1993

Image coding for monochrome and colour images.

Napiluon Petrus Shlimon

University of Windsor

Recommended Citation

Shlimon, Napiluon Petrus, "Image coding for monochrome and colour images." (1993). *Electronic Theses and Dissertations*. 2377.
https://scholar.uwindsor.ca/etd/2377

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

IMAGE CODING
FOR MONOCHROME AND COLOUR IMAGES

by

Napiluon Petrus Shlimon

A Thesis

Submitted to the Faculty of Graduate Studies
through the Department of Electrical Engineering
in partial fulfilment of the requirements
for the Degree of Master of Applied Sciences
at the University of Windsor

Windsor, Ontario, Canada
1993

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-83074-3

Canada

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Name [illegible]

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

[illegible] — Electronics & Electrical
SUBJECT TERM

0544
SUBJECT CODE

UMI

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture	0729
Art History	0377
Cinema	0900
Dance	0378
Fine Arts	0357
Information Science	0723
Journalism	0391
Library Science	0399
Mass Communications	0708
Music	0413
Speech Communication	0459
Theater	0465

EDUCATION

General	0515
Administration	0514
Adult and Continuing	0516
Agricultural	0517
Art	0273
Bilingual and Multicultural	0282
Business	0688
Community College	0275
Curriculum and Instruction	0727
Early Childhood	0518
Elementary	0524
Finance	0277
Guidance and Counseling	0519
Health	0680
Higher	0745
History of	0520
Home Economics	0278
Industrial	0521
Language and Literature	0279
Mathematics	0280
Music	0522
Philosophy of	0998
Physical	0523
Psychology	0525
Reading	0535
Religious	0527
Sciences	0714
Secondary	0533
Social Sciences	0534
Sociology of	0340
Special	0529
Teacher Training	0530
Technology	0710
Tests and Measurements	0288
Vocational	0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language	
General	0679
Ancient	0289
Linguistics	0290
Modern	0291
Literature	
General	0401
Classical	0294
Comparative	0295
Medieval	0297
Modern	0298
African	0316
American	0591
Asian	0305
Canadian (English)	0352
Canadian (French)	0355
English	0593
Germanic	0311
Latin American	0312
Middle Eastern	0315
Romance	0313
Slavic and East European	0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy	0422
Religion	
General	0318
Biblical Studies	0321
Clergy	0319
History of	0320
Philosophy of	0322
Theology	0469

SOCIAL SCIENCES

American Studies	0323
Anthropology	
Archaeology	0324
Cultural	0326
Physical	0327
Business Administration	
General	0310
Accounting	0272
Banking	0770
Management	0454
Marketing	0338
Canadian Studies	0385
Economics	
General	0501
Agricultural	0503
Commerce-Business	0505
Finance	0508
History	0509
Labor	0510
Theory	0511
Folklore	0358
Geography	0366
Gerontology	0351
History	
General	0578
Ancient	0579
Medieval	0581
Modern	0582
Black	0328
African	0331
Asia, Australia and Oceania	0332
Canadian	0334
European	0335
Latin American	0336
Middle Eastern	0333
United States	0337
History of Science	0585
Law	0398
Political Science	
General	0615
International Law and Relations	0616
Public Administration	0617
Recreation	0814
Social Work	0452
Sociology	
General	0626
Criminology and Penology	0627
Demography	0938
Ethnic and Racial Studies	0631
Individual and Family Studies	0628
Industrial and Labor Relations	0629
Public and Social Welfare	0630
Social Structure and Development	0700
Theory and Methods	0344
Transportation	0709
Urban and Regional Planning	0999
Women's Studies	0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture	
General	0473
Agronomy	0285
Animal Culture and Nutrition	0475
Animal Pathology	0476
Food Science and Technology	0359
Forestry and Wildlife	0478
Plant Culture	0479
Plant Pathology	0480
Plant Physiology	0817
Range Management	0777
Wood Technology	0746
Biology	
General	0306
Anatomy	0287
Biostatistics	0308
Botany	0309
Cell	0379
Ecology	0329
Entomology	0353
Genetics	0369
Limnology	0793
Microbiology	0410
Molecular	0307
Neuroscience	0317
Oceanography	0416
Physiology	0433
Radiation	0821
Veterinary Science	0778
Zoology	0472
Biophysics	
General	0786
Medical	0760

EARTH SCIENCES

Biogeochemistry	0425
Geochemistry	0996
Geodesy	0370
Geology	0372
Geophysics	0373
Hydrology	0388
Mineralogy	0411
Paleobotany	0345
Paleoecology	0426
Paleontology	0418
Paleozoology	0985
Palynology	0427
Physical Geography	0368
Physical Oceanography	0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences	0768
Health Sciences	
General	0566
Audiology	0300
Chemotherapy	0992
Dentistry	0567
Education	0350
Hospital Management	0769
Human Development	0758
Immunology	0982
Medicine and Surgery	0564
Mental Health	0347
Nursing	0569
Nutrition	0570
Obstetrics and Gynecology	0380
Occupational Health and Therapy	0354
Ophthalmology	0381
Pathology	0571
Pharmacology	0419
Pharmacy	0572
Physical Therapy	0382
Public Health	0573
Radiology	0574
Recreation	0575
Speech Pathology	0460
Toxicology	0383
Home Economics	0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry	
General	0485
Agricultural	0749
Analytical	0486
Biochemistry	0487
Inorganic	0488
Nuclear	0738
Organic	0490
Pharmaceutical	0491
Physical	0494
Polymer	0495
Radiation	0754
Mathematics	0405
Physics	
General	0605
Acoustics	0986
Astronomy and Astrophysics	0606
Atmospheric Science	0608
Atomic	0748
Electronics and Electricity	0607
Elementary Particles and High Energy	0798
Fluid and Plasma	0759
Molecular	0609
Nuclear	0610
Optics	0752
Radiation	0756
Solid State	0611
Statistics	0463

Applied Sciences

Applied Mechanics	0346
Computer Science	0984
Engineering	
General	0537
Aerospace	0538
Agricultural	0539
Automotive	0540
Biomedical	0541
Chemical	0542
Civil	0543
Electronics and Electrical	0544
Heat and Thermodynamics	0348
Hydraulic	0545
Industrial	0546
Marine	0547
Materials Science	0794
Mechanical	0548
Metallurgy	0743
Mining	0551
Nuclear	0552
Packaging	0549
Petroleum	0765
Sanitary and Municipal	0554
System Science	0790
Geotechnology	0428
Operations Research	0796
Plastics Technology	0795
Textile Technology	0994

PSYCHOLOGY

General	0621
Behavioral	0384
Clinical	0622
Developmental	0620
Experimental	0623
Industrial	0624
Personality	0625
Physiological	0989
Psychobiology	0349
Psychometrics	0632
Social	0451

APPROVED BY:

Dr. M. A. Sid-Ahmed

Dr. J. J. Soltis

Prof. P. H. Alexander

Dr. H. E. Toews

To my parents

Abstract

This work is an investigation of different algorithms to implement a lossy compression scheme. Special emphasis was focused on the quantization techniques. A CODEC (coder/decoder) based on a scheme proposed for standardization by a group known as JPEG (Joint Photographic Experts Group) was developed. Finally, a new decoding approach was developed, based on modifying the concepts of transition table used in compilers to break a binary string into variable length codes.

The JPEG algorithm works in sequential mode by dividing the image into small blocks of 8x8 pixels. Each block is compressed separately by processing it through an 8x8 Discrete Cosine Transform, Quantization, Run length and Huffman coding.

The two dimensional DCT was implemented by a fast 1-D DCT expanded into a 2-D DCT, using the row-column method.

Quantization is carried out by dividing the transformed block by the "JPEG scaling matrix" and rounding the results to the nearest integer. It was found to work well for a large number of images.

Four static Huffman code tables are used to convert the quantized DCT coefficients into variable length codes for both monochrome and colour images.

The algorithm is capable of obtaining varying compression ratios by simply changing the scaling factor of the "JPEG Quantization Matrix" . The bit rate achieved was in the range of 1 bit/pixel for images indistinguishable from the original. Higher compression ratios can be obtained at the cost of lower image quality.

ACKNOWLEDGEMENTS

I am greatly indebted to my supervisor, Dr. M. A. Sid-Ahmed for his valuable advice, support, patience, encouragement, and tough criticism during my thesis research.

I would like to take this opportunity and express my appreciation to Dr. J. J. Soltis. His valuable suggestions were a great help in the preparation of this thesis. I extend my sincere thanks to other members of my committee, Professor P. H. Alexander and Dr. H. E. Toews for their advice and suggestions.

I am very grateful to my friend Jun Cao. His valuable discussions and support helped me to pass crucial times. Thanks also goes to my friend Emmanuel Youkhannis for lending me his computer, and special thanks to Laura Thomas for her help in revising the grammatical structure of this thesis.

I am also grateful to my parents and brothers for their great patience and support during my work.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF APPENDICES

CHAPTER ONE

INTRODUCTION

Image compression is one of the major areas in digital image processing. The goal of image compression is to reduce the bit rate required to represent an image by removing the redundancy.

There are two types of compression techniques: the first one recovers the compressed image completely and is called *reversible (or error-free) compression*, while the other recovers the image features without duplicating the original values of image data and is called *irreversible (or lossy) compression*.

During the last two decades, the demand for presenting images in digital form has increased due to the significant reduction in the cost of image scanners, photographs, and printed texts. As a result, converting images into a digital form has become easy and cheap.

In addition to the mentioned advantage, digital presentation allows visual information to be manipulated easily. This fact, combined with the development of fast computers in recent years, has resulted in the use of digital imaging in different fields such as astronomy, remote sensing, and the medical field.

Despite the mentioned advantages, there is a potential problem with digital images. A large number of bits are required to represent them. This representation contains a significant amount of redundancy. The image compression field aims to take advantage of this redundancy in order to reduce the number of bits required to represent an image. The result is significant savings in the memory needed for image storage or in the channel capacity required for image transmission.

The main objective of this research was to develop a CODEC (coder/decoder) for the compression and decompression of monochrome and colour images. This CODEC should be able to compress images with varying compression ratios in the range of 8-16:1 or to 1 bit/pixel. With these compression ratios, the reconstructed image should be indistinguishable from the original; however, higher compression ratios can be achieved with less quality images. The developed algorithm was based on a scheme presented by a standardization group known as JPEG (Joint Photographic Experts Group) [1].

1.1 Digital Image Generation and Presentation

The photosensitive devices are the main source for generating digital images, such as TV camera tubes, flying spot scanners, and microdensitometers [2]. In these devices, a reflected light from a real image is received and converted into electrical charge and the amount of this electrical charge is proportional to the amount of light reflected from the image. The electrical charge is used to produce a time-varying signal (analog signal) corresponding to a sequential scan of the image or scene. In order to generate a digital image, the analog signal produced is first sampled at discrete locations, where these samples are called pixels (or picture elements). Secondly, these continuous-valued sampled points are quantized to one of the levels in order to generate a digital image. Digital images are represented by a two dimensional array. The elements of this array are called pixels, and the values of these pixels are one of the quantization levels generated from the previous step. The number of quantization levels required to represent an image is dependent on the application. For bilevel images, only two levels (1 bit/pixel) are required since each pixel is either black or white. For monochrome images it is common to use 8 bits/pixel (256 levels), and for colour images, 15 or 24 bits/pixel are needed.

