DCT device, U6. U4 is an eight bit latch that provides general purpose output bits for controlling the block size (BS0..BS2), precision (PR),

type of transform (F/I) and the enable control (EN) of the DCT device. U5 is an 8 bit buffer that allows read-back of the contents of the 8 bit latch.

U2 and U3 are two 8 bit buffers that interface the STV3200CP data ports to the IBM data bus. Note that data bit 9 acts to control the DSYNC signal when writing the image data for a DCT or when reading the data from an IDCT. The DSYNC signal indicates the start of the block being read or written. Similarly, data bit 12 acts to control the FSYNC signal when reading the DCT coefficients from a DCT or when writing the coefficients for an IDCT. The FSYNC signal indicates the start of the coefficient block.

U1 is an EPLD (Electrically Programmable Logic Device) which performs all the necessary decoding to avoid any bus contention or timing problems. The content of the EPLD U1 is shown in Appendix C.

The hardware is mapped into the IBM port map at the addresses shown in Table 4.1. Note that the function of the data bus depends on the state of the F/I signal.

Port Address	Action	Size	DCT/IDCT	Function
0x300	Read	Word	DCT	Read Coefficient
0x300	Write	Word	DCT	Write DCT Input Data
0x300	Read	Word	IDCT	Read IDCT Output Data
0x300	Write	Word	IDCT	Write Coefficient
0x302	Read	Byte	DCT	Read Command Port
0x302	Write	Byte	DCT	Write Command Port
0x302	Read	Byte	IDCT	Read Command Port
0x302	Write	Byte	IDCT	Write Command Port

Table 4.1 Port functions for the IBM DCT interface.

The function of each data bit for the various ports is shown in Table 4.2. Note that the exclamation mark is used to indicate that the signal is active low, that is, when this signal is zero then it is activated.

Bit Number	Coefficient Data	Image Data	Command Port
D0	Coefficient D0	Data D0	Block Size BS2
D1	Coefficient D1	Data D1	Block Size BS1
D2	Coefficient D2	Data D2	Block Size BS0
D3	Coefficient D3	Data D3	!PR
D4	Coefficient D4	Data D4	!F/I
D5	Coefficient D5	Data D5	!Enable
D6	Coefficient D6	Data D6	unused
D7	Coefficient D7	Data D7	unused
D8	Coefficient D8	Data D8	unused
D9	Coefficient D9	DSYNC	unused
D10	Coefficient D10	unused	unused
D11	Coefficient D11	unused	unused
D12	FSYNC	unused	unused

Table 4.2 Bit definitions for all ports of the IBM DCT interface.

4.7.3 The Driver For the Interface

In order to support the hardware interface it was necessary to design and implement drivers to control the hardware. The driver for the IBM DCT Interface is shown in appendix C. It consists of two C source files, the hardware DCT header file (*hdct.h*) and the hardware DCT driver (*hdct.c*). It provides functions to initialise the DCT device, perform a forward DCT and perform an IDCT. These will be described in more detail in the following sections.

4.7.3.1 Driver Initialisation

Before using the forward or inverse transform, it is necessary to initialise the STV3200 DCT transform chip. This is achieved by calling the function *init_DCT()*. The function sets up the control registers of the STV3200 and sends a train of 130 dummy values to clear out the devices internal buffer and to prepare it for an incoming block of data.

4.7.3.2 The Hardware Forward DCT

The forward DCT function prototype is defined in the C programming language as follows:

*void fdct (intu16 xdct, intu16 ydct, ints8 far *source, ints16 far *destination)*

The inputs *xdct* and *ydct* specify the block size to be used for the forward DCT transform. The input source is a pointer to the input data. Note that this input differs slightly from the software DCT in that the input consists of 8 bit signed data values compared to the 16 bit signed data values of the software DCT. Also the size of the input source is dependent on the DCT block size whereas the software DCT had a constant data block size of 16x16 regardless of the DCT block size. The destination is a pointer to a block of signed 16 bit integers. The size of the block is again dependent on the DCT block size used.

The input data is converted to nine bit signed integer values and clocked into the STV3200 device. Following the data transfer a further 131 clock pulses must be sent before the STV3200 begins to output the DCT results. The DCT results are then

clocked out of the STV3200, sign extended to 16 bit signed integers and stored in the destination block.

4.7.3.3 The Hardware Inverse DCT

The inverse DCT function prototype is defined in the C programming language as follows:

*void fidct (intu16 xdct, intu16 ydct, ints16 far *source, ints8 far *destination)*

The inputs *xdct* and *ydct* specify the block size to be used for the inverse DCT transform. The input source is a pointer to the input DCT coefficients. This input differs slightly to the software inverse DCT in that the size of the input source is dependent on the inverse DCT block size, whereas the software inverse DCT had a constant data block size of 16x16 regardless of the inverse DCT block size. The destination is a pointer to the output block of signed 8 bit integers. The size of the block dependent on the inverse DCT block size used.

The input DCT coefficients are converted back to 12 bit signed integer values and clocked into the STV3200 device. Following the data transfer a further 131 clock pulses must be sent before the STV3200 begins to output the inverse DCT results. The inverse DCT results are then clocked out of the STV3200, clipped to fit within an 8 bit signed integer and stored in the destination block. The STV3200 device should clip the output values itself though this operation does not seem to work correctly and had to be performed in software.

4.7.3.4 Problems Associated With the Hardware DCT

There are two major problems with the performance of the hardware DCT. The first is that all the data must be continually clocked in and out of the STV3200 by software. This tends to be quite a tedious and time consuming task for a Complex Instruction Set Computer (CISC) such as the Intel 80486. This is further slowed down by the reduced bus speed of the IBM AT standard bus where the 33Mhz 80486 is slowed down to a rate of 8Mhz. The best solution for this is to design a stand-alone system based around a Reduced Instruction Set Computer (RISC) such as the Intel i860 to improve throughput. Another improvement would be to have Direct Memory Access (DMA) capabilities to avoid transferring the data via software. Unfortunately both of these would add considerably to the cost of the project in terms of hardware costs and in terms of time. Since the ideal rate of the transforms from the data sheets of the STV3200 is already known, it is not really necessary to construct a more complex device. It is more important to examine the characteristics of this device so as to provide an improved compression algorithm.

Another problem associated with the hardware DCT is that there is a great deal of overhead required if the blocks are transformed individually: it is more efficient to bulk transform several blocks at one time. This problem is quite easily overcome in software by combining a group of blocks together and then transforming them. This is exploited later in the development of the OptIC algorithm, see section 5 for a description of the modified DCT algorithm.

4.8 Post-Processing

The post-processing stage after the inverse DCT transform, consists of two stages: bias removal, and reordering.

4.8.1 Bias Removal

This stage removes the bias that was added prior to the forward transform. It simply adds an offset of 128 to the output data.

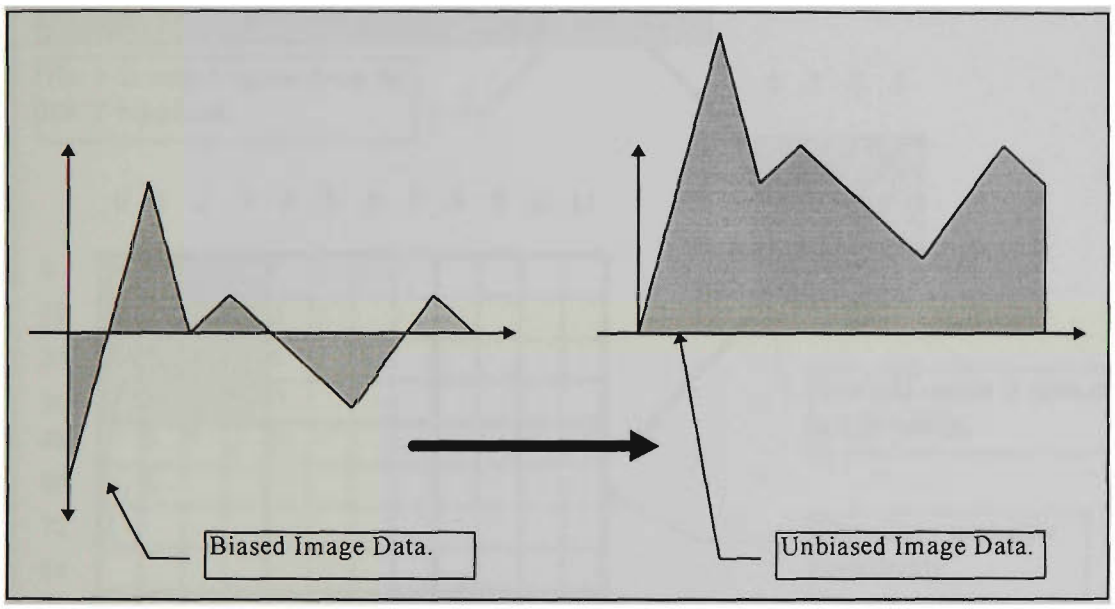


Fig. 4-10 Image data bias removal.

4.8.2 Re-Ordering

Once the block has been fully restored it is then re-ordered back to its previous form, i.e. as it was in the original image. This involves taking the output of the IDCT, which is an *NxN* element vector in the case of the hardware *NxN* IDCT algorithm, and converting it to an *NxN* 2-D matrix (this is the form of the data after a software *NxN* IDCT). This 2-D matrix is then inserted back into the larger image array. This

procedure is illustrated in Fig. 4-11 where after completion of a 4x4 hardware IDCT the 16 element output vector must be converted to a 2-D matrix, the value in each element box represents some sample data whereas the value along the side of the array represents the address of that element. This 2-D matrix is then stored in the 12x12 pixel image array. Note that the memory locations of the IDCT output and those of the image do not correspond, that is why there is a need for the re-ordering.

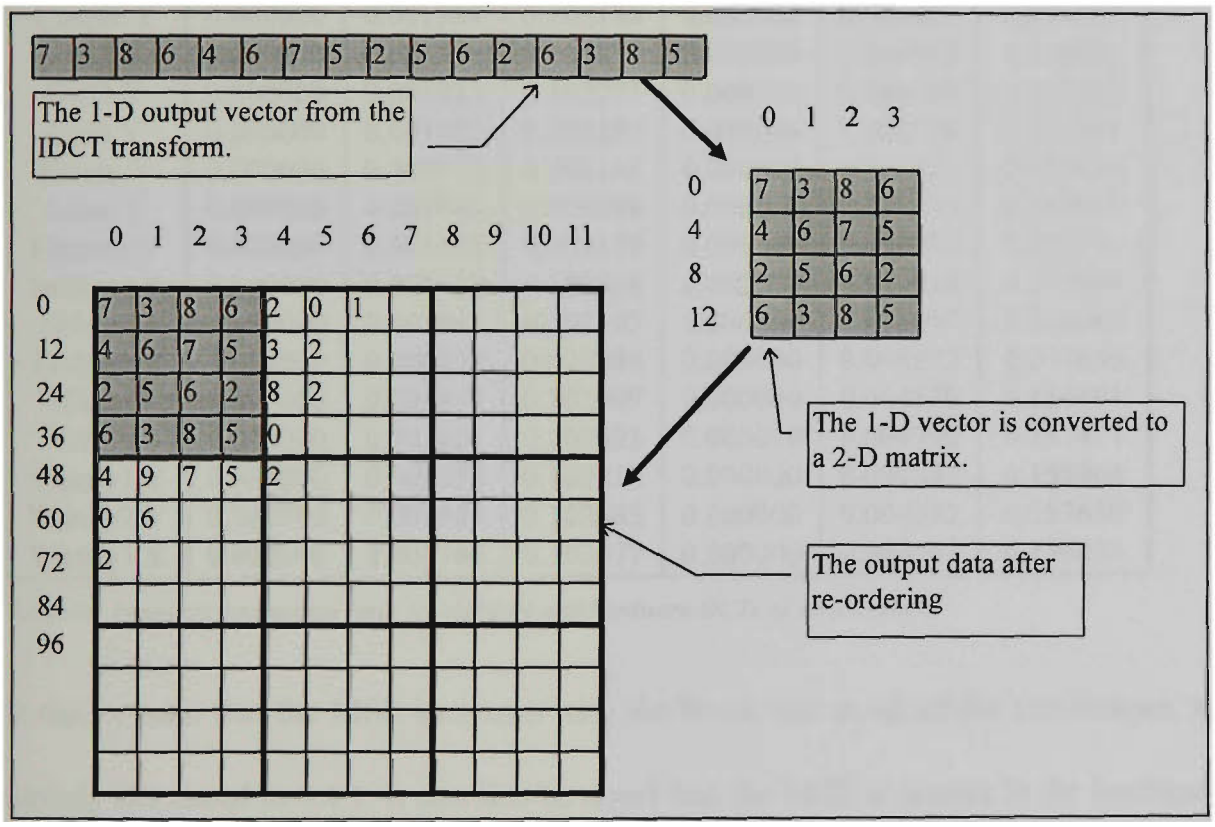


Fig. 4-11 Illustration of the data re-ordering procedure

4.9 Results

4.9.1 Reconstruction Error

Both the hardware and the software DCT algorithms were tested to compare the errors introduced after reconstruction of the intensity images in the test set. The tests were

performed using block sizes of 4x4, 8x8 and 16x16. The error is measured by calculating the MSE between the original image and the reconstructed image. The results of these tests are shown in Table 4.3.

	Software DCT Block Size			Hardware DCT Block Size		
Image File	4x4	8x8	16x16	4x4	8x8	16x16
Airplane.Y	0.000000	0.001820	0.103535	0.000000	0.004837	0.157028
Baboon.Y	0.000000	0.002140	0.103390	0.000004	0.005238	0.157040
Beans1.Y	0.000000	0.001953	0.101593	0.000000	0.004745	0.158203
Beans2.Y	0.000000	0.001816	0.101547	0.000000	0.004776	0.156799
Couple.Y	0.000000	0.001984	0.102188	0.000000	0.004913	0.155807
Girl1.Y	0.000000	0.001938	0.104858	0.000000	0.004913	0.156052
Girl2.Y	0.000000	0.001923	0.103271	0.000000	0.004410	0.157227
Girl3.Y	0.000000	0.001892	0.101196	0.000000	0.004578	0.156555
House.Y	0.000000	0.002075	0.102142	0.000000	0.004822	0.158615
Lena.Y	0.000000	0.001945	0.104008	0.000000	0.004833	0.157051
Peppers.Y	0.000000	0.001938	0.104179	0.000000	0.004459	0.156776
Sailboat.Y	0.000000	0.001888	0.102558	0.000000	0.004910	0.157990
Splash.Y	0.000000	0.001961	0.103107	0.000000	0.004807	0.156040
Testpatt.Y	0.000000	0.000557	0.028084	0.000000	0.008522	0.077656
Tiffany.Y	0.000000	0.001877	0.102909	0.000000	0.004570	0.154491
Tree.Y	0.000000	0.001801	0.102692	0.000000	0.004990	0.157471
Wendy1.Y	0.000000	0.001774	0.102810	0.000000	0.004841	0.155964
Wendy2.Y	0.000000	0.001884	0.102665	0.000000	0.004852	0.157650
Wendy3.Y	0.000000	0.001766	0.103077	0.000000	0.004761	0.156033

Table 4.3 Image reconstruction error for software and hardware DCTs of various sizes.

It can be seen that the MSE increases with the block size in all of the test images, as already discussed in 4.4.1. It can also be noted that the MSE is greater in the hardware transform than in the software transform. This is because the hardware transform uses integer arithmetic in its internal calculations and so rounding errors are more evident. The software algorithm uses high precision floating point arithmetic to perform its calculations and is so more accurate and less prone to rounding errors.

4.9.2 Timing Benchmarks

The second test performed on the DCT algorithms is to examine their computational speed. These tests were done with both the software and hardware algorithm on the forward and inverse transform. The tests were also performed using block sizes of 4x4, 8x8 and 16x16. Ideal results were also calculated for the STV3200 from the data sheets to compare the results of the hardware algorithm with that of the ideal maximum throughput. The results gained from the tests are summarised in Table 4.4.

	Software Algorithm		Hardware Algorithm		Hardware (Ideal)	
Block Size	DCT	IDCT	DCT	IDCT	DCT	IDCT
4x4	12.91mS	13.35mS	341 μ S	341 μ S	1.07 μ S	1.067 μ S
8x8	67.34mS	72.23mS	621 μ S	615 μ S	4.27 μ S	4.267 μ S
16x16	335.1mS	376.8mS	1.719mS	1.697mS	17.1 μ S	17.07 μ S

Table 4.4 Time required to complete the specified cosine transform with block sizes of 4x4, 8x8 and 16x16.

A number of characteristics can be seen from these results. Firstly, the software algorithm is a great deal slower than the hardware algorithm, particularly for larger block sizes. For the 16x16 DCT the software algorithm is almost 200 times slower. Secondly, it can be seen that a doubling in block size causes an increase in processing time of a factor of over four in software, but the corresponding increase in the hardware processing is less than three. This is because of the larger overheads in the hardware algorithm when dealing with smaller blocks, see section 4.7.3.4. In order to transform a 16x16 block using DCTs with a block size of 8x8, four 8x8 transforms would be required. Therefore if the processing time for a 16x16 transform is less than four times that for an 8x8 transform, it would be more advantageous to use the 16x16

transform. Thus for the hardware algorithm, it is faster to use a larger block size than it is to use a smaller one. This is not, however, true with the software algorithm since the 16x16 transform is more than four times slower than the 8x8 transform.

Another interesting result is that the STV3200 is not used anywhere close to its potential abilities. Only a fraction of its power is tapped because of the poor architecture of the Intel 80486 with respect to Input/Output transfers as discussed earlier, see section 4.7.3.4.

4.9.3 Entropy Effects

The final test performed was to measure the entropy effects on the image data after the transformation. The entropy of the original image was measured and then the image was transformed using both the software and hardware algorithm with block sizes of 4x4, 8x8 and 16x16. The results of these tests are shown in Table 4.5.

From the results a number of patterns can be seen. Firstly, a DCT with a block size of 4x4 does not really provide any benefits to the entropy of the image and in some cases increases it. A block size of 8x8 does, however, provide great improvements in entropy, improving in some cases by over two bits. A block size of 16x16 provides even greater improvements, over 3 bits in some cases. Improvements in entropy occurred in all but one image - *Testpatt.Y*. The reasons for this were discussed in 4.4.1.

		Software DCT Block Size			Hardware DCT Block Size		
Image	Original	4x4	8x8	16x16	4x4	8x8	16x16
Airplane.Y	6.70576	6.59883	5.46118	4.49233	6.59815	5.46163	4.49679
Baboon.Y	7.35814	8.23258	7.22579	6.24640	8.23203	7.22536	6.24667
Beans1.Y	5.72470	5.04377	3.90887	2.98390	5.04333	3.91436	3.02061
Beans2.Y	6.24254	5.57542	4.47563	3.61587	5.57615	4.47897	3.64295
Couple.Y	6.42737	6.49216	5.37668	4.50975	6.49188	5.37726	4.51369
Girl1.Y	7.05377	6.69353	5.51193	4.55283	6.69317	5.51229	4.55489
Girl2.Y	5.60747	5.63987	4.59549	3.76962	5.63783	4.59681	3.77790
Girl3.Y	7.26169	6.38091	5.25496	4.30994	6.38057	5.25522	4.31520
House.Y	6.50448	6.47716	5.34951	4.37570	6.47580	5.35076	4.38083
Lena.Y	7.44776	6.73084	5.53930	4.53030	6.73056	5.53951	4.53318
Peppers.Y	7.59430	6.96417	5.78915	4.80178	6.96405	5.78887	4.80349
Sailboat.Y	7.48579	7.52480	6.39000	5.40044	7.52451	6.38964	5.40114
Splash.Y	7.25848	6.21584	5.03779	4.05277	6.21559	5.03766	4.05722
Testpatt.Y	0.98935	3.98828	6.16478	6.20373	4.04839	6.17924	6.20320
Tiffany.Y	6.60048	6.68245	5.50143	4.51656	6.68241	5.50145	4.51991
Tree.Y	7.31444	7.66236	6.67389	5.80216	7.66187	6.67400	5.80349
Wendy1.Y	6.82276	5.69242	4.37729	3.33852	5.69006	4.38009	3.35043
Wendy2.Y	7.09684	6.64787	5.37983	4.32710	6.64699	5.38041	4.33512
Wendy3.Y	7.82136	6.53512	5.26977	4.22150	6.53454	5.27025	4.22606

Table 4.5 Entropy of the image before and after using the software and hardware DCT.

4.10 Conclusion to the Chapter

Several important conclusions may be drawn from the results obtained in this chapter. The first is that the hardware implementation of the DCT/IDCT algorithm is a great deal faster than the software algorithm, up to 200 times in some cases. The results also indicate that for the hardware DCT/IDCT algorithm, the processing time for a set image size decreases with a larger block size, that is, a block size of 16x16 is favourable.

The entropy effect results also tend to indicate that the entropy improves greatly with the larger block sizes. With the 16x16 block size maximum reduction of entropy was obtained.

There is only one disadvantage in shifting towards a larger block size. The reconstruction error increases because of increased rounding errors. However, when the reconstruction error was compared with the entropy gained it was found that for reconstruction errors of greater than two it was still advantageous to use a larger block size, see section 4.4.1 for further details.

In summing up, it was found that the most suitable DCT/IDCT implementation for use in the OptIC algorithm would be the hardware implementation with a block size of 16x16.

5. The Quantiser

5.1 Introduction

The quantiser forms the second stage of the OptIC algorithm. It has the effect of reducing the entropy of the image data by reducing the resolution of the DCT coefficients. This is done by scaling down the coefficient magnitudes by a given amount and removing any fractional portions. At dequantisation the coefficient is scaled up by the same amount. The greater the scaling the greater the resultant error in the reconstructed coefficient.

The different coefficients are quantised by differing extents depending upon their importance. This is primarily determined by the degree of visual error introduced after reconstruction. It is also determined to a lesser extent by the degree of error introduced with respect to the reduction of entropy after quantisation. This circular definition implies that an iterative process is required to determine an optimum quantiser for the algorithm described in this thesis. This process defined in more detail in section 5.6.

For this research it was stated that a low error rate is desirable. In particular this error rate must fall below that where it is perceivable by the human eye. Note that the Mean Square Error (MSE) does not give a true indication of the subjective error assessment but does, however, give an estimate when dealing with a large test space. The

difference between the MSE and the subjective error assessment becomes greater as more HVS properties are incorporated into the algorithm.

Before the quantiser was designed, it was important to make a further study of the DCT coefficient properties and to incorporate their effects as perceived by the Human Visual System.

5.2 The DCT Coefficient Properties

The properties of the DCT coefficients are now looked at in greater detail with particular emphasis on their numerical and functional aspects.

5.2.1 Numerical Properties

The first of the numerical properties to be considered was the statistical qualities of the transformed images. An observation was made of the frequency of occurrences of a particular symbol, where a symbol is defined as a member of the possible set of values which the coefficients may take. As the coefficients are 12 bit signed values, the symbol set includes all values from -2048 to 2047 inclusive. The frequency is measured by transforming all the intensity images in the test set using the DCT transform described in Section 4.7, and counting the occurrence of each unique symbol. The images are then compared with each other to find the minimum, maximum and average frequency of each symbol. It should be noted that the frequencies of the smaller 256x256 images were appropriately scaled by a factor of

four so that they may be accurately compared with the results obtained with the 512x512 images. The results of this are shown in the graph in Fig. 5-1.

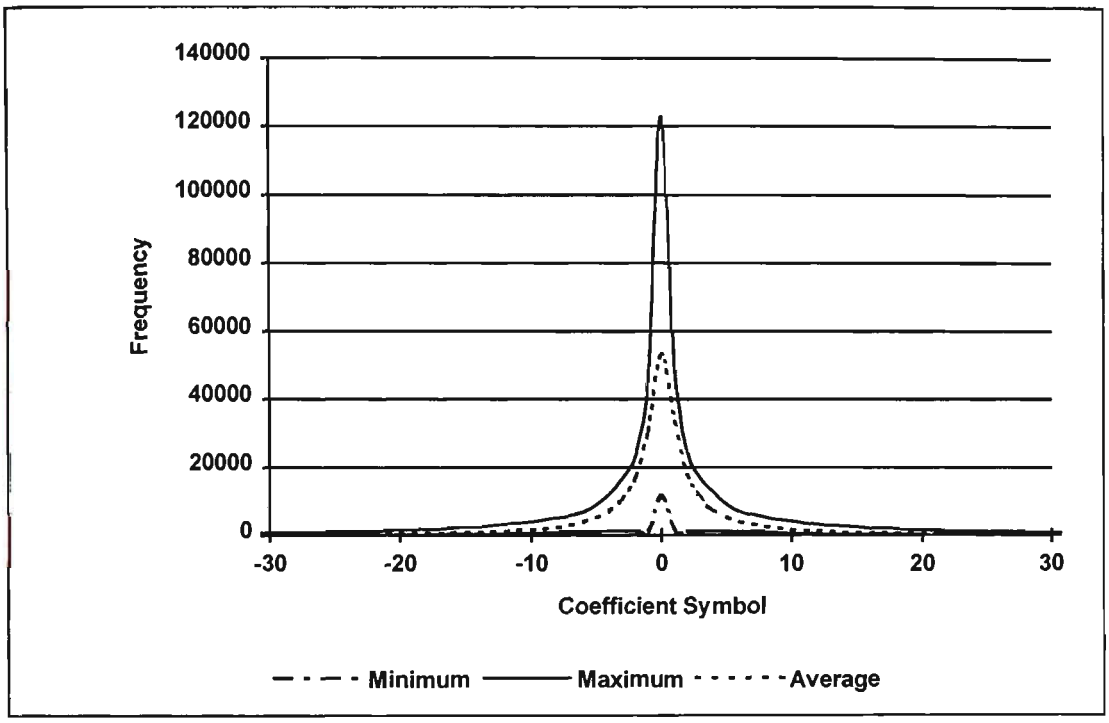


Fig. 5-1 Frequency of DCT coefficient symbols

The frequency data shown in Fig. 5-1 is very important in determining how the data will respond to a statistical coder. The more concentrated the symbols are within a particular range the lower the entropy of the data and so it is more compressible. Only coefficient values ranging from -30 to 30 are shown in the graph since only these values have significant frequencies. Outside this range the frequency values drop dramatically. It can be seen that the peak frequency occurs with the zero coefficient symbol (which represents between one fifth to one half of the transformed image data). Furthermore a great majority of the symbol values lie within the range of -10 to 10.

The second analysis made on the DCT coefficients is to make an observation of what the average, peak negative and positive magnitudes are for each coefficient after transformation. To obtain the required data for this test, the intensity images in the test set were transformed using the DCT transform (4.14). The minimum and maximum magnitudes of each coefficient were then recorded for each image. Finally, the values obtained for each image were compared with those of the other images to determine the overall most negative (Fig. 5-2), average negative (Fig. 5-3), most positive (Fig. 5-4) and average positive magnitudes (Fig. 5-5) for each coefficient in the DCT. The coefficients are, in fact, discrete values, valid only at the intersection of the horizontal and vertical lines on the diagrams. Figures 5-2 and 5-4 are included for comparison purposes, and not to imply any continuous nature of the magnitude of the coefficients.

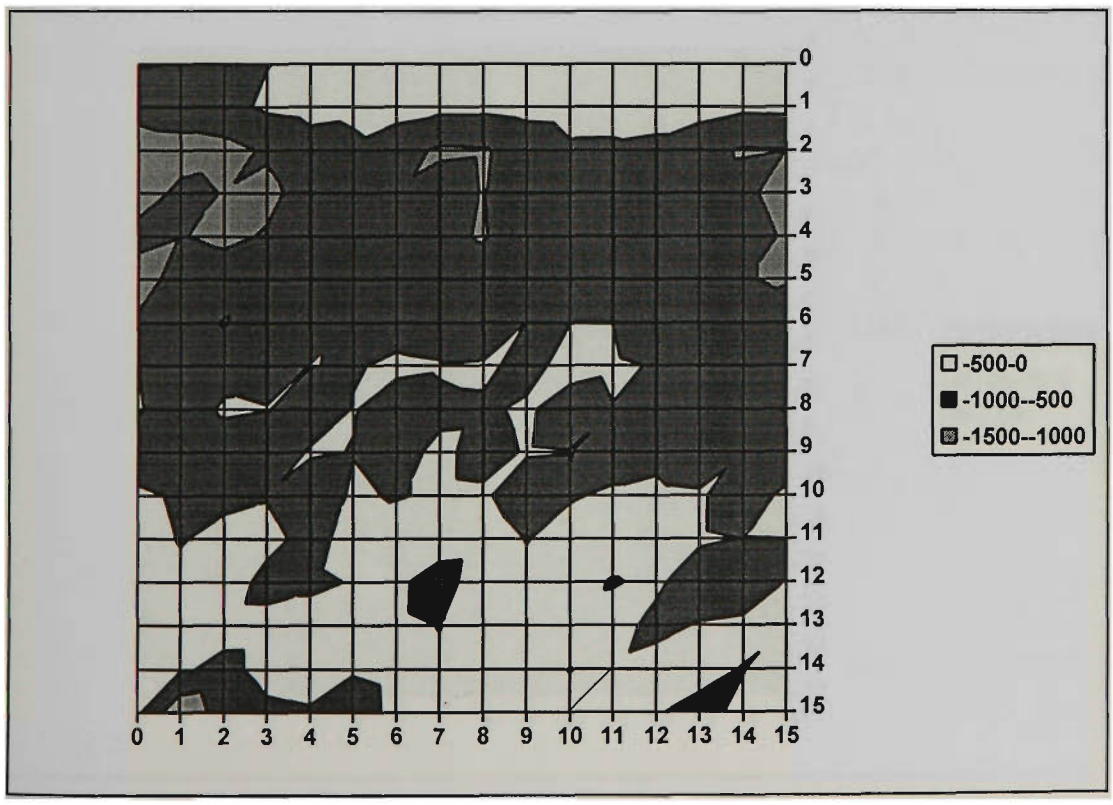


Fig. 5-2 Most negative magnitudes of DCT coefficients

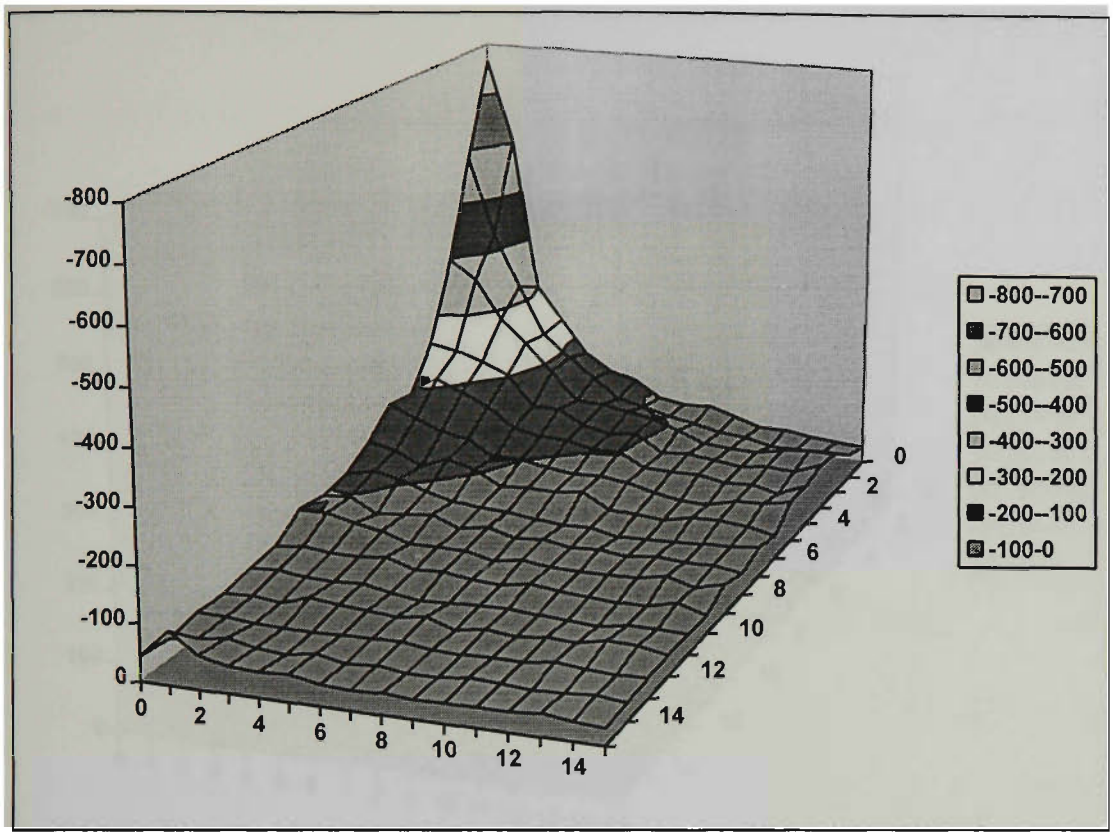


Fig. 5-3 Average negative magnitudes of DCT coefficients

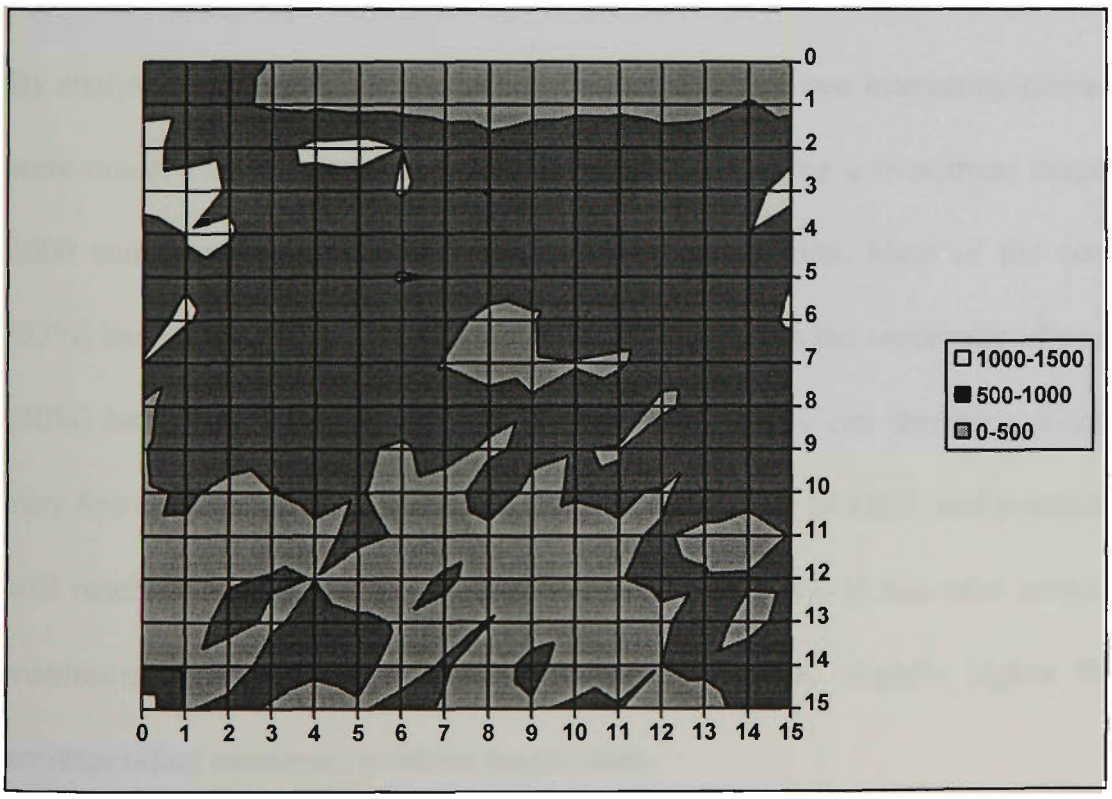


Fig. 5-4 Most positive magnitudes of DCT coefficients

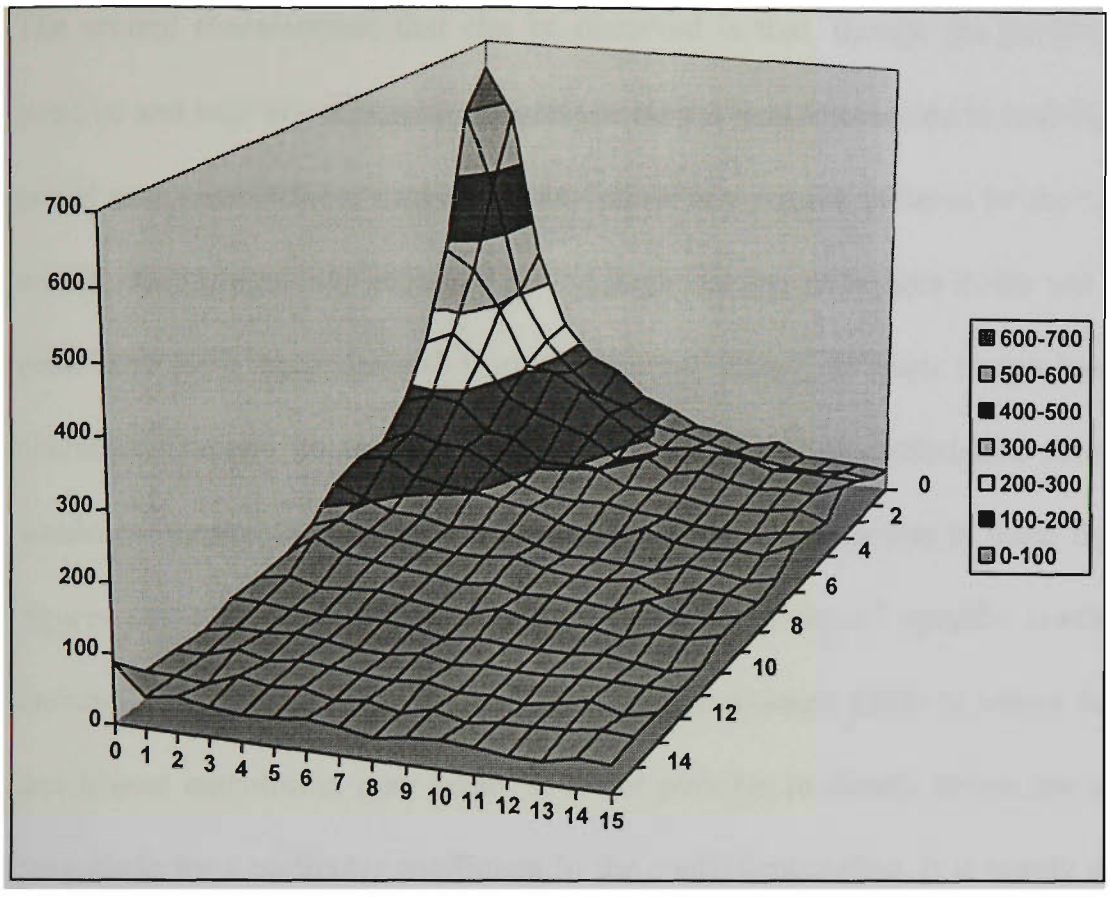


Fig. 5-5 Average positive magnitudes for DCT coefficients

By analysing the results shown in Fig. 5-2 and Fig. 5-4 two interesting characteristics were noted. Firstly, the number of coefficients exceeding a maximum magnitude of 1000 composes approximately 7% of all the coefficients. Most of the coefficients (53%) have magnitudes in the range of 500 to 1000 with the remainder of coefficients (40%) having magnitudes in the range of 0 to 500. We can therefore conclude that very few coefficients will exceed an absolute magnitude of 1000, and practically none will reach the possible maximum of magnitude of 2047. It was also noted that the maximum negative magnitudes of the coefficients are slightly higher than their corresponding maximum positive magnitudes.

The second characteristic that can be observed is that, though the positions of the positive and negative maximum magnitude ranges tend to coincide in both figures, the actual positions of these ranges do not follow any regular patterns in the coefficient matrix. This irregularity is caused by the large number of images in the test set. Each peak may have been derived from a different image, as each image has its own characteristics and its own peaks associated with these characteristics. As such, one would not expect there to be any correlation between the points in these figures, the figures can only give information about the magnitudes of specific coefficients in isolation. Fig. 5-2 and Fig. 5-4 only give an approximate guide to where the highest and lowest magnitudes may occur. It is not possible to clearly define the maximum magnitude for a particular coefficient in the coefficient matrix. It is purely dependent on the test set of images used.

An analysis of the graphs in Fig. 5-3 and Fig. 5-5 verify the characteristic found previously that the negative magnitudes are on average greater than those of the positive magnitudes. Also, unlike the graphs showing the most negative and positive magnitudes, the average magnitudes follow a trend. The different ranges of average magnitudes are formed as concentric sectors about the DCT coefficient (0,0). The values of the average magnitude reduce sharply as we move away from the DC coefficient (0,0). Those coefficients that are more than 7 units in distance from the DC coefficient (0,0) have average magnitudes of less than 100 (The effect of quantisation on these characteristics is discussed further in section 5.3).

5.2.2 Functional Properties

The functional properties of the DCT coefficients describe the contribution of each coefficient in rebuilding the transformed image and, in particular, how this relates to the HVS. In section 4.4.2.1, it was noted that the most important property in the HVS was that we are most sensitive to lower frequency changes of intensity. That is, smooth transitions from one intensity to another over a large area.

Another important property is that for higher frequency coefficients we are able to notice changes at lower intensity levels more than at higher levels. In effect the eye follows a non-linear response to changes in these coefficients. This can be demonstrated by the quantisation of any signal. For low amplitude signals the effect of the quantisation will appear greater than for higher amplitude signals. This is simply because the percentage change in magnitude caused by the quantisation is much greater for smaller magnitudes. This is particularly evident in high frequency data. With low amplitude signals, the intensity of each symbol can be easily measured by the human eye with respect to neighbouring pixels. As all the pixels will have similar intensities, each pixel can be used as a reference to measure the intensity of an adjacent pixel. Any slight changes caused by quantisation will, in this situation, be easily perceived. However, as the amplitude and frequency of the signals increases, the human eye loses its reference by which the intensities of adjacent pixels may be measured. The sudden light-dark changes thus make it difficult to visualise the

changes caused by quantisation [TZO84]. This property is the justification for the use of a non-uniform quantisation [RAO90].

The usefulness of a non-uniform quantiser is more easily shown by the example in Fig. 5-6 and Fig. 5-7. Fig. 5-6 shows an image segment before quantisation (left) and after quantisation by a factor of 16 (right). The images are composed of vertical bars each of which has a random low intensity. If the two images are compared visually it is easy to see that images are not identical. The columns have clearly changed in intensity. It can be concluded that the level of quantisation in this situation was too high.

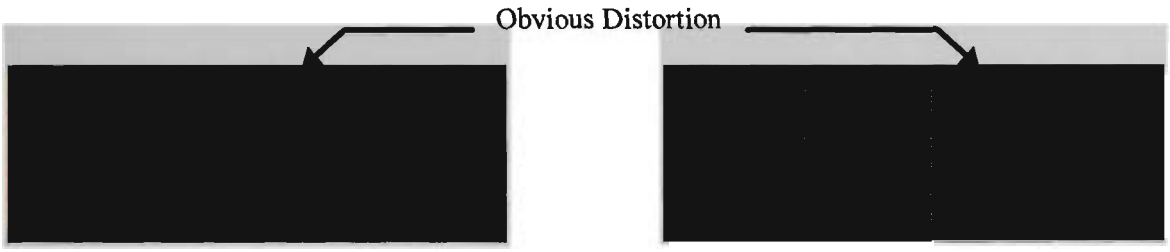


Fig. 5-6 An example of quantisation of a signal with low intensity data.

