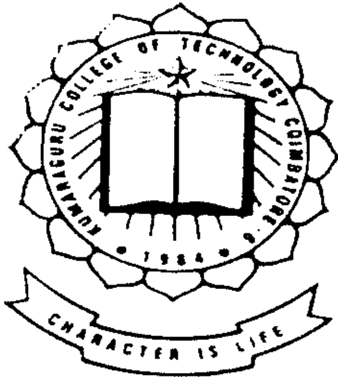
P-512

DIGITAL IMAGE COMPRESSION



1997 - 2001

Project work

Submitted By

Kumaravel J
Pragnesh K R V
Sathish Karthik R
Venkat Rama Rao A

Under the guidance of

Dr.S.Thangasamy Ph.D
Head of the Department
Computer Science and Engineering

In partial fulfillment of the requirements for the award of degree of Bachelor of Engineering (B.E) in Computer Science and Engineering of the Bharathiyar University, Coimbatore.

Department of Computer Science and Engineering
Kumaraguru College of Technology
Coimbatore - 641 006

Kumaraguru College of Technology

Coimbatore - 641 006

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Project Report entitled

Digital Image Compression

has been submitted by

Mr/Ms. KUMARAVEL J, PRAGNESH KRV, SATHISH KARTHIK K, VENKAT RAMA RAO A

In partial fulfillment of the requirements for the award of degree of Bachelor of Engineering (B.E) in Computer Science and Engineering of the Bharathiyar University, Coimbatore during the year 2000-2001

KCT LIBRARY Coimbatore 641 006

S. Jl 10/3/01
Guide

S. Jl 15/3/01
Head of the Department

Certified that the candidate was examined by us in the Project Viva-Voce examination held on 12.03.2001 and the University Register Number is 9727K0152, 9727K0159, 9727K0175, 9727K0520

Internal Examiner

External Examiner

Declaration

We,Kumaravel J, Pragnesh K R V, Sathish Karthik R,Venkat Rama Rao A ,hereby declare that this project work entitled "Digital Image Compression" submitted to Kumaraguru College of Technology , Coimbatore (Affiliated to Bharathiyar University) is a record of original work done by us under the supervision and guidance of Dr.S.Thangasamy Ph.D, Head of the Department, Department of Computer Science and Engineering.

Name of the candidate	Register Number	Signature of the candidate
Kumaravel J	9727K0152	J Kumaravel
Pragnesh K R V	9727K0159	[signature]
Sathish Karthik R	9727K0175	R. Sathish Karthik
Venkat Rama Rao A	9727K0520	A. Venkat Rama

Countersigned :

Staff in Charge :

S. Thangasamy 10/3/'01

Dr.S.Thangasamy Ph.D

Head of Department

Department of Computer Science and Engineering

Kumaraguru College of Technology

Coimbatore - 641 006

Place :

Date :

Acknowledgement

We deem this a rare opportunity to express our sincere thanks and deep gratitude to all those who were responsible for the knowledge and experience we acquired during our project as well as our course.

We would like to thank ***Dr.K.K.Padmanaban Ph.D*** *principal, Kumaraguru college of Technology for his kind co-operation and encouragement.*

We would also like to thank ***Dr.S.Thangasamy Ph.D****, Head of the Computer science and Engineering Department for being our guide.*

Our heartfelt thanks to ***staff members of Computer Science Department*** *, Kumaraguru college of Technology for having provided us the necessary infrastructure and for also having reminded our deadlines.*

Respectful thanks to our family members for their continued encouragement and support during the project.

Our special thanks to ***Mr.Raja Vaidyanathan ASIC Engineer Texas Instruments*** *, for his technical advice and also the reason behind the project's implementation in Linux.*

Last but not the least , we would like to thank all those who were involved in this project indirectly

CONTENTS

SYNOPSIS 1

1) INTRODUCTION

1.1) Existing system 4

1.2) Need for computerization 4

1.3) Proposed system 5

2) REQUIREMENT ANALYSIS

2.1) Hardware Requirements 8

2.2) Software Requirements 8

3) HISTORICAL PERSPECTIVE 11

4) NEED FOR COMPRESSION...12

4.1) How can images can be compressed.................................12

4.2) Compression standards..13

4.3) Lossy & Lossless Compression...15

5) GENERAL BLOCK DIAGRAM 18

6) Why DCT ?..20

7) Why Entropy encoding..21

8) PHASES OF COMPRESSION 22
8.1) Discrete Cosine Transform 22
8.2) Quantization 25
8.3) Zig Zag scan 27
8.4) Differential Pulse Code Modulation 27
8.5) Run Length Encoding 27
8.6) Entropy Encoding 28
8.7) Implementation of DCT based coding 29

9) PHASES OF DECOMPRESSION 31
9.1) Decoding 31
9.2) Dequantization 31
9.3) Inverse DCT 32

10) APPLICATIONS OF IMAGE COMPRESSION..................33

11) FUTURE TRENDS AND CONCLUSION 35

12) BIBLIOGRAPHY 37

13) SOURCE CODE 39

14) SAMPLE OUTPUTS 85

15) TEST RESULTS 92

SYNOPSIS

This project titled "DIGITAL IMAGE COMPRESSION" compresses images using the JPEG format.

This project deals with a Fourier transform called Discrete Cosine Transform which converts images from a spectral framework to a spatial framework. Once the spatial representation is obtained , the high frequency values (or the irrelevant information) can be eliminated under the pretext that they are not visible to the human eye. The remaining redundant values can be encoded to give the compressed image.

The coding is done in C using LINUX; one of the most powerful operating systems of its kind .

The synchronized and the simultaneous display of two images on a single display unit, so that the variation in quality of the decompressed image can be easily compared with that of the original image , and conversion of a 256 color image into a gray image of photo quality are some of the features offered by this project.

The ability to transfer images across the network is achieved using streaming sockets.

1) INTRODUCTION

1.1) Existing System

The information highway, digital television, teleconferencing, videophones and telemedicine are a few of the facets of the emerging digital age. Developments in digital image processing have aided their progression from design concepts to everyday experience . Digital image compression methods have assisted this progression .

Digital imaging is now prevalent in many areas . The storage and transmission requirements of digital images are formidable . Image compression techniques seek to maximize transfer speeds and efficiently use archive space while retaining sufficient quality of image.

1.2) Need for Computerization

Compression is the art of significantly reducing the physical size of a block of information offering the following capabilities such as .

1) Storing more information on the same media.
2) Reducing the transfer time of data on a network.
3) Improving the usage and reproduction of image with minimum degradation.

This project examines compression techniques for still images .

Digital images contain a great deal of data. Electronic transmission of uncompressed digital images can be very time-consuming. Sending a 10 Mb image of a surface area of just 10mmx10mm would take approximately 45 minutes via a conventional phone line and a modem operating at 28,800 bits per second. Image compression techniques can reduce the size (and hence cost) of any archive and can decrease transfer times.

Image compression operations reduce the data content of a digital image and represent the image in a more compact form, usually just before storage or transmission . Gray scale, color or binary images can be compressed and different types of compression may be used for different applications such as in medical imaging, finger-printing/security, seismology and astronomy.

1.3) Proposed System

There are two types of image compression - lossy and lossless. Lossy compression results in the decompressed image being similar but not the same as the original image. This is because some of the original data has been discarded and/or changed. Lossless compression, however, retains the exact data of the original image bit for bit. The compression ratio can be defined as the ratio between the data content to be compressed and the data that results after decompression . Lossy techniques can provide compression ratios of up to 100:1 and beyond. Lossless compression ratios are much lower, however, achieving

rates of approximately 3:1 . In general as the lossy compression ratio increases so does the degradation of the image.

Image compression is usually a two-way process involving compression and decompression. This process may not be symmetrical, i.e. the time taken and/or computing power for one process may differ from the other given the type of compression algorithm used.

Many techniques are available to compress images, namely Fractals, Wavelets, and many transforms such as Fast Fourier transforms , Haar transform and the Hilbert transform. The most popular among these are the Fractals, the Wavelets and the FFTs.

DCT is prevalent in the field of image compression. The technique used to compress image depends upon the environment and the quantity of data that will be processed. The application which we have designed deals with only images of around 100K and finds application in Local Area Networks.

Though wavelets and fractals provide a higher compression ratio they are slow and they are ideal only for images of larger size say 18MB.This is the reason behind the selection of DCT(Discrete Cosine Transform).

The proposed system is developed in Linux since much of image processing is not done in this OS. The field of image processing can be powerfully implemented in Linux by embedding the powerful OS concepts.

2) REQUIREMENT ANALYSIS

2.1) Hardware Requirements

1. Linux Server
2. 2 PC's connected to the Network

2.2) Software Requirements

1. Red Hat Linux Version 6.2

About Linux

Linux is an operating system for PC computers that use 386, 486 or Pentium processors such as IBM compatibles. Linux was developed in the early 1990s by Linus Torvald along with other programmers around the world. As an operating system it performs the same function as that of DOS or Windows. However , Linux is distinguished by its power and flexibility.

Its development paralleled the entire communication revolution over the past several decades .Linux ,like all versions of Unix adds two more features .Linux is a multi-user and multitasking system.

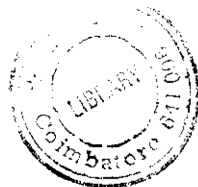
Linux can be divided into four major components : the kernel , the shell ,the file structure and the utilities .The kernel is the core program that runs programs and manages hardware devices such as disks and printers. The shell provides an interface for the user. It receives commands from the user and sends those commands to the kernel for

execution. The File structure organizes the way files are stored on storage device such as disk. Files are organized into directories. Each directory may contain any number of sub-directories , each holding files.

Together , the kernel , the shell, and the file structure form the basic the basic OS structure . With these three one can run programs, manage files and interact with the system. In addition Linux has software programs called utilities that have come to be considered standard features of the system. The utilities are specialized programs , such as editors , compilers , and communication programs , that perform standard computing operations.

