TR 89-14 V

An Adaptive Discrete Cosine Transform Coding Scheme for Digital X-Ray Images

Thesis
submitted in partial fulfilment of the
requirements of the degree of

MASTER OF SCIENCE

of Rhodes University

by

Ivan Hugh Mclean

May 1988

ABSTRACT

The ongoing development of storage devices and technologies for medical image management has led to a growth in the digital archiving of these images. The characteristics of medical x-rays are examined, and a number of digital coding methods are considered. An investigation of several fast cosine transform algorithms is carried out. An adaptive cosine transform coding technique is implemented which produces good quality images using bit rates lower than 0.38 bits per picture element.

Acknowlegements

Firstly, my thanks must go to George Wells, Richard Foss and Justin Jonas for all their invaluable help.
Special thanks to my supervisor prof. Gerhard de Jager for all the help and encouragement during the year.
Finally, thanks to my family and to Lauren for all the love and support.

Contents

page

1.INTRODUCTION 1

2. SYSTEM DESCRIPTION 4

2.1 Hardware 4
2.2 Software 5
2.2.1 RMF 5
2.2.2 Starlink 5

3. AN INVESTIGATION OF THE ATTRIBUTES OF DIGITAL X-RAYS 7

3.1 Test image dimensions 7
3.2 Notes on reducing irrelevant anatomy 7
3.3 Image statistics 10
3.4 Fidelity criteria 13

4. AN OVERVIEW OF IMAGE CODING TECHNIQUES 14

4.1 Reversable information preserving coding methods 14
4.1.1 Huffman and Shannon codes 14
4.1.2 Bit plane encoding 15
4.2 Psychovisual coders 15
4.2.1 The Synthetic high system 15
4.3 Predictive coding 16
4.3.1 DPCM 16
4.4 Cluster coding methods 16
4.4.1 The Blob algorithm 17
4.5 Hybrid coding methods 17

5. TRANSFORM CODING 18

6. THE DCT 21

6.1 Coefficient bound 22

7. AN INVESTIGATION OF FDCT ALGORITHMS 23

7.1 Indirect computation 23
7.2 Recursive computation 24
7.3 Direct factorization 24

8. DCT CODING TECHNIQUES 27

8.1 Zonal coding 27
8.2 Threshold coding 31
8.3 An adaptive DCT coding scheme 40

9. ASPECTS FOR FURTHER STUDY 45

9.1 Reduction of blocking effects 45
9.1.1 Overlap method 45
9.1.2 Filtering method 45
9.1.3 Lapped orthogonal transforms 45
9.2 Bit allocation 46
9.3 Nonuniform quantization 46
9.4 Error protection 46

10. DISCUSSION AND CONCLUSION 48

BIBLIOGRAPHY 49

PROGRAM LISTINGS

1. Introduction

The growth of digital and electronic imaging systems has revolutionized the requirements and technology for aquiring, recording, processing, storing, transmitting, and retrieving images.

An assortment of new storage devices and technologies has recently become available, allowing digital (or electronic) archiving of medical images. The most talked about device is probably the digital optical recorder, which allows the recording of up to 1000 m bytes of data on each side of an optical disk. But more readily available and less expensive systems are on hand- namely the new digital videodisks and the new high-capacity fast access magnetic tapes and disks [19].

Methods employed in the management of centralized analogue film library archiving systems will have to be changed or adapted to the management of digital image systems. At present, a film file room manages both the analog generated radiographic images and the video raster film recordings of digital diagnostic images. The transmission and retrieval of film images is very straight forward: someone physically takes the film out of the file room and carries it to where it is needed [6].

Duernickx [6] states that in 10 years, an medium-size hospital produces about five million pictures, most of them high resolution chest x-rays. To make matters worse, plates are retained for between 10 to 30 years after the initial investigation, depending on the legislation of the country in question [15].

The radiological information is conventionally recorded on film, because in the past only film was able to record the fine detail, wide dynamic range and large area of the image in sufficiently short time. However, film is increasingly expensive to use, bulky to store and inconvenient to distribute. In addition, the information contained therein is difficult to manipulate.

On the other hand, the digital representation of an x-ray image requires a large number of bits. Although there is some disagreement over this issue, Lux [15][1] states that a typical 40 x 40 cm x-ray image requires about 10^8 bits.

Among the many inherent shortcomings of digitized film images are film dynamic range, scatter, and digitizing instrumentation imperfections [26]. A digital image aquisition system obviates many of these. The past few years have seen the development of x-ray detection systems capable of capturing these images directly in photoelectronic form. In order to overcome the size limitations of conventional x-ray image intensifiers, as well as to largely eliminate the image degredation caused by scattered radiation, at least two manufacturers of diagnostic systems have constructed dedicated digital chest machines based on scanning x-ray beam geometry.

One of the many advantages of storing the x-ray image in digital form is that it is then possible to process the image data using the full range of image processing techniques widely used in a variety of other application areas. These techniques include image enhancement, restoration, quantative analysis and data compression. The success of these techniques in a wide variety of applications points to the possibility of similar benefits being reaped from their application to digital x-ray images.

Digitization would also make possible centralized storage facilities for large urban areas, allowing for fast and efficient retrieval of any plate at the touch of a key. Remote diagnosis would also be incorporated into such a system.

Duernickx [6] is of the opinion that the major challenge now facing the medical imaging community does not involve the process of imaging per se but rather the management of images once they are aquired. The need is evident for minimizing redundancy in images in order to reduce storage costs and improve transmission speed. Digital coding methods are well known in applications such as TV, video telephones and facsimile machines. It is neccessary, however, to adapt the corresponding coding techniques to suit x-ray statistics. The criteria for

[1] The scanning rate and number of bits/pixel required for the digital representation of an x-ray picture are controversial issues, and will not be considered here.

judging reconstructed radiographs are strict in medical diagnostics because errors of reconstruction may influence diagnostic reliability [15].

It is the aim of this dissertation to find the best coding algorithm for this application, based on the parameters of complexity of implementation and performance, and to evaluate this method by means of a computer simulation.

This dissertation is divided into ten chapters.

Chapter Two describes the hardware and software used to digitize the x-ray images.

Chapter Three involves an investigation of the statistics of x-ray images.

Chapter Four presents a brief overview of widely used image coding techniques.

Chapter Five provides an introduction to transform image coding.

Chapter Six looks at the Discrete Cosine Transform as the optimum transform.

Chapter Seven presents a review of the many algorithms which have arisen over the last decade for implementing the Fast DCT.

Chapter Eight looks at three different methods of transform DCT image coding, and presents computer simulation results for each. An adaptive method is presented which achieves good quality images using bit rates lower than 0.38 bits per picture element. Included in this chapter are the results obtained using each of these methods, in the form of photographs, error maps and graphs.

Chapter Nine looks at aspects for further study, and is mainly concerned with the "fine tuning" of the algorithm presented.

Chapter Ten provides a discussion and conclusion on the results presented.

2. System description

2.1 Hardware

To capture an image, use was made of the PCEYE VIDEO CAPTURE SYSTEM (Chorus Data Systems), interfaced to a CW16 Excel microcomputer.

The PC-EYE package includes a TM-34K CCD video camera, together with an extensive set of machine language subroutines to allow easy capture, storage, manipulation and display of an image. Software interfaces are provided to enable these subroutines to be called from applications programs written in C or Basic.

The CW16 microcomputer was equipped with a 20M byte hard disk, 640 K memory, 8 mhz clock, and maths coprocessor. A Tecmar Graphics Master high resolution graphics display adapter capable of displaying pictures of 640 x 400 pixels with 4 bits/pixel (16 grey levels) was used. This was unable to display the 640 x 512 x 6 pixel image which the system was capable of storing on disk or memory. The IBM colour display served as a rough guide to the nature of the image to be captured on disk. From the display the size, lighting and orientation could be distinguished and subsequently corrected.

The video camera produces an interlaced video signal which forms 30 frames per second. This signal is digitized at a rate greater than that at which the micro can transfer data to memory [2]. Aquisition must therefore be delayed to enable the data to be written to memory. To overcome this problem, a monitor was connected to the External Video Output, thus providing a real time view finder.

Images were sent to a Digital Equipment Corporation Vax-11/730 (Vax) minicomputer via a standard RS232C serial communications line, and displayed on a Tektronix M4115B graphics terminal, maximum resolution 1280 x 1024 pixels with 8 bits/pixel.

2.2 Software

A C program (PC2.C) was created in order to access the PC-EYE functions. These functions provide control of the hardware and set of utilities to move image data through the system and to capture an image on disk. The user specifies the name of the output file and the drive it is to reside on. The horizontal and vertical dimensions of the image may be specified, as well as the black and white levels to be used. The values entered for these levels control reference voltages to the A/D converter which digitizes the video signal.

Images on the VAX are stored as Binary Data Files, which have a .BDF appended to the file name. Before the PC-EYE images could be displayed or processed, they had to be converted to this format. The main difference between the CW-16 PC/PC-EYE capture format and the VAX/STARLINK display format lies in their respective co-ordinate numbering conventions. PC-EYE numbers co-ordinates according to the standard video scanning sequence. The scan begins at the top left-hand corner of the image (scanning rows first), and therefore this is chosen as the origin, with co-ordinate values increasing downwards and to the right. The image display terminal used on the VAX numbers the co-ordinates according to the usual mathematical format with the origin at the bottom left-hand corner and values increasing upwards and to the right. The program PCEYE.FOR was modified to perform this conversion (see Appendix B).

2.2.1 RMF

A locally written version of CDC's RMF system was used to send images to the VAX. This version enables a microcomputer to function as a terminal for the VAX. This made it possible for image files to be transferred from the IBM PC to the VAX and vice versa [2].

2.2.2 Starlink

Extensive use was made of the Starlink Software Collection which resides on the VAX. Although written for astronomical applications, some of the display and manipulation subroutines proved valuable.

Because the 256 x 256 picture element test images were too small for display purposes (approximately 4cm x 4cm on the Tek screen), use was made of the Starlink program, **Expand.** This program takes the original image and expands it by interpolating new data values between the old ones. Sine(theta)/theta interpolation is used. The impact on the transform of the image is not important since the resulting image is used for display purposes only.

3. An investigation of the attributes of digital x-rays

3.1 Test image dimensions

A digital image is a 2-dimensional array of picture elements or "pixels". Each pixel has a binary value corresponding to the light intensity at that point in the image. This value is called the "grey-level" of the pixel. The number of discrete grey-levels from black to white is fixed and determined by the number of bits that make up the binary values. In our case, each picture is a 256 x 256 matrix, each element of which is an 8 bit number. Therefore the total number of bits required to represent the picture is $2^8 \times 2^8 \times 2^3 = 2^{19}$. The reason for the use of such a small image is the time factor involved in processing such an image. The coding algorithms were simulated in finite precision wordlengths on a CW16 micro. Even using the FDCT algorithm proposed by Chen *et al* [4], the time taken to perform the transform, compression and inverse transform is approximately 35 minutes. If a 400 x 400 pixel input image is used, this time may run into several hours.

3.2 Notes on reducing irrelevant anatomy

X-ray images contain a large amount of redundant information. Although this is partly due to the inherent anatomical redundancy and inherent strong correlation between adjacent pixels (q.v. section 3.3), another factor is the irrelevant anatomy that is recorded [26].

Fig 1.1(a) shows the original x-ray, showing a patient with a broken neck. In this case the doctor is not concerned with features such as the shoulders or head of the patient, although it is wise to include these features to a certain extent, in order to allow for some perspective and orientation. Fig 1.1(b) shows the image that would be stored in the image filing system. This 256 x 256 pixel image represents a 1:4 savings in storage cost, while retaining all the relevant information of the original radiograph. Fig 1.2(a) shows the original x-ray of a patient with a fractured leg. Fig 1.2(b) is the final image, ready for processing. Once again redundant information is discarded.

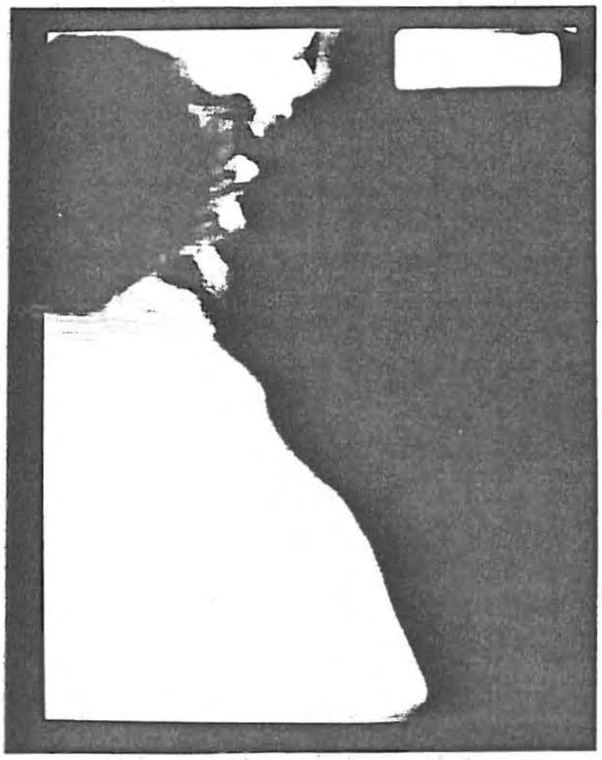


Fig 1.1(a) Original x-ray, showing irrelevant anatomy.

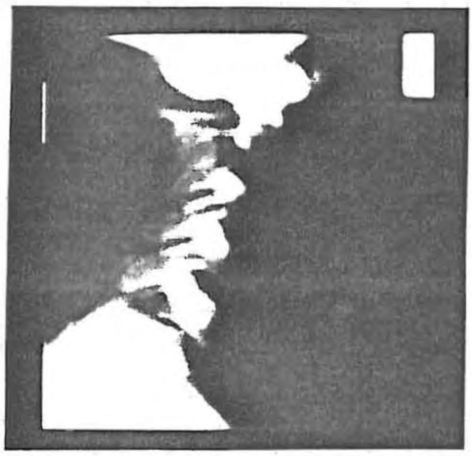


Fig 1.1(b) Stored image.

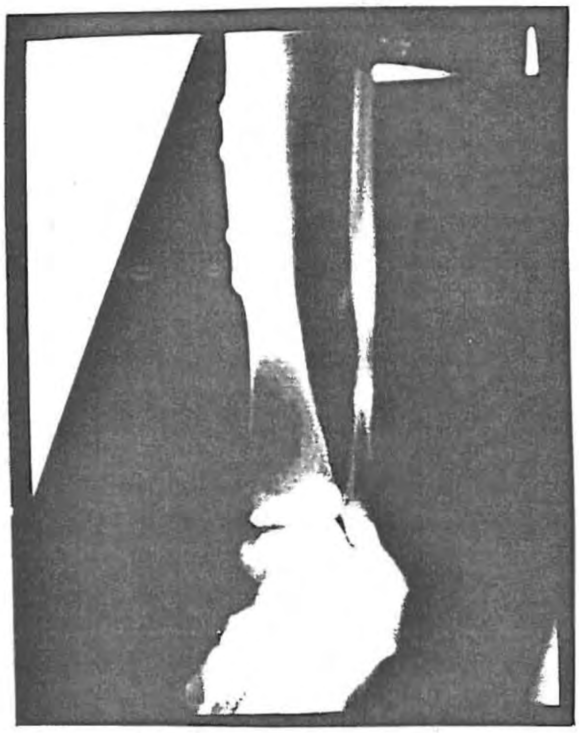


Fig 1.2(a). Original x-ray

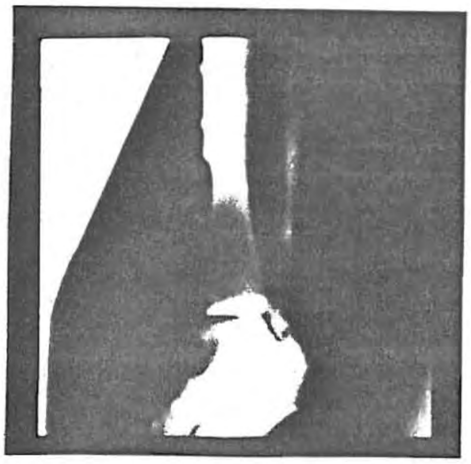


Fig 1.2(b) Stored image.

It can be appreciated that the image capturing stage is an extremely important one. It is of little use to attempt to maximise the compression rate while retaining large areas of redundant information.

3.3 Image statistics

Since the compression rate depends mainly on picture statistics, it is important to understand some of the characteristics of x-rays. Reininger and Gibson [22] show that for a "large class" of images the average intensity is best approximated by a Gaussian distribution and that the non-dc coefficients are best approximated by the Laplacian distribution. Using Laplacian quantizers for the non-dc transform coefficients can improve the quality of the reconstructed image as compared to Gaussian quantizers. Care must be taken to correctly classify the input image, since some images, like some aerial images are still best represented by Gaussian statistics.

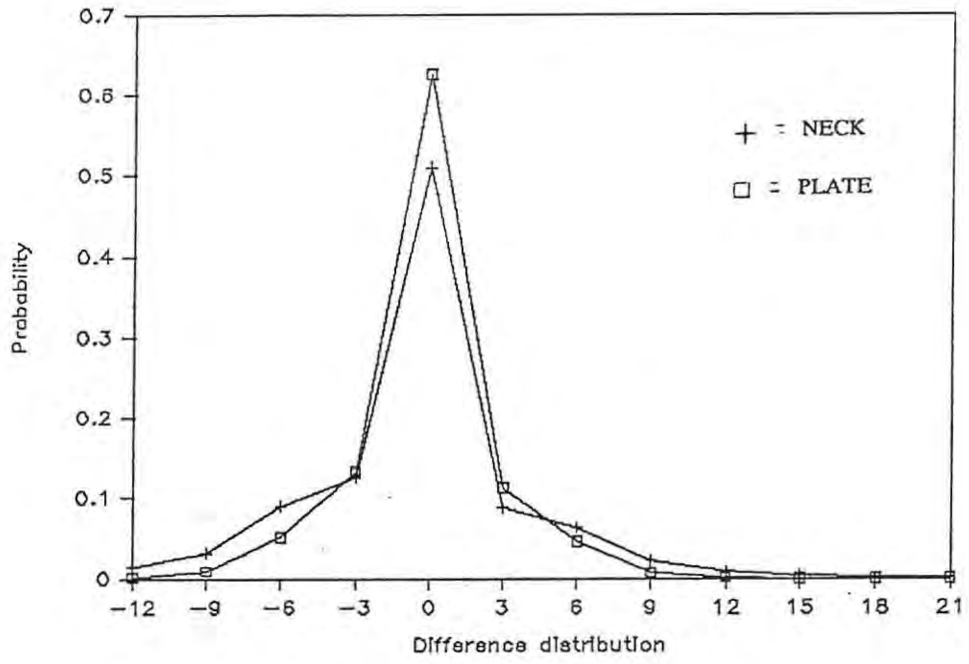


Fig 2.1 Difference distribution for PLATE and NECK.

The distribution of differences between adjacent pixels was investigated. It was found that this distribution approaches a Laplacian distribution rather than a Gaussian for all x-rays tested (Fig 2.1). This is in agreement with the findings of Tesic *et al* [26].

The autocorrelation function (acf) tells us how close, or similar neighbouring pixels are, on the average. Figs 2.2 and 2.3 show the autocorrelation functions for **PLATE** and **NECK** respectively.

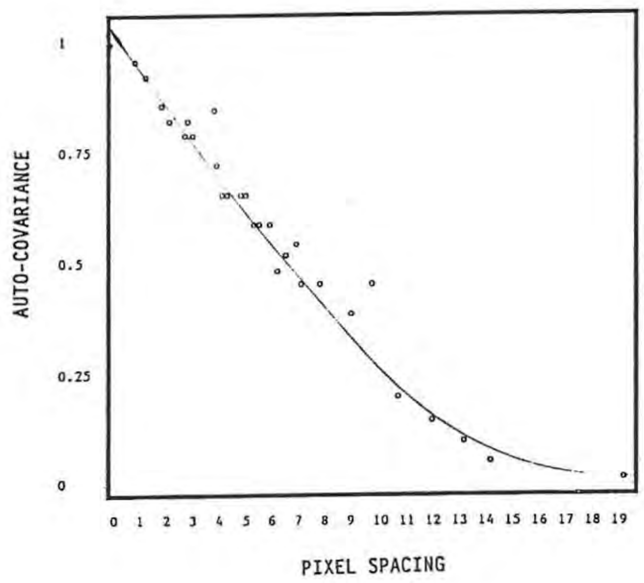


Fig 2.2 Autocorrelation function for PLATE.