1.2 Why Image Compression?

In order to represent images adequately, the number of pixels has to be increased, which results in higher resolution. The increased volume of data required to present an image has provoked the need for reducing the size of storage or channel width required to store or transmit an image respectively.

The following examples taken from [3], show the amount of storage and time required to store and transmit the following types of images:

1. A low-resolution, TV quality, three coloured video image (512x512 pixels,8 bits/pixel), requires approximately 6 Mbits.
2. A 24x36-mm(35mm) negative photograph, scanned at 12 μm: 3000x2000 pixels, three coloured, 8 bits/pixel, requires approximately 144 Mbits.
3. A transmission of low-resolution (512x512x8 bits/pixel), colour video image over telephone lines, using a 9600 baud (bits/s) modem takes around 11 minutes for the transmission of just a single image, which is unacceptable for most applications.

1.3 Overview of Data Compression Techniques

The compression ratio is defined as the ratio of the original image size to the compressed image size. The compression techniques can be categorized as either reversible or irreversible based on whether or not there have been errors introduced to the reconstructed image data.

Reversible compression, which is the subject of Chapter 2, completely recovers the compressed image, and is identical to the original image on a pixel-by-pixel basis. This technique is ideal; however, the amount of compression is slightly low compared to irreversible compression. This kind of compression is widely used in medical imaging, such as x-ray images, where the minute details play an important role in obtaining proper diagnosis.

Irreversible compression, which is the subject of Chapter 3, recovers the compressed image, but the reconstructed image contains some degradations. However, the compression ratios obtained are much higher compared to the ones obtained using reversible compression. Due to the limitations of the human visual system sensitivity, these degradations may not be visually apparent. Therefore, this kind of compression is useful in certain applications where minute details are not very important, such as TV images, remote sensing, and satellite imagery.

The majority of reversible compression techniques are based on the principles of information theory [4], using an information measure called entropy. The principle of entropy states that more probable data carry less information and less probable data carry more information. Several methods have been developed to reduce the data rate to the data source entropy. One procedure was discovered by Huffman and is known as Huffman Coding [5]. Although Huffman coding is theoretically optimum, a more efficient method called Arithmetic Coding [6] can achieve optimum performance. Run-length Coding [7] is another reversible method that works efficiently in compressing bilevel images due to the high correlation between adjacent pixels.

The irreversible compression techniques mainly include three components: orthogonal transformation, quantization, and encoding. Orthogonal transformations, such as Karhunen-Loeve Transform (KLT), Discrete Cosine Transform (DCT), and Walsh-Hadamard Transform (WHT) aim to map a set of correlated pixels into a set of uncorrelated coefficients and preserve most of the energy (the sum of the squared pixel values) in fewer numbers of coefficients. The KLT (Karhunen-Loeve transform) is the optimal transform [8], but it requires a higher number of computations and suffers from a lack of fast algorithms. The DCT (Discrete Cosine Transform) [9] is the most popular transform because of its performance that approximates the KLT in optimality and in the high speed of its algorithms. The transform coefficients are floating point numbers. They have a wide dynamic range and they occupy a considerable amount of memory. In order to reduce this dynamic range, quantization is needed. Quantization is considered the only step where losses in image features are incorporated. Following quantization, many small amplitude coefficients are reduced to zero, which results in a presentation that can be encoded and further compressed by reversible compression.

1.4 Image Quality Evaluation

No distortion is introduced to images which are compressed using reversible compression techniques. Therefore, image quality evaluation applies only to those images which are compressed with irreversible compression techniques. The distortion measure is necessary to evaluate the quality of reconstructed image. The mean-square-error (MSE) is the traditional method, but human visual perception of distortion does not resemble the MSE [10]. MSE can be used as a first pass measure, but the final determination of the quality is currently made by human observation.

Statistical methods are used to analyze the results of the objective test. One of these methods is the ROC (Receiver Operating Characteristic) [11], widely used in medical imaging. This method is advantageous because it is independent of the reader's biases or detection criteria.

Another method known as Kappa Statistics [12,13] measures the observer agreement minus the factor of chance expectation. It is the most popular method.

A third technique uses paired film readings and rank analysis to establish a compression ratio threshold above which the quality degradation is detectable. This threshold can be a measure of quality in such a way that whenever an image is compressed with ratios below the threshold, it still retains all image features.

1.5 Image Compression Standardization Groups

The main purpose of image compression standardization is to produce an international CODEC for the compression and decompression of images. As a result, the cost of video compression equipment will be reduced and the problem of equipment inter-operability will be solved. The efforts toward this direction can be summarized in three main groups[3]:

1. CCITT (Consultive Committee of International Telephone and Telegraph): this group worked on developing a CODEC for bilevel images, such as texts and documents. These efforts succeeded in developing such a CODEC, but it can not be used for monochrome and colour images. This CODEC is widely used in facsimile machines.
2. JPEG (Joint Photographic Experts Group): this group was formed in 1986 under the auspices of ISO (International Standards Organization) and CCITT. This group worked on developing an international CODEC for continuous-tone, still frame, monochrome and colour images. A detailed description of a JPEG CODEC will be the subject of Chapter 4.
3. MPEG (Moving Pictures Experts Group): this group has been working since 1988 under the auspices of ISO to develop a standard CODEC for storage and retrieval of moving images and sound. The MPEG standard aims to be a general purpose technique for applications in electronic publishing, education, and games.

1.6 Thesis Organization

This thesis is organized as follows: Chapter 2 presents the background and underlying principles of reversible (or error-free) compression techniques. The concepts of Entropy, Entropy Coding, Run Length, and Lemple-Ziv Coding are discussed. In chapter 3, the principles of several lossy (or irreversible) compression techniques are presented. The concepts of Orthogonal Transforms, Block Cosine Truncation, Quantization, and image compression with Block Truncation are described. Chapter 4 presents the complete description of the JPEG Baseline Compression Algorithm for monochrome and colour images. An example block of the LENA image was used to illustrate the concepts of each step in the algorithm. The required time for coding of monochrome and colour images, the compression ratios, and the reconstructed images after compression are included. Chapter 5 summarizes the results of this work and includes concluding remarks and some suggestions for future work. Three appendices are also included: Appendix A includes the Huffman code tables used in the JPEG compression algorithm for monochrome and colour images; Appendix B includes a source code in "C" for implementing the JPEG compression algorithm for monochrome images, while Appendix C includes a source code in "C" for implementing the JPEG compression algorithm for colour images.

CHAPTER TWO

ERROR-FREE COMPRESSION TECHNIQUES

Some applications require that the reconstructed image after compression be recovered completely. Compression techniques that have this ability are called error-free (or lossless, or reversible) compression techniques. These methods are widely used in the medical field for compressing x-ray images. Any error introduced to the x-ray image may lead to diagnostic inaccuracy. The compression ratios achieved using these techniques are smaller than those obtained using lossy compression techniques.

The majority of lossless compression methods utilize the statistical characteristics of an image, such as a histogram, pattern repetition, and entropy. These quantities are used to reduce the redundancy in an image. The compression ratios of error-free methods for medical images fall in the range of 3:1 [15].

Several lossless methods have been developed to reduce the bit rate. These methods rely on the principles of information theory [4] in order to reduce the data rate to the data source entropy. In the following sections, the concepts of Entropy and several of the most popular coding methods are discussed.

2.1 Entropy

Entropy can be defined as the average amount of information per symbol coming from a source[4]. The bit is the unit used to measure the entropy, and the entropy is calculated using the following formula.

$$H(S) = -\sum_{i=1}^{n} P(S_i)\ \log_2(S_i) \qquad \textit{bits/symbol} \qquad ....(2.1)$$

where: $H(S)$ is the entropy.

$P(S)$ is the probability of occurrence of symbols.

n is the total number of source symbols.

Lossless image compression uses entropy coding, such as Huffman or Arithmetic Coding, to generate variable length codewords. These codewords are assigned to source symbols in such away that the shortest code represents the most probable symbol and vice versa. The average length of these codewords can be calculated using the following formula:

$$L_{av} = \frac{1}{n} \sum_{i=1}^{n} P(S_i)\ L_i \qquad \textit{bits/symbol} \qquad ....(2.2)$$

where: L_{av} is the average length.

L_i is the codeword length.

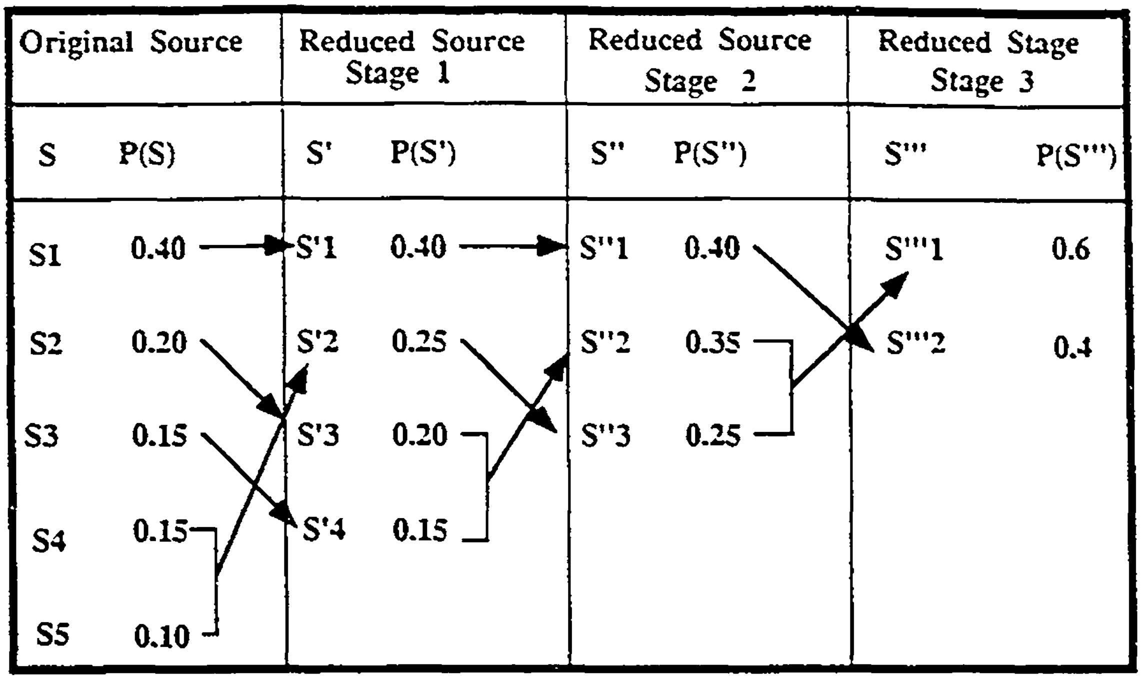
It is important to note that the best lossless compression method can achieve a reduction in the bit rate less than or equal to the entropy rate. The following sections present some of the most popular entropy codings.