Fig. 5-7 contains the same intensities as those in Fig. 5-6 but with a high frequency signal superimposed on top of it. After quantisation by a factor of 16 the errors caused by the quantisation are still the same as those in Fig. 5-6 but are no longer as apparent. It can thus be concluded that with high magnitude, high frequency data added to the signal it is possible to quantise by a greater degree without degrading image quality.

Fig. 5-7 An example of quantisation of a signal with high magnitude and high frequency content.

The non-linear quantiser can take into consideration such effects so as to maximise the possible level of quantisation without compromising image quality.

5.3 Quantisation Effects On DCT Coefficients

It is now important to examine the effect of quantisation on the coefficients. It was determined in section 4.4.2.3 that the entropy improvements provided by each particular coefficient were not dramatically varied throughout the different coefficients. For this reason, and to satisfy the requirement of minimal error in reconstruction, the error characteristics form the basis for determining the method of quantisation.

5.3.1 Error Effects On the Reconstructed Image Data

The quantiser introduces a second level of error into the algorithm. The first was introduced by the DCT transformation and reconstruction of the image. It can be seen, by referring to Table 4.3, that the forward-inverse hardware DCT introduces a maximum MSE of approximately 0.16 with a block size of 16x16. This can now be compared to the MSE obtained when the coefficients are scaled down by factors of 1 to 16. To make this comparison the images were all transformed using a 16x16 DCT and each coefficient individually scaled down by factors in the range of 2 to 16 and

then restored and compared with the original image to determine the amount of MSE introduced. This resulted in an MSE measurement for each of the 256 coefficients (16x16) when scaled by factors of 2 to 16 (15 cases) for each of the 19 intensity images in the test set. Since special emphasis is placed on minimising error, the 19 images were compared taking the largest occurrence of MSE among them. The graphs shown in Fig. 5-8, Fig. 5-9, Fig. 5-10 and Fig. 5-11 show the maximum MSE when each coefficient was individually scaled by factors of 2, 4, 8 and 16 respectively for the images in the test set.

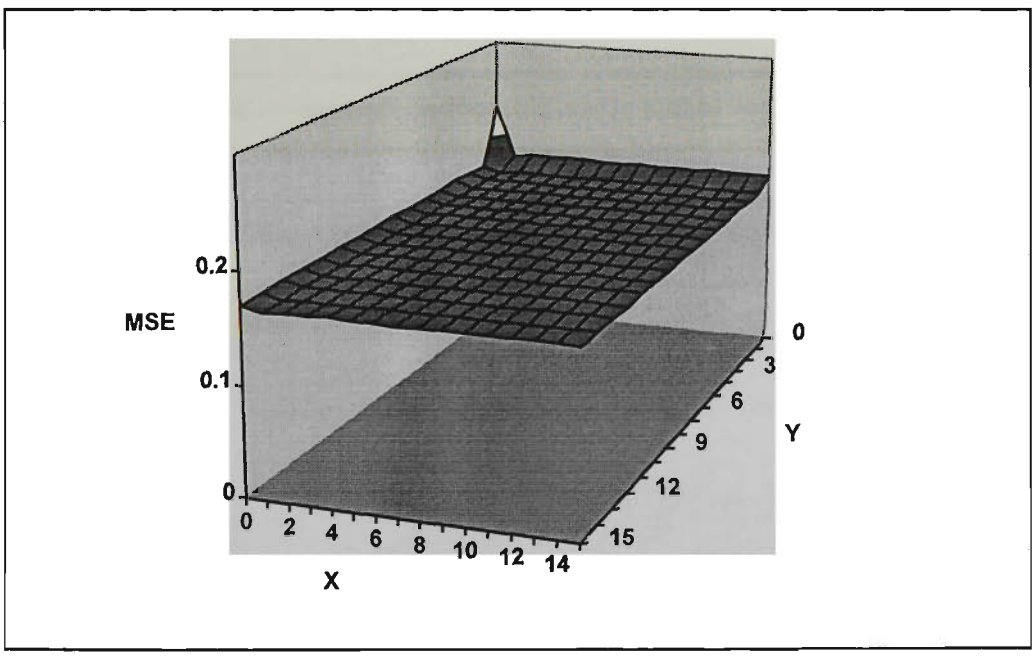


Fig. 5-8 Maximum MSE obtained with coefficients scaled by a factor of 2.

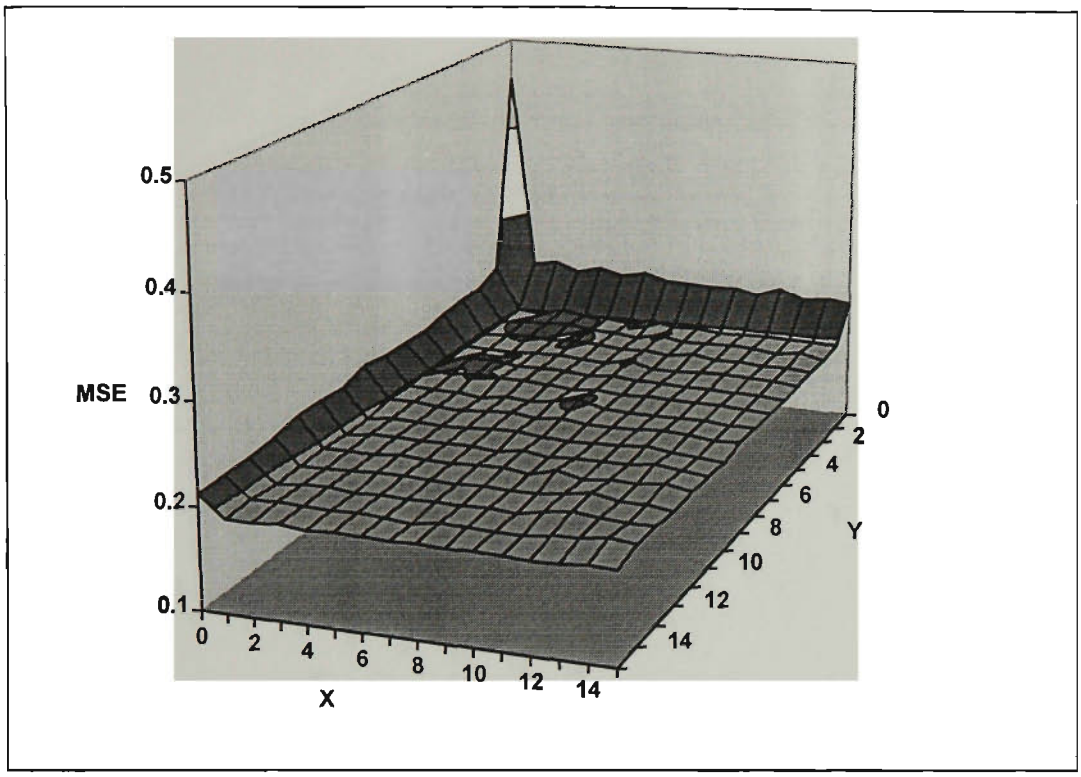


Fig. 5-9 Maximum MSE obtained with coefficients scaled by a factor of 4.

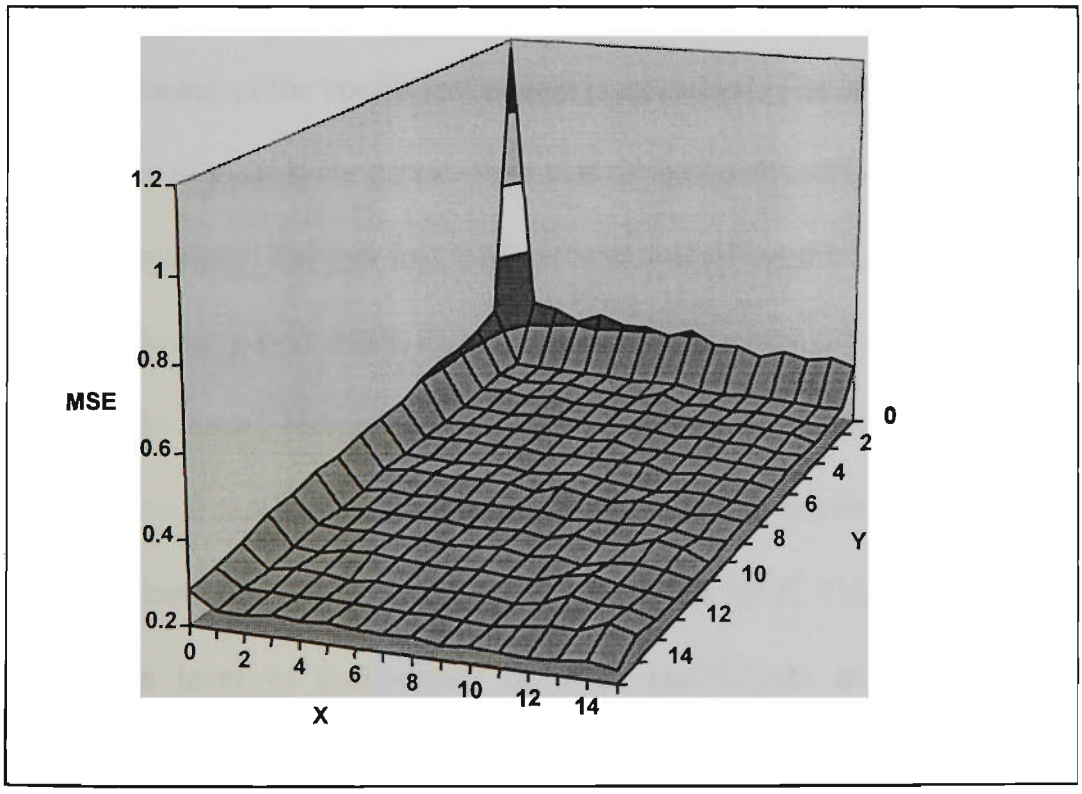


Fig. 5-10 Maximum MSE obtained with coefficients scaled by a factor of 8.

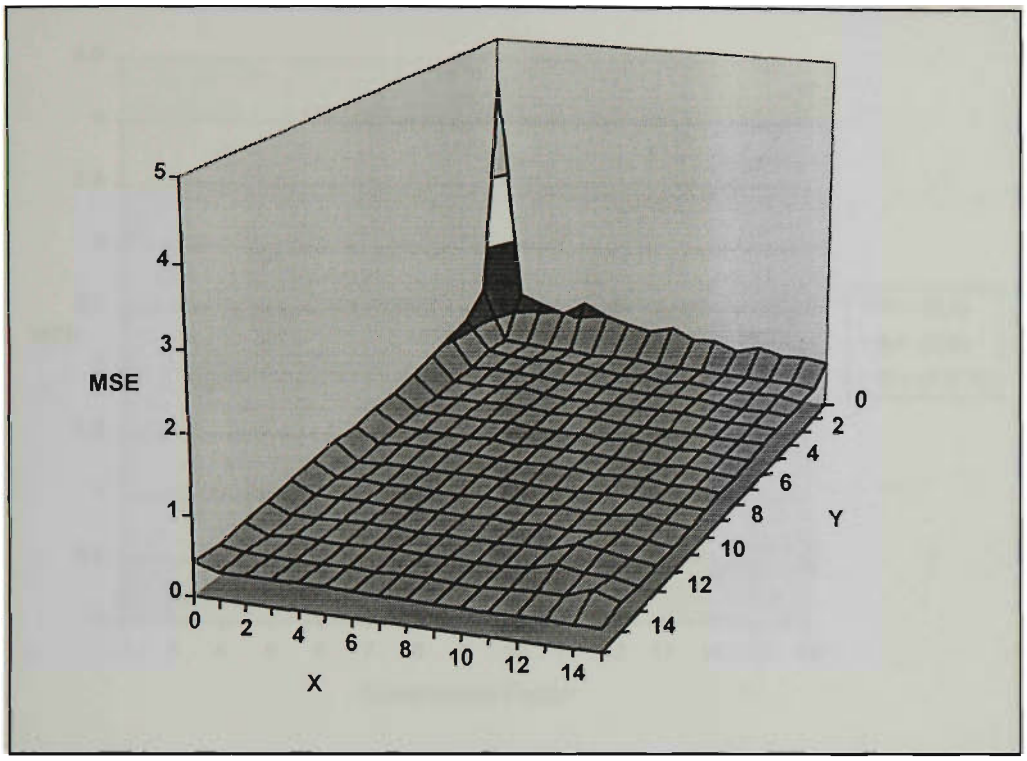


Fig. 5-11 Maximum MSE obtained with coefficients scaled by a factor of 16.

The most distinctive feature of these figures is that the greatest MSE occurs when the DC component of the coefficient matrix is quantised. The MSE in this situation is up to an order of magnitude greater than that of the other coefficients when quantised by the same quantity. For low levels of quantisation all but the DC coefficient have errors of approximately 0.17 MSE, this is only marginally higher than the base level of error obtained by errors internal to the DCT algorithm itself. Even with higher levels of quantisation it can be seen that the coefficients that do not lie on the (0,x) or (x,0) axis of coefficients contribute very little to the overall MSE. This indicates that there is a reasonable level of redundancy in these coefficients and little changes in the information content of the image when these are quantised.

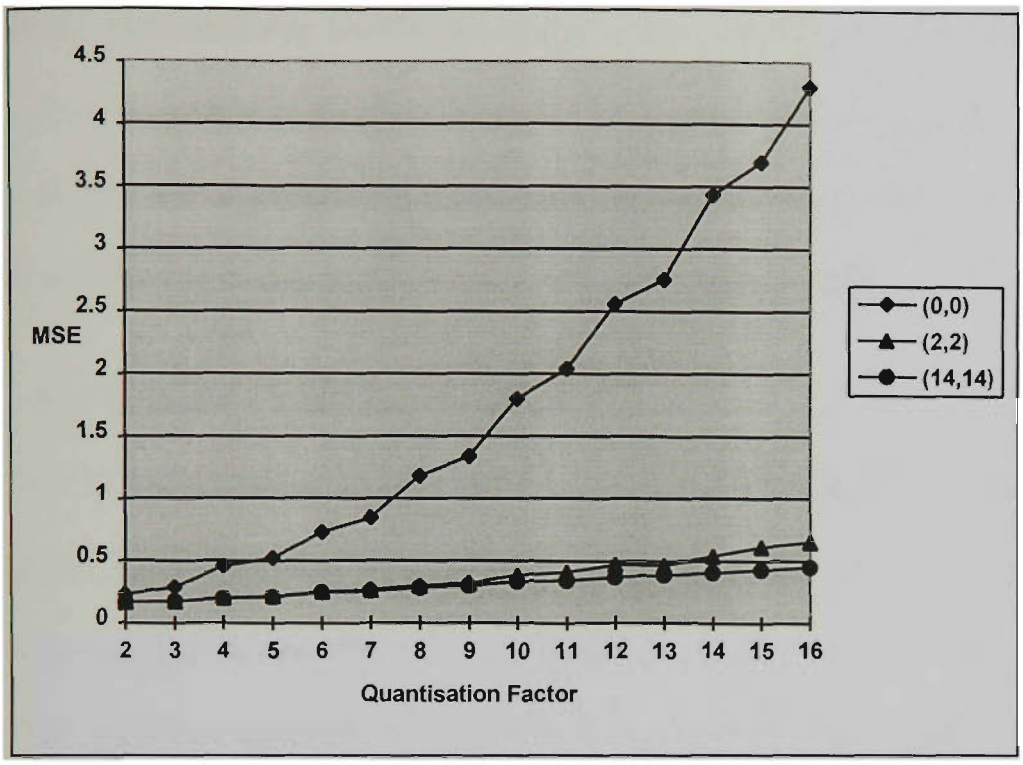


Fig. 5-12 Maximum MSE for quantisation of coefficients along diagonal of coefficient matrix.

The effect of the different levels of quantisation on the MSE can be seen more clearly in Fig. 5-12. Where a graph of the MSE versus the quantisation factor is plotted for several coefficients that appear along the diagonal of the coefficient matrix. Once again the high impact of quantising the DC coefficient on the MSE can be seen clearly. The second feature that may be noticed is how sharply the MSE drops as we quantise coefficients further from the DC coefficient.

In conclusion a number of important factors have arisen in determining the form of the quantiser. Firstly, it is important to avoid quantisation of the DC coefficient by factors of more than 2 as it very quickly affects the overall quality of the image. Secondly, care must be taken when quantising those coefficients that lie along (0, x) and (x, 0) of the quantisation matrix as these also contribute significant amounts to the

error. The remaining coefficients are reasonably resistant to error after quantisation and can be quantised by factor of 16 or even more in some situations. The last point is that the level of quantisation can be a function of the distance from the DC coefficient, the exact relationship depending on the required level of MSE.

5.3.2 Numerical Effects of Quantisation

The first numerical effect that will be examined is the frequency spread of the symbol space. The symbol space in this context can be defined as all of the possible values that a coefficient can take. To examine this the images were all transformed using the DCT and then quantised by a factor of 16. The minimum, maximum and average frequencies of all the symbols were then determined. The different images were then compared to determine the absolute minima and maxima throughout the entire test set. The results of this test are shown in Fig. 5-13. When these results are compared with the results obtained before quantisation in Fig. 5-1 the effect of quantisation can be noted. The graph is compressed along the symbol axis. By quantising the symbols their total number is reduced and they are shifted closer to zero. With extreme quantisation most, or all of the symbol values will become zero, but the MSE will of course, become unreasonably high.

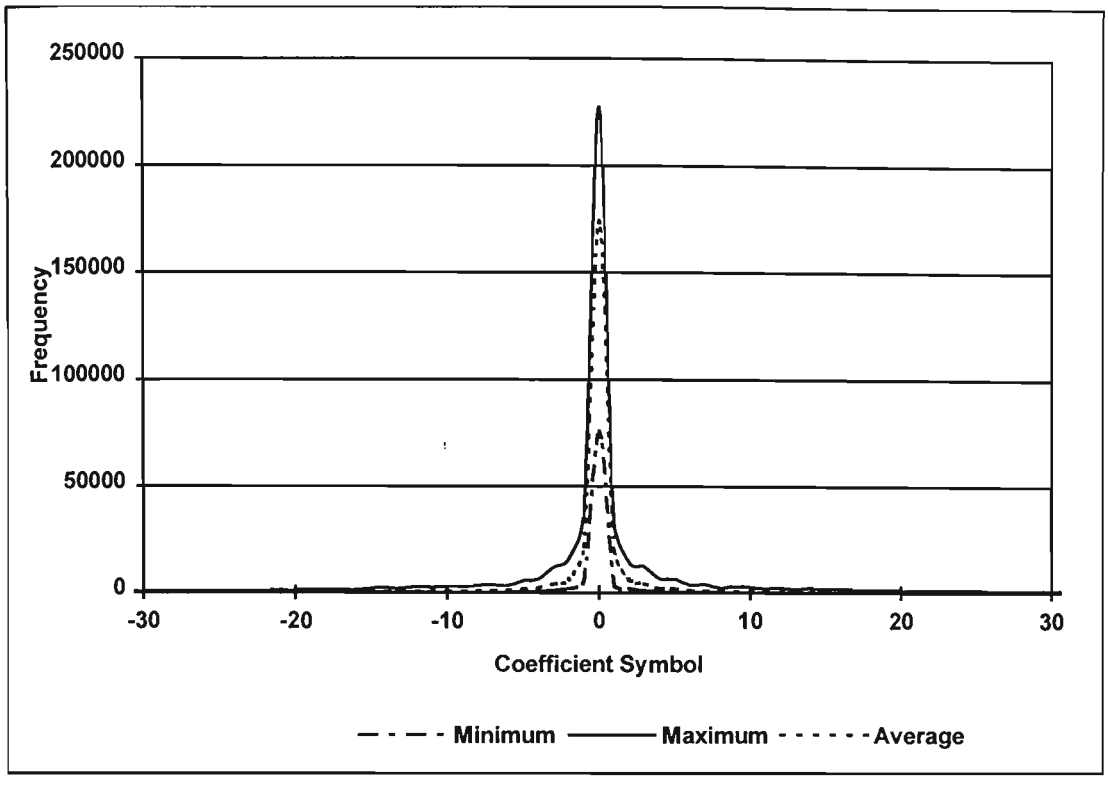


Fig. 5-13 Frequency of DCT coefficient symbols.

It is also quite easy to deduce the effect on the graphs shown in Fig. 5-2 to Fig. 5-5. The graphs will simply be scaled down by the same factor as the quantisation factor. Therefore a quantisation factor of 8 will reduce all the magnitudes by a factor of 8.

The final effect that should be examined is that of a non-uniform quantiser on the frequency of symbols. This type of quantiser is built up using a piece-wise algorithm, rather than a continuous curve. The quantiser consists of numerous linear quantisers which are activated at different points depending on the magnitude of the symbol. The characteristics of a typical non-uniform quantiser are shown in Fig. 5-14. This figure shows how the quantised symbol can be determined from the original symbol. Further

details of the non-uniform quantiser used in the algorithm described in this thesis can be found in section 5.6.

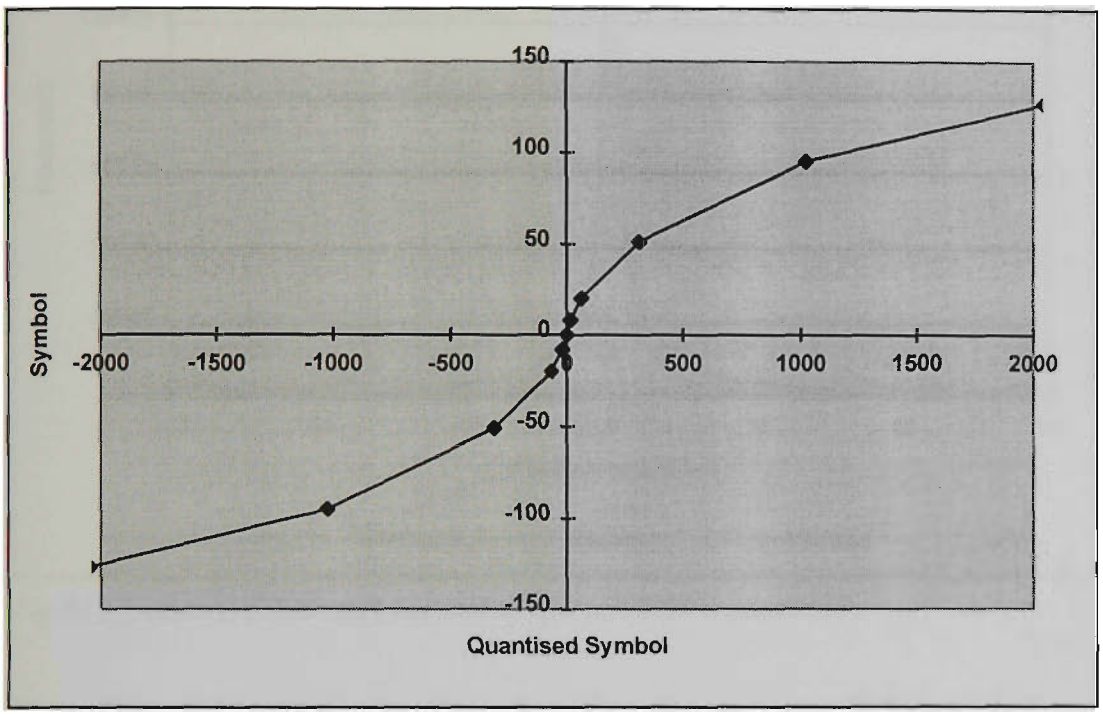


Fig. 5-14 A simple non-uniform quantiser.

All the intensity images were once again transformed and then quantised using the non-uniform quantiser with all coefficients being treated equally. It would have been better to incorporate the HVS into this and quantise the lower frequency coefficients less than the higher frequency coefficients. However, for demonstration purposes and for an accurate comparison with the previous linear quantiser tests, all of the coefficients have been treated equally. The minimum, maximum and average frequencies were determined using the same techniques described earlier. The results of the tests are shown in Fig. 5-15.

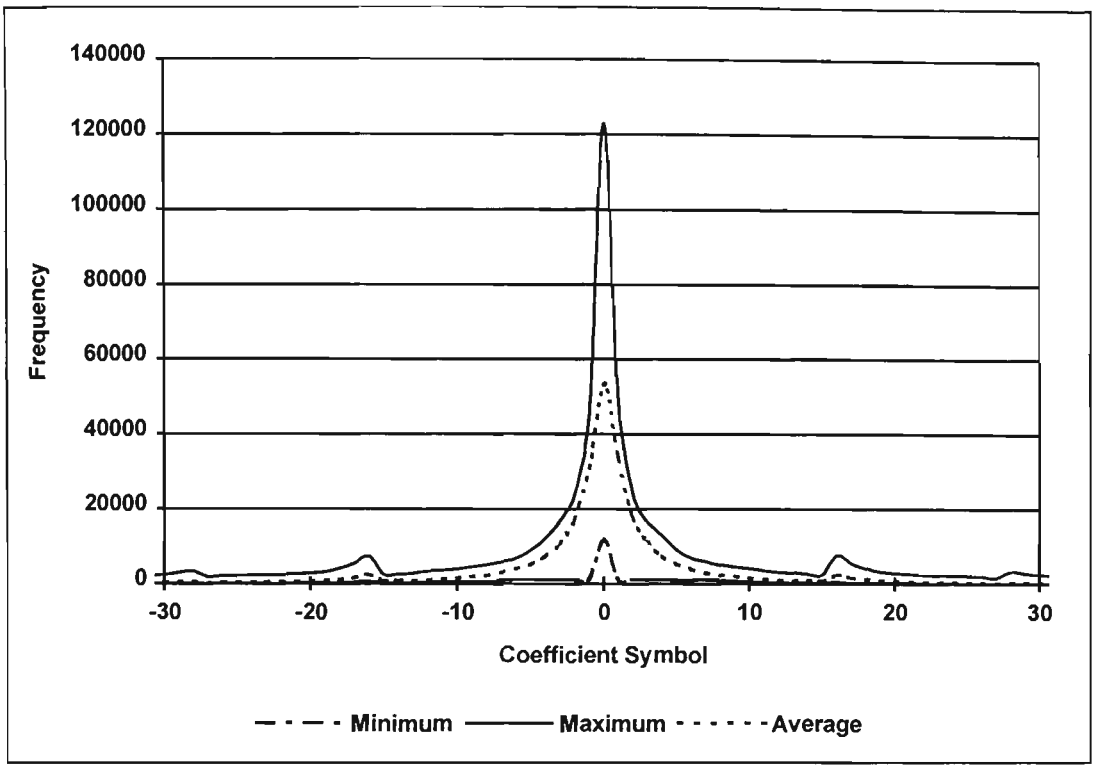


Fig. 5-15 Frequency of DCT coefficient symbols after non-uniform quantisation.

From Fig. 5-15 it can be seen that the distribution of the symbols are spread considerably more than that in Fig. 5-13. However, it should be noted that in the same magnified range of -30 to 30, both diagrams contain approximately the same amount of data and both now contain a total of less than 256 symbols after quantisation. Limiting the number of symbols to a number less than 256 is significant in that the information can then be contained within one eight-bit byte.

Another characteristic of the non-uniform quantiser is the formation of several crests in the frequency plot. This is caused by the non-continuous approximation to the non-uniform quantiser, at each break-over point several different symbols are mapped into the same symbol thereby increasing the frequency of this symbol. This actually is better than standard quantisation where a large number of coefficients are reduced to

zero value because the quantisation is greater than the coefficient value itself. This results in high losses and the all too familiar blocking effect when the image is reconstructed. With the non-uniform quantiser the coefficients are not cleared but rather shifted down by differing amounts depending on the magnitude of the coefficient in the first place. The smaller the coefficient magnitude, the less it is quantised. This does not necessarily result in a lower MSE but rather an image with a lowered entropy and with a lower level of visual degradation in the reconstructed image.

5.4 JPEG Quantiser

After close examination of the effects of quantisation on the DCT coefficients and on the resulting reconstruction error, it is worthwhile examining the operation of the JPEG quantiser. The JPEG algorithm is chosen here because it is a standard and there are a number of implementations available which may be used to compare with the algorithm described in this thesis. It is based on the DCT and as it is transform based, it contains a coefficient quantiser. Through examination of the JPEG quantiser, it is possible to determine to what extent it utilises the characteristics of the DCT coefficients.

The JPEG quantiser is composed of a matrix in which each element contains the quantisation factor for the corresponding DCT coefficient. Since the quantisation is performed by simple division it does not make use of the non-uniform effects

described earlier. Unfortunately the JPEG standard does not provide standard values for the quantisation matrix, it only recommends a set of values. One typical example of this quantisation matrix for the luminance component of an image is shown in Table 5.1.

x\y	0	1	2	3	4	5	6	7
0	16	11	10	16	24	40	51	61
1	12	12	14	19	26	58	60	55
2	14	13	16	24	40	57	69	56
3	14	17	22	29	51	87	80	62
4	18	22	37	56	68	109	103	77
5	24	35	55	64	81	104	113	92
6	49	64	78	87	103	121	120	101
7	72	92	95	98	112	100	103	99

Table 5.1 A typical JPEG quantisation matrix [PEN90].

Several observations can be made from the quantisation matrix. The first is that the matrix has only 64 elements; that is JPEG is based on a DCT with a block size of 8x8. As was determined earlier, this not an ideal choice for higher quality compression. The JPEG algorithm was tailored to provide rather large levels of compression without too much emphasis on the quality, but at a reasonably fast rate. JPEG catered for those requiring higher quality by defining a separate lossless algorithm which in most cases would only compress by a factor of two.

The second observation that can be made is that the different coefficients are quantised by different amounts. In particular, the lower frequency coefficients are quantised to a lesser extent than those of higher frequency. This conforms with the observations made earlier. The function relating the quantisation level to the DCT coefficient contains some anomalies, for example, the coefficient (6, 6) is quantised more heavily than (7, 7) even though it is closer to the DC coefficient. These are dependent on the image test set used to determine the quantisation matrix required to generate minimal error.

The last observation is that the values in the quantisation matrix are rather high. This results in a reconstructed image with obvious errors. The JPEG specification claims [PEN90] that if the quantisation matrix is divided by a factor of two the resulting image will normally be indistinguishable from the original. This, however, is not true if close examination is made between the two images. The JPEG quantiser introduces obvious distortions in the reconstructed image, note particularly the circled area in Fig. 5-16. These distortions are more noticeable on a video monitor than in the hardcopy images in this document. Even though the differences are not particularly evident they are still too much as far as this research project is concerned, this gives an indication of the level of quality that this algorithm is trying to achieve.

Fig. 5-16 Errors introduced in the JPEG quantiser.

The original can also be compared with the reconstructed image obtained with the OptIC quantiser described in this chapter, see Fig. 5-17.

Fig. 5-17 A comparison with the OptIC algorithm.

5.5 Development in the Compression Algorithm

An improvement was made in the DCT hardware driver described in the previous chapter. This was to modify the driver so that it may transform more than one block at a time thus reducing the amount of overhead required per block transformed and so

improving the efficiency of the algorithm. The forward and inverse transformations are shown diagrammatically in Fig. 5-18.

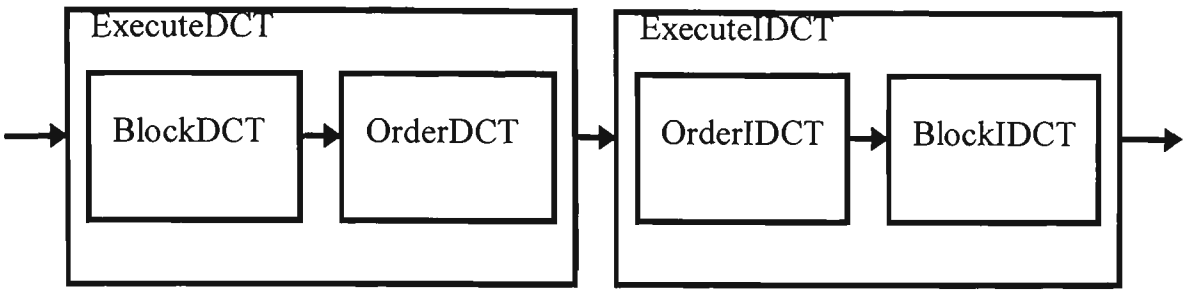


Fig. 5-18 Modified forward and inverse hardware DCT driver software.

Many different routines were written to perform specific functions within the overall algorithm. The two functions *ExecuteDCT* and *ExecuteIDCT* provide functionality for transforming an entire image in the forward or inverse direction. These two functions are composed of two further functions. *BlockDCT* and *BlockIDCT* send one line of blocks from the image to the hardware transform device. *OrderDCT* and *OrderIDCT* perform two functions, quantisation/dequantisation and ordering. The ordering component of the ordering function will be described in more detail in chapter 6. The C source code listing for these functions can be found in Appendix D.

5.6 Quantiser Realisation

The quantiser developed for implementation of this compression algorithm takes into account a number of the characteristics that were discussed in this chapter. The first characteristic employed was to classify the coefficients by their sensitivity to quantisation in terms of MSE. The second was to design individually tuned quantisers for each of the classed coefficients using a non-uniform quantiser. The tuning of the

quantisers was achieved through an iterative process involving subjective quality assessment. This tuning allowed the HVS to be incorporated into the quantiser more effectively.

It was highlighted in 5.1 that the MSE can be used as an approximate indicator of subjective image quality for a large image test set. As extensive tests of coefficient sensitivity to quantisation in terms of MSE were made in 5.3.1 for the entire image test set, it was possible to classify the coefficients into groups that represented different levels of sensitivity. It should be noted that this use of MSE was used only to group the coefficients into like types and does not actually define how the quantiser will operate on these groups. The actual definition of the quantiser must incorporate the HVS and so will be performed during the iterative tuning process.

The data used to classify the coefficients was obtained from Fig.5-11 which graphed the maximum MSE obtained by quantising each coefficient by a factor of 16. The maximum was taken over the entire image set and so provided worst case figures. The coefficients were classified into eight separate groups (0 to 7) depending on their sensitivity to quantisation. The result of this classification can be found in Table 5.2. Note that group 0 coefficients produce very high levels of MSE when quantised whereas group 7 coefficients produce low levels of MSE when quantised.

It should be noted that the results of the grouping actually support some of the theory described with respect to the HVS, this is because of the large image set. For example, the DC coefficient and those coefficients along the axis are in the lower groups indicating that they are sensitive to quantisation. This is also true in terms of the HVS, that is, quantisation of these coefficients should be minimised to acheive improved subjective image quality. The same is true for the higher frequency coefficients which are in the higher groups indicating that they are insensitive to quantisation. It does not, however, fully incorporate the HVS as the quantiser itself has not been defined by using each of the images individually through subjective assessment.

x\y	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	2	2	2	3	3	3	3	3	3	4	5	6	7
1	1	2	2	2	3	3	4	4	4	5	5	6	7	7	7	7
2	1	2	2	2	3	3	4	4	4	5	5	6	7	7	7	7
3	2	2	2	3	3	3	4	4	5	5	5	6	6	7	7	7
4	2	3	3	3	3	4	4	4	5	5	5	6	6	6	7	7
5	2	3	3	3	4	4	4	5	5	5	6	6	6	6	7	7
6	3	4	4	4	4	4	5	5	5	5	6	6	6	6	7	7
7	3	4	4	4	4	5	5	5	5	6	6	6	6	6	7	7
8	3	4	4	5	5	5	5	5	6	6	6	6	6	7	7	7
9	3	5	5	5	5	5	5	6	6	6	6	6	6	7	7	7
10	3	5	5	5	5	6	6	6	6	6	6	6	7	7	7	7
11	3	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7
12	4	7	7	6	6	6	6	6	6	6	7	7	7	7	7	7
13	5	7	7	7	6	6	6	6	7	7	7	7	7	7	7	7
14	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
15	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7

Table 5.2 Coefficient quantisation types.

In order to incorporate the HVS, eight quantiser types were defined for the eight groups of coefficients.The quantiser is not based on a continuous function but rather an approximation of one built with linear sections. By using the linear sections, greater control is gained over the shape of the quantiser function and it can also be implemented with a very fast look-up table requiring no multiplication function. The linear sections are defined in the array constants *quant_x* and *quant_y,* where *quant_x* is the coefficient magnitude at which a new linear section begins and *quant_y* defines the slope of the linear section. The function *gen_quant* generates a look-up table for a particular quantisation type given the data stored in *quant_x* and *quant_y*. It generates two tables in parallel, the first is the quantisation table (*hForward*) and the second is the de-quantisation table (*hBackward*). The characteristics of the eight quantiser types in this compression algorithm can be seen in Fig. 5-19.

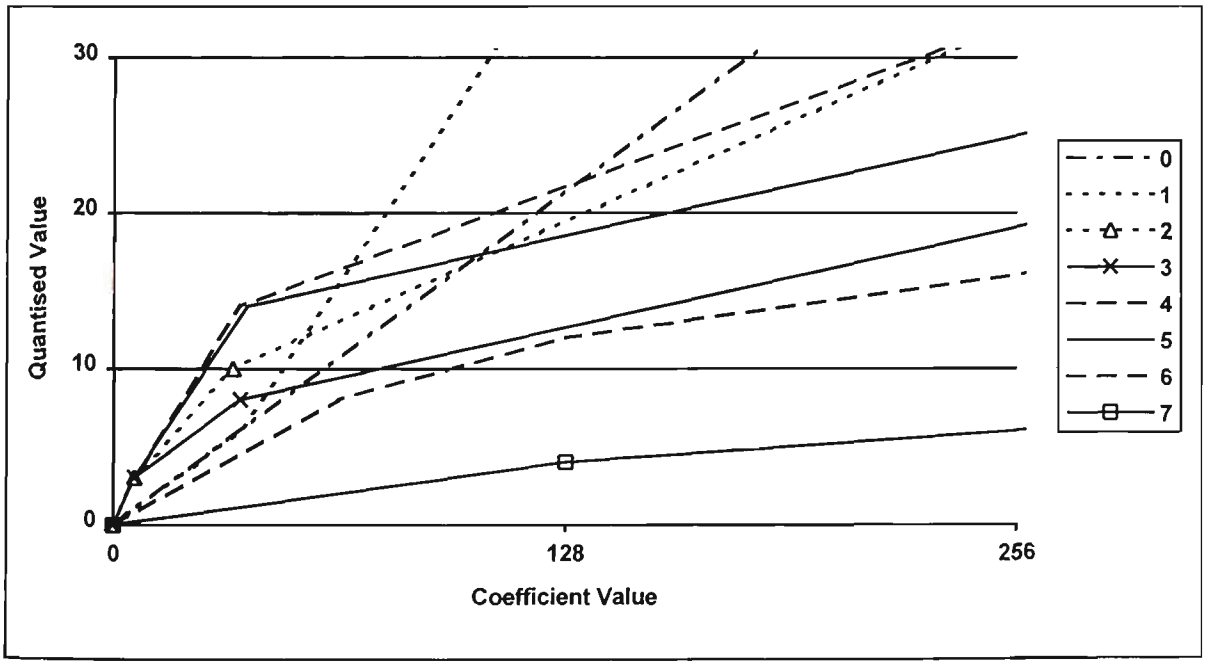


Fig. 5-19 Characteristics of the eight quantisation types.

The exact form of the quantisation types were determined through an iterative process. The iteration process is three fold: the top level iteration is to process the quantiser types from level 7 down to level 0, the second level is to process the separate linear sections that form the non-uniform quantiser, that is, to define *quant_x* and *quant_y* and the final level is to determine the quantisation level possible for the current quantisation type and linear section. This final level of iteration requires testing each image with the current quantiser definition, if this resulted in no visible restoration error, then the current quantisation level of the current quantiser type and linear section was increased. If there was visible reconstruction error then the previous iterations quantisation level was used and processing was continued with the next linear section or quantisation type.

5.7 Results

5.7.1 Frequency Distribution

The frequency distribution of the transformed images is shown in Fig. 5-20. As in section 5.3.2 the minimum, maximum and average frequencies for the coefficient symbol values from -30 to 30 are shown for the intensity images in the test set.

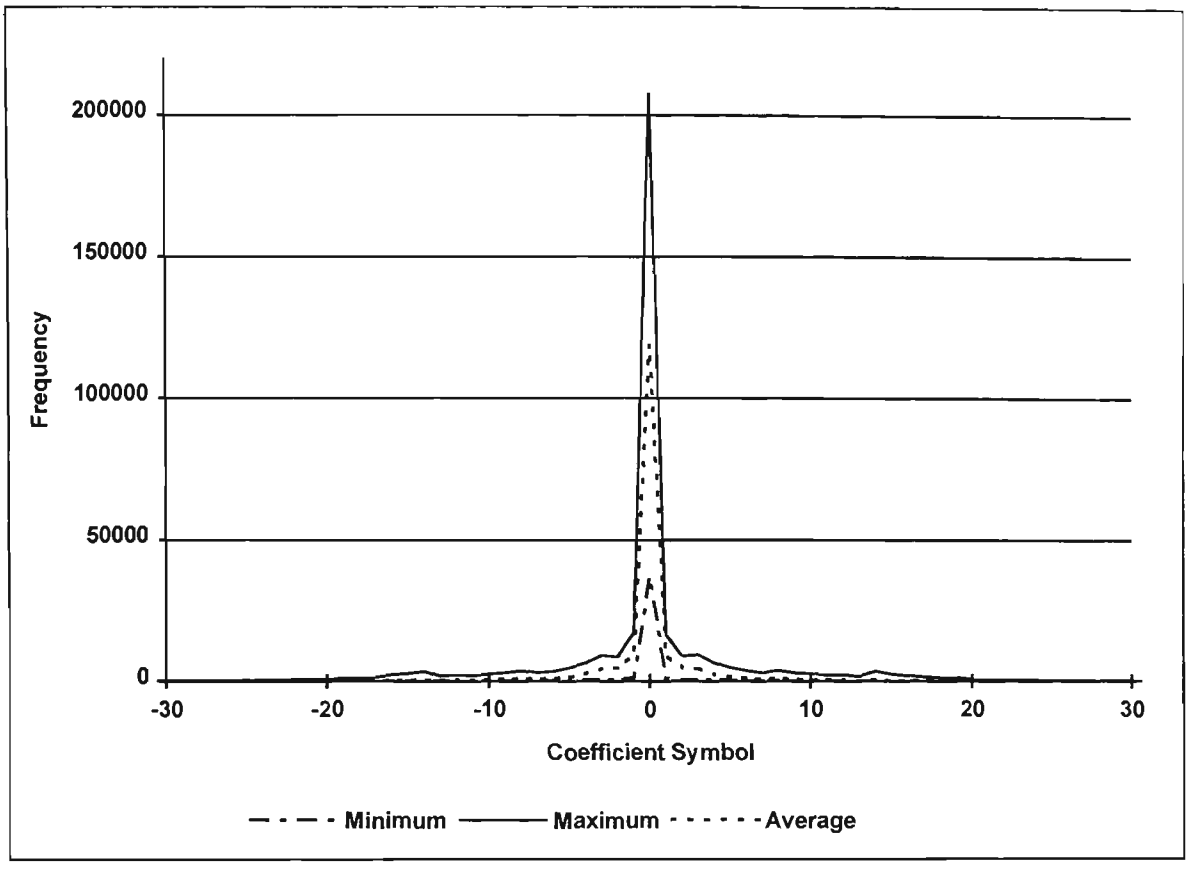


Fig. 5-20 Frequency of DCT coefficient symbols after quantisation.