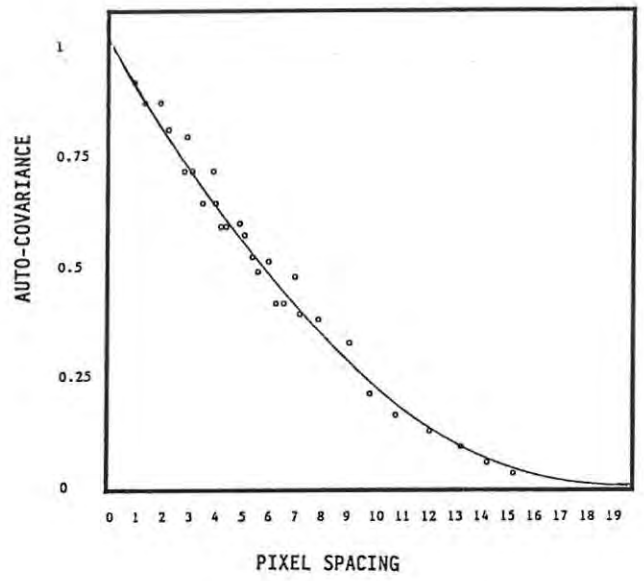


Fig 2.3 Autocorrelation function for NECK.

The statistics presented in Figs 2.1 - 3 show much correlation inherent in this type of image. In a "busy" picture, the autocorrelation function should fall off very rapidly, while in a "smooth" picture the falloff should be more gradual. The quantity ρ has values in the range (-1, +1). A value of $\rho = 1$ implies perfect positive correlation (any two samples of a constant waveform), while a value of $\rho = -1$ suggests perfect negative correlation (a sinusoid sampled only at its positive and negative peaks). A value of $\rho = 0$ implies total decorrelation, as between pixels in a totaly random white noise image [12].

The approximate values of p for PLATE and NECK are 0.94 and 0.93 respectively. This proves that for both images, pixel values in a small area are strongly correlated. Unser [27] shows that the DCT provides a very close approximation of the Karhunen-Loeve transform for positive correlation values close to unity (q.v. Chapter 5). More specifically, it is known that for a Markov signal with correlation coefficient $1 >= \rho >= 0.5$, the DCT is the best substitute for the optimal Karhunen-Loeve transform [26].

3.4 Fidelity Criteria

A number of fidelity criteria have been created to measure the quality of a reconstructed image [7,21,23]. In this dissertation the NMSE or normalized mean square error will be used as an objective measure of the coding efficiency of the DCT coding algorithms presented.

The NMSE is given by Pratt [21] as

$$\mathrm{NMSE} = \frac{\frac{1}{JK}\sum_{j=0}^{J-1}\sum_{k=0}^{K-1}\left[f(j,k) - \hat{f}(j,k) \right]^2}{\frac{1}{JK}\sum_{j=0}^{J-1}\sum_{k=0}^{K-1}\left[f(j,k) \right]^2}$$

This is a measure of the normalized mean square error between the spatial picture before coding and the reconstructed picture after coding. Normalization is achieved by dividing the mean square error by the mean signal energy within the picture (see Appendix Q).

The second, and in this case more important criterion, is the subjective fidelity criterion, corresponding to how "good" the images look to human observers, or in this case, the trained eye of the radiologist or physician. It is on this criterion that the ultimate success or failure of the coding algorithm will depend, since the human visual system has particular characteristics so that two pictures having the same percentage of NMSE may appear to have drastically different visual qualities. For example, the human visual system has a logarithmic sensitivity to light intensity so that errors in dark areas of an image are much more noticeable than errors of the same magnitude in light areas. The human visual system is also sensitive to abrupt spatial changes in grey level so that reconstruction errors on or near the edges are more bothersome than reconstruction errors in background texture [7].

The inadequacy of a simple measurement like the NMSE as a perceptually meaningful measure of image quality can be shown by the simple example of a small geometric tilt of an image. Although the tilted image would be perceptually quite acceptable for an application such as this, the resulting value for the NMSE would be relatively large.

4. An overview of image coding techniques.

We have seen that the digital representation of an x-ray image requires a very large number of bits. The aim of digital image coding is to code the image using a smaller number of code bits, while satisfying the often delicate balance between reconstruction quality and compression rate.

Coding methods may be placed into a number of classes. Initially, a distinction may be made between information lossless techniques, which allow reconstruction of the original image exactly, and information lossy techniques which introduce some distortion. Furthermore, the distinction can be made between coding for storage of images or coding for bandwidth reduction of image transmission. A third classification may be made, based on the space where the method is applied. A technique which manipulates the values of the pixels in an appropriate manner is labelled as a spatial method. Methods which operate on the set of transform coefficients are known as transform methods.

Kunt [13] states that early efforts in image coding were guided solely by information theory, and as a result the upper bound of the compression rate was soon reached. Shreiber [24] was one of the first to exploit the properties of the human visual system, leading the way for new coding schemes capable of achieving impressive compression rates. These techniques which model the human visual system are known as psychovisual coders or psychophysical coders. There are, of course, many other classifications which can be made. Some coding schemes will belong to a number of different classes while others may not seem to fit into any category.

What follows is a brief discription of some well known coding techniques.

4.1 Reversable information preserving coding methods

4.1.1 Huffman and Shannon codes

Early methods used Huffman and Shannon codes [7,20] to reduce the average code length. Using these methods, different length PCM (Pulse Code Modulation) code words are assigned to different "events".

The intuitive idea is to assign short code words to the most probable events (no change in successive brightness levels) and longer code words to the least likely events (large differences in successive brightness levels) to obtain a code with a small average length.

The variable length of the code words make this method more difficult to implement than a fixed length code. Another problem with these methods is that a change of input image would require a new code mapping to ensure minimal length.

The properties of the human visual system are not exploited at all, and the resulting compression ratio is approximately 2.5 : 1.

4.1.2 Bit plane encoding

Bit plane encoding [7,13,14,20] is a constant-word-length PCM encoding technique, in which the code words are conceptually organized into planes corresponding to their pixel position. The most significant planes are organized so as to occupy the bottom plane. It has been found [14] that the most significant bits, occupying the bottom plane, seldom change in most images, while the least significant bits occupying the top plane fluctuate almost randomly. Compression is achieved by run length coding the bit state transitions in each bit plane. Compression ratio is roughly 2 : 1.

4.2 Psychovisual coders

4.2.1 The Synthetic high system

In the early days of image coding, the computational load of the synthetic high algorithm was thought to be too heavy, but present technology makes this method easy to implement. The original image is broken up into two parts : the low pass picture giving the general area brightness without sharp contours, and the high pass picture containing sharp edge information [13]. The low frequency picture is then quantized and coded at a low bit rate, while the high-frequency picture is coded at a higher rate [13,14,20].

The synthetic high system was far in advance for its time, and its elegant exploitation of the properties of the human visual system, allowed it to achieve an impressive compression ratio of approximately 8.5 : 1

4.3 Predictive coding

4.3.1 DPCM

DPCM (Differential Pulse Code Modulation) [7,23,26] makes use of the fact that for most images, adjacent pixels are highly correlated. Thus if pixel x_{i-1} has a certain value, then the adjacent pixel x_i along the same scan line is likely to have a similar value.

The pixel to be sampled is predicted from the previous element. The transmitter predictor then computes the error between the actual pixel value and the predicted value. This difference will, on the average, be significantly smaller in magnitude than the value of the pixel.

Although the simplicity of DPCM is an advantage, this technique only allows compression ratios of around 2.5 : 1. DPCM also suffers from several problems, including granular noise and error propagation at the receiver.

The development of a predictive coding scheme for digital chest images is described by Tesic *et al* [26].

4.4 Cluster coding methods

Cluster coding methods differ from other techniques such as transform and predictive coding techniques where the correlated data are transformed to generate uncorrelated elements prior to quantization and transmission. In cluster coding, the data are grouped to form a number of clusters.

4.4.1 The Blob algorithm

The blob algorithm is an adaptive method which examines each four adjacent pixels and uses a hypothesis testing method to decide if the pixels are close enough to be joined with the adjacent pixels or if they belong to a new class. In this manner, the entire image is partitioned into blobs such that all pixels within a blob have similar grey levels and textures.

As a result of this similarity, only the boundary information and one representative value for each blob is needed by the receiver in order to reconstruct that blob. The boundary information for each blob may be extracted using contour tracing algorithms.

4.5 Hybrid coding methods.

Hybrid coding methods combine the strong points of predictive and transform coding to achieve good performance at bit rates lower than 1 bit/pixel. A common method is to take a one-dimensional unitary transform along the rows of an image and then encode these coefficients in a DPCM technique across the transformed rows. The advantages of such a method are speed of operation and memory reduction. Pyramidal coding and Anisotropic nonstationary predictive coding are two better known hybrid methods. The latter method makes extensive use of the properties of the human visual system to achieve a compression rate of greater than 10 : 1.

5. Transform coding

The subject of transform image coding has been exhaustively covered in the literature, a good summary having been presented by Wintz [30]. Transform coding uses a linear mathematical operator to create a new domain. Statistically dependant pixels are transformed into a set of "more independant" coefficients. The basic advantage of an image transform coding system is that the two-dimensional transform of an image has energy distribution more suitable for coding that the spatial domain presentation.

As a result of the inherent pixel-to-pixel correlation of an x-ray image, the energy of the transform domain tends to be clustered into a relatively small number of transform samples. To achieve compression, low magnitude transform samples can be grossly quantized, without introducing serious picture reconstruction quality.

Since the transformation is a reversable process without loss of information, it is the quantization and coder operation after transformation which actually provides the data compression. The coder plays the part of a sample selector which decides which of the quantized co-efficient samples are to be selected for transmission or storage, and which are discarded.

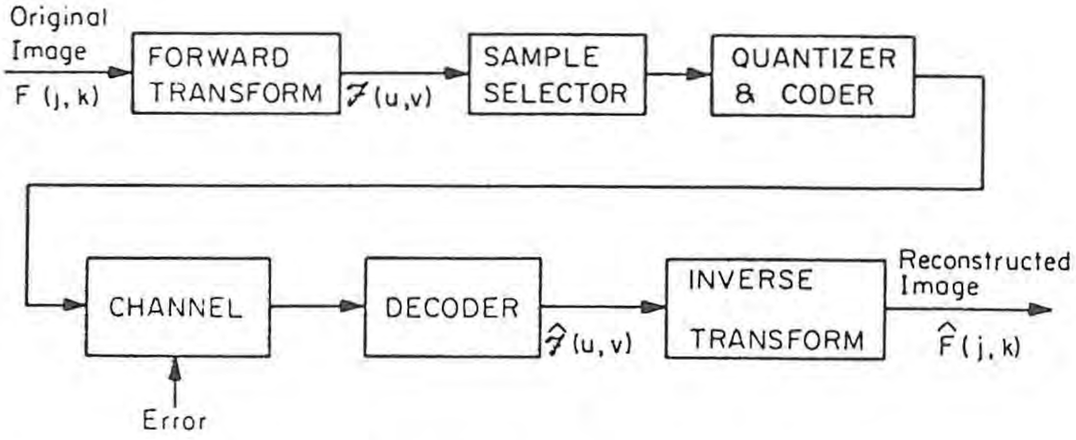


Fig 3.1 Transform image coding system. Figure reproduced from Pratt [21].

By representing an image by uncorrelated data we make each element of that data a unique property of the data. The data is then classified according to degree of significance of their contribution to both the information content and the subjective quality of the picture [23].

After the data is categorized, those elements of the data that are unimportant from the point of view of the grey scale and spatial resolution capability of the receiver can be neglected. This allows a major degree of picture compression.

In transform coding, the input image is partitioned into a number of n x n arrays with n << N as illustrated in Fig 3.2.

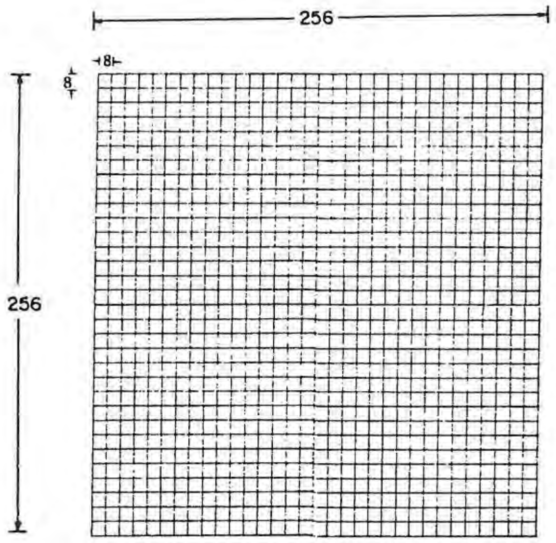
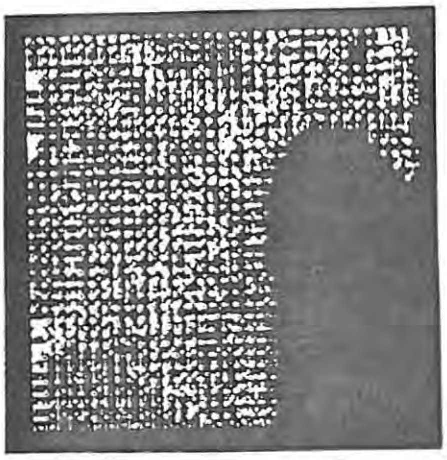


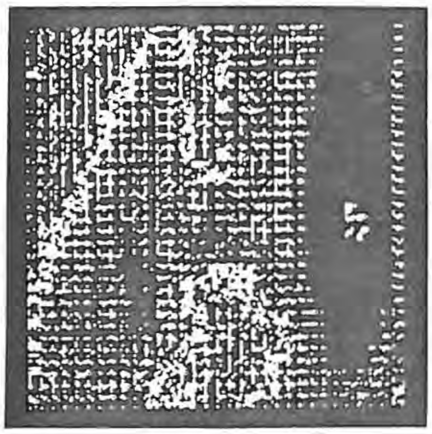
Fig 3.2 Partitioning a 256 x 256 picture into 1024 8x8 subpictures

Each sub-image is then coded independantly, and the separate sub-image blocks are reassembled by the decoder. By segmenting the image, the local characteristics of the image are exploited.

Fig 3.3 shows the cosine transforms for NECK and PLATE. Since the dynamic range of transform components is very large (q.v. section 6.1) it is necessary to compress the coefficient values to produce a useful display.



a



b

Fig 3.3 a) Log scaled DCT of NECK. b) Log scaled DCT of PLATE.

Transform coding techniques are more complex than other conventional compression techniques, but they achieve a higher compression ratio and less distortion. The optimum transform is the Karhunen-Loeve transformation. Unfortunately, no fast computational algorithm exists for the KL transform. It has been shown that for many practical situations the cosine transform, which has many fast algorithms, yields results almost identical to the K-L transformation.

To conclude this chapter, some of the advantages of transform coding are listed below [20].

1. Superior coding at low bit rates. (lower than 1 bit/pixel)
2. Produces uncorrelated coefficients from a highly correlated image field.
3. Fast convolutional algorithms for implementation.
4. Less sensitive to image statistics.
5. Better subjective error performance (than DPCM).

6. The DCT

The one-dimensional cosine transform of a discrete function f(j), j = 0,1,...N-1 is defined as [4]

$$F(u) = \frac{2c(u)}{N} \sum_{j=0}^{N-1} f(j) \cos\left[\frac{(2j+1)u\pi}{2N}\right]$$

$$c(u) = \frac{1}{\sqrt{2}}, \ldots\ldots \text{ for } u=0$$
$$= 1 \ldots\ldots \text{ for } u = 1,2,\ldots\ldots N-1.$$
$$= 0, \ldots\ldots\ldots\ldots \text{elsewhere.}$$

and the inverse transform is

$$f(j) = \sum_{u=0}^{N-1} c(u)F(u)\cos\left[\frac{(2j+1)u\pi}{2N}\right]$$

$$j = 0,1, \ldots N-1.$$

The attractiveness of the DCT for this application is two-fold : (a) it is nearly optimal for images with high positive values of adjacent-sample correlation. And (b) it can be computed via any one of a number of fast algorithms [4,5,9,10,16,17].

It is true that in the class of unitary transforms with a known fast computational algorithm the DCT has a superior energy compression property. The DCT also results in the same energy compaction performance as the optimum Karhunen-Loeve transform for most images.

6.1 Coefficient bound

Before going on to consider a fast algorithm for the DCT, it is vital to consider the coefficient bound. The maximum coefficient value of the cosine transform is very important for finite word length computer simulations of the various coding algorithms, since the dc-coefficient is linearly quantized. This is due to the fact that quantization errors in the dc-coefficient cause artificial sub-block boundary discontinuities at reconstruction and should be avoided.

Oosthuizen [20] shows that the maximum coeffient bound for the one-dimensional Chen-DCT matrix implementation is

$$\mathrm{Fk\ max} = \frac{2}{N\sqrt{2}} \sum_{j=0}^{N-1} x(j) \leq \sqrt{2}\,(\mathrm{f\ max})$$

$$\text{since } x(j) \geq 0 \text{ for all.}$$

By defining the maximum spatial pixel value as 255 (8 bit input image) the maximum coefficient bound is calculated as

$$\mathrm{Fk\ max} = \sqrt{2}\,(255) = 360{,}62$$

Accepting the DCT to be unitary the maximum coefficient bound for the 2-dimensional DCT is calculated as

$$\mathrm{Fk\ max/2D} = \sqrt{2}\,(\mathrm{Fk\ max}) = 2\,(\mathrm{f\ max}) = 510$$

Note that the maximum coefficient bound is not affected by the transform block size. The coefficient bound for the Chen algorithm is also substantially less than most other FDCT algorithms.

7. An investigation of FDCT algorithms

The DCT performs closely to the statistically-optimal Karhunen-Loeve transform for a wide class of signals. Fast algorithms for the DCT are therefore of significant practical interest. Initially, use of the DCT in a variety of applications was limited due to lack of an efficient algorithm. In recent years however, many fast algorithms for computing the DCT have been published. These algorithms can be classified into one of the following categories based on their methods of approach: a) indirect computation [5,16,17], b) recursive computation [10], or c) direct factorization [4,9].

Fast Fourier transforms (FFT's) and Walsh-Hadamard transforms are used to obtain the DCT in (a), but unnecessary operations are often involved in the computation steps. Using sparse matrix factorization, the algorithms in (b) gain speed advantages over the other methods. The recursive approach in (c) is intended to generate higher order DCT's from lower order DCT's.

7.1 Indirect computation

Early methods of implementing the DCT utilized a double size Fast Fourier Transform (FFT) algorithm employing complex arithmetic throughout the computation. Makhoul [16] demonstrates the computation of the DCT of a sequence from just an N-point DFT of a reordered version of the same sequence, with a resulting saving of 1/2.

Makhoul shows how the DCT of a 2-D real sequence $\{x(n_1, n_2), 0 <= n_1 <= N1 - 1, 0 <= n_2 <= N2-1 \}$ can be computed using an (N1 x N2) - point real DFT instead of the (2N1 x 2N2) - point real DFT required in the traditional method, resulting in a saving of 1/4. Makhoul claims performance similiar to the Chen algorithm for N a power of 2.

Malvar [17] presents an approach for the computation of the DCT in which a Discrete Hartley transform is performed on a permutation of the incoming data, and the result is passed through a plane rotation.

7.2 Recursive computation

Hou [10] introduces a recursive DCT algorithm which allows the generation of higher order DCTs from two identical lower order DCT's. This feature offers flexibility in programming different sizes of DCT's, as well as requiring fewer multipliers and adders than any other known DCT algorithm. This method proved to be one of the most promising encountered in this survey, providing excellent performance without introducing great complexity.

7.3 Direct factorization

For this application, this approach seemed the most promising. Two algorithms were considered - the method proposed by Haque [9] and the well known Chen *et al* [4] algorithm.