2.2 Huffman Coding

Huffman codes are variable length, uniquely decodable codes or prefix codes[5]. They achieve the optimal requirement which minimizes the average length of the encoded messages. For a finite set of inputs X_i, $i=1,\ldots,N$, the code generation procedure can be summarized as follows:

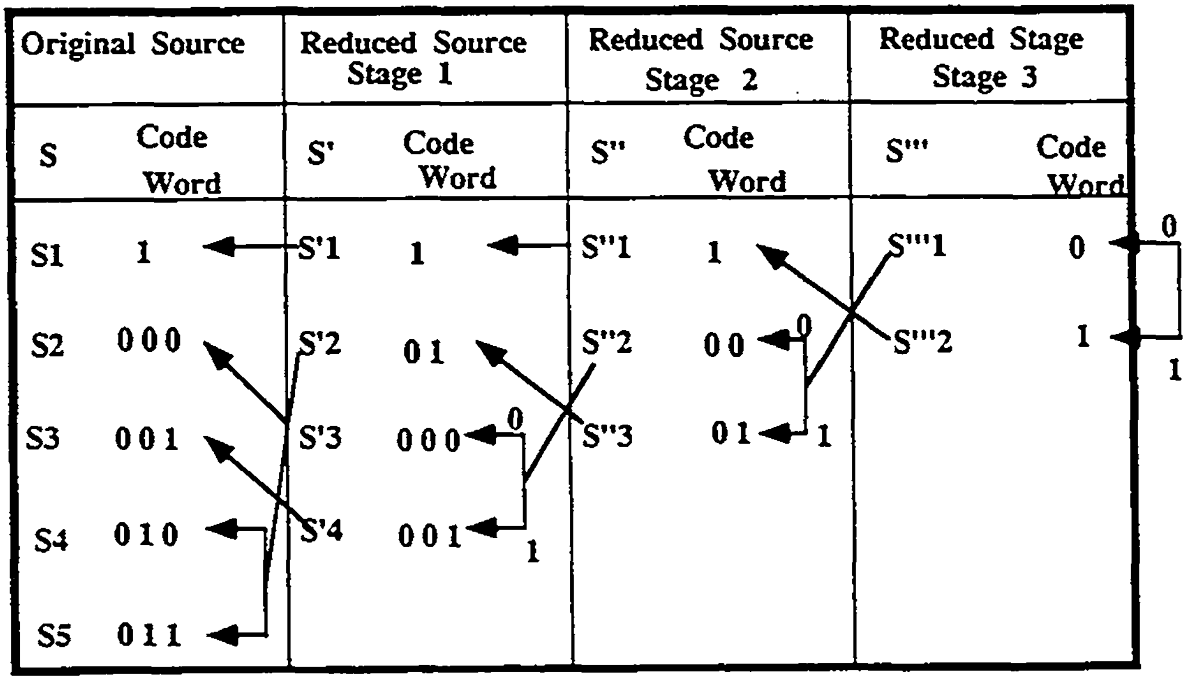
1. Find the probability of each symbol and order them from the highest probable symbol to the lowest probable symbol.
2. Sum the least two probabilities and reorder the new probabilities again from the highest to the lowest probable symbol.
3. Repeat Step 2 until the list contains only two elements.
4. The code word generation starts with assigning '0' and '1' to the last two elements obtained from Step 3. The codewords for the previous reduced stage are found by appending '0' and '1' to the codeword corresponding to the two least probable symbols. This process is continued until Huffman codes for the original source symbols are found.

The coding process merely substitutes these codes for the symbols, and concatenates them to perform the final coded message. The expected bit rate should be less than the original bit rate. Figure 2.1 shows an illustrative example of Huffman code generation. The average length is 2.2 bits/symbol, and the entropy rate is 2.15 bits/symbol.



Original Source		Reduced Source Stage 1		Reduced Source Stage 2		Reduced Stage Stage 3	
S	P(S)	S'	P(S')	S"	P(S")	S'''	P(S''')
S1	0.40	S'1	0.40	S"1	0.40	S'''1	0.6
S2	0.20	S'2	0.25	S"2	0.35	S'''2	0.4
S3	0.15	S'3	0.20	S"3	0.25		
S4	0.15	S'4	0.15				
S5	0.10						

a)Source reduction process



Original Source		Reduced Source Stage 1		Reduced Source Stage 2		Reduced Stage Stage 3	
S	Code Word	S'	Code Word	S"	Code Word	S'''	Code Word
S1	1	S'1	1	S"1	1	S'''1	0
S2	0 0 0	S'2	0 1	S"2	0 0	S'''2	1
S3	0 0 1	S'3	0 0 0	S"3	0 1		
S4	0 1 0	S'4	0 0 1				
S5	0 1 1						

b) Codeword construction process

Figure 2.1: Example Huffman code generation.

2.2.1 Fast Huffman-Type Code Generator

The generation of Huffman codes, using the previous method, requires a large memory size if the source contains a large number of symbols. In order to overcome the complications and memory problems associated with this method, a simple and fast algorithm [16] is presented. This algorithm consists mainly of two processes: contraction and expansion.

a)Contraction Process

The contraction process is basically a series of contraction stages. The two least probable symbols are combined into a new symbol. The new symbol is inserted above the symbols whose probabilities are equal to or less than the new probability. During this time, the number of symbols that have fallen below this new symbol is recorded and named as an expansion index E(I). The algorithm for the contraction process is shown below, where:

I	represents the involved contraction stage;
TEMP	represents the temporary variable storing the sum of the two lowest probabilities at each contraction stage;
P[Si]	represents the probability of occurrence of symbol Si;
N	is the number of symbols in the source;
LP	represents the variable that gives the location of the symbol whose probability has to be compared with TEMP;
M	is a variable counting how many symbols have fallen blow the new symbol at each contraction stage;
MCL	is the maximum code length.

The algorithm is:

```
1.    Initialize   I=1, LP=N-2, M=0, TEMP=P[N]+P[N-1];

2.    if(TEMP >= P[LP]) then

                P[LP+1]=P[LP];
                increment M;
                decrement LP;

                if(LP = 0)then go to step 6
                else  go to step 2
      else
                P[LP+1]=TEMP;
                EI[I]=M;
                increment I;

                if(I = N-2) then go to end
      end if

3.    TEMP=P[N-I]+P[N-I+1];

4.    if( LP = N-I) then  decrement LP
      else
                decrement M;

                if( M > 0 )then
                      for J=N-I to LP+2 do
                            P[J]=P[J-1];
                end if
      end if

5.    go to step 2

6.    EI[I]=M,  decrement M, increment  I

7.    if( I = N-2) then go to end,       else go to step 6

8.    end
```

b)Expansion Process

The expansion process generates the codewords by generating the code lengths first. These code lengths are used in the next step to generate the codewords. For the code lengths generating step, the expansion indices generated from the contraction process are used to generate a set of new values. These new values are termed A(J), J=1...Y. A(J) is the codeword; J is the length of codeword; and Y is the maximum length. The algorithm for the expansion process is shown below :

```
1.    Initialize A[1]=1, MCL=2, A[MCL]=2, I=N-3;

2.    While( I > 0 )  do
          if(EI[I] < A[MCL]) then

              decrement  A[MCL];
              increment  MCL;
              A[MCL]=2;

          else
              decrement A[MCL-1];
              increment A[MCL] by 2;
          end if

          decrement I;

      end while

3.    end
```

2.3 Arithmetic Coding

Arithmetic Coding [6] is another coding method that belongs to entropy coding. This method is considered to be more efficient and gives more compact representation than Huffman Coding. The difference between Huffman coding and arithmetic coding can be expressed as follows: Arithmetic Coding takes a string of symbols (such as one line of image) and a code is generated for the whole string as one piece; Huffman coding generates the code for each symbol in that string first. The encoding process is a concatenation of these codes to generate the code for the whole string.

During the Arithmetic Coding process, a message is represented by an interval of real numbers between 0 and 1. As the message becomes longer, the interval needed to represent it becomes smaller, and the number of bits needed to specify that interval grows [17]. Successive symbols of the message reduce the size of the interval in accordance with the symbol probabilities generated by the model. The more likely symbols reduce the range by less than the unlikely symbols, and therefore add fewer bits to the message. An illustrative example presented in [6] is included here to illustrate the concepts of the Arithmetic Encoding process.

Assume that the string needed to be coded consists of four symbols "a b c". The four symbols have relative frequencies and their probabilities are shown in Table 2.1.

Symbol	Codeword	Probability	Cumulative Prob.
a	0	.100	.000
b	1 0	.010	.100
c	1 0	.001	.110
d	1	.001	.111

Table 2.1: Example arithmetic coding

As shown in figure 2.4, the coding process starts with subdividing the unit interval into intervals. The code point at the edges of each interval represents the sum of probabilities of the preceding symbols. The width of the interval to the right of each code point corresponds to the probability of that symbol. The code points are represented as a binary fraction corresponding to the fractional values of the cumulative distribution function assigned to each symbol. Figure 2.5 shows the continuation of the encoding process. The symbol "a" has been encoded to [0.0.1), ("0" is included in the interval). The next step is to subdivide this interval into the same proportions as the original unit interval; thus, the subinterval assigned to the second "a" is [0.0.01). For the third symbol "b", the subinterval will be [0.001.0.0011). Note that each of the two leading zeros in the binary representation come from the codeword (Table 2.1) of the two symbols "a" which precede the symbol "b". For the fourth "c", the corresponding subinterval is [0.0010110,0.0010111). The final code assigned to the string "a b c" is given by "0010110".

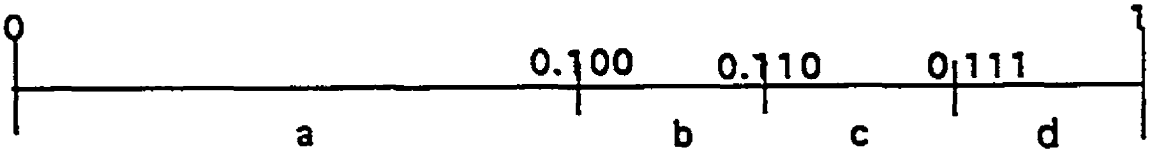


Figure 2.2: Codewords of table 2.1 as points on unit interval.

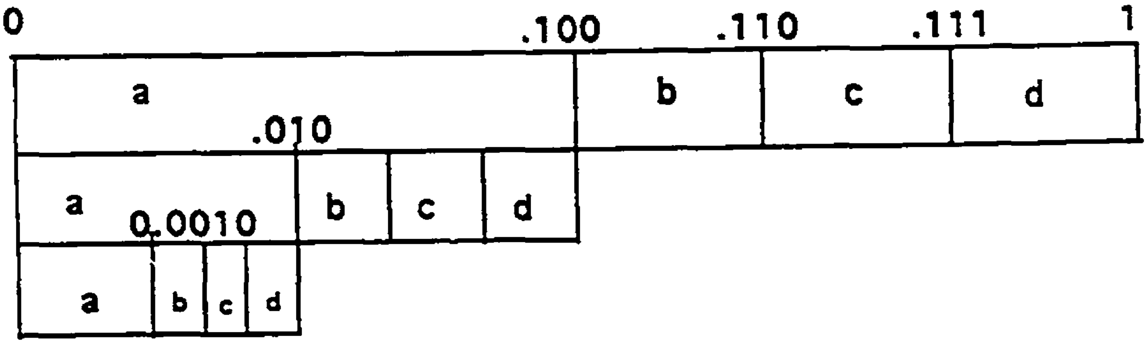


Figure 2.3: Successive subdivision of unit interval for code of Table 2.1 and data string "aabc".