The frequency distribution's characteristic is similar to that of the non-uniform quantiser distribution shown in Fig. 5-15. It differs only in that the new distribution exhibits a higher level of quantisation, that is, there is a higher concentration of symbols in the -4 to 4 range. The existence of a slight bulge about the -3 and 3 symbol values and the slight bulges about the -15 and 15 symbol values are also indicative of the non-uniform quantiser.

5.7.2 Reconstruction Error

The image reconstruction error after quantisation with the non-uniform quantiser and the JPEG quantiser is shown for all of the intensity images in Table 5.3.

Image File	Hardware DCT MSE	Non-uniform Quantiser MSE	JPEG Quantiser MSE
Airplane.Y	0.157028	5.549461	3.438511
Baboon.Y	0.157040	35.092472	8.885120
Beans1.Y	0.158203	2.166992	1.400757
Beans2.Y	0.156799	3.110580	1.887634
Couple.Y	0.155807	6.557709	3.406158
Girl1.Y	0.156052	6.747757	4.122482
Girl2.Y	0.157227	4.548828	2.563141
Girl3.Y	0.156555	5.282883	3.289993
House.Y	0.158615	6.971359	3.507980
Lena.Y	0.157051	6.442287	4.604576
Peppers.Y	0.156776	11.092140	6.831085
Sailboat.Y	0.157990	16.493061	7.869686
Splash.Y	0.156040	3.998802	3.219627
Testpatt.Y	0.077656	28.986454	2.292976
Tiffany.Y	0.154491	8.026783	4.489658
Tree.Y	0.157471	19.055832	7.117462
Wendy1.Y	0.155964	2.161739	1.987961
Wendy2.Y	0.157650	4.058239	2.382908
Wendy3.Y	0.156033	3.511997	2.882843

Table 5.3 Image reconstruction error for the non-uniform and JPEG quantiser.

The error increase was greatest for the images *Baboon.Y* and *Testpatt.Y*. These images contain a large amount of high frequency data. That is, there are very rapid transitions between the intensity levels of adjacent pixels. As high frequency components of the image are being heavily quantised, the MSE of these images must increase more than in images where high frequency data is not so prevalent. These errors, however, are not perceived by the human eye and so are not of great concern. A hard copy of the original and reconstructed images can be found in Appendix A Image Test Set for direct comparison purposes.

Comparing the non-uniform quantiser MSE results with the JPEG quantiser MSE results, the uninitiated would assume that the latter was the superior of the two. However, in this situation, the reverse is true. Both of the results relate to the same level of perceivable image quality as visualised with the human eye. The results show that the non-uniform quantiser was able to add more non-perceivable distortions to the image than the JPEG quantiser. These non-perceivable distortions are as defined by the HVS. By adding more quantiser-introduced distortions it is possible to reduce the entropy of the image and so improve the compression factor after coding, see chapters 6 and 7 for further details on coding.

5.7.3 Timing Benchmarks

To transform and quantise an entire 512x512 intensity image now takes 3.5 seconds. This requires 1024 16x16 DCT transforms which equates to approximately 3.4mS per transform including quantisation. This is almost twice the time required for just a single 16x16 transform but includes all the overhead for pre-processing and quantisation.

5.7.4 Entropy Effects

Equation (5.1) defines the size of an image when compressed using an ideal statistical non-adaptive coder, as will be described in chapter 7. The original image size and the new image size are measured in bytes.

$$NewSize = \frac{OriginalSize \times Entropy}{8} \qquad (5.1)$$

The effect on the entropy of the images after quantisation is shown in Table 5.4.

Image File	Original	After DCT	After Quantisation	Original Image Size	Compressed Image Size[1]
Airplane.Y	6.7056	4.49679	2.277871	262144	74642
Baboon.Y	7.35814	6.24667	3.434531	262144	112543
Beans1.Y	5.72470	3.02061	1.215568	65536	9958
Beans2.Y	6.24254	3.64295	1.659312	65536	13594
Couple.Y	6.42737	4.51369	2.263864	65536	18546
Girl1.Y	7.05377	4.55489	2.261720	65536	18529
Girl2.Y	5.60747	3.77790	1.603080	65536	13133
Girl3.Y	7.26169	4.31520	2.139159	65536	17524
House.Y	6.50448	4.38083	2.153411	65536	17641
Lena.Y	7.44776	4.53318	2.248318	262144	73673
Peppers.Y	7.59430	4.80349	2.349689	262144	76995
Sailboat.Y	7.48579	5.40114	2.770187	262144	90774
Splash.Y	7.25848	4.05722	1.982384	262144	64959
Testpatt.Y	0.98935	6.20320	3.816599	262144	125062
Tiffany.Y	6.60048	4.51991	2.220038	262144	72747
Tree.Y	7.31444	5.80349	3.106545	65536	25449
Wendy1.Y	6.82276	3.35043	1.472530	262144	48252
Wendy2.Y	7.09684	4.33512	2.287059	262144	74943
Wendy3.Y	7.82136	4.22606	2.180153	262144	71440

Table 5.4 Entropy of the image after quantisation.

Note that the entropy has improved on average by approximately 2.2 bits per pixel for all of the images. This represents an overall reduction of approximately 185Kb for a 256Kb image after transformation and quantisation.

These results will not be compared with other compression techniques at this point in the development of the algorithm. The reason is that the algorithm requires the run-length coding and statistical coding stages before the full compression stage is completed. The comparison will instead be performed in chapter 8, where the algorithm is analysed as a whole.

[1] The compressed image size is estimated based on the entropy using (5.1).

5.8 Conclusion to the Chapter

The non-uniform quantiser was quite successful in reducing the entropy of the image without any visible effect on the reconstructed image. The implementation of the new DCT and quantiser also proved to be quite fast, the entire transformation and quantisation stage requiring an execution time of only 3.5 seconds for a 512x512 intensity image.

6. The Run-Length Coder

6.1 Introduction

In section 5.7.1 it was shown that there was a very large number of zero symbols in each transformed image. This suggests that the probability of having a run of adjacent zero valued symbols is very high. An extremely efficient algorithm for compressing this form of data is the run-length coder. The run-length coder forms the third function block in the compression algorithm and is the first layer of compression.

The run-length coder is ideal for this application in that it is computationally very fast as it requires no complex arithmetic. The algorithm is also lossless and so will not add to the errors already accumulated after transforming and quantising the image. Furthermore, run-length coders are more efficient than statistical coders in this application as they treat large blocks of data as one or two symbols whereas the statistical coders must treat each symbol individually, see chapter 7.

6.2 A Basic Run-Length Coder

The basic operation of a run-length coder is to replace a run of equal symbols with a smaller, less frequent symbol. Fig. 6-1 is a typical sequence of symbols.

0	0	0	1	0	0	0	0	0	0	0	3	2

Fig. 6-1 A typical sequence of symbols.

The sequence contains four different symbol values : 0, 1, 2 and 3. However, the zero symbol occurs more often than the others and often forms runs. There are two methods by which this sequence can be coded using run-length coding techniques. The first technique uses a new symbol, or a less probable symbol, to represent a run of a more probable symbol. This is generally as shown in the Table 6.1 where *s0'*, *s1'*, *s2'*, and *s3'* are the run-length coded symbol representations and *s0*, *s1*, *s2*, and *s3* are the former symbol representations.

Run-length Coded Symbol	Original Symbol(s)
s0'	s0
s1'	s1
s2'	(s0,s0,s0)
(s3',sx)	sx

Table 6.1 Run-length coding technique 1.

In technique 1 the *s0'* and *s1'* symbols still represent the original symbols *s0* and *s1* respectively, however, symbol *s2'* now represents a run of three *s0* symbols. Symbol *s2* must now somehow be represented with the remaining symbol value, that is, symbol *s3*. This is done by creating a composite symbol value where symbol *s3'* is followed by the symbol that this composite represents. Thus (*s3'*,*s2*) represents symbol *s2*, (*s3'*, *s3*) represents symbol *s3*, etc. [Note that the codes (*s3'*,*s0*) and (*s3'*,*s1*), though possible, are redundant as they can be represented by the shorter *s0'* and *s1'* symbols respectively]. Using this technique we can recode the sequence in Fig. 6-1 to that shown in Fig. 6-2.

2	1	2	2	0	3	3	3	2

Fig. 6-2 A run-length coded sequence - technique 1.

Note that the run-length coded sequence is four symbols shorter than the original sequence, which is a 30% reduction in the sequence length.

The second technique by which run-length coding can be applied is by representing a run length of *n* by the form (*s2'*, *n*). Using this method it is possible to encode the symbol set as shown in Table 6.2.

Run-length Coded Symbol	Original Symbol
s0'	s0
s1'	s1
(s2', n)	(s0, s0, s0, ...)
(s3', sx)	sx

Table 6.2 Run-length coding technique 2.

With this technique the symbol sequence (*s2'*,4) now represents (*s0,s0,s0,s0*), (*s2'*,6) now represents (*s0,s0,s0,s0,s0,s0*) and so on. The symbols s2 and s3 are still represented by the larger composite symbols (*s3',s2*) and (*s3',s3*) respectively. Using this mapping of symbols the original sequence can be run-length coded in the sequence shown in Fig. 6-3.

2	3	1	2	7	3	3	3	2

Fig. 6-3 A run-length coded sequence - technique 2.

Once again, the length of the resultant sequence reduced by 30% when compared to the original.

The two different techniques have advantages and disadvantages depending on the statistics of the input sequence being coded. For sequences containing many short runs of a single symbol, the first technique is more efficient as there is less overhead

in representing the run-length. That is, only one symbol is required to represent a run of symbols, whereas two symbols are required in the second technique.

The second technique is more efficient in situations where there are very large runs of a particular symbol. The first technique is not efficient here as it can only represent fixed lengths of a symbol and so the symbol must be repeated several times to represent the large run. For example, to represent a run of twenty zeros would require a sequence of eight symbols (*s2'*,*s2'*,*s2'*,*s2'*,*s2'*,*s2'*,*s0'*,*s0'*) using technique 1, whereas using technique 2 the sequence would only require the two symbols (*s2'*, 20).

6.3 The DC Coefficients

Before commencing the design of the run-length coder there is a further improvement that may be done in reference to the DC coefficients after transformation. In general, the DC coefficients have reasonable large values as they represent the average intensity of the block. However, as the average intensity of adjacent blocks does not vary greatly it would be advantageous to use difference coding for the DC coefficient.

Difference coding of the DC coefficients is also used in the JPEG algorithm. The OptIC algorithm differs in that the result of the difference coding is also run-length coded. In JPEG the DC coefficients are first difference coded then statistically coded, the run-length coder is not used. The reason for using the run-length coder in the OptIC algorithm is to convert the 12 bit coefficient values to 8 bit symbols.

Difference coding is very simple to implement and would be performed after the quantisation stage. The difference coder is defined in (6.1) and the difference decoder is defined in (6.2). The difference decoder would be performed after the run-length decoding and before the dequantisation.

$$X'(0)_{n-1} = X(0)_{n-1} - X(0)_n \quad n = 1, \cdots, NumberOfBlocks - 1 \tag{6.1}$$

$$X(0)_{n-1} = X'(0)_{n-1} + X'(0)_n \quad n = NumberOfBlocks - 1, \cdots, 1 \tag{6.2}$$

The results gained through the introduction of the DC difference coding are shown in Table 6.3. Note that improvements were gained in all but two images, *Lena.Y* and *Wendy1.Y*.

Image File	Without DC Difference Coding	With DC Difference Coding
Airplane.Y	2.282744	2.277871
Baboon.Y	3.440941	3.434531
Beans1.Y	1.220645	1.215568
Beans2.Y	1.664439	1.659312
Couple.Y	2.264821	2.263864
Girl1.Y	2.265231	2.261720
Girl2.Y	1.607322	1.603080
Girl3.Y	2.141358	2.139159
House.Y	2.157442	2.153411
Lena.Y	2.247754	2.248318
Peppers.Y	2.352095	2.349689
Sailboat.Y	2.775944	2.770187
Splash.Y	1.985177	1.982384
Testpatt.Y	3.824703	3.816599
Tiffany.Y	2.221588	2.220038
Tree.Y	3.112865	3.106545
Wendy1.Y	1.471500	1.472530
Wendy2.Y	2.287395	2.287059
Wendy3.Y	2.187438	2.180153

Table 6.3 Entropy improvements gained by using DC difference coding.

6.4 *Input Statistics*

As has been discussed, it is important to understand the statistics of the input data before the run-length coder is designed. The most useful test is to measure the probability of a zero symbol value coefficient. This is done for each coefficient and over the entire set of test images. The result of this test is shown in Fig. 6-4.

It can be seen that the probability of having a zero symbol value coefficient is very high for coefficients furthest from the DC coefficient (0,0). The statistics suggest two methods by which we can modify the ordering of the data to improve the result of the run-length coder.

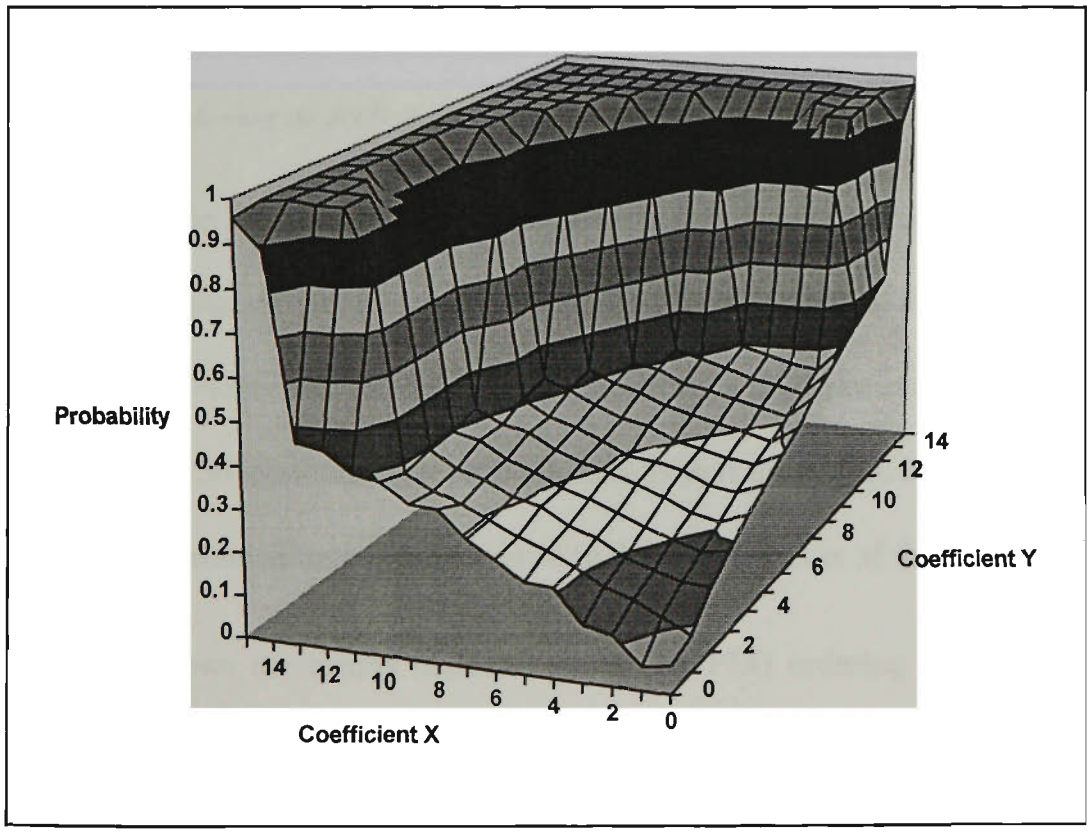


Fig. 6-4 Probability of a zero symbol value for each coefficient.

The first method is to follow the technique used by JPEG and reorder the coefficients within one block. This can be done by forming a vector containing all of the coefficients in order of lowest to highest probability of zero symbol value. When run-length coded, the first part of the vector would show very little signs of compression. However, it is very likely that the remainder of the vector will contain mainly zero symbol values and so will be readily compressed by the run-length coder. An example of this can be seen in Fig. 6-5. The DC and lower frequency components of the DCT tend to prevent long runs from forming.

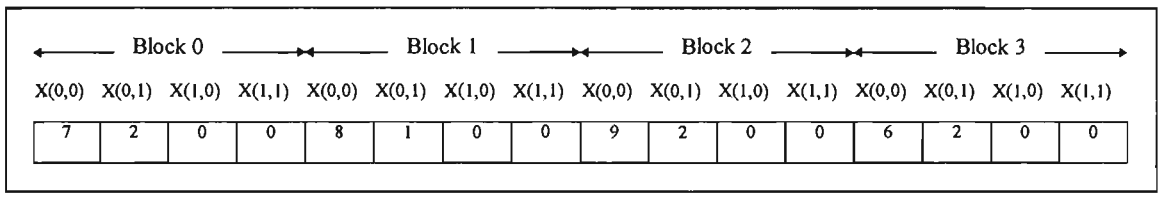


Fig. 6-5 An example using the JPEG ordering method.

If we apply the JPEG ordering method to a typical *512x512* pixel image, 1024 *16x16* DCT transforms are required each containing 256 coefficients. It is very unlikely that the DC coefficients will be zero and so, in virtually all situations, the run-length will be less than 256 symbols. The problem here is that those values that are likely to be zero are not grouped together thus limiting the effectiveness of the run-length coder.

Table 6.4 shows the results obtained when the JPEG ordering method is used. The table contains data relating to the zero symbol and includes the maximum run size, the total number of runs and the number of specific run lengths. The specific lengths

range between two and ten. Note that in most of the images it is rare to produce runs of length greater than six.

Image	Number of Runs	Maximum Run Size	Run of 2	Run of 3	Run of 4	Run of 5	Run of 6	Run of 7	Run of 8	Run of 9	Run of 10
Airplane.Y	7121	12	4610	1389	673	266	98	28	43	0	0
Baboon.Y	1200	4	1058	128	14	0	0	0	0	0	0
Beans1.Y	5810	44	2413	1305	826	487	277	162	95	99	59
Beans2.Y	4243	16	1931	940	575	339	174	104	74	51	29
Couple.Y	2431	15	1479	529	218	108	44	28	6	11	1
Girl1.Y	1421	6	1118	238	52	12	1	0	0	0	0
Girl2.Y	3413	8	2302	695	275	77	50	9	5	0	0
Girl3.Y	1943	8	1429	362	119	21	9	2	1	0	0
House.Y	2269	9	1566	463	161	52	19	5	1	2	0
Lena.Y	353	8	317	6	28	0	0	1	1	0	0
Peppers.Y	336	3	319	17	0	0	0	0	0	0	0
Sailboat.Y	101	3	84	17	0	0	0	0	0	0	0
Splash.Y	746	9	515	128	69	16	3	1	0	14	0
Testpatt.Y	2703	255	365	448	140	182	182	154	154	546	14
Tiffany.Y	417	4	369	34	14	0	0	0	0	0	0
Tree.Y	1217	13	735	279	99	43	28	15	6	6	3
Wendy1.Y	1191	7	816	195	76	88	2	14	0	0	0
Wendy2.Y	952	14	478	201	163	81	17	6	1	0	2
Wendy3.Y	797	7	460	225	79	18	1	14	0	0	0

Table 6.4 Results obtained using the JPEG ordering method.

The second method is an improvement over the first and is the method used in the OptIC algorithm. This method groups all like coefficients from all of the blocks in an image together. That is, all (0,0) coefficients are in one group, all (0,1) coefficients are in another group and so on. These groups are sorted in order of lowest to highest probability of zero symbol values for the coefficients contained within that group. An example of the OptIC ordering method is shown in Fig. 6-6. Note how the X(1,0) and X(1,1) coefficients now form a large run.

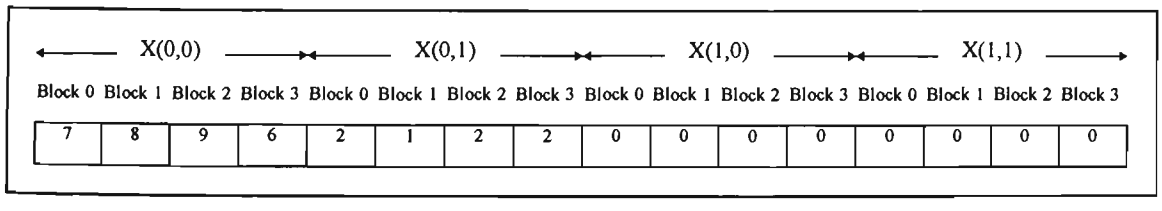


Fig. 6-6 An example using the OptIC ordering method.

When the OptIC ordering method is applied to a typical *512x512* pixel image that has been transformed by a *16x16* DCT transform an improvement in the run-lengths occurs. The reason is that we have grouped all of the coefficients that are unlikely to have a zero value together so that they do not disrupt the runs we can obtain with the coefficients that are more likely to be zero. A further improvement is that groups of 1024 are now formed as this is the number of block transforms required to process a *512x512* pixel image, this is four times longer than the groups formed in first method. Also, as these groups are adjacent to other groups that are very likely to have zero value symbols it is possible to have runs of several thousands of symbols.

Image	Number of Runs	Maximum Run Size	Run of 2	Run of 3	Run of 4	Run of 5	Run of 6	Run of 7	Run of 8	Run of 9	Run of 10
Airplane.Y	9423	18434	3448	1586	974	618	436	329	221	185	156
Baboon.Y	9358	962	3554	1528	808	549	421	321	221	179	156
Beans1.Y	1577	4743	251	133	185	91	226	81	46	26	63
Beans2.Y	2180	4698	647	304	221	170	206	110	63	50	36
Couple.Y	2871	4625	1056	478	316	190	131	113	78	51	49
Girl1.Y	2904	3399	1008	482	287	171	134	103	92	60	66
Girl2.Y	3003	4696	790	434	347	201	142	97	110	111	89
Girl3.Y	2630	4638	922	446	254	162	122	113	84	44	62
House.Y	2540	4663	879	437	266	186	129	69	89	51	40
Lena.Y	10460	12488	3954	1941	988	585	408	337	269	180	172
Peppers.Y	10984	7700	4229	2010	1123	681	434	333	261	171	177
Sailboat.Y	10116	15442	3838	1802	954	632	419	311	237	198	188
Splash.Y	9805	18445	3814	1676	1051	633	464	342	291	189	174
Testpatt.Y	7603	160	2518	864	365	145	89	22	14	3	2
Tiffany.Y	11722	5133	4193	2124	1213	735	560	427	361	237	192
Tree.Y	2477	4615	889	450	283	205	167	146	88	61	29
Wendy1.Y	9994	18615	3113	1526	1069	774	569	509	408	301	208
Wendy2.Y	8447	18720	2838	1435	902	588	425	271	234	221	196
Wendy3.Y	8546	18578	3360	1529	887	530	414	282	259	156	132

Table 6.5 Results obtained using the OptIC ordering method.

Table 6.5 shows the zero symbol run-length results when the second ordering method is used. Note that the size of the runs has increase dramatically, in some cases it has increased in size by a factor of more than 5000. The number of runs has also changed,

where two runs may merge or where new runs a formed. In all but four images (Beans1.Y, Beans2.Y, Girl2.Y and Testpatt.Y) the number of runs have increased and the size of the runs have increased. In all of the images, however, the sizes of the runs on average increased.

6.5 Input Ordering

In the software implementation of the OptIC ordering method, the ordering is performed by look-up tables in two stages. The first is to collect like coefficients from all the blocks into groups. When grouping these coefficients an order in which they will be extracted from the blocks, that is, the block ordering must be defined. The blocks are ordered by scanning rows from the top of the image down to the bottom. Each even row is scanned from left to right and each odd row from right to left.

In general, an image may contain sections of uniform characteristics that may spread over several adjacent blocks. Some of these relations may still appear in one form or another after transformation. As such, adjacent blocks will often contain related information. By alternating the direction of scanning from row to row a sudden change in the block characteristics as we change from one row to another is avoided.

Table 6.6 shows the ordering used in the case of a *256x256* image. The table is generated by the subroutine *CreateBlockOrder* and stored in the array variable *blockorder*, refer to Appendix D. *CreateBlockOrder* takes the parameter *width* as the

number of blocks which is dependent on the image size and so too is the ordering of the blocks.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
95	94	93	92	91	90	89	88	87	86	85	84	83	82	81	80
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
127	126	125	124	123	122	121	120	119	118	117	116	115	114	113	112
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
159	158	157	156	155	154	153	152	151	150	149	148	147	146	145	144
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
191	190	189	188	187	186	185	184	183	182	181	180	179	178	177	176
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
223	222	221	220	219	218	217	216	215	214	213	212	211	210	209	208
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
255	254	253	252	251	250	249	248	247	246	245	244	243	242	241	240

Table 6.6 Scanning order for blocks in a 256x256 image after transformation.

The second stage of the ordering process is to define the order in which the coefficient groups will be stored. This is based of the probability of the coefficients having a symbol value of zero. By using the data shown in Fig. 6-4 and sorting the probability values from lowest to highest a coefficient ordering array can be formed. The result is shown in Table 6.7.

y\x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	16	0	1	32	17	48	2	33	18	49	34	19	3	50	35	20
1	64	65	80	36	51	4	66	21	81	37	67	52	82	96	5	22
2	97	38	83	53	68	23	54	98	69	99	112	113	84	6	85	39
3	70	24	114	55	100	115	40	129	7	101	25	71	86	56	116	130
4	41	87	102	131	145	26	72	117	57	128	146	144	8	88	42	132
5	118	103	133	73	161	58	147	148	119	9	89	104	162	74	163	160
6	134	120	149	105	176	135	164	10	150	11	192	208	12	13	27	43
7	59	177	90	178	60	179	75	106	121	136	151	165	91	180	137	195
8	166	122	107	76	152	181	167	92	182	77	196	168	153	123	138	197
9	108	183	139	14	184	124	212	93	198	154	199	140	109	213	169	170
10	224	155	156	186	125	200	214	215	171	185	201	44	28	45	240	46
11	29	47	61	30	62	31	94	78	110	63	209	193	111	95	241	79
12	194	210	126	243	172	157	127	245	232	246	242	244	173	189	226	141
13	211	216	220	143	229	217	230	247	159	204	231	142	227	233	158	202
14	221	174	187	205	188	236	228	190	234	235	203	219	175	225	237	248
15	218	250	206	191	238	223	222	251	249	207	254	239	252	253	15	255

Table 6.7 Ordering for coefficients after transformation, where the coefficients are represented by (x,y).

From Table 6.7 it can be seen that, by following the order in sequence from 0 to 255, the first coefficient stored will be coefficient (1,0), followed by coefficients (2,0), (6,0), (12,0) and (5,1). The coefficient ordering table is stored in the integer constant array *coefficientOrder* and is used in the procedure *CreateBlockOrder* to generate the array *coeffOrder*. The array *coeffOrder* is the coefficient ordering matrix multiplied by a factor equivalent to the number of blocks in the image. This factor is used to separate the groups of differing coefficients.

The ordering of the data in preparation for the run-length coding is performed in the procedures *OrderDCT* and *OrderIDCT* along with the quantisation. Refer to Appendix D for the listing of the C software implementation.

6.6 Run-Length Coder Design

After the ordering of the coefficients it is important to take a closer look at the characteristics of the data. The most important information at this stage is to determine which symbols most often form runs and the length of the runs. This was done by transforming, quantising and ordering all of the intensity images in the test set and then counting the number of runs between lengths of two and nine. A summary of the results of this test is shown graphically in Fig. 6-7.

From Fig. 6-7 it can be seen that the symbol that most frequently forms runs is the zero symbol. It can also be noted that a large number of symbols in the range of -4 to

4 also often form runs of length 2. The final important characteristic is that apart from the zero symbol, few symbols form runs of length greater than 3.

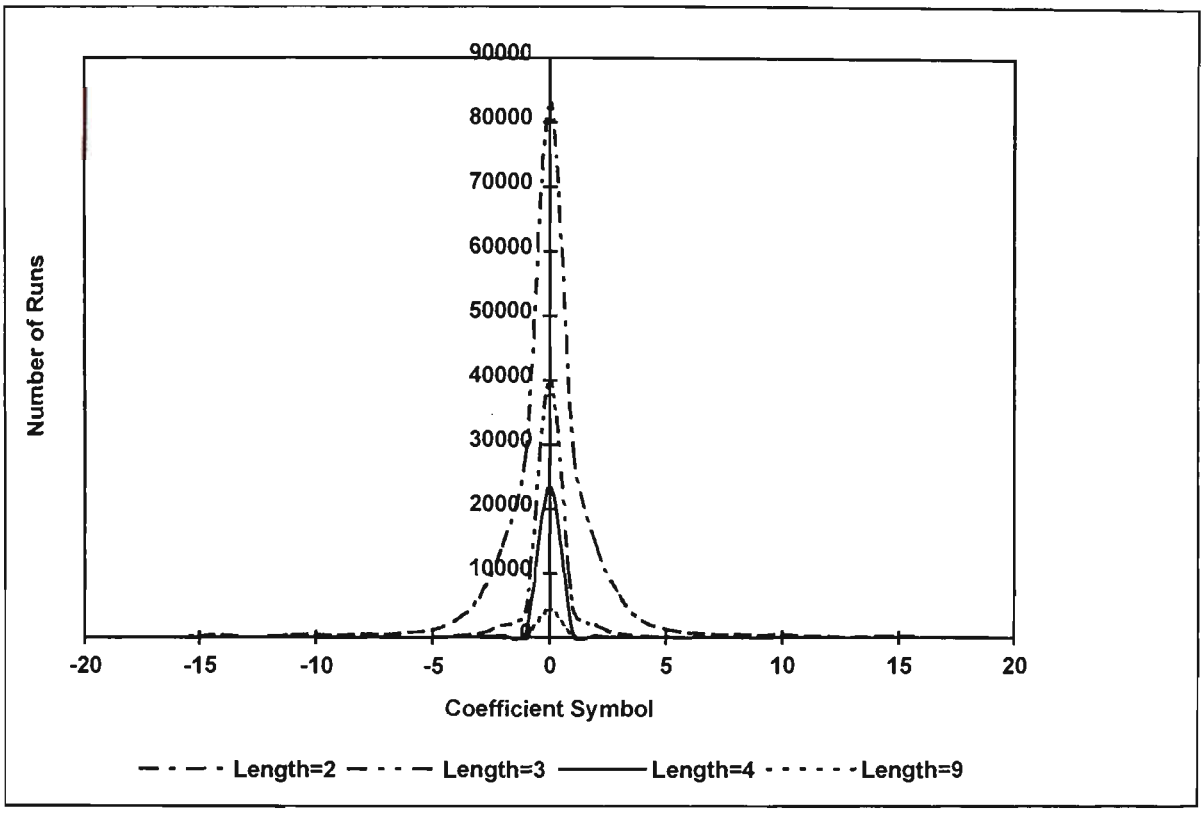


Fig. 6-7 Number of runs per given coefficient symbol in image test set.

The test results suggest that both of the run-length methods described in section 6.2 are required to optimally code the transformed and quantised image data. That is, a small run should be represented by single symbol, whereas, a large run can be represented by a multi-symbol sequence with the run-length and perhaps the original symbol coded within it.

The coefficients that are produced by the DCT transform are all 12 bits in size. As most of these coefficients lie within the range of -100 to 100 it is better to store these

as an eight bit symbol. Symbols which are too large to be represented within these eight bits must be represented by a slightly longer string of symbols. As the large symbol values are relatively improbable this will have little effect on the efficiency of the run-length coder.

Using the test results the symbol mappings shown in Table 6.8, Table 6.9 and Table 6.10 were designed for the OptIC run-length coder.

Run-length Coded Symbol	Original Symbol
(127)	(0, 0)
(126)	(0, 0, 0)
(125)	(0, 0, 0, 0)
(124)	(0, 0, 0, 0, 0)
(123)	(0, 0, 0, 0, 0, 0)
(122)	(0, 0, 0, 0, 0, 0, 0)
(121)	(0, 0, 0, 0, 0, 0, 0, 0)
(120)	(0, 0, 0, 0, 0, 0, 0, 0, 0)
(119, n-10)	(0) x n, where n = 10..265

Table 6.8 Run-length coding of zero symbol runs.

Run-length Coded Symbol	Original Symbol
(-128)	(-3, -3)
(-127)	(-2, -2)
(-126)	(-1, -1)
(-125)	(1, 1)
(-124)	(2, 2)
(-123)	(3, 3)
(-122)	(-1, -1, -1)
(-121)	(1, 1, 1)

Table 6.9 Run-length coding of short runs of highly probable symbols.

Run-length Coded Symbol	Original Symbol
(118, SS, S(n-1))	(S) x n, where S = -2048..2047, n = 1..16
(-120, (n-2)S)	(S) x n, where S = -16..15, n = 2..9
(-119, S, (n-1))	(S) x n, where S = -128..127, n = 1..256
(-118, n, n, nS, S)	(S) x n, where S = -2048..2047, n = 1..1048677

Table 6.10 Run-length codes for large runs.

Runs formed with the zero symbol are coded with the run-length codes shown in Table 6.8. Small runs from 2 to 9 symbols in length formed with the zero symbol can be represented by the symbols 120 to 127 respectively. Larger runs are represented by the two symbol code (119,n-10) where n represents the length of the run. This can represent runs between 10 and 265. Run-lengths greater than 265 must be represented by the -118 code as shown in Table 6.10. Small runs of symbols -3, -2, -1, 1, 2 and 3 are also represented by a single symbol as shown in Table 6.9.

Table 6.10 contains the remaining codes for coding large run-lengths. Note there are a number of variations to optimise the coding for various combinations of symbol and run-length. The symbol 118 is used to represent symbols lost in defining all of the other symbol codings and those symbols which are too large to represent within the eight bit constraint of the run-length coder symbol.

The entire run-length coder and decoder can be found in Appendix D. The run-length coder function prototype is defined in the C programming language as follows:

*DWORD RunLengthCode (int *DCT, char *RunLength, DWORD ImageSize)*

The function takes as parameters a pointer *DCT* to the transformed and quantised image data, a pointer *RunLength* to a buffer where the coded image data is to be stored and *ImageSize* which contains the number of pixels in the image. The function returns the size of the run-length coded output stored in the *RunLength* buffer.

The run-length decoder function prototype is similarly defined in the C programming language as follows:

*DWORD RunLengthDecode (char *RunLength, int *DCT, DWORD CodeSize)*

In the decoder the parameters are the reverse of the coder. The pointer *RunLength* points to the buffer where the run-length coded image is stored, the pointer *DCT* points to the buffer where the result of the decoder is to be stored and *CodeSize* contains the number of symbols stored in the run-length buffer.

6.7 Results

6.7.1 Timing Benchmarks

The entire transformation, quantisation and run-length coding process was timed to check its execution speed. The full compression process required 5.25 seconds to complete for a *512x512* pixel image. The decompression process required 5.4 seconds to complete for the same size image.

6.7.2 Entropy Effects

The effect on the image entropy after the run-length coding is shown in Table 6.11. From this table it can be seen that there are improvements gained through the run-length coding for all of the images. In some of the images the improvements are quite dramatic. On average, there is a 15% improvement in the final compressed image size after run-length coding when compared to the possible final compressed image size immediately after quantisation only.

Image File	Run Length Code File Size	Entropy After Run-Length Coding	Image Size After Quantisation[1]	Compressed Image Size[1]
Airplane.Y	99484	5.038965	74642	62663
Baboon.Y	152203	5.244405	112543	99777
Beans1.Y	11669	5.476905	9958	7989
Beans2.Y	16192	5.500693	13594	11134
Couple.Y	26058	4.959877	18546	16156
Girl1.Y	26158	4.950389	18529	15187
Girl2.Y	19471	4.841252	13133	11784
Girl3.Y	24666	4.861619	17524	14990
House.Y	24112	4.905119	17641	14785
Lena.Y	102873	4.848337	73673	62346
Peppers.Y	109386	4.829962	76995	66042
Sailboat.Y	123101	5.119864	90774	78783
Splash.Y	94102	4.572128	64959	53781
Testpatt.Y	147953	5.821420	125062	107663
Tiffany.Y	107231	4.775544	72747	64011
Tree.Y	33373	5.346665	25449	22305
Wendy1.Y	72027	4.633005	48252	41713
Wendy2.Y	94084	5.225760	74943	61458
Wendy3.Y	97471	4.775387	71440	58183

Table 6.11 Entropy of image data after run-length coding.

6.8 Conclusion to the Chapter

The results indicate that the run-length coder improved the amount by which the image could be compressed. It also indicated that there is still a requirement for a statistical coder to further reduce the size of the compressed image. This is obvious because the entropies shown in Table 6.11 are lower than the number of bits currently used to represent each symbol, that is, each entropy value is less than 8 bits in size. The statistical coder will be more closely examined in the following chapter.

[1] The image size after quantisation is based on the entropy obtained immediately after quantisation as obtained in section 5.7.4 using equation (5.1). The compressed image size is based on the entropy obtained after run-length coding the image if it was to be passed through a statistical coder. The resultant size is once again based on equation (5.1).

7. The Statistical Coder

7.1 Introduction

The statistical coder forms the fourth and final functional unit in the OptIC algorithm and is the second layer of compression. The prime purpose of an ideal statistical coder is to compress a set of data to the theoretical size possible as determined by the entropy of the data. The relation between the entropy of a data set and the possible compressed size of the data was shown in (5.1) in chapter 5.

It is advantageous to use statistical coder wherever the size of the data symbol is greater than the entropy of the entire data set, the greater the difference the better the results after compression. In the results obtained for the run-length coder the data symbols were eight bits in size. The entropies shown in Table 6.4 suggest that a statistical coder would provide substantial improvements in the size of the image data.

Statistical coders have been available since the work of Shannon [SHA48] and have been extensively researched. There are now a very wide range of coders available, providing different compromises between computational speed, algorithm complexity and compression efficiency. The purpose of this chapter is to describe and implement two different statistical coders and test their computational speed and efficiency when combined with the current image compression algorithm.

Before discussing the coders it is useful to look at the image statistics obtained after the run-length coding and try to visualise how a statistical coder can compress this

data further. The frequency of all eight bit symbols from the run-length coded intensity images was measured and graphed in Fig. 7-1.

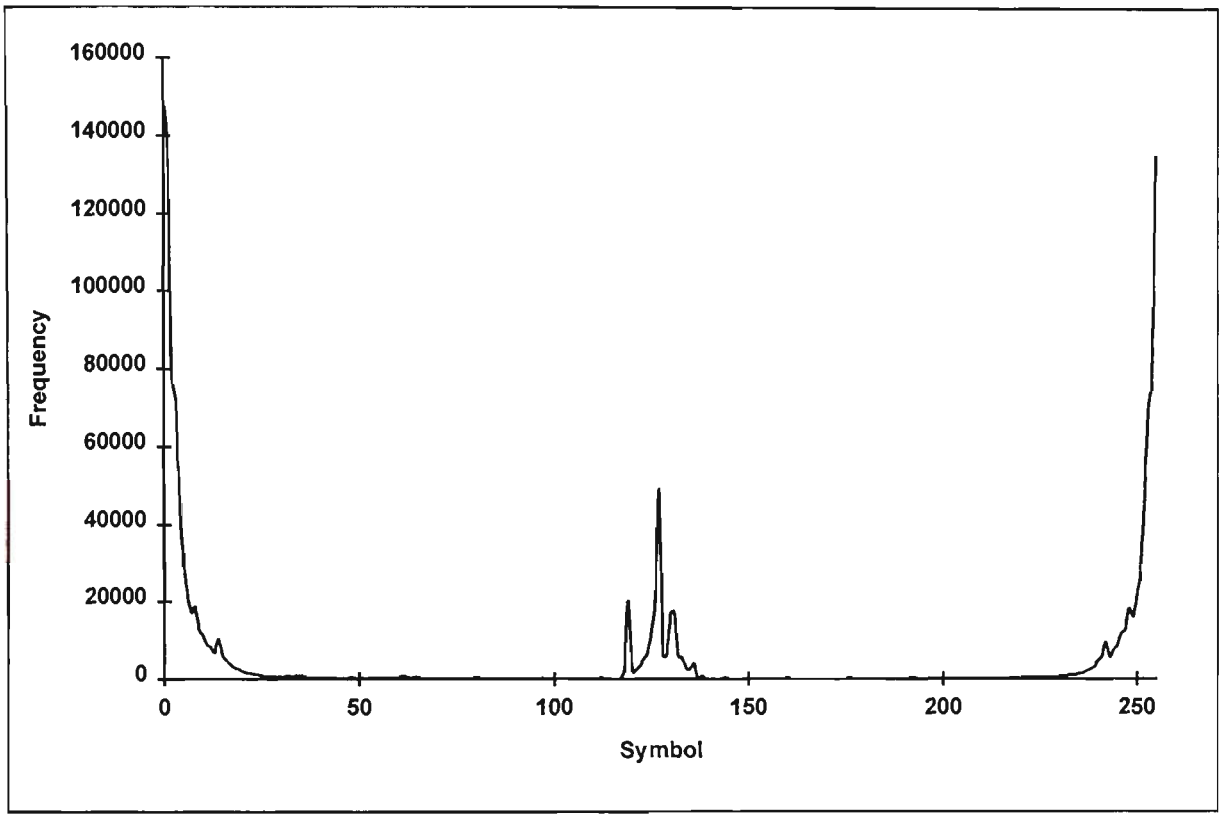


Fig. 7-1 Symbols frequencies for the run-length coded images.

From Fig. 7-1 it can be seen that the images are mainly composed of symbols in the extremes of the possible range of symbol values, that is, most symbols are close to zero or close to 255. These are all the zero and -1 symbols that did not form runs. There is also a peak at the centre of the symbol range, these are the symbols that were used to code all of the runs during the run-length coding process.

The basis for statistical coding is rather simple. Fundamentally, it operates by replace the more probable symbols with short symbol codes and the less probable symbols with larger symbol codes. The coders that will be investigated here are the Huffman

coder and the Arithmetic Coder. The Huffman Coder is a very well known algorithm that is a computationally fast algorithm but not as efficient as the arithmetic coder which is more complex and computationally slower.

The two coders were implemented using both fixed and adaptive models. A fixed model uses a fixed set of statistics that approximates the statistics of the input data. If the statistics of the data change then the fixed coder will not perform very well. An adaptive model measures the statistics of the data during the coding process and can thus adapt to changes in statistics and provide better results than that of the fixed model. In general, the adaptive model is algorithmically more complex than the fixed model so there is generally a compromise between the efficiency of the algorithm and its computational speed.

The two coders were also be compared against commercially available coders such as *ARJ*, *LHARC* and *PKZIP*.

7.2 The Huffman Coder

The Huffman coder was originally described in a paper by Huffman [HUF52]. This paper describes the method by which the Huffman codes are generated for all of the symbols dependent on their probability of occurrence.

The first step in coding sequence is to sort all of the probabilities of the symbols from highest to lowest probability. The two symbols of least probability of occurrence are taken out of the sorted list, their probabilities are summed and are then re-inserted into

the list. This is repeated for the next two least probable symbols and continued until all of the symbols in the list have been summed into a single value. The number of times that a symbol is taken out of the list and summed indicates the size of the code that the symbol will be represented by. The code itself is defined by adding a zero to the code if the symbol has the second lowest probability and a one if the symbol has the lowest probability. This is procedure is more simply understood when shown diagrammatically as in Fig 7-2. The procedure is for a symbol set containing 13 unique symbols.