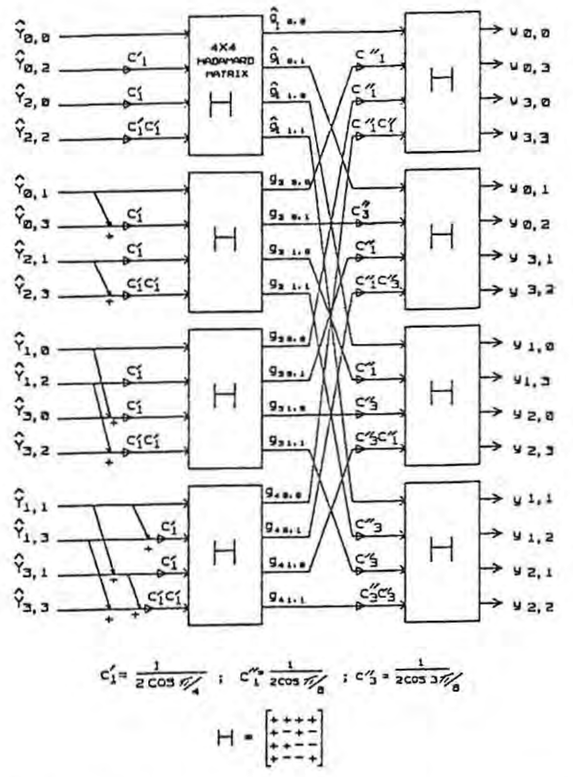

4X4 HADAMARD MATRIX
H
$c'_1 = \frac{1}{2\cos \pi/4}$; $c''_1 = \frac{1}{2\cos \pi/8}$; $c''_3 = \frac{1}{2\cos 3\pi/8}$
$H = \begin{bmatrix} + & + & + & + \\ + & - & + & - \\ + & + & - & - \\ + & - & - & + \end{bmatrix}$

Fig 4.1 A 4x4 2-D inverse fast cosine transform. Figure reproduced from Haque [9].

Because of the separability of the 2-D DCT kernel, a 1-D FCT can be used to compute the 2-D FCT, first taking the 2-D data along the columns and then along the rows. But this method requires increased computation. Haque [9] introduces a 2-D FCT algorithm for $2^m \times 2^n$ data points which requires only real multiplications and additions.

This algorithm works directly on 2-D data sets and not separately on columns and rows, as done by the 1-D FCT. For this algorithm, the number of real multiplications without initial normalizations, for M x N data samples, is $(3/8)MN\log_2(MN)$, where M and N are powers of 2.

The paper by Haque is very brief, and an extension of the algorithm for 8x8 or 16x16 input matrices is not shown. Another problem arises with this algorithm when larger data blocks are used. If a 128 x 128 block is used, the corresponding maximum coefficient is 1660, which could result in some overflow problems in a finite word length machine [9].

The FDCT method by Chen *et al* [4] appears simplest to interpret, and while not being the most efficient, represents one technique for methodical extension, to any desired value of $N = 2^m, m \geqslant 2$.

The signal flow graph consists of alternating cosine/sine butterfly matrices with binary matrices to reorder the matrix elements to a form which preserves a recognisable bit reversed pattern at every other node.

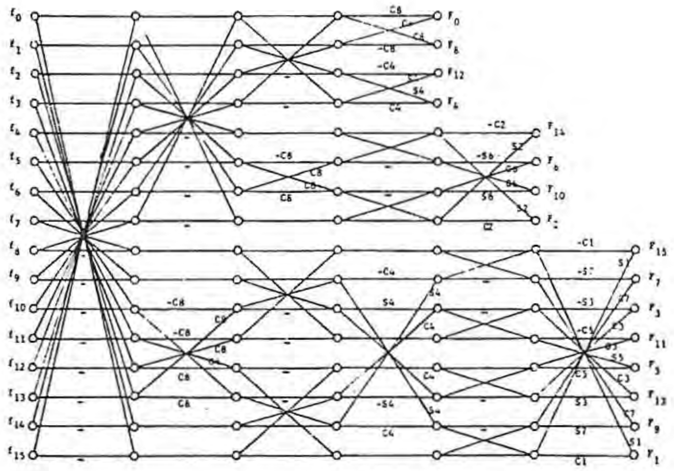


Fig 4.2 Chen FDCT flow graph for N = 4, N = 8, N = 16. Ci = cos *i*, Si = sin *i*. Figure reproduced from Chen *et al* [4].

Practical system application is attained by maintaining a balance between **complexity of implementation** and **performance.** The success of the Chen *et al* [4] FDCT algorithm can be seen by its widespread use in practical applications [20,26], where its simplicity and ease of implementation give it the edge on later, more efficient methods. Another advantage of the Chen *et al* method is that it gives the user a "feel" for the workings of the DCT. As a start, the algorithm was implemented on a spreadsheet for 8 data points, making it possible to see the intermediate stages as results "filtered through". Fig 4.2 is a signal flow graph for N = 4,8 and 16. Note that this graph is bidirectional, i.e. the inverse transform may be computed by introducing the vector [F] at the output and recovering the vector [f] at the input [4].

A drawback of the Chen algorithm is that it is not 2-dimensional. When operating on a block of image data, the rows are processed first. The matrix is then transposed, and the columns are then transformed to produce the final result.

This algorithm was implemented by the program FDCT.C (Appendix H) and was used exclusively for all the coding algorithms.

8. DCT coding techniques

8.1 Zonal coding

In a transform coding system, there are two basic strategies of selecting those coefficients which are to be retained : zonal sampling and threshold sampling. In the zonal coding system a set of zones is established in each transform block. The reconstruction is then made with these subsets of transform samples, usually containing the low frequency coefficients. For digital transmission, each component in a zone is quantized and assigned a binary code word. The number of quantization levels is usually made proportional to the estimated variance of the component, and the number of code bits is made proportional to the expected probability of occurrence.

There are several types of zones that could logically be employed for zonal sampling; for example, a rectangular, elliptical or triangular zone. Both analytic and experimental studies have shown that the optimum zone for a mean square error criterion is the so-called maximum variance zone [21]. The subset is chosen from those coefficents with the largest variances, while the remainder are discarded.

If we assume that N_B (u,v) code bits are assigned to each of the remaining coefficients, then we have

$L(u,v) = 2^{NB(u,v)}$ quantization levels.

Thus a total $N_B = \sum_u \sum_v N_B(u, v)$ bits are required to code the picture. For the bit assignment $N_B(u,v)$ for each coefficient we use the log variance relation described by Pratt[21]

$$N_B(u, v) = \frac{N_B}{N^2} + 2 \log_{10} [V_{\mathcal{F}}(u, v)] - \frac{2}{N^2} \sum_{a=1}^{N} \sum_{b=1}^{N} \log_{10} [V_{\mathcal{F}}(a, b)]$$

where $V_{\mathcal{F}}(u, v)$ is the variance of a transform coefficient.

It should be noted that quantization errors in the dc-coefficient causes artificial sub-block boundary discontinuities at reconstruction and should be avoided. The dc-coefficient is therefore always linearly quantized and coded without thresholding [20]. For the other coefficients however, the optimum placement of the quantization decision and reconstruction levels to minimize the mean square reconstruction error for a given number of quantization levels is given by the Max quantization strategy. [12,18,21]

This technique is known as block quantization. It is significantly more efficient than using the same number of bits for all coefficients. Its disadvantage lies in the problems inherent in handling binary words of unequal lengths.

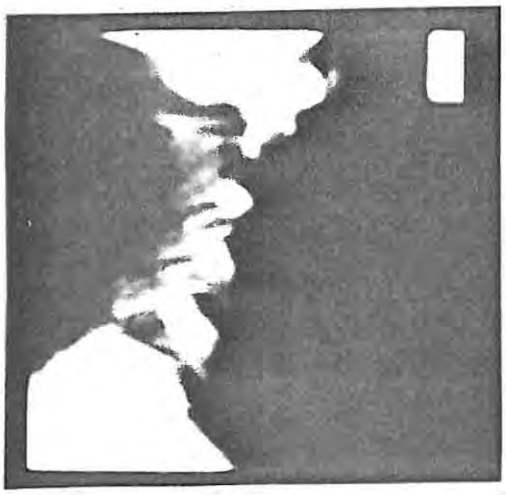
9	6	5	4	4	4	3	2
5	3	2	1	1	0	0	0
3	2	1	0	0	0	0	0
2	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0
0	0	0	1	1	0	0	0
1	0	0	0	0	0	0	0

a

9	5	4	3	2	2	1	0
4	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

b

Fig 5.1 (a) and (b) Bit assignments for DCT zonal coding of NECK.
8 x 8 blocks at a rate of a) 1, and b) 0.5 bits per pixel.


a

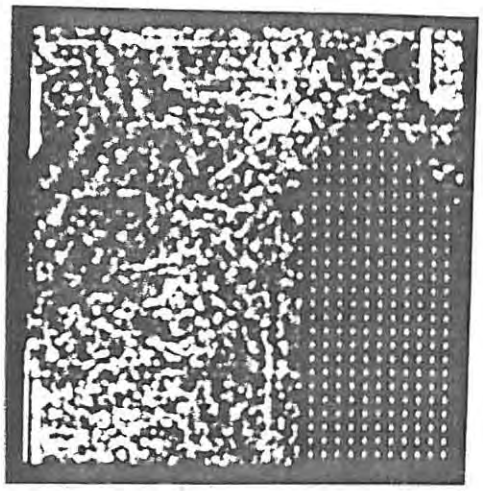
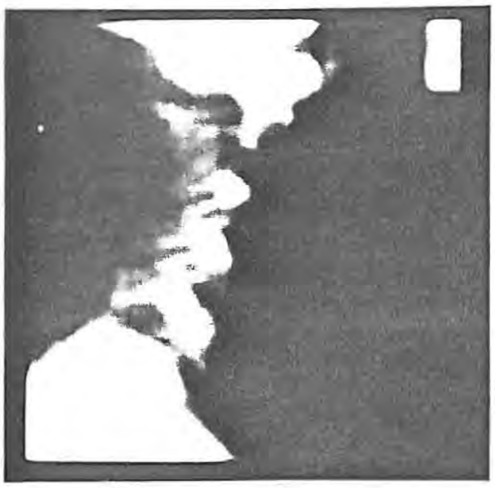

b

Fig 5.2 (a). Example of NECK, DCT zonal coded at 1 bit/pixel. %NMSE = 0.196.
b) Resulting error map.


a

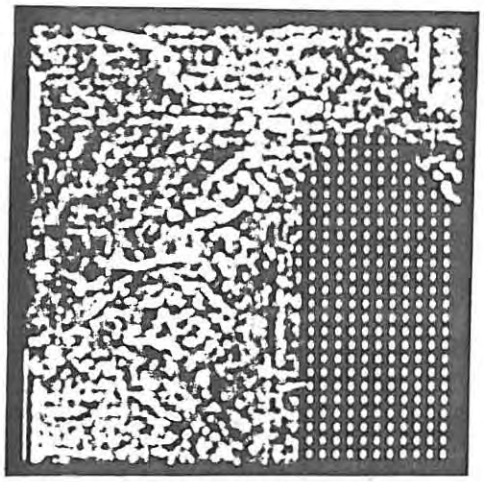

b

Fig 5.3 (a). NECK, DCT zonal coded at 0.5 bits/pixel. %NMSE = 0.332.
b) Resulting error map.

Wintz [30] states that in some cases the subjective quality can be slightly improved by assigning more bits to the coefficients with larger variances and fewer to the coefficients with smaller variances than suggested by the rule $m_{kl} \sim \log \sigma_{kl}^2$. Wintz states that this is probably due to the fact that the sensitivity of the human visual system to distortion as well as the coefficient variances is inversely proportional to spatial frequency. This fact was kept in mind when adjusting the bit allocation map, since the bit allocation scheme proposed by Pratt [21], does not produce the desired number of bits every time.

8.2 Threshold coding

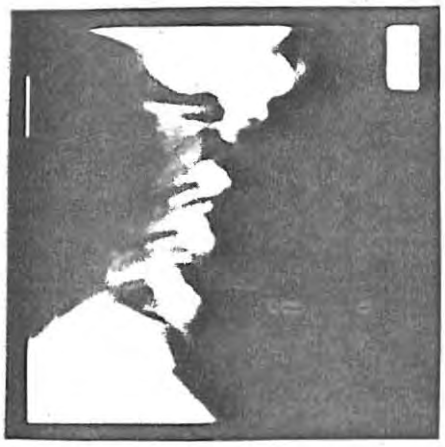
The difficulty with the zonal filter sampling method of bandwidth reduction is that large magnitude samples are indiscriminately discarded. An obvious answer to this problem is to code only those samples whose magnitudes are above a given threshold level. With this coding method it becomes necessary to provide information as to the location of significant samples. Selection of a threshold level for a given transform is generally a compromise between the number of samples deleted and the error resulting from the deletion of samples.

In threshold coding, the retained coefficients are not fixed a priori as in zonal coding, but are adaptively chosen to match input statistics. It is also possible to keep the total number of bits per block constant by using a variable threshold for transmitted coefficient magnitudes.

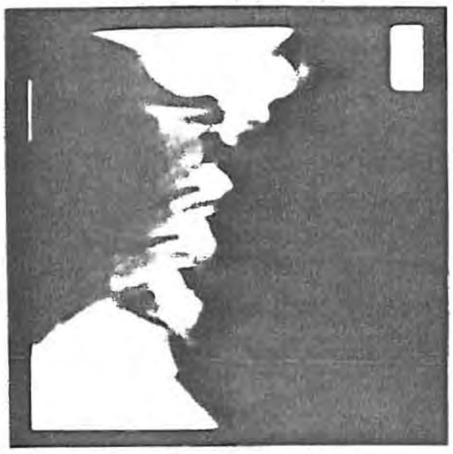
The first step in determining the optimum threshold is to get an indication of the distribution of the DCT ac-coefficients. This is shown in Fig 6.1.

THRESHOLD ABS VALUE	NECK	PLATE
1	89.1922	85.014343
2	95.739746	93.460083
3	96.986389	95.542908
4	97.514343	96.383667
5	97.862244	96.832275

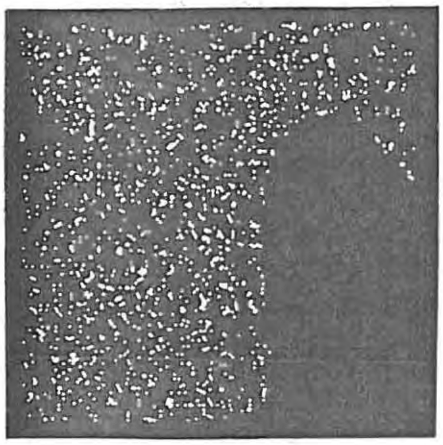
Fig 6.1 Distribution of Cosine Transform coefficients.
Percentage of coefficents below threshold value for NECK and PLATE.



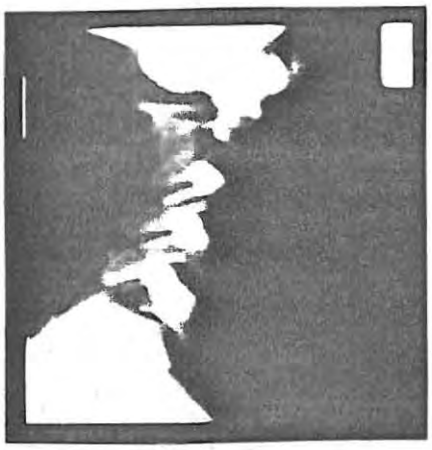
(a)



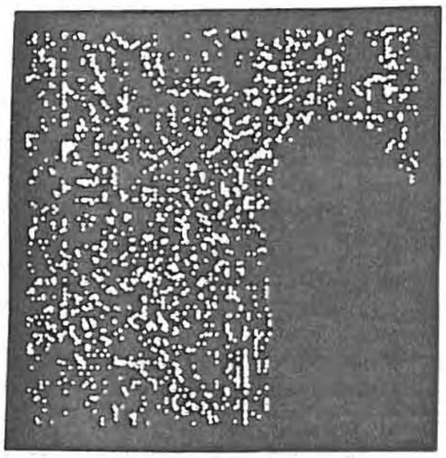
(b)



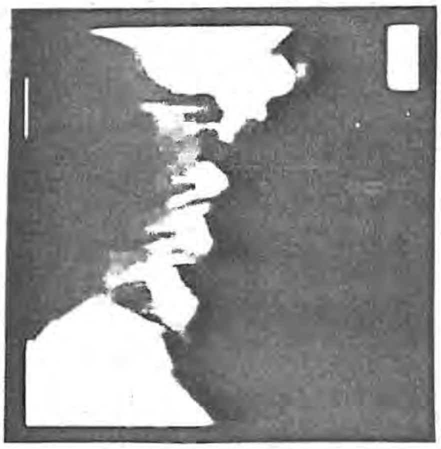
(c)

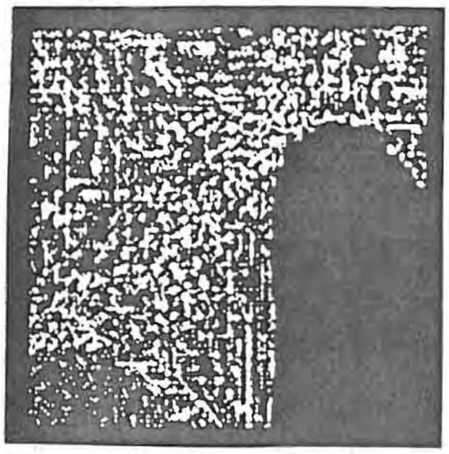


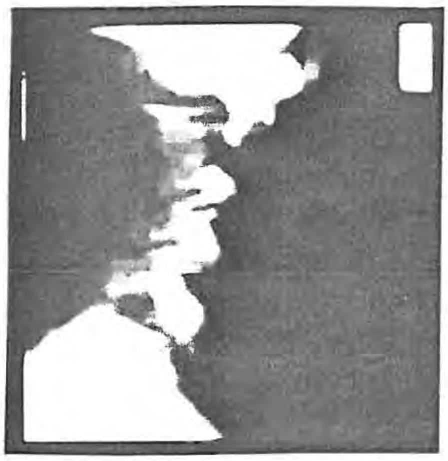
(d)

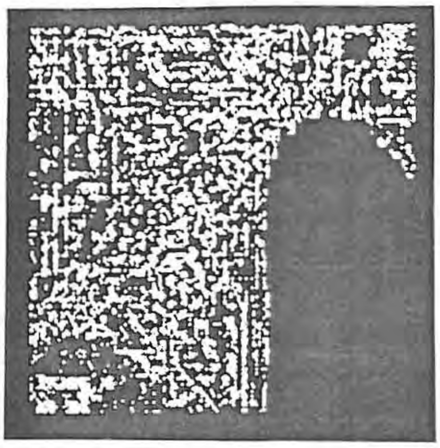


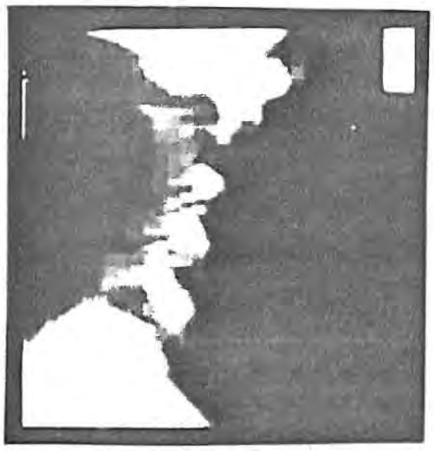
(e)


(f)


(g)


(h)


(i)


(j)

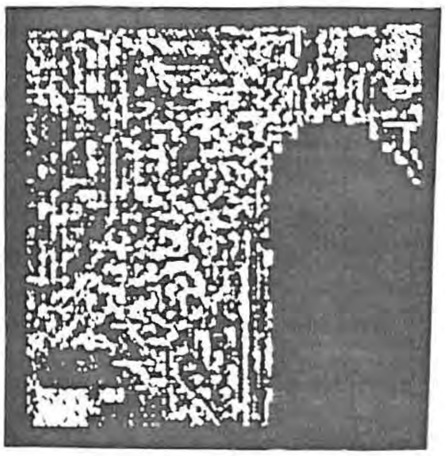
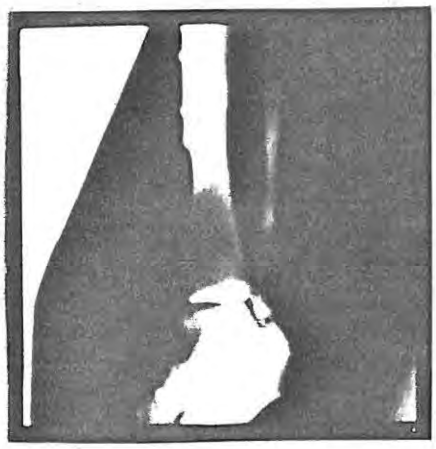
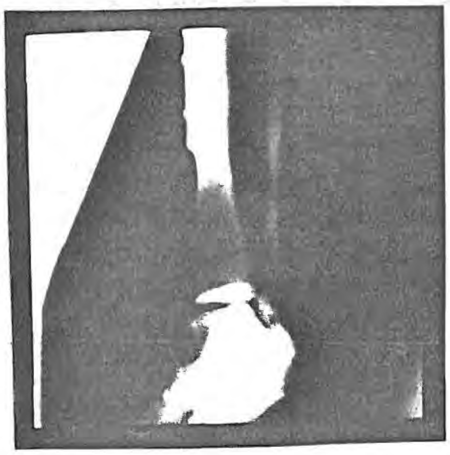

(k)

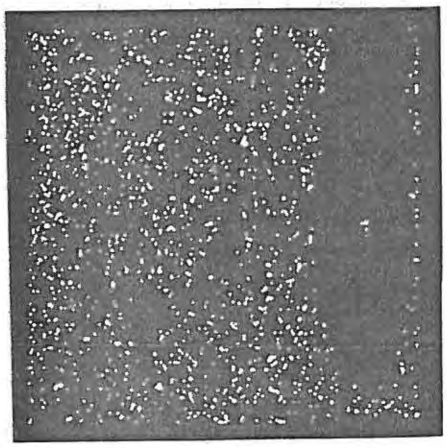
Fig 6.2. Results of DCT threshold coding NECK. a) original, b) threshold = 1, %NMSE = 0.058. d) threshold = 2, %NMSE = 0.112. f) threshold = 3, %NMSE = 0.144. h) threshold = 4, %NMSE = 0.172, and j) threshold = 5, %NMSE = 0.201.
Fig 6.2 (c), (e), (g) ,(i) and (k) show the resulting error maps.