The reason we chose Linux for image compression is that there isn't any necessity for interrupts to set the 320x200 , 256 color mode or even any other mode. It is even flexible for images that have 32 bits/pixel .

Concepts such as semaphores and mutual exclusions can be implemented in the C available for Linux. File transfer across the network can be easily implemented through socket programming with relatively fewer lines of code for which in DOS requires huge lines of codes with too many interrupts. Interrupts result in task switching and that is not a desired feature in an efficient program . Several Interrupts in DOS are replaced by simple library functions. This reduces the burden on the processor.


P-542

About C

A Pseudo code "program" often leads to hazy or incomplete definition full of lines like "PROCESS FILE UNTIL OUT OF DATA ". The result is that Pseudo code is easy to read but not so easy to translate into a working program .

If pseudo code is unsatisfactory the next best choice is to use a conventional programming language. Though hundreds of choices are available , C seems the best choice for this type of work for several good reasons.

First in many respects C has become ***lingua franca*** of programmers . That C compilers support computers ranging from a lowly 8051 micro controller to super computers capable of 100 MIPS has had much to do with this.

A second reason for using C is that a few constructs it uses as basic language elements are easily translated to other languages . So a data compression program illustrated in C can be converted to a working PASCAL program through a relatively straightforward translation procedure. Even assembly language programmers should find the process relatively painless.

Perhaps the most important reason for using C is simply one of efficiency.

3) Historical Perspective

The first notable research on image compression was carried out in the 1950's by Harrison and Huffman who specialized in Predictive Coding.

The 1960's saw the emergence of transform coding which dominated research for all of the seventies and the early eighties. From the early sixties to present lunar scientists have also played an important role as drivers of image compression development and its use.

These are based on mathematical processes that were formulated in the nineteenth and early part of this century respectively.

Recent research has also centered on a psycho-visual approach which aims to develop perceptually-tuned image compression systems . This approach was previously popular in the sixties .

4) NEED FOR COMPRESSION

Image compression is necessitated by

- A need to store data efficiently in available memory (ex cameras)
- A need to transmit data efficiently over a wide communication channel

4.1) How can images be compressed ?

Images can be compressed by two ways

1) Redundancy :

   There are certain parts of images that are repeated very often that can be very well coded .

2) Irrelevancy :

   This technique can be used in **continuous tone images** where variation in intensities of adjacent pixels cannot be identified by the observer . These pixels are said to be irrelevant .

4.2) Compression Standards

(a) JPEG

The Joint Photographic Experts Group standard is intended for compression of color or gray-scale images of natural real-world scenes. It is usually used in lossy mode.

It uses several cascaded compression modes. Primarily, an image is transformed to the frequency domain using the Discrete Cosine Transform. The resulting smaller-valued frequency components are rejected, leaving behind the larger-valued components. These are then Differential Pulse Code Modulation coded and then Huffman coded.

The adjustable nature of JPEG compression allows for variable compression ratios and fine-tuning the algorithm for a particular application's requirements.

(b) MPEG

This standard for motion video and audio is a successor to the H.261 which was based on DCT and Huffman coding. The MPEG compression standard also uses DCT and Huffman coding methods but in conjunction with inter-frame coding techniques which are utilized to give better compression ratios than in still image schemes. MPEG-1 and MPEG-2 are intended for low-resolution image sequences and higher-resolution sequences respectively.

The Motion Picture Experts Group recently met in Dublin to complete work on the new MPEG-4 standard and to initiate specification of the future MPEG-7 standard. The MPEG-4 is a departure from the previous standards in that it concentrates on unified audio-visual objects and scenes rather than frames. MPEG-7 is intended to aid the location of audio-visual content rather like the way text-based search engines operate.

4.3) Lossless and Lossy Image Compression

In various digital image applications that require image compression, some of them do not allow error in reconstructed image, and thus lossless compression techniques are used. For examples, medical imaging prefers lossless image compression, since compressing digital radiographs with lossy techniques may divert diagnosis accuracy. Applications such as astronomical imaging also require lossless compression, since the image is going to be processed by computing devices instead of human eyes. But as in lossless digital data compression, there is a trade-off of compression ratio for the error-free reconstruction. Nevertheless, the above applications show the great need of efficient lossless image compressions technique. If this loss restriction does not exist, lossy image compression can be used. Similar to the case of lossless image compression, lossy image compression also has many different kinds of application area. Some examples can be found even in daily life, such as digital camera and personal World Wide Web (WWW) publications. These applications often require less image fidelity in reconstructed image. Thus, they can afford some loss of information during the coding process. In addition, most of them require that the reconstructed images are recognized only by human beings, permitting plenty of visual characteristics that can be exploited. Therefore, due to the human-based nature of lossy image compression technique, its compression performance usually outperforms that of lossless compression. This is because it can have more factors that can be exploited to achieve higher compression performance, in terms of both

compression ratio and visual quality of reconstructed image. The theoretical possibility of image compression, no matter lossless or lossy , is primarily based on the redundancy inside image. The redundancy is due to the correlation between neighbouring pixels that, their physical values are expected to be near. This assumption is true in most of the part of image, where occasional exceptions only occur at major edges of objects in image. Based on this assumption, prediction models can be constructed easily for correlating pixels. In other words, if one pixel is given accurately, its surrounding pixels are nearly known. This correlation points out that there is little useful information in pixel values of an image. By representing only this useful information, we should be able to compress the original image to a fraction of its size. In addition to this redundancy in image, lossy image compression can also utilize the characteristics of Human Visual System(HVS) as mentioned above. These characteristics actually divided information in image as two types: relevant to HVS and irrelevant to HVS. By excluding information irrelevant to HVS, a better compression performance can be expected. Some HVS characteristics such as spatial and spectral redundancies help predicts what information among pixels are relevant to our vision. As a consequence, an image compression algorithm should have considerations concerning the above arguments. In order to be efficient enough for practical applications, a good image compression algorithm should at least possess the following properties.

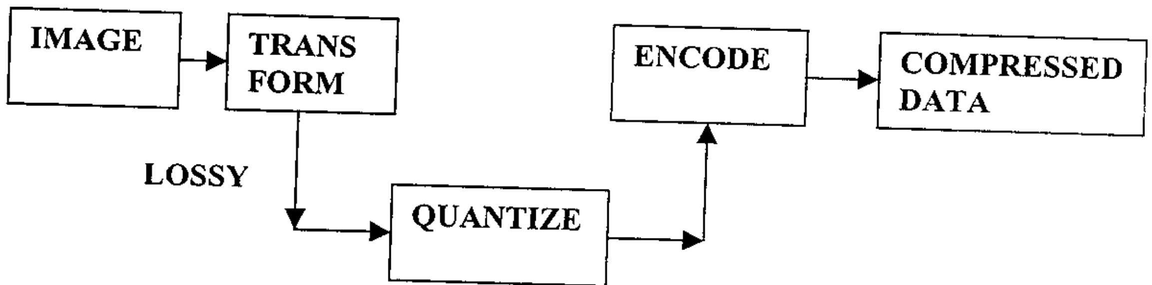
a) ***Good prediction model.*** The incoming pixel values are actually events with different probability of occurrence. A good predictor can predict with high probability a required pixel by using its surrounding. Therefore, by coding what are required by the predictor during prediction instead of coding the original pixels can make use of this high probability distribution. This distribution is usually easier to be compressed by statistical coder than the original distribution does.

b) ***Adequate statistical coding.*** Huffman or arithmetic coding are some good coding techniques that can help us to get near-optimal/optimal average code length

c) ***Utilize two spatial dimensions.*** Since image is two-dimensional, so the context is more complicated than that of one-dimensional signal . If two-dimensional considerations are taken into account in the coding techniques , the coding efficiency should be expected to increase.

In modern image compression formats, these properties are often present. Different techniques approach these properties in different ways. In the following sections, we are going to study some modern compression techniques and formats.

5) GENERAL BLOCK DIAGRAM :



a) TRANSFORM THE IMAGE

b) QUANTIZE THE IMAGE

c) ENCODE THE IMAGE

Step 1: Transform the image into a set of basis functions

- A transformation is a process which converts a set of numbers into a new set of numbers.
- This is more amenable to efficient coding
- Reduces redundancy

  eg : Fractal , Wavelet , DCT & Fourier Transforms

Step 2: Quantization of transformed image

- Reduces no of values
- Introduces errors
- Trades off image quality for data rate

eg) scalar & vector quantization

Step 3: Encode the transformed or the quantized data

- The image is a set of symbols(o/p of a quantizer)
- a uniquely decodable code is assigned for each symbol
- codes are assigned by exploiting statistical redundancy

eg) Arithmetic, Huffman ,Entropy encoding

6) Why DCT ?

Discrete frequency transforms provide a method to obtain a global view of data within a window. Discrete cosine transform is the frequency transform for practical image processing because of its excellent energy compaction property. Another reason for its popularity is the existence of a fast implementation for the algorithm.

Stationary signals having complex waveforms (usually more than one wave component in it) can be easily represented in **spatial frequency domain** with the help of DCT. The DCT matrix is usually an 8x8 matrix as per jpeg standards with high and low frequency components . This is the stage where compression can be enforced by eliminating high frequency values under the pretext that they are not visible to the human eye. These set of high frequency values are said to be irrelevant.

7) Why Entropy encoding?

A prior information of image statistics is required for Huffman coding .Since Huffman coding achieves the minimum amount of redundancy in a fixed set of variable length codes this does not mean Huffman coding is an optimal coding method. But it provides best approximation for coding symbols when used with fixed width codes .