As an example the procedure will be followed to determine the Huffman code for the symbol marked with a '*'. The procedure is followed step-wise in Table 7.1. As the symbol is taken out of the sorted list 5 times, that is, in stages 2, 4, 7, 10 and 11 the final Huffman code has 5 binary digits.

Stage	Probability	Sum	Probability Position	New Code
2	0.04	0.08	Second Last	0
4	0.08	0.14	Second Last	00
7	0.14	0.24	Second Last	000
10	0.24	0.60	Last	1000
11	0.60	1.00	Second Last	01000

Table 7.1 Step by step definition of a Huffman code.

After coding all the symbols in Fig 7-2 it is possible to create a symbol to Huffman code conversion table, see Table 7.2.

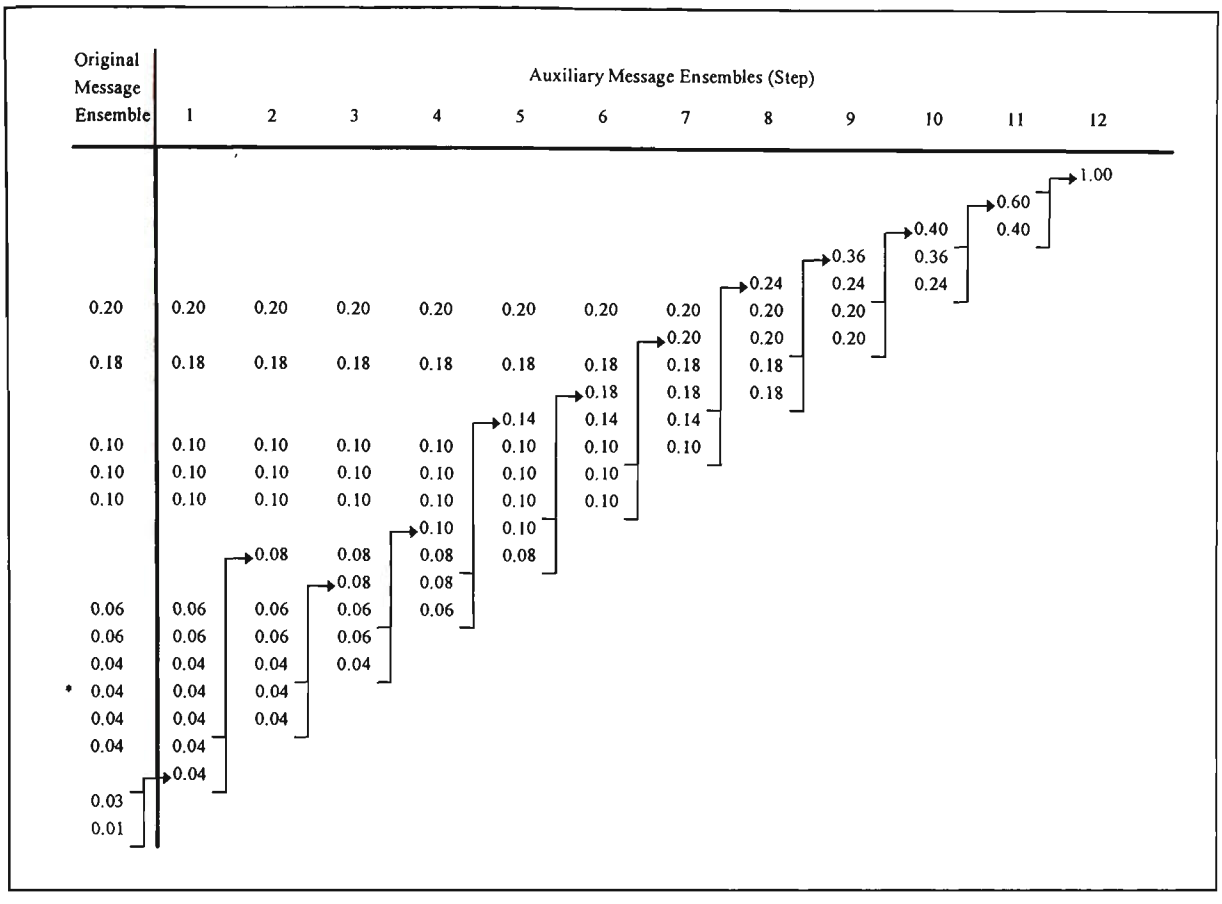


Fig. 7-2 A sample Huffman coding process[HUF52].

Symbol	P(symbol)	Code
1	0.20	10
2	0.18	000
3	0.10	011
4	0.10	110
5	0.10	111
6	0.06	0101
7	0.06	00100
8	0.04	00101
9	0.04	01000
10	0.04	01001
11	0.04	00110
12	0.03	001110
13	0.01	001111

Table 7.2 Huffman codes for the sample symbols.

7.2.1 The Fixed Huffman Coder

The code for the fixed Huffman coder and decoder can be found in Appendix D. The fixed Huffman coder assumes a static probability distribution for the image set. The frequency distribution shown in Fig. 7-1 was used to create a probability distribution and thus provide the data required to create the Huffman codes using the procedure outlined in section 7.2. The codes were stored in the constant array *sym* to allow for a direct mapping between the original symbol code and the Huffman code used to represent this code. As an array contains elements which are all of the same length it was necessary to create another array which contained the Huffman codes lengths, this array was stored in the variable *len*.

The fixed Huffman coding function prototype is defined in the C programming language as follows:

*DWORD StatisticalCode (char *source, char *dest, DWORD length)*

The pointer *source* points to the data to be coded and the pointer *dest* points to the buffer where the Huffman coded data will be stored while *length* indicates the length of the source buffer. On return the function returns the length of the coded data.

The coder operates by simply reading a code from the source buffer, mapping that code to a Huffman code via the array *sym* and then storing the code in the destination buffer. The array *len* is also examined to determine how many bits actually need to be

stored. Note that the Huffman code can be between 3 to 15 bits in length in this implementation.

The fixed Huffman decoder function prototype is defined in the C programming language as follows:

*DWORD StatisticalDecode (char *source, char *dest, DWORD length)*

The parameters are basically the same as for the coder but the functionality is now reversed. The pointer *source* points to the data to be decoded and the pointer *dest* points to the buffer where the decoded data will be stored and *length* indicates the length of the source buffer. On return the function returns the length of the decoded data.

The decoder reads information from the source buffer on a bit by bit basis and attempts to recognise a valid code. This is slightly more complicated than the coder in that the code value and the code length must both be used to determine if the code is indeed a valid code. For example, the codes 000 and 0000 are different codes even though both their values are zero. To perform this code validation the two dimensional array *convert* is used which has indices formed by the current code and current code length. The contents of the array contain a zero if the code is invalid and the original symbol value if the code is valid. The zero symbol is treated separately as a special case to avoid problems in the decoding. If the current code is invalid then another bit is read from the source buffer and the process is repeated until a valid code is formed.

7.2.2 The Adaptive Huffman Coder

In the quantisation and ordering process the like coefficients of the DCT were grouped together. In general the characteristics of one type of coefficient will differ from another type of coefficient implying that their statistical characteristics would also differ. It would, therefore, be advantageous to create a coder that could adapt to changes in the data statistics to provide the best coding method at that instant.

The adaptive Huffman coder in appendix D is a modified version of the fixed Huffman coder. Its C function prototype is identical to that of the fixed Huffman coder. The coder works by keeping a record of the frequency of occurrence of each symbol which is initially set to the statistics shown in Fig. 7-1. As each symbol is read from the input source this frequency distribution is correspondingly updated. An upper bound is also set for the frequency of any symbol. When this bound is reached all of the frequencies are scaled down by a factor of two. The purpose of this is to introduce a 'forgetting' functionality, thus improving the response of the coder to changes in the data statistics.

The frequencies in the adaptive coder are always maintained in descending order in the array *freq* and the list of Huffman codes are stored in the array *sym* in increasing length size. The corresponding length of the code is stored in the array *len*. As the frequencies are constantly shuffled around as the statistics of the source data change, it is necessary to create two arrays *char_to_index* and *index_to_char*. These provide a link the between the symbol value and an index which points to the appropriate

element in the frequency, Huffman code and Huffman code length arrays. As a symbol becomes more frequent its index begins to move up through the frequency list and so its coded value will become shorter in length. Similarly as a symbol becomes less frequent it will be moved down the frequency list and be associated with a longer code.

7.3 The Arithmetic Coder

The arithmetic coder, though still a statistical coder, takes a completely different approach to coding the symbol set. Both the fixed and adaptive versions of the adaptive coder are based on the coder described in [WIT87]. The principle operation of the arithmetic coder is most easily explained by an example and diagrammatically as in Fig. 7-3. In this example the probability distribution shown in Table 7.3 is assumed. The symbol set is composed of the six symbols {A, E, I, O, U, !} where the symbol *!* is used to represent the end of the sequence.

Symbol	Probability
A	0.2
E	0.3
I	0.1
O	0.2
U	0.1
!	0.1

Table 7.3 A sample probability distribution.

The arithmetic coder begins with an initial range, typically [0, 1) which represents the interval $0 \leq x < 1$. This range is split into sub-ranges, one for each symbol, that are proportional in size to the probability of the symbol it represents. Using the

probabilities of the symbols listed in Table 7.3, the range can be split up as shown in Fig. 7-3. In this example the symbol *A* is represented by the range [0, 0.2), symbol *E* by [0.5, 0.5), symbol *I* by [0.5, 0.6) and so on.

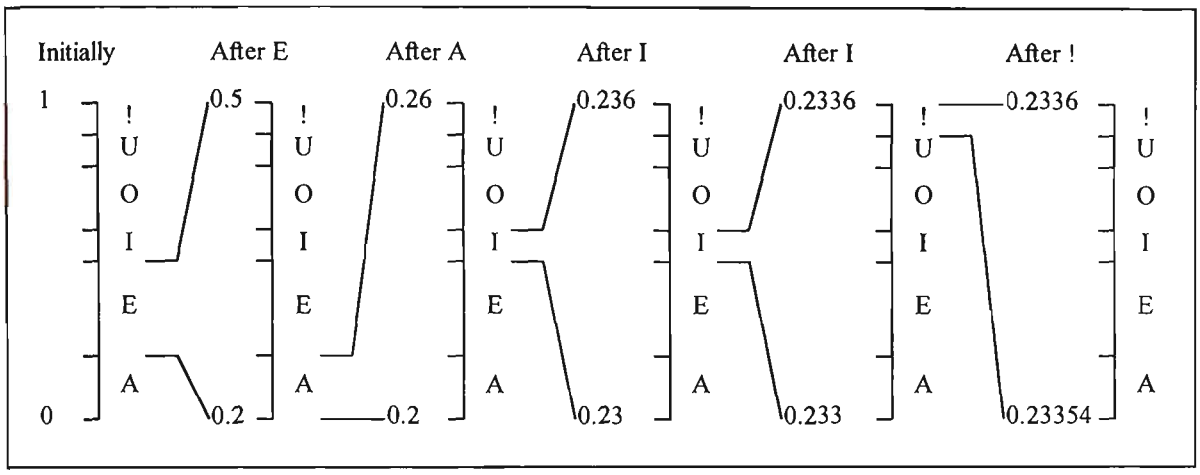


Fig. 7-3 Arithmetic coded example for the sequence {E, A, I, I, !} [WIT87].

As a symbol is read from the input source the sub-range for that symbol is taken as the new range. The new range is then again split into sub-ranges representing all of the possible symbols where the size of each sub-range is proportional to the probability of the symbol it represents. This subdivision is continued until all of the symbols in the input source have been coded. Referring to the example it can be seen that the initial range is [0, 1). After reading the symbol *E* the range is narrowed to [0.2, 0.5). This new range is now split into sub-ranges in preparation for the next symbol. Thus if the next symbol were to be an *A* the range would become [0.2, 0.26), an *E* would give a range [0.26, 0.35), an *I* would give a range of [0.35, 0.38) and so on. Fig. 7-3 shows how the range is narrowed when the sequence {*E*, *A*, *I*, *I*, *!*} is coded using the arithmetic coder. By storing one of the values in the final range after coding it is

possible to represent the entire sequence. Thus 0.23354, 0.23355 and even 0.2335455 can be used to represent the sequence {*E*, *A*, *I*, *I*, *!*} when coded.

The decoding process is similar to the encoding process. The decoder begins with the initial range [0, 1) and creates the sub-ranges as with the encoder. It then determines in which sub-range the coded values lies within. Using the code generated in the coding example, 0.23354, it is clearly seen that this lies within the sub-range [0.2, 0.5) which represents the symbol *E*. As each symbol is decoded the sub-range for that symbol then becomes the new range and is divided into a new set of sub-ranges as for the encoder. So the new range after decoding the symbol E becomes [0.2, 0.5) and its sub-ranges become [0.2, 0.26) for A, [0.26, 0.35) for E, [0.35, 0.38] for I, etc. From the new set of sub-ranges it can be seen that the coded symbol set, 0.23354, lies within the sub-range for *A*. It can thus be deduced that the next symbol in the sequence is *A* and the new range will become [0.2, 0.26), the sub-range for the *A* symbol. This procedure is repeated until the final symbol is decoded, represented by the *!* symbol.

It can be noted that for large sequences and for large symbol spaces a great deal of precision is required to represent the final code, making the arithmetic coder inherently complex. Also the action of creating sub-ranges requires multiplications and divisions, and these operation greatly increase the processing time. There does exist arithmetic coders which do not require any multiplications or divisions

[CHE91]. These are, however, only approximations and as such they do not perform as well as a true arithmetic coder.

7.3.1 Fixed

The C source listing of this coder can be found in Appendix D. The fixed arithmetic coder has a static record of the frequency distribution of the symbols built into the coder. These statistics are the same as those shown in Fig. 7-1 except that they are scaled down to keep the cumulative frequency below 16383 which is a limit for this implementation of the arithmetic coder. The frequency information is stored in the array *freq*.

Rather than using extremely large floating point values, which would be impractically slow, the algorithm performs all calculations using integer arithmetic and uses an incremental transmission method to avoid the need of high precision arithmetic. The incremental transmission method works by storing bits which will no longer change and keep only those bits which may change as a result of a new sub-range.

Using the sub-ranges formed while coding the sequence {*E*, *A*, *I*, *I*, *!*} in Fig. 7-3, it can be seen that after coding symbol *A* the range is reduced to [0.2, 0.26). No matter how much further this range is reduced nothing can change the first digit after the decimal point, it will always remain a 2. It is thus possible to store this digit. Similarly, after the first *I* symbol is coded the range becomes [0.23, 0.236), here the second digit also becomes static and may be stored.

To reduce the precision required and to simplify the calculations required to perform the arithmetic coding it is also necessary to scale up the range after the static digits have been stored. In the example the range could be increased by a factor of ten after the removal of each digit, so the range [0.2, 0.26) would become [0, 0.06) after the removal of the first digit and then [0, 0.6) after the range has been expanded.

The C function prototype for the fixed arithmetic coder is identical to that of the Huffman coders. The coder differs from the original algorithm described in [WIT87] in that the coding is performed in memory rather than using file operations in order to improve its performance.

7.3.2 Adaptive

The software listing for the adaptive coder can be found in Appendix D. This algorithm is based on the algorithm described in [WIT87]. The base of the adaptive arithmetic coder is identical to that of the fixed coder. It only differs in that the data statistics are measured during coding using methods similar to that used in the adaptive Huffman coder described in section 7.2.2.

The C function prototype for the adaptive arithmetic coder is identical to that of the Huffman coder.

7.4 Results

7.4.1 Image Size

The fixed Huffman, adaptive Huffman, fixed arithmetic and adaptive arithmetic statistical coders were used to compress the output of the run-length coder. The image sizes after compression are shown in Table 7.4. From this table, it can be seen that the adaptive arithmetic coder provided the best results on average. It was the only coder that in some cases produced results better than was predicted from the original entropy of the run-length data. The second most efficient coder was the adaptive Huffman coder.

Image	Compressed Image Size[1]	Fixed Huffman	Adaptive Huffman	Fixed Arithmetic	Adaptive Arithmetic
Airplane.Y	62663	63281	62984	62826	62430
Baboon.Y	99777	102325	101041	101368	98954
Beans1.Y	7989	8299	8193	8236	8108
Beans2.Y	11134	11471	11392	11361	11289
Couple.Y	16156	16390	16324	16289	16328
Girl1.Y	15187	16389	16349	16282	16342
Girl2.Y	11784	12211	11981	12095	11919
Girl3.Y	14990	15244	15202	15153	15155
House.Y	14785	14977	14936	14910	14940
Lena.Y	62346	63261	62971	62975	62375
Peppers.Y	66042	66870	66481	66682	65795
Sailboat.Y	78783	79745	79251	79059	78150
Splash.Y	53781	55327	54693	55250	53605
Testpatt.Y	107663	117601	112805	116331	106414
Tiffany.Y	64011	65145	64287	64824	63762
Tree.Y	22305	22803	22665	22576	22406
Wendy1.Y	41713	43967	42347	43746	41530
Wendy2.Y	61458	62256	61638	61769	60718
Wendy3.Y	58183	59004	58424	58790	57497

Table 7.4 Image sizes after compression using various statistical coders.

[1] The compressed image size is based on the entropy of the output from the run length coder.

Various commercial statistical coders were also tested to compare with the Huffman and arithmetic coders. The image size after compression using the ARJ, LZH and ZIP coders is shown in Table 7.5 for all of the intensity images in the test set.

When the results in Table 7.5 are compared with those in Table 7.4 it can be seen that the commercial coders are out-performed by all of the Huffman and arithmetic coders. The commercial coders perform well in only one image, *Testpatt.Y*.

Image	Compressed Image Size[2]	ARJ	LZH	ZIP
Airplane.Y	62663	66174	65588	66067
Baboon.Y	99777	104854	103843	104753
Beans1.Y	7989	8502	8393	8489
Beans2.Y	11134	11950	11763	11921
Couple.Y	16156	17494	17238	17547
Girl1.Y	15187	17492	17231	17524
Girl2.Y	11784	12791	12581	12751
Girl3.Y	14990	16359	16145	16404
House.Y	14785	16202	15995	16284
Lena.Y	62346	66827	66311	66727
Peppers.Y	66042	70721	70067	70487
Sailboat.Y	78783	82837	81884	82638
Splash.Y	53781	57728	57305	57475
Testpatt.Y	107663	105371	104609	105427
Tiffany.Y	64011	68021	67366	67835
Tree.Y	22305	23671	23322	23671
Wendy1.Y	41713	44067	43684	43827
Wendy2.Y	61458	64305	63661	64230
Wendy3.Y	58183	61707	61193	61571

Table 7.5 Image sizes after compression.

[2] The compressed image size is based on the entropy of the output from the run length coder.

7.4.2 Timing Benchmarks

Timing benchmarks were made for the full compression of each image using the fixed Huffman, adaptive Huffman, fixed arithmetic and adaptive arithmetic coders. The results of these benchmarks are shown in Table 7.6. These benchmarks include the time taken to perform the transformation, quantisation, ordering, run-length coding and statistical coding.

Image	Fixed Huffman	Adaptive Huffman	Fixed Arithmetic	Adaptive Arithmetic
Airplane.Y	6.1	6.4	15.1	17.5
Baboon.Y	6.9	7.4	21.1	24.2
Beans1.Y	1.3	1.4	2.5	3.0
Beans2.Y	1.4	1.4	2.9	3.5
Couple.Y	1.5	1.6	3.9	4.6
Girl1.Y	1.6	1.6	3.9	4.6
Girl2.Y	1.4	1.5	3.2	3.9
Girl3.Y	1.5	1.6	3.7	4.4
House.Y	1.5	1.6	3.7	4.4
Lena.Y	6.2	6.5	15.3	17.7
Peppers.Y	6.2	6.6	15.9	18.3
Sailboat.Y	6.5	6.9	17.7	20.5
Splash.Y	6.0	6.3	14.3	16.3
Testpatt.Y	6.9	7.4	21.7	24.8
Tiffany.Y	6.2	6.5	15.7	18
Tree.Y	1.7	1.7	4.8	5.7
Wendy1.Y	5.7	5.9	12.0	13.7
Wendy2.Y	6.0	6.3	14.7	16.8
Wendy3.Y	6.0	6.4	14.7	16.8

Table 7.6 Times for the full compression of the images using various statistical coders.

From Table 7.6 it can be seen that even though the arithmetic coder performed well in terms of its coding efficiency, it did not perform well in the timing benchmark. This is due to the large number of multiplications required. The fastest of the four coders was

the fixed Huffman coder with the adaptive Huffman coder coming a close second. Note that the time for compression is closely linked to the size of the data being compressed and so varies with the different images.

The timing benchmarks were also made for the decompression cycle of all of the images using the same coders. The results of these tests are shown in Table 7.7. These benchmarks include the time taken to perform the statistical decoding, run-length decoding, re-ordering, de-quantisation and inverse transformation.

Image	Fixed Huffman	Adaptive Huffman	Fixed Arithmetic	Adaptive Arithmetic
Airplane.Y	6.4	7.7	25.8	21.4
Baboon.Y	7.4	9.4	37.7	30.0
Beans1.Y	1.4	1.6	3.6	3.4
Beans2.Y	1.5	1.7	4.7	4.2
Couple.Y	1.6	2.0	6.7	5.6
Girl1.Y	1.6	2.0	6.7	5.6
Girl2.Y	1.5	1.8	5.2	4.5
Girl3.Y	1.6	1.9	6.4	5.4
House.Y	1.6	1.9	6.2	5.3
Lena.Y	6.4	7.7	26.3	21.7
Peppers.Y	6.5	7.9	27.6	22.4
Sailboat.Y	6.8	8.4	31.0	25.0
Splash.Y	6.2	7.4	24.2	19.7
Testpatt.Y	7.6	9.8	37.7	31.1
Tiffany.Y	6.5	7.8	27.0	22.0
Tree.Y	1.8	2.3	8.4	7.0
Wendy1.Y	5.9	6.8	19.4	16.3
Wendy2.Y	6.3	7.6	24.9	20.6
Wendy3.Y	6.3	7.5	25.1	20.5

Table 7.7 Times for the full decompression of the images using various statistical coders.

Once again the fixed Huffman coder is the fastest of the four coders and the adaptive Huffman came a close second. It is interesting to note that in the decompression cycle

the fixed arithmetic coder was slower than the adaptive arithmetic coder even though algorithmically it is simpler. The reason for this unexpected result is that the speed of the arithmetic coders is highly dependent on how differently the statistics of the image data differs from the statistics stored within the coder itself. In the case of the adaptive arithmetic coder the statistics within the coder are constantly modified to approximate the current statistics of the image and so improving its performance. The fixed arithmetic coder, however, has a fixed set of statistics and will not adjust itself when the image statistics deviate from this set, thus its performance is reduced greatly under these conditions.

7.4.3 Entropy Effects

It is useful to measure the entropy of the compressed images to examine if the statistical coders have compressed the images as fully as possible. The entropies of all the images when compressed using the four coders described are shown in Table 7.8.

In all cases the entropy was extremely close to 8 which is also the number of bits used to store the symbols. This indicates that it is not possible to further compress the image using only a statistical coder.

Image	Fixed Huffman	Adaptive Huffman	Fixed Arithmetic	Adaptive Arithmetic
Airplane.Y	7.996891	7.972004	7.996598	7.933456
Baboon.Y	7.997963	7.974088	7.998123	7.933871
Beans1.Y	7.977401	7.954711	7.973617	7.912978
Beans2.Y	7.984922	7.961536	7.982749	7.922467
Couple.Y	7.988093	7.961451	7.988116	7.927724
Girl1.Y	7.989398	7.954440	7.988587	7.923846
Girl2.Y	7.984555	7.944551	7.987139	7.878017
Girl3.Y	7.988747	7.952564	7.987257	7.913958
House.Y	7.988116	7.959225	7.987530	7.928264
Lena.Y	7.996942	7.960810	7.996957	7.932413
Peppers.Y	7.997294	7.971006	7.997393	7.936137
Sailboat.Y	7.997479	7.978558	7.997671	7.935063
Splash.Y	7.996948	7.942108	7.996639	7.903789
Testpatt.Y	7.998079	7.937926	7.998382	7.872324
Tiffany.Y	7.996413	7.957128	7.996956	7.919576
Tree.Y	7.991157	7.965613	7.991533	7.925362
Wendy1.Y	7.995791	7.940319	7.996129	7.886421
Wendy2.Y	7.996740	7.977821	7.997060	7.938452
Wendy3.Y	7.997233	7.968289	7.996465	7.919862

Table 7.8 Image entropies after compression.

7.5 Conclusion

The results indicate that a compromise exists between the processing speed of an algorithm and the level of compression obtainable by it. In terms of compression, the adaptive arithmetic coder and the adaptive Huffman coder proved to be the highest performers. In terms of processing speed, the two Huffman coders proved to be a great deal faster than the arithmetic coders. Overall it appears that the adaptive Huffman coder provides the best compromise between speed and efficiency. It performed only marginally slower than the fixed Huffman coder and was only slightly less efficient than the adaptive arithmetic coder.

8. Algorithm Performance

8.1 Introduction

It is important to compare the OptIC algorithm designed in this thesis with other available algorithms to provide a true analysis of its performance. Two common algorithms were used for this comparison, these are the DPCM and JPEG compression algorithms. The features that will be compared include the level of compression obtained, the level of MSE introduced and the compression time. All of the comparisons are made with respect to the adaptive Huffman version of the OptIC algorithm described in chapter 7.

The DPCM algorithm used in these tests is similar to that used in the lossless JPEG algorithm [PEN90, RAO90]. It consists of a simple one-dimensional predictor and an arithmetic coder. The DPCM algorithm is *lossless* and so does not introduce any distortion into the reconstructed image.

The JPEG algorithm is an implementation of the JPEG algorithm by the *Independent JPEG Group* and was written by Thomas G. Lane (this is a shareware product available through the internet). This implementation is made up of two executable files *CJPEG.EXE* (version 4.0) and *DJPEG.EXE* (version 4.0) which form the compression and decompression algorithms respectively, both run in an MS-DOS environment. Lane's implementation of the JPEG algorithm also incorporates a quality factor ranging from extremely poor quality (0%) to extremely high quality

(100%). Through experimentation, it was found that a quality level of 92% gave results that were indistinguishable from the original images, in all of the images in the test set. This was the quality level used in the comparison tests.

8.2 Compressed Image Size

The results in Table 8.1 show the sizes of all of the intensity images after compression using the DPCM, JPEG and new algorithm. All of the sizes are shown in 8 bit bytes.

Image	Original Image Size	DPCM	JPEG	Algorithm Results
Airplane.Y	262144	149135	65766	62984
Baboon.Y	262144	206682	123351	101041
Beans1.Y	65536	25798	9209	8193
Beans2.Y	65536	29904	11991	11392
Couple.Y	65536	38270	16252	16324
Girl1.Y	65536	39411	16769	16349
Girl2.Y	65536	30180	11155	11981
Girl3.Y	65536	39413	15784	15202
House.Y	65536	36317	16192	14936
Lena.Y	262144	165637	67976	62971
Peppers.Y	262144	167289	76421	66481
Sailboat.Y	262144	183537	93794	79251
Splash.Y	262144	137319	55297	54693
Testpatt.Y	262144	37774	126534	112805
Tiffany.Y	262144	157594	66063	64287
Tree.Y	65536	44902	25621	22665
Wendy1.Y	262144	120696	41743	42347
Wendy2.Y	262144	146345	66108	61638
Wendy3.Y	262144	140626	61302	58424

Table 8.1 A comparison of intensity image sizes after compression with DPCM, JPEG and the new algorithm.

From Table 8.1 it can be seen that the OptIC algorithm is the best performer in terms of the level of compression obtained. On average, it gives a compression factor of 4.38 compared to 3.93 for the JPEG algorithm and 1.9 for the DPCM algorithm.

The OptIC algorithm performed better than the JPEG algorithm in all but three images: *Couple.Y*, *Girl2.Y* and *Wendy3.Y*. When compared to the DPCM algorithm, the OptIC algorithm was only out-performed with the image *Testpatt.Y*. This image, however, is an artificially generated image and does not exhibit characteristics which are typical with *natural* images.

The tests were also performed on the green component of those images which were originally represented in colour. The results of these tests can be found in Table 8.2.

Image	Original Image Size	DPCM	JPEG	Algorithm Results
Airplane.G	262144	154213	69978	66227
Baboon.G	262144	212894	130579	106455
Beans1.G	65536	28115	10178	9675
Beans2.G	65536	32084	13023	12783
Couple.G	65536	38513	16784	16769
Girl1.G	65536	39437	17554	16950
Girl2.G	65536	32071	12014	12896
Girl3.G	65536	40934	17263	16276
House.G	65536	38043	17360	15889
Lena.G	262144	174577	78428	69379
Peppers.G	262144	173967	84268	73162
Sailboat.G	262144	195347	105782	86792
Splash.G	262144	149003	66445	62588
Tiffany.G	262144	171872	81075	73130
Tree.G	65536	45797	26808	23753
Wendy2.G	262144	144731	66748	62276
Wendy3.G	262144	119575	63898	59701

Table 8.2 A comparison of green image sizes after compression with DPCM, JPEG and the new algorithm.

The results shown in Table 8.2 for the green component images do not differ greatly from those of the intensity image results. The average compression factor for the DPCM algorithm is 1.71, 3.79 for the JPEG algorithm and 4.05 for the OptIC algorithm. Once again the OptIC algorithm out-performs the other algorithms.

8.3 MSE Levels Introduced By the Algorithms

The MSE introduced after reconstruction using the DPCM, JPEG and OptIC algorithm for all of the intensity images is shown in Table 8.3. As described in section 3.5.1, each line in Table 8.3 represents the MSE for each image at the extreme just before errors became noticable, therefore, at this point, with respect to what the human eye can perceive, all of the reconstructed images appear identical to the original images. As such, the MSE, in this situation, does not represent the visible level of distortion introduced into the image by the various algorithms. Instead it gives an indication of the mean opinion score (MOS) and, therefore, the degree to which the HVS has been incorporated into the algorithm.

Image	DPCM	JPEG	Algorithm Results
Airplane.Y	0	3.438511	5.549461
Baboon.Y	0	8.885120	35.092472
Beans1.Y	0	1.400757	2.166992
Beans2.Y	0	1.887634	3.110580
Couple.Y	0	3.406158	6.557709
Girl1.Y	0	4.122482	6.747757
Girl2.Y	0	2.563141	4.548828
Girl3.Y	0	3.289993	5.282883
House.Y	0	3.506980	6.971359
Lena.Y	0	4.604576	6.442287
Peppers.Y	0	6.831085	11.092140
Sailboat.Y	0	7.869686	16.493061
Splash.Y	0	3.219627	3.998802
Tiffany.Y	0	2.292976	28.986454
Tree.Y	0	4.489658	12.564140
Testpatt.Y	0	7.117462	22.768616
Wendy1.Y	0	1.987961	2.161739
Wendy2.Y	0	2.382908	4.058239
Wendy3.Y	0	2.882843	3.511997

Table 8.3 MSE introduced after reconstruction of intensity images using DPCM, JPEG and the new algorithm.

From Table 8.3 it can be seen that the DPCM does not incorporate the HVS at all. It can also be seen that the OptIC algorithm incorporates the HVS more effectively than the JPEG algorithm, as indicated by the higher MSE results.

The same tests were also applied to the green component of all of the images that were originally in colour. The results of these tests are shown in Table 8.4. These results do not differ greatly from those in Table 8.3 and the same conclusions may be drawn. That is, the OptIC algorithm incorporates the HVS to a greater degree than both DPCM and JPEG.

Image	DPCM	JPEG	Algorithm
Airplane.G	0	3.983284	6.829433
Baboon.G	0	9.089264	41.363766
Beans1.G	0	1.802444	2.977798
Beans2.G	0	2.328156	4.252090
Couple.G	0	3.940475	7.745407
Girl1.G	0	4.882935	8.511765
Girl2.G	0	3.118912	5.235733
Girl3.G	0	3.988754	6.581909
House.G	0	4.096909	8.461914
Lena.G	0	6.109669	9.337826
Peppers.G	0	7.393932	14.774143
Sailboat.G	0	8.557076	26.638569
Splash.G	0	4.626606	6.225773
Tiffany.G	0	6.209301	8.026783
Tree.G	0	7.296997	19.055832
Wendy2.G	0	2.710674	4.450462
Wendy3.G	0	3.525803	4.352993

Table 8.4 MSE introduced after reconstruction of green images using DPCM, JPEG and the new algorithm

8.4 Timing Benchmarks

It was found in chapter 7 that with the OptIC algorithm, the greatest amount of time required to process an image was 9.8 seconds with image *Testpatt.Y*. This was for the

decompression component of the algorithm. Similar tests were also run with the DPCM algorithm and Lane's implementation of the JPEG algorithm using the same test platform. Only the *512x512* intensity images were used in the tests. For the JPEG algorithm, *Testpatt.Y* also took the greatest time to process, 7.1 seconds in this case. In the DPCM case there was little difference between the processing times of the different images, all of them required approximately 38 seconds.

It is interesting to note that the DPCM algorithm is a great deal slower than both the JPEG algorithm and the OptIC algorithm. The reason for this is that the arithmetic coder used in the DPCM algorithm is a computationally intensive algorithm and as there are no other compression stages after the DPCM predictor, the coder is left to do all of the coding. In chapter 7 the OptIC algorithm was also tested using an arithmetic coder. It performed much better than the DPCM algorithm because it had two coding stages. The first coding stage, that is, the run-length coder, reduced the amount of data reaching the arithmetic coder and so in effect increasing the computational speed of the algorithm.

It was noted that the OptIC algorithm did not perform as quickly as the JPEG algorithm. The new algorithm was approximately 40% slower than the JPEG algorithm. One reason for this is the slow nature of the input and output port handling of the test system. As the DCT is hardware based it must pass all of the data through the test system's input and output ports. This procedure forms a bottleneck and reduces the performance of the algorithm as a whole. It should also be noted that

Lane's implementation of the JPEG algorithm runs under MS-DOS whereas the OptIC algorithm runs under the Windows environment. As such, Lane's software does not suffer from the large number of overheads associated with the Windows operating environment. One of the greatest of which is its memory management facility, this is used quite extensively by the OptIC algorithm to gain access to large blocks of memory. Possible improvements to the OptIC algorithm that may overcome these bottlenecks will be discussed further in Chapter 9.

9. Conclusions

9.1 Discussion of the Project

9.1.1 Project Aims

The project aims for this project were introduced in section 3.2.2. The OptIC algorithm satisfies all but one of these aims in that, on the hardware platform used for experimentation it could exceed the execution time limit of five seconds maximum. In chapter 7, it was found that the OptIC algorithm took on average 8 seconds to compress a 512x512 pixel image but the worst case was 9.8 seconds depending on the characteristics of the image. It should be noted that the tests were run on a 33MHz 486SX based machine without a floating point co-processor. This machine has now been superseded by the faster 100Mhz 486DX and the Pentium based machines, which give much higher levels of performance. The timing benchmarks could also be improved by rewriting some of the more time-critical components of the software in assembly language.

On average the new algorithm compressed images by a factor of four. Under no situations did this factor fall below two, as required by the project aims.

An optimal combination of hardware and software was proposed. The software provided flexibility to the algorithm whereas the hardware provided raw speed. The combination of hardware and software also gave a feasible low-cost high-speed solution. In this implementation of the algorithm the hardware was limited to

processing the DCT function. The combination of hardware and software was in most respects optimal, however, the 80486SX processor used in the test system did limit the full potential of the algorithm. These limitations lie in the architecture of the IBM PC, the PC/AT bus on to which the DCT hardware was attached was limited to an 8Mhz bandwidth regardless of the processor speed. This was overcome in later systems with the introduction of the local bus where the bus speed could be matched with that of the processor speed, this can give rise to a factor of four increase in bus bandwidth in the case of a 33Mhz processor. Other limitations exist due to the slow I/O data transfer instructions that exist on CISC processors such as the 80486SX, this could be overcome by using Direct Memory Access (DMA) techniques to transfer data to and from the DCT hardware at the same bandwidth as the bus.

This algorithm is ideal for use in parallel processing situations. An extremely large image can be divided into easy to handle sub-images. The sub-images would then be passed through a parallel processing network arranged in a grid formation. As the algorithm does not introduce any visible distortion into the reconstructed image, the sub-images can be handled completely independently of each other without any fear of blocking. Also, as there are no dependencies between the sub-images, no calculation bottlenecks are introduced and no inter-communication links are required in the parallel network. The result of this is a direct relationship between the execution time and the number of parallel processors introduced into the network. For example,

nine parallel processors will compute nine times faster than a single processor with the same size image.

9.1.2 Algorithm Disadvantages

There are two main disadvantages with the new algorithm. The first is the large memory requirement of the algorithm. The algorithm requires a buffer to hold the entire transformed image before it can begin to process it. This is because it needs to group all like coefficents, to do this all of the blocks must be transformed and stored before hand. Further memory is also required to store all of the look-up tables for the quantiser.

The second disadvantage with the algorithm is that it does not perform very well when implemented on CISC based processors. The reasons for this have already been discussed in chapter 4.

9.1.3 Algorithm Advantages

The algorithm requires only simple memory manipulation, addition and subtraction. No floating point multiplications or divisions are required by this implementation. All complex arithmetic is performed either in hardware or by look-up tables. The nature of the algorithm allows it to be easily adapted for DSP (Digital Signal Processing) and Reduced Instruction Set Computers (RISC) processors where all the CISC problems could be alleviated. This would greatly increase the performance of the algorithm.

The algorithm can be more easily adapted for progressive transmission than algorithms such as JPEG and DPCM. In progressive transmission, the quality of the image is improved progressively scan by scan until the required quality level is reached. This feature is useful if a sneak preview is required of an image, but one does not want to wait for the full quality image to be transmitted. This algorithm is ideal for this in that it groups all like coefficients together. One group of coefficients can be sent in each scan and the image subsequently reconstructed with all of the coefficient groups received at that point in time. Once all of the coefficients groups have been transmitted, the highest image quality level has been reached.

The JPEG algorithm is not well adapted to progressive transmission as it stores coefficients in block groups. That is, all the coefficients associated for one block are grouped together. For this reason, all of the coefficients need to be transferred before a full picture can be reconstructed.

9.2 Suggestions For Future Work

9.2.1 Adapting the Algorithm For a Different Platform

To overcome the numerous bottlenecks that exist on the 80486SX test system the algorithm could be ported to a system with a greater I/O bandwidth and with greater processing power. The simplest alternative would be to use a Pentium based system to test this algorithm. The Pentium processor is available in higher clock speeds than the 80486SX, has a more efficient instruction set and contains an in-built co-processor.

Based on Dhrystone MIPS [HAL95], this should give a at least a factor of six improvement in performance.

Even greater improvements could be made if the OptIC algorithm is ported to an Intel i860 or a TMS32040 DSP based system. These systems contain very fast instructions for manipulating memory and I/O, they also have extremely fast integer arithmetic instructions, e.g. multiply and divide.

9.2.2 Adapting the Algorithm For Motion Pictures

At present the algorithm is designed for still images. It is possible that further compression can be obtained by considering the time dimension. This could be done by incorporating some of the techniques described in section 2.4.

9.2.3 Adapting the Algorithm For Colour Images

The current algorithm has been optimised for black and white intensity images. It would also be useful to remove some of the redundancies apparent when dealing with colour images. Here some of the colour space transformation techniques described in section 2.5 could be introduced.

10. Bibliography

[AHM74] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete cosine transform", *IEEE Trans. Comput.*, vol. C-23, pp. 90-93, Jan. 1974.

[AMO89] H. Amor, D.Biere, A. Tescher, "Technical Issues in Low Rate Transform Coding", *Optical Engineering*, Vol. 28, No. 7, July 1989, 700-707

[ART88] A. Artieri, S. Kritter, F. Jutand, and N. Demassieux, "A one-chip VLSI for real time two-dimensional discrete cosine transform", *1988 Intl. Symp. on Circuits and Systems*, pp. 701-704, Helsinki, Finland, Jun. 1988.

[BAR88] M. Barnsley, A. Sloan, "A Better way to Compress Images", *Australian Personal Computing*, Feb. 1988, pp. 75-93

[CHA87] W. K. Cham, and Y. T. Chan, "Integer Discrete Cosine Transforms", *ISSPA 87, signal process., theories, implementations and appl.*, pp.674-676, 1987.

[CHE91] D. Chevion, E. D. Karnin, E. Walach, "High Efficiency, Multiplication Free Approximation of Arithmetic Coding", *IEEE*, pp.43-52, 1991.

[CLA85] R. J. Clarke, "Transform Coding of Images", *Academic Press*, pp.389-390, 1985

[COR90] I. Corbett, "Moving Pictures : Image Processing for Telecommunications", *IEE Review*, July/August, 1990, pp.257-261

[EKS84] M. P. Ekstrom, "Digital Image Processing Techniques", *Academic Press Inc.*, pp. 205-211, 1984.

[FAI95] J. Fairall, "Data Compression advances bit by bit", *Australian Electronics Monthly*, Vol. 28, No. 7, pp. 80-82, Jul., 1995.

[GIL95] B. Gillhoff, "Video Research first in Australia", *Australian Electronics Monthly*, Vol. 28, No. 7, pp. 56-57, Jul., 1995.

[GRU92] L. Grunin, "Something Lossed, Something Gained - Image Compression For PC Graphics", *PC Magazine*, pp. 337-350, Apr. 28, 1992.

[HAL95] T. R. Halfhill, "Intel's P6", *BYTE Magazine*, pp. 42-58, Apr., 1995.

[HOS86] K. Hosaka, "A New Picture Quality Evaluation Method", *Proc. International Picture Coding Symposium*, Tokyo, Japan, Apr., 1986.

[HUA74] T. S. Huang, H. Meyr, H. G. Rosdolsky, "Optimum run-length codes", IEEE Trans. Commun., COM-22, pp. 825-835, Jun., 1974.

[HUF52] D. A. Huffman, "A Method of the Construction of Minimum-Redundancy Codes", *Proceedings IRE*, vol. 40, pp. 1098-1101, Sep. 1952.

[JUT87] F. Jutand, N. Demassieux, M. Dana, J-P. Durandeau, G. Corcordel, A. Artieri, E. Mackowiack, and L. Bergher, "A 13.5MHz single chip multiformat discrete cosine transform", *Visual Commun. and Image Process. II, SPIE*, vol. 845, pp. 6-12, Cambridge, MA, Oct. 1987.

[KAR47] K. Karhunen, "Ueber lineare methoden in der Wahrscheinlichkeitsrechnung", *Ann. Acad. Sci. Fenn. Ser A.I. Math. Phys.*, vol. 37, 1947.

[KOU89] W. Kou, J. W. Mark, "A New Look at DCT-Type Transforms", *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 37, No. 12, Dec. 1989, pp. 1899-1908

[LEE84] B. G. Lee, "FCT - a fast cosine transform", *Intl. Conf. on Acoust., Speech, and Signal Process.*, pp. 38A.3.1-28A.3.3, San Diego, CA, Mar. 1984.

[LEO93] M. Leonard, "JPEG Compression Chip Cuts System Design Tasks", *EDN*, pp. 109-113, Mar. 1993

[LOE60] M. Loeve, "Probability theory", 2nd Ed., Princeton, NJ, *Van Nostrand*, pp. 478, 1960.