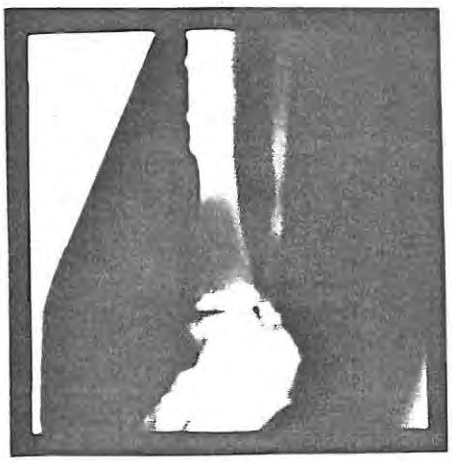
(a)



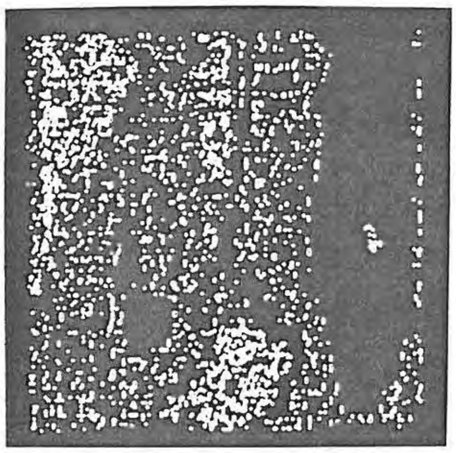
(b)



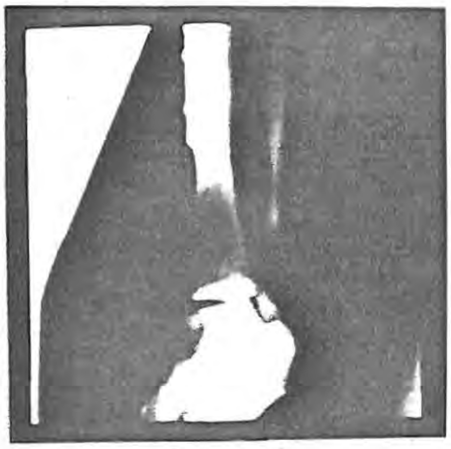
(c)



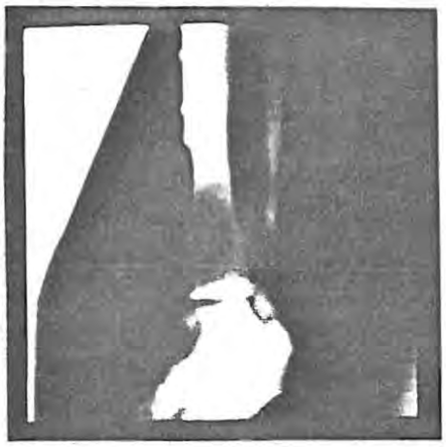
(d)

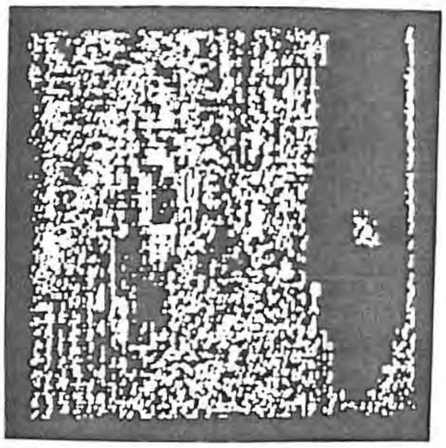


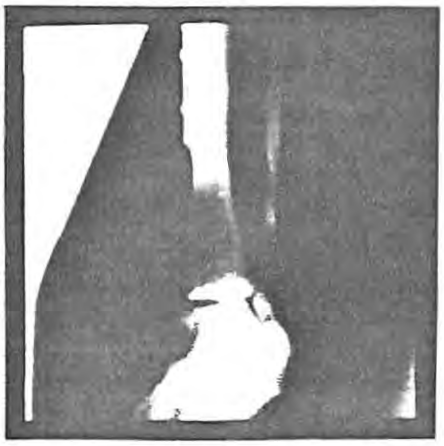
(e)


(f)

(g)


(h)


(i)


(j)

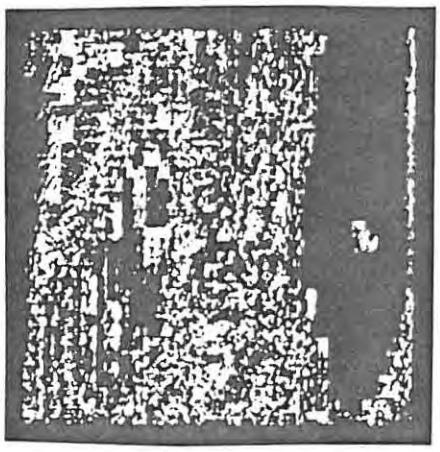

(k)

Fig 6.3. Results of DCT threshold coding PLATE. a) original, b) threshold = 1, %NMSE = 0.024. d) threshold = 2, %NMSE = 0.049. f) threshold = 3, %NMSE = 0.074. h) threshold = 4, %NMSE = 0.091, and j) threshold = 5, %NMSE = 0.107.

Fig 6.3 (c), (e), (g) ,(i) and (k) show the resulting error maps.

These computer simulation results, using real arithmetic on both images, have shown no visible degredation in picture reconstruction quality using a threshold value of 2. Note however, the blocking effects which worsen as the threshold value is raised above this level.

The biggest advantage of the threshold coding algorithm is the fact that the high frequency cosine coefficients are not simply discarded as is the case with zonal selection techniques [21]. The threshold operation selects only the most important (largest) frequency coefficients for coding, irrespective of position in the sub-block. As a result, this method is capable of high compression rates. Oosthuizen [20] reports good picture reconstruction quality at rates of 0,5 bits/pixel using the threshold selection technique.

On the other hand, one of the major drawbacks of this system is that although it is adaptive, it is an open loop system and the compression rate is image content dependant. Choosing the N-largest coefficients in each block will however produce a fixed compression rate system [21].

Although for the same number of transmitted samples a threshold mask would give a better choice of transmission samples (ie lower distortion) it would also result in an increased bit rate because the addresses of the transmitted samples (ie the boundary of the threshold mask) have to be coded for every image block. Typically, a sample line by line run-length coding scheme is implemented to code the transmission boundaries in the threshold mask. Usually this results in a somewhat more complex scheme than zonal transform coding. However threshold coding has its merits since it is adaptive in nature and is particularly useful **when the image statistics might change so rapidly that a fixed zonal mask is inefficient** [11].

Pratt [21] describes the following simple, but quite efficient run length coding technique to code the positions of the significant coefficients.

1) The first sample along each line is coded regardless of its magnitude. A position code word of all zeros or all ones affixed to the amplitude provides a line syncronization code group;

2) The amplitude of the second run-length code word is the coded amplitude of the next significant sample;

3) If a significant sample is not encountered after scanning the maximum run length of samples, the position and amplitude code bits are set to all ones to indicate a maximum run length.

Although the position coding adds to the bit rate, as a result of the adaptivity of this method, its performance is somewhat better than the simpler zonal coding process.

With standard thresholding coding technique described above, the number of code bits transmitted is image dependant. A variation of threshold coding, called N-largest coding, has been developed for communication links in which the transmission bit rate and picture transmission rate is fixed. This method codes the N-largest coefficients in a block regardless of their values. In effect, the threshold is adaptively set for each block to achieve the desired transmission rate without introducing great complexity [21].

8.3 An adaptive DCT coding scheme

Adaptive transform coding methods are based on the fact that images are composed of objects which are in turn a combination of edges and areas of varying degrees of detail. Using small block sizes, the degree of picture detail in each block changes rapidly. Transform encoders can be made to adapt to local image structure by allowing a number of "modes" of operation, and for each subimage, choosing the mode that is most efficient for that subimage [7]. Although such adaptive transform coders have proved to be highly effective, they do introduce system complexity, since extensive bookkeeping information indicating which mode was used must be coded along with the sub-block coeffients. The key to a practical system implementation lies in selecting an efficient coding scheme which achieves a successful compromise between complexity and performance. A comprehensive survey of many of these adaptive transform coding schemes is provided by Habibi [8].

Adaptive schemes using a finite number of bit maps are appealing because of the low overhead involved. According to Staelin *et al* [25], the best choice of the number and form of these maps is a difficult problem which has received little attention. Staelin uses pattern recognition techniques in designing a classificatory scheme for adaptive transform coding. Although performing well, this scheme introduces additional complexity. Tesic [26] segments chest x-ray images into regions of high and low entropy and uses data compression and pattern recognition techniques to achieve a relatively large compression ratio.

The technique developed in this dissertation is based on the work of Chen & Smith [3]. The basic approach is to divide the transform sub-blocks to be coded into a finite number of classes, according to their "activity index", which represents the amount of "activity" or "detail" in each transform sub-block. Bits are then distributed among "busy" and "quiet" image areas according to the level of activity, with more bits going to high activity classes and fewer bits to classes with low activity [20]. A block diagram of the cosine transform adaptive coding system is presented in Fig 7.1.

The algorithm is basically a two pass algorithm : the first pass generates the sub-block classification maps, sets up the bit assignment matrices and calculates the normalization factors needed. The second pass multiplies the normalization factor, quantizes the transform coefficients, codes the data, adds the overhead information and stores the resulting data.

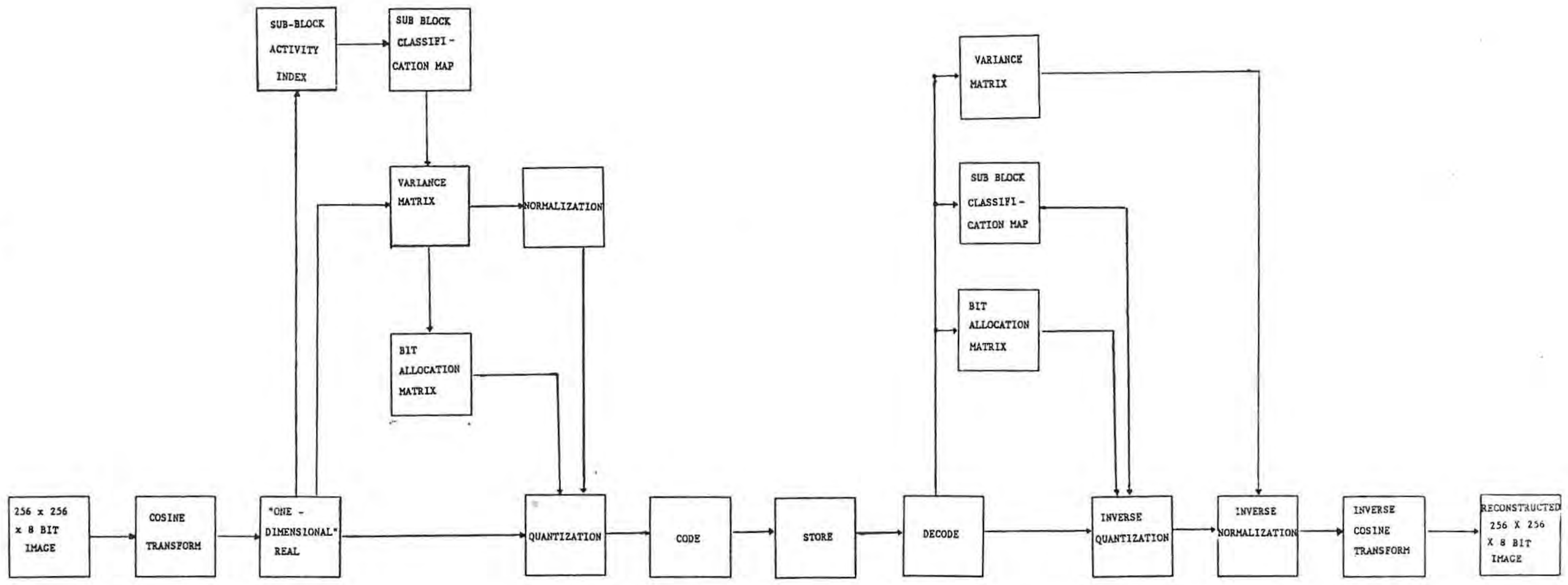


Fig 7.1 The Cosine transform adaptive coding system.

In an attempt to improve the performance of the Chen and Smith algorithm, a few minor changes were made. Instead of using the sum of the squares of the coefficients in the transform domain in order to classify each sub-block, the sum of the absolute values (activity index) was used. The transformed coefficients are stored as a one-dimensional vector in order to simplify the classification, normalization, quantization and coding operations which follow. The normalization factor for each coefficient is calculated directly from the square root of the corresponding entry in the variance matrix. The bit allocation algorithm is based on the procedure described by Pratt [21]. Obviously, an increased number of classes would improve the image quality for a given bit rate. Using 16 classes, it was possible to code both NECK and PLATE at rates of below 0.3 bits/pixel. According to Staelin *et al* [25] the point of diminishing returns is reached for 16 to 32 classes for 8 x 8 sub-blocks, but is not reached for 16 x 16 blocks. (Other suggestions for the optimization of this technique are presented in the next chapter.)

If one looks at the bit allocation maps for NECK (Fig 7.3), it can be seen that for those sub-blocks in the lowest activity index, only 1 bit per 64 pixels (0.016 b/p) is needed to provide a faithful reconstruction of large areas of the original image. X-ray images, with their large areas of uniform brightness, are well suited to this algorithm. Another feature of this algorithm is the manner in which large magnitude high frequency coefficients are protected by obtaining more bits than smaller nearby lower frequency samples.

3	3	2	1	4	3	3	2
4	4	2	2	3	2	2	1
4	3	1	4	1	3	2	1
3	2	1	1	4	1	3	3
2	4	4	3	1	2	1	1
3	2	4	4	3	1	1	1
2	2	1	3	3	4	1	1
4	1	4	2	1	4	2	1

Fig 7.2 Classification of 8 x 8 transform sub blocks for NECK.
Class 1 is the lowest activity, while class 4 is the highest.

1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Class 1

8	2	0	0	0	0	0	0
1	0	0	0	0	1	1	0
0	0	0	0	0	1	1	0
0	0	0	0	0	0	1	0
0	0	0	0	1	0	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Class 2

8	4	2	0	0	0	1	0
4	2	0	0	0	1	1	1
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1

Class 3

8	5	4	4	4	3	3	2
5	3	1	1	0	0	0	0
2	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Class 4

Fig 7.3 Bit allocation maps for NECK. Bits/pixel = 0.367.

Unlike most other adaptive methods of this nature, the bookkeeping information is kept to a minimum. According to Chen and Harrison Smith [3], less than 0.034 bits of overhead information is needed to code a one bit monochrome image of 256 x 256 pixels. As the image dimensions increase, this figure is dramatically reduced.

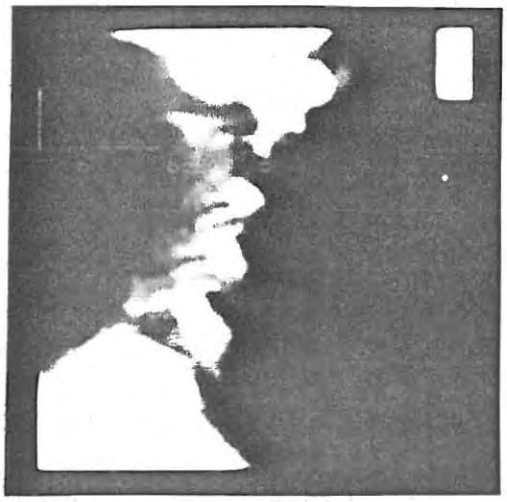


Fig 7.4. Adaptive coding scheme. NECK coded at 0.367 b/p. %NMSE = 0.153. Bit maps used are displayed in Fig 7.3.

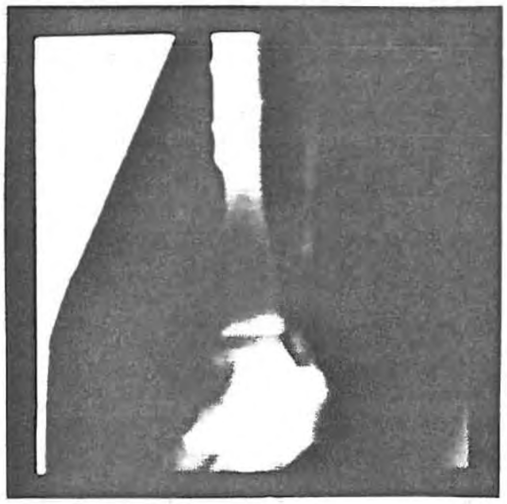


Fig 7.5. Adaptive coding scheme. PLATE coded at 0.354 b/p. %NMSE = 0.119.

9. Aspects for further study

9.1 Reduction of blocking effects

Transform coding takes advantage of the correlation between adjacent pixels by reducing the redundancy. Because of block segmentation, statistical dependancies beyond the block boundaries are not taken into account.

In very low bit rate applications, the block boundaries may become visible. With the DCT, mismatches between adjacent blocks are visible, especially in highly structured regions of the image [24]. Blocking effects can be the limiting factor in reducing the bit rate. Various schemes have been proposed in order to reduce this problem.

9.1.1 Overlap method

The visibility of boundaries can be reduced by overlapping the blocks before transform coding. The pixels on the border of each region would then be coded in two or more regions. The main disadvantage of this method is the increased bit rate resulting from the redundancy of the image representation due to overlapping blocks.

9.1.2 Filtering method

Using this method, an image enhancement procedure is applied **after** decoding. Since the discontinuities are similar to very sharp edges, a low pass filter is used to smooth out the image. An obvious advantage of the filtering method is the fact that it does not increase the bit rate.

9.1.3 Lapped orthogonal transforms

Staelin *et al* [25] introduces a new type of transform which significantly reduces block noise, while retaining a low bit rate. The lapped orthogonal transform is defined as as a separable unitary transformation for which the basis functions corresponding to adjacent data blocks overlap. Results obtained showed that the LOT makes an image look smoother than the DCT coded image. It also renders the block noise less noticeable.

9.2 Bit allocation

The bit assignment approach described by Pratt [21] and adopted in the simulation algorithm does not provide optimum performance since the real numbers need to be "fiddled" to provide the desired bit rate. A computational method for bit assignment based on a technique called marginal analysis has been found to achieve performance superior to other algorithms. This approach allocates the total number of available bits on a bit by bit basis. Although this approach provides superior results, the computational load is extremely heavy. Tzou [27] proposes a fast computational algorithm for optimum bit assignment which is based on marginal analysis but has been shown to be much more efficient than any other algorithms. The use of such an algorithm however, would depend on the tradeoff between increased complexity and computation time against the improvement in quality of the reconstructed image.

9.3 Nonuniform quantization

The aim of the quantizer is to determine the optimum transition and reconstruction levels for the transform coefficients. The quantizer used in this simulation is the uniform Max quantizer [18]. Tesic [27] states that for input data with a Laplacian distribution, the **nonuniform quantizer** will be 4.3 dB better than the optimum uniform quantizer. Once again, the reason for not using the nonuniform quantizer is the burden of increased complexity and computation time, since the nonuniform quantizer consists of a uniform quantizer preceeded and succeeded by nonlinear transforms. Once again the increased performance would have to be weighed up carefully against these negative factors.