Arithmetic Coding can achieve a code rate approximately equal to the entropy without the condition that the probabilities are an integral power of one half. In most situations, arithmetic coding can replace Huffman coding and it gives better performances in both coding speed and coding rate.

2.4 Run-Length Coding

Run-length Coding is a mapping of the sequence of the same symbol in a string of data into a "count field" and "identifier" of the repeated character [18]. Run-length coding is widely used with data that contains a long string of the same symbol. This kind of redundancy can be effectively reduced by run-length coding.

Run-length coding was found to be very effective in compressing bilevel images, such as texts, and documents, and the reason is obvious: bilevel images contain long runs of concatenated black and white pixels. On the other hand, this type of coding is rarely applied directly to noisy or complicated images. Instead, it can be included in the final stages of a compression scheme to obtain a presentation that can be further compressed using Entropy Coding. A good example of this is the JPEG compression scheme. Hence, the run-length coding is the step after quantization.

The effect of run length coding in reducing the bit rate is shown below, where the input data is assumed to be one line of a bilevel image containing 16 pixels of black and white.

1- *B B B B B B W W W W W W B B W W*

2- (*B*,6) (*W*,6) (*B*,2) (*W*,2)

The second presentation requires fewer bits when the run length is very long.

2.5 Lempel-Ziv Coding

The Lempel-Ziv Coding [19] is defined as a mapping of a string of data into fixed-length codes. The procedure for generating these codes requires no prior information about the input data statistics and can be done in one pass.

The algorithm is assumed to be adaptive because it starts with an empty table of symbol strings, and builds the table during both compression and decompression. Therefore, there is no need to store or transmit the codeword table for decoding.

One of the simplest compression procedures is the Lempel-Ziv-Welch (LZW) method [20]. The LZW algorithm compresses data by generating a translation table, or string table, and mapping the input strings to fixed length codes. A common table size used is a modest 2 to power 16, which is represented by a 16 bit binary word. One of the main properties of the LZW string is the prefix property. The prefix property states that each string in the table has a prefix string also in the table. For example, if a string wK composed of string w and character K is in the table, then w is also in the table.

CHAPTER THREE

LOSSY COMPRESSION TECHNIQUES

Lossy (or irreversible) compression maintains the image features, but the reconstructed image contains some degradations. This means that the recovered image is not numerically identical to the original image. These degradations may not be visually apparent.

Despite the mentioned disadvantage, the compression ratio achieved using lossy techniques is higher than the one achieved using lossless techniques. However, the higher compression achieved is at the expense of more degradations added to the compressed image.

Lossy compression techniques include more than one stage. These stages might not contribute to the compression directly. Instead, they provide a representation that can be compressed further using lossless compression techniques. The block diagram for a typical lossy compression technique is shown in figure 3.1.

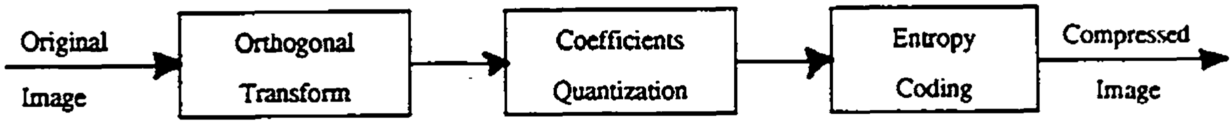


Figure 3.1: Block diagram of a lossy compression technique.

The first stage is the orthogonal transformation, such as DCT, WHT, or KLT. This first stage serves to map correlated pixels into uncorrelated coefficients and to compact most of the energy in the low frequency components. As a result, the low frequency components of the transformed coefficients will have larger amplitudes, while the high frequency components will have smaller amplitudes. The stage following orthogonal transformation is quantization. This stage is the focal point of compression. Quantization techniques aim to reduce the dynamic range of the transform coefficients which are floating point numbers by preserving the large amplitude coefficients. At the same time, the small amplitude coefficients can be reduced to zero without introducing much degradation to the image. The data obtained from the quantization step is represented in such a way that is suitable to be further compressed using lossless compression, mainly entropy coding techniques. A complete description of the most popular lossy compression techniques is presented in the following sections.

3.1 Orthogonal Transforms

Many transforms have been used for data compression and each one has its own characteristics [21]. The Discrete Fourier Transform, implemented by the Fast Fourier Transform (FFT) was the first one to be applied in data compression[22]. The KLT [8] is the optimal transform, but it requires heavy computation. The DCT [23] has gained wide popularity because of its performance, which approximates KLT optimality, and because of the high speed of its algorithms. The performance of different transforms is shown in figure 3.2 [15], where the NMSE is the normalized mean square error and is defined as:

$$NMSE = \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} |f(i,j) - f'(i,j)|^2}{\sum_{i=1}^{N} \sum_{j=1}^{N} |f(i,j)|^2} \quad \text{........ (3.1)}$$

where f(i,j) are the original image pixels, and f'(i,j) are the reconstructed image pixels.

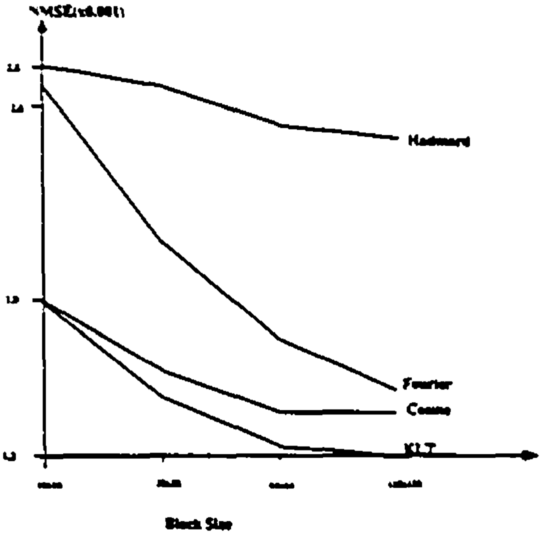


Figure 3.2: Comparison of different transforms. The normalised MSE (with 25% retained coeff.) is plotted against the block size.

3.1.1 Discrete Cosine Transform (DCT)

The discrete cosine transform was first discovered by N.Ahmed [9], and is recognized as the best way to encode digital images because of its performance in energy packing capabilities; that is, fewer coefficients will have large magnitudes while other coefficients will have small amplitudes. The high speed of the two dimensional DCT algorithms also makes it very desirable. The 1-D DCT of a sequence of data x(n) is defined as:

$$X(k) = \frac{2\epsilon_k}{N} \sum_{n=0}^{N-1} X(n) \cos \frac{\pi(2n+1)k}{2N} \quad \text{........ (3.2)}$$

$$k=0,\ldots,N-1$$

and the inverse DCT can be expressed as:

$$X(n) = \sum_{k=0}^{N-1} \epsilon_k X(k) \cos \frac{\pi(2n+1)k}{2N} \ldots\ldots\ldots\ldots (3.3)$$

$$n=0,\ldots,N-1$$

Where : N is the total number of input data

$$\epsilon_k = \frac{1}{\sqrt{2}} \quad for \quad k=0$$

$$= 1 \quad otherwise$$

The fast algorithm for implementing 1-D DCT adopted here was presented in [24]. First, the scaling factors are assumed to be absorbed in X(k) and taken as unity. It has been proven in [23] that for better efficiency, the input sequence should be rearranged as the following:

$$xrs(n)=x(2n) \quad \text{and} \quad xrs(N-n-1)=x(2n+1) \ldots\ldots\ldots\ldots (3.4)$$

$$n=0,\ldots\ldots N/2-1$$

Then the DCT is converted into a DFT with a variable phase angle by substituting (3.4) into (3.2) to get:

$$X(k) = \sum_{n=0}^{N-1} xrs(n) \cos \frac{\pi(4n+1)k}{2N} \ldots\ldots\ldots\ldots (3.5)$$

$$k=0,\ldots,N-1$$

The vector Radix 2 approach is utilized here to compute the odd and even indexed input elements of k separately. For the even indexed elements:

$$X(2k) = \sum_{n=0}^{N/2-1} xrs(n)\cos\frac{\pi(4n+1)k}{N} - \sum_{n=0}^{N/2-1} xrs(n+N/2)\cos\frac{\pi(4n+2N+1}{N}$$

$$= \sum_{n=0}^{N/2-1} [xrs(n)+xrs(n+N/2)]\ \cos\frac{\pi(4n+1)k}{2(N/2)} \ldots\ldots\ldots\ldots(3.6)$$

(3.6) is recognized to be as N/2 point DCT. The odd indexed elements of the input can be expressed as:

$$X(2k+1) = \sum_{n=0}^{N/2-1} [xrs(n)-xrs(n+N/2)]\cos\frac{\pi(4n+1)(2k+1)}{2N} \ldots\ldots(3.7)$$

Since:

$$\cos[(2k+1)\phi] = 2\cos\phi\cos(2k\phi) - \cos[(2k+1)\phi] \ldots\ldots\ldots\ldots\ldots\ldots(3.8)$$

$$\textit{with}$$

$$\phi = \frac{\pi(4n+1)}{2N}$$

Substituting (3.8) into (3.7)we get:

$$X(2k+1) = \sum_{n=0}^{N/2-1} 2[xrs(n)-xrs(n+N/2)]\cos\frac{\pi(4n+1)k}{2N}\ \cos\frac{\pi(4n+1)k}{2(N/2)}$$

$$- \sum_{n=0}^{N/2-1} [xrs(n)-xrs(n+N/2)]\cos\frac{\pi(4n+1)(2k-1)}{2N} \ldots\ldots(3.9)$$

The latter term is recognized as X(2k-1). This means that the odd indexed components X(2k+1) can be recursively computed from a N/2 point DCT and X(2k-1). When k=0, X(-1) is considered to be equal to X(1), and X(1) is used to compute X(3) and so on. The flowgraph for 1-D DCT FCT with N=8 is shown in figure (3.3).

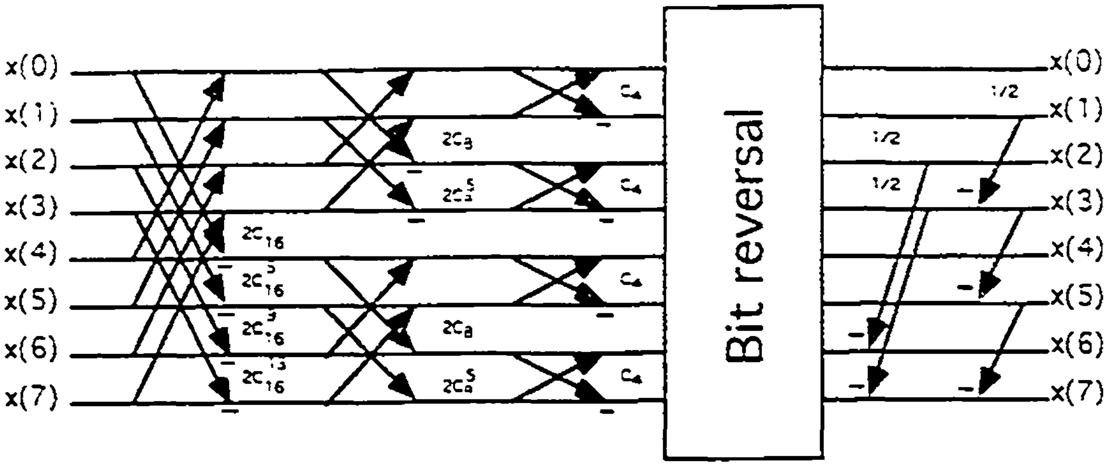


Figure 3.3: Flowgraph for N=8 1-D Fast Discrete Cosine Transform.