[MIY85] M. Miyahara, K. Kotani, "Block distortion in orthogonal transform coding - analysis, minimization, and distortion measure", *IEEE Trans. Commun.*, vol. COM-33, pp. 90-96, Jan., 1985.

[NGA86] K. N. Ngan, K. S. Leong and H. Singh, "Cosine transform coding incorporating human visual system model", presented at *SPIE Fiber 86*, Cambridge, MA, pp.165-171, Sept. 14-20, 1986.

[PEN90] W. B. Pennebaker, J. L. Mitchell, "JPEG Technical Specifications, Revision 5", *Joint Photographic Experts Group*, Jan. 2, 1990.

[QUI93] R. A. Quinnell, "EDN-Special Report : Hands On Image Compression Part 1", *EDN*, Jan. 21, 1993, pp.62-71

[RAB89] M. Rabbini, S. Daly, "An Optimized image data compression technique utilized in the Kodak SV9600 Still Video Transceiver", *SPIE, Optical Sensors and Electronic Photography*, Vol. 1071, 1989, pp.254-256

[RAO90] K. R. Rao, and P. Yip, "Discrete Cosine Transform Algorithms, Advantages Applications", San Diego, *CA:Academic Press*, 1990.

[REZ87] S. M. Rezaul Hasan, "A New VLSI Architecture for Image Data Rate Discrete Cosine Transform Processor", *ISSPA 87, signal process., theories, implementations and appl.*, pp.750-755

[SAU94] D. Saupe, R. Hamzaoui, "A Review of the Fractal Image Compression Literature", *Computer Graphics*, Vol. 28, No. 4, pp. 268-276, Nov., 1994

[SHA48] C. E. Shannon, "A Mathematical Theory of Communication", *Bell Systems Technical Journal*, vol. 27, pp. 379-423, Jul. 1948.

[SKA94] W. Skarbek, "Banach constructor in fractal compression", *Machine Graphics & Vision 3*, pp.431-441, Jan.-Feb. 1994.

[STA88] M. K. Stauffer, S. Eidson, "Image Compression with VLSI", *Telephony*, pp.26-30, Jan. 11, 1988.

[SUE86] N. Suehiro, and M. Hatori, "Fast Algorithms for the DFT and Other Sinusoidal Transforms", *IEEE Trans. Acoust., Speech, and Signal Process.*, vol. ASSP-34, no. 3, pp. 642-644, Jun. 1986.

[THE89] L. D. Thede, S. C. Kwatra, "A Hybrid Data Compression Scheme Using Quaternary Decomposition and Selective Multistage Vector Quantisation", *Proc. IEEE*, 1989, pp.1901-1905.

[THI92] A. N. Thiele, "A History of High Definition Television (HDTV) For Entertainment Purposes", *Monitor*, pp. 7-24, Vol. 17, Issue 3, Nov., 1992.

[TSA89] Y. T. Tsai, "Real-time architecture for error-tolerant color picture compression", *Digital Image Processing Applications*, Vol. 1075, 1989

[TZO84] K. H. Tzou, T. R. Hsing and J. G. Dunham, "Applications of physiological human visual system model to image compression", *SPIE Proc.*, vol. 504, pp. 419-424, 1984.

[WAN84] Z. D. Wang, "Fast Algorithms for the Discrete W Transform and for the Discrete Fourier Transform", *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. ASSP-32, no. 4, Aug. 1984.

[WIT87] I. H. Witten, R. M. Neal, J. G. Cleary, "Arithmetic Coding for Data Compression", *Communications of the ACM*, vol. 30, no. 6, Jun. 1987.

[WOO94] S. Woolley, "Rate/Distortion performance of fractal transforms for image compression", *Fractals 2*, pp. 395-398, Mar. 1994.

Appendix A Image Test Set

Standard Images (512x512)

Fig. A-1 Airplane.Y original image.

Contrast	Medium
Brightness	Medium
Characteristics	Contains a number of straight line edges. The characters on the plane help test the clarity obtained at the output of the compression algorithm. The words "GENERAL DYNAMICS" on the tail are only just readible in the original and slight distortions can greatly affect the readability of this text.

Fig. A-2 Airplane.Y reconstructed image.

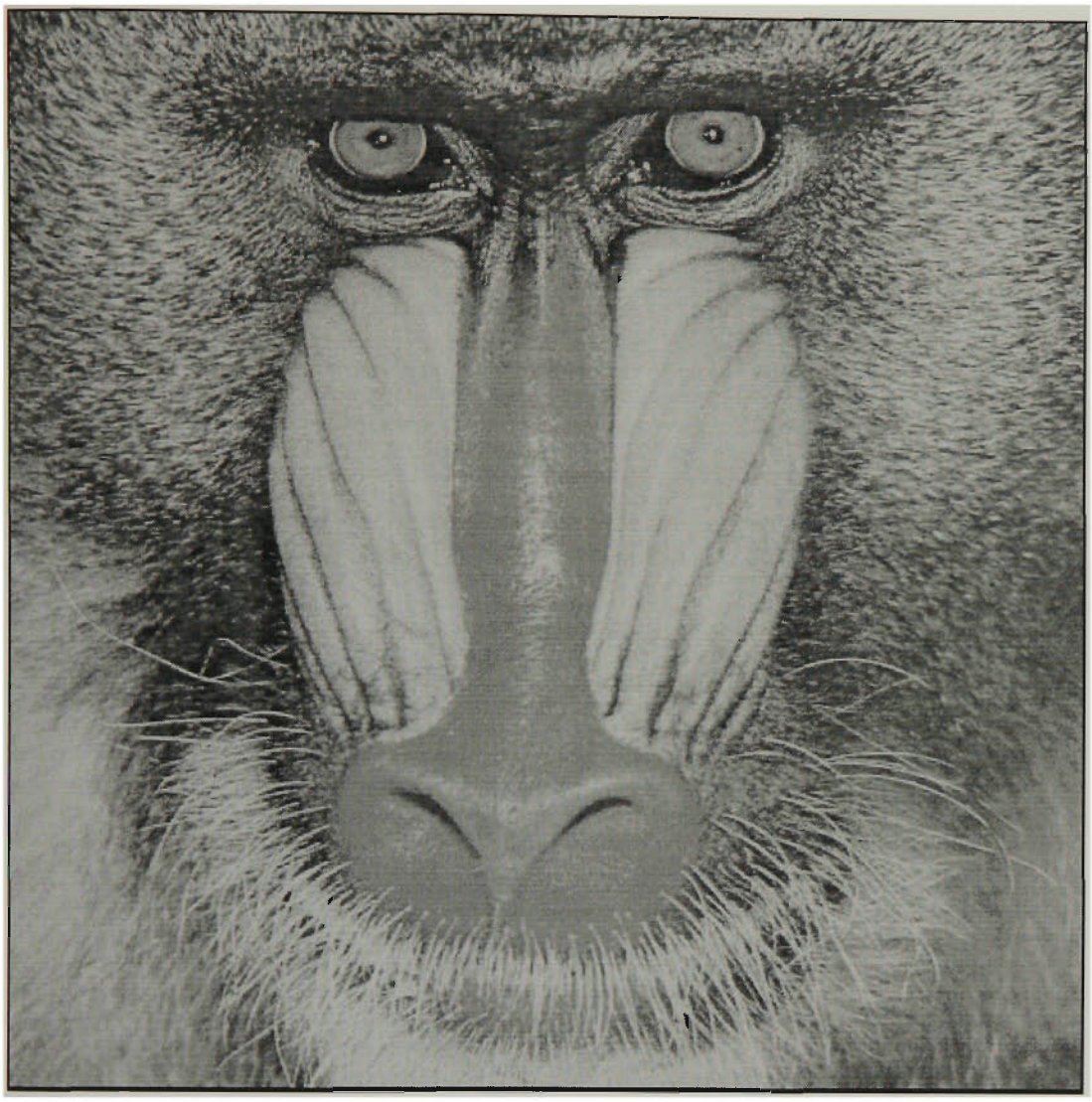


Fig. A-3 Baboon.Y original image.

Contrast	Medium-High
Brightness	Medium
Characteristics	Contains a large number of high frequency data. These sharp contrasts test the algorithms ability to compress high detail images.

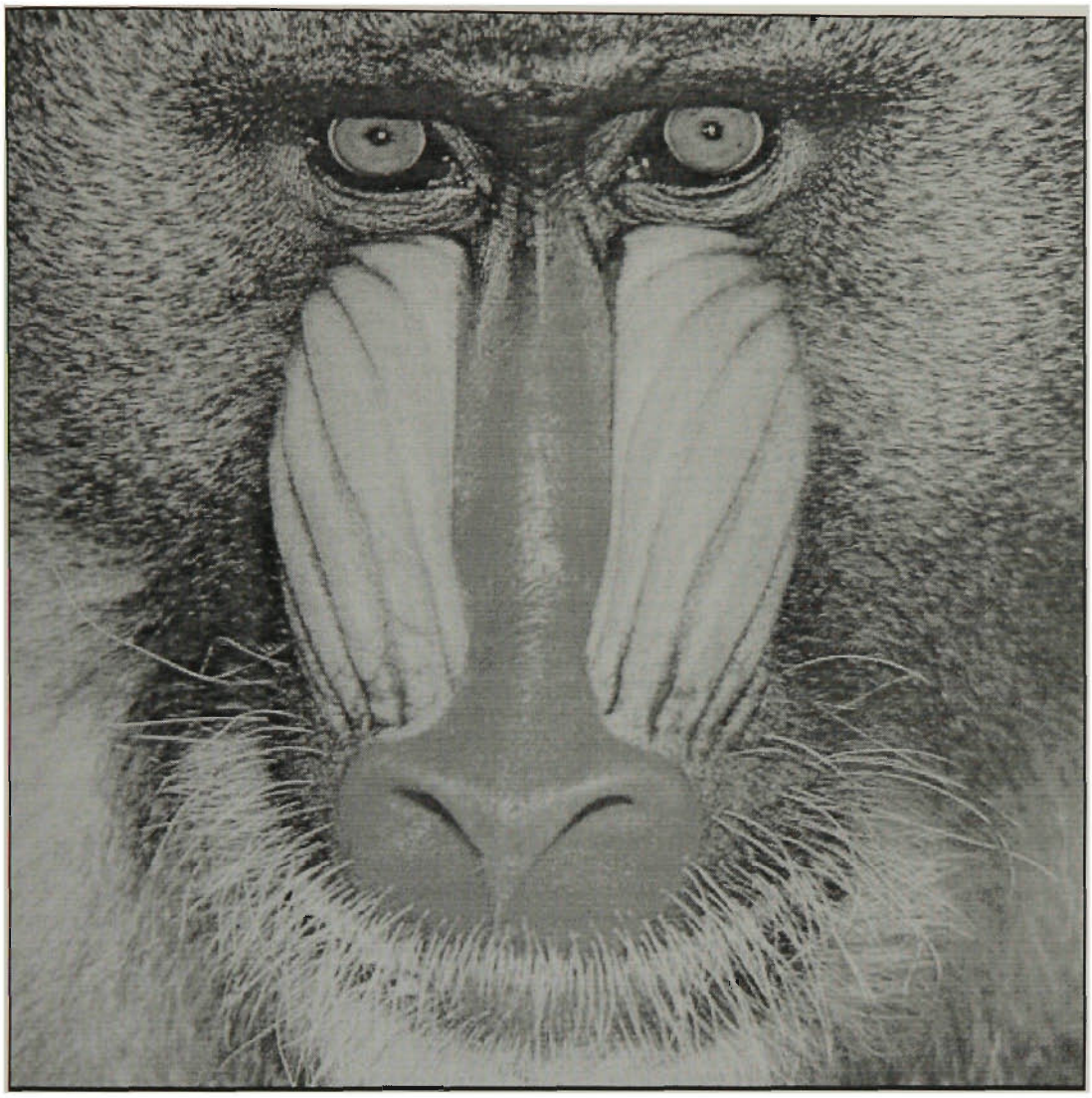


Fig. A-4 Baboon.Y reconstructed image.

Fig. A-5 Lena.Y original image.

Contrast	Medium
Brightness	Medium
Characteristics	Contains a large number of smooth intensity transitions (low frequency data). The hat feathers also contain high levels of medium frequency data.

Fig. A-6 Lena.Y reconstructed image.

Fig. A-7 Peppers.Y original image.

Contrast	Medium
Brightness	Medium
Characteristics	Contains a number of low frequency data surrounded with high contrast edges.

Fig. A-8 Peppers.Y reconstructed image.

Fig. A-9 Sailboat.Y original image.

Contrast	Medium-High
Brightness	Medium
Characteristics	The trees in the foreground are extremely dark whereas the clouds in the background are fairly bright. The image contains several long straight edges along the trunks of the trees, the pine needles form high detail and high frequency data.

Fig. A-10 Sailboat.Y reconstructed image.

Fig. A-11 Splash.Y original image.

Contrast	High
Brightness	Medium
Characteristics	Contains a large number of low frequency areas surrounded by high contrast edges.

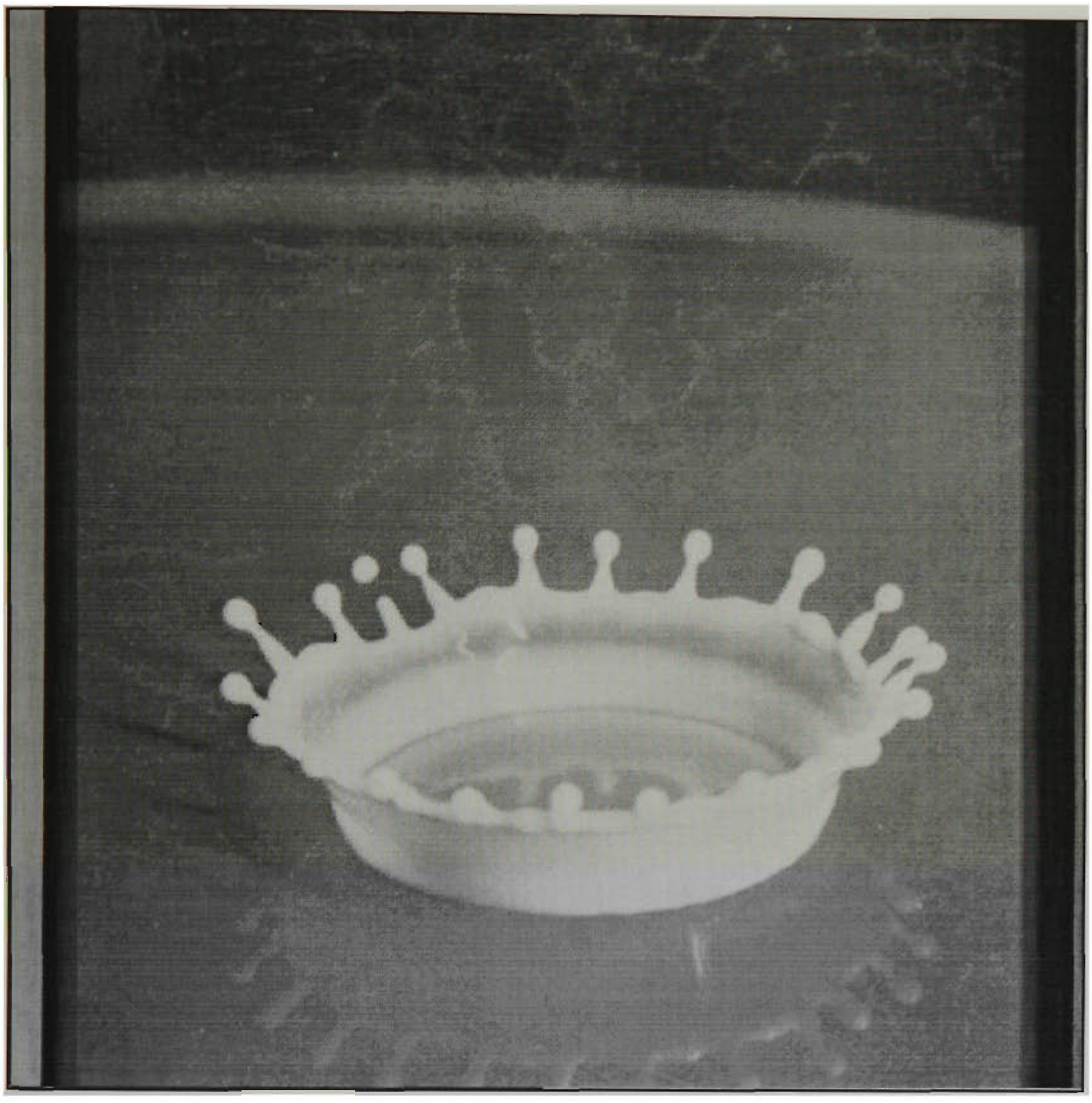


Fig. A-12 Splash.Y reconstructed image.

Fig. A-13 Tiffany.Y original image.

Contrast	Low
Brightness	High
Characteristics	High average intensity due to the low contrast and high brightness of the image.

Fig. A-14 Tiffany.Y reconstructed image.

Standard Images (256x256)

Fig. A-15 Beans1.Y original image.

Contrast	Medium
Brightness	Medium
Characteristics	Contains a low frequency content background.

Fig. A-16 Beans1.Y reconstructed image.

Fig. A-17 Beans2.Y original image.

Contrast	Medium
Brightness	Medium
Characteristics	Contains a low frequency content background.

Fig. A-18 Beans2.Y reconstructed image.

Fig. A-19 Couple.Y original image.

Contrast	Low
Brightness	Low
Characteristics	A very low average intensity due to the low contrast and low brightness of the image.

Fig. A-20 Couple.Y reconstructed image.

Fig. A-21 Girl1.Y original image.

Contrast	Low
Brightness	Low
Characteristics	A very low average intensity due to the low contrast and low brightness of the image.

Fig. A-22 Girl1.Y reconstructed image.

Fig. A-23 Girl2.Y original image.

Contrast	Low
Brightness	Medium
Characteristics	Contains a low frequency background.

Fig. A-24 Girl2.Y reconstructed image.

Fig. A-25 Girl3.Y original image.

Contrast	High and Low
Brightness	Medium
Characteristics	Contains high contrast borders surrounding a low contrast, medium brightness image.

Fig. A-26 Girl3.Y reconstructed image.

Fig. A-27 House.Y original image.

Contrast	Medium
Brightness	Medium
Characteristics	Contains a low frequency background.

Fig. A-28 House.Y reconstructed image.

Fig. A-29 Tree.Y original image.

Contrast	High
Brightness	Medium
Characteristics	A high contrast image with great detail.

Fig. A-30 Tree.Y reconstructed image.

Supplimentary Images

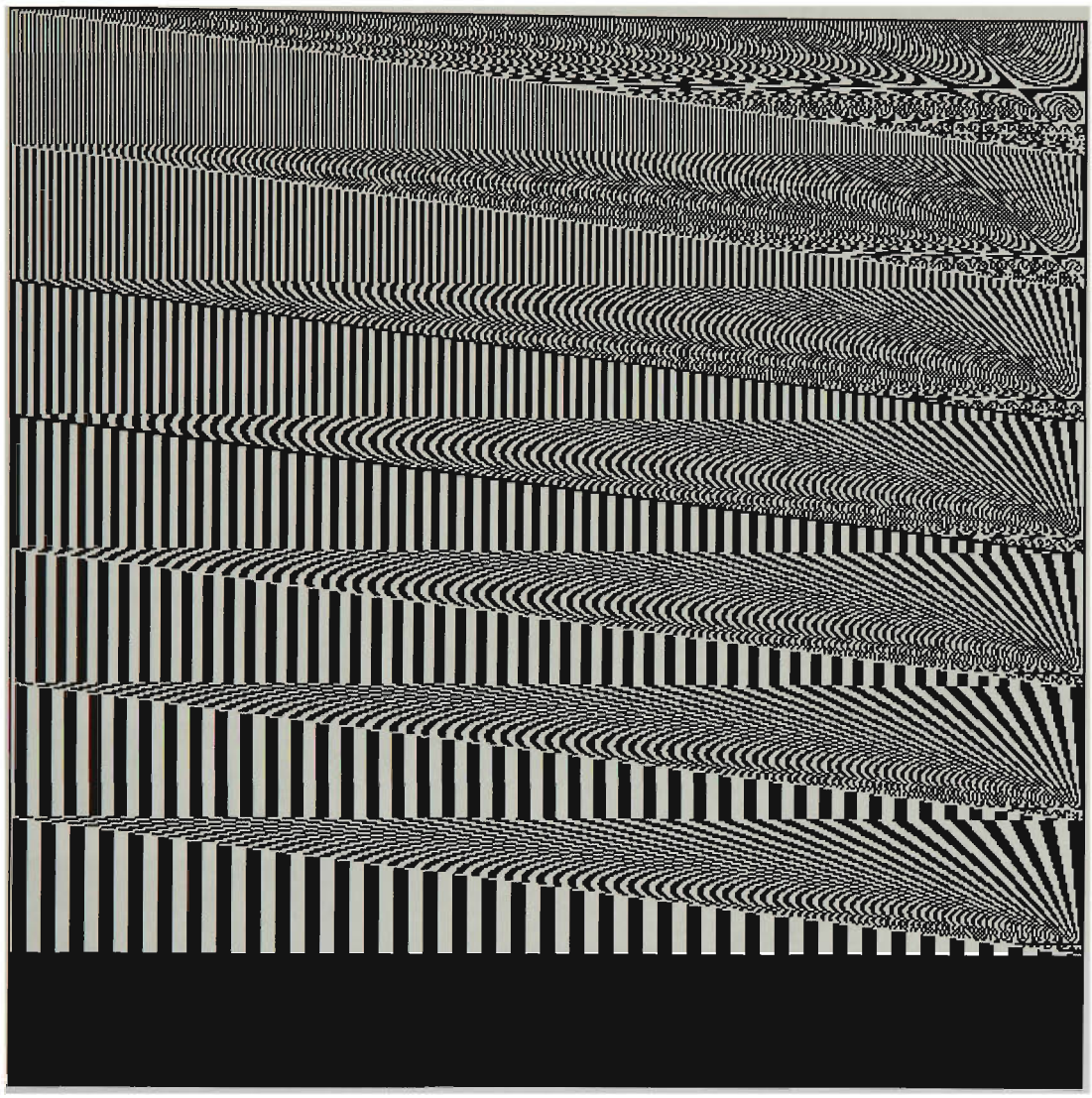


Fig. A-31 Testpatt.Y original image.

Contrast	Extremely High
Brightness	High and Low
Characteristics	An extremely high contrast artificially generated image. The image contains lines of varying angles and widths to test the compression algorithm against various two-dimensional frequencies.

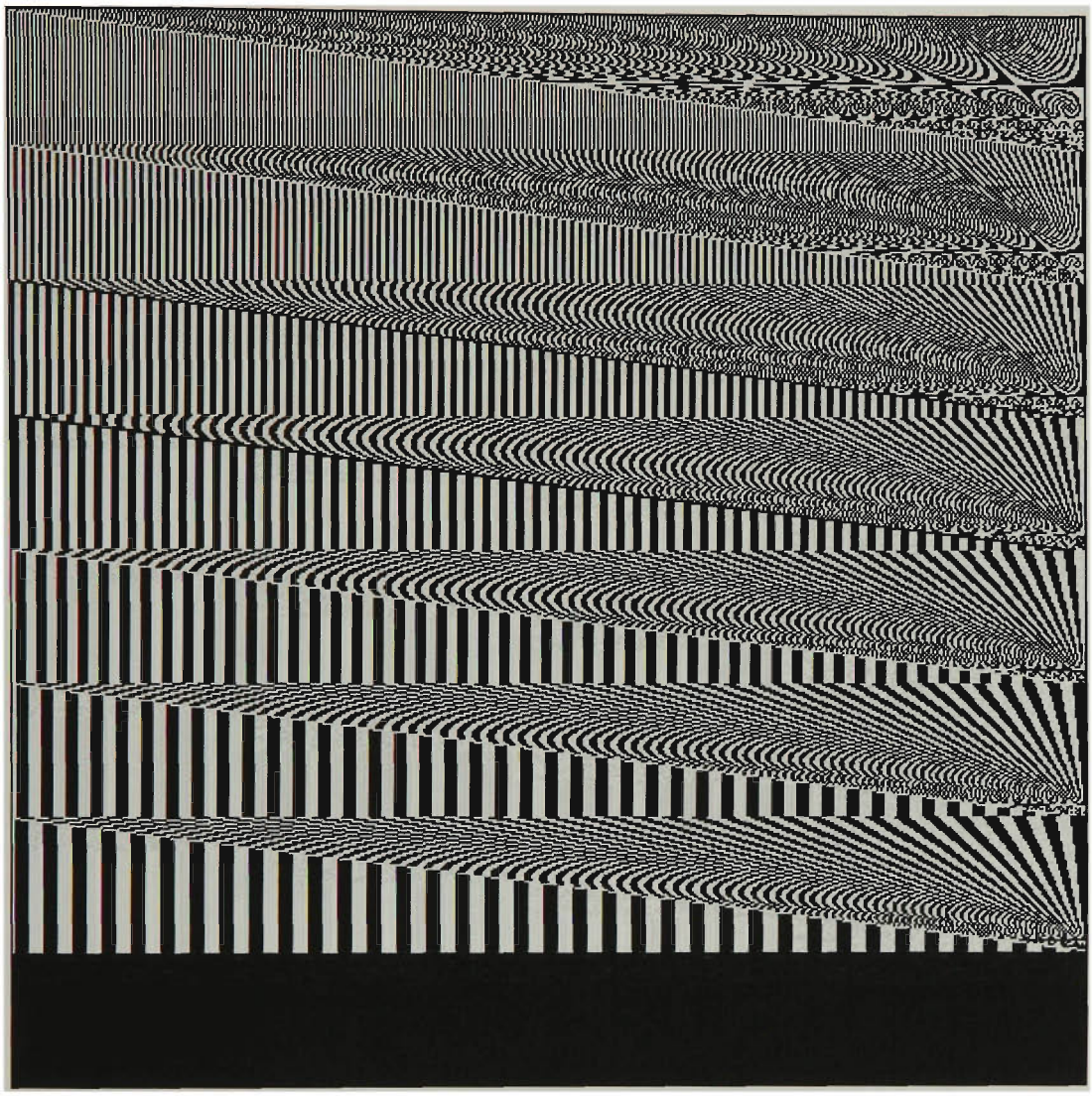


Fig. A-32 Testpatt.Y reconstructed image.

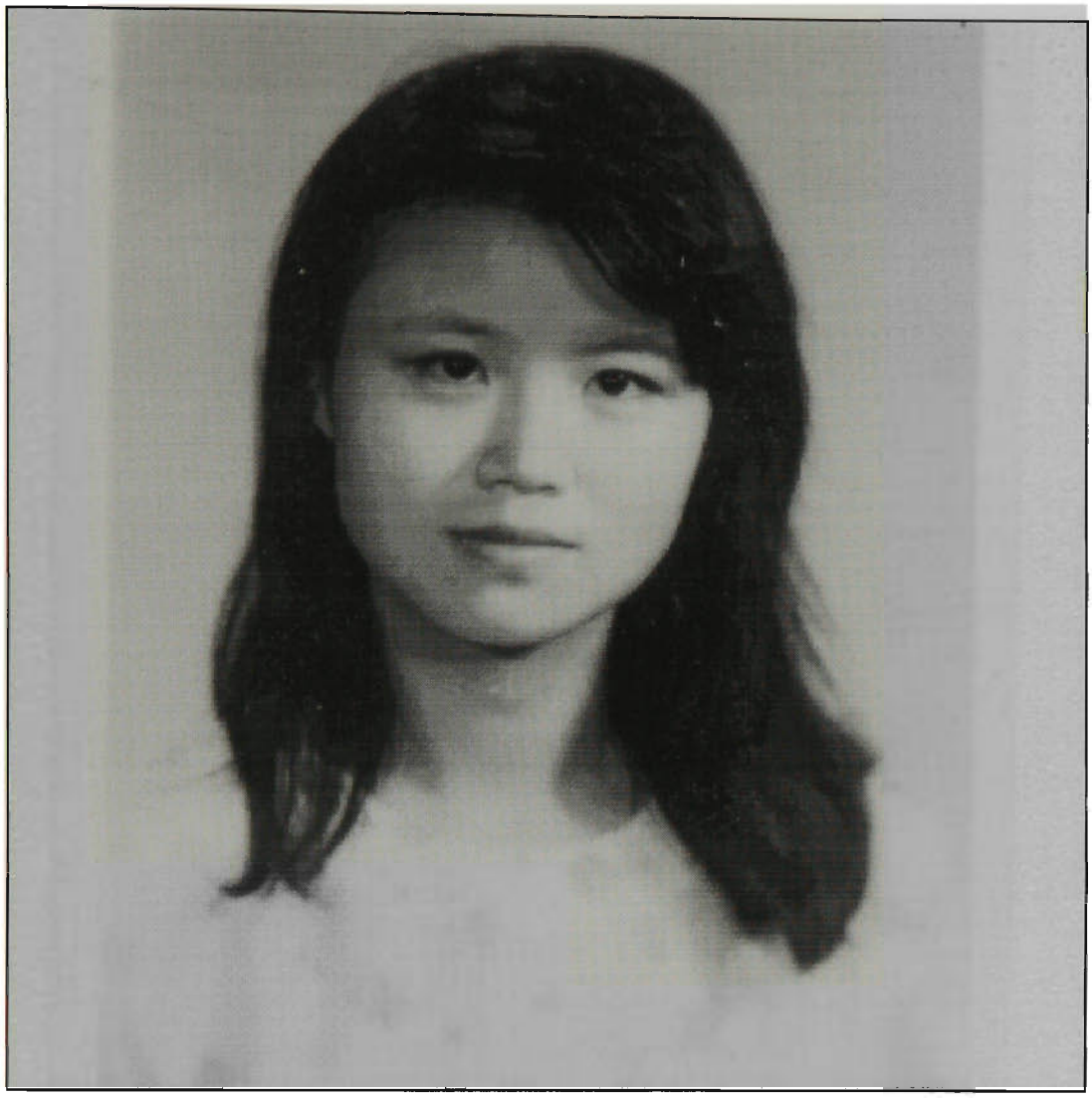


Fig. A-33 Wendy1.Y original image.

Contrast	High
Brightness	Medium
Characteristics	A high contrast scanned photographic image.

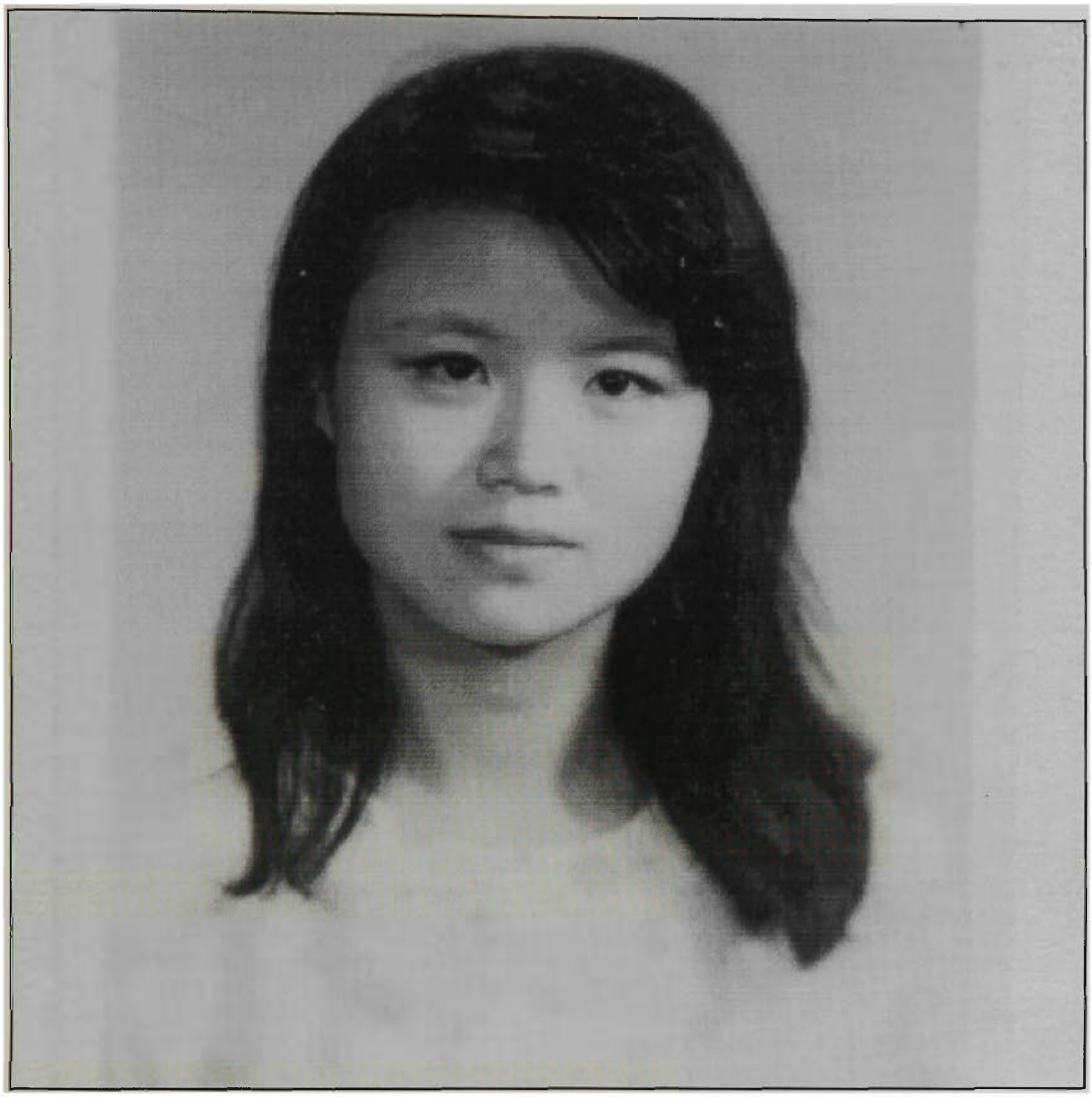


Fig. A-34 Wendy1.Y reconstructed image.

Fig. A-35 Wendy2.Y original image.

Contrast	Low
Brightness	Low
Characteristics	A low contrast image with low brightness. The image is from a scanned colour photograph.

Fig. A-36 Wendy2.Y reconstructed image.

Fig. A-37 Wendy3.Y original image.

Contrast	Low
Brightness	Low
Characteristics	A low contrast image with low brightness. The image is from a scanned colour photograph.

Fig. A-38 Wendy3.Y reconstructed image.

Appendix B Software DCT Algorithm

Software DCT Header File - DCT.H

```
/*****************************************************************************/
/*                                                                           */
/*                    B.G. Lee's DCT / IDCT Algorithm Header File            */
/*                                                                           */
/*                Department of Electrical and Electronic Engineering        */
/*                        Victoria University of Technology                  */
/*                                (Footscray Campus)                         */
/*                                  P.O. Box 14428,                          */
/*                              Melbourne Mail Centre,                       */
/*                                 Melbourne, 3000.                          */
/*                                                                           */
/*     Date         : July 6, 1994                                           */
/*                                                                           */
/*     Author       : Emil Lenc                                              */
/*                                                                           */
/*     File Name    : dct.h                                                  */
/*                                                                           */
/*     Supervisors  : Alec Simcock & Ann Pleasants                           */
/*                                                                           */
/*     Internet     : emil@cabsav.vut.edu.au                                 */
/*                                                                           */
/*****************************************************************************/

void fdct4_1d (float d[]) ;
void fidct4_1d (float d[]) ;
void fdct8_1d (float d[]) ;
void fidct8_1d (float d[]) ;
void fdct16_1d (float d[]) ;
void fidct16_1d (float d[]) ;
intsl6 round (float f) ;
void fdct (intul6 xdct, intul6 ydct, intsl6 i[16][16], intsl6 o[16][16]) ;
void fidct (intul6 xdct, intul6 ydct, intsl6 i[16][16], intsl6 o[16][16]) ;

/*****************************************************************************/
/*                                  End of dct.h                             */
/*****************************************************************************/
```