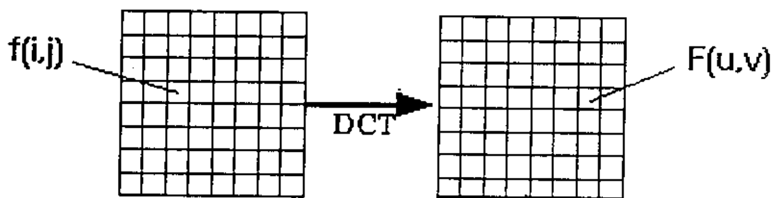
The problem with Huffman or any other coding is that they use integral number of bits in each code for example 2.5 bits cannot be represented using them ,instead it can be either 2 or 3 bits . This problem is eliminated in entropy encoding and bit codes like 2.5 or 3.5 can be very well expressed with ease. This makes entropy encoding a better suited and optimal encoding technique.

8) PHASES OF COMPRESSION

- DCT (Discrete Cosine Transformation)
- Quantization
- Zigzag Scan
- DPCM on DC component
- RLE on AC Components
- Entropy Coding

8.1) Discrete Cosine Transform (DCT)

- From spatial domain to frequency domain:



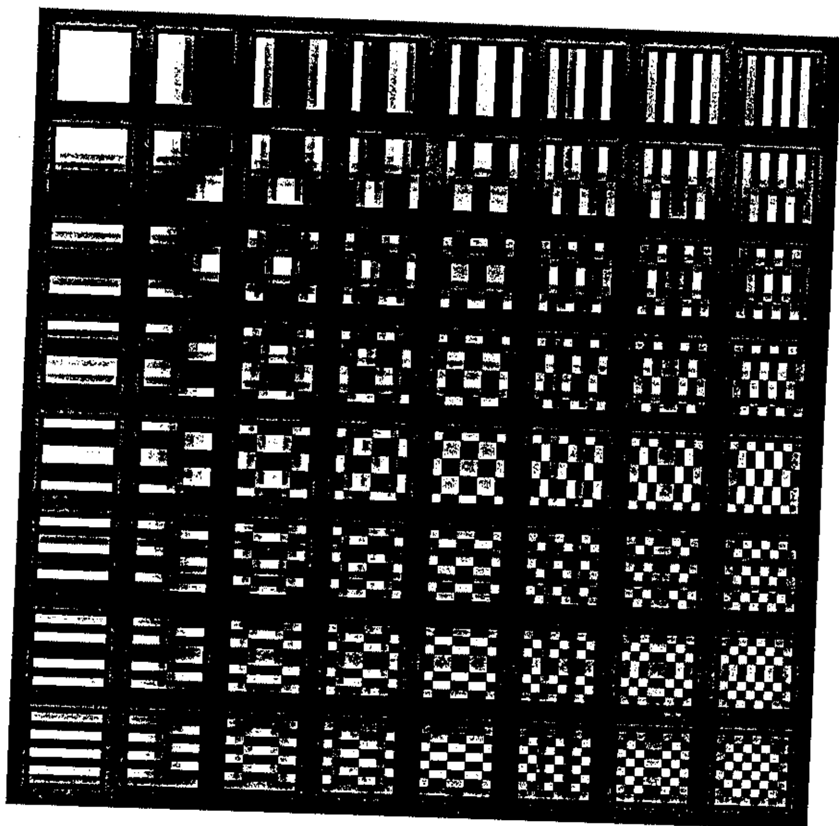
- **DEFINITIONS**

Discrete Cosine Transform (DCT):

$$F(u,v) = \frac{\Lambda(u)\Lambda(v)}{4} \sum_{i=0}^{7}\sum_{j=0}^{7} \cos\frac{(2i+1)\cdot u\pi}{16} \cdot \cos\frac{(2j+1)\cdot v\pi}{16} \cdot f(i,j)$$

$$\Lambda(\xi) = \begin{cases} \frac{1}{\sqrt{2}} & for\ \xi = 0 \\ 1 & otherwise \end{cases}$$

- The 64 (8 x 8) DCT basis functions:



One alternative method to the DFT is the Discrete Cosine Transform (DCT). The last term, *Transform*, means what it implies, a conversion from one domain to another. A domain is a set for the definition of some entity. In this case, the information carried by a signal is transformed from the time domain (or spatial domain) into the frequency domain.

Cosine is one of the trigonometric functions. It is periodic, and an even function. In this method, it is used as the basis function for the transform.

The term *Discrete* means that the value of some variable is known only at certain points in time or space. These points are usually uniformly distributed. Discrete is commonly regarded as the opposite of continuous when the value of the variable is known in every point in time or space. The process of converting a continuous-time signal to a discrete-time signal is called *sampling*. *Quantization*, on the other hand, means selecting a discrete value to represent a continuous one, thus these two terms are related but should not be mixed. In this method (DCT), it is very natural to use discrete values since the implementation will be executed in a computer with the capability to store only discrete values from a finite set.

8.2) Quantization

DCT is a lossless transformation that does not actually perform compression. It prepares for the "lossy",or quantization ,stage of the process.

The DCT output matrix takes more space to store than the original matrix of pixels .The input to the DCT function consists of eight-bit pixel values ,but the values that come out can range from a low of –1,024 to a high of 1,023 occupying eleven bits. Something drastic has to happen before the DCT matrix can take up less space.

Quantization is simply the process of reducing the number of bits needed to store an integer value by reducing the number of bits needed to store an integer value by reducing the precision of the integer . Once a DCT image has been compressed we can generally reduce the precision of the coefficients more and more as we move away from the DC coefficient at the origin.

The JPEG algorithm implements Quantization matrix.For every element position in the DCT matrix ,a corresponding value in the Quantization matrix gives a quantum value .The quantum value indicates what the step size is going to be for the element in the compressed rendition of the picture ,with values ranging from one to 255.

The actual formula for quantization is quite simple and is made use of in this project.

$$\text{Quantized Value}(I,J) = \frac{DCT(I,J)}{Quantum(I,J)} \quad \text{Rounded to nearest integer}$$

$F'[u, v] = \text{round}\,(F[u, v] / q[u, v])$.

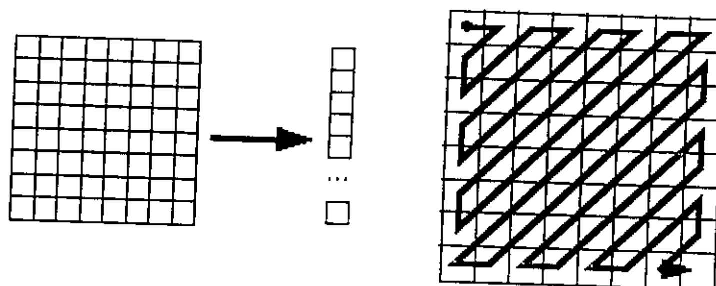
Why? -- To reduce number of bits per sample

Example: 101101 = 45 (6 bits).
$q[u, v] = 4$ --> Truncate to 4 bits: 1011 = 11.

- Quantization error is the main source of the Lossy Compression.

8.3) Zig-zag Scan

After quantization the resulting matrix would look like an upper triangular matrix .Hence the best way to Run Length Encode the zeros or the high frequency values is to scan the matrix in a zig-zag pattern to obtain maximum length of a particular run



8.4) Differential Pulse Code Modulation (DPCM) on DC component

- DC component is large and varied, but often close to previous value.
- Encode the difference from previous 8 x 8 blocks -- DPCM

8.5) Run Length Encode (RLE) on AC components

- 1 x 64 vector has lots of zeros in it
- Keeps *skip* and *value*, where *skip* is the number of zeros and *value* is the next non-zero component.
- Send (0,0) as end-of-block sentinel value.

8.6) Entropy Coding

- Categorize DC values into SIZE (number of bits needed to represent) and actual bits.

TABLE REPRESENTING SIZE AND VALUE

SIZE	VALUE
1	-1, 1
2	-3, -2, 2, 3
3	-7..-4, 4..7
4	-15..-8, 8..15
.	.
.	.
.	.
10	-1023..-512, 512..1023

8.7) Implementation of DCT-based coding

The main procedures for all encoding processes is based on the DCT. It illustrates the special case of a single-component image; this is an appropriate simplification for overview purposes, because all processes specified in this specification operate on each image component independently.

In the encoding process the input component's samples are grouped into 8 x 8 blocks, and each block is transformed by the forward DCT (FDCT) into a set of 64 values referred to as DCT coefficients. One of these values is referred to as the DC coefficient and the other 63 as the AC coefficients.

Each of the 64 coefficients is then quantified using one of 64 corresponding values from a quantization table. No default values for quantization tables are specified in this specification ; applications may specify values which customise picture quality for their particular image characteristics, display devices, and viewing conditions. This step determines the compression rate you want and thus produces the lossy compression.

After quantization, the DC coefficient and the 63 AC coefficients are prepared for entropy encoding. The previous quantized DC coefficient is used to predict the current quantized DC coefficient, and the difference is encoded. The 63 quantized AC coefficients undergo no such differential encoding, but are converted into a one-dimensional zig-zag sequence.

The quantized coefficients are then passed to an entropy encoding procedure which compresses the data further. One of two entropy coding procedures can be used.

The following steps are taken for compressing each 8x8 block.

1. Shift the block
2. Perform a FDCT on the block
3. Quantize the block
4. Subtract the last DC coefficient from the current DC coefficient
5. Zigzag the block
6. Zero run length encode the block
7. Break down the non-zero coefficients into variable-length binary numbers & their lengths
8. Entropy encode the run lengths & binary number lengths
9. Write the entropy encoded information & binary numbers to the output

9) PHASES OF DECOMPRESSION

1) Decoding
2) Dequantization
3) Inverse DCT application to the dequantized image.

9.1) Decoding and Dequantization

During decoding the dequantization formula operates in reverse:

DCT(I,J)=Quantized Value(I,J) * Quantum(I,J)

Once again , from this we can see when you use large quantum values, you run the risk of generating large errors in the DCT output during dequantization .Fortunately, errors generated in the high frequency components during dequantization normally do not have a serious effect on picture quality.