3.1.1.1 Two Dimensional Discrete Cosine Transform

The two dimensional DCT of a sequence of data X(n1,n2) can be expressed as:

$$X(k_1,k_2) = \frac{2\epsilon_{k1}\epsilon_{k2}}{\sqrt{N_1N_2}} \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1,n_2) \cos\frac{\pi(2n_1-1)k_1}{2N_1} \cos\frac{\pi(2n_2+1)k_2}{2N_2} \qquad \text{......(3.10)}$$

The inverse 2-D DCT is expressed as:

$$x(n_1,n_2) = \frac{2}{\sqrt{N_1N_2}} \sum_{k_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} \epsilon_{k1}\epsilon_{k2} \cos\frac{\pi(2n_1+1)k_1}{2N_1} \cos\frac{\pi(2n_2+1)k_2}{2N_2} \qquad \text{.......(3.11)}$$

The separability property of DCT is adopted here to compute the two dimensional DCT by a series of one dimensional transforms. As a result, equation (3.10) can be rewritten in the following form [23]:

$$X(k_1,k_2)=\sqrt{\frac{2}{N_1}}\epsilon_{k1}\sum_{n_1=0}^{N_1-1}\left[\sqrt{\frac{2}{N_2}}\epsilon_{k2}\sum_{n_2=0}^{N_2-1}x(n_1,n_2)\cos\frac{\pi(2n_2+1)k_2}{2N_2}\right]\cos\frac{\pi(2n_1+1)k_1}{2N_1}\;.....(3.12)$$

The inner summation is the N_2 -point one dimensional DCT of the rows of the $N_1 x N_2$ matrix, while the outer summation represents the N_1 -point one dimensional DCT of the columns of the semi-transformed matrix. This implies that 2-D DCT can be implemented by N_1 (N_2 *points DCT'S*) along the rows of ($N_1 x N_2$) matrix, followed by N_2 (N_1 *points DCT'S*) along the columns obtained after the row transformation as shown below:

$$\begin{bmatrix} x(0,0)\,.......\,x(0,N_2-1) \\ x(1,0)\,.......\,x(1,N_2-1) \\ .... \\ x(N_1-1,0)\,..\,x(N_1-1,N_2-1) \end{bmatrix} \begin{array}{c} (N_2-1)\ \textit{point} \\ 1\text{-}D\ \textit{DCT along} \\ \textit{the rows} \end{array} \begin{bmatrix} \bar{x}(0,0)\,.......\,\bar{x}(0,N_2-1) \\ ... \\ ... \\ \bar{x}(N_1-1,0)\,..\,\bar{x}(N_1-1,N_2-1) \end{bmatrix}$$

$$\begin{array}{c} (N_1-1)\ \textit{point} \\ 1\text{-}D\ \textit{DCT} \\ \textit{along the columns} \end{array}$$

$$\begin{bmatrix} X(0,0)\,........\,X(0,N_2-1) \\ ...... \\ ..... \\ X(N_1-1,0)\,...\,X(N_1-1,N_2-1) \end{bmatrix}$$

3.1.1.2 Computer Program for Implementing 2-D DCT

A computer program in "C" for implementing a 2-D DCT for any size of input matrix is included in Appendix B. The algorithm is based on calculating a 1-D DCT using a Vector Radix-2 approach, and expanding it to a 2-D DCT using the row-column method. The program was used to transform a grey level image with a size of (256x256 pixels) by subdividing the image into small blocks. The size of block is controlled by the user. In order to avoid redundant computations, two look-up tables were generated before blocking the image, one for bit reversing and the other for cosine factors. Table 3.1 shows the time required to transform an image of 256x256 pixels into the transform domain using the Fast Cosine Transform and Direct Cosine Transform. The effect of block size on transforming time is also included. These results were obtained using an IBM compatible XT, with 33 MHZ microprocessor speed.

Block size	t(sec) using FCT	t(min) using DCT
8x8	8	7.2
16x16	21	28.5
32x32	33	114.5

Table 3.1: Time required to convert grey level image with size (256x256) pixels using FCT and DCT with different block sizes.

Table 3.1 shows that the time required to transform an image by blocking increases as the block size increases. Large size images (NxN) are usually divided into (NxN /MxM) blocks of size MxM. The computation time for block transform takes only $\log_2 M/\log_2 N$ of what is needed for the transform of the entire image at one time.

3.1.1.3 Image Compression with Block Cosine Truncation

If an NxN image is divided into (KxK) MxM submatrices with N=KM, the partitioned matrix can be denoted as f, where:

$$f=\begin{bmatrix} f(1,1) & f(1,2) & \ldots\ldots\ldots & f(1,M) \\ f(2,10 & f(2,2) & \ldots\ldots\ldots & f(2,M) \\ \ldots & \ldots & \ldots & \ldots \\ f(M,1) & f(M,2) & \ldots\ldots\ldots\ldots & f(M,M) \end{bmatrix} \ldots\ldots\ldots (3.13)$$

The DCT for each MxM is calculated and each pixel is mapped into DCT coefficients. A new matrix F of size MxM if obtained so that:

$$F(i,j) = DCT(f(i,j)) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (3.14)$$

Due to the energy packing property of DCT transform, the low frequency coefficients that are located at the upper left corner of F(i,j) have higher magnitudes than the coefficients elsewhere. Image compression is obtained by truncating the coefficients of low magnitude (high frequency components) and coding the low frequency components for each block, thus reducing the block bit rate. Image reconstruction is achieved by substituting zeros for the truncated coefficients to complete the block size. Next, the inverse DCT is computed for each block, and unblocking is performed to reconstruct the original image. The truncation of high frequency components results in degradation added mainly to the edges (borderlines between dark and bright regions) of the image.

3.1.2 Karhunen-Loeve Transform (KLT)

The KLT is the optimum transform in the energy-packing sense [8]; if only a limited number of transform coefficients are retained, the KLT will contain a larger fraction of the total energy compared to any other transform as shown in Figure (3.2). However, there are limitations to using the KLT in image compression, since heavy computations are needed to calculate the covariance function and the KLT is lacking in fast algorithms.

The compuation of KLT coefficients is done through the following steps[25]:

$$\sum_{m=1}^{N}\sum_{n=1}^{N} r(k,l;m,n)\ \phi(i,j;m,n) = \lambda_{i,j}\ \phi(i,j;k,l) \ldots\ldots (3.15)$$

where r(k,l;m,n) is the image covariance function:

$$r(k,l;m,n) = E\ [f(k,l)\ \ f^{*}(m,n)\ ] \ldots\ldots\ldots\ldots\ldots\ldots (3.16)$$

Thus the KLT transform is:

$$f_{i,j} = \sum_{k=1}^{N}\sum_{l=1}^{N} g_{k,l}\ \phi^{*}(i,j;k\ l) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (3.17)$$

$$g_{k,l} = \sum_{i=1}^{N}\sum_{j=1}^{N} f(k,l)\ \phi(i,j;k,l) \ldots\ldots\ldots (3.18)$$

and

$$E\,[g(k,l)\ \ g^{*}(m,n)] = 0$$
$$\text{unless } k=m\ ,\ l=n$$

3.2 Quantization

Using compression by quantization reduces the bit rate with a good approximation to the data dynamic range by reducing the number of representative levels. The key point to compression is referred to as quantization. Unfortunately, errors introduced into image data are mainly due to quantization.

During the last two decades, many quantization techniques have been developed: vector quantization [26], nonuniform quantization [27], DPCM [28,29], and block quantization [28] are some of the most popular. The subsequent discussion describes quantization, DPCM, and DCT transform coefficient quantization.

3.2.1 Vector Quantization

Vector quantization decomposes the image into a set of vectors. These vectors are selected in different ways, such as colour component of a pixel, the intensity values of spatially contiguous groups, or as transformed components of these groups. Next, a codebook of representative vectors is generated by training a large number of images using an iterative approach [30]. The codebook is usually stored on both the coder and decoder sides. Compression is achieved by replacing a vector from the image by the index of its closest match vector in the codebook. During the decoding process, the indices are used to retrieve the vectors from the codebook. These vectors are combined to reconstruct the compressed image. High compression ratios can be obtained using this method and the resulting images are indistinguishable from the original. The only drawback of this method is the high complexity of codebook generation. A bit rate of 0.5-1.44 bits/pixel can be obtained using vector quantization [3].

3.2.2 Delta Pulse Code Modulation (DPCM)

DPCM is the quantization technique that has to be applied directly to the image because it works very efficiently with highly correlated data. This method is based on the prediction of a pixel from one or more neighbouring pixels. The error image is then generated by subtracting the original pixel from the predicted one which results in a highly reduced dynamic range of error image. The pixel configuration of a third order predictor is shown in Figure (3.4):

$$\begin{bmatrix} X_2 & X_3 \cdots\cdots \\ X_1 & X_p \cdots\cdots \\ \cdot & \cdot\ \cdots\cdots \\ \cdot & \cdot\ \cdots\cdots \end{bmatrix}$$

Fig 3.4: Configuration of third order predictor.

The third order predictor can be expressed as:

$$X_p = a_1X_1 + a_2X_2 + a_3X_3 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (3.19)$$

Where: a_1, a_2, a_3 *are the weights*
X_1, X_2, X_3 *neighbouring pixels*
X_p *is the predicted pixel*

The higher order predictors give better results, but there is a small marginal gain beyond a third order predictor. After constructing the predicted pixels, the error image is generated as the difference between the original and the predicted pixels, using the following formula:

$$E = X_o - X_p \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (3.20)$$

where X_o is the original pixel

Finally the error image is quantized by a Lloyd-Max quantizer[31]. This type of quantizer has nonuniform decision regions. The decoding process will recover the image by accumulating the quantized error. A bit rate of 1-3 bits/pixel can be obtained using a non-adaptive DPCM [3].

3.2.3 DCT coefficients Quantization

The histogram of the DCT coefficients takes the shape of a Gaussian distribution [15] as shown in Figure (3.5):

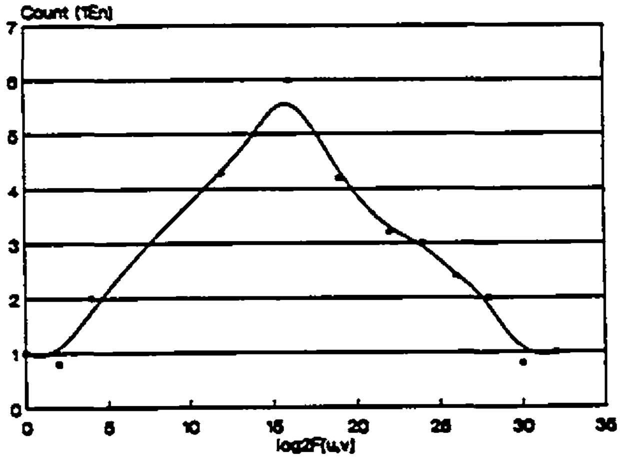


Figure (3.5): Histogram of DCT coefficients.