Software DCT Source File - DCT.C

```
/*****************************************************************************/
/*                                                                           */
/*                          B.G. Lee's DCT / IDCT Algorithm                  */
/*                                                                           */
/*                Department of Electrical and Electronic Engineering        */
/*                        Victoria University of Technology                  */
/*                                (Footscray Campus)                         */
/*                                  P.O. Box 14428,                          */
/*                              Melbourne Mail Centre,                       */
/*                                 Melbourne, 3000.                          */
/*                                                                           */
/*     Date         : July 6, 1994                                           */
/*                                                                           */
/*     Author       : Emil Lenc                                              */
/*                                                                           */
/*     File Name    : dct.c                                                  */
/*                                                                           */
/*     Supervisors  : Alec Simcock & Ann Pleasants                           */
/*                                                                           */
/*     Internet     : emil@cabsav.vut.edu.au                                 */
/*                                                                           */
/*****************************************************************************/
#include <math.h>
#include <mem.h>
#include "types.h"

/*****************************************************************************/
/* Various constant parameters required during the transform calculations.   */
/*****************************************************************************/
#define C0  0.500000
#define C1  0.502419
#define C2  0.509796
#define C3  0.522499
#define C4  0.541196
#define C5  0.566944
#define C6  0.601435
#define C7  0.646822
#define C8  0.707107
#define C9  0.788155
#define C10 0.899976
#define C11 1.060678
#define C12 1.306563
#define C13 1.722447
#define C14 2.562915
#define C15 5.101149

/*****************************************************************************/
/* Perform a 4 point one-dimensional forward cosine transform on the input vector d (type */
/* float). The results of the transform are stored in the same vector.       */
/*****************************************************************************/
void fdct4_1d (float d[])
{
  float temp0, temp1, temp2, temp3 ;

  temp0 = d[0] + d[2] ;                     /* First level of butterfly calculations.   */
  temp1 = d[1] + d[3] ;
  temp2 = (d[0] - d[2]) * C4 ;
  temp3 = (d[1] - d[3]) * C12 ;
  d[0] = (temp0 + temp1) ;                  /* Second level of butterfly calculations.  */
  d[1] = (temp0 - temp1) * C8 ;
  d[2] = (temp2 + temp3) ;
  d[3] = (temp2 - temp3) * C8 ;
  d[2] += d[3] ;
}
```

```
/*****************************************************************************/
/* Perform a 4 point one-dimensional inverse cosine transform on the input vector d (type */
/* float). The results of the transform are stored in the same vector.       */
/*****************************************************************************/
void fidct4_1d (float d[])
{
  float temp0, temp1, temp2, temp3 ;

  d[3] += d[2] ;                          /* First level of butterfly calculations.   */
  d[1] *= C8 ;
  d[3] *= C8 ;
  temp0 = d[0] + d[1] ;
  temp1 = d[0] - d[1] ;
  temp2 = (d[2] + d[3]) * C4 ;
  temp3 = (d[2] - d[3]) * C12 ;
  d[0] = temp0 + temp2 ;                  /* Second level of butterfly calculations.  */
  d[1] = temp1 + temp3 ;
  d[2] = temp0 - temp2 ;
  d[3] = temp1 - temp3 ;
}

/*****************************************************************************/
/* Perform a 8 point one-dimensional forward cosine transform on the input vector d (type */
/* float). The results of the transform are stored in the same vector.       */
/*****************************************************************************/
void fdct8_1d (float d[])
{
  float temp[8] ;

  temp[0] = d[0] + d[4] ;                 /* First level of butterfly calculations.   */
  temp[1] = d[1] + d[5] ;
  temp[2] = d[2] + d[6] ;
  temp[3] = d[3] + d[7] ;
  temp[4] = (d[0] - d[4]) * C2 ;
  temp[5] = (d[1] - d[5]) * C6 ;
  temp[6] = (d[2] - d[6]) * C14 ;
  temp[7] = (d[3] - d[7]) * C10 ;
  fdct4_1d (&temp[0]) ;                   /* Do partial DCT on current data.          */
  fdct4_1d (&temp[4]) ;
  d[0] = temp[0] ;                        /* Complete the transform on entire vector. */
  d[1] = temp[1] ;
  d[2] = temp[2] ;
  d[3] = temp[3] ;
  d[4] = temp[4] + temp[6] ;
  d[5] = temp[5] + temp[7] ;
  d[6] = temp[6] + temp[5] ;
  d[7] = temp[7] ;
}

/*****************************************************************************/
/* Perform a 8 point one-dimensional inverse cosine transform on the input vector d (type */
/* float). The results of the transform are stored in the same vector.       */
/*****************************************************************************/
void fidct8_1d (float d[])
{
  float temp[8], x ;

  x = d[6] + d[4] ;                       /* First level of butterfly calculations.   */
  d[7] += d[5] ;
  d[5] += d[6] ;
  d[6] = x ;
  fidct4_1d (&d[0]) ;                     /* Do partial IDCT on current data.         */
  fidct4_1d (&d[4]) ;
  temp[0] = d[0] ;                        /* Second level of butterfly calculations.  */
  temp[1] = d[1] ;
  temp[2] = d[2] ;
  temp[3] = d[3] ;
  temp[4] = d[4] * C2 ;
  temp[5] = d[5] * C6 ;
  temp[6] = d[6] * C14 ;
  temp[7] = d[7] * C10 ;
  d[0] = temp[0] + temp[4] ;              /* Final level of butterfly calculations.   */
  d[1] = temp[1] + temp[5] ;
  d[2] = temp[2] + temp[6] ;
  d[3] = temp[3] + temp[7] ;
  d[4] = temp[0] - temp[4] ;
  d[5] = temp[1] - temp[5] ;
  d[6] = temp[2] - temp[6] ;
  d[7] = temp[3] - temp[7] ;
}
```

```
/*****************************************************************************************/
/* Perform a 16 point one-dimensional forward cosine transform on the input vector d     */
/* (type float). The results of the transform are stored in the same vector.             */
/*****************************************************************************************/
void fdct16_1d (float d[])
{
  float temp[16] ;

  temp[0] = d[0] + d[8] ;                    /* First level of butterfly calculations.   */
  temp[1] = d[1] + d[9] ;
  temp[2] = d[2] + d[10] ;
  temp[3] = d[3] + d[11] ;
  temp[4] = d[4] + d[12] ;
  temp[5] = d[5] + d[13] ;
  temp[6] = d[6] + d[14] ;
  temp[7] = d[7] + d[15] ;
  temp[8] = (d[0] - d[8]) * C1 ;
  temp[9] = (d[1] - d[9]) * C3 ;
  temp[10] = (d[2] - d[10]) * C7 ;
  temp[11] = (d[3] - d[11]) * C5 ;
  temp[12] = (d[4] - d[12]) * C15 ;
  temp[13] = (d[5] - d[13]) * C13 ;
  temp[14] = (d[6] - d[14]) * C9 ;
  temp[15] = (d[7] - d[15]) * C11 ;
  fdct8_1d (&temp[0]) ;                      /* Do partial DCT on current data.          */
  fdct8_1d (&temp[8]) ;
  memcpy (d, temp, sizeof (temp) >> 1) ;     /* Complete the transform on entire vector. */
  d[8] = temp[8] + temp[12] ;
  d[9] = temp[9] + temp[13] ;
  d[10] = temp[10] + temp[14] ;
  d[11] = temp[11] + temp[15] ;
  d[12] = temp[12] + temp[10] ;
  d[13] = temp[13] + temp[11] ;
  d[14] = temp[14] + temp[9] ;
  d[15] = temp[15] ;
}

/*****************************************************************************************/
/* Perform a 16 point one-dimensional inverse cosine transform on the input vector d     */
/* (type float). The results of the transform are stored in the same vector.             */
/*****************************************************************************************/
void fidct16_1d (float d[])
{
  float temp[16], x1, x2, x3 ;

  x1 = d[9] + d[14] ;                        /* First level of butterfly calculations.   */
  x2 = d[10] + d[12] ;
  x3 = d[11] + d[13] ;
  d[12] += d[8] ;
  d[13] += d[9] ;
  d[14] += d[10] ;
  d[15] += d[11] ;
  d[9] = x1 ;
  d[10] = x2 ;
  d[11] = x3 ;
  fidct8_1d (&d[0]) ;                        /* Do partial IDCT on current data.         */
  fidct8_1d (&d[8]) ;
  memcpy (temp, d, sizeof (temp) >> 1) ;     /* Second level of butterfly calculations.  */
  temp[8] = d[8] * C1 ;
  temp[9] = d[9] * C3 ;
  temp[10] = d[10] * C7 ;
  temp[11] = d[11] * C5 ;
  temp[12] = d[12] * C15 ;
  temp[13] = d[13] * C13 ;
  temp[14] = d[14] * C9 ;
  temp[15] = d[15] * C11 ;
  d[0] = temp[0] + temp[8] ;                 /* Complete the transform on entire vector. */
  d[1] = temp[1] + temp[9] ;
  d[2] = temp[2] + temp[10] ;
  d[3] = temp[3] + temp[11] ;
  d[4] = temp[4] + temp[12] ;
  d[5] = temp[5] + temp[13] ;
  d[6] = temp[6] + temp[14] ;
  d[7] = temp[7] + temp[15] ;
  d[8] = temp[0] - temp[8] ;
  d[9] = temp[1] - temp[9] ;
  d[10] = temp[2] - temp[10] ;
  d[11] = temp[3] - temp[11] ;
  d[12] = temp[4] - temp[12] ;
  d[13] = temp[5] - temp[13] ;
  d[14] = temp[6] - temp[14] ;
  d[15] = temp[7] - temp[15] ;
}
```

```
/*****************************************************************************************/
/* Round off the input coefficient to the nearest 12 bit signed integer.                 */
/*****************************************************************************************/

intsl6 round_coef (float f)
{
  intsl6 temp ;

  temp = (f >= 0) ? (intsl6)(f + 0.5) : (intsl6)(f - 0.5) ;
  if (temp > 2047)
    temp = 2047 ;
  else
    if (temp < -2048)
      temp = -2048 ;
  return (temp) ;
}

/*****************************************************************************************/
/* Round off the input data to the nearest 8 bit signed integer.              */
/*****************************************************************************************/

intsl6 round_data (float f)
{
  intsl6 temp ;

  temp = (f >= 0) ? (intsl6)(f + 0.5) : (intsl6)(f - 0.5) ;
  if (temp > 127)
    temp = 127 ;
  else
    if (temp < -128)
      temp = -128 ;
  return (temp) ;
}

/*****************************************************************************************/
/* Perform an (xdct x ydct) two-dimensional forward discrete cosine transform on the     */
/* input array and store the results in the output array. The transform is of type DCT-II */
/* and simulates (approximately) those values which are produced by the SGS-Thomson DCT  */
/* transform chip - STV3200.                                                             */
/*****************************************************************************************/
/* xdct and ydct can only take on the values 4, 8 or 16.                                 */
/* The input and output matrices must be 16 x 16 16 bit signed integers.                 */
/*****************************************************************************************/
void fdct (intul6 xdct, intul6 ydct, intsl6 i[16][16], intsl6 o[16][16])
{
  float y[16], t[16][16], t2[16][16] ;
  intsl6 j, jj ;

  switch (xdct)
    {
      case 4 :  for (j = 0 ; j < ydct ; j++)  /* Perform a 4 point DCT in x dimension.    */
                  {
                    y[0] = i[j][0] ;          /* Correct order for input to transform.    */
                    y[1] = i[j][1] ;
                    y[2] = i[j][3] ;
                    y[3] = i[j][2] ;
                    fdct4_1d (y) ;            /* Transform the input vector.              */
                    t[j][0] = y[0]/2 ;        /* Adjust output after transform.           */
                    t[j][1] = y[2] ;
                    t[j][2] = y[1] ;
                    t[j][3] = y[3] ;
                  }
                break ;
      case 8 :  for (j = 0 ; j < ydct ; j++)  /* Perform an 8 point DCT in x dimension.   */
                  {
                    y[0] = i[j][0] ;          /* Correct order for input to transform.    */
                    y[1] = i[j][1] ;
                    y[2] = i[j][3] ;
                    y[3] = i[j][2] ;
                    y[4] = i[j][7] ;
                    y[5] = i[j][6] ;
                    y[6] = i[j][4] ;
                    y[7] = i[j][5] ;
                    fdct8_1d (y) ;            /* Transform the input vector.              */
                    t[j][0] = y[0]/2 ;        /* Adjust output after transform.           */
                    t[j][1] = y[4] ;
                    t[j][2] = y[2] ;
                    t[j][3] = y[6] ;
                    t[j][4] = y[1] ;
                    t[j][5] = y[5] ;
                    t[j][6] = y[3] ;
                    t[j][7] = y[7] ;
                  }
                break ;
```

```
    case 16 : for (j = 0 ; j < ydct ; j++)  /* Perform a 16 point DCT in x dimension.   */
                {
                  y[0] = i[j][0] ;          /* Correct order for input to transform.    */
                  y[1] = i[j][1] ;
                  y[2] = i[j][3] ;
                  y[3] = i[j][2] ;
                  y[4] = i[j][7] ;
                  y[5] = i[j][6] ;
                  y[6] = i[j][4] ;
                  y[7] = i[j][5] ;
                  y[8] = i[j][15] ;
                  y[9] = i[j][14] ;
                  y[10] = i[j][12] ;
                  y[11] = i[j][13] ;
                  y[12] = i[j][8] ;
                  y[13] = i[j][9] ;
                  y[14] = i[j][11] ;
                  y[15] = i[j][10] ;
                  fdct16_1d (y) ;           /* Transform the input vector.              */
                  t[j][0] = y[0] / 2 ;      /* Adjust the output after the transform.   */
                  t[j][1] = y[8] ;
                  t[j][2] = y[4] ;
                  t[j][3] = y[12] ;
                  t[j][4] = y[2] ;
                  t[j][5] = y[10] ;
                  t[j][6] = y[6] ;
                  t[j][7] = y[14] ;
                  t[j][8] = y[1] ;
                  t[j][9] = y[9] ;
                  t[j][10] = y[5] ;
                  t[j][11] = y[13] ;
                  t[j][12] = y[3] ;
                  t[j][13] = y[11] ;
                  t[j][14] = y[7] ;
                  t[j][15] = y[15] ;
                }
              break ;
  }
switch (ydct)                               /* Transform in the y dimension to complete */
  {                                         /* the two-dimensional transformation.      */
    case 4 :  for (j = 0 ; j < xdct ; j++)  /* Perform a 4 point DCT in y dimension.    */
                {
                  y[0] = t[0][j] ;          /* Correct order for input to transform.    */
                  y[1] = t[1][j] ;
                  y[2] = t[3][j] ;
                  y[3] = t[2][j] ;
                  fdct4_1d (y) ;            /* Transform the input vector.              */
                  t2[0][j] = y[0] ;         /* Adjust the output after the transform.   */
                  t2[1][j] = y[2]*2 ;
                  t2[2][j] = y[1]*2 ;
                  t2[3][j] = y[3]*2 ;
                }
              break ;
    case 8 :  for (j = 0 ; j < xdct ; j++)  /* Perform a 8 point DCT in y dimension.    */
                {
                  y[0] = t[0][j] ;          /* Correct order for input to transform.    */
                  y[1] = t[1][j] ;
                  y[2] = t[3][j] ;
                  y[3] = t[2][j] ;
                  y[4] = t[7][j] ;
                  y[5] = t[6][j] ;
                  y[6] = t[4][j] ;
                  y[7] = t[5][j] ;
                  fdct8_1d (y) ;            /* Transform the input vector.              */
                  t2[0][j] = y[0]/4 ;       /* Adjust the output after the transform.   */
                  t2[1][j] = y[4]/2 ;
                  t2[2][j] = y[2]/2 ;
                  t2[3][j] = y[6]/2 ;
                  t2[4][j] = y[1]/2 ;
                  t2[5][j] = y[5]/2 ;
                  t2[6][j] = y[3]/2 ;
                  t2[7][j] = y[7]/2 ;
                }
              break ;
```

```
      case 16 : for (j = 0 ; j < xdct ; j++)  /* Perform a 16 point DCT in y dimension.   */
                  {
                    y[0] = t[0][j] ;          /* Correct order for input to transform.    */
                    y[1] = t[1][j] ;
                    y[2] = t[3][j] ;
                    y[3] = t[2][j] ;
                    y[4] = t[7][j] ;
                    y[5] = t[6][j] ;
                    y[6] = t[4][j] ;
                    y[7] = t[5][j] ;
                    y[8] = t[15][j] ;
                    y[9] = t[14][j] ;
                    y[10] = t[12][j] ;
                    y[11] = t[13][j] ;
                    y[12] = t[8][j] ;
                    y[13] = t[9][j] ;
                    y[14] = t[11][j] ;
                    y[15] = t[10][j] ;
                    fdct16_1d (y) ;            /* Transform the input vector.              */
                    t2[0][j] = y[0] / 16 ;     /* Adjust the output after the transform.   */
                    t2[1][j] = y[8] / 8 ;
                    t2[2][j] = y[4] / 8 ;
                    t2[3][j] = y[12] / 8 ;
                    t2[4][j] = y[2] / 8 ;
                    t2[5][j] = y[10] / 8 ;
                    t2[6][j] = y[6] / 8 ;
                    t2[7][j] = y[14] / 8 ;
                    t2[8][j] = y[1] / 8 ;
                    t2[9][j] = y[9] / 8 ;
                    t2[10][j] = y[5] / 8 ;
                    t2[11][j] = y[13] / 8 ;
                    t2[12][j] = y[3] / 8 ;
                    t2[13][j] = y[11] / 8 ;
                    t2[14][j] = y[7] / 8 ;
                    t2[15][j] = y[15] / 8 ;
                  }
                break ;
    }
  for (j = 0 ; j < ydct ; j++)                /* Convert the output to signed integers.   */
    for (jj = 0 ; jj < xdct ; jj++)
      o[jj][j] = round_coef (t2[j][jj]) ;
}

/*****************************************************************************************/
/* Perform an (xdct x ydct) two-dimensional inverse discrete cosine transform on the     */
/* input array and store the results in the output array. The transform is of type       */
/* IDCT-II and simulates (approximately) those values which are produced by the          */
/* SGS-Thomson DCT transform chip - STV3200.                                             */
/*****************************************************************************************/
/* xdct and ydct can only take on the values 4, 8 or 16.                                 */
/* The input and output matrices must be 16 x 16 16 bit signed integers.                 */
/*****************************************************************************************/
void fidct (intu16 xdct, intu16 ydct, ints16 i[16][16], ints16 o[16][16])
{
  float y[16], t[16][16], t2[16][16] ;
  ints16 j, jj ;

  switch (xdct)
    {
      case 4 :  for (j = 0 ; j < ydct ; j++)  /* Perform a 4 point IDCT in x dimension.   */
                  {
                    y[0] = i[j][0] ;          /* Correct order for input to transform.    */
                    y[1] = i[j][2] ;
                    y[2] = i[j][1] ;
                    y[3] = i[j][3] ;
                    fidct4_1d (y) ;           /* Transform the input vector.              */
                    t[j][0] = y[0] ;          /* Adjust the output after the transform.   */
                    t[j][1] = y[1] ;
                    t[j][2] = y[3] ;
                    t[j][3] = y[2] ;
                  }
                break ;
```

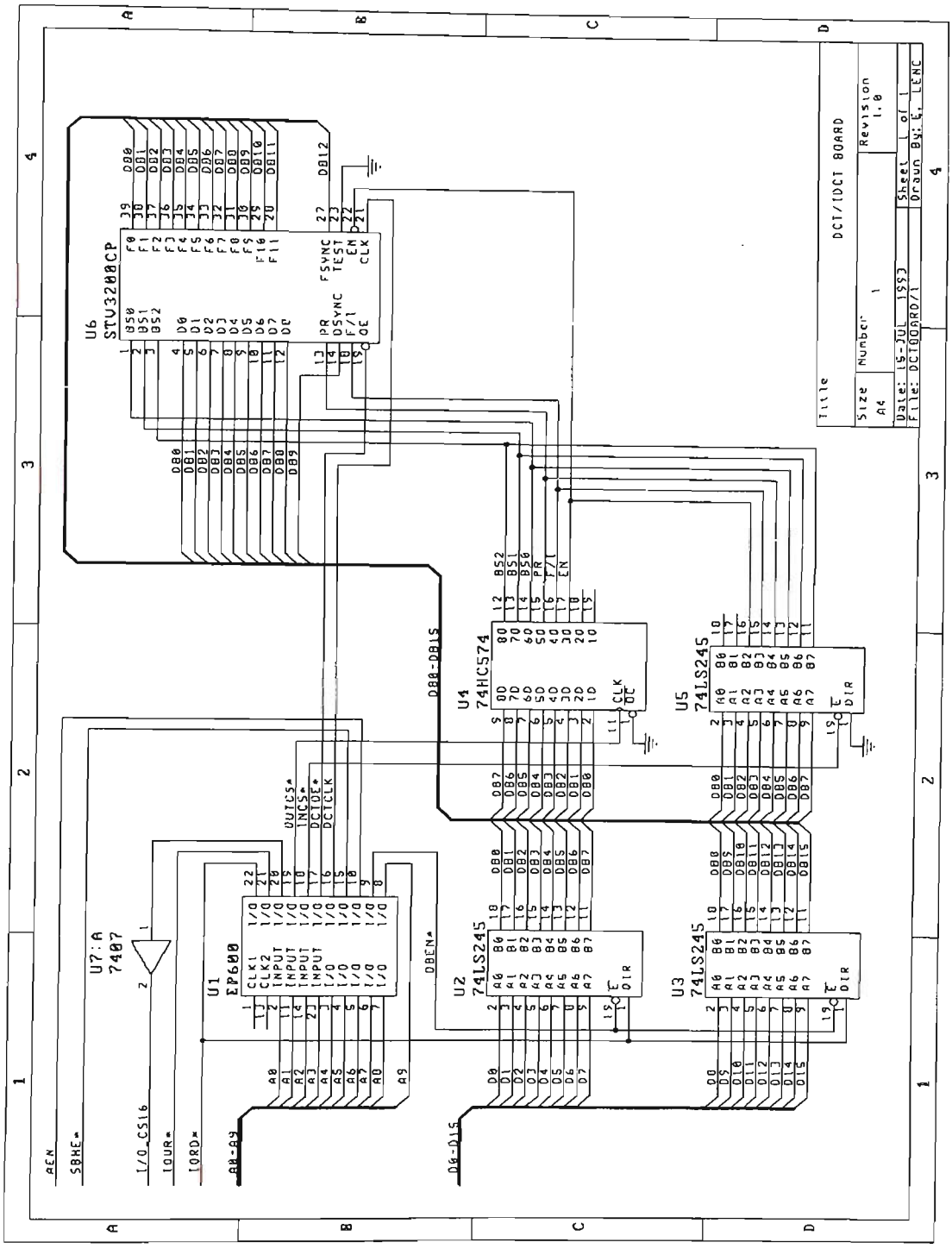
```
    case 8 :  for (j = 0 ; j < ydct ; j++)  /* Perform a 8 point IDCT in x dimension.   */
                {
                  y[0] = i[j][0] ;          /* Correct order for input to transform.    */
                  y[1] = i[j][4] ;
                  y[2] = i[j][2] ;
                  y[3] = i[j][6] ;
                  y[4] = i[j][1] ;
                  y[5] = i[j][5] ;
                  y[6] = i[j][3] ;
                  y[7] = i[j][7] ;
                  fidct8_1d (y) ;           /* Transform the input vector.              */
                  t[j][0] = y[0] ;          /* Adjust the output after the transform.   */
                  t[j][1] = y[1] ;
                  t[j][2] = y[3] ;
                  t[j][3] = y[2] ;
                  t[j][4] = y[6] ;
                  t[j][5] = y[7] ;
                  t[j][6] = y[5] ;
                  t[j][7] = y[4] ;
                }
              break ;
    case 16 : for (j = 0 ; j < ydct ; j++)  /* Perform a 16 point IDCT in x dimension.  */
                {
                  y[0] = i[j][0] ;          /* Correct order for input to transform.    */
                  y[1] = i[j][8] ;
                  y[2] = i[j][4] ;
                  y[3] = i[j][12] ;
                  y[4] = i[j][2] ;
                  y[5] = i[j][10] ;
                  y[6] = i[j][6] ;
                  y[7] = i[j][14] ;
                  y[8] = i[j][1] ;
                  y[9] = i[j][9] ;
                  y[10] = i[j][5] ;
                  y[11] = i[j][13] ;
                  y[12] = i[j][3] ;
                  y[13] = i[j][11] ;
                  y[14] = i[j][7] ;
                  y[15] = i[j][15] ;
                  fidct16_1d (y) ;          /* Transform the input vector.              */
                  t[j][0] = y[0] ;          /* Adjust the output after the transform.   */
                  t[j][1] = y[1] ;
                  t[j][2] = y[3] ;
                  t[j][3] = y[2] ;
                  t[j][4] = y[6] ;
                  t[j][5] = y[7] ;
                  t[j][6] = y[5] ;
                  t[j][7] = y[4] ;
                  t[j][8] = y[12] ;
                  t[j][9] = y[13] ;
                  t[j][10] = y[15] ;
                  t[j][11] = y[14] ;
                  t[j][12] = y[10] ;
                  t[j][13] = y[11] ;
                  t[j][14] = y[9] ;
                  t[j][15] = y[8] ;
                }
              break ;
  }
switch (ydct)
  {
    case 4 :  for (j = 0 ; j < xdct ; j++)  /* Perform a 4 point IDCT in y dimension.   */
                {
                  y[0] = t[0][j] ;          /* Correct order for input to transform.    */
                  y[1] = t[2][j] ;
                  y[2] = t[1][j] ;
                  y[3] = t[3][j] ;
                  fidct4_1d (y) ;           /* Transform the input vector.              */
                  t2[0][j] = y[0]/8 ;       /* Adjust the output after the transform.   */
                  t2[1][j] = y[1]/8 ;
                  t2[2][j] = y[3]/8 ;
                  t2[3][j] = y[2]/8 ;
                }
              break ;
```

```
      case 8 :  for (j = 0 ; j < xdct ; j++)  /* Perform a 8 point IDCT in y dimension.   */
                  {
                    y[0] = t[0][j] ;          /* Correct order for input to transform.    */
                    y[1] = t[4][j] ;
                    y[2] = t[2][j] ;
                    y[3] = t[6][j] ;
                    y[4] = t[1][j] ;
                    y[5] = t[5][j] ;
                    y[6] = t[3][j] ;
                    y[7] = t[7][j] ;
                    fidct8_1d (y) ;           /* Transform the input vector.              */
                    t2[0][j] = y[0]/8 ;       /* Adjust the output after the transform.   */
                    t2[1][j] = y[1]/8 ;
                    t2[2][j] = y[3]/8 ;
                    t2[3][j] = y[2]/8 ;
                    t2[4][j] = y[6]/8 ;
                    t2[5][j] = y[7]/8 ;
                    t2[6][j] = y[5]/8 ;
                    t2[7][j] = y[4]/8 ;
                  }
                break ;
      case 16 : for (j = 0 ; j < xdct ; j++)  /* Perform a 16 point IDCT in y dimension.  */
                  {
                    y[0] = t[0][j] ;          /* Correct order for input to transform.    */
                    y[1] = t[8][j] ;
                    y[2] = t[4][j] ;
                    y[3] = t[12][j] ;
                    y[4] = t[2][j] ;
                    y[5] = t[10][j] ;
                    y[6] = t[6][j] ;
                    y[7] = t[14][j] ;
                    y[8] = t[1][j] ;
                    y[9] = t[9][j] ;
                    y[10] = t[5][j] ;
                    y[11] = t[13][j] ;
                    y[12] = t[3][j] ;
                    y[13] = t[11][j] ;
                    y[14] = t[7][j] ;
                    y[15] = t[15][j] ;
                    fidct16_1d (y) ;          /* Transform the input vector.              */
                    t2[0][j] = y[0]/8 ;       /* Adjust the output after the transform.   */
                    t2[1][j] = y[1]/8 ;
                    t2[2][j] = y[3]/8 ;
                    t2[3][j] = y[2]/8 ;
                    t2[4][j] = y[6]/8 ;
                    t2[5][j] = y[7]/8 ;
                    t2[6][j] = y[5]/8 ;
                    t2[7][j] = y[4]/8 ;
                    t2[8][j] = y[12]/8 ;
                    t2[9][j] = y[13]/8 ;
                    t2[10][j] = y[15]/8 ;
                    t2[11][j] = y[14]/8 ;
                    t2[12][j] = y[10]/8 ;
                    t2[13][j] = y[11]/8 ;
                    t2[14][j] = y[9]/8 ;
                    t2[15][j] = y[8]/8 ;
                  }
    }
  for (j = 0 ; j < ydct ; j++)                /* Convert the output to signed integers.   */
    for (jj = 0 ; jj < xdct ; jj++)
      o[jj][j] = round_data (t2[j][jj]) ;
}

/*****************************************************************************************/
/*                                      End of dct.c                                     */
/*****************************************************************************************/
```

Appendix C Hardware DCT Interface

Schematic Diagram of STV3200 Interface



IBM Decoder Contents for STV3200 Interface

```
Emil Lenc
Electrical Engineering V.U.T.
14 January 1992
1.00
B
EP600
Address decoding for the I.B.M. - STV3200 interface

OPTIONS: SECURITY = OFF
PART: EP600

INPUTS:

A0@2, A1@11, A2@14, A3@23, A4@3,
A5@4, A6@5, A7@6,  A8@7,  A9@8,
SBHE@15, IORD@22, IOWR@21, AEN@10

OUTPUTS:

IOCS16@20, OUTCS@19, INCS@18,
DCTOE@17, DCTCLK@16, DBEN@9

NETWORK:

A0              =       INP(A0)
A1              =       INP(A1)
A2              =       INP(A2)
A3              =       INP(A3)
A4              =       INP(A4)
A5              =       INP(A5)
A6              =       INP(A6)
A7              =       INP(A7)
A8              =       INP(A8)
A9              =       INP(A9)
SBHE            =       INP(SBHE)
IORD            =       INP(IORD)
IOWR            =       INP(IOWR)
AEN             =       INP(AEN)

IOCS16          =       CONF(IOCS16,VCC)
OUTCS           =       CONF(OUTCS,VCC)
INCS            =       CONF(INCS,VCC)
DCTOE           =       CONF(DCTOE,VCC)
DCTCLK          =       CONF(DCTCLK,VCC)
DBEN            =       CONF(DBEN,VCC)

EQUATIONS:

address         =       A9 & A8 & !A7 & !A6 & !A5 & !A4 & !A3 & !A2 & !AEN ;

DCTOE           =       !(address & !A1 & !IORD & !SBHE) ;
DCTCLK          =       address & !A1 & !IOWR & !SBHE ;

OUTCS           =       !(address & A1 & !A0 & !IOWR) ;
INCS            =       !(address & A1 & !A0 & !IORD) ;

IOCS16          =       !(address & !A1) ;
DBEN            =       !(!DCTOE + DCTCLK + !OUTCS + !INCS) ;

END$
```

STV3200 Driver Header File - HDCT.H

```
/*****************************************************************************************/
/*                                                                                       */
/*                              STV3200 Driver Header File                               */
/*                                                                                       */
/*                  Department of Electrical and Electronic Engineering                  */
/*                          Victoria University of Technology                            */
/*                                  (Footscray Campus)                                   */
/*                                    P.O. Box 14428,                                    */
/*                                Melbourne Mail Centre,                                 */
/*                                   Melbourne, 3000.                                    */
/*                                                                                       */
/*      Date          : July 6, 1994                                                     */
/*                                                                                       */
/*      Author        : Emil Lenc                                                        */
/*                                                                                       */
/*      File Name     : hdct.h                                                           */
/*                                                                                       */
/*      Supervisors   : Alec Simcock & Ann Pleasants                                     */
/*                                                                                       */
/*      Internet      : emil@cabsav.vut.edu.au                                           */
/*                                                                                       */
/*****************************************************************************************/

#define FDCT          0x08
#define IDCT          0x00

void setup_DCT (intu16 x_size, intu16 y_size, intu16 DCT_type) ;
void init_DCT () ;
void fdct (intu16 xdct, intu16 ydct, ints8 far *source, ints16 far *destination) ;
void fidct (intu16 xdct, intu16 ydct, ints16 far *source, ints8 far *destination) ;

/*****************************************************************************************/
/*                                       End of dct.h                                    */
/*****************************************************************************************/
```

STV3200 Driver Source File - HDCT.C

```
/*****************************************************************************/
/*                                                                           */
/*                          STV3200P Driver For IBM PC                       */
/*                                                                           */
/*              Department of Electrical and Electronic Engineering          */
/*                      Victoria University of Technology                    */
/*                              (Footscray Campus)                           */
/*                                P.O. Box 14428,                            */
/*                             Melbourne Mail Centre,                        */
/*                                Melbourne, 3000.                           */
/*                                                                           */
/*     Date         : July 6, 1994                                           */
/*                                                                           */
/*     Author       : Emil Lenc                                              */
/*                                                                           */
/*     File Name    : hdct.c                                                 */
/*                                                                           */
/*     Supervisors  : Alec Simcock & Ann Pleasants                           */
/*                                                                           */
/*     Internet     : emil@cabsav.vut.edu.au                                 */
/*                                                                           */
/*****************************************************************************/
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#include "types.h"
#include "memtype.h"

/*****************************************************************************/
/* Various constant parameters specific for the STV3200 Board.               */
/*****************************************************************************/

#define DCT_base      0x300                 /* Base address of the STV3200 Interface.   */
#define DCT_data      DCT_base              /* Data address of the STV3200 Interface.   */
#define DCT_control   DCT_base+2            /* Control address of the STV3200 Interface.*/
#define DCT_status    DCT_base+2            /* Status address of the STV3200 Interface. */

#define size_16x16    0x00                  /* Control mask for 16x16 transforms.       */
#define size_8x16     0x80                  /* Control mask for 8x16 transforms.        */
#define size_16x8     0x40                  /* Control mask for 16x8 transforms.        */
#define size_8x8      0xC0                  /* Control mask for 8x8 transforms.         */
#define size_4x8      0x20                  /* Control mask for 4x8 transforms.         */
#define size_8x4      0xA0                  /* Control mask for 8x4 transforms.         */
#define size_4x4      0x60                  /* Control mask for 4x4 transforms.         */

#define DCT_chip_enable  0x00               /* Chip enable for the STV3200.             */
#define DCT_chip_disable 0x04               /* Chip disable for the STV3200.            */

#define FDCT          0x08                  /* Control mask for forward transform.      */
#define IDCT          0x00                  /* Control mask for inverse transform.      */

#define DCT_PR_low    0x00                  /* Rounding control for the STV3200.        */
#define DCT_PR_high   0x10
```

```
/*****************************************************************************************/
/* setup_DCT sets up the STV3200 to operate with the given size block size and initializes*/
/* the device for the given type of transform : Forward / Inverse.                        */
/*****************************************************************************************/

void setup_DCT (intu16 x_size, intu16 y_size, intu16 DCT_type)
{
  intu32 DCT_size ;


  switch (x_size)                              /* Determine the DCT_size mask according to */
    {                                          /* x_size and y_size.                       */
      case 4  : switch (y_size)
                {
                  case 4  :  DCT_size = size_4x4 ;
                             break ;
                  case 8  :  DCT_size = size_4x8 ;
                             break ;
                  default :  fprintf (stderr, "Size Error\n") ;
                             exit (1) ;
                }
      break ;
      case 8  : switch (y_size)
                {
                  case 4  :  DCT_size = size_8x4 ;
                             break ;
                  case 8  :  DCT_size = size_8x8 ;
                             break ;
                  case 16 :  DCT_size = size_8x16 ;
                             break ;
                  default :  fprintf (stderr, "Size Error\n") ;
                             exit (1) ;
                }
              break ;
      case 16 : switch (y_size)
                {
                  case 8  :  DCT_size = size_16x8 ;
                             break ;
                  case 16 :  DCT_size = size_16x16 ;
                             break ;
                  default :  fprintf (stderr, "Size Error\n") ;
                             exit (1) ;
                }
    }
  outportb (DCT_control, DCT_size+DCT_chip_enable+DCT_type+DCT_PR_low) ;
}


/*****************************************************************************************/
/* Initialize the DCT as described in data sheets for the STV3200 device.                */
/*****************************************************************************************/

void init_DCT ()
{
  intu16 x ;

  setup_DCT (16, 16, FDCT) ;                   /* Assume a 16x16 forward transform.        */
  for (x = 0 ; x < 130 ; x++)
    outport (DCT_data, 0x200) ;                /* Output 130 values to device to clear it. */
}
```

```
/*****************************************************************************************/
/* Perform a forward DCT on the source data and output the results in the destination.   */
/*****************************************************************************************/

void fdct (intu16 xdct, intu16 ydct, ints8 far *source, ints16 far *destination)
{
  intu16 array_size ;

  setup_DCT (xdct, ydct, FDCT) ;              /* Set up the DCT size.                    */
  array_size = xdct * ydct ;
        asm         PUSH    DS                /* Save segment registers.                 */
        asm         PUSH    ES
        asm         LDS     SI,source         /* Get the source address.                 */
        asm         LES     DI,destination    /* Get the destination address             */
        asm         MOV     DX,DCT_data       /* Point to the STV3200 data register.     */

        asm         CLD
        asm         MOV     CX,array_size     /* Move the entire array to the STV3200.   */
        asm         DEC     CX

        asm         LODSB                     /* Get a byte from the source.             */
        asm         CBW                       /* Convert it to a signed word.            */
        asm         AND     AH,0x01           /* First word needs bit 9 low.             */
        asm         OUT     DX,AX             /* Output the data to the STV3200.         */
  MOVE1: asm        LODSB                     /* Get a byte from the source.             */
        asm         CBW                       /* Convert it to a signed word.            */
        asm         OR      AH,0x02           /* Successive words need bit 9 set.        */
        asm         OUT     DX,AX             /* Output the data word to the STV3200.    */
        asm         LOOP    MOVE1             /* Repeat until entire block sent.         */

        asm         MOV     CX,131            /* Dummy writes to flush out the output.   */
        asm         MOV     AX,0x200
        asm         MOV     BX,AX
  MOVE2: asm        OUT     DX,AX
        asm         LOOP    MOVE2

        asm         MOV     CX,array_size
  MOVE3: asm        IN      AX, DX            /* Get the data from the STV3200.          */
        asm         TEST    AH, 0x08          /* Perform sign extension of the 12 bit data*/
        asm         JE      MOVEP
        asm         OR      AH, 0xF0
        asm         JMP     MOVE4
  MOVEP: asm        AND     AH, 0x0F
  MOVE4: asm        STOSW
        asm         MOV     AX,BX
        asm         OUT     DX,AX             /* A dummy write to clock next data out.   */
        asm         LOOP    MOVE3

        asm         POP     ES                /* Restore registers.                      */
        asm         POP     DS
}
```

```
/*****************************************************************************************/
/* Perform an inverse DCT on the source data and output the results in the destination.  */
/*****************************************************************************************/

void fidct (intu16 xdct, intu16 ydct, ints16 far *source, ints8 far *destination)
{
  intu16 array_size ;
  setup_DCT (xdct, ydct, IDCT) ;              /* Set up the DCT size.                    */
  array_size = xdct * ydct ;
         asm          PUSH   DS               /* Save segment registers.                 */
         asm          PUSH   ES
         asm          LDS    SI,source        /* Get the source address.                 */
         asm          LES    DI,destination   /* Get the destination address.            */
         asm          MOV    DX,DCT_data      /* Point to the STV3200 data register.     */

         asm          CLD
         asm          MOV    CX,array_size    /* Move the entire array to the STV3200.   */
         asm          DEC    CX

         asm          LODSW                   /* Get a byte from the source.             */
         asm          AND    AH,0x0F          /* Bit 12 must be zero for first word.     */
         asm          OUT    DX,AX            /* Output the word.                        */
  MOVE1: asm          LODSW                   /* Get a byte from the source.             */
         asm          OR     AH,0x10          /* Successive words have bit 12 set.       */
         asm          OUT    DX,AX            /* Output a word.                          */
         asm          LOOP   MOVE1            /* Continue until entire block sent.       */

         asm          MOV    CX,131           /* Dummy writes to flush out the output.   */
         asm          MOV    AX,0x1000
         asm          MOV    BX,AX
  MOVE2: asm          OUT    DX,AX
         asm          LOOP   MOVE2

         asm          MOV    CX,array_size
  MOVE3: asm          IN     AX,DX            /* Get the results from the STV3200.       */
         asm          TEST   AH,0x01          /* Clip the 9 bit data to 8 bits.          */
         asm          JE     MOVE4
         asm          CMP    AL,0x80
         asm          JA     MOVE5
         asm          MOV    AL,0x80
         asm          JMP    MOVE5
  MOVE4: asm          CMP    AL,0x80
         asm          JB     MOVE5
         asm          MOV    AL,0x7F
  MOVE5: asm          STOSB                   /* Store the data in the destination.      */
         asm          MOV    AX,BX
         asm          OUT    DX,AX            /* Output a dummy value to get next word.  */
         asm          LOOP   MOVE3

         asm          POP    ES               /* Restore registers.                      */
         asm          POP    DS
}

/*****************************************************************************************/
/*                                    End of hdct.c                                      */
/*****************************************************************************************/
```

Appendix D Algorithm Software

WINDCT.DEF - Definition File

```
NAME            windct
EXETYPE         WINDOWS
STUB            'WINSTUB.EXE'
CODE            PRELOAD MOVEABLE
DATA            PRELOAD MOVEABLE MULTIPLE
HEAPSIZE        1024
STACKSIZE       5120
EXPORTS         WndProc
```

WINDCT.RC - Resource File

```
/*****************************************************************************
WINDCT.RC
produced by Borland Resource Workshop
*****************************************************************************/

#include "windct.h"
#define ICON_1    1
#define MENU_1    1

windct MENU
{
 POPUP "&File"
 {
  MENUITEM "&Open", IDM_FILEOPEN
  MENUITEM "&Save Compressed", IDM_SAVECOMPRESSED, GRAYED
  MENUITEM "S&ave Decompressed", IDM_SAVEDECOMPRESSED, GRAYED
  MENUITEM SEPARATOR
  MENUITEM "E&xit", IDM_FILEEXIT
 }

 POPUP "&Tools"
 {
  MENUITEM "&Compress", IDM_TOOLSCOMPRESS, GRAYED
  MENUITEM "&Decompress", IDM_TOOLSDECOMPRESS, GRAYED
  MENUITEM "&Full Cycle", IDM_TOOLSFULLCYCLE, GRAYED
 }

}

windct ICON "windct.ico"
DIALOG_JPEG DIALOG 74, 41, 166, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION
CAPTION "JPEG Quality Level"
FONT 8, "MS Sans Serif"
{
 DEFPUSHBUTTON "OK", IDOK, 14, 37, 50, 14, BS_DEFPUSHBUTTON | NOT WS_TABSTOP
 PUSHBUTTON "Cancel", IDCANCEL, 101, 37, 50, 14, WS_TABSTOP
 CTEXT "Quality Level", -1, 37, 4, 55, 8, SS_CENTER | NOT WS_GROUP
 SCROLLBAR IDC_HSCROLL1, 19, 15, 92, 9, SBS_HORZ | WS_GROUP
 EDITTEXT IDC_EDIT1, 117, 13, 22, 13, WS_BORDER | WS_GROUP | WS_TABSTOP
}
```