9.3) Inverse DCT

The Inverse DCT is performed using the exact reverse of the operations performed in the DCT . First ,the DCT values in the N-by-N matrix are multiplied by the cosine transform matrix .The result of this transformation is stored in a temporary N-by-N matrix of doubles .This matrix is then multiplied by the transposed cosine transform matrix .The result of this multiplication is rounded ,scaled to the correct unsigned character range of zero to 255,then stored in the output block of pixels.

9.3.1) Inverse Discrete Cosine Transform (IDCT)

$$\tilde{f}(i,j) = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7}\Lambda(u)\Lambda(v)\cos\frac{(2i+1)\cdot u\pi}{16}\cdot\cos\frac{(2j+1)\cdot v\pi}{16}\cdot F(u,v$$

$$\Lambda(\xi) = \begin{cases} \frac{1}{\sqrt{2}} & for\ \xi = 0 \\ 1 & otherwise \end{cases}$$

10) APPLICATIONS OF DIGITAL IMAGE COMPRESSION

10.1) Image Compression in Healthcare

a) Radiology

The archiving and transmission requirements of digital images in radiology are very high and image compression has alleviated some of the burden Browning et al (1996) believe that without the use of significant levels of compression, image transfer speeds would be unacceptably slow. Pratt et al (1998) found that PACS (Picture Archiving and Communication System) and computed radiography systems would provide cost savings only if long-term archives were compressed at a rate of 10:1.

Many compression algorithms have been used including Wavelets , principal components neural network, multilevel decomposition and discrete cosine transform coding. Maldijan et al (1997) obtained 90% compression using the wavelet transform with minimal loss of image detail and recorded standard modem transmission times of less than five seconds per compressed image.

Concerns about image degradation post-compression have been widely investigated. Most of these studies have been carried out using chest radiographs (the most common x-ray examination). Erickson et al (1997) found that lossy compression of 40:1 or more could be used without perceptible loss in the representation of anatomical structures. Savenko et al (1998) confirmed these findings and further stated that there was no

statistically significant difference in diagnostic accuracy at 80:1 compression. Both used wavelet image compression. Cox et al (1996) used a standard discrete cosine transform (DCT) technique and observed that performance on the clinical task of detecting chest nodules would not be affected materially by compression of up to 44:1. Kido et al (1996) also used the DCT method but found that the interpretation of images with a compression ratio of 30:1 was significantly less accurate than that of uncompressed images ($p<.05$).

b) Cardiology

Silber et al (1997) investigated the impact of various compression rates on interpretation of digital coronary angiograms and observed no clinically relevant loss of information at a ratio of 6:1. They noted that the ACC (American College of Cardiology)/ACR (American College of Radiology)/NEMA (National Electrical Manufacturers Association) guidelines only allow lossless compression of 2:1 therefore the angiograms could not be viewed in real-time directly from the CD-ROM. Elion and Whiting (1996) are wary of using lossy compression on coronary angiograms for fear of reducing any feature's detectablity. This, they state, would be clear evidence of the potential loss of clinical information. Holmes et al (1998), however, suggest that lossy compression rates of up to 10:1 may be clinically acceptable. A ratio of 10:1 would result in the dramatic decrease in the data rate requirements for real-time display of 512X512X8 bit images from 7.5 MB/sec to 750 KB/sec.

11) Future Trends and Conclusion

The systems of compression of images described here are not the only ones one can find on the market, but are normalized systems.

Systems such as fractal compression give high compression rates. These are based on a sharp analysis of the texture of the image and are very slow for the compression and fast for the decompression. These systems may be interesting but are far from a normalization process.

The search for even more efficient compression algorithms will continue into the next century. Martyn (1996) believes that this search will move away from pixel-based methods toward mathematical methods similar to wavelets and fractals.

Todd-Pokropek (1995) believes that the medical imaging department of the future will be an audio-visual centre that will be primarily concerned about extracting information from digital images.

It is expected that newer mathematical transforms that compress images with comparatively less degradation in quality, will be developed in the near future. Whatever the future brings, image compression technology will be playing a significant role in the delivery of healthcare for some time to come.

12) BIBLIOGRAPHY

Books

Stones , R., Matthew , N., "***Beginning Linux Programming*** ", Wrox Press , 1102 Warwick Road , Birmingham –B276BH, Second Edition.

Nelson , M., Gailly , J.L., "***The Data Compression Book***" , BPB Publications , B-14 Connaught Place , New Delhi -110001, Second Edition.

Petersen Richard , "***The Complete Reference - LINUX***" ,Tata McGraw Hill Publishing Limited , New Delhi, Second Edition.

URLs

1) Ivey ,E., "***JpegCompression***",

   http://web.usxchange.net/elmo/jpeg.htm

2) Chiang ,L., "***Digital Data Compression***"

   http://www.image.cityu.edu.hk/~loben/thesis/node10.html

3) Belov ,C.,"***Discrete Cosine Transform***",

   http://www.cabelov.com/flo/scopro/jsdct.shtml

4) Dominic Maguire ,D.C.R .,"***Current Trends in Digital Image Compression*** ",

   http://twinpentium.lcp.linst.ac.uk/library/current.htm

DISP.C

```
/* Converts a 256 color image  into gray scale for photo
quality.   The intermediate steps and also  the appearance of
the picture in    various RGB combinations are  shown

 The header file vga.h contains functions that support
graphics. It
needs to be compiled as follows :
   cc –lvga disp.c –o disp

 The file disp ontains the executable code. When disp is run
it displays the image supplied as argument at the linux shell
prompt.

Usage :     ./disp  <Filename>
Eg:         ./disp cheetah.raw

where cheetah.raw is an image file.

*/
 # include <stdio.h>
 # include <sys/types.h>
 # include <sys/time.h>
 # include <unistd.h>
 # include <vga.h>
```

```
main(int argc,char* argv[])
{
 int u,cx,x,mode,a,b,c,r,x1,y1,y;
 FILE *fptr;
 struct timeval t;  /*   defined       */
 fd_set* rfds;      /*   to set delay */

If(argc!=2)
   {
          printf("\n Invalid arguments ");
          printf("\n Usage : ./disp file\n");
   }
 printf("Enter the Image width :  ");
 scanf("%d",&x);
 printf("Enter the Image Height :  ");
 scnf("%d",&y);
 printf("Enter the Graphics mode :  ");
 scanf("%d",&mode);
 if(mode>10) mode=5;
 if(x==y==0)
  {
    printf("\n 320x200 image aasumed ");
    x=320;
    y=200;
  }
 fptr=fopen(argv[1],"rb");
 rewind(fptr);
```

```
 vga_setmode(mode);
/* sets 320*200 screen with 256 colors */
 vga_clear();
 for(r=0;r<7;r++)
 {
   for(y1=0;y1<y;y1++)
   {
    for(x1=0;x1<x;x1++)
    {
      vga_setcolor(fgetc(fptr)>>r);
      vga_drawpixel(x1,y1);
    }
   }
  t.tv_usec=0; /* sets delay for 2 seconds */
  t. tv_sec=2;
  select(1,rfds,NULL,NULL,&t);
  vga_clear();
  rewind(fptr);
 }
 rewind(fptr);

/* Indicates various stages during gray scale conversion  and
includes
   the mathematical computation for the necessary  conversion
*/
  for(u=0;u<5;u++)
  {
```

```
        for(y1=0;y1<200;y1++)
        {
         for(x1=0;x1<320;x1++)
         {
               r=fgetc(fptr)>>u;
               if(r<=16)
               r+=16;
               vga_setcolor(r);
               vga_drawpixel(x1,y1);
         }
        }

    t.tv_usec=0;
    t.tv_sec=2;
    select(1,rfds,NULL,NULL,&t);  /*  delay of 2 seconds */
   if(u!=4) vga_clear();          /* between images     */
   rewind(fptr);
  }
   t.tv_usec=0;
   t.tv_sec=8;
   select(1,rfds,NULL,NULL,&t);
fclose(fptr);
}
```

SYNCDISP.C

/* Displays 2 images simultaneously . This uses multithreading concept to display 2 images simultaneously. Two semaphore variables are used to achieve synchronization so that the main thread doesn't hold the processor for a long time without giving it to the second thread.

It should be compiled as follows :

cc syncdisp.c –lvga –lpthread –o syncdisp

The executable file will be available in syncdisp and should be run as follows

```
     ./syncdisp    <filename1>       <filename2>
eg)  ./syncdisp    cheetah.raw       deccheetah.raw
```