9.4 Error protection

A major advantage of image transform coding is its tolerance to channel errors. The inherent "error averaging" property of transform coding allows an error in a transform coefficient to be distributed over the entire sub picture, making it less objectionable from a human visual standpoint. On the other hand, using a predictive compression technique, an error in an element of the differential data would result in an incorrect reproduction of a picture element, and as a result of the predictor at the receiver, this error would propagate to neighbouring pixels.

Although errors in transform coefficients are not as annoying as the distortions introduced by other coding schemes, attention must be paid to toward suppressing error influences, since they might lead to misinterpretations by physicians.

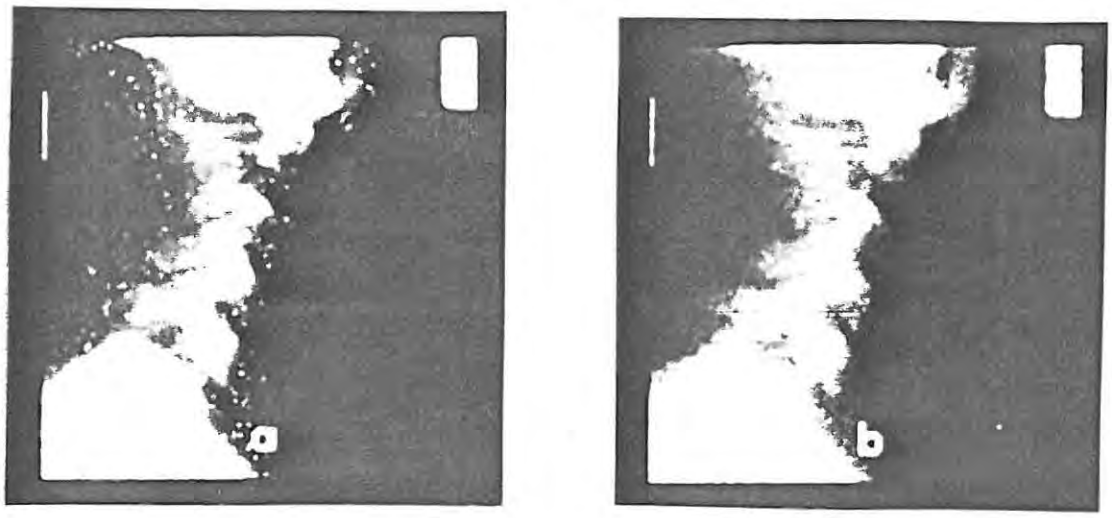

a b

Fig 8.1 Binary symmetric channel noise in spatial and transform domain transmission.
a) 10^{-2} error rate in the spatial domain. b) 10^{-2} error rate in transform domain.

The effect of channel errors and the use of error correcting codes for transform coding has been investigated, among others, by Lux [15], Andrews[1], Oosthuizen [20] and Wintz [29].

Oosthuizen [20] states that for high quality image reconstruction, no meaningful error protection is necessary for transmission probability error rates of less than 10^{-3} and peak-to-peak signal to rms noise higher than 30 dB. Chen and Harrison Smith [3] state that for an adaptive transform coding scheme of this nature, only the overhead part of the data (classification map, normalization factors and bit tables) requires error correction to produce yreliable data transmission. If protection of this nature were required, it could be added to the algorithm with a negligible increase in the bit rate and a minimum of fuss.

10. Discussion and conclusion

An adaptive transform coding technique for digital x-ray images has been developed. A number of computer simulations have been conducted to evaluate the performance of the coder for x-ray images. The research has shown that the coding of these images offers the possibility of high compression rates. The statistics show much correlation inherent in this type of image, and the classification of image sub-blocks by activity level provides efficient coding for the transform domain elements.

The adaptive coding technique requires minimal overhead information yet performs extremely well for this class of image, while the flexibility of this simulation code allows easy expansion to any image size or number of classes. The results indicate that the adaptive cosine transform scheme can produce good quality image reconstruction at bit rates below 0.38 bits/pixel. Greater compression could be achieved if larger error in the decompressed images were found acceptable for diagnostic purposes.

As far as implementation is concerned, the adaptive transform coding technique shows better results than the other methods discussed, although the implementation of such a system would require more complex hardware.

Tesic [26] states that the ultimate compression algorithm is the equivalent of the radiologist's dictated report. Research in machine vision may ultimately succeed in this endeavor although the answer seems very far away at this stage and much effort is still needed. At present less ambitious algorithms, although having many drawbacks, have proven more fruitful.

Bibliography

[1] H. C. Andrews. **Computer Techniques in Image Processing.** Academic Press. New York. 1970.

[2] A.D. Brink. **The Selection and Evaluation of Grey-Level Thresholds Applied to Digital Images.** M.Sc. Thesis, Department of Physics and Electronics, Rhodes University. 1987.

[3] W. H. Chen and C. Harrison Smith. **Adaptive Coding of Monochrome and Colour Images.** IEEE Trans on Comm., vol COM-25, no. 11, pp. 1285-1292. Nov. 1977.

[4] W.H. Chen, C. Harrison Smith, and S. C. Fralick. **A Fast Computational Algorithm for the Discrete Cosine Transform.** IEEE Trans. on Comm. VOL. COM-25, NO 9, Sept 1977. pp. 1004-1009.

[5] R. J Clarke. **Relation between the Karhunen-Loeve and Cosine Transforms.** IEEE Proc. vol 128. Pt. F no-6. pp. 359-360. Nov. 1981.

[6] A. J Duernickx. **Digital Picture Archiving and Communications Systems in Medicine.** Computer. vol 16. no 8. pp. 14-16. August 1983.

[7] R. C. Gonzalez and P Wintz. **Digital Image Processing.** Addison-Wesley, Reading Massachusetts. 1977.

[8] A. Habibi. **Survey of Adaptive image Coding Techniques.** IEEE Trans on Comm. vol COM-25, no 11. pp. 1275-1284. Nov. 1977.

[9] Munsi Alual Haque. **A Two-Dimensional Fast Cosine Transform.** IEEE Trans. on Acoustics, Speech and Signal Processing, vol. ASSP - 33, No. 6, pp. 1532-1538. Dec. 1985.

[10] H. S. Hou. **A Fast Recursive Algorithm for Computing the Discrete Cosine Transform.** IEEE Trans. on Acoustics, Speech and signal processing. vol ASSP-35, no 10, pp. 1455-1461. Oct 1977.

[11] J. R. Jain. **Image data compression.** Proc of the IEEE. vol 69. no 3. pp. 367-369. March 1981.

[12] N. S. Jayant and P. Noll. **Digital coding of Waveforms.** Prentice Hall, New Jersey. 1984.

[13] Marut Kunt. **Second-Generation Image Coding Techniques.** Proceedings of the IEEE, vol 73, No 4, pp. 549-574. April 1985.

[14] M. Kunt. **An electronic file for x-ray pictures.** Lecture notes in computer science. vol 80. Springer-Verlag, Berlin. 1980.

[15] P. Lux. **Redundancy reduction in radiographic pictures.** Optica Acta, vol 24, No 4, pp. 349-365. 1977.

[16] John Makhoul. **A Fast Cosine Transform in One and Two Dimensions.** IEEE Trans. on Acoustics, Speech and Signal Processing, vol ASSP-28, No 2, pp 27-34. Feb. 1980.

[17] H. Malvar. **Fast Computation of the Discrete Cosine Transform through the Fast Hartley Transform.** Research Lab of Electronics. Massachussetts Institute of Technology. Cambridge, MA. 021139, USA.

[18] Joel Max. **Quantizing for minimum distortion.** IRE Trans. Inform. Theory, vol IT-6, pp. 7-12. March 1960.

[19] Dietrich Meyer-Ebrecht and Thomas Wendler. **An architectural route through PACS.** Computer. vol 16. no 8. pp. 19-28. August 1983.

[20] M. J. Oosthuizen. **A Low Bit Rate Encoder Algorithm for Television Signals.** Ph.D. Thesis. Department of Electronic engineering, University of Pretoria. July 1985.

[21] Pratt, W. K. **Digital Image Processing.** John Wiley & Sons, New York. 1978.

[22] Reininger, R. C., and J. D. Gibson., **"Distribution of the Two-Dimensional DCT Coefficients for Images".**, IEEE Trans. Commun., vol. COM-31, No. 6, June. 1983.

[23] A. Rosenfeld and A. C. Kak. **Digital Picture Processing.** Academic Press, New York. 1976.

[24] W. F. Schreiber. **Fundamentals of Electronic Imaging Systems.** Springer Verlag, Berlin. 1986.

[25] D. H. Staelin, A.D.S. Ali, P.M. Cassereau, G. de Jager, and A.M. Kirkland, **Efficient Coding of Video Images.** M. I. T. Research Laboratory of Electronics, Cambridge Massachusetts. August 1986.

[26] M. M Tesic, V.C. Chen, and K. L. Lauro. **Digital Chest image compression through coding by synthesis.** SPIE. vol 454. Application of Optical Instrumentation in Medicine XII. pp. 331-338. 1984.

[27] Kou-Hu Tzou. **A fast computational approach to the design of block quantization.** IEEE Trans on Acoustics, Speech and signal processing. vol. ASSP-35. no 2. pp. 235-237. Feb. 1987.

[28] M. Unser. **On the approximation of the Discrete Karhunen-Loeve Transform for stationary processes.** Signal Processing. vol 7. pp. 231-249. Jan. 1984.

[29] M. Vetterli, and H. J Nussbaumer. **Simple FFT and DCT Algorithms with reduced number of operations.** Elsevier Science Publishers, B.V. (North-Holland) pp. 267-278. 1984.

[30] P. A. Wintz. **Transform Picture Coding.** Proceedings of the IEEE, vol 60, No 7, pp. 809-819. July 1972.

[31] R. C. Wintz, and P. Wintz. **Digital Image Processing.** Addison Wesley. Reading, Massachusetts. 1977.

[32] The PC-EYE Video Capture System Technical Reference Manual.

PROGRAM LISTINGS

APPENDIX A

```
/*________________________PC2.C_______________________________*/
/* This program gives the user the ability to cature, store and display   */
/* an image captured using the PC-EYE System. The image is then processed */
/* and RMF 'ed to the VAX for display purposes.                           */
/* I H Mclean 28/4/87                                                     */
/*________________________________________________________________________*/
#include <stdio.h>
#include <fcntl.h>

void main()
{
  int
  newimage,                /* are we going to capture two images ?  */
  bwflag,                  /*  "boolean" flag to test b/w levels    */
  horizontal, vertical,    /* dimensions of captured image          */
  pbase, gbase,            /* pointers to memory                    */
  dmem, mmem,              /* memory locations - start of image     */
  inter, xtrg,
  iomode, memtyp,          /* parameters which set the base type    */
  reply,                   /* parameter supplied by user            */
  black, white,            /* the b/w levels - 8 & 26 optimum       */
  left, top,
  voff, hoff,              /* parameters which ajust image to screen */
  count,                   /* the throttle count                    */
  type,                    /* memory type - display or main memory  */
  ioff, ichan,
  unpack;                  /* pixels are unpacked - 1 p/b           */

  extern int
  hmax, vmax,              /* maximun dimensions    640 x 400       */
  bitpel,                  /* bits per pel                          */
  dpbyt,                   /* pels per byte                         */
  punp;                    /* packed or unpacked flag               */

  short
    dispmod;

  char c, d,          /* simple variables for user input       */
       filename;      /* will hold the name of the file on disk */

  bwflag = 0;         /* no error yet          */
  newimage = 0;       /* no new image as yet */
  lab2:
  dispmod = 0X0;     /* set screen to text mode */
  SETCRT(dispmod);
```

```
 printf(" _______________________________________\n");
 printf("|                                       |\n");
 printf("|  PCI  MENU                            |\n");
 printf("|                                       |\n");
 printf("|  S - SAVE AN IMAGE ON DISK            |\n");
 printf("|  F - TO RESET FRAME SIZE              |\n");
 printf("|  G - TO GRAB ANOTHER IMAGE            |\n");
 printf("|  L - SET BLACK/WHITE LEVELS           |\n");
 printf("|  A - ADD TWO IMAGES - ON DISK         |\n");
 printf("|  D - STATS ON AN IMAGE ON DISK        |\n");
 printf("|  W - STATS ON AN IMAGE IN MEMORY      |\n");
 printf("|  E - STATS ON AN IMAGE IN DISP MEMORY |\n");
 printf("|  M - TO RETURN TO THE MENU            |\n");
 printf("|  Q - TO QUIT                          |\n");
 printf("|                                       |\n");
 printf("|  PRESS ANY KEY TO CONTINUE            |\n");
 printf("|_______________________________________|\n");

d = getch();
if (d == 'q') goto end;
if (d == 'a') {addimages();}

labl:
dispmod = 0X0;
SETCRT(dispmod);
printf("NAME OF FILE : \n"); scanf("%s",filename);
printf("FRAME WIDTH :\n"); scanf("%d",&horizontal);
printf("FRAME HEIGHT :\n"); scanf("%d",&vertical);
left = (640-horizontal)/2; top = (400-vertical)/2;

lab:
dispmod = 0X0;
SETCRT(dispmod);
/* SET BASE ADDRESSES */
pbase = 0X310; gbase = 0X3D0; dmem = 0XA000; mmem = 0X2000;
reply = CADDR(pbase,gbase,dmem,mmem);
if (reply != 0) printf("ILLEGAL ADDRESS");
if (reply == 0)
  {
    /* SET BASE CONFIGURATION */
    inter = 0; xtrg = 0; iomode = 0; memtyp = 5;
    reply = CSETMOD(inter,xtrg,iomode,memtyp);
    if (reply != 0) printf("ILLEGAL MEMORY TYPE");
  }
if (reply == 0)
  {
    setlev:
    /* AJUST BLACK/WHITE LEVELS */
    black = 10; white = 35;
    if (bwflag == 1)
      {
        dispmod = 0X0;
        SETCRT(dispmod);
```

```
          printf("BLACK LEVEL : \n"); scanf("%d",&black);
          printf("WHITE LEVEL : \n"); scanf("%d",&white);
        }
      reply = CCALIB(black,white);
      if (reply == 1) printf("VALUE OUT OF RANGE");
      if (reply == 10) printf("BLACK LEVEL GREATER THAN WHITE LEVEL");
    }
  if (reply == 0)
    {
      /* ESTABLISH DISPLAY FRAME */
      reply = CDFRAME(left,horizontal,top,vertical);
      if (reply == 8) printf("VALUE OUT OF RANGE");
      if (reply == 7) printf("UNEVEN MARGIN SETTING");
    }
  if (reply == 0)
    {
      /* AJUST IMAGE TO SCREEN */
      hoff = 60; voff = 12;
      reply = CAFRAME(hoff,voff);
      if (reply == 12) printf("VALUE OUT OF RANGE");
      if (reply == 11) printf("UNEVEN MARGIN SETTING");
    }
  if (reply == 0)
    {
      /* SET THROTTLE COUNT */
      count = 14;       /* 14 */
      reply = CTHROTL(count);
      if (reply == 1) printf("VALUE OUT OF RANGE");
      if (reply == 2) {
                        printf("THROTTLE COUNT POTENTIALLY TOO SMALL");
                        reply = 0;
                      }
    }
  if (reply == 0)
    {
      /* SET A PC-EYE REGISTER */
      ioff = 1; ichan = (4-1)*16;
      reply = CSETREG(ioff,ichan);
      if (reply == 1) printf("VALUE OUT OF RANGE - CANNOT EXCEED 1 BYTE");
    }
  if (reply == 0)
    {
      /* SET THE DISPLAY MODE ie GRAPHICS MODE */
      dispmod = 0XF;
      CSETCRT(dispmod);          /* no reply is returned */
    }
  {
    /* ACQUIRE AN IMAGE AN STORE IN MAIN MEMORY */
    unpack = 0;
    reply = CMDMA(unpack);
    if (reply == 4) printf("FIFO OVERFLOW ERROR");
  }
```

```
    {
      /* TRANSFER FROM MAIN MEMORY TO DISPLAY MEMORY */
      type = 0;
      CMMTDISP(type); /* no reply */
    }
    {
      c = getch();      /* do */
      if (c == 'g') goto lab;
      if (c == 'f') goto lab1;
      if (c == 'm') goto lab2;
      if (c == 'q') goto end;
      if (c == 'l')
                  {
                    bwflag = 1;
                    goto setlev;
                  }
    if (c == 's')
      {
        dispmod = 0X0;
        SETCRT(dispmod);
        bitpel = 6; dpbyt = 1; punp = 0;  /* 6 bits/pixel, 1 pixel/byte */
        reply = CEYEWRIT(filename,3);
      }
    if (c == 'e') {stsdisp(horizontal, vertical);}
    if (c == 'd') {
                  dispmod = 0XF;
                  CSETCRT(dispmod);        /* no reply is returned */
                  reply = CEYEREAD("b:t",3);
                  CMMTDISP(0);
                }

    }
    {
      goto lab2;
      end:
      dispmod = 0X0;
      SETCRT(dispmod);
    }
}
/*_____________________________________________________________________*/

/* statsdisp is a function similiar to the STATS routine on the VAX. It */
/* provides the user with statistics such as mean, highest and lowest   */
/* pixel values.                                                        */

stsdisp(h,v)
    int
       h, v;   /* horizontal & vertical dimensions */
{      float
       total;

    unsigned
      c, dispmod, col, row,
      n, pixval,
      picarray[16];            /* holds the values of the pixels */
      for (n = 0; n <= 15; ++n )  {picarray[n] = 0;}
      row = 0; total = 0;
     /* The pixels are read directly from memory, giving the user the */
     /* option to change certain parameters before storing the image  */
      while (row != h)
      {
        col = 0;
        while (col != v)
        {
          pixval = CRDRWCL(row,col);     /* grab pixel from memory */
          ++picarray[pixval];
          total = total + pixval;
          ++col;
        }
        ++row;
      }
    dispmod = 0X0;     /* set screen to text mode */
    SETCRT(dispmod);
    printf("DIMENSIONS : %d  x  %d \n",h, v);
    printf("AVERAGE PIXEL VALUE :  %d \n",total/(h*v));
    printf("\n"); printf("\n");
    printf("PIXEL VALUE      FREQUENCY \n");
    for (n = 0; n <= 15; ++n )
      {
        printf(" %2u_______________%u \n",n,picarray[n]);
      }
    printf("TOTAL :              %f \n \n \n",total);
    printf("          ANY KEY TO RETURN TO MENU");
    c = getch();
  }    /*stsdisp*/
/*_____________________________________________________________________*/

/* This function enables the pixel values of two images of EQUAL SIZE to   */
/* be added. This effectively doubles the dynamic range of the resulting   */
/* image.                                                                  */

  addimages()
{
  int
   d;   /* I/O result */

  long
    h, v,      /* horizontal and vertical dimensions    */
    count;

  short
    dispmod;

  extern int
  _fmode;

  char  p1, p2,       /* input pixels from files 1 & 2  */
        outbyte,      /* output byte                    */
        file1, file2,
        outfile;
```

```
FILE  *f1, *f2, *fout;  /* pointers to our files */

    {
     dispmod = 0X0;
     SETCRT(dispmod);
     printf("HORIZONTAL SIZE : \n"); scanf("%d",&h);
     printf("VERTICAL SIZE : \n"); scanf("%d",&v);
     printf("NAME OF FILE NO 1 : \n"); scanf("%s",file1);
     printf("NAME OF FILE NO 2 : \n"); scanf("%s",file2);
     printf("NAME OF OUTPUT FILE : \n"); scanf("%s",outfile);
     printf("%d \n",h*v);
     _fmode = 0X8000;
     f1 = fopen(file1,"r"); f2 = fopen(file2,"r");
     fout = fopen(outfile,"w+");
     for  (count = 1; count <= 256; ++count)
     {
       d = fscanf(f1,"%c",&p1);
       d = fscanf(f2,"%c",&p2);
       d = putc(p1,fout);
     }
     for  (count = 1; count <= 256000; ++count)   /* 640 x 400 */
          {
            d = fscanf(f1,"%c",&p1);
            d = fscanf(f2,"%c",&p2);
            outbyte = p1 + p2;
            d = putc(outbyte,fout);
          }
     fclose(f1); fclose(f2);
     fclose(fout);
   }
 }     /* addimages */
/*___________________PC2.C______________________________________*/
```