WINDCT.CPP - Main Program

```
//
//
//                    Image Compression Software For The IBM PC
//
//                Department of Electrical and Electronic Engineering
//                        Victoria University of Technology
//                                (Footscray Campus)
//                                  P.O. Box 14428,
//                              Melbourne Mail Centre,
//                                 Melbourne, 3000.
//
//      Date         : July 7, 1994
//
//      Author       : Emil Lenc
//
//      File Name    : windct.cpp
//
//      Supervisors  : Alec Simcock & Ann Pleasants
//
//      Internet     : emil@cabsav.vut.edu.au
//
//

#define STRICT                                  // strict type checking

//
// Define macro to store a byte in the specified buffer and automatically increment the
// pointer such that it points to the next byte in the buffer.
//
#define mput(buffer,value) (*((buffer)++) = (value))

//
// Define macro to get a byte from the specified buffer and automatically increment the
// pointer such that it points to the next byte in the buffer.
//
#define mget(buffer) (*((buffer)++))

#include <windows.h>                            // include file for all windows apps
#include <commdlg.h>                            // for GetOpenFileName()
#include <dos.h>                                // DOS functions
#include <math.h>                               // Math functions
#include <memory.h>                             // Memory functions
#include <string.h>                             // String functions
#include "windct.h"

//
// Various constant parameters specific for the STV3200 Board
//

#define DCT_base      0x300                     // Base address of the STV3200 Interface
#define DCT_data      DCT_base                  // Data address of the STV3200 Interface
#define DCT_control   DCT_base+2                // Control address of the STV3200 Interface
#define DCT_status    DCT_base+2                // Status address of the STV3200 Interface

#define size_16x16    0x00                      // Control mask for 16x16 transforms

#define FDCT          0x08                      // Control mask for forward transform
#define IDCT          0x00                      // Control mask for inverse transform

#define BUFFER_SIZE  256                        // Buffer size
#define TRANSFORM_BUFFER_SIZE 32768L            // Transformation Buffer Size


// Quantiser type definitions for each coefficient

const int quant_type [256] = {
  0, 1, 1, 2, 2, 2, 3, 3, 3, 3, 3, 3, 4, 5, 6, 7,
  1, 2, 2, 2, 3, 3, 4, 4, 4, 5, 5, 6, 7, 7, 7, 7,
  1, 2, 2, 2, 3, 3, 4, 4, 4, 5, 5, 6, 7, 7, 7, 7,
  2, 2, 2, 3, 3, 3, 4, 4, 5, 5, 5, 6, 6, 7, 7, 7,
  2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 7, 7,
  2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 6, 6, 6, 6, 7, 7,
  3, 4, 4, 4, 4, 4, 5, 5, 5, 5, 6, 6, 6, 6, 7, 7,
  3, 4, 4, 4, 4, 5, 5, 5, 5, 6, 6, 6, 6, 6, 7, 7,
  3, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 7, 7, 7,
  3, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 7, 7, 7,
  3, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7,
  3, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7,
  4, 7, 7, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7,
  5, 7, 7, 7, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 7,
  6, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
  7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7 } ;
```

```
// Non-uniform quantiser linear segment definitions.

int quant_x [8][8] = {
  {  0, 2048, 2048, 2048, 2048, 2048, 2048, 2048 },
  {  0,   24,  255, 2048, 2048, 2048, 2048, 2048 },
  {  0,    6,   34, 2048, 2048, 2048, 2048, 2048 },
  {  0,    6,   36, 2048, 2048, 2048, 2048, 2048 },
  {  0,    6,   38, 2048, 2048, 2048, 2048, 2048 },
  {  0,    6,   38, 2048, 2048, 2048, 2048, 2048 },
  {  0,   64,  128, 2048, 2048, 2048, 2048, 2048 },
  {  0,  128, 2048, 2048, 2048, 2048, 2048, 2048 } } ;

int quant_y [8][8] = {
  {  6,    6,    6,    6,    6,    6,    6,    6 },
  {  2,    3,    4,    4,    4,    4,    4,    4 },
  {  2,    4,   10,   10,   10,   10,   10,   10 },
  {  2,    6,   20,   20,   20,   20,   20,   20 },
  {  2,    3,   12,   12,   12,   12,   12,   12 },
  {  2,    3,   20,   20,   20,   20,   20,   20 },
  {  8,   16,   32,   32,   32,   32,   32,   32 },
  { 32,   64,   64,   64,   64,   64,   64,   64 } } ;

// Order in which the coefficients are stored.

int coefficientOrder [256] = {
   16,  0,  1, 32, 17, 48,  2, 33, 18, 49, 34, 19,  3, 50, 35, 20,
   64, 65, 80, 36, 51,  4, 66, 21, 81, 37, 67, 52, 82, 96,  5, 22,
   97, 38, 83, 53, 68, 23, 54, 98, 69, 99,112,113, 84,  6, 85, 39,
   70, 24,114, 55,100,115, 40,129,  7,101, 25, 71, 86, 56,116,130,
   41, 87,102,131,145, 26, 72,117, 57,128,146,144,  8, 88, 42,132,
  118,103,133, 73,161, 58,147,148,119,  9, 89,104,162, 74,163,160,
  134,120,149,105,176,135,164, 10,150, 11,192,208, 12, 13, 27, 43,
   59,177, 90,178, 60,179, 75,106,121,136,151,165, 91,180,137,195,
  166,122,107, 76,152,181,167, 92,182, 77,196,168,153,123,138,197,
  108,183,139, 14,184,124,212, 93,198,154,199,140,109,213,169,170,
  224,155,156,186,125,200,214,215,171,185,201, 44, 28, 45,240, 46,
   29, 47, 61, 30, 62, 31, 94, 78,110, 63,209,193,111, 95,241, 79,
  194,210,126,243,172,157,127,245,232,246,242,244,173,189,226,141,
  211,216,220,143,229,217,230,247,159,204,231,142,227,233,158,202,
  221,174,187,205,188,236,228,190,234,235,203,219,175,225,237,248,
  218,250,206,191,238,223,222,251,249,207,254,239,252,253, 15,255 } ;

const int BlockSize = 16 ;                    // Size of DCT Block Size
const int DCTSetup = size_16x16 ;             // Set up value for Hardware DCT

PSTR szProgName = "windct";                   // application name
HGLOBAL hImageIn, hImageDCT, hImageOut, hForward[8], hBackward[8] ;
HGLOBAL hImageRunLength, hImageCompressed ;
int huge *hpForward [8], huge *hpBackward [8] ;
int blockorder [1024] ;
DWORD coefforder [256] ;
DWORD dwImageSize, dwImageWidth, dwRLCodeLength, dwCodeLength ;
BOOL bImageLoaded = FALSE ;
BOOL bImageCompressed = FALSE ;
BOOL bImageDecompressed = FALSE ;

#pragma argsused                              // ignore unused arguments

#include "huffadap.cpp"                       // Statistical Coder to use
```

```
//
// Output the code for the particular symbol and run-length.
//
void output_code (int symbol, unsigned long length, char huge **dest)
{
  union _rdata {
    unsigned long full ;
    unsigned char part [4] ;
  } rdata ;

  switch (symbol) {
    case -3 :                                        // Handle the -3 symbol
      switch (length) {
        case 1 : mput (*dest, 0xFD) ;
          break ;
        case 2 : mput (*dest, 0x80) ;
          break ;
        default :
          if (length < 10) {
            mput (*dest, 0x88) ;
            mput (*dest, (char)(0x1D | ((length - 2) << 5))) ;
          } else if (length < 257) {
            mput (*dest, 0x89) ;
            mput (*dest, 0xFD) ;
            mput (*dest, (char)(length - 1)) ;
          } else {
            mput (*dest, 0x8A) ;
            rdata.full = ((length - 1) << 12) | 0x0FFD ;
            mput (*dest, rdata.part[0]) ;
            mput (*dest, rdata.part[1]) ;
            mput (*dest, rdata.part[2]) ;
            mput (*dest, rdata.part[3]) ;
          }
      }
      break ;
    case -2 :                                        // Handle the -2 symbol
      switch (length) {
        case 1 : mput (*dest, 0xFE) ;
          break ;
        case 2 : mput (*dest, 0x81) ;
          break ;
        default :
          if (length < 10) {
            mput (*dest, 0x88) ;
            mput (*dest, (char)(0x1E | ((length - 2) << 5))) ;
          } else if (length < 257) {
            mput (*dest, 0x89) ;
            mput (*dest, 0xFE) ;
            mput (*dest, (char)(length - 1)) ;
          } else {
            mput (*dest, 0x8A) ;
            rdata.full = ((length - 1) << 12) | 0x0FFE ;
            mput (*dest, rdata.part[0]) ;
            mput (*dest, rdata.part[1]) ;
            mput (*dest, rdata.part[2]) ;
            mput (*dest, rdata.part[3]) ;
          }
      }
      break ;
    case -1 :                                        // Handle the -1 symbol
      switch (length) {
        case 1 : mput (*dest, 0xFF) ;
          break ;
        case 2 : mput (*dest, 0x82) ;
          break ;
        case 3 : mput (*dest, 0x86) ;
          break ;
        default :
          if (length < 10) {
            mput (*dest, 0x88) ;
            mput (*dest, (char)(0x1F | ((length - 2) << 5))) ;
          } else if (length < 257) {
            mput (*dest, 0x89) ;
            mput (*dest, 0xFF) ;
            mput (*dest, (char)(length - 1)) ;
          } else {
            mput (*dest, 0x8A) ;
            rdata.full = ((length - 1) << 12) | 0x0FFF ;
            mput (*dest, rdata.part[0]) ;
            mput (*dest, rdata.part[1]) ;
            mput (*dest, rdata.part[2]) ;
            mput (*dest, rdata.part[3]) ;
          }
      }
      break ;
```

```
case 0 :                                        // Handle the 0 symbol
  if (length == 1)
    mput (*dest, 0x00) ;
  else
    if (length < 10)
      mput (*dest, (char)(0x7F - length + 2)) ;
    else
      if (length < 266) {
        mput (*dest, 0x77) ;
        mput (*dest, (char)(length - 10)) ;
      } else {
        mput (*dest, 0x8A) ;
        rdata.full = ((length - 1) << 12) ;
        mput (*dest, rdata.part[0]) ;
        mput (*dest, rdata.part[1]) ;
        mput (*dest, rdata.part[2]) ;
        mput (*dest, rdata.part[3]) ;
      }
  break ;
case 1 :                                        // Handle the 1 symbol
  switch (length) {
    case 1 : mput (*dest, 0x01) ;
      break ;
    case 2 : mput (*dest, 0x83) ;
      break ;
    case 3 : mput (*dest, 0x87) ;
      break ;
    default :
      if (length < 10) {
        mput (*dest, 0x88) ;
        mput (*dest, (char)(0x01 | ((length - 2) << 5))) ;
      } else if (length < 257) {
        mput (*dest, 0x89) ;
        mput (*dest, 0x01) ;
        mput (*dest, (char)(length - 1)) ;
      } else {
        mput (*dest, 0x8A) ;
        rdata.full = ((length - 1) << 12) | 0x0001 ;
        mput (*dest, rdata.part[0]) ;
        mput (*dest, rdata.part[1]) ;
        mput (*dest, rdata.part[2]) ;
        mput (*dest, rdata.part[3]) ;
      }
  }
  break ;
case 2 :                                        // Handle the 2 symbol
  switch (length) {
    case 1 : mput (*dest, 0x02) ;
      break ;
    case 2 : mput (*dest, 0x84) ;
      break ;
    default :
      if (length < 10) {
        mput (*dest, 0x88) ;
        mput (*dest, (char)(0x02 | ((length - 2) << 5))) ;
      } else if (length < 257) {
        mput (*dest, 0x89) ;
        mput (*dest, 0x02) ;
        mput (*dest, (char)(length - 1)) ;
      } else {
        mput (*dest, 0x8A) ;
        rdata.full = ((length - 1) << 12) | 0x0002 ;
        mput (*dest, rdata.part[0]) ;
        mput (*dest, rdata.part[1]) ;
        mput (*dest, rdata.part[2]) ;
        mput (*dest, rdata.part[3]) ;
      }
  }
  break ;
```

```
    case 3 :                                        // Handle the 3 symbol
      switch (length) {
        case 1 : mput (*dest, 0x03) ;
          break ;
        case 2 : mput (*dest, 0x85) ;
          break ;
        default :
          if (length < 10) {
            mput (*dest, 0x88) ;
            mput (*dest, (char)(0x03 | ((length - 2) << 5))) ;
          } else if (length < 257) {
            mput (*dest, 0x89) ;
            mput (*dest, 0x03) ;
            mput (*dest, (char)(length - 1)) ;
          } else {
            mput (*dest, 0x8A) ;
            rdata.full = ((length - 1) << 12) | 0x0003 ;
            mput (*dest, rdata.part[0]) ;
            mput (*dest, rdata.part[1]) ;
            mput (*dest, rdata.part[2]) ;
            mput (*dest, rdata.part[3]) ;
          }
      }
      break ;
    default :                                       // Handle all other symbols
      if (length == 1) {
        if (symbol > -118 && symbol < 118)
          mput (*dest, (char)(symbol)) ;
        else {
          mput (*dest, 0x76) ;
          rdata.full = (symbol << 4) ;
          mput (*dest, rdata.part[0]) ;
          mput (*dest, rdata.part[1]) ;
        }
      } else if (length == 2 && symbol > -118 && symbol < 118) {
        mput (*dest, symbol) ;
        mput (*dest, symbol) ;
      } else if (length < 10 && symbol > -17 && symbol < 16) {
        mput (*dest, 0x88) ;
        mput (*dest, (char)(((symbol & 0x1F) | ((length - 2) << 5)))) ;
      } else if (length < 257 && symbol > -129 && symbol < 128) {
        mput (*dest, 0x89) ;
        rdata.full = ((length - 1) << 8) | (symbol & 0xFF) ;
        mput (*dest, rdata.part[0]) ;
        mput (*dest, rdata.part[1]) ;
      } else if (length < 16) {
        mput (*dest, 0x76) ;
        rdata.full = (symbol << 4) | (length - 1) ;
        mput (*dest, rdata.part[0]) ;
        mput (*dest, rdata.part[1]) ;
      } else {
        mput (*dest, 0x8A) ;
        rdata.full = ((length - 1) << 12) | (symbol & 0x0FFF) ;
        mput (*dest, rdata.part[0]) ;
        mput (*dest, rdata.part[1]) ;
        mput (*dest, rdata.part[2]) ;
        mput (*dest, rdata.part[3]) ;
      }
  }
}
```

```
//
// The run-length coder
//
DWORD RunLengthCode (int huge *indata, char huge *outdata, DWORD input_size)
{
  int last_code, next_code ;
  unsigned long run_length ;
  char huge *o = outdata ;

  run_length = 0 ;                              // Reset the run length of the current data.
  for (;;) {                                    // Process all the input data.
    if (!run_length) {                          // Is the current run length = 0 ?
      if (!(input_size--))                      // If nothing left to code then exit
        break ;
      last_code = *(indata++) ;                 // Otherwise start up a new run.
      run_length++ ;                            // The run length is now one greater.
    }
    if (!(input_size--)) {                      // If so then code it.
      output_code (last_code, run_length, &o) ;
      break ;                                   // and exit.
    }
    next_code = *(indata++) ;                   // Get the next byte from the source data.
    if (last_code == next_code)                 // Is the new data the same as the last ?
      run_length++ ;                            // If so then the run length is one greater.
    else {                                      // Otherwise output the previous run.
      output_code (last_code, run_length, &o) ;
      last_code = next_code ;                   // Start a new run of the the new data.
      run_length = 1 ;                          // The run is of length 1 so far.
    }
  }
  return (DWORD)(o - outdata) ;
}

//
// Output a run-length of a given symbol.
//
void output_run (int symbol, DWORD long length, int huge **dest)
{
  for (int i = 0 ; i < length ; i++)
    mput (*dest, symbol) ;
}

//
// The run-length decoder.
//
DWORD RunLengthDecode (char huge *indata, int huge *outdata, DWORD input_size)
{
  int huge *dest = outdata ;
  int tempi ;
  union _rdata {
    DWORD full ;
    unsigned char part [4] ;
  } rdata ;

  for (;;) {
    if (!(input_size--))
      break ;
    rdata.full = *(indata++) ;
    switch (rdata.part [0]) {
      case 0x80 :
        output_run (0xFFFD, 2, &dest) ;
        break ;
      case 0x81 :
        output_run (0xFFFE, 2, &dest) ;
        break ;
      case 0x82 :
        output_run (0xFFFF, 2, &dest) ;
        break ;
      case 0x83 :
        output_run (0x0001, 2, &dest) ;
        break ;
      case 0x84 :
        output_run (0x0002, 2, &dest) ;
        break ;
      case 0x85 :
        output_run (0x0003, 2, &dest) ;
        break ;
      case 0x86 :
        output_run (0xFFFF, 3, &dest) ;
        break ;
      case 0x87 :
        output_run (0x0001, 3, &dest) ;
        break ;
```

```
      case 0x88 :
        rdata.full = 0 ;
        rdata.part [0] = *(indata++) ;
        input_size-- ;
        tempi = (int)(rdata.part[0] & 0x1F) ;
        tempi |= (tempi & 0x10)?0xFFF0:0x0000 ;
        output_run (tempi, (rdata.full >> 5) + 2, &dest) ;
        break ;
      case 0x89 :
        tempi = *(indata++) ;
        rdata.full = 0 ;
        rdata.part [0] = *(indata++) ;
        input_size -= 2 ;
        tempi |= (tempi & 0x80)?0xFF80:0x0000 ;
        output_run (tempi, rdata.full + 1, &dest) ;
        break ;
      case 0x8A :
        rdata.part[0] = *(indata++) ;
        rdata.part[1] = *(indata++) ;
        rdata.part[2] = *(indata++) ;
        rdata.part[3] = *(indata++) ;
        input_size -= 4 ;
        tempi = (int)(rdata.full & 0x0FFF) ;
        tempi |= (tempi & 0x0800)?0xF800:0x0000 ;
        output_run (tempi, (rdata.full >> 12) + 1, &dest) ;
        break ;
      case 0x76 :
        rdata.part[0] = *(indata++) ;
        rdata.part[1] = *(indata++) ;
        input_size -= 2 ;
        tempi = (int)(rdata.full >> 4) ;
        tempi |= (tempi & 0x0800)?0xF800:0x0000 ;
        output_run (tempi, (rdata.part[0] & 0x0F) + 1, &dest) ;
        break ;
      case 0x77 :
        rdata.full = 0 ;
        rdata.part [0] = *(indata++) ;
        input_size-- ;
        output_run (0x0000, rdata.full + 10, &dest) ;
        break ;
      case 0x78 :
        output_run (0x0000, 9, &dest) ;
        break ;
      case 0x79 :
        output_run (0x0000, 8, &dest) ;
        break ;
      case 0x7A :
        output_run (0x0000, 7, &dest) ;
        break ;
      case 0x7B :
        output_run (0x0000, 6, &dest) ;
        break ;
      case 0x7C :
        output_run (0x0000, 5, &dest) ;
        break ;
      case 0x7D :
        output_run (0x0000, 4, &dest) ;
        break ;
      case 0x7E :
        output_run (0x0000, 3, &dest) ;
        break ;
      case 0x7F :
        output_run (0x0000, 2, &dest) ;
        break ;
      default :
        rdata.full |= (rdata.full & 0x80)?0xFF80:0x0000 ;
        output_run ((int)rdata.full, 1, &dest) ;
    }
  }
  return ((DWORD)(dest - outdata)) ;
}

//
// Create the block ordering for the current image.
//
void CreateBlockOrder (int width)
{
  int *cp ;
  int x, y, numBlocks ;

  cp = blockorder ;
  numBlocks = (width / 16) * (width / 16) ;
  for (y = 0 ; y < width / 16 ; y++)
    for (x = 0 ; x < width / 16 ; x++)
      *(cp++) = (y*(width/16)) + ((y & 1)?((width/16)-x-1):x) ;
  for (y = 0 ; y < 256 ; y++)
    coefforder [y] = (DWORD)coefficientOrder [y] * (DWORD)numBlocks ;
}
```

```
//
// Create the quantiser table.
//
void gen_quant (int *qf, int *qb, int *qx, int *qy)
{
  int xx = 0, s, ss = 0, x ;

  for (x = -2048 ; x < 2048 ; x++)
    {
      qf [x + 2048] = 0 ;
      if (x >= 0)
        qb [x + 2048] = 2047 ;
      else
        qb [x + 2048] = -2048 ;
    }
  qb [2048] = 0 ;
  for (x = 0 ; x < 2048 ; x++)
    {
      if (x > *qx) {
        qx++ ;
        s = *(qy++) ;
      }
      ss++ ;
      qf[x + 2048] = xx ;
      qf[2048 - x] = -xx ;
      if (ss == s) {
        ss = 0 ;
        qb [xx + 2048 + 1] = x + (s + 1) / 2 + 1 ;
        qb [2048 - xx - 1] = -x - (s + 1) / 2 - 1 ;
        xx++ ;
      }
  }
  qf [0] = qf [1] ;
}

//
// Adjust the appearance of the menus depending on the current status of the algorithm.
//
void ModifyMenus (HWND hWnd, BOOL SAVECOM, BOOL SAVEDCOM, BOOL COM, BOOL DCOM, BOOL FULL)
{
  HMENU hMenu, hFile, hTools ;

  hMenu = GetMenu (hWnd) ;
  hFile = GetSubMenu (hMenu, 0) ;
  hTools = GetSubMenu (hMenu, 1) ;
  if (SAVECOM)
    EnableMenuItem (hFile, IDM_SAVECOMPRESSED, MF_BYCOMMAND | MF_ENABLED) ;
  else
    EnableMenuItem (hFile, IDM_SAVECOMPRESSED, MF_BYCOMMAND | MF_GRAYED) ;
  if (SAVEDCOM)
    EnableMenuItem (hFile, IDM_SAVEDECOMPRESSED, MF_BYCOMMAND | MF_ENABLED) ;
  else
    EnableMenuItem (hFile, IDM_SAVEDECOMPRESSED, MF_BYCOMMAND | MF_GRAYED) ;
  if (COM)
    EnableMenuItem (hTools, IDM_TOOLSCOMPRESS, MF_BYCOMMAND | MF_ENABLED) ;
  else
    EnableMenuItem (hTools, IDM_TOOLSCOMPRESS, MF_BYCOMMAND | MF_GRAYED) ;
  if (DCOM)
    EnableMenuItem (hTools, IDM_TOOLSDECOMPRESS, MF_BYCOMMAND | MF_ENABLED) ;
  else
    EnableMenuItem (hTools, IDM_TOOLSDECOMPRESS, MF_BYCOMMAND | MF_GRAYED) ;
  if (FULL)
    EnableMenuItem (hTools, IDM_TOOLSFULLCYCLE, MF_BYCOMMAND | MF_ENABLED) ;
  else
    EnableMenuItem (hTools, IDM_TOOLSFULLCYCLE, MF_BYCOMMAND | MF_GRAYED) ;
}

//
// Initialize the DCT as described in data sheets for the STV3200 device.
//
void InitDCT (void)
{
  WORD x, z ;

  outportb (DCT_control, FDCT) ;              // Assume a 16x16 forward transform.
  for (x = 0 ; x < 130 ; x++)
    outport (DCT_data, 0x200) ;               // Output 130 values to device to clear it.

  for (x = 0 ; x < 8 ; x++)
    gen_quant (hpForward[x], hpBackward[x], quant_x [x], quant_y [x]) ;
}
```

```
//
// Load the image into memory.
// The image must be in the PPM format
//
int LoadImage (HWND hWnd, LPCSTR szFileName)
{
  HFILE hFileIn ;
  char huge* hpImageIn ;
  DWORD dwBytesRead ;
  char  s512x512 [] = { 'P', '5', 0x0A, '5', '1', '2', ' ', '5', '1', '2', 0x0A, '2', '5', '5', 0x0A
} ;
  char  s256x256 [] = { 'P', '5', 0x0A, '2', '5', '6', ' ', '2', '5', '6', 0x0A, '2', '5', '5', 0x0A
} ;
  char  sHeader [15] ;

  hFileIn = _lopen (szFileName, OF_READ) ;
  if (hFileIn == HFILE_ERROR)
    return NULL ;
  dwBytesRead = _lread (hFileIn, sHeader, sizeof (sHeader)) ;
  if (dwBytesRead != sizeof (sHeader))
    return NULL ;
  if (strncmp (s512x512, sHeader, sizeof (s512x512)) == 0)
    dwImageWidth = 512 ;
  else
    if (strncmp (s256x256, sHeader, sizeof (s256x256)) == 0)
      dwImageWidth = 256 ;
    else
      return NULL ;
  if (bImageLoaded)
    {
      GlobalFree (hImageOut) ;
      GlobalFree (hImageDCT) ;
      GlobalFree (hImageIn) ;
      GlobalFree (hImageRunLength) ;
      GlobalFree (hImageCompressed) ;
      bImageLoaded = FALSE ;
    }
  dwImageSize = (DWORD)dwImageWidth * (DWORD)dwImageWidth ;
  hImageIn = GlobalAlloc (GMEM_MOVEABLE, dwImageSize) ;
  hImageDCT = GlobalAlloc(GMEM_MOVEABLE, dwImageSize << 1) ;
  hImageOut = GlobalAlloc (GMEM_MOVEABLE, dwImageSize) ;
  hImageRunLength = GlobalAlloc (GMEM_MOVEABLE, dwImageSize) ;
  hImageCompressed = GlobalAlloc (GMEM_MOVEABLE, dwImageSize) ;
  if(!hImageDCT || !hImageIn || !hImageOut || !hImageRunLength || !hImageCompressed)
    {
      MessageBox(hWnd, "Error Allocating Memory", szProgName, MB_SYSTEMMODAL | MB_OK) ;
      return NULL ;
    }
  hpImageIn = (char huge*)GlobalLock(hImageIn) ;
  if (_hread (hFileIn, hpImageIn, dwImageSize) != dwImageSize)
    return NULL ;
  _lclose (hFileIn) ;
  GlobalUnlock (hImageIn) ;
  bImageLoaded = TRUE ;
  bImageDecompressed = bImageCompressed = FALSE ;
  CreateBlockOrder ((int)dwImageWidth) ;
  return 1 ;
}
```

```
//
// Provide the window interface to open an image file.
//
int OpenFile (HWND hWnd)
{
  char szPath[BUFFER_SIZE];          // buffer for path and file name
  char szTitle[BUFFER_SIZE];         // buffer for file name only
  char szString[BUFFER_SIZE];        // buffer for various strings
  OPENFILENAME ofn;
  LPCSTR szFilter[] = { "Intensity files (*.Y)", "*.Y",
                        "Green files (*.G)", "*.G",
                        "All files (*.*)", "*.*",
                        "" };

  szPath[0] = '\0';                  // empty the name field
  memset(&ofn, 0, sizeof(OPENFILENAME)); // zero the structure
  ofn.lStructSize = sizeof(OPENFILENAME);
  ofn.hwndOwner = hWnd;              // owner is main window
  ofn.lpstrFilter = szFilter[0];     // filter string array
  ofn.lpstrFile = szPath;            // path+name buffer
  ofn.nMaxFile = BUFFER_SIZE;        // size of above
  ofn.lpstrFileTitle = szTitle;      // file name buffer
  ofn.nMaxFileTitle = BUFFER_SIZE;   // size of above
  ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;
  if(!GetOpenFileName (&ofn))        // get the path+name
    return NULL;                     // user pressed Cancel
  if (!LoadImage (hWnd, szPath))
    {
      MessageBox (NULL,"Error: Cannot open file", szProgName, MB_OK | MB_ICONINFORMATION);
      return NULL;
    }
  ModifyMenus (hWnd, FALSE, FALSE, TRUE, FALSE, TRUE) ;
  return 1 ;
}

//
// The window interface to save the final compressed image to disk.
//
int SaveCompressed (HWND hWnd)
{
  HFILE hFileOut ;
  char huge* hpImageCompressed ;     // pointer to memory (note: huge)
  char szPath[BUFFER_SIZE] ;         // buffer for path and file name
  char szTitle[BUFFER_SIZE] ;        // buffer for file name only
  char szString[BUFFER_SIZE] ;       // buffer for various strings
  OPENFILENAME sfn ;
  LPCSTR szFilter[] = { "Compressed Image files (*.AC)", "*.AC",
                        "All files (*.*)", "*.*",
                        "" } ;

  szPath[0] = '\0' ;                 // empty the name field
  memset(&sfn, 0, sizeof(OPENFILENAME)) ; // zero the structure
  sfn.lStructSize = sizeof(OPENFILENAME) ;
  sfn.hwndOwner = hWnd ;             // owner is main window
  sfn.lpstrFilter = szFilter[0] ;    // filter string array
  sfn.lpstrFile = szPath ;           // path+name buffer
  sfn.nMaxFile = BUFFER_SIZE ;       // size of above
  sfn.lpstrFileTitle = szTitle ;     // file name buffer
  sfn.nMaxFileTitle = BUFFER_SIZE ;  // size of above
  sfn.Flags = OFN_PATHMUSTEXIST | OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT ;
  if(!GetSaveFileName (&sfn))        // get the path+name
    return NULL;                     // user pressed Cancel
  hpImageCompressed = (char huge*)GlobalLock (hImageCompressed) ;
  hFileOut = _lcreat (szPath, 0) ;
  if (!hFileOut)
    {
      MessageBox(hWnd, "Error Opening Output File", szProgName, MB_SYSTEMMODAL | MB_OK) ;
      return NULL ;
    }
  if (_hwrite (hFileOut, hpImageCompressed, dwCodeLength) != dwCodeLength)
    {
      MessageBox(hWnd, "Error Writing Output File", szProgName, MB_SYSTEMMODAL | MB_OK) ;
      return NULL ;
    }
  _lclose (hFileOut) ;
  GlobalUnlock (hImageDCT) ;
  return 1 ;
}
```

```
//
// The window interface to store the reconstructed image onto disk.
//
int SaveDecompressed (HWND hWnd)
{
  HFILE hFileOut ;
  char huge *hpImageOut ;
  char szPath[BUFFER_SIZE] ;          // buffer for path and file name
  char szTitle[BUFFER_SIZE] ;         // buffer for file name only
  char szString[BUFFER_SIZE] ;        // buffer for various strings
  OPENFILENAME ofn ;
  DWORD dwBytesWrite ;
  LPCSTR szFilter[] = { "Reconstructed Image files (*.NEW)", "*.NEW",
                        "All files (*.*)", "*.*",
                        "" } ;

  char  s512x512 [] = { 'P','5',0x0A,'5','1','2',' ','5','1','2',0x0A,'2','5','5',0x0A } ;
  char  s256x256 [] = { 'P','5',0x0A,'2','5','6',' ','2','5','6',0x0A,'2','5','5',0x0A } ;

  szPath[0] = '\0' ;                  // empty the name field
  memset(&ofn, 0, sizeof(OPENFILENAME)) ; // zero the structure
  ofn.lStructSize = sizeof(OPENFILENAME) ;
  ofn.hwndOwner = hWnd ;              // owner is main window
  ofn.lpstrFilter = szFilter[0] ;     // filter string array
  ofn.lpstrFile = szPath ;            // path+name buffer
  ofn.nMaxFile = BUFFER_SIZE ;        // size of above
  ofn.lpstrFileTitle = szTitle ;      // file name buffer
  ofn.nMaxFileTitle = BUFFER_SIZE ;   // size of above
  ofn.Flags = OFN_PATHMUSTEXIST | OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT ;
  if(!GetSaveFileName (&ofn))         // get the path+name
    return NULL;                      // user pressed Cancel
  hpImageOut = (char huge *)GlobalLock (hImageOut) ;
  hFileOut = _lcreat (szPath, 0) ;
  if (!hFileOut)
    {
      MessageBox(hWnd, "Error Opening Output File", szProgName, MB_SYSTEMMODAL | MB_OK) ;
      return NULL ;
    }
  if (dwImageSize == 65536L)
    dwBytesWrite = _hwrite (hFileOut, s256x256, sizeof (s256x256)) ;
  else
    dwBytesWrite = _hwrite (hFileOut, s512x512, sizeof (s512x512)) ;
  if (dwBytesWrite != sizeof (s256x256) ||
      _hwrite (hFileOut, hpImageOut, dwImageSize) != (dwImageSize))
    {
      MessageBox(hWnd, "Error Writing Output File", szProgName, MB_SYSTEMMODAL | MB_OK) ;
      return NULL ;
    }
  _lclose (hFileOut) ;
  GlobalUnlock (hImageOut) ;
  return 1 ;
}
```

```
//
// Assembly coded routine to perform a DCT on a section of image.
//
void BlockDCT (char huge *source, int huge *destination)
{
  WORD Count1 = BlockSize * BlockSize + 131 ;
  WORD Count2 = TRANSFORM_BUFFER_SIZE - Count1 ;
        asm         PUSH   DS              // Save segment registers.
        asm         PUSH   ES
        asm         LDS    SI,source       // Get the source address.
        asm         LES    DI,destination  // Get the destination address
        asm         MOV    DX,DCT_data     // Point to the STV3200 data register.

        asm         CLD
        asm         MOV    CX,Count1       // Move the entire array to the STV3200.
        asm         DEC    CX

        asm         MOV    AL,[SI]         // Get a byte from the source.
        asm         INC    SI
        asm         SUB    AL,0x80         // Add a negative bias to input data.
        asm         CBW                    // Convert it to a signed word.
        asm         AND    AH,0x01         // First word needs bit 9 low.
        asm         OUT    DX,AX           // Output the data to the STV3200.
  MOVE1: asm        MOV    AL,[SI]         // Get a byte from the source.
        asm         INC    SI
        asm         SUB    AL,0x80         // Add a negative bias to input data.
        asm         CBW                    // Convert it to a signed word.
        asm         OR     AH,0x02         // Successive words need bit 9 set.
        asm         OUT    DX,AX           // Output the data word to the STV3200.
        asm         LOOP   MOVE1           // Repeat until entire block sent.

        asm         MOV    CX,Count2
  MOVE3: asm        IN     AX, DX          // Get the data from the STV3200.
        asm         TEST   AH, 0x08        // Perform sign extension of the 12 bit data
        asm         JE     MOVEP
        asm         OR     AH, 0xF0
        asm         JMP    MOVE4
  MOVEP: asm        AND    AH, 0x0F
  MOVE4: asm        MOV    ES:[DI],AX
        asm         ADD    DI,2
        asm         MOV    AL,[SI]         // Get a byte from the source.
        asm         INC    SI
        asm         SUB    AL,0x80         // Add a negative bias to input data.
        asm         CBW                    // Convert it to a signed word.
        asm         OR     AH,0x02         // Successive words need bit 9 set.
        asm         OUT    DX,AX           // Output the data word to the STV3200.
        asm         LOOP   MOVE3

        asm         MOV    CX,Count1
        asm         MOV    BX,0x0200
  MOVE5: asm        IN     AX,DX           // Get the data from the STV3200.
        asm         TEST   AH,0x08         // Perform sign extension of the 12 bit data
        asm         JE     MOVEO
        asm         OR     AH,0xF0
        asm         JMP    MOVE6
  MOVEO: asm        AND    AH,0x0F
  MOVE6: asm        MOV    ES:[DI],AX
        asm         ADD    DI,2
        asm         MOV    AX,BX
        asm         OUT    DX,AX           // A dummy write to clock next data out.
        asm         LOOP   MOVE5

        asm         POP    ES              // Restore registers.
        asm         POP    DS
}
```

```
//
// Assembly coded routine to perform an IDCT on a section of image coefficients.
//
void BlockIDCT (int huge *source, char huge *destination)
{
  WORD Count1 = BlockSize * BlockSize + 131 ;
  WORD Count2 = TRANSFORM_BUFFER_SIZE - Count1 ;
        asm         PUSH   DS              // Save segment registers.
        asm         PUSH   ES
        asm         LDS    SI,source       // Get the source address.
        asm         LES    DI,destination  // Get the destination address.
        asm         MOV    DX,DCT_data

        asm         CLD
        asm         MOV    CX,Count1       // Move the entire array to the STV3200.
        asm         DEC    CX

        asm         MOV    AX,[SI]         // Get a word from the source.
        asm         ADD    SI,2
        asm         AND    AH,0x0F         // Bit 12 must be zero for first word.
        asm         OUT    DX,AX           // Output the word.
  MOVE1: asm        MOV    AX,[SI]         // Get a byte from the source.
        asm         ADD    SI,2
        asm         OR     AH,0x10         // Successive words have bit 12 set.
        asm         OUT    DX,AX           // Output a word.
        asm         LOOP   MOVE1           // Continue until entire block sent.

        asm         MOV    CX,Count2
  MOVE3: asm        IN     AX,DX           // Get the data from the STV3200.
        asm         TEST   AH,0x01         // Clip the 9 bit data to 8 bits.
        asm         JE     MOVE4
        asm         CMP    AL,0x7F
        asm         JA     MOVE5
        asm         MOV    AL,0x80
        asm         JMP    MOVE5
  MOVE4: asm        CMP    AL,0x80
        asm         JB     MOVE5
        asm         MOV    AL,0x7F
  MOVE5: asm        ADD    AL,0x80
        asm         MOV    ES:[DI],AL
        asm         INC    DI
        asm         MOV    AX,[SI]         // Get a byte from the source.
        asm         ADD    SI,2
        asm         OR     AH,0x10         // Bit 12 must be set
        asm         OUT    DX,AX           // Output the data word to the STV3200.
        asm         LOOP   MOVE3

        asm         MOV    BX,0x1000
        asm         MOV    CX,Count1
  MOVE2: asm        IN     AX,DX           // Get the results from the STV3200.
        asm         TEST   AH,0x01         // Clip the 9 bit data to 8 bits.
        asm         JE     MOVE6
        asm         CMP    AL,0x80
        asm         JA     MOVE7
        asm         MOV    AL,0x80
        asm         JMP    MOVE7
  MOVE6: asm        CMP    AL,0x80
        asm         JB     MOVE7
        asm         MOV    AL,0x7F
  MOVE7: asm        ADD    AL,0x80
        asm         MOV    ES:[DI],AL      // Store the data in the destination.
        asm         INC    DI
        asm         MOV    AX,BX
        asm         OUT    DX,AX           // Output a dummy value to get next word.
        asm         LOOP   MOVE2
        asm         POP    ES              // Restore registers.
        asm         POP    DS
}

//
// Order and dequantise the DCT coefficients after a run-length decode.
//
void OrderIDCT (int huge *hpSource, int huge *hpDestination, int blockStart)
{
  WORD i, block ;
  unsigned char x = 0 ;
  int huge *hpSrc ;

  block = blockStart ;
  for (i = 0, x = 0 ; i < TRANSFORM_BUFFER_SIZE ; i++, x++)
    {
      hpSrc = hpSource + blockorder [block] + coefforder [x] ;
      *hpDestination = *hpSrc ;
      *(hpDestination++) = hpBackward [quant_type [x]][*hpDestination + 2048] ;
      if (x == 255)
        block++ ;
    }
}
```

```
//
// Perform the reverse DCT on an entire image.
//
int ExecuteIDCT (HWND hWnd, int huge* hpDCT, char huge* hpImage)
{
  int y, x, yy, xx, blockNumber, blockStart ;
  unsigned int c ;
  int huge *hpLineIn ;
  HGLOBAL hLineIn, hLineOut ;
  char huge *hpLineOut ;
  char huge *l ;

  outportb (DCT_control, IDCT + DCTSetup) ;
  hLineIn = GlobalAlloc(GMEM_MOVEABLE, TRANSFORM_BUFFER_SIZE << 1) ;
  hLineOut = GlobalAlloc(GMEM_MOVEABLE, TRANSFORM_BUFFER_SIZE) ;
  if(!hLineIn || !hLineOut)
    {
      MessageBox(hWnd, "Can't allocate memory", szProgName, MB_SYSTEMMODAL | MB_OK) ;
      return NULL ;
    }
  hpLineIn = (int huge*)GlobalLock (hLineIn) ;
  hpLineOut = (char huge *)GlobalLock (hLineOut) ;
  blockNumber = (dwImageWidth / 16) * (dwImageWidth / 16) ;
  for (x = blockNumber - 1 ; x > 0 ; x--)
    hpDCT [x - 1] = hpDCT [x - 1] + hpDCT [x] ;
  for (y = c = blockNumber = blockStart = 0 ; y < dwImageWidth ; y += BlockSize)
    for (x = 0 ; x < dwImageWidth ; x += BlockSize)
      {
        if ((c % TRANSFORM_BUFFER_SIZE) == 0)
          {
            OrderIDCT (hpDCT, hpLineIn, blockNumber) ;
            BlockIDCT (hpLineIn, hpLineOut) ;
            l = hpLineOut ;
            blockStart = blockNumber + 1 ;
          }
        for (yy = 0 ; yy < BlockSize ; yy++)
          for (xx = 0 ; xx < BlockSize ; xx++)
            hpImage [(y + yy) * dwImageWidth + xx + x] = *(l++) ;
        blockNumber++ ;
        c += 256 ;
      }
  GlobalUnlock (hLineOut) ;
  GlobalFree (hLineOut) ;
  GlobalUnlock (hLineIn) ;
  GlobalFree (hLineIn) ;
  return 1 ;
}

//
// Quantise and order the coefficients in preparation for the run-length coder.
//
void OrderDCT (int huge *hpSource, int huge *hpDestination, int blockStart)
{
  WORD i, block ;
  unsigned char x = 0 ;
  int huge *hpDest ;

  block = blockStart ;
  for (i = 0 ; i < TRANSFORM_BUFFER_SIZE ; i++, x++)
    {
      hpDest = hpDestination + blockorder [block] + coefforder [x] ;
      *hpDest = *(hpSource++) ;
      *hpDest = hpForward [quant_type [x]][*hpDest + 2048] ;
      if (x == 255)
        block++ ;
    }
}
```

```
//
// Perform a forward DCT on the entire image.
//
int ExecuteDCT (HWND hWnd, char huge *hpImage, int huge *hpDCT)
{
  char huge *hpLineIn, huge *l ;
  int huge *hpLineOut ;
  int x, y, xx, yy, blockNumber, blockStart ;
  unsigned int c ;
  HGLOBAL hLineIn, hLineOut ;

  outportb (DCT_control, FDCT + DCTSetup) ;
  hLineIn = GlobalAlloc (GMEM_MOVEABLE, TRANSFORM_BUFFER_SIZE) ;
  hLineOut = GlobalAlloc (GMEM_MOVEABLE, TRANSFORM_BUFFER_SIZE << 1) ;
  if(!hLineIn || !hLineOut)
    {
      MessageBox(hWnd, "Can't allocate memory", szProgName, MB_SYSTEMMODAL | MB_OK) ;
      return NULL ;
    }
  l = hpLineIn = (char huge *)GlobalLock (hLineIn) ;
  hpLineOut = (int huge *)GlobalLock (hLineOut) ;
  for (y = c = blockNumber = blockStart = 0 ; y < dwImageWidth ; y += BlockSize)
    for (x = 0 ; x < dwImageWidth ; x += BlockSize)
      {
        for (yy = 0 ; yy < BlockSize ; yy++)
          for (xx = 0 ; xx < BlockSize ; xx++)
            *(l++) = hpImage [(y + yy) * dwImageWidth + xx + x] ;
        c += 256 ;
        if ((c % TRANSFORM_BUFFER_SIZE) == 0)
          {
            BlockDCT (l = hpLineIn, hpLineOut) ;
            OrderDCT (hpLineOut, hpDCT, blockStart) ;
            blockStart = blockNumber + 1 ;
          }
        blockNumber++ ;
      }
  blockNumber = (dwImageWidth / 16) * (dwImageWidth / 16) ;
  for (x = 1 ; x < blockNumber ; x++)
    hpDCT [x - 1] = hpDCT [x - 1] - hpDCT [x] ;
  GlobalUnlock (hLineOut) ;
  GlobalFree (hLineOut) ;
  GlobalUnlock(hLineIn) ;
  GlobalFree(hLineIn) ;
  return 1 ;
}

//
// Perform the full compression algorithm on an image.
//
int Compress (HWND hWnd)
{
  int huge *hpImageDCT ;
  char huge *hpImageIn, huge *hpImageRunLength, huge *hpImageCompressed ;

  hpImageDCT = (int huge *)GlobalLock (hImageDCT) ;
  hpImageIn = (char huge *)GlobalLock (hImageIn) ;
  ExecuteDCT (hWnd, hpImageIn, hpImageDCT) ;
  GlobalUnlock (hImageIn) ;
  hpImageRunLength = (char huge *)GlobalLock (hImageRunLength) ;
  dwRLCodeLength = RunLengthCode (hpImageDCT, hpImageRunLength, dwImageSize) ;
  GlobalUnlock (hImageDCT) ;
  hpImageCompressed = (char huge *)GlobalLock (hImageCompressed) ;
  dwCodeLength = StatisticalCode (hpImageRunLength, hpImageCompressed, dwRLCodeLength) ;
  GlobalUnlock (hImageCompressed) ;
  GlobalUnlock (hImageRunLength) ;
  ModifyMenus (hWnd, TRUE, FALSE, TRUE, TRUE, TRUE) ;
  bImageCompressed = TRUE ;
  bImageDecompressed = FALSE ;

  return 1 ;
}
```

```
//
// Perform the entire image decompression procedure for an image compressed by
// this algorithm.
//
int Decompress (HWND hWnd)
{
  int huge *hpImageDCT ;
  char huge *hpImageOut, huge *hpImageRunLength, huge *hpImageCompressed ;

  hpImageRunLength = (char huge *)GlobalLock (hImageRunLength) ;
  hpImageCompressed = (char huge *)GlobalLock (hImageCompressed) ;
  dwRLCodeLength = StatisticalDecode (hpImageCompressed, hpImageRunLength, dwCodeLength) ;
  GlobalUnlock (hImageCompressed) ;
  hpImageDCT = (int huge *)GlobalLock (hImageDCT) ;
  dwImageSize = RunLengthDecode (hpImageRunLength, hpImageDCT, dwRLCodeLength) ;
  hpImageOut = (char huge *)GlobalLock (hImageOut) ;
  GlobalUnlock (hImageRunLength) ;
  ExecuteIDCT (hWnd, hpImageDCT, hpImageOut) ;
  GlobalUnlock (hImageDCT) ;
  GlobalUnlock (hImageOut) ;
  ModifyMenus (hWnd, TRUE, TRUE, TRUE, TRUE, TRUE) ;
  bImageDecompressed = TRUE ;

  return 1 ;
}

//
// Process a command issued by a menu item.
//
void MenuCommand (HWND hWnd, WPARAM wMenuID)
{
  switch (wMenuID)
    {
      case IDM_FILEOPEN :
        OpenFile (hWnd) ;
        break ;
      case IDM_TOOLSCOMPRESS :
        Compress (hWnd) ;
        break ;
      case IDM_SAVECOMPRESSED :
        SaveCompressed (hWnd) ;
        break ;
      case IDM_SAVEDECOMPRESSED :
        SaveDecompressed (hWnd) ;
        break ;
      case IDM_TOOLSDECOMPRESS :
        Decompress (hWnd) ;
        break ;
      case IDM_TOOLSFULLCYCLE :
        Compress (hWnd) ;
        Decompress (hWnd) ;
        break ;
      case IDM_FILEEXIT :
        if (bImageLoaded)
          {
            GlobalFree (hImageOut) ;
            GlobalFree (hImageDCT) ;
            GlobalFree (hImageIn) ;
            GlobalFree (hImageRunLength) ;
            GlobalFree (hImageCompressed) ;
          }
        PostQuitMessage(0) ;
        break ;
    }
}
```

```
//
// main window procedure -- receives messages
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
  int i ;

  switch(msg)
    {
      case WM_CREATE :
        for (i = 0 ; i < 8 ; i++)
          {
            hForward [i] = GlobalAlloc (GMEM_MOVEABLE, 8192) ;
            hpForward [i] = (int huge *)GlobalLock (hForward [i]) ;
            hBackward [i] = GlobalAlloc (GMEM_MOVEABLE, 8192) ;
            hpBackward [i] = (int huge *)GlobalLock (hBackward [i]) ;
          }
        InitDCT () ;
        break ;
      case WM_COMMAND :
        MenuCommand (hWnd, wParam) ;
        break ;
      case WM_DESTROY :
        for (i = 0 ; i < 8 ; i++)
          {
            GlobalFree (hForward [i]) ;
            GlobalFree (hBackward [i]) ;
          }
        if (bImageLoaded)
          {
            GlobalFree (hImageOut) ;
            GlobalFree (hImageDCT) ;
            GlobalFree (hImageIn) ;
            GlobalFree (hImageRunLength) ;
            GlobalFree (hImageCompressed) ;
          }
        PostQuitMessage(0) ;
        break ;
       default :
        return (DefWindowProc(hWnd, msg, wParam, lParam)) ;
    }
  return 0L ;
}
```

```
//
// Windows main program.
//
int PASCAL WinMain(HINSTANCE hInstance,                    // which program are we?
        HINSTANCE hPrevInst,        // is there another one?
        LPSTR lpCmdLine,            // command line arguments
        int nCmdShow)               // window size (icon, etc)
{
  HWND hWnd;                        // window handle from CreateWindow
  MSG msg;                          // message from GetMessage
  WNDCLASS wndclass;                // window class structure

  if(!hPrevInst)                    // if this is first such window
    {
      wndclass.style         = CS_HREDRAW | CS_VREDRAW;  // style
      wndclass.lpfnWndProc   = (WNDPROC)WndProc;         // WndProc address
      wndclass.cbClsExtra    = 0;    // no extra class data
      wndclass.cbWndExtra    = 0;    // no extra window data
      wndclass.hInstance     = hInstance;                // which program?
                               // stock arrow cursor
      wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW);
                               // stock blank icon
      wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
      wndclass.lpszMenuName  = szProgName;               // menu name
                               // white background
      wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
      wndclass.lpszClassName = szProgName;               // window class name

      RegisterClass(&wndclass);     // register the class
    }                               // end if

  hWnd = CreateWindow(szProgName,   // window class name
            szProgName,             // caption
            WS_OVERLAPPEDWINDOW,    // style
            CW_USEDEFAULT,          // default x position
            CW_USEDEFAULT,          // default y position
            CW_USEDEFAULT,          // default width
            CW_USEDEFAULT,          // default height
            NULL,                   // parent's handle
            NULL,                   // menu handle
            hInstance,              // which program?
            NULL);                  // no init data

  ShowWindow(hWnd, nCmdShow);       // make window visible

   // message loop
  while( GetMessage(&msg,0,0,0) )   // get message from Windows
    {
      TranslateMessage(&msg);       // convert keystrokes
      DispatchMessage(&msg);        // call windows procedure
    }
  return msg.wParam;                // return to Windows
}                                   // end WinMain
```