The code written under the main function displays filename1 and that written under thread_function displays the filename2*/

```
# include <stdio.h>
# include <unistd.h>
# include <pthread.h>
# include <semaphore.h>
# include <stdlib.h>
```

```
# include <string.h>
# include <vga.h>
# ifndef _REENTRANT
# define _REENTRANT
# endif
void *thread_function(void *arg);

/*

   Two semaphore variables are defined so as to
   achieve synchronization

*/

sem_t bin_sem;
sem_t bin_sem1;

FILE *fptr;
FILE *fptr1;
/* beginning of main thread */

main(int argc,char* argv[])
{
 int res;
 int u=0,cx,x,modes,a,b,c,r;
 pthread_t a_thread;
 void *thread_result;
```

```
if(argc!=3)
        {
          printf("Invalid arguments \n");
          printf("Usage : ./syncdisp file1 file2 \n");
             exit(0);
        }
fptr=fopen(argv[1],"rb");
fptr1=fopen(argv[2],"rb");
vga_setmode(5);
vga_clear();
rewind(fptr);

   /* Initialize semaphore variable */

res=sem_init(&bin_sem,0,0);
if(res!=0)
      {
        perror("semaphore initialization failed ");
        exit(EXIT_FAILURE);
      }
  /* Initialize semaphore variable */

    res=sem_init(&bin_sem1,0,0);
    if(res!=0)
      {
        perror("semaphore initialization failed ");
        exit(EXIT_FAILURE);
```

```
}

/* Creation of a thread -- ctrl passed to thread_function */


res=pthread_create(&a_thread,NULL,thread_function,NULL);
        if(res!=0)
            {
             perror("Thread Creation failed ");
             exit(EXIT_FAILURE);
            }

              /* Displays file 1   */

       while(u!=5)
       {
       for(cx=0;cx<100;cx++)
       {
        for(x=0;x<320;x++)
        {
         r=fgetc(fptr)>>u;
         if(r<=16)
         r+=16;
         vga_setcolor(r);
         vga_drawpixel(x,cx);
       }
```

```
    sem_post(&bin_sem);
    sem_wait(&bin_sem1);
    sleep(0.5);
  }
   rewind(fptr);
  u++;
 }
 sleep(10);
sem_destroy(&bin_sem);
exit(EXIT_SUCCESS);
}

/*  Beginning of new thread  */
void *thread_function(void *arg)
{
 int v=0,cx,x,r;

 /* Displays file 2 */
 sem_wait(&bin_sem);
 rewind(fptr1);
 while(v!=5)
 {
    for(cx=0;cx<100;cx++)
    {
          for(x=0;x<320;x++)
          {
           r=fgetc(fptr1)>>v;
```

```
            if(r<=16)
            r+=16;
            vga_setcolor(r);
            vga_drawpixel(x,100+cx);
          }
          sem_post(&bin_sem1);
          sem_wait(&bin_sem);
    }
          sleep(4);
          vga_clear();
          rewind(fptr1);
       v++;
       }
  fclose(fptr);
  pthread_exit(NULL);
}
```

SERVER.C

```
/* This program simulates a server .It waits for a client to
connect and once the connection is established it verifies
whether the file requested by a client is available in its
current directory . If found it sends the file to the client , else
returns the message file not found in server.
   This program is written in such a way that even remote
clients can connect to the machine where  this program is
executed .This is usually executed in the background as
./server & at the shell prompt.

Usage : ./server &
*/

# include <stdio.h>
# include <unistd.h>
# include <netinet/in.h>
# include <sys/types.h>
# include <sys/socket.h>
# include <arpa/inet.h>
# include <stdlib.h>
main()
{
 int server_sockfd,client_sockfd;
 int i=0;
```

```
char *ch=(char*)malloc(20);
char *senddata;
int client_len,server_len,ret;
struct sockaddr_in server_address;
        struct sockaddr_in client_address;
FILE *fp;
unlink("server_socket");
server_sockfd=socket(AF_INET,SOCK_STREAM,0);
server_address.sin_family=AF_INET;
server_address.sin_addr.s_addr=INADDR_ANY;
server_address.sin_port=9734;
server_len=sizeof(server_address);
bind(server_sockfd,(struct sockaddr
*)&server_address,server_len);
listen(server_sockfd,5);
while(1)
{
 printf("server waiting ");
 client_len=sizeof(client_address);
 client_sockfd=accept(server_sockfd,(struct
 sockaddr*)&client_address,&client_len);
 read(client_sockfd,ch,20);
 printf("\n%s\n",ch);
 senddata=(char*)malloc(6400000);
 fp=fopen(ch,"r");
 if(fp==NULL) {
          strcpy(senddata,"FILE NOT AVAILABLE IN
```

```
SERVER");
                ret=write(client_sockfd,senddata,strlen(senddata));
                printf("\n no of bytes sent = %d\n",ret);
                close(client_sockfd);
                exit(0);
               }
        while(!feof(fp))
        {
          senddata[i]=fgetc(fp);
          i++;
        }
         ret=write(client_sockfd,senddata,strlen(senddata)-1);
         printf("\n no of bytes sent = %d\n",ret);
         close(client_sockfd);
         if(ret!=0) exit(0); /* exit when data is sent */
      }
     }
```

Get.c

```
/* This program when run on a machine will simulate a client. It requests
a file from the server and if  available , will be returned by the server.

Usage :  ./get <srcfilename>  <destfilename>

Srcfilename is the one available in the server and destfilename is the
name of the file that is obtained from the server and stored in the client's
directory

*/

# include <stdio.h>
# include <unistd.h>
# include <netinet/in.h>
# include <sys/types.h>
# include <sys/socket.h>
# include <arpa/inet.h>
# include <string.h>
int main(int argc,char *argv[])
{
        int sockfd;
        int len,i=0;
```

```
struct sockaddr_in address;
int result;
FILE *fp;



char *chnew;
char *ch=argv[1];
if(argc!=3)
{
 printf("\nINVALID ARGUMENTS \n");
 printf("./get requestFilename  newFilename \n");
 exit(0);
}
sockfd=socket(AF_INET,SOCK_STREAM,0);
address.sin_family=AF_INET;
address.sin_addr.s_addr=inet_addr("199.199.199.3");
address.sin_port=9734;
len=sizeof(address);
result=connect(sockfd,(struct sockaddr *)&address,len);
if(result == -1)
{
    perror("oops:client");
    exit(1);
}
write(sockfd,ch,strlen(ch));
chnew=(char*)malloc(64000);
read(sockfd,chnew,64000);
```

```
        printf("%d",strlen(chnew));
        fp=fopen(argv[2],"w+");
        if(fp==NULL) {
                printf("\nerror opening file\n");
                exit(0);
               }
   for(i=0;i<strlen(chnew);i++)
        {
         printf("%c",chnew[i]);
         putc(chnew[i],fp);
        }
        fclose(fp);
      /*write(1,chnew,strlen(chnew));*/
      close( sockfd );
      exit(0);
}
```