Thus, applying uniform quantization directly introduces a large amount of error into the compressed image. In order to reduce the errors, large amplitude coefficients should be preserved, since they contain most of the energy, while the small amplitude coefficients can be reduced to zero. The following discussion explains three approaches to coefficient quantization.

A- Histogram Equalization Approach

Histogram equalization [2] modifies the histogram from a Gaussian distribution to a more flattened distribution. The procedure for histogram modification includes calculating the cumulative distribution function (CDF) of the DCT coefficients as follows:

$$T(x_k) = \sum_{j=0}^{k} P_r(x_j) \qquad (3.21)$$

$$k=0,\ldots\ldots,L-1$$

where: P_r is the probability of occurrence of each coefficient

L is the total number of levels

and

$$P_r\,(x_k) = \frac{n_k}{n} \qquad (3.22)$$

n is the total number of the coefficients

Next, uniform quantization is applied to the data obtained from the previous transformation. This method was implemented in order to reduce the dynamic range of DCT coefficients obtained from the transforming blocks of (16x16 pixels) of the LENA monochrome image to 32 levels. The reconstructed image contained a large amount of errors and the compression ratio obtained after applying RLC and Huffman Coding was 2:1 or 4 bits/pixel.

B- Statistical Approach

The statistical approach quantizes the DC component of each block by using a first order DPCM. The AC coefficients that occupy the same location in each block are grouped in a set. Then, the standard deviation of each set is calculated using the following formula[40]:

$$\sigma=\sqrt{\frac{\sum_{i=1}^{n} y_i^2-\frac{\left(\sum_{i=1}^{n} y_i\right)^2}{n}}{n}} \quad \ldots\ldots\ldots\ldots\ldots\ldots (3.23)$$

Each AC coefficient in a set is divided by the standard deviation of that set. Finally, uniform quantization is applied to the new data. This procedure was implemented using the LENA monochrome image with a block size of (16x16 pixels and 8 bits/pixel). The dynamic range of the DCT coefficients was reduced to 32 levels and the reconstructed image contained noticeable degradations especially at the block edges (blocking effect). The compression ratio obtained was in the range of 2.6:1, or 3 bits/pixel.

In comparison with the histogram equalization method, the degradation added to the images were less and the compression ratio was higher.

C- JPEG scaling matrix

The JPEG method is based on dividing each block of DCT coefficients by a scaling matrix known as a "user's scaling matrix". The elements of this matrix represent the quantization step size. In order to preserve coefficients of large magnitudes, the step size is reduced for low frequency components, and increased for high frequency components. A complete description of the method and the results obtained using this method are presented in Chapter 4.

3.3 Image Compression with Block Truncation Coding

Using this technique [32], the image is divided into (MxM) blocks. Each block is coded separately into a two level signal. If the total number of pixels in a block is k=mxm, then the pixels' mean and variance are computed using the following equations:

$$\overline{X} = \frac{1}{k} \sum_{i=1}^{k} X_i \dots\dots\dots\dots\dots\dots\dots (3.24a)$$

$$\overline{X^2} = \frac{1}{k} \sum_{i=1}^{k} X_i^2 \dots\dots\dots\dots\dots\dots\dots (3.24b)$$

$$\sigma^2 = \overline{X^2} - (\overline{X})^2 \dots\dots\dots\dots\dots\dots\dots (3.24c)$$

Next, a threshold value X_{th}, and two output levels "a" and "b" are found for the one bit quantizer, such that:

$$if \;\; X_i \geq X_{th} \;\; output = b \dots\dots\dots (3.25a)$$

$$if \;\; X_i < X_{th} \;\; output = a \dots\dots\dots (3.25b)$$

If "q" is the number of pixels in a block that are greater than Xth, then "a" & "b" are calculated as follows:

$$X_{th} = \overline{X}$$

$$m\overline{X} = (m-q)\,a + q\,b \dots\dots\dots\dots\dots\dots (3.26)$$

$$m\overline{X^2} = (m-q)\,a^2 + q\,b^2 \dots\dots\dots\dots\dots (3.27)$$

solving for "a" and "b" yields:

$$a = \bar{X} - \sigma \sqrt{\frac{m-q}{q}} \ldots\ldots\ldots\ldots\ldots\ldots (3.28)$$

$$b = \bar{X} + \sigma \sqrt{\frac{m-q}{q}} \ldots\ldots\ldots\ldots\ldots\ldots (3.28)$$

The following example illustrates the algorithm for bit allocation in a few steps. Consider a block of an image expressed in the form:

$$X = \begin{bmatrix} 121 & 114 & 56 & 47 \\ 37 & 200 & 247 & 255 \\ 16 & 0 & 12 & 169 \\ 43 & 5 & 7 & 251 \end{bmatrix}$$

a) The mean and variance are computed first for the entire block:

$$\bar{X} = 98.75, \quad \sigma = 92.95, \quad q = 7$$

$$a = 16.7 - 17 \;, \; b = 204.2 - 204$$

b) A bit plan is constructed so that each pixel location is coded as "1" or "0" depending on whether or not the pixel is greater than the mean as shown:

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

c)Bit plan, mean, and variance are sent to the receiver.

d)The block is reconstructed by substituting "a" for "0" and "b" for "1".

For good image quality, the bit rate obtained using this method was 2 bits/pixel.

CHAPTER FOUR

JPEG STILL PICTURE COMPRESSION ALGORITHM

The Joint Photographic Experts Group (JPEG) is a joint ISO/CCITT technical committee which has been working for the past few years to establish the first international compression standard for monochrome and colour images [1]. The produced standard serves general purposes and can be utilized in a wide variety of image storage and communications applications. In order to meet the differing needs of many applications, the JPEG standard includes two basic compression methods. One is a DCT-based method, which is adopted for "lossy" compression and is adopted in this research. The other is a predictive method that is specified for "lossless" compression. The baseline algorithm which represents the "lossy" technique is shown in Figure (4.1).

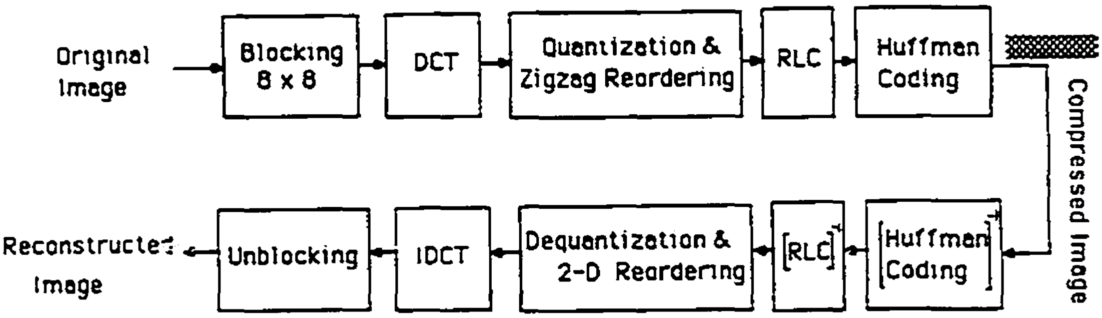


Figure 4.1: JPEG baseline lossy algorithm.

The upper and lower parts of the scheme represent the baseline coder and decoder respectively. The baseline algorithm operates in a sequential mode: the image is processed by taking small blocks from left to right and top to bottom in a single pass, by compressing one block at a time. The image is compressed at the end of the coder side.

The decoding process represents the inverse of coder operations: the last in will be first out. At the end of the decoding process, the reconstructed image will contain some degradations that may not be visually apparent. Indistinguishable images are obtained for a bit rate of 1 bit/pixel. Explanation of the JPEG baseline coder and decoder are presented in the following sections.

4.1 JPEG Baseline Coder

The JPEG baseline coder consists of lossy compression, represented by DCT and quantization, while lossless compression is represented by Run Length Coding and Huffman Coding. Prior to the mentioned steps, the image is subdivided into small blocks of 8x8 pixels. A complete discussion of each coder step is presented in the following sections.

4.1.1 Blocking

Blocking subdivides the image into 8x8 pixels. The advantages behind blocking the image can be summarized in three main points:

1. Buffering requirements are reduced and inexpensive implementation is provided.
2. Blocking provides pixels that are more highly correlated, which results in higher energy compactness in the DCT domain.
3. The speed of implementation is increased (Table 3.1).

Despite these advantages, smaller blocks provide higher degradations resulting from the quantization step. These degradations will appear as a "blocking effect". Figure (4.2) is an illustrative example for image blocking.

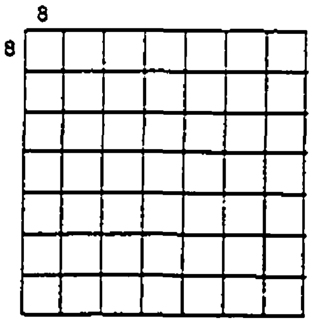


Figure 4.2: Image blocking.

4.1.2 Forward Discrete Cosine Transform (FDCT)

Prior to computing the FDCT, the block pixels are level shifted to a signed two's complement representation. The result will be better precision in computing the DCT coefficients. For 8-bits input precision, the level shift is achieved by subtracting 128 from each unsigned pixel. For 12-bit input precision, the level shift is achieved by subtracting 2048. The 2-D DCT is used to transform each block into the DCT domain as follows:

$$F(u,v)=\frac{1}{4}C(u)\,C(v)\sum_{x=0}^{7}\sum_{y=0}^{7}f(x,y)\,\cos\frac{\pi(2x+1)u}{16}\cos\frac{\pi(2y+1)v}{16} \quad (4.1)$$

$$\text{Where : } u,v=0,1,\ldots\ldots,7$$

$$C(u),C(v)=\frac{1}{\sqrt{2}} \quad \text{for } u,v=0$$

$$=1 \quad \text{otherwise}$$

The fast algorithm for implementing (4.1) was presented in section (3.1.1.1). Figure (4.3) shows a block of 8x8 pixels taken from a 256x256 pixel LENA image. This block has been level shifted by subtracting 128 from each pixel and applying equation (4.1) on block pixels. The DCT coefficient located on the upper left corner of the transformed block, represents the average brightness of the block and is known as "DC-coefficient", while the remaining sixty-three coefficients are referred to as "AC-coefficients".