APPENDIX B

```
      PROGRAM PCEYE
C     =============
C
C     Program to convert pc-eye format data to starlink bdf.
C
C     Original written by J.L.Jonas
C     Modified by A.D.Brink,01/11/1986
C
C     Modified once again by I.H. Mclean, 16/3/87
C     The modifications will enable the program to unpack
C     the 2 pixels per byte, storing each 4 bit pixel in
C     a single byte and masking out the high order nibble.
C
      CHARACTER FNAME*16
      INTEGER POINTER,  SIZE(2), JUNK(16)
C     INCLUDE 'STARDIR:FMTPAR.FOR'
C
        CALL RDKEYC ('INPUT',.FALSE.,1,FNAME,JUNK,ISTAT)
        OPEN (UNIT=10,FORM='UNFORMATTED',RECORDTYPE='FIXED',
     *        NAME=FNAME,STATUS='OLD')
        SIZE(1) = 640
        SIZE(2) = 400
        CALL RDKEYI ('SIZE',.TRUE.,2,SIZE,JUNK,ISTAT)
        CALL WRIMAG ('OUTPUT',FMT_UB,SIZE,2,POINTER,ISTAT)
C
        K = 0
        DO I = 1 , 4
          READ (10) JUNK
        END DO
        CALL READ_AND_COPY (%VAL(POINTER),SIZE(1),SIZE(2))
        CALL EXIT
      END
      SUBROUTINE READ_AND_COPY (A, M, N)
C     =========================
C
C Convert PC-EYE 6 bits/pel format to Starlink .BDF format
C with 8 bits/pel.No multiplication to convert 64 levels to
C 256 occurs in this version.
C
      BYTE A(M*N)
      BYTE TEMPARRAY(64*128)
      BYTE TEMPARRAY2(64*128)

        ITOT = (M*N) / 2 - 1
        DO K = 0 ,ITOT , 64
          READ (10) (TEMPARRAY(I) , I = K + 1 , MIN (K + 64 , ITOT))
        END DO
```

```
C  CODE WHICH INVERTS THE IMAGE - IBM FORMAT TO VAX FORMAT

         DO P = 0,N-1
           DO Q = 0,(M/2)-1
             TEMPARRAY2((P*(M/2)) + Q) = TEMPARRAY((N-P+1)*(M/2)+Q)
           END DO
         END DO

C  CODE WHICH UNPACKS THE 2 PIXELS CONTAINED IN EACH BYTE
C  THE HIGHER ORDER NIBBLE OF EACH BYTE IS MASKED

        DO X = 1, ITOT
          A((2*X-1)) = ((TEMPARRAY2(X) .AND. 'F0'X)/16) .AND. '0F'X
          A(2*X)     = (TEMPARRAY2(X) .AND. '0F'X)
        END DO
        RETURN
      END
```

APPENDIX C

```
/*___________________________ADD.C___________________________*/

#include <stdio.h>
#include <fcntl.h>
#include <math.h>

void main()
{
/* This program enables the pixel values of 4 images of EQUAL SIZE to be  */
/* added. This allows us to produce 8 b/p images.                         */
/* I H Mclean 28/4/87                                                     */

  int
    bitnum, d;

  long
    size,
    h, v, count;

  extern int
  _fmode;

  char  p1, p2, p3, p4, file1, file2, file3, file4, outfile1;

  FILE  *f1, *f2, *f3, *f4, *fout1;

    {
      printf("HORIZONTAL SIZE : \n"); scanf("%d",&h);
      printf("VERTICAL SIZE : \n"); scanf("%d",&v);
      printf("NAME OF FILE NO 1 : \n"); scanf("%s",file1);
      printf("NAME OF FILE NO 2 : \n"); scanf("%s",file2);
      printf("NAME OF FILE NO 3 : \n"); scanf("%s",file3);
      printf("NAME OF FILE NO 4 : \n"); scanf("%s",file4);
      printf("NAME OF OUTPUT FILE  : \n"); scanf("%s",outfile1);
      _fmode = 0X8000;
      size = h*v;
      f1 = fopen(file1,"r");
      f2 = fopen(file2,"r");
      f3 = fopen(file3,"r");
      f4 = fopen(file4,"r");
      fout1 = fopen(outfile1,"w+");
      for  (count = 1; count <= 256; ++count)     /* copy the headers */
      {
        d = fscanf(f1,"%c",&p1);
        d = fscanf(f2,"%c",&p2);
        d = fscanf(f3,"%c",&p3);
        d = fscanf(f4,"%c",&p4);
        d = putc(p1,fout1);
      }
    for  (count = 1; count <= size; ++count)
         {
           d = fscanf(f1,"%c",&p1);
           d = fscanf(f2,"%c",&p2);
           d = fscanf(f3,"%c",&p3);
           d = fscanf(f4,"%c",&p4);
           bitnum = (p1+p2+p3+p4) ;
           d = putc(bitnum,fout1);
        }
    fclose(f1); fclose(f2);
    fclose(f3); fclose(f4);
    fclose(fout1);
  }
}
/*_________________________ADDS.C________________________________*/
```

APPENDIX D

```
/*______________________________DCT8.C___________________________________*/

/*  A program which implements the Discrete Cosine Transform on a          */
/*  8 x 8 x 8 bit image.                                                   */
/*  This program was written in order to compare the results obtained with */
/*  the various FDCT algorithms.                                           */
/*  I H Mclean 18/8/87                                                     */

#include <stdio.h>
#include <fcntl.h>
#include <math.h>

void main()
{
  #define PIDIV16  0.19635   /* pi divided by 16 see def of DCT */
  #define TWODIVN  0.25      /* 2/N where N = 8. See def of DCT */

  int
   ret,       /* return code for fseek function  */

   j,         /*  The formula for the DCT on which this program is    */
   k,         /*  based uses j and k in its notation. I do the same   */
              /*  merely to avoid confusion.                          */

   dispmod,   /* parameter to a PCEYE function   */
   d;         /* result of IO      */

  float
     fj, Fk,          /* input & output bytes resp. See formula. */
     res,             /* result of raising 2 to a certain power  */
     Summation, ck;

   long
     count;

  extern int
  _fmode;

  char
    file1,
    outfile1; /* input and output files resp. */


  FILE  *f1, *fout1;     /* pointers to our input & output files resp. */

    {
     printf("NAME OF INPUT FILE : \n"); scanf("%s",file1);
     printf("NAME OF OUTPUT FILE  : \n"); scanf("%s",outfile1);
     _fmode = 0X8000;
     f1 = fopen(file1,"r");  fout1 = fopen(outfile1,"w+");
```

```
      for (count = 1; count <= 8; ++count)
       {
        for  (k = 0; k <= 7; ++k)    /* k as in F(k) the DCT */
          {
            Summation = 0;                           /* calc a new "pixel" */
            if (k == 0) {ck = 0.7071;} else {ck = 1;}
            for  (j = 0; j <= 7; ++j)                  /* j = 0,1,..., N-1 */
              {
                d = fscanf(f1,"%10f",&fj);    /* get value from input file */
                res = (2*j+1) * k * PIDIV16;
                Summation = Summation + (fj * cos(res));
              }
            Fk  = 0.25 * ck * Summation;  /* must round to 1 byte */
            fprintf(fout1,"%10f",Fk);
            if (k <= 6) {ret = fseek(f1,(-8*10),1);} /* use same 8 input bytes */
          }
       }
    fclose(f1); fclose(fout1);  /* close input & output files */
  }
}
/*___________________________DCT8.C___________________________________*/
```

APPENDIX E

```
/*_________________________ PRINTFILE.C_____________________________ */

/* This simple program prints out the pixels in a digital image - row for  */
/* row. The input image is assumed to have the dimensions 256 x 256.       */
/* Used for checking the accuracy of reconstructed images.                 */
/*  I Mclean 8/11/87                                                       */

#include <stdio.h>
#include <fcntl.h>
#include <math.h>

void main()
{
  int
   linecount,
   lineno,        /* line number */
   spaces,        /* spaces between printouts - for more than 1 histogram */
   d;             /* I/O result                                           */

  extern int _fmode;

  long
   pos,
   count;        /* loop counter - must be long, since > than maxint */

  char
   reply,        /* user input */
   wait,         /* user input - pause between images */
   inbyte,       /* input byte */
   file1;        /* input file */

 FILE  *f1;      /* pointer to input file */

   {
     lineno = 0;
     spaces = 85; /* printing the first row */
     printf("NAME OF INPUT FILE : \n");  scanf("%s",file1);
     _fmode = 0X8000;              /* using translated I/O */
     f1 = fopen(file1,"r");
     for  (count = 1; count <= 256; ++count)
     {
       d = fscanf(f1,"%c",&inbyte);
     }
     for  (count = 1; count <= 65536; ++count)  /* 256 x 256 */
     {
       d = fscanf(f1,"%c",&inbyte);
       pos = ftell(f1);
       if (inbyte < 0) {printf("pos = %d",pos); goto end;}
       if (spaces == 85) {printf("
                  ");;}
       if (spaces == 80) {printf("
             ");;}
```

```
         if (spaces == 75) {printf("
              ");}
         if (spaces == 70) {printf("
           ");}
         if (spaces == 65) {printf("
        ");}
         if (spaces == 60) {printf("
     ");}
         if (spaces == 55) {printf("
  ");}
         if (spaces == 50) {printf("
 ");}
         if (spaces == 45) {printf("                                               ");}

         if (spaces == 40) {printf("                                            ");}
         if (spaces == 35) {printf("                                         ");}
         if (spaces == 30) {printf("                                      ");}
         if (spaces == 25) {printf("                                   ");}
         if (spaces == 20) {printf("                                ");}
         if (spaces == 15) {printf("                             ");}
         if (spaces == 10) {printf("                          ");}
         if (spaces == 5) {printf("                        ");}
         if (spaces == 0) {printf("                     ");}
         printf("%d \n",inbyte);
         ++lineno;
         if (count < 13)
         {
          if (lineno%12 == 0) {scanf("%c",wait);}
         }
         else
          { if (lineno%24 == 0) {scanf("%c",wait);}}
         if (count%256 == 0)    /* end of a row */
         {
           for  (linecount = 1; linecount <= 15; ++linecount)  {printf("\n");}
           printf("DO ANOTHER ROW : Y/N \n"); reply = getch();
           spaces -= 5;
           lineno = 0;
           if (reply == 'n') {goto end;}          /*  print in next row */
         }
      }  /* for */
end:
   fclose(f1);
  }
}
/*___________________________PRINTFILE.C_________________________________ */
```

APPENDIX F

```
/*___________________________SHOWBLOCK.C___________________________*/

/* This program outputs any 8x8 block in a digital image. Using this    */
/* window the user may move around the file and check the details of    */
/* reconstructed images.                                                */
/* I H Mclean 19/10/87                                                  */

#include <stdio.h>
#include <fcntl.h>
#include <math.h>

void main()
{

#define NDIV2    4        /*  N/2 - See def                    */
#define NEXTROW 248       /* 256 - 8, the amount to move to next row */
#define TRUE    1         /* Booleans.                         */
#define FALSE   0         /* Booleans.                         */

  int
    tempval,
    row1, col1,
    ret, d;

  extern int
   _fmode;

  long
    temp, count, count1;

  char outbyte, file1;

  FILE  *f1, fout1;

     {  _fmode = 0X8000;
        printf("NAME OF INPUT FILE : \n"); scanf("%s",file1);
repeat: printf("ROW NO (-1 TO END) : \n"); scanf("%d",&row1);
        if (row1 == -1) {goto end;}
        printf("COL NO : \n"); scanf("%d",&col1);
        f1 = fopen(file1,"r");
        printf("row = %d ",row1); printf("col = %d \n",col1);
        temp = (row1*256)+col1+256;                     /* origin of block */
        printf("temp is %d \n",temp);                   /* print out pos   */
        ret = fseek(f1,temp,0);
        for (count1 = 0; count1 <= 7; ++count1)
         {                                   /* read in 8x8 file */
          for (count = 0; count <= 7; ++count)
           {
            d = fscanf(f1,"%c",&outbyte);
            tempval = (int)outbyte;
            printf(" %3d",tempval);
            if (count == 7) {ret = fseek(f1,NEXTROW,1); printf("\n"); }
           }
        }
        fclose(f1);
        goto repeat;
      }
end:
}  /* showblock */
/*______________________________SHOWBLOCK.C______________________________*/
```

APPENDIX G

```
/*______________________ PIXDIFF.C___________________________  */

/* This simple program outputs a histogram of the difference between       */
/* adjacent pixels in the input image. This gives an indication of the     */
/* correlation between pixels in a small area.                             */
/*  I Mclean 8/11/87                                                       */

#include <stdio.h>
#include <fcntl.h>
#include <math.h>

void main()
{

  int
   xlint, x2int,
   diff,         /* difference between ajacent pixels */
   dim,          /* dimension of input image */
   d;            /* I/O result                 */

  extern int _fmode;

  long
    pixcounter,  /* sum of all pixel values */
    count,
    col, row;    /* loop counter - must be long, since > than maxint */

  float
    hist[200];   /* hold the values of the pixel distribution        */

  char
    x1, x2,
    reply,       /* user input */
    wait,        /* user input - pause between images */

    inbyte,      /* read header */

    file1;       /* input file */

  FILE  *f1;     /* pointer to input file */

  {
start:
      printf("NAME OF INPUT FILE 1 : \n");  scanf("%s",file1);
      printf("IMAGE DIMENSION : \n");  scanf("%d",&dim);
      _fmode = 0X8000;              /* using translated I/O */
      f1 = fopen(file1,"r");
      for  (count = 1; count <= 256; ++count)
      {
        d = fscanf(f1,"%c",&inbyte);
      }
```

```
        for  (count = 1; count <= 200; ++count)
        {
          hist[count-1] = 0;            /* set array to zero */
        }
        for  (row = 1; row <= 256; ++row)   /* 256 x 256 - 1 */
        {
          d = fscanf(f1,"%c",&x1);
          x1int = (int)x1;
          if (x1int < 0) {x1int = x1int + 256;}  /* 8 bits 0 - 127 */
          x1int /= 3;
          for  (col = 1; col <= 255; ++col)   /* 256 -1 : already read */
            {
              d = fscanf(f1,"%c",&x2);
              x2int = (int)x2;
              if (x2int < 0) {x2int = x2int + 256;}
              x2int /= 3;
              diff = (x2int - x1int);
              ++hist[diff+100];        /* increment that element or "bin" */
              x1int = x2int;
            }
        }
        fclose(f1);
        pixcounter = 0;
        for  (count = 1; count <= 200; ++count)
        {
          printf("%d ",count-101);
          printf("%10.2f \n",hist[count-1]);     /* print it out */
          if (count%12 == 0) {scanf("%c",wait);}
        }
        printf("DO ANOTHER IMAGE : Y/N : \n"); reply = getch();
        if (reply == 'y') {goto start;}
    }
}
/*________________________HISTOGRAM______________________________ */
```

APPENDIX H

```
/*______________________________FDCT.C________________________________*/

#include <stdio.h>
#include <fcntl.h>
#include <math.h>

void main()
{

/* This program implements the Fast Discrete Cosine Transform developed */
/* by Chen, Smith & Fralick. This algorithm is AT LEAST 8 times faster  */
/* than the conventional DCT. This program forms the kernal of all the  */
/* DCT coding schemes proposed.                                         */
/* The FDCT is performed in one dimension ONLY, and each 8 x 8 block    */
/* thus be transposed in order to perform the 2-D transform.            */
/* The frequency domain data so generated is used by other progs        */
/* such as BITALLOC.C to provide statistics on the input image.         */
/* I H Mclean 19/10/87                                                  */

#define PIDIV16  0.19635  /* pi divided by 16 see def of DCT         */
#define PIDIV4   0.785398 /* pi divided by 16 see def of DCT         */
#define PIDIV8   0.392699 /* pi divided by 16 see def of DCT         */
#define TWODIVN  0.25     /* 2/N where N = 8. See def of DCT         */
#define NDIV2    4        /*  N/2 - See def                          */
#define NEXTROW 248       /* 256 - 8, the amount to move to next row */
#define TRUE     1        /* Booleans.                               */
#define FALSE    0        /* Booleans.                               */

  int
   ret,
   d;

  extern int
   _fmode;    /* translated IO flag */

  long
    temp,
    count,
    count1;

  register long
    row, col,
    row1, col1;    /* row and col numbers of input & output files */

  float
    eightbit,       /* each pixel is read as an eight bit number */
    mainarr[8][8]; /* and processed in 8x8 "blocks"            */

  char
   outbyte, outfile1[8], file1;

  FILE  *f1, *fout1;
```

```
        {
          _fmode = 0X8000;    /* translated I/O */
          printf("*******************WARNING**************************** \n");
          printf("* COMPUTATION IS IN PLACE! - USE A COPY OF INPUT FILE * \n");
          printf("*****************************************************  \n\n");
          printf("    NAME OF INPUT FILE : \n"); scanf("%s",file1);
          fout1 = fopen("bud","w");              /* use proper name - C forgets! */
          for (row1 = 0; row1 <= 248; row1+=8)
             {
               for (col1 = 0; col1 <= 248; col1+=8)
               {
                 f1 = fopen(file1,"r");
                 printf("row = %d ",row1); printf("col = %d \n",col1);
                 temp = (row1*256)+col1+256;    /* new file pos */
                 ret = fseek(f1,temp,0);
                 for (count1 = 0; count1 <= 7; ++count1)
                   {
                    for (count = 0; count <= 7; ++count)
                    {
                      d = fscanf(f1,"%c",&outbyte);
                      eightbit = (float)outbyte;
                      if (eightbit < 0) {eightbit += 256;}  /* eight bit goes -ve */
                      mainarr[count][count1] = eightbit;
                      if (count == 7) {ret = fseek(f1,NEXTROW,1);;}
                    }
                   }
                fclose(f1);
                fdct(mainarr);

/*___________________WRITE FILE AS 1-D REAL__________________________*/

               for (row = 0; row <= 7; ++row)
                {
                 for (col = 0; col <= 7; ++col)
                  {
                    fprintf(fout1,"%12f",mainarr[col][row]);
                  }
                }

          }
        }


  fclose(fout1);

/*______________CALL RELEVANT CODING ROUTINES________________________*/

/*
    actindex(fout1);
    bitalloc(outfile1);
*/

/*___________________________________________________________________*/

    fout1 = fopen("bud","r");
```

```
        for (rowl = 0; rowl <= 248; rowl+=8)
          {
         for (coll = 0; coll <= 248; coll+=8)
          {
            for (countl = 0; countl <= 7; ++countl)
              {                                       /*  read in 8x8 file  */
                for (count = 0; count <= 7; ++count)
                  {
                    d = fscanf(foutl,"%12f",&mainarr[count][countl]);
                  }
              }
          ifdct(mainarr);
          rep(filel,mainarr,coll,rowl);
         }
        }

      fclose(foutl);

}
}
/*__________INCLUDE CODE DEPENDING ON WHICH ALGORITHM IS USED___________*/

/*
#include<fdctadap2.c>
#include<fdctadap4.c>
#include<fdctadapl.c>
*/

/*____________________________________________________________________*/

rep(filel,insertarr,startcol,startrow)
char filel;
float insertarr[][8];
long  startcol,startrow;
{
  int
   ret;        /* return code for fseek function  */

 register int row, col;

  long
    startblock;  /* start pos of block to be inserted */

  float
    frac;        /* fractional part of the real coefficient */

  char
    outcoeff;

  FILE  *fl, *foutl;  /* pointers to our input & output files resp. */
    {
      fl = fopen("jez" /* filel */,"r+");
      startblock = (startrow*256)+startcol+256;   /* origin of block */
      ret = fseek(fl,startblock,0);
```

```
        for (row = 0; row <= 7; ++row)
         {                                          /* read in 8x8 file */
          for (col = 0; col <= 7; ++col)
           {
             outcoeff = (char)(insertarr[col][row]);
             frac = insertarr[col][row] - (int)(insertarr[col][row]);
             if (frac >= 0.5) {outcoeff += 1;}
             ret = fseek(f1,0,1);
             fprintf(f1,"%c",outcoeff);
         /*    printf(" %d",outcoeff);   */
             if (col == 7) {ret = fseek(f1,NEXTROW,1); /* printf("\n"); */ }
           } /* for */
         }  /* for */
       fclose(f1);   /* close input & output files */
     }
   }
   /*_________________________FDCT.C______________________________*/
```