Comp.c

```
/*
 * This is the DCT module, which implements a graphics
compression
 * program based on the Discrete Cosine Transform.  It
needs to be
 * linked with the standard support routines.
 *
 It should be compiled as follows
      cc  -lm comp.c –o comp

 Usage : ./comp   <srcFile>     <compressedFile>

 */

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

#define ROWS         200
#define COLS         320
#define N             8

/*
 * This macro is used to ensure correct rounding of integer
```

```
values.
 */
#define ROUND( a )  (((a)<0)?(int)((a) - 0.5 ):(int) ( (a) +
0.5 ) )

char *CompressionName = "DCT compression";
char *Usage          = "infile outfile [quality]\nQuality from
0-25";



unsigned char PixelStrip[ N ][ COLS ];
double C[ N ][ N ];
double Ct[ N ][ N ];
int InputRunLength;
int OutputRunLength;
int Quantum[ N ][ N ];

struct zigzag {
   int row;
   int col;
} ZigZag[ N * N ] =
{
   {0, 0},
   {0, 1}, {1, 0},
   {2, 0}, {1, 1}, {0, 2},
   {0, 3}, {1, 2}, {2, 1}, {3, 0},
   {4, 0}, {3, 1}, {2, 2}, {1, 3}, {0, 4},
```

```
    {0, 5}, {1, 4}, {2, 3}, {3, 2}, {4, 1}, {5, 0},
    {6, 0}, {5, 1}, {4, 2}, {3, 3}, {2, 4}, {1, 5}, {0, 6},
    {0, 7}, {1, 6}, {2, 5}, {3, 4}, {4, 3}, {5, 2}, {6, 1}, {7, 0},
    {7, 1}, {6, 2}, {5, 3}, {4, 4}, {3, 5}, {2, 6}, {1, 7},
    {2, 7}, {3, 6}, {4, 5}, {5, 4}, {6, 3}, {7, 2},
    {7, 3}, {6, 4}, {5, 5}, {4, 6}, {3, 7},
    {4, 7}, {5, 6}, {6, 5}, {7, 4},
    {7, 5}, {6, 6}, {5, 7},
    {6, 7}, {7, 6},
    {7, 7}
};

 typedef struct bit_file
   {
    FILE *file;
    unsigned char mask;
    int rack;
    int pacifier_counter;
   } BIT_FILE;

BIT_FILE *OpenOutputBitFile(char *name);
 int quality;
 main(int argc,char* argv[])
  {
   BIT_FILE *output;
   FILE *input;
   printf("Enter the Q - Factor : ");
```

```
  scanf("%d",&quality);
  printf("        RESULTANT MATRIX  \n");
  input=fopen(argv[1],"rb");
  output= OpenOutputBitFile(argv[2]);

  CompressFile(input,output);
  CloseOutputBitFile(output);
  fclose(input);
  printf("\ndone!\n");
  getchar();
 }
```

```
/*
 * The initialization routine has the job of setting up the
Cosine
 * Transform matrix, as well as its transposed value.  These
two matrices
 * are used when calculating both the DCT and its inverse.
In addition,
  * the quantization matrix is set up based on the quality
 parameter
  *   passed   to this routine.  Additionally, the two run length
 parameters
  *   are both set to 0.
 */
```

```
Initialize(int quality)
{
   int i;
   int j;
   double pi = atan( 1.0 ) * 4.0;
   OutputRunLength = 0;
   InputRunLength = 0;
   for ( j = 0 ; j < N ; j++ ) {
        C[ 0 ][ j ] = 1.0 / sqrt( (double) N );
        Ct[ j ][ 0 ] = C[ 0 ][ j ];
   }
   for ( i = 1 ; i < N ; i++ ) {
        for ( j = 0 ; j < N ; j++ ) {
           C[ i ][ j ] = sqrt( 2.0 / N ) *
                           cos( pi * ( 2 * j + 1 ) * i / ( 2.0 * N ) );
           Ct[ j ][ i ] = C[ i ][ j ];
        }
   }


}

/*
 * This routine is called when compressing a grey scale file.
It reads
 * in a strip that is N (usually 8) rows deep and COLS
```

```
(usually 320)
 * columns wide.  This strip is then repeatedly processed, a
block at a
 * time, by the forward DCT routine.
 */
ReadPixelStrip( FILE *input,unsigned char strip[ N ][ COLS
])
{
   int row;
   int col;
   int c;

   for ( row = 0 ; row < N ; row++ )
         for ( col = 0 ; col < COLS ; col++ )
         {
           c = getc( input );
           strip[ row ][ col ] = (unsigned char) c;
         }
}
/*
 * This routine reads in a DCT code from the compressed
file.  The code
 * consists of two components, a bit count, and an encoded
value.  The
 * bit count is encoded as a prefix code with the following
binar
 * values:
```

```
*
*           Number of Bits   Binary Code
*           --------------   -----------
*                 0              00
*                 1              010
*                 2              011
*                 3              1000
*                 4              1001
*                 5              1010
*                 6              1011
*                 7              1100
*                 8              1101
*                 9              1110
*                 10             1111
*
 * A bit count of zero is followed by a four bit number
telling how many
 * zeros are in the encoded run.  A value of 1 through ten
indicates a
 * code value follows, which takes up that many bits.  The
encoding of values
 * into this system has the following characteristics:
*
*       Bit Count           Amplitudes
*       ---------      ------------------------
*           1                  -1, 1
*           2              -3 to -2, 2 to 3
```

```
 *        3          -7 to -4, 4 to 7
 *        4         -15 to -8, 8 to 15
 *        5         -31 to -16, 16 to 31
 *        6         -63 to -32, 32 to 64
 *        7        -127 to -64, 64 to 127
 *        8        -255 to -128, 128 to 255
 *        9        -511 to -256, 256 to 511
 *       10        -1023 to -512, 512 to 1023
 *
 */



OutputCode( BIT_FILE *output_file,int code)
{
  int top_of_range;
  int abs_code;
  int bit_count;

  if ( code == 0 ) {
      OutputRunLength++;
      return;
  }
  if ( OutputRunLength != 0 ) {
      while ( OutputRunLength > 0 ) {
         OutputBits( output_file, 0L, 2 );
         if ( OutputRunLength <= 16 ) {
```

```
                OutputBits( output_file,
                        (unsigned long) ( OutputRunLength - 1
), 4 );
                OutputRunLength = 0;
            } else {
                OutputBits( output_file, 15L, 4 );
                OutputRunLength -= 16;
            }
        }
    }
    if ( code < 0 )
        abs_code = -code;
    else
        abs_code = code;
    top_of_range = 1;
    bit_count = 1;
    while ( abs_code > top_of_range ) {
        bit_count++;
        top_of_range = ( ( top_of_range + 1 ) * 2 ) - 1;
    }
    if ( bit_count < 3 )
        OutputBits( output_file, (unsigned long) ( bit_count +
1 ), 3 );
    else
        OutputBits( output_file, (unsigned long) ( bit_count +
5 ), 4 );
    if ( code > 0 )
```

```
        OutputBits( output_file, (unsigned long) code,
bit_count );
   else
        OutputBits( output_file, (unsigned long) ( code +
top_of_range ),
                    bit_count );
}

/*
 * This routine takes DCT data, puts it in Zig Zag order, the
quantizes
 * it, and outputs the code.
 */

WriteDCTData( BIT_FILE *output_file,int output_data[ N
][ N ])
{
  int i,j;
  int row;
  int col;
  double result;
  for ( i = 0 ; i < N ; i++ )
       for ( j = 0 ; j < N ; j++ )
       {
           Quantum[ i ][ j ] = 1 + ( ( 1 + i + j )  * quality );
           printf("%d \t",Quantum[i][j]);
       }
```

```
    for ( i = 0 ; i < ( N * N ) ; i++ ) {
        row = ZigZag[ i ].row;
        col = ZigZag[ i ].col;
        result = output_data[ row ][ col ] / Quantum[ row ][
col ];
        OutputCode( output_file, ROUND( result ) );
    }
}

/*
 *              DCT = C * pixels * Ct
 */

ForwardDCT( unsigned char *input[ N ],int output[ N ][ N
])
{
  double temp[ N ][ N ];
  double temp1;
  int i;
  int j;
  int k;

/* MatrixMultiply( temp, input, Ct ); */
  for ( i = 0 ; i < N ; i++ ) {
      for ( j = 0 ; j < N ; j++ ) {
         temp[ i ][ j ] = 0.0;
         for ( k = 0 ; k < N ; k++ )
```

```
                temp[ i ][ j ] += ( (int) input[ i ][ k ] - 128 ) *
                                    Ct[ k ][ j ];
        }
    }

/*  MatrixMultiply( output, C, temp ); */
    for ( i = 0 ; i < N ; i++ ) {
        for ( j = 0 ; j < N ; j++ ) {
            temp1 = 0.0;
            for ( k = 0 ; k < N ; k++ )
                temp1 += C[ i ][ k ] * temp[ k ][ j ];
            output[ i ][ j ] = ROUND(temp1);
        }
    }
}

/*
 * The Inverse DCT routine implements the matrix function:
 *
 *              pixels = C * DCT * Ct
 */



/*
 * This is the main compression routine.  By the time it gets
called,
 * the input and output files have been properly opened, so
```

```
all it has to
 * do is the compression.  Note that the compression routine
expects an
 * additional parameter, the quality value, ranging from 0 to
25.
 */

CompressFile(FILE *input,BIT_FILE *output)
{
   int row;
   int col;
   int i;
   unsigned char *input_array[ N ];
   int output_array[ N ][ N ];
   printf( "Using quality factor of %d\n", quality );
   Initialize( quality );

   OutputBits( output, (unsigned long) quality, 8 );
   for ( row = 0 ; row < ROWS ; row += N ) {
        ReadPixelStrip( input, PixelStrip );
        for ( col = 0 ; col < COLS ; col += N ) {
           for ( i = 0 ; i < N ; i++ )
                input_array[ i ] = PixelStrip[ i ] + col;
           ForwardDCT( input_array, output_array );
           WriteDCTData( output, output_array );
        }
   }
```

```
 OutputCode( output, 1 );

}

BIT_FILE *OpenOutputBitFile(char *name)
{
BIT_FILE *bit_file;
bit_file=(BIT_FILE *) calloc(1,sizeof(BIT_FILE));
if (bit_file==NULL)
return(bit_file);
bit_file->file = fopen(name,"wb");
bit_file ->rack=0;
bit_file ->mask=0x80;
bit_file ->pacifier_counter=0;
return(bit_file);
}


CloseOutputBitFile(BIT_FILE *bit_file)
{
if(bit_file->mask!=0x80)
if(putc(bit_file->rack,bit_file->file)!=bit_file->rack)
 {
 printf("\n\n\n");
 //fatal_error("Fatal error in CloseBitFile!\n");
 }
```

```
fclose(bit_file->file);
free((char *)bit_file);
}
OutputBits(BIT_FILE *bit_file,unsigned long code,int
count)
{
unsigned long mask;
mask= 1L<<(count-1);
while(mask!=0)
{
if(mask & code)
 bit_file->rack|=bit_file->mask;
bit_file->mask >>=1;
if(bit_file->mask==0)
{
if (putc(bit_file->rack,bit_file->file)!=bit_file->rack)
{
printf("\n\n\n");
//fatal_error("Fatal error in OutputBit!\n");
}
else
if((bit_file->pacifier_counter++ & 2047)==0)
{
 }
bit_file->rack=0;
bit_file->mask=0x80;
    }
```

```
mask >>= 1;
}
}
FilePrintBinary(file,code,bits)
FILE *file;
unsigned int code;
int bits;
{
unsigned int mask;
mask = 1<< (bits-1);
while(mask!=0)
{
if(code & mask)
fputc('1',file);
else
fputc('0',file);
mask>>=1;
}
}
```