$$\begin{bmatrix} 47 & 42 & 43 & 46 & 48 & 52 & 51 & 47 \\ 48 & 45 & 44 & 46 & 45 & 54 & 54 & 47 \\ 43 & 44 & 46 & 46 & 48 & 53 & 52 & 47 \\ 46 & 46 & 48 & 43 & 50 & 48 & 50 & 50 \\ 44 & 43 & 49 & 47 & 49 & 52 & 54 & 51 \\ 42 & 43 & 47 & 48 & 50 & 51 & 51 & 46 \\ 47 & 46 & 47 & 46 & 46 & 52 & 49 & 50 \\ 43 & 46 & 44 & 44 & 49 & 54 & 52 & 49 \end{bmatrix} \quad \begin{bmatrix} -81 & -86 & -85 & -82 & -80 & -76 & -77 & -81 \\ -80 & -83 & -84 & -82 & -83 & -74 & -74 & -81 \\ -85 & -84 & -82 & -82 & -80 & -75 & -76 & -81 \\ -82 & -82 & -80 & -85 & -78 & -80 & -78 & -78 \\ -84 & -85 & -79 & -81 & -79 & -76 & -74 & -77 \\ -86 & -85 & -81 & -80 & -78 & -77 & -77 & -82 \\ -81 & -82 & -81 & -82 & -82 & -76 & -79 & -78 \\ -85 & -82 & -84 & -84 & -79 & -74 & -76 & -79 \end{bmatrix}$$

(a) (b)

$$\begin{bmatrix} -642.75 & -26.27 & -2.45 & 11.62 & -9.54 & 4.18 & 3.31 & 1.42 \\ -1.47 & 0.582 & 1.32 & 3.011 & 0.672 & 3.34 & -0.4 & -0.18 \\ -2.14 & -0.06 & 1.715 & 4.84 & -0.095 & 0.571 & -0.20 & -4.18 \\ -0.17 & 0.215 & -2.194 & -0.678 & 2.297 & 2.836 & 0.671 & 0.243 \\ 0.707 & -1.563 & 1.25 & -0.37 & 1.5 & -2.113 & -0.247 & 2.796 \\ -2.32 & 3.407 & 1.154 & -1.418 & 2.536 & -2.559 & 1.153 & 1.081 \\ -3.32 & -2.183 & -5.453 & 0.556 & -0.23 & -0.517 & -1.46 & 0.776 \\ 1.92 & -1.936 & -0.179 & -0.71 & -0.459 & 1.36 & 1.619 & -1.34 \end{bmatrix}$$

(c)

Figure 4.3: a) Original block from LENA image. b) Block (a) level shifted by subtracting 128. c) The transformed block using 8x8 2-D DCT.

4.1.3 Quantization

The DCT coefficients are quantized to reduce their magnitude and increase the number of zero-valued coefficients. The quantization step includes dividing each block by what is called the "JPEG Scaling Matrix", where each element in the scaling matrix represents the quantization step size, the DCT coefficients of each block that occupy the same location are considered to have uniform distribution. The bit rate of an encoded image can be varied by scaling this matrix up and down. By scaling up the matrix, the quantization step size is increased, the bit rate is reduced, and more errors are introduced to DCT coefficients and vice versa. Figure (4.4) shows the " JPEG Scaling Matrix" that was found to work well on a large number of monochrome images and is also used to quantize the luminance signal of colour images.

$$Q(u,v) = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Figure 4.4: JPEG scaling matrix.

The matrix is generated "empirically through rigorous subjective testing which estimates the human psycho-visual thresholds for each DCT coefficient" [1]. A detailed description of experimental results on subjective testing of human visual systems is presented in [34].

During the generation process, CCIR 601 resolution testing images were used, and the elements of this matrix were derived based upon the principles of the sectioning method. This method first initializes the matrix elements to a certain value. One of the matrix elements is then varied, while the remaining sixty-three elements are kept constant until a "just-noticeable" change is detected in the recovered image. This process is carried out for each element in the matrix in order to complete one pass. Achieving optimum results requires more than one pass. The resulting quantized coefficients are given by:

$$F^{*}(u,v) = Nearest\ integer\ \left(\frac{F(u,v)}{Q(u,v)}\right)\ldots\ldots\ldots\ldots\ldots\ldots\ldots(4.2)$$

Quantizing the DCT coefficients shown in fig(4.3) results in the matrix shown in fig (4.5).

$$\begin{bmatrix} -80 & -4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 4.5: The DCT quantized coefficients.

Notice the large number of zero-valued coefficients generated from the quantization step. After quantizing the DCT coefficients, but before they are entropy coded, DC prediction is carried out. DC prediction means that the DC coefficient of the previous block is subtracted from the DC coefficient of the current block as follows:

$$DIFF = F^{*}(0) - PRED \qquad \dots\dots\dots\dots\dots\dots\dots (4.3)$$

where : $DIFF$ is the difference.

$F^{*}(0)$ is the current block DC coeff.

$PRED$ is the previous block DC coeff.

4.1.4 2-D to 1-D Zigzag Ordering

Prior to Entropy Coding, the quantized DCT coefficients are reordered into 1-D sequence according to a zigzag scan [35]. Figure (4.6) illustrates the route of zigzag scan.

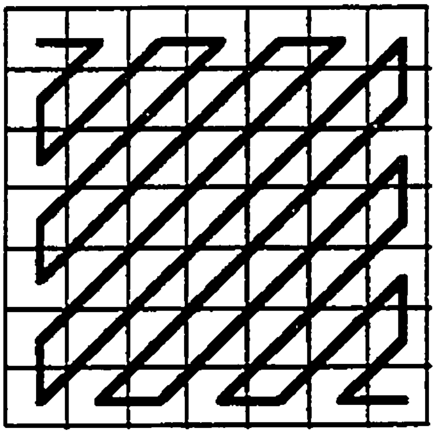


Figure 4.6: The route of zigzag scan.

With zigzag scanning, longer runs of zero-valued coefficients can be obtained because there is a high correlation between the DCT coefficients in the zigzag route. The long runs of zero-valued coefficients contribute to achieving higher compression ratios. The resulting array after applying the zigzag scan on the matrix in figure (4.5) is shown in figure (4.7).

4.1.5 Entropy Coding (RLC and Huffman Coding)

Entropy coding consists of coding the DC-coefficient separately from AC-coefficient. The following sections describe the respective coding methods used for DC and AC coefficients.

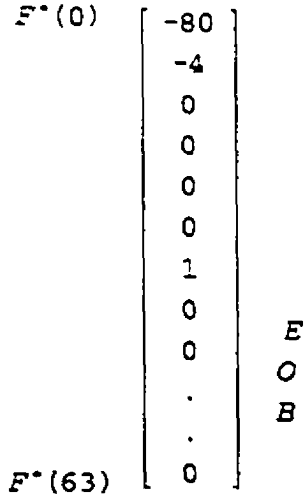


Figure 4.7: 1-D array resulting from zigzag reordering the matrix in figure (4.5).

4.1.5.1 DC Coefficients Coding

Due to the high correlation between the DC coefficients of the neighbouring blocks, the DC prediction produces differential DC coefficients which are small in magnitude. Table A1 (which is included in Appendix A) represents all the possible values of the differential DC coefficients indicated as categories in the first column. The ranges of each category are included in the second column while the third column consists of the Huffman code assigned to each category.

The coding process is implemented by three steps. Step 1: the category of the differential DC coefficient is found and the corresponding code is written to a buffer. Step 2: the sign bit is concatenated to the Huffman code(sign bit will be "1" if DIFF is -ve value and "0" if DIFF is +ve value). Step 3: the actual value is represented by number of bits determined by the category value. The three codes are concatenated to perform the final code which represents the differential DC coefficient DIFF. Figure (4.8) shows the structure of differential DC coefficient code:

category code	sign bit	actual value code

Figure(4.8): DC and AC coefficients code structure.

4.1.5.2 AC Coefficients Coding

Prior to assigning any code for AC coefficients, the number of zeros preceding the nonzero-valued AC coefficients is calculated as shown below:

```
F*(0)  [ -80 ]
       [  -4 ]
       [   0 ]
       [   0 ]     Zero Run Length      AC-VALUE
       [   0 ]
       [   0 ]           0                 -4
       [   1 ]           4                  1
       [   0 ]
       [   0 ]           0                  0       EOB
       [   . ]
       [   . ]
       [   . ]
F*(63) [   0 ]
```

The length of zero-valued coefficients and the category of the AC value following the zero-valued coefficients are used to retrieve the code corresponding to them from Huffman code tables. Table A2 (which is included in Appendix A) is one of the tables used by the JPEG algorithm to encode the AC coefficients. This table can be represented by a two dimensional array, the row index representing the number of zeros preceding the nonzero AC coefficient and the column index representing the category of the AC coefficient. The elements of this array are the Huffman codes. In order to assign a code to an AC coefficient, the number of zeros preceding the AC coefficient has to be known and the category of that AC coefficient should be calculated also. Once the code is retrieved, the coding process is similar to the DC coding process as shown in Figure (4.8).

Two special cases must be carefully considered in the AC coefficients coding process. First, the run length of zeros exceeding fifteen will be broken into a run length of fifteen zeros and category "0", and the remaining zeros will be coded normally. Secondly, zeros that represent the last coefficients in the block are coded as "0" run length and "0" category and are known as End Of Block (EOB).

The mentioned coding procedure for differential DC and AC coefficients was applied on the array shown in figure (4.7) and the resulting code is shown:

11110 1 0000101   100 1 001   111011 0 1   1010

The number of bits required to represent this block has been reduced from (64x8 bits/pixel= 512 bits/block) to 32 bits/block. The number of bits required to represent a block differs from one block to the other, depending on the statistical distribution of the pixels. This results in differing lengths of zero runs generated from the quantization step.

4.2 JPEG Baseline Decoder

The decoding process represents the inverse of coding operations in a "last-in-first-out" fashion: Huffman decoding is performed first, followed by Run Length decoding, zigzag reordering, dequantization, IDCT, and unblocking.

The compressed image is represented by a long string of "0"'s and "1"'s that represent the compressed blocks. The Huffman code tables are stored at the decoder side. These tables are needed in the process of reconstructing the original image. The steps necessary to decode the compressed image are described in the following sections.

4.2.1 Entropy Decoding (Huffman and RL Decoding)

The Huffman decoding process recovers the category, sign, and the actual value of the DC and AC coefficients on block-by-block basis from a string of "0"'s and "1"'s. The Huffman code tables are used to generate another set of tables known as "transition tables" [36]. These tables are used as a tool to break the long binary string into variable length codes representing the category, sign bit, the actual value of DCT quantized coefficients, and the End Of Block (EOB).

4.2.1.1 Transition Table

Transition tables are widely used in compilers to recognize a language in a program that takes an input as a string and breaks it into statements, either existing in that language as a "valid statement" or not existing as a "valid statement". For the latter case, error messages are received. The transition table is indexed by a state and vocabulary symbol. It also represents a software implementation of a state transition diagram which is usually used for sequential logic design. When implementing sequential logic design, a designer starts with a state transition diagram and derives the flip-flop logic outputs. A software designer can implement this transition diagram with a transition table. The advantage of using transition tables is in providing fast access to the transition of a given state on a given character. The only disadvantage of these tables is that they occupy a great deal of space when the input symbols are large. Figure (4.9) is an example of a simple transition table for the expression "--Not(Eol)*Eol" [36].

Transition table entries are either a state or an error flag. If we are in state S, reading character C, then T(S,C) will be the next state we visit, or it will be an error flag indicating that C cannot be part of the current expression. In figure (4.9), the error entries are shown as a blank entries.

State	Characters				
	-	Eol	a	b	....
1	2				
2	3				
3	3	4	3	3	3
4					

Figure 4.9: Example transition table for recognizing the expression "--Not(Eol)*Eol".