APPENDIX I

```
/*__________________FDCTADP1.C_________________________________*/

/*____________________CHEN FDCT_________________________________*/

/* Data is passed as 8 x 8 blocks which are then transformed using the    */
/* Chen algorithm.                                                        */

fdct(fdctarr)
float fdctarr[][8];

{
 int
   first;     /* is it the first pass ? */

 register long
   coeff, coeff1, coeff3;

  float
    a[8], b[8], c[8],
    d[8], e[8], f[8]; /* hold the different stages of the comp.*/

/*________________ MAIN LOOP ________________________________*/

     { first = TRUE;  /* first pass ! */
start: for (coeff1 = 0; coeff1 <= 7; ++coeff1)
        {
           for (coeff = 0; coeff <= 7; ++coeff)
            {
              if (first == TRUE) { a[coeff] = fdctarr[coeff][coeff1]; }
              else {a[coeff] = fdctarr[coeff1][coeff];}
            }
/*________________ FIRST STAGE : ___________________________*/

            b[0] = a[0]+a[7]; b[1] = a[1]+a[6];  b[2] = a[2]+a[5];
            b[3] = a[3]+a[4]; b[4] = -a[4]+a[3];  b[5] = -a[5]+a[2];
            b[6] = -a[6]+a[1]; b[7] = -a[7]+a[0];

/*________________SECOND STAGE : ___________________________*/

            c[0] = b[0]+b[3];  c[1] = b[1]+b[2]; c[2] = -b[2]+b[1];
            c[3] = -b[3]+b[0]; c[4] = b[4];
            c[5] = -(cos(PIDIV4)*b[5]) + (cos(PIDIV4)*b[6]);
            c[6] =  (cos(PIDIV4)*b[6]) + (cos(PIDIV4)*b[5]);
            c[7] = b[7];

/*________________THIRD STAGE : ____________________________*/

            d[0] =  (cos(PIDIV4)*c[0]) + (cos(PIDIV4)*c[1]);
            d[1] =  -(cos(PIDIV4)*c[1]) + (cos(PIDIV4)*c[0]);
            d[2] =  (sin(PIDIV8)*c[2]) + (cos(PIDIV8)*c[3]);
            d[3] =  (cos(3*PIDIV8)*c[3]) - (sin(3*PIDIV8)*c[2]);
            d[4] =  c[4]+c[5]; d[5] = -c[5]+c[4]; d[6] = -c[6]+c[7];
            d[7] =  c[7]+c[6];
```

```
/*________________FOURTH STAGE : ________________________________*/

              e[0] = d[0];  e[1] = d[1];  e[2] = d[2]; e[3] = d[3];
              e[4] =  (sin(PIDIV16)*d[4]) + (cos(PIDIV16)*d[7]);
              e[5] =  (sin(5*PIDIV16)*d[5]) + (cos(5*PIDIV16)*d[6]);
              e[6] =  (cos(3*PIDIV16)*d[6]) - (sin(3*PIDIV16)*d[5]);
              e[7] =  (cos(7*PIDIV16)*d[7]) - (sin(7*PIDIV16)*d[4]);

/*________________FIFTH STAGE : _________________________________*/

              f[0] = TWODIVN*e[0];  f[1] = TWODIVN*e[4];
              f[2] = TWODIVN*e[2];  f[3] = TWODIVN*e[6];
              f[4] = TWODIVN*e[1];  f[5] = TWODIVN*e[5];
              f[6] = TWODIVN*e[3];  f[7] = TWODIVN*e[7];
              for  (coeff3 = 0; coeff3 <= 7; ++coeff3)
               { if (first == TRUE) {fdctarr[coeff3][coeff1] = f[coeff3];}
                 else {fdctarr[coeff1][coeff3] = f[coeff3];}
               }
          }
        if (first == TRUE)  {
                              first = FALSE;  /*  go into second pass */
                              goto start;
                            }

      }
}  /* FDCT */
/*________________CHEN IFDCT______________________________________*/

ifdct(ifdctarr)
float ifdctarr[][8];

{
 int
   first;      /* is it the first pass ? */

register  long  coeff, coeff1, coeff3;

  float
    a[8], b[8], c[8],
    d[8], e[8], f[8]; /* hold the different stages of the comp.*/

      {
        first = TRUE;  /* first pass ! */

/*________________ MAIN LOOP _____________________________________*/

start:  for (coeff1 = 0; coeff1 <= 7; ++coeff1)
          {
            for (coeff = 0; coeff <= 7; ++coeff)
             {
               if (first == TRUE) { a[coeff] = ifdctarr[coeff][coeff1]; }
               else {a[coeff] = ifdctarr[coeff1][coeff];}
             }
/*________________ FIRST STAGE : ______________________________*/

             b[4] = a[1]; b[5] = a[5]; b[6] = a[3]; b[7] = a[7];
```

```
/*_______________SECOND STAGE : ______________________________*/

            c[0] = a[0];  c[1] = a[4];  c[2] = a[2]; c[3] = a[6];
            c[4] =  (sin(PIDIV16)*b[4]) - (sin(7*PIDIV16)*b[7]);
            c[5] =  (sin(5*PIDIV16)*b[5]) - (sin(3*PIDIV16)*b[6]);
            c[6] =  (cos(3*PIDIV16)*b[6]) + (cos(5*PIDIV16)*b[5]);
            c[7] =  (cos(7*PIDIV16)*b[7]) + (cos(PIDIV16)*b[4]);

/*_______________THIRD STAGE : ______________________________*/

            d[0] =  (cos(PIDIV4)*c[0]) + (cos(PIDIV4)*c[1]);
            d[1] =  -(cos(PIDIV4)*c[1]) + (cos(PIDIV4)*c[0]);
            d[2] =  (sin(PIDIV8)*c[2]) - (sin(3*PIDIV8)*c[3]);
            d[3] =  (cos(3*PIDIV8)*c[3]) + (cos(PIDIV8)*c[2]);
            d[4] =  c[4]+c[5]; d[5] = -c[5]+c[4]; d[6] = -c[6]+c[7];
            d[7] =  c[7]+c[6];

/*_______________FOURTH STAGE : ______________________________*/

            e[0] = d[0]+d[3];  e[1] = d[1]+d[2]; e[2] = -d[2]+d[1];
            e[3] = -d[3]+d[0]; e[4] = d[4];
            e[5] = -(cos(PIDIV4)*d[5]) + (cos(PIDIV4)*d[6]);
            e[6] =  (cos(PIDIV4)*d[6]) + (cos(PIDIV4)*d[5]);
            e[7] = d[7];

/*_______________FIFTH STAGE : ______________________________*/

            f[0] = e[0]+e[7];  f[1] = e[1]+e[6];
            f[2] = e[2]+e[5];  f[3] = e[3]+e[4];
            f[4] = -e[4]+e[3]; f[5] = -e[5]+e[2];
            f[6] = -e[6]+e[1]; f[7] = -e[7]+e[0];
            for  (coeff3 = 0; coeff3 <= 7; ++coeff3)
             {
               if (first == TRUE) {ifdctarr[coeff3][coeff1] = f[coeff3];}
               else {ifdctarr[coeff1][coeff3] = f[coeff3];}
             }  /* for */
        }
      if (first == TRUE)  {
                             first = FALSE;  /*  go into second pass */
                             goto start;
                          } /* if */
    }
}    /* IFDCT */
/*____________________________________________________________*/
```

APPENDIX J

```
/*______________________FDCTADP2.C________________________________*/

actindex()
{

/* This procedure works out the activity index for each sub block.        */
/* A new file is opened, into which the classifications for each sub -     */
/* block are stored. This file is then referenced the bitalloc procedure */
/* to perform the adaptive coding.                                         */
  int
   ret,                  /* return code for fseek function         */
   classcount,           /* simple counter for members in a class */
   classno,              /* the clas numbering, ie 1,2,3 or 4      */
   low,                  /* flag for sorting algorithm             */
   filepos,              /* pos in the file                        */
   intactive,            /* must convert to integer for storage    */
   d,
   coeff;                /* coeff of the 8 x 8 block               */

 register int
   row;

  float
    activity,    /* activity index of each sub block          */
    energy;      /* each coefficient of the 8 x 8 sub block */


  FILE  *f1, *f2;  /* pointers to our input & output files resp. */
    {
      f1 = fopen("bud","r");
      f2 = fopen("actvfile","w");
      for (row = 1; row <= 1024; ++row)      /* 1024 */
       {
         activity = 0;
         d = fscanf(f1,"%12f",&energy);          /* skip DC coefficient */
         for (coeff = 2; coeff <= 64; ++coeff)
           {
             d = fscanf(f1,"%12f",&energy);
             energy = abs(energy);
             activity += energy;
           }  /* for  coeff */
         intactiv = (int)activity;
         fprintf(f2,"%4d",intactiv);
       } /* for row */
      fclose(f1); fclose(f2);  /* close input & output files */

/*______________________________________________________________________*/

/* Classify each 8 x 8 sub-block by looking at the "activity index" for    */
/* that block. Blocks are placed into 1 of 4 classes.                      */
      f2 = fopen("actvfile","r+");
      filepos = 0;                  /*  pos in file  */
      classno = -1;
      low = 0;                      /* we need to sort activity levels */
      classcount = 0;
loop:
        do
           {
             if (filepos == 1024) {filepos = 0; rewind(f2); ++low;}
             d = fscanf(f2,"%4d",&intactive);
             ++filepos;
           }
        while (intactive < 0);

        if (intactive == low)           /* then add it to class */
          {
            ++classcount;               /*  another member of class */
            ret = fseek(f2,-4,1);
            fprintf(f2,"%4d",classno);
            ret = fseek(f2,4,1);
            rewind(f2); filepos = 0;
          }
        if (classcount < 256) goto loop;

/*____________otherwise we have completed another class_____________*/

        if (classno > -4)
          {
            classcount = 0;
            classno -= 1;
            goto loop;
          }
     fclose(f2);
  }
}
/*_____________________FDCTADP2.C________________________________*/
```

APPENDIX K

```
/*_____________________________FDCTADP4.C________________________*/

/* The adaptive quantization and bit assignment are performed by this   */
/* suite of procedures.                                                 */

bitalloc()

{
/* This procedure implements  adaptive bit assignment for each          */
/* coefficient in the transform domain. The average value of each coeff. */
/* is displayed, as well as the variance. The bit assignment algorithm  */
/* is to be found in Pratt pg 673.                                      */
/* I H Mclean 3/11/87                                                   */

  int
    ret,            /* return code for fseek function */
    bitmatrix[64][4],
    totalbits[4];   /* total no of bits assigned so far */

  long int
   B,            /* the no of block code bits - supplied by user */
   N,            /* block size, eg 64, 256 etc                   */
   variance,     /* the variance at this stage                   */
    d;

  long
    class,       /* which class the 8x8 matrix belongs to */
    coeff;

  double
    result;       /* stores square root of variance */

  float
    temp,

    largedc[4],   /* largest DC coefficient for eac class */
    outreal,      /* final product - quantized - "dequantized" */
    realbyte,
    quanbyte,
    totalarr[64][4],
    vararr[64][4],  /* variances of the coefficients for each class  */
    sumlogvar[4],
    twolvar[4],
    twonlogvar[4],
    BdivN;          /* B divided by N.                            */

  register long rowl;

  char
     wait;

  FILE  *fl, *foutl;
  {
    for (class = 0; class <= 3; ++class)
     {
       for (coeff = 0; coeff <= 63; ++coeff)  /* zero the arrays */
       {
         totalarr[coeff][class] = 0;
         vararr[coeff][class] = 0;
       }
    }
    N = 64;
    B = 32;
    BdivN = 0.1;                        /* 0.5 bits pixel */
/*________________________________AVERAGES________________________________*/

    fout1 = fopen("bud","r");
    f1 = fopen("actvfile","r");  /* contains classification data */
    for (row1 = 1; row1 <= 1024; ++row1)
      {
        d = fscanf(f1,"%4d",&class);
        class = abs(class);
        --class;
          for (coeff = 1; coeff <= 64; ++coeff)
           {
             d = fscanf(fout1,"%12f",&realbyte);
             totalarr[coeff-1][class] += realbyte;
           }
     }
    fclose(fout1); fclose(f1);
    for (coeff = 1; coeff <= 64; ++coeff)
      {
        for (class = 0; class <= 3; ++class)
         {
          totalarr[coeff-1][class] = totalarr[coeff-1][class]/(float)256;
         }
      }
/*________________________________VARIANCES________________________________*/

    fout1 = fopen("bud","r");
    f1 = fopen("actvfile","r");  /* contains classification data */
    for (class = 0; class <= 3; ++class)
     {
       largedc[class] = 0;       /* got to find largest DC coeff */
     }
    for (row1 = 1; row1 <= 1024; ++row1)
      {
        d = fscanf(f1,"%4d",&class);
        class = abs(class);
        --class;
          for (coeff = 1; coeff <= 64; ++coeff)
           {
             d = fscanf(fout1,"%12f",&realbyte);
             if ((coeff == 1) && (realbyte > largedc[class]))
             {largedc[class] = realbyte;}
             variance = realbyte - totalarr[coeff-1][class];
             variance = variance*variance;
             vararr[coeff-1][class] += (float)variance;
           }
          }
        fclose(foutl); fclose(f1);
        for (class = 0; class <= 3; ++class)  {twonlogvar[class] = 0;}
        for (coeff = 1; coeff <= 64; ++coeff)
          {
            for (class = 0; class <= 3; ++class)
             {
               vararr[coeff-1][class] /= 256;
               sumlogvar[class] = log10((float)vararr[coeff-1][class]);
               twonlogvar[class] += (float)sumlogvar[class];
             }
          }
/*__________________BIT ALLOCATION________________________________*/

        for (class = 0; class <= 3; ++class)
          {
            twonlogvar[class] *= 2;
            twonlogvar[class] /= N;
          }
retry:
        for (class = 0; class <= 3; ++class)
        {totalbits[class] = 0;}   /* no bits assigned */
        for (coeff = 2; coeff <= 64; ++coeff)
          {
            for (class = 0; class <= 3; ++class)
             {
              twolvar[class] = log10((float)vararr[coeff-1][class]);
              temp = BdivN + (2*twolvar[class]) - twonlogvar[class];
              if (temp < 0) {temp = 0;}
              bitmatrix[coeff-1][class] = (int)temp;
              totalbits[class] += bitmatrix[coeff-1][class];
             }
          }
       for (class = 0; class <= 3; ++class)  /* dc coeff is sent uncoded! */
         {
           if (largedc[class] = 0) {bitmatrix[0][class] = 0;}
           if (largedc[class] > 0) {bitmatrix[0][class] = 1;}
           if (largedc[class] > 1) {bitmatrix[0][class] = 2;}
           if (largedc[class] > 3) {bitmatrix[0][class] = 3;}
           if (largedc[class] > 7) {bitmatrix[0][class] = 4;}
           if (largedc[class] > 15) {bitmatrix[0][class] = 5;}
           if (largedc[class] > 31) {bitmatrix[0][class] = 6;}
           if (largedc[class] > 63) {bitmatrix[0][class] = 7;}
           if (largedc[class] > 127) {bitmatrix[0][class] = 8;}
           if (largedc[class] > 255) {bitmatrix[0][class] = 9;}
           if (largedc[class] > 511) {bitmatrix[0][class] = 10;}
           totalbits[class] += bitmatrix[0][class];
         }

/* We must now check to see whether the right number of bits have been   */
/* assigned. If not, then increase BdivN and try again - until totalbits */
/* is greater than the desired BIT RATE. Once it is greater, it can be   */
/* reduced to the correct number.                                         */

  /*     if (totalbits < 32) {BdivN += 0.1; goto retry;}  */

/*_______________________________________________________________________*/
```

```
/*      for (class = 0; class <= 3; ++class)
         {
          for (coeff = 1; coeff <= 64; ++coeff)
           {
             printf("%d ",bitmatrix[coeff-1][class]);
             if (coeff%8 == 0) {printf("\n");}
           }
         }
*/

/*__________WAIT FOR KEYPRESS SO USER CAN SEE BIT MAPS_________________*/

  printf("waiting !!!!!"); scanf("%c",wait); scanf("%c",wait);
  for (class = 0; class <= 3; ++class)
      {
        printf("%d \n",totalbits[class]);
      }

/*   if (totalbits > 32)
       {
         pos = 63;
         while (totalbits > 32)
         {
           if (bitmatrix[pos] > 0) {--bitmatrix[pos]; --totalbits;}
           --pos;
           if (pos == 0) {pos = 63;}   dc - coeff is untouched
         }

      }
*/
    printf("\n \n");
    for (class = 0; class <= 3; ++class)
      {
       for (coeff = 1; coeff <= 64; ++coeff)
         {
           printf("%d ",bitmatrix[coeff-1][class]);
           if (coeff%8 == 0) {printf("\n");}
         }
      }

/*____________________QUANTIZE COEFFICENTS______________________________*/

      foutl = fopen("bud","r+");
      for (rowl = 1; rowl <= 1024; ++rowl)
        {
            for (coeff = 1; coeff <= 64; ++coeff)
             {
               d = fscanf(foutl,"%12f",&realbyte);

/*_______________________NORMALIZATION______________________________*/

/* Each coefficient is divided by its standard deviation before it is    */
/* quantized. This is done in order ot prevent "clipping", which occurs  */
/* when coefficent values "overflow" - ie the number of bits allocated   */
/* to a certain coefficent cannot represent its value.                   */
```

```
result = sqrt(vararr[coeff-1][class]);
if (coeff != 1) {realbyte /= result;}
if (bitmatrix[coeff-1][class] == 0)  {outreal = 0;}
if (bitmatrix[coeff-1][class] == 1)
 {
   if (realbyte < 0) {outreal = -0.7071;}
   else {outreal = 0.7071;}
 }

if (bitmatrix[coeff-1][class] == 2)
 {
   quanbyte = (int)(realbyte/1.0874);
   quanbyte +=2;
   if (quanbyte > 3) {quanbyte = 3;}
   if (realbyte < 0)
     {
       quanbyte -= 1;
       if (quanbyte < 0 ) {quanbyte = 0;}
     }
   outreal = (((quanbyte - 2)*2)+1) * 0.5437;
 }

if (bitmatrix[coeff-1][class] == 3)
 {
   quanbyte = (int)(realbyte/0.7309);
   quanbyte +=4;
   if (quanbyte > 7) {quanbyte = 7;}
   if (realbyte < 0)
     {
       quanbyte -= 1;
       if (quanbyte < 0 ) {quanbyte = 0;}
     }
   outreal = (((quanbyte - 4)*2)+1) * 0.36545;
 }
 if (bitmatrix[coeff-1][class] == 4)
 {
   quanbyte = (int)(realbyte/0.461);
   quanbyte +=8;
   if (quanbyte > 15) {quanbyte = 15;}
   if (realbyte < 0)
     {
       quanbyte -= 1;
       if (quanbyte < 0 ) {quanbyte = 0;}
     }
    outreal = (((quanbyte - 8)*2)+1) * 0.2305;
 }
 if (bitmatrix[coeff-1][class] == 5)
 {
   quanbyte = (int)(realbyte/0.28);
   quanbyte +=16;
   if (quanbyte > 31) {quanbyte = 31;}
   if (realbyte < 0)
     {
       quanbyte -= 1;
       if (quanbyte < 0 ) {quanbyte = 0;}
     }
    outreal = (((quanbyte - 16)*2)+1) * 0.14;
```

```
                }
               if (bitmatrix[coeff-1][class] == 6)
                {
                  quanbyte = (int)(realbyte/0.1657);
                  quanbyte +=32;
                  if (quanbyte > 63) {quanbyte = 63;}
                  if (realbyte < 0)
                    {
                      quanbyte -= 1;
                      if (quanbyte < 0 ) {quanbyte = 0;}
                    }
                  outreal = (((quanbyte - 32)*2)+1) * 0.08285;
                }

               if (bitmatrix[coeff-1][class] == 7)
                {
                  quanbyte = (int)(realbyte/0.0961);
                  quanbyte +=64;
                  if (quanbyte > 127) {quanbyte = 127;}
                  if (realbyte < 0)
                    {
                      quanbyte -= 1;
                      if (quanbyte < 0 ) {quanbyte = 0;}
                    }
                  outreal = (((quanbyte - 64)*2)+1) * 0.04805;
                }

               if (bitmatrix[coeff-1][class] == 8)
                {
                  quanbyte = (int)(realbyte/0.0549);
                  quanbyte +=128;
                  if (quanbyte > 255) {quanbyte = 255;}
                  if (realbyte < 0)
                    {
                      quanbyte -= 1;
                      if (quanbyte < 0 ) {quanbyte = 0;}
                    }
                 outreal = (((quanbyte - 128)*2)+1) * 0.02745;
                }

               if (coeff == 1)
                {
                  outreal = (int)realbyte;
                }

             if (coeff != 1) {outreal *= result;}
             ret = fseek(foutl,-12,1);
             fprintf(foutl,"%12f",outreal);
             ret = fseek(foutl,0,1);
           }  /* for coeff */
         }
      fclose(foutl);

    }
}
/*____________________FDCTADP4.C________________________________*/
```