Huffman Coder - Fixed

```
//
//
//                                Fixed Huffman Coder
//
//                  Department of Electrical and Electronic Engineering
//                          Victoria University of Technology
//                                  (Footscray Campus)
//                                    P.O. Box 14428,
//                                 Melbourne Mail Centre,
//                                    Melbourne, 3000.
//
//      Date          : July 7, 1994
//
//      Author        : Emil Lenc
//
//      File Name     : hufffix.cpp
//
//      Supervisors   : Alec Simcock & Ann Pleasants
//
//      Internet      : emil@cabsav.vut.edu.au
//
//

// Definition of Huffman Codes

unsigned int sym [256] = {
  3    ,0    ,8    ,3    ,23   ,56   ,42   ,118  ,19   ,105  ,96   ,34   ,233  ,216  ,78   ,214  ,
  145  ,439  ,319  ,303  ,936  ,871  ,836  ,599  ,282  ,1836 ,1724 ,1271 ,1269 ,562  ,561  ,3757 ,
  1404 ,3678 ,1834 ,1835 ,3719 ,3351 ,2545 ,2536 ,2805 ,2541 ,2394 ,7501 ,7428 ,7350 ,7431 ,7430 ,
  3676 ,7427 ,7335 ,7355 ,7354 ,7426 ,6697 ,5629 ,6908 ,5628 ,5609 ,5622 ,2317 ,1266 ,1157 ,6909 ,
  3349 ,1152 ,5623 ,6696 ,2253 ,2242 ,14669,2264 ,15006,14663,15033,13805,14848,14019,4626 ,14703,
  2551 ,14018,9583 ,13803,13403,13800,6903 ,15035,14021,14849,14660,11263,11262,4613 ,15007,3664 ,
  2530 ,2265 ,9255 ,14662,9272 ,10149,4533 ,4483 ,10112,9275 ,29434,9581 ,9278 ,30069,30064,9229 ,
  2548 ,29435,10119,30009,29719,29322,430  ,38   ,877  ,296  ,434  ,150  ,218  ,98   ,41   ,25   ,
  195  ,194  ,119  ,16   ,172  ,174  ,302  ,297  ,465  ,7011 ,1858 ,26804,20236,20237,10113,29748,
  2544 ,9225 ,29433,29323,4482 ,28041,29749,26805,28040,4535 ,9254 ,10114,4534 ,13401,30008,10184,
  2531 ,30068,29337,29718,4532 ,10187,30065,10139,9279 ,11240,29336,29432,9582 ,9224 ,9228 ,9273 ,
  2807 ,10186,9274 ,10148,11261,10138,10115,11216,9580 ,10198,10185,10199,13400,11241,11217,11260,
  2547 ,4615 ,13801,14858,13804,13802,14875,14702,2252 ,15005,5058 ,2240 ,2243 ,5098 ,4638 ,5068 ,
  3455 ,5075 ,5613 ,5621 ,5612 ,7359 ,7425 ,7008 ,7500 ,7436 ,2316 ,1127 ,2312 ,2535 ,2540 ,2550 ,
  1196 ,3666 ,3674 ,3756 ,3759 ,567  ,1406 ,1726 ,1753 ,1874 ,577  ,700  ,870  ,938  ,298  ,419  ,
  71   ,208  ,73   ,173  ,228  ,235  ,99   ,106  ,18   ,115  ,40   ,55   ,22   ,2    ,5    ,15   } ;

// Huffman Code Lengths

char len [256] = {
  3 ,3 ,4 ,4 ,5 ,6 ,6 ,7 ,6 ,7 ,7 ,7 ,8 ,8 ,7 ,8 ,8 ,9 ,9 ,9 ,10,10,10,10,10,11,11,11,11,11,11,12,
  11,12,11,11,12,12,12,12,12,12,12,13,13,13,13,13,12,13,13,13,13,13,13,13,13,13,13,13,12,11,11,13,
  12,11,13,13,13,13,14,13,14,14,14,14,14,14,13,14,12,14,14,14,14,14,13,14,14,14,14,14,14,13,14,12,
  12,13,14,14,14,14,14,14,14,14,15,14,14,15,15,14,12,15,14,15,15,15,9 ,6 ,10,9 ,9 ,8 ,8 ,7 ,6 ,5 ,
  8 ,8 ,7 ,6 ,8 ,8 ,9 ,9 ,9 ,13,11,15,15,15,15,14,15,12,14,15,15,14,15,15,15,15,14,14,14,14,14,15,14,
  12,15,15,15,14,14,15,14,14,14,15,15,14,14,14,14,12,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,
  12,13,14,14,14,14,14,14,13,14,13,13,13,13,13,13,12,13,13,13,13,13,13,13,13,13,12,12,12,12,12,12,
  11,12,12,12,12,11,11,11,11,11,10,10,10,10,9 ,9 ,8 ,8 ,7 ,8 ,8 ,8 ,7 ,7 ,6 ,7 ,6 ,6 ,5 ,4 ,4 ,4 } ;

unsigned long int out_buffer = 0 ;          // Temporary output buffer
int buf_length = 0 ;                        // Number of bits stored in temporary output buffer.
unsigned char convert [16][2048] ;          // Conversion from Huffman code to original symbol.

// Output the required code of given length to the destination buffer.

void output (unsigned char huge **dest, unsigned long int S, int L)
{
  out_buffer <<= L ;
  out_buffer |= S ;
  buf_length += L ;
  while (buf_length > 7) {
    mput (*dest, out_buffer >> (buf_length - 8)) ;
    buf_length -= 8 ;
  }
}

// Output any remaining bits to the output buffer.

void flush_buffer (unsigned char huge **dest)
{
  out_buffer <<= 11 ;
  out_buffer |= 0x72C ;
  buf_length += 11 ;
  mput (*dest, out_buffer >> (buf_length - 8)) ;
  out_buffer = 0 ;
  buf_length = 0 ;
}
```

```
// Perform the Huffman coding on the given input source of given size.

DWORD StatisticalCode (unsigned char huge *source, unsigned char huge *destination, DWORD length)
{
  unsigned char huge *d ;
  unsigned char c ;
  unsigned long int total = 0 ;

  out_buffer = 0 ;                          // Clear the temporary output buffer.
  buf_length = 0 ;
  d = destination ;                         // Make a copy of the destination pointer.

  while (1==1) {
    if (total == length)                    // If no more to read then exit
      break ;
    c = mget (source) ;                     // Get the next byte.
    total++ ;
    output (&d, sym [c], len [c]) ;         // Convert the byte to the appropriate code.
  }
  if (buf_length)                           // Anything still left in the buffer ?
    flush_buffer (&d) ;                     // If so flush it out

  return (DWORD)(d - destination) ;         // Return the length of the coded data.
}

// Perform the Huffman decoding on the given input source of given length.

DWORD StatisticalDecode (unsigned char huge *source, unsigned char huge *destination, DWORD length)
{
  int i, j ;
  unsigned char huge *d ;
  unsigned char c ;

  d = destination ;                         // Make a copy of the destination pointer.
  out_buffer = 0 ;                          // Clear the temporary output buffer.
  buf_length = 0 ;
  for (i = 0 ; i < 16 ; I++)                // Set up the conversion table.
    for (j = 0 ; j < 2048 ; j++)
      convert [i][j] = 0 ;
  for (j = 0 ; j < 256 ; j++) {
    i = convert [len[j]][sym[j] & 0x7FF] ;
    if (i == 0)
      convert [len[j]][sym[j] & 0x7FF] = j ;
  }
  while (1==1) {
    c = mget (source) ;                     // Read data from the input buffer
    if (length-- == 0)
      break ;
    for (i = 0 ; i < 8 ; i++) {             // Search for a valid code
      out_buffer = (out_buffer << 1) | ((c & 0x80)?1:0) ;
      c <<= 1 ;
      buf_length++ ;
      if (buf_length == 3) {
        if (out_buffer == 3) {
          mput (d, 0) ;
          buf_length = 0 ;
          out_buffer = 0 ;
        } else if (out_buffer == 0) {
          mput (d, 1) ;
          buf_length = 0 ;
          out_buffer = 0 ;
        }
      } else if (convert [buf_length][out_buffer & 0x7FF] != 0) {
        j = convert [buf_length][out_buffer & 0x7FF] ;
        mput (d, j) ;
        buf_length = 0 ;
        out_buffer = 0 ;
      }
    }
  }
  return (DWORD)(d - destination) ;
}
```

Huffman Coder - Adaptive

```
//
//
//                                Adaptive Huffman Coder
//
//                    Department of Electrical and Electronic Engineering
//                            Victoria University of Technology
//                                  (Footscray Campus)
//                                    P.O. Box 14428,
//                                Melbourne Mail Centre,
//                                   Melbourne, 3000.
//
//      Date          : July 7, 1994
//
//      Author        : Emil Lenc
//
//      File Name     : huffadap.cpp
//
//      Supervisors   : Alec Simcock & Ann Pleasants
//
//      Internet      : emil@cabsav.vut.edu.au
//
//

// Define the Huffman symbol codes.

unsigned int sym [256] = {
  3    ,0    ,8    ,3    ,2    ,5    ,15   ,23   ,25   ,22   ,56   ,42   ,19   ,38   ,41   ,16   ,
  18   ,40   ,55   ,118  ,105  ,96   ,34   ,78   ,98   ,119  ,73   ,99   ,106  ,115  ,233  ,216  ,
  214  ,145  ,150  ,218  ,195  ,194  ,172  ,174  ,71   ,208  ,173  ,228  ,235  ,439  ,319  ,303  ,
  430  ,296  ,434  ,302  ,297  ,465  ,298  ,419  ,936  ,871  ,836  ,599  ,282  ,877  ,577  ,700  ,
  870  ,938  ,1836 ,1724 ,1271 ,1269 ,562  ,561  ,1404 ,1834 ,1835 ,1266 ,1157 ,1152 ,1858 ,1196 ,
  567  ,1406 ,1726 ,1753 ,1874 ,3757 ,3678 ,3719 ,3351 ,2545 ,2536 ,2805 ,2541 ,2394 ,3676 ,2317 ,
  3349 ,2551 ,3664 ,2530 ,2548 ,2544 ,2531 ,2807 ,2547 ,3455 ,2316 ,1127 ,2312 ,2535 ,2540 ,2550 ,
  3666 ,3674 ,3756 ,3759 ,7501 ,7428 ,7350 ,7431 ,7430 ,7427 ,7335 ,7355 ,7354 ,7426 ,6697 ,5629 ,
  6908 ,5628 ,5609 ,5622 ,6909 ,5623 ,6696 ,2253 ,2242 ,2264 ,4626 ,6903 ,4613 ,2265 ,7011 ,4615 ,
  2252 ,5058 ,2240 ,2243 ,5098 ,4638 ,5068 ,5075 ,5613 ,5621 ,5612 ,7359 ,7425 ,7008 ,7500 ,7436 ,
  14669,15006,14663,15033,13805,14848,14019,14703,14018,9583 ,13803,13403,13800,15035,14021,14849,
  14660,11263,11262,15007,9255 ,14662,9272 ,10149,4533 ,4483 ,10112,9275 ,9581 ,9278 ,9229 ,10119,
  10113,9225 ,4482 ,4535 ,9254 ,10114,4534 ,13401,10184,4532 ,10187,10139,9279 ,11240,9582 ,9224 ,
  9228 ,9273 ,10186,9274 ,10148,11261,10138,10115,11216,9580 ,10198,10185,10199,13400,11241,11217,
  11260,13801,14858,13804,13802,14875,14702,15005,29434,30069,30064,29435,30009,29719,29322,26804,
  20236,20237,29748,29433,29323,28041,29749,26805,28040,30008,30068,29337,29718,30065,29336,29432} ;

// Define the statistical qualities of the image test set.

DWORD P [256] = {
  150224L,136944L,135484L,79162L ,77323L ,71507L ,70378L ,49048L ,
  46263L ,45308L ,29924 ,29221 ,21927 ,21158 ,21090 ,20346 ,
  18545 ,18249 ,17184 ,17133 ,16984 ,15996 ,12590 ,12526 ,
  11808 ,11704 ,11460 ,10240 ,9525  ,8743  ,8734  ,8227  ,
  7370  ,6850  ,6617  ,6334  ,5988  ,5844  ,5791  ,5663  ,
  5621  ,5591  ,5040  ,4830  ,4614  ,3943  ,3554  ,3345  ,
  3199  ,3167  ,2804  ,2535  ,2530  ,2502  ,2484  ,2381  ,
  2086  ,1994  ,1786  ,1709  ,1697  ,1506  ,1386  ,1273  ,
  1150  ,1112  ,1013  ,975   ,928   ,920   ,916   ,879   ,
  832   ,810   ,738   ,732   ,677   ,660   ,648   ,609   ,
  586   ,572   ,567   ,542   ,542   ,535   ,530   ,524   ,
  506   ,473   ,470   ,465   ,452   ,438   ,434   ,400   ,
  384   ,358   ,357   ,348   ,346   ,343   ,339   ,339   ,
  339   ,337   ,335   ,329   ,326   ,321   ,321   ,309   ,
  298   ,297   ,291   ,276   ,256   ,253   ,247   ,246   ,
  244   ,240   ,240   ,239   ,239   ,237   ,236   ,236   ,
  233   ,229   ,219   ,217   ,215   ,210   ,210   ,191   ,
  190   ,187   ,187   ,185   ,184   ,182   ,181   ,177   ,
  175   ,172   ,165   ,161   ,160   ,152   ,147   ,145   ,
  144   ,140   ,140   ,137   ,137   ,137   ,136   ,134   ,
  134   ,133   ,130   ,130   ,129   ,126   ,120   ,119   ,
  119   ,117   ,117   ,114   ,112   ,111   ,109   ,109   ,
  109   ,109   ,108   ,102   ,102   ,102   ,100   ,98    ,
  98    ,97    ,95    ,95    ,95    ,94    ,94    ,92    ,
  90    ,88    ,87    ,87    ,87    ,86    ,85    ,84    ,
  84    ,83    ,82    ,81    ,81    ,81    ,80    ,80    ,
  80    ,80    ,79    ,79    ,78    ,78    ,77    ,76    ,
  76    ,76    ,75    ,75    ,74    ,74    ,73    ,72    ,
  72    ,71    ,70    ,70    ,70    ,70    ,68    ,67    ,
  67    ,67    ,66    ,65    ,64    ,64    ,63    ,63    ,
  62    ,60    ,60    ,59    ,59    ,59    ,57    ,57    ,
  56    ,55    ,55    ,54    ,53    ,45    ,43    ,37    } ;
```

```
// Define the size of the Huffman codes.

char len [256] = {
  3 ,3 ,4 ,4 ,4 ,4 ,4 ,5 ,5 ,5 ,6 ,6 ,6 ,6 ,6 ,6 ,6 ,6 ,7 ,7 ,7 ,7 ,7 ,7 ,7 ,7 ,7 ,7 ,8 ,8 ,
  8 ,8 ,8 ,8 ,8 ,8 ,8 ,8 ,8 ,8 ,8 ,8 ,9 ,9 ,9 ,9 ,9 ,9 ,9 ,9 ,9 ,9 ,9 ,10,10,10,10,10,10,10,10,
  10,10,11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,12,12,12,12,12,12,12,12,12,12,
  12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,13,13,13,13,13,13,13,13,13,13,13,13,
  13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,
  14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,
  14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,
  14,14,14,14,14,14,14,14,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15} ;

// Define the initial index to symbol convertion table.

unsigned char init_ind2sym [256] = {
  0  ,1  ,255,2  ,254,3  ,253,127,4  ,252,5  ,251,6  ,126,250,119,
  8  ,248,131,130,7  ,249,247,9  ,246,125,10 ,14 ,242,11 ,245,12 ,
  244,124,13 ,15 ,241,128,129,133,243,132,123,16 ,240,136,17 ,122,
  118,239,18 ,19 ,134,238,135,121,237,20 ,120,21 ,236,22 ,235,23 ,
  234,24 ,233,138,25 ,35 ,34 ,232,231,26 ,230,32 ,27 ,28 ,61 ,224,
  62 ,65 ,229,30 ,29 ,228,31 ,227,36 ,33 ,48 ,226,225,95 ,208,37 ,
  64 ,176,40 ,80 ,223,112,144,38 ,192,41 ,222,39 ,221,96 ,160,42 ,
  60 ,218,220,219,43 ,216,217,46 ,47 ,44 ,49 ,53 ,214,213,52 ,51 ,
  45 ,50 ,137,215,63 ,56 ,86 ,54 ,67 ,57 ,55 ,66 ,59 ,211,210,212,
  58 ,205,209,207,202,206,78 ,193,93 ,97 ,71 ,200,68 ,204,69 ,203,
  87 ,74 ,72 ,94 ,201,198,195,76 ,89 ,199,79 ,70 ,73 ,99 ,81 ,90 ,
  77 ,88 ,75 ,83 ,196,197,194,85 ,84 ,157,91 ,188,92 ,191,180,189,
  169,190,187,183,185,165,177,159,186,101,179,114,167,181,182,104,
  142,155,82 ,172,107,184,168,178,108,105,100,175,154,98 ,111,174,
  145,173,153,164,156,102,103,148,109,161,166,110,115,158,150,143,
  116,163,113,146,106,171,170,162,147,117,149,152,151,139,141,140 } ;

unsigned long int out_buffer = 0 ;
int buf_length = 0 ;
int char_to_index [256] ;                    // To index from character
unsigned char index_to_char [256] ;          // To character from index
unsigned int freq [256] ;
int huge convert [13][2048] ;

// Initialise the Huffman coder variables.

void init_coder ()
{
  int i ;

  for (i = 0 ; i < 256 ; i++) {
    index_to_char [i] = init_ind2sym [i] ;
    char_to_index [index_to_char [i]] = i ;
    freq [i] = P [i] >> 8 ;
  }
}

// Update the statistics of the current image data.

void update_freq (int s)
{
  int s2 ;
  unsigned int temp1, temp2 ;

  if (freq [s] == 1024)                      // Check if rescaling is required to forget
    for (s2 = 0 ; s2 < 256 ; s2++)           // older data which is now not used.
      freq [s2] >>= 1 ;
  for (s2 = s ; freq [s2] == freq [s2 - 1] ; s2--)
    ;                                        // Check if data position should be moved.
  if (s2 < s) {                              // If so, then move it.
    temp1 = index_to_char [s] ;
    temp2 = index_to_char [s2] ;
    index_to_char [s] = temp2 ;
    index_to_char [s2] = temp1 ;
    char_to_index [temp1] = s2 ;
    char_to_index [temp2] = s ;
  }
  freq [s2]++ ;                              // Update the frequency of the current symbol.
}

// Output the code of given length to the output buffer.

void output (unsigned char huge **dest, unsigned long int S, int L)
{
  out_buffer <<= L ;
  out_buffer |= S ;
  buf_length += L ;
  while (buf_length > 7) {
    mput (*dest, out_buffer >> (buf_length - 8)) ;
    buf_length -= 8 ;
  }
}
```

```
// Flush out any remaining bits in the output buffer.

void flush_buffer (unsigned char huge **dest)
{
  out_buffer <<= 11 ;
  out_buffer |= 0x72C ;
  buf_length += 11 ;
  mput (*dest, out_buffer >> (buf_length - 8)) ;
  out_buffer = 0 ;
  buf_length = 0 ;
}

// Perform the Huffman coding of the given source data of given length.

DWORD StatisticalCode (unsigned char huge *source, unsigned char huge *destination, DWORD length)
{
  unsigned char huge *d ;
  unsigned char c ;
  unsigned long int total = 0 ;

  out_buffer = 0 ;                          // Clear the temporary output buffer.
  buf_length = 0 ;
  d = destination ;                         // Make a copy of the destination pointer.
  init_coder () ;                           // Initialise the coder
  while (1==1) {
    if (total == length)                    // If no more data then exit
      break ;
    c = mget (source) ;                     // Get a byte from the input buffer.
    c = char_to_index [c] ;                 // Convert to an index.
    update_freq (c) ;                       // Update the image statistics.
    total++ ;
    output (&d, sym [c], len [c]) ;         // Output the Huffman Code.
  }
  if (buf_length)                           // Any more bits in the output buffor ?
    flush_buffer (&d) ;                     // If so, then flush them out.

  return (DWORD)(d - destination) ;         // Return the length of the coded data.
}

// Perform the Huffman decoding of the given source data of given length.

DWORD StatisticalDecode (unsigned char huge *source, unsigned char huge *destination, DWORD length)
{
  int i, j ;
  unsigned char huge *d ;
  unsigned char c ;

  d = destination ;                         // Make a copy of the destination pointer.
  init_coder () ;                           // Initialise the coder.
  out_buffer = 0 ;                          // Clear the temporary output buffer.
  buf_length = 0 ;
  for (i = 0 ; i < 13 ; I++)                // Initialise the conversion table
    for (j = 0 ; j < 2048 ; j++)
      convert [i][j] = -1 ;
  for (j = 0 ; j < 256 ; j++)
    convert [len[j]-3][sym[j] & 0x7FF] = j ;
  while (1==1) {
    c = mget (source) ;                     // Get a byte from the input source
    if (length-- == 0)
      break ;
    for (i = 0 ; i < 8 ; i++) {             // Search for a Huffman code
      out_buffer = (out_buffer << 1) | ((c & 0x80)?1:0) ;
      c <<= 1 ;
      buf_length++ ;
      if (buf_length > 2)
        if (convert [buf_length-3][out_buffer & 0x7FF] != -1) {
              j = convert [buf_length-3][out_buffer & 0x7FF] ;
              mput (d, index_to_char [j]) ;
          update_freq (j) ;
              buf_length = 0 ;
              out_buffer = 0 ;
        }
    }
  }
  return (DWORD)(d - destination) ;
}
```

Arithmetic Coder - Fixed

```
//
//
//                              Fixed Arithmetic Coder
//
//                 Department of Electrical and Electronic Engineering
//                         Victoria University of Technology
//                                  (Footscray Campus)
//                                    P.O. Box 14428,
//                                 Melbourne Mail Centre,
//                                    Melbourne, 3000.
//
//      Date         : July 7, 1994
//
//      Author       : Emil Lenc
//
//      File Name    : aritfix.cpp
//
//      Supervisors  : Alec Simcock & Ann Pleasants
//
//      Internet     : emil@cabsav.vut.edu.au
//
//

// Size of Arithmetic Code Values.

#define Code_value_bits 16                      // Number of bits in a code value
typedef  DWORD code_value ;                     // Type of an arithmetic code value

#define Top_value (((DWORD)1<<Code_value_bits)-1)       // Largest code value

// Half and quarter points in the code value range.

#define First_qtr (Top_value/4+1)               // Point after first quarter
#define Half      (2*First_qtr)                 // Point after first half
#define Third_qtr (3*First_qtr)                 // Point after third quarter

// The set of symbols that may be encoded.

#define No_of_chars 256                         // Number of character symbols
#define EOF_symbol No_of_chars                  // Index of EOF symbol
#define No_of_symbols (No_of_chars+1)           // Total number of symbols
#define Max_frequency 16383                     // Maximum allowed frequency count 2^14-1

// Cumulative frequency table

int cum_freq [No_of_symbols + 1] ;              // Cumulative symbol frequencies

// Bit input routines

int bit_buffer ;                                // Bits waiting to be input
int bits_to_go ;                                // Number of bits still in buffer
int garbage_bits ;                              // Number of bits past end-of-file

code_value low, high ;                          // Ends of the current code region
code_value value ;                              // Currently-seen code value
int bits_to_follow ;                            // Number of opposite bits to output after
                                                // the next bit.
int freq [No_of_symbols] = {                    // Symbol frequencies
  1767,1611,931,841,544,352,257,199,218,147,134,102,96,77,120,74,
  56,41,32,29,23,20,17,14,13,10,9,7,7,6,6,6,
  8,5,10,10,5,4,3,3,4,3,3,3,2,2,2,2,
  5,2,2,2,2,2,2,2,2,2,2,3,7,6,2,
  4,6,2,2,1,1,1,1,1,1,1,1,1,1,1,
  4,1,1,1,1,1,2,1,1,1,1,1,1,1,1,5,
  3,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  4,1,1,1,1,1,37,239,21,28,39,59,80,137,248,577,
  68,68,201,202,65,66,29,29,46,2,11,1,1,1,1,1,
  3,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  3,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  4,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  3,1,1,1,1,1,1,1,1,1,1,1,2,1,1,
  5,1,2,2,2,2,2,2,2,3,3,3,3,3,4,
  7,5,5,6,6,6,8,9,10,11,13,16,19,24,29,37,
  54,70,112,66,86,102,138,148,214,188,248,343,533,827,909,1593,1 } ;


//
// Initialize the arithmetic encoder output stream.
//
void start_outputing_bits ()
{
  bit_buffer = 0 ;                              // Buffer is empty to start with
  bits_to_go = 8 ;
}
```

```
//
// Output a bit to the destination buffer.
//
void output_bit (DWORD bit, unsigned char huge **destination)
{
  bit_buffer >>= 1 ;                          // Shift the current bits down one bit.
  if (bit)
    bit_buffer |= 0x80 ;                      // If new bit is a '1' add it to buffer.
  if (!(--bits_to_go))                        // Is the byte buffer full ?
    {                                         // Output buffer to destination if it is.
      mput(*destination, bit_buffer) ;
      bits_to_go = 8 ;
    }
}

//
// Flush out any remaining bits upon completion of the arithmetic coding process.
//
void done_outputing_bits (unsigned char huge **destination)
{
  mput(*destination, bit_buffer >> bits_to_go) ;
}

//
// Output bits plus following opposite bits.
//
void bit_plus_follow (DWORD bit, unsigned char huge **destination)
{
  output_bit (bit, destination) ;             // Output the bit.
  while (bits_to_follow)
    {
      output_bit (!bit, destination) ;        // Output bits to follow opposite bits.
      bits_to_follow-- ;                      // Set bits_to_follow to zero.
    }
}

//
// Initialize the encoder.
//
void start_encoding ()
  {
    low = 0 ;                                 // Full code range.
    high = Top_value ;
    bits_to_follow = 0 ;                      // No bits to follow next.
  }

//
// Encode the given symbol into the destination buffer.
//
void encode_symbol (int symbol, int cum_freq[], unsigned char huge **destination)
{
  DWORD range ;                               // Size of the current code region.

  range = (DWORD)(high - low) + 1 ;
                                              // Narrow the region for the current symbol.
  high = low + (range * cum_freq [symbol]) / cum_freq [0] - 1 ;
  low = low + (range * cum_freq [symbol + 1]) / cum_freq [0] ;
  for (;;)
    {                                         // Loop to output bits.
      if (high < Half)
        bit_plus_follow (0, destination) ;    // Output 0 if in low half.
      else
        if (low >= Half)
          {                                   // Output 1 if in high half.
            bit_plus_follow (1, destination) ;
            low -= Half ;
            high -= Half ;                    // Subtract offset to top.
          }
        else
          if (low >= First_qtr && high < Third_qtr)
            {                                 // Output opp. bit later if in middle half.
              bits_to_follow++ ;
              low -= First_qtr ;              // Subtract offset to middle.
              high -= First_qtr ;
            }
          else
            break ;                           // Otherwise exit loop.
      low <<= 1 ;
      high = (high << 1) + 1 ;                // Scale up code range.
    }
}
```

```
//
// Initialize the adaptive source model.
//
void start_fixed_model()
{
  int i ;

  cum_freq [No_of_symbols] = 0 ;
  for (i = No_of_symbols-1 ; i >= 0 ; i--)        // Reset the frequency counts.
      cum_freq [i] = cum_freq [i + 1] + freq [i] ;
}

//
// Complete the encoding process.
//
void done_encoding (unsigned char huge **destination)
{
  bits_to_follow++ ;                              // Output two bits that select the quarter
  if (low < First_qtr)                            // that the current code range contains.
    bit_plus_follow (0, destination) ;
  else
    bit_plus_follow (1, destination) ;
}

//
// Perform arithmetic coding of the source data of length 'length' and store the coded
// results in the destination buffer.
//
DWORD StatisticalCode (unsigned char huge *source, unsigned char huge *destination, DWORD length)
{
  int symbol ;
  unsigned char huge *d ;

  d = destination ;                               // Make a copy of the destination pointer.
  start_fixed_model () ;                          // Initialize the arithmetic coder.
  start_outputing_bits () ;
  start_encoding () ;
  for (;;)                                        // Loop through the input buffer.
    {
      if (!(length--))                            // Has all the data been coded ?
        break ;                                   // If so then exit.
      symbol = mget(source) ;                     // Get the next available byte of data.
      encode_symbol (symbol, cum_freq, &d) ;      // Encode that symbol.
    }
  encode_symbol (EOF_symbol, cum_freq, &d) ;      // Encode the EOF symbol.
  done_encoding (&d) ;                            // Send the last few bits.
  done_outputing_bits (&d) ;
  return (DWORD)(d - destination) ;               // Return the length of the coded data.
}

//
// Initialize the arithmetic decoder.
//
void start_inputing_bits ()
{
  bits_to_go = 0 ;                                // Buffer starts out with no bits in it.
  garbage_bits = 0 ;
}

//
// Get a bit from the source buffer.
//
int input_bit (unsigned char huge **source)
{
  int t ;

  if (!bits_to_go)                                // Any bits in the current buffer ?
    {
      bit_buffer = mget(*source) ;                // If not read a full 8 bits of data.
      bits_to_go = 8 ;
    }
  t = bit_buffer & 1 ;                            // Return the next bit from the bottom of
  bit_buffer >>= 1 ;                              // the byte.
  bits_to_go -= 1 ;
  return t ;
}
```

```
//
// Start the decoding process.
//
void start_decoding (unsigned char huge **source)
{
  int i ;

  value = 0 ;                                   // Input bits to fill the code value.
  for (i = 1 ; i <= Code_value_bits ; i++)
    value = (value << 1) + input_bit (source) ;
  low = 0 ;                                     // Full code range.
  high = Top_value ;
}

//
// Decode the next symbol from the source data.
//
int decode_symbol (int cum_freq[], unsigned char huge **source)
{
  DWORD range ;                                 // Size of current code region.
  int cum ;                                     // Cumulative frequency calculated.
  int symbol ;                                  // Symbol decoded.

  range = (DWORD)(high-low) + 1 ;
                                                // Find cumulative frequency for value.
  cum = (((DWORD)(value - low) + 1) * cum_freq [0] - 1) / range ;
  for (symbol = 1 ; cum_freq [symbol] > cum ; symbol++)
    ;                                           // Then find symbol.
  symbol-- ;
                                                // Narrow down the code region.
  high = low + (range * cum_freq [symbol]) / cum_freq [0] - 1 ;
  low = low + (range * cum_freq [symbol + 1]) / cum_freq [0] ;
  for (;;)
    {                                           // Loop to get rid of bits.
      if (high >= Half)                         // Expand low half.
        if (low >= Half)
          {                                     // Expand high half.
            value -= Half ;
            low -= Half ;                       // Subtract offset to top.
            high -= Half ;
          }
        else
          if (low >= First_qtr && high < Third_qtr)
            {                                   // Expand middle half.
              value -= First_qtr ;
              low -= First_qtr ;                // Subtract offset to middle.
              high -= First_qtr ;
            }
          else
            break ;                             // Otherwise exit loop.
      low <<= 1 ;
      high = (high << 1) + 1 ;                  // Scale up code range.
      value = (value << 1) + input_bit (source) ;         // Move in next input bit.
    }
  return symbol ;
}

//
// Perform arithmetic decoding of the source data and store the decoded results in the
// destination buffer.
//
DWORD StatisticalDecode (unsigned char huge *source, unsigned char huge *destination, DWORD length)
{
  int symbol ;
  unsigned char huge *d ;

  d = destination ;
  start_fixed_model () ;                        // Initialize the arithmetic coder.
  start_inputing_bits () ;
  start_decoding (&source) ;
  for (;;)                                      // Loop through characters.
    {
      symbol = decode_symbol (cum_freq, &source) ;// Decode next symbol.
      if (symbol == EOF_symbol)
        break ;                                 // Exit loop if EOF symbol.
      mput(d, symbol) ;                         // Store it in the destination buffer.
    }
  return (DWORD)(d - destination) ;             // Return the length of the decoded data.
}
```

Arithmetic Coder - Adaptive

```
//
//
//                            Adaptive Arithmetic Coder
//
//                 Department of Electrical and Electronic Engineering
//                         Victoria University of Technology
//                                  (Footscray Campus)
//                                    P.O. Box 14428,
//                                 Melbourne Mail Centre,
//                                    Melbourne, 3000.
//
//      Date         : July 7, 1994
//
//      Author       : Emil Lenc
//
//      File Name    : aritadap.cpp
//
//      Supervisors  : Alec Simcock & Ann Pleasants
//
//      Internet     : emil@cabsav.vut.edu.au
//
//

// Size of Arithmetic Code Values.

#define Code_value_bits 16                      // Number of bits in a code value
typedef  DWORD code_value ;                     // Type of an arithmetic code value

#define Top_value (((DWORD)1<<Code_value_bits)-1)       // Largest code value

// Half and quarter points in the code value range.

#define First_qtr (Top_value/4+1)               // Point after first quarter
#define Half      (2*First_qtr)                 // Point after first half
#define Third_qtr (3*First_qtr)                 // Point after third quarter

// The set of symbols that may be encoded.

#define No_of_chars 256                         // Number of character symbols
#define EOF_symbol No_of_chars                  // Index of EOF symbol
#define No_of_symbols (No_of_chars+1)           // Total number of symbols
#define Max_frequency 16383                     // Maximum allowed frequency count 2^14-1

// Translation tables between character and symbol indexes.

int char_to_index [No_of_chars] ;               // To index from character
unsigned char index_to_char [No_of_symbols] ; // To character from index

// Cumulative frequency table

int cum_freq [No_of_symbols + 1] ;              // Cumulative symbol frequencies

// Bit input routines

int bit_buffer ;                                // Bits waiting to be input
int bits_to_go ;                                // Number of bits still in buffer
int garbage_bits ;                              // Number of bits past end-of-file

code_value low, high ;                          // Ends of the current code region
code_value value ;                              // Currently-seen code value
int bits_to_follow ;                            // Number of opposite bits to output after
                                                // the next bit.
int freq [No_of_symbols] ;                      // Symbol frequencies


//
// Initialize the arithmetic encoder output stream.
//
void start_outputing_bits ()
{
  bit_buffer = 0 ;                              // Buffer is empty to start with
  bits_to_go = 8 ;
}
```

```
//
// Output a bit to the destination buffer.
//
void output_bit (DWORD bit, unsigned char huge **destination)
{
  bit_buffer >>= 1 ;                          // Shift the current bits down one bit.
  if (bit)
    bit_buffer |= 0x80 ;                      // If new bit is a '1' add it to buffer.
  if (!(--bits_to_go))                        // Is the byte buffer full ?
    {                                         // Output buffer to destination if it is.
      mput(*destination, bit_buffer) ;
      bits_to_go = 8 ;
    }
}

//
// Flush out any remaining bits upon completion of the arithmetic coding process.
//
void done_outputing_bits (unsigned char huge **destination)
{
  mput(*destination, bit_buffer >> bits_to_go) ;
}

//
// Output bits plus following opposite bits.
//
void bit_plus_follow (DWORD bit, unsigned char huge **destination)
{
  output_bit (bit, destination) ;             // Output the bit.
  while (bits_to_follow)
    {
      output_bit (!bit, destination) ;        // Output bits to follow opposite bits.
      bits_to_follow-- ;                      // Set bits_to_follow to zero.
    }
}

//
// Initialize the encoder.
//
void start_encoding ()
  {
    low = 0 ;                                 // Full code range.
    high = Top_value ;
    bits_to_follow = 0 ;                      // No bits to follow next.
  }

//
// Encode the given symbol into the destination buffer.
//
void encode_symbol (int symbol, int cum_freq[], unsigned char huge **destination)
{
  DWORD range ;                               // Size of the current code region.

  range = (DWORD)(high - low) + 1 ;
                                              // Narrow the region for the current symbol.
  high = low + (range * cum_freq [symbol]) / cum_freq [0] - 1 ;
  low = low + (range * cum_freq [symbol + 1]) / cum_freq [0] ;
  for (;;)
    {                                         // Loop to output bits.
      if (high < Half)
        bit_plus_follow (0, destination) ;    // Output 0 if in low half.
      else
        if (low >= Half)
          {                                   // Output 1 if in high half.
            bit_plus_follow (1, destination) ;
            low -= Half ;
            high -= Half ;                    // Subtract offset to top.
          }
      else
        if (low >= First_qtr && high < Third_qtr)
          {                                   // Output opp. bit later if in middle half.
            bits_to_follow++ ;
            low -= First_qtr ;                // Subtract offset to middle.
            high -= First_qtr ;
          }
        else
          break ;                             // Otherwise exit loop.
      low <<= 1 ;
      high = (high << 1) + 1 ;                // Scale up code range.
    }
}
```

```
//
// Initialize the adaptive source model.
//
void start_adaptive_model()
{
  int i ;

  for (i = 0 ; i < No_of_chars ; i++)          // Set up tables that translate between the
    {                                          // symbol indexes and the characters.
      char_to_index [i] = i ;
      index_to_char [i] = i ;
    }
  for (i = 0 ; i < No_of_symbols ; i++)        // Reset the frequency counts.
    {
      freq [i] = 1 ;
      cum_freq [i] = No_of_symbols - i ;
    }
 cum_freq [i] = 0 ;
}

//
// Update the adaptive model for the new symbol data.
//
void update_adaptive_model (int symbol)
{
   int i ;                                     // New index for symbol.
   int cum ;
   int ch_i, ch_symbol ;

                                               // Scale the frequency counts if necessary.
   if (cum_freq [0] == Max_frequency)
     for (cum = 0, i = No_of_symbols - 1 ; i >= 0 ; i--)
       cum_freq [i] = cum += (freq [i] = (freq [i] + 1) >> 1) ;
   for (i = symbol ; freq [i] == freq [i - 1] && i ; i--)
     ;                                         // Find symbol's new index.
   if (i != symbol)                            // Swap the indexes if they have changed.
     {
       ch_i = index_to_char [i] ;
       ch_symbol = index_to_char [symbol] ;
       index_to_char [i] = ch_symbol ;
       index_to_char [symbol] = ch_i ;
       char_to_index [ch_i] = symbol ;
       char_to_index [ch_symbol] = i ;
     }
  freq [i]++ ;                                 // Increment the frequency count for the
  while (i > -1)                               // symbol and update the cumulative
    cum_freq [i--]++ ;                         // frequencies.
}

//
// Complete the encoding process.
//
void done_encoding (unsigned char huge **destination)
{
  bits_to_follow++ ;                           // Output two bits that select the quarter
  if (low < First_qtr)                         // that the current code range contains.
    bit_plus_follow (0, destination) ;
  else
    bit_plus_follow (1, destination) ;
}

//
// Perform arithmetic coding of the source data of length 'length' and store the coded
// results in the destination buffer.
//
DWORD StatisticalCode (unsigned char huge *source, unsigned char huge *destination, DWORD length)
{
  int ch, symbol ;
  unsigned char huge *d ;

  d = destination ;                            // Make a copy of the destination pointer.
  start_adaptive_model () ;                    // Initialize the arithmetic coder.
  start_outputing_bits () ;
  start_encoding () ;
  for (;;)                                     // Loop through the input buffer.
    {
      if (!(length--))                         // Has all the data been coded ?
        break ;                                // If so then exit.
      ch = mget(source) ;                      // Get the next available byte of data.
      symbol = char_to_index [ch] ;            // Translate to an index.
      encode_symbol (symbol, cum_freq, &d) ;   // Encode that symbol.
      update_adaptive_model (symbol) ;         // Update the model.
    }
  encode_symbol (EOF_symbol, cum_freq, &d) ;   // Encode the EOF symbol.
  done_encoding (&d) ;                         // Send the last few bits.
  done_outputing_bits (&d) ;
  return (DWORD)(d - destination) ;            // Return the length of the coded data.
}
```

```
//
// Initialize the arithmetic decoder.
//
void start_inputing_bits ()
{
  bits_to_go = 0 ;                              // Buffer starts out with no bits in it.
  garbage_bits = 0 ;
}

//
// Get a bit from the source buffer.
//
int input_bit (unsigned char huge **source)
{
  int t ;

  if (!bits_to_go)                              // Any bits in the current buffer ?
    {
      bit_buffer = mget(*source) ;              // If not read a full 8 bits of data.
      bits_to_go = 8 ;
    }
  t = bit_buffer & 1 ;                          // Return the next bit from the bottom of
  bit_buffer >>= 1 ;                            // the byte.
  bits_to_go -= 1 ;
  return t ;
}

//
// Start the decoding process.
//
void start_decoding (unsigned char huge **source)
{
  int i ;

  value = 0 ;                                   // Input bits to fill the code value.
  for (i = 1 ; i <= Code_value_bits ; i++)
    value = (value << 1) + input_bit (source) ;
  low = 0 ;                                     // Full code range.
  high = Top_value ;
}

//
// Decode the next symbol from the source data.
//
int decode_symbol (int cum_freq[], unsigned char huge **source)
{
  DWORD range ;                                 // Size of current code region.
  int cum ;                                     // Cumulative frequency calculated.
  int symbol ;                                  // Symbol decoded.

  range = (DWORD)(high-low) + 1 ;
                                                // Find cumulative frequency for value.
  cum = (((DWORD)(value - low) + 1) * cum_freq [0] - 1) / range ;
  for (symbol = 1 ; cum_freq [symbol] > cum ; symbol++)
    ;                                           // Then find symbol.
  symbol-- ;
                                                // Narrow down the code region.
  high = low + (range * cum_freq [symbol]) / cum_freq [0] - 1 ;
  low = low + (range * cum_freq [symbol + 1]) / cum_freq [0] ;
  for (;;)
    {                                           // Loop to get rid of bits.
      if (high >= Half)                         // Expand low half.
        if (low >= Half)
          {                                     // Expand high half.
            value -= Half ;
            low -= Half ;                       // Subtract offset to top.
            high -= Half ;
          }
        else
          if (low >= First_qtr && high < Third_qtr)
            {                                   // Expand middle half.
              value -= First_qtr ;
              low -= First_qtr ;                // Subtract offset to middle.
              high -= First_qtr ;
            }
          else
            break ;                             // Otherwise exit loop.
      low <<= 1 ;
      high = (high << 1) + 1 ;                  // Scale up code range.
      value = (value << 1) + input_bit (source) ;          // Move in next input bit.
    }
  return symbol ;
}
```

```
//
// Perform arithmetic decoding of the source data and store the decoded results in the
// destination buffer.
//
DWORD StatisticalDecode (unsigned char huge *source, unsigned char huge *destination, DWORD length)
{
  int ch, symbol ;
  unsigned char huge *d ;

  d = destination ;
  start_adaptive_model () ;
  start_inputing_bits () ;
  start_decoding (&source) ;
  for (;;)                                    // Loop through characters.
    {
      symbol = decode_symbol (cum_freq, &source) ;// Decode next symbol.
      if (symbol == EOF_symbol)
        break ;                               // Exit loop if EOF symbol.
      ch = index_to_char [symbol] ;           // Translate to a character.
      mput(d, ch) ;                           // Store it in the destination buffer.
      update_adaptive_model (symbol) ;        // Update the model.
    }
  return (DWORD)(d - destination) ;           // Return the length of the decoded data.
}
```