Decomp.C

```
/*
* This is the module, which implements a decompression
* program based on the Inverse Discrete Cosine Transform.
*
*  It should be compiled as follows :
*    cc –lm decomp.c –o decomp
* Usage :
   ./decomp <compresedFile>  <decompressedFile>  <qf>

   qf => quality factor with which the image was
   compressed
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/*
 * A few parameters that could be adjusted to modify the
decompression * algorithm.  The first two define the number
of rows and columns in
 * the gray scale image.  The last one, 'N', defines the DCT
block
 * size.
 */
```

```
#define ROWS        200
#define COLS         320
#define N            8

/*
 * This macro is used to ensure correct rounding of integer
values.
 */
#define ROUND( a )     ( ( (a) < 0 ) ? (int) ( (a) - 0.5 ) :  (int)
( (a) + 0.5 ) )

char *CompressionName = "DCT compression";
char *Usage          = "infile outfile [quality]\nQuality from
0-25";


unsigned char PixelStrip[ N ][ COLS ];
double C[ N ][ N ];
double Ct[ N ][ N ];
int InputRunLength;
int OutputRunLength;
int Quantum[ N ][ N ];

struct zigzag {
   int row;
   int col;
} ZigZag[ N * N ] =
```

```
{
  {0, 0},
  {0, 1}, {1, 0},
  {2, 0}, {1, 1}, {0, 2},
  {0, 3}, {1, 2}, {2, 1}, {3, 0},
  {4, 0}, {3, 1}, {2, 2}, {1, 3}, {0, 4},
  {0, 5}, {1, 4}, {2, 3}, {3, 2}, {4, 1}, {5, 0},
  {6, 0}, {5, 1}, {4, 2}, {3, 3}, {2, 4}, {1, 5}, {0, 6},
  {0, 7}, {1, 6}, {2, 5}, {3, 4}, {4, 3}, {5, 2}, {6, 1}, {7, 0},
  {7, 1}, {6, 2}, {5, 3}, {4, 4}, {3, 5}, {2, 6}, {1, 7},
  {2, 7}, {3, 6}, {4, 5}, {5, 4}, {6, 3}, {7, 2},
  {7, 3}, {6, 4}, {5, 5}, {4, 6}, {3, 7},
  {4, 7}, {5, 6}, {6, 5}, {7, 4},
  {7, 5}, {6, 6}, {5, 7},
  {6, 7}, {7, 6},
  {7, 7}
};
typedef struct bit_file
  {
   FILE *file;
   unsigned char mask;
   int rack;
   int pacifier_counter;
  } BIT_FILE;
BIT_FILE *OpenInputBitFile(char *name);
int quality;
main(int argc,char* argv[])
```

```
{
 int* w;
 FILE *output;
 BIT_FILE *input;
 if(argc!=4)
  {
   printf("\n Invalid arguments ");
   printf("\n Usage : ./decomp compressedfile decompress-
file\n");
   exit(0);
  }
  quality=atoi(argv[3]);
  input=OpenInputBitFile(argv[1]);
  output=fopen(argv[2],"wb");
  ExpandFile(input,output);
  CloseInputBitFile(input);
  printf("done !!!");
  fclose(output);
  getchar();
 }


   BIT_FILE *OpenInputBitFile(char *name)
   {
        BIT_FILE *bit_file;
        bit_file=(BIT_FILE *) calloc(1,sizeof(BIT_FILE));
          if (bit_file==NULL)
```

```
        return(bit_file);
        bit_file->file = fopen(name,"rb");
        bit_file ->rack=0;
        bit_file ->mask=0x80;
        bit_file ->pacifier_counter=0;
        return(bit_file);
    }

    CloseInputBitFile(BIT_FILE *bit_file)
    {
        fclose(bit_file->file);
        free((char *)bit_file);
    }

    unsigned long InputBits(BIT_FILE *bit_file,int
bit_count)
    {
        unsigned long mask;
        unsigned long return_value;
        mask=1L<<(bit_count-1);
        return_value =0;
        while(mask!=0)
        {
            if(bit_file->mask==0x80)
            {
                bit_file->rack = getc(bit_file->file);
                if(bit_file->rack == EOF)
```

```
                    {
                        printf("\n\n\n");
                    }
                    if((bit_file->pacifier_counter++ &
2047)==0)
                    {
                    }
                }
                if(bit_file->rack & bit_file->mask)
                return_value |=mask;
                mask>>=1;
                bit_file->mask>>=1;
                if(bit_file->mask==0)
                bit_file->mask=0x80;
            }/* end of while */
    return(return_value);
    }


    FilePrintBinary(FILE *file,unsigned int code,int bits)
    {
    unsigned int mask;
    mask = 1<< (bits-1);
        while(mask!=0)
        {
            if(code & mask)
            fputc('1',file);
```

```
            else
            fputc('0',file);
             mask>>=1;
        }
    }
    /*        end    */



Initialize( int quality )
{
    int i;
    int j;
    double pi = atan( 1.0 ) * 4.0;

    for ( i = 0 ; i < N ; i++ )
        for ( j = 0 ; j < N ; j++ )
            Quantum[ i ][ j ] = 1 + ( ( 1 + i + j )  * quality );
    OutputRunLength = 0;
    InputRunLength = 0;
    for ( j = 0 ; j < N ; j++ ) {
        C[ 0 ][ j ] = 1.0 / sqrt( (double) N );
        Ct[ j ][ 0 ] = C[ 0 ][ j ];
    }
    for ( i = 1 ; i < N ; i++ ) {
        for ( j = 0 ; j < N ; j++ ) {
            C[ i ][ j ] = sqrt( 2.0 / N ) *
```

```
                    cos( pi * ( 2 * j + 1 ) * i / ( 2.0 * N ) );
            Ct[ j ][ i ] = C[ i ][ j ];
        }
    }
}



int InputCode( BIT_FILE *input_file)
{
    int bit_count;
    int result;

    if ( InputRunLength > 0 ) {
        InputRunLength--;
        return( 0 );
    }
    bit_count = (int) InputBits( input_file, 2 );
    if ( bit_count == 0 ) {
        InputRunLength = (int) InputBits( input_file, 4 );
        return( 0 );
    }
    if ( bit_count == 1 )
        bit_count = (int) InputBits( input_file, 1 ) + 1;
    else
        bit_count = (int) InputBits( input_file, 2 ) + (
  bit_count << 2 ) - 5;
```

```
  result = (int) InputBits( input_file, bit_count );
  if ( result & ( 1 << ( bit_count - 1 ) ) )
      return( result );
  return( result - ( 1 << bit_count ) + 1 );
}

/*
 * This routine reads in a block of encoded DCT data from a
compressed file.
 * The routine reorders it in row major format, and
dequantizes it using
 * the quantization matrix.
 */

ReadDCTData( BIT_FILE *input_file,int input_data[ N ][ N
])
{
   int i;
   int row;
   int col;

   for ( i = 0 ; i < ( N * N ) ; i++ ) {
        row = ZigZag[ i ].row;
        col = ZigZag[ i ].col;
        input_data[ row ][ col ] = InputCode( input_file ) *
                                   Quantum[ row ][ col ];

    }
```

```
}

/*
 * This routine outputs a code to the compressed DCT file.
For specs
 * on the exact format, see the comments that go with
InputCode, shown
 * earlier in this file.
 */



WritePixelStrip( FILE *output,unsigned char strip[ N ][
COLS ])
{
   int row;
   int col;

   for ( row = 0 ; row < N ; row++ )
        for ( col = 0 ; col < COLS ; col++ )
          putc( strip[ row ][ col ], output );
}

/*
  Applies inverse DCT and represents the image in the time
  domain
 */
```

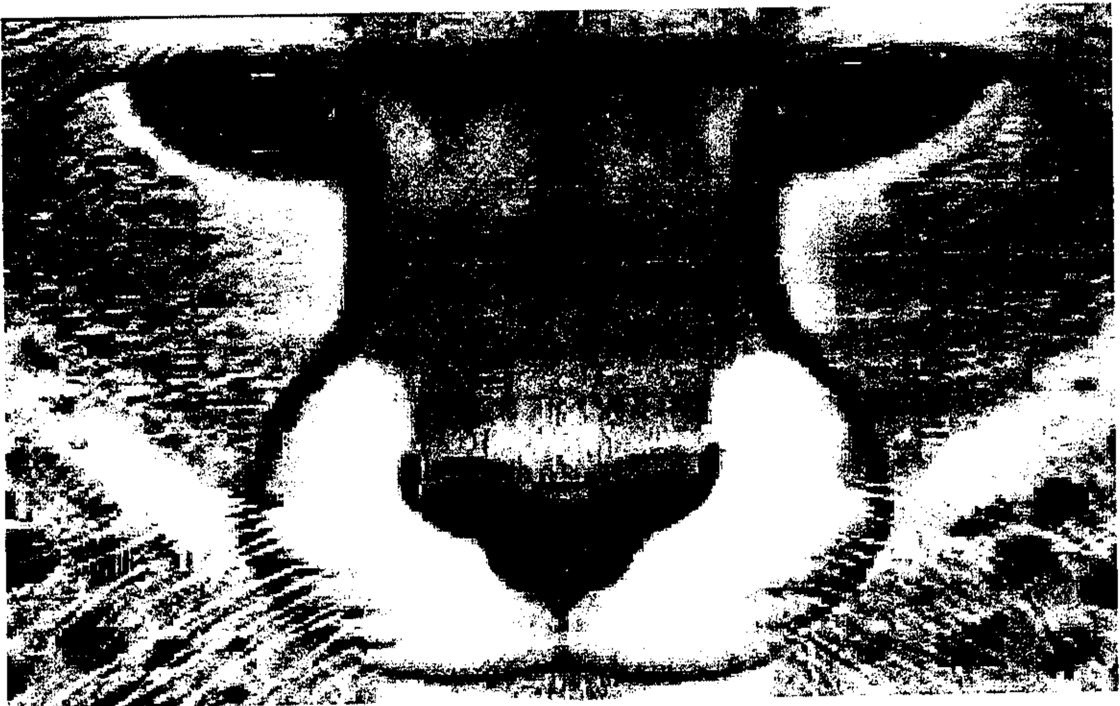
```
InverseDCT( int input[ N ][ N ],unsigned char *output[ N ])
{
   double temp[ N ][ N ];
   double temp1;
   int i;
   int j;
   int k;

/*  MatrixMultiply( temp, input, C ); */
    for ( i = 0 ; i < N ; i++ ) {
         for ( j = 0 ; j < N ; j++ ) {
            temp[ i ][ j ] = 0.0;
            for ( k = 0 ; k < N ; k++ )
                 temp[ i ][ j ] += input[ i ][ k ] * C[ k ][ j ];
         }
    }

/*  MatrixMultiply( output, Ct, temp ); */
    for ( i = 0 ; i < N ; i++ ) {
         for ( j = 0 ; j < N ; j++ ) {
            temp1 = 0.0;
            for ( k = 0 ; k < N ; k++ )
                 temp1 += Ct[ i ][ k ] * temp[ k ][ j ];
            temp1 += 128.0;
            if ( temp1 < 0 )
                 output[ i ][ j ] = 0;
```

```
            else if ( temp1 > 255 )
                output[ i ][ j ] = 255;
            else
                output[ i ][ j ] = (unsigned char) ROUND(
temp1 );
        }
    }
}


/*
 * The expansion routine reads in the compressed data from
the DCT file,
 * then writes out the decompressed grey scale file.
 */

ExpandFile( BIT_FILE *input,FILE *output)
{
   int row;
   int col;
   int i;
   int input_array[ N ][ N ];
   unsigned char *output_array[ N ];
   /*int quality=3;*/

   quality = (int) InputBits( input, 8 );
   printf( "\rUsing quality factor of %d\n", quality );
```