APPENDIX L

```
/*_____________________________THRESHOLD SAMPLING________________________*/

/* Threshold sampling - only those coefficients above a certain threshold */
/* value are retained - the rest are set to zero.                         */

filter(filtarr)
float filtarr[][8];
{
  float
    tempval,      /* temporary storage for pixel [0][0] */
    threshpix;    /* pixel being processed */

  int
   col,row;

    {
      tempval = filtarr[0][0];
      for  (row = 0; row <= 7; ++row)     /* set temparr = 0 */
       {
         for  (col = 0; col <= 7; ++col)
           {
             threshpix = filtarr[row][col];
             if (abs(threshpix) < 2.0) {filtarr[row][col] = 0;}
           }  /* for */
       }  /* for */
       filtarr[0][0] = tempval;    /* retain dc coeff */
    }
  }
/*______________________________________________________________________*/
```

APPENDIX M

```
/*_______BIT ALLOCATION AND QUANTIZATION FOR ZONAL CODING SCHEME______*/

bitalloc()

{
/* These procedures implement a bit assignment for each coefficient in  */
/* the transform domain. The average value of each coeff. is displayed  */
/* as well as the variance. The bit assignment algorithm is to be found */
/* in Pratt pg 673.                                                     */
/* I H Mclean 3/11/87                                                   */

#define BLOCKS  1024

  int
    ret,            /* return code for fseek function   */
    bitmatrix[64],  /* bit assignment are stored here   */
    totalbits,      /* total no of bits assigned so far */
    pos;            /* position in the 8x8 block        */

  long int
   B,           /* the no of block code bits - supplied by user */
   N,           /* block size, eg 64, 256 etc                   */
   temp,
   variance,    /* the variance at this stage                   */
    d;

  long
    coeff;

  double
    result;       /* stores square root of variance */

  float
    largedc,      /* largest DC coefficient */
    outreal,      /* final product - quantized - "dequantized" */
    realbyte,     /* input byte must be converted to real      */
    quanbyte,
    varfloat,     /* holds the REAL value of variance          */
    totalarr[64], /* total of each coefficient                 */
    vararr[64],   /* varainces of the coefficients             */
    bitsarr[64],  /* holds the real value of the bits allocated*/
    sumlogvar,
    sumvars,
    twolvar,
    twonlogvar,
    BdivN;        /* B divided by N. See formula.              */

 register long rowl, coll;

  char
      wait,      /* wait for user input */
     outbyte, outfilel[8],
     filel;

 FILE  *f1, *foutl;
 {
   for (coeff = 0; coeff <= 63; ++coeff)  /* zero the arrays */
   {
     totalarr[coeff] = 0;
     vararr[coeff] = 0;
   }
   N = 64;
   B = 32;
   BdivN = 0.1;                          /* 0.5 bits pixel */
/*______________________________AVERAGES_____________________________*/

   fout1 = fopen("bud" /* outfile1  */,"r");
   for (row1 = 1; row1 <= 1024; ++row1)
     {
        for (coeff = 1; coeff <= 64; ++coeff)
         {
           d = fscanf(fout1,"%12f",&realbyte);
           totalarr[coeff-1] += realbyte;
         }
     }
   fclose(fout1);
   for (coeff = 1; coeff <= 64; ++coeff)
     {
       totalarr[coeff-1] = totalarr[coeff-1]/(float)1024;  /* 256 x 256 /8 */
     }
/*______________________________VARIANCES____________________________*/

   fout1 = fopen("bud" /* outfile1  */,"r");
   largedc = 0;                     /* got to find largest DC coeff */
   for (row1 = 1; row1 <= 1024; ++row1)
     {
        for (coeff = 1; coeff <= 64; ++coeff)
         {
           d = fscanf(fout1,"%12f",&realbyte);

/*____Find largest DC coefficient, since it is sent "as is"____________*/

           if ((coeff == 1) && (realbyte > largedc))
             {largedc = realbyte;}
           variance = realbyte - totalarr[coeff-1];
           variance = variance*variance;
           vararr[coeff-1] = vararr[coeff-1] + (float)variance;
         }
     }
   fclose(fout1);
   twonlogvar = 0;
   printf("variances are : \n");
   for (coeff = 1; coeff <= 64; ++coeff)
     {
       vararr[coeff-1] /= 1024;
       sumlogvar = log10((float)vararr[coeff-1]);
       twonlogvar += (float)sumlogvar;
     }
```

```
/*_________________BIT ALLOCATION______________________________*/

        twonlogvar *= 2;
        printf("log10 (twonlogvar) * 2 = %f \n",twonlogvar);
        twonlogvar /= N;
        printf("log10 (twonlogvar)/n = %f \n",twonlogvar);
retry:  totalbits = 0;    /* no bits assigned */
        for (coeff = 2; coeff <= 64; ++coeff)
          {
            twolvar = log10((float)vararr[coeff-1]);
            bitsarr[coeff-1] = BdivN + (2*twolvar) - twonlogvar;
            if (bitsarr[coeff-1] < 0) {bitsarr[coeff-1] = 0;}
            bitmatrix[coeff-1] = (int)bitsarr[coeff-1];
            totalbits += bitmatrix[coeff-1];
          }
        if (largedc > 63) {bitmatrix[0] = 7;}   /* dc coeff is sent uncoded! */
        if (largedc > 127) {bitmatrix[0] = 8;}
        if (largedc > 255) {bitmatrix[0] = 9;}
        if (largedc > 511) {bitmatrix[0] = 10;}
        totalbits += bitmatrix[0];


/* We must now check to see wether the right number of bits have been    */
/* assigned. If not, then increase BdivN and try again - until totalbits */
/* is greater than the desired BIT RATE. Once it is greater, it can be   */
/* reduced to the correct number                                         */


        if (totalbits < 32) {BdivN += 0.1; goto retry;}

/*______________________________________________________________________*/

        for (coeff = 1; coeff <= 64; ++coeff)
          {
            printf("%d ",bitmatrix[coeff-1]);
            if (coeff%8 == 0) {printf("\n");}
          }
       printf("%d \n",totalbits);
       if (totalbits > 32)
        {
          pos = 63;
          while (totalbits > 32)
          {
            if (bitmatrix[pos] > 0) {--bitmatrix[pos]; --totalbits;}
            --pos;
            if (pos == 0) {pos = 63;}  /* dc - coeff is untouched */
          }
        }
       printf("\n \n");
       for (coeff = 1; coeff <= 64; ++coeff)
         {
           printf("%d ",bitmatrix[coeff-1]);
           if (coeff%8 == 0) {printf("\n");}
         }
        printf("largedc = %f \n ",largedc);
```

```
/*__________________QUANTIZE COEFFICENTS______________________________*/

       foutl = fopen("bud","r+");
       for (rowl = 1; rowl <= 1024; ++rowl)
         {
             for (coeff = 1; coeff <= 64; ++coeff)
              {

                d = fscanf(foutl,"%12f",&realbyte);

/*______________________NORMALIZATION_____________________________*/
/* Each coefficient is divided by its standard deviation before it is     */
/* quantized. This is done in order ot prevent "clipping", which occurs   */
/* when coefficent values "overflow" - ie the number of bits allocated    */
/* to a certain coefficent cannot represent its value.                    */

                result = sqrt(vararr[coeff-1]);
                if (coeff != 1) {realbyte /= result;}
                if (bitmatrix[coeff-1] == 0)  {outreal = 0;}
                if (bitmatrix[coeff-1] == 1)
                 {
                   if (realbyte < 0) {outreal = -0.7071;}
                   else {outreal = 0.7071;}
                 }

                if (bitmatrix[coeff-1] == 2)
                 {
                   quanbyte = (int)(realbyte/1.0874);
                   quanbyte +=2;
                   if (quanbyte > 3) {quanbyte = 3;}
                   if (realbyte < 0)
                     {
                       quanbyte -= 1;
                       if (quanbyte < 0 ) {quanbyte = 0;}
                     }
                   outreal = (((quanbyte - 2)*2)+1) * 0.5437;
                 }

                if (bitmatrix[coeff-1] == 3)
                 {
                   quanbyte = (int)(realbyte/0.7309);
                   quanbyte +=4;
                   if (quanbyte > 7) {quanbyte = 7;}
                   if (realbyte < 0)
                     {
                       quanbyte -= 1;
                       if (quanbyte < 0 ) {quanbyte = 0;}
                     }
                   outreal = (((quanbyte - 4)*2)+1) * 0.36545;
                 }

                if (bitmatrix[coeff-1] == 4)
                 {
                   quanbyte = (int)(realbyte/0.461);
                   quanbyte +=8;
                   if (quanbyte > 15) {quanbyte = 15;}
```

```
    if (realbyte < 0)
      {
        quanbyte -= 1;
        if (quanbyte < 0 ) {quanbyte = 0;}
      }
     outreal = (((quanbyte - 8)*2)+1) * 0.2305;
 }

if (bitmatrix[coeff-1] == 5)
 {
   quanbyte = (int)(realbyte/0.28);
   quanbyte +=16;
   if (quanbyte > 31) {quanbyte = 31;}
   if (realbyte < 0)
     {
       quanbyte -= 1;
       if (quanbyte < 0 ) {quanbyte = 0;}
     }
    outreal = (((quanbyte - 16)*2)+1) * 0.14;
 }

if (bitmatrix[coeff-1] == 6)
 {
   quanbyte = (int)(realbyte/0.1657);
   quanbyte +=32;
   if (quanbyte > 63) {quanbyte = 63;}
   if (realbyte < 0)
     {
       quanbyte -= 1;
       if (quanbyte < 0 ) {quanbyte = 0;}
     }
   outreal = (((quanbyte - 32)*2)+1) * 0.08285;
 }

if (bitmatrix[coeff-1] == 7)
 {
   quanbyte = (int)(realbyte/0.0961);
   quanbyte +=64;
   if (quanbyte > 127) {quanbyte = 127;}
   if (realbyte < 0)
     {
       quanbyte -= 1;
       if (quanbyte < 0 ) {quanbyte = 0;}
     }
   outreal = (((quanbyte - 64)*2)+1) * 0.04805;
 }

if (bitmatrix[coeff-1] == 8)
 {
   quanbyte = (int)(realbyte/0.0549);
   quanbyte +=128;
   if (quanbyte > 255) {quanbyte = 255;}
   if (realbyte < 0)
     {
       quanbyte -= 1;
       if (quanbyte < 0 ) {quanbyte = 0;}
     }
  outreal = (((quanbyte - 128)*2)+1) * 0.02745;
```

```
                }

                if (coeff == 1)
                {
                  outreal = (int)realbyte;
                }
              if (coeff != 1) {outreal *= result;}
              ret = fseek(fout1,-12,1);
              fprintf(fout1,"%12f",outreal);
              ret = fseek(fout1,0,1);

          }   /* for coeff */
        }
      fclose(fout1);
    }
}
/*____________________BITALLOC.C__________________________________*/
```

APPENDIX N

```
/*________________________GNOISE.C_______________________________*/

/* A simple program which generates a 256 x 256 block of uncorrelated    */
/* Gaussian random variables - noise. This noise may then be increased   */
/* or decreased in power and added to any image of the same size.        */
/* The algorithm to convert independant, uniform random numbers to       */
/* uncorrelated Gaussian random variables appears in ---- , pg571.       */
/* I H Mclean 19/8/87                                                    */

#include <stdio.h>
#include <fcntl.h>
#include <math.h>
#include <limits.h>

void main()
{
  int
   d;         /* result of IO     */

  long
    count;   /* loop counter - must be long since > maxint */

  double
       w, w1, temp,          /* see formula for generating  */
       x, x1, r, y;          /* Gaussian noise.             */

  extern int
  _fmode;                    /* translated IO flag          */

  char  in1, in2,            /* two input bytes             */
        outbyte, outbyte1,
        outfile1;            /* output file                 */

  FILE  *f1,*fout1;          /* pointer to our output file. */

    {
      printf("NAME OF OUTPUT FILE  : \n"); scanf("%s",outfile1);
      _fmode = 0X8000;
      fout1 = fopen(outfile1,"w+");
      for  (count = 1; count <= 256; ++count)   /* copy the file header ! */
      {
       d = fscanf(f1,"%c",&outbyte);
       d = putc(outbyte,fout1);
       }
      x = drand48();
      for  (count = 1; count <= 16384; ++count) /* generate noise  */
      {
       x1 = drand48();
       r = log(1/x);
       y = sqrt(20*r);
       temp = (6.283*x1);
       w = y*cos(temp);
       w1 = y*sin(temp);
```

```
        outbyte = (char)(w);
        outbytel = (char)(wl);
        x = xl;
        d = putc(outbyte,foutl);
        d = putc(outbytel,foutl);
       }
      fclose(foutl);   /* close input & output files  */
     }
   }
/*________________________GNOISE.C__________________________________*/
```

APPENDIX O

```
/*____________________________NOISE.C______________________________________*/

/* A simple program which simulates a binary symmetric channel. The error */
/* rate of the channel is fixed by the user. In a binary symmetric channel*/
/* the probability of receiving an incorrect symbol is given by p for the */
/* transmission of 1's or 0's.                                            */
/* I H Mclean 19/8/87                                                     */

#include <stdio.h>
#include <fcntl.h>
#include <math.h>
#include <limits.h>

void main()
{
  int
   rate,      /* error rate  */
   bit,
   bytes,
   filecount,
   bitfield,
   tempint,
   d;         /* result of IO  */

  long
    row,          /* no of image rows */
    bytecounter, /* counts unitl no of bytes in file have been read */
    count;        /* loop counter - must be long since > maxint */

  double
       randomno;

  extern int
  _fmode;

  char
        outbyte, outbytel,
        outfilel;  /* output file. */

 FILE  *f1,*foutl;     /* pointer to our output file. */

   {
     printf("WARNING !! - OUTPUT FILE MUST HAVE ZERO MEAN - USE STATS !\n");
     printf("NAME OF OUTPUT FILE  : \n"); scanf("%s",outfilel);
     printf("ERROR RATE : \n"); scanf("%d",rate);
     _fmode = 0X8000;
     foutl = fopen(outfilel,"w+");
     bytecounter = 0;
     row = 0;
     for  (count = 1; count <= 256; ++count)   /* copy the file header ! */
     {
      d = putc(0,foutl);
     }
     while (bytecounter < 65536)
       {
         randomno = drand48();
         randomno *= rate;             /* fix the error rate */
         bytes = (int)randomno/8;
         bit = (int)randomno - 8*bytes;
         bitfield = 1;                     /* 00000001 */
         bitfield = bitfield << bit;
         filecount = 1;
         while (filecount <= bytes)
          {
            ++bytecounter;
            if (bytecounter == 65536) {goto end;}
            ++filecount;
            d = putc(0,foutl);
          }

         tempint = 0;
         tempint ^= bitfield;
         outbyte = (char)tempint;
         d = putc(outbyte,foutl);
      end:
      }
    fclose(foutl);  /* close input & output files  */
   }
 }
/*______________________NOISE.C_________________________________*/
```

APPENDIX P

```
/*__________________________THOLD.C________________________________*/

/* This program calculates the percentage of coefficients in the        */
/* frequency domain which are less than a certain threshold value.      */
/* Useful in determining the optimum threshold values in DCT-THRESHOLD  */
/* CODING.                                                              */
/*  I H Mclean 18/8/87                                                  */

#include <stdio.h>
#include <fcntl.h>
#include <math.h>

void main()
{
  int
   size,   /* dimension of input image */
   d;      /* result of IO            */

  long
    below1, below2, below3, below4, below5,  /* coefficient counters */
    count;   /* loop counter - must be long since > maxint */

   float
    inbyte,                         /* read from 1 d real file    */
    per1, per2, per3, per4, per5;  /* % coefficient below thresh */

  extern int
  _fmode;     /* translated I/O  */

  char
    file1;

  FILE  *f1;     /* pointers to our input & output files resp. */

    {
      size = 65536;
      below1 = 0;
      below2 = 0;
      below3 = 0;
      below4 = 0;
      below5 = 0;
      printf("NAME OF INPUT FILE : \n"); scanf("%s",file1);
      _fmode = 0X8000;               /* translated IO */
      f1 = fopen(file1,"r");
      for (count = 1; count <= size; ++count)
      {
        d = fscanf(f1,"%12f",&inbyte);
        if (abs(inbyte) < 1) {++below1;}
        if (abs(inbyte) < 2) {++below2;}
        if (abs(inbyte) < 3) {++below3;}
        if (abs(inbyte) < 4) {++below4;}
        if (abs(inbyte) < 5) {++below5;}
      }
    fclose(fl);   /* close input file */
    printf("belowl = %f \n",belowl);
    printf("below2 = %f \n",below2);
    printf("below3 = %f \n",below3);
    printf("below4 = %f \n",below4);
    printf("below5 = %f \n",below5);
    perl = (float)((float)belowl/65536 )*100;      /* (int)size */
    per2 = (float)((float)below2/65536 )*100;
    per3 = (float)((float)below3/65536 )*100;
    per4 = (float)((float)below4/65536 )*100;
    per5 = (float)((float)below5/65536 )*100;

    printf(" percent coefficient below threshold 1 is %f \n",perl);
    printf(" percent coefficient below threshold 2 is %f \n",per2);
    printf(" percent coefficient below threshold 3 is %f \n",per3);
    printf(" percent coefficient below threshold 4 is %f \n",per4);
    printf(" percent coefficient below threshold 5 is %f \n",per5);
  }
}  /* THOLD */
/*_________________________THOLD.C_________________________________*/
```

APPENDIX Q

```
/*______________________________NMSE.C________________________________*/

/* This program calculates the N.M.S.E. (normalized mean square error)    */
/* between an original image and its reconstruction.                      */
/*  I Mclean 3/2/87                                                       */

#include <stdio.h>
#include <fcntl.h>
#include <math.h>
void main()
{

#define ONEDIVM2  65536   /* 1/M**2  see formula */

  int
    d;   /* IO return */

  extern int _fmode;     /* translated IO flag */

  long
    count;             /* has to be long, since > than maxint */

  float
    x,y,

    fjk2,              /* f(j,k)**2  */
    p1real, p2real,    /* for values > 127, since 8 bits goes -ve */
    diff,
    totdiff;           /* sum of differences between pixels */

  char  p1, outfile1, file2, p2,  file1;

  FILE  *fout1, *f2, *f1;

   {
     printf("NAME OF INPUT FILE 1 : \n"); scanf("%s",file1);
     printf("NAME OF INPUT FILE 2 : \n"); scanf("%s",file2);
     _fmode = 0X8000;
     totdiff = 0;
     fjk2 = 0;
     f1 = fopen(file1,"r");
     f2 = fopen(file2 ,"r");
     for  (count = 1; count <= 256; ++count)     /* scan over headers */
     {
       d = fscanf(f1,"%c",&p1);
       d = fscanf(f2,"%c",&p2);
     }
     for  (count = 1; count <= 65536; ++count)  /* assumed 256 x 256 */
     {
       d = fscanf(f1,"%c",&p1);
       p1real = (float)p1;
       if (p1 < 0) {p1real += 256;}   /* if p1 > 127 it goes -ve */
       fjk2 += (p1real*p1real);
```

```
          d = fscanf(f2,"%c",&p2);
          p2real = (float)p2;
          if (p2 < 0) {p2real += 256;}     /* goes -ve if > than 127 */
          diff = (p1 - p2);
          if ((p1 < 0) && (p2 > 0))  {diff = p1real - p2real;}
          if ((p1 > 0) && (p2 < 0))
           {
             if (p1 < 50) {diff = p1-p2;}
             if (p1 >= 50) {diff = p1real - p2real;}
           }
         if (diff > 15)        /* check for large errors in reconstruction */
         { printf("p1 = %d ",p1); printf("p2 = %d ",p2);
           printf("p1real = %f ",p1real); printf("p2real = %f \n",p2real);
         }
        totdiff += diff*diff;
        }
        fclose(f1); fclose(f2);
        printf("average diff = %f \n",totdiff/65536);
        printf("totdiff = %f \n\n",totdiff);
        printf(" fjk2 = %f \n",fjk2);
        fjk2 /= ONEDIVM2;
        totdiff /= ONEDIVM2;          /* 1/M**2 */
        totdiff /= fjk2;
        printf("  _______________________\n");
        printf(" /  % NMSE = %f \n",totdiff*100);
        printf(" _______________________");
  }
}
/*_________________________NMSE.C_________________________________*/
```