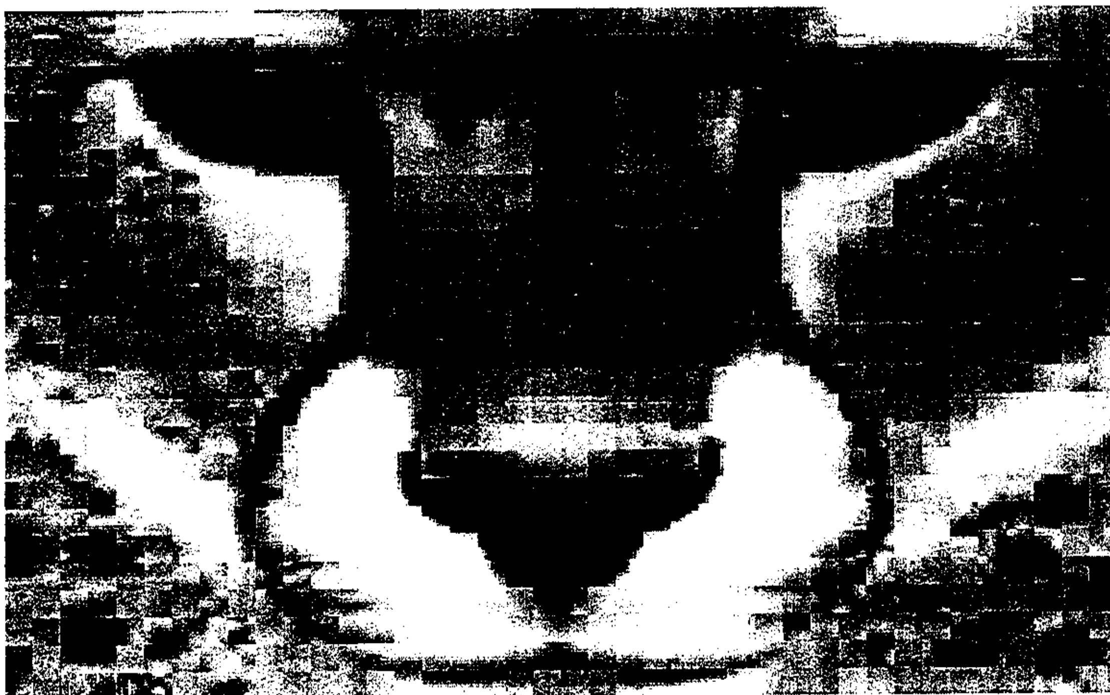
```
Initialize( quality );
for ( row = 0 ; row < ROWS ; row += N ) {
      for ( col = 0 ; col < COLS ; col += N ) {
          for ( i = 0 ; i < N ; i++ )
                output_array[ i ] = PixelStrip[ i ] + col;
          ReadDCTData( input, input_array );
          InverseDCT( input_array, output_array );
      }
      WritePixelStrip( output, PixelStrip );
   }
}
```

NAME	CHEETAH.RAW
TYPE	ORIGINAL IMAGE
SIZE	64000 bytes
WIDTH	320 pixels
HEIGHT	200 pixels
No of bits per pixel	8

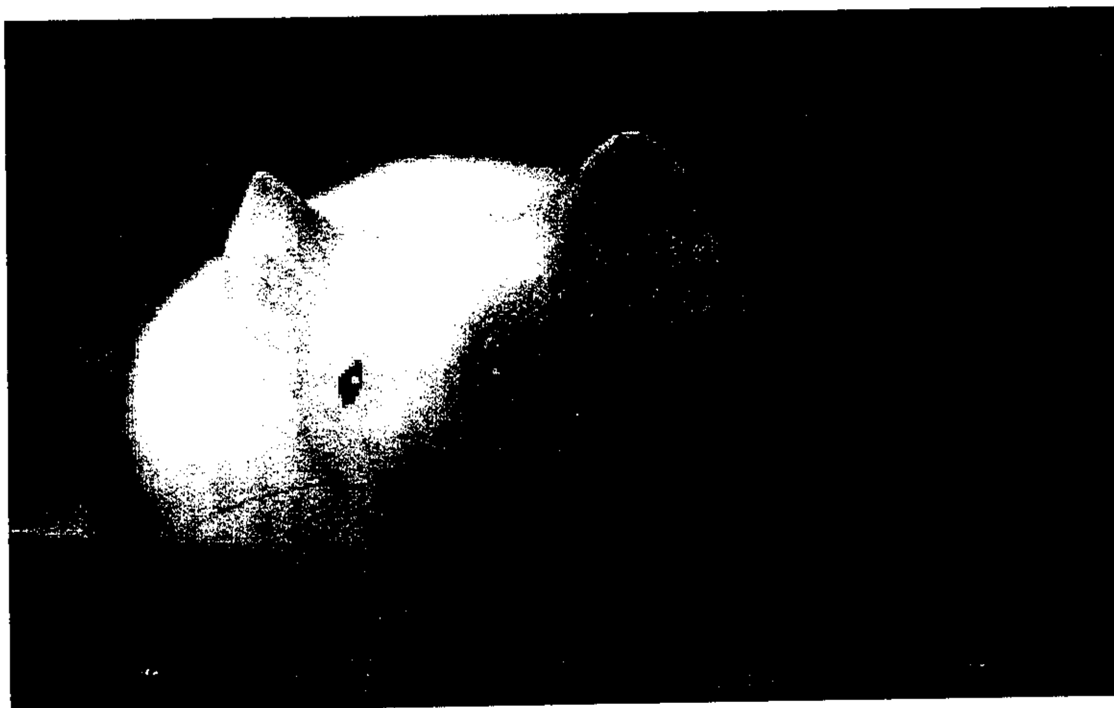
NAME	CHEET.RAW
TYPE	DECOMPRESSED IMAGE
WIDTH	320 pixels
HEIGHT	200 pixels
No of bits per pixel	8
SIZE	64, 000 bytes (decompressed) 10, 232 bytes (compressed)
QUALITY FACTOR	5
COMPRESSION RATIO	84 %



NAME	CHEET2.RAW
TYPE	Decompressed Image
SIZE	64, 000 bytes (decompressed) 5093 bytes (compressed)
QUALITY FACTOR	25
COMPRESSION RATIO	92 %

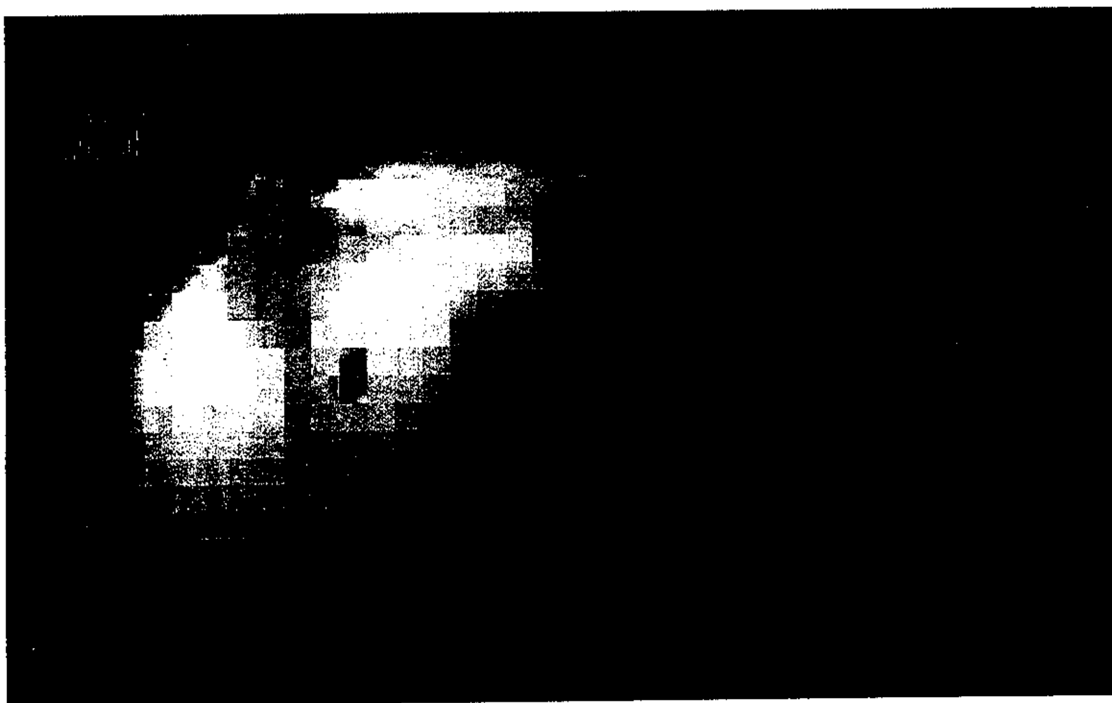


NAME	MOUSE.RAW
TYPE	ORIGINAL IMAGE
SIZE	64000 bytes
WIDTH	320 pixels
HEIGHT	200 pixels
No of bits per pixel	8



NAME	DEM1.RAW
TYPE	DECOMPRESSED IMAGE
WIDTH	320 pixels
HEIGHT	200 pixels
No of bits per pixel	8
SIZE	64, 000 bytes (decompressed) 5651 bytes (compressed)
QUALITY FACTOR	5
COMPRESSION RATIO	92 %

NAME	DEM2.RAW
TYPE	DECOMPRESSED IMAGE
WIDTH	320 pixels
HEIGHT	200 pixels
No of bits per pixel	8
SIZE	64, 000 bytes (decompressed) 4337 bytes (compressed)
QUALITY FACTOR	25
COMPRESSION RATIO	94 %



COMPARISON TABLE

File	Quality	Starting Size	Compressed size	Compression ratio
Cheetah.raw	5	64000	10,232	84%
Cheetah.raw	25	64000	5093	92%
Mouse.raw	5	64000	5651	92%
Mouse.raw	25	64000	4337	94%