The concepts of transition tables used in compilers are adopted here to recognize a binary string. In this case, the transition table will consist of only two columns, one for "0", and the other for "1" as shown in figure (4.10):

State	Characters	
	"0"	"1"
0	1	6
1	2	3
2	-1	-1000
3	4	5
4	-2	-1000
5	-1000	-3
6	7	10
7	8	9
8	-4	-1000
9	-1000	-5
10	11	12
11	-6	-1000
12	13	14
13	-7	-1000
14	15	16
15	-8	-1000
16	17	18
17	-9	-1000
18	19	20
19	-10	-1000
20	21	22
21	-11	-1000
22	23	-1000

Figure (4.10) : A segment of transition table generated from Huffman codes table A1.

The algorithm to generate the transition tables in figure (4.10) is indicated below:

```
Initialize   new_state=1
for i=1 to (total number of codes) do
 {
    Initialize state=0
    for   bit_number=1    to    code length
     {
       head= LSB of current code
       if ( T(state,head) is not blank) then
         {
            T(state,head)=new_state
            state=new_state
            new_state=new_state+1
         }
       else    state=T(state,head)
       Shift current code right by one bit
     }
    T(state,head)= - category(or index)
 }
```

The transition table for DC coefficients was generated from the Huffman code Table A1, while the AC transition table was generated from Table A2. These two tables are used to obtain the category for DC values and an index for AC values. The AC index contains the AC category and zero run length through the transformation:

$$K = i*11 - j \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (4.4)$$

where: K is the index
i is the zero length
j is the category

In order to discriminate between the state value and the category, or the index values, a negative sign is appended to the category and the index values while blank locations are represented by -1000. The algorithm for decoding the DC-category or AC index is shown below:

```
Initialize state=0
while ( End Of File is not encountered ) do
        head=current bit from file
        new_state=T(state,head)
        state=new_state
        if ( T(state,head) < 0 and not equal to -1000)then

                category(or index)= absolute value(T(state,head))
                state=0

        else if ( T(state,head) is equal -1000)then
                lexical error
        end if
end loop
```

Once the DC_category is known, the actual DC value is extracted from the bits following the sign bit and determined by the value of category. For example, if the category value is equal to three, the three bits following the sign bit represent the actual DC value.

4.2.2 1-D to 2-D Zigzag Reordering

The data obtained from the previous step represent the quantized DCT coefficients ordered according to a zigzag scan. In order to arrange these coefficients in the normal 2-D form, the inverse of zigzag scanning is performed.

4.2.3 Dequantization

Dequantization is carried out by multiplying the quantized matrix by the JPEG scaling matrix shown in figure (4.4) as follows:

$$F(u,v) = F^{*}(u,v)\ Q(u,v) \ldots\ldots\ldots\ldots (4.5)$$

This step is considered to be irreversible because of the rounding error introduced in the quantization step. Figure (4.11) shows the block obtained after applying (4.5) on the quantized block shown in figure (4.5):

$$\begin{bmatrix}
-640 & -22 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

Figure (4.11): Block obtained from dequantization.

4.2.4 Inverse Discrete Cosine Transform(IDCT)

The inverse DCT is used to obtain the level shifted pixels using the following equation:

$$F(x,y) = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} c(u)\,c(v)\,F(u,v)\cos\frac{\pi(2x+1)u}{16}\cos\frac{\pi(2y+1)v}{16} \quad ..(4.6)$$

$$x,y=0,1,\ldots,7$$

The fast algorithm for implementing IDCT is based on vector Radix-2. and follows the same approach used for the forward fast DCT. The original unsigned block pixels are obtained by adding 128 to the signed pixels obtained from (4.6). Figure (4.12) shows the block obtained from applying (4.5) and (4.6) on the block in figure(4.11).

4.3 Computer Program for Implementing JPEG Baseline Compression Scheme

The JPEG baseline compression algorithm was implemented in "C" and is included in Appendix B. This program utilizes the Huffman Tables A1 and A2. and the scaling matrix shown in figure (4.4) for coding and decoding procedures. Table (4.1) shows the scaling factors. the compression ratio. and the bit rate obtained from compressing (256x256 pixels and 8 bits/pixel) LENA image. Figures 4.13 through 4.18 show the original and recovered LENA and GIRL images. For different scaling factors of the "JPEG quantization matrix". Tables 4.1 and 4.2 show the scaling factor. compression ratio. and the bit rate corresponding to these Figures. Scaling up the quantization matrix results in higher compression. but this results in inferior image quality as shown in Figures 4.13 through 4.18. The time required for the coding process was 49 seconds, while the decoding process required only 42 seconds. An IBM compatible PC with 33 MHZ microprocessor speed was used to obtain these results.

$$\begin{bmatrix} -82 & -82 & -81 & -80 & -79 & -78 & -77 & -77 \\ -82 & -82 & -81 & -80 & -79 & -78 & -77 & -77 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ & & & & & & & \\ -82 & -82 & -81 & -80 & -79 & -78 & -77 & -77 \end{bmatrix} \begin{bmatrix} 46 & 46 & 47 & 48 & 49 & 50 & 51 & 51 \\ 46 & 46 & 47 & 48 & 49 & 50 & 51 & 51 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ & & & & & & & \\ 46 & 46 & 47 & 48 & 49 & 50 & 51 & 51 \end{bmatrix}$$

(a) (b)

$$\begin{bmatrix} 47 & 42 & 43 & 46 & 48 & 52 & 51 & 47 \\ 48 & 45 & 44 & 46 & 45 & 54 & 54 & 47 \\ 43 & 44 & 46 & 46 & 48 & 53 & 52 & 47 \\ 46 & 46 & 48 & 43 & 50 & 48 & 50 & 50 \\ 44 & 43 & 49 & 47 & 49 & 52 & 54 & 51 \\ 42 & 43 & 47 & 48 & 50 & 51 & 51 & 46 \\ 47 & 46 & 47 & 46 & 46 & 52 & 49 & 50 \\ 43 & 46 & 44 & 44 & 49 & 54 & 52 & 49 \end{bmatrix}$$

(c)

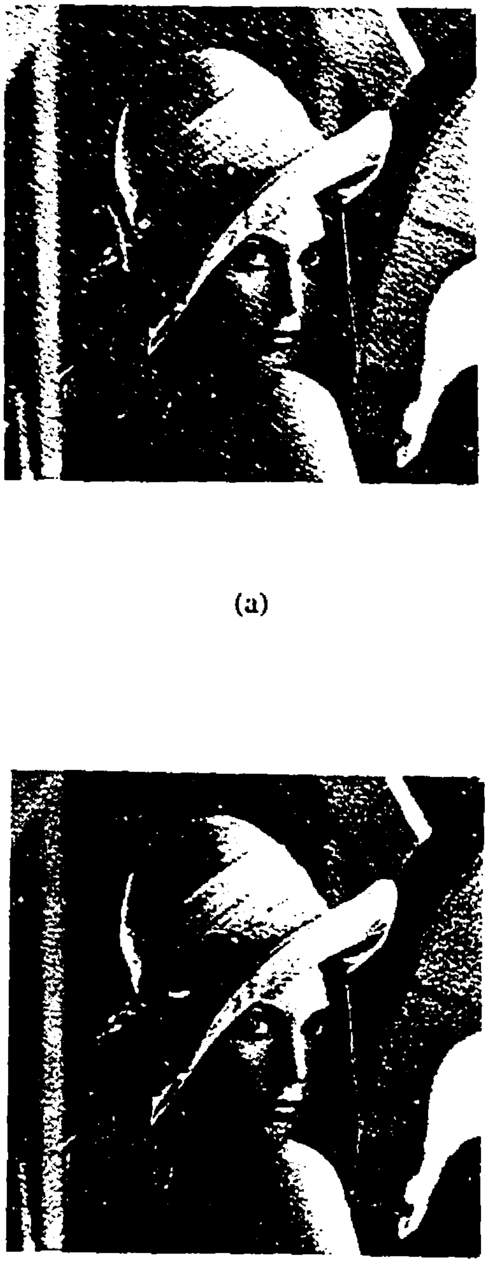
Figure 4.12 : (a) Block obtained from IDCT. (b) Original block after compression. (c) Original block before compression.

Scaling Factor	Compression Ratio	Bits/pixel
0.5	5.96	1.3
0.75	7.6	1.05
1	9.05	0.88
2	13.67	0.58
3	17.26	0.46
4	20.32	0.39

Table 4.1 : The effect of scaling the JPEG quantization matrix on the compression ratio and bit rate of LENA image (256x256x8 bits/pixel).

Scaling Factor	Compression Ratio	Bits/pixel
0.5	8	1
0.75	10	0.8
1	11.81	0.67
2	17.63	0.45
3	22	0.36
4	25.56	0.31

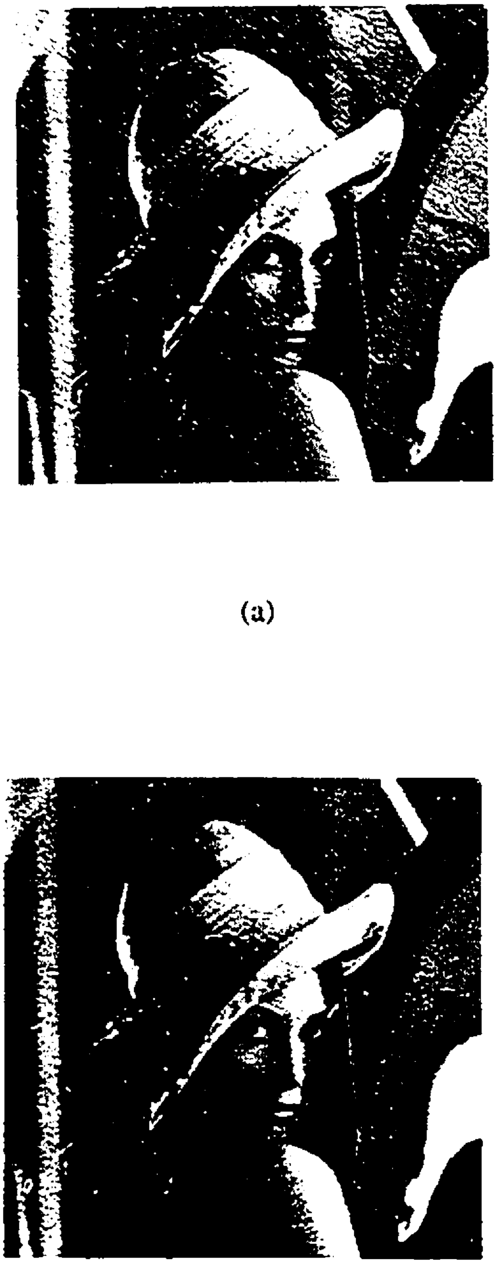
Table 4.2 : The effect of scaling the JPEG quantization matrix on the compression ratio and bit rate of GIRL image (256x256x8 bits/pixel).



(a)

(b)

Figure 4.13: (a) Original LENA image with 256x256 pixels and 8 bits/pixel.
(b) Compressed LENA image with 1.3 bits/pixel and quantization scaling factor = 0.5.



(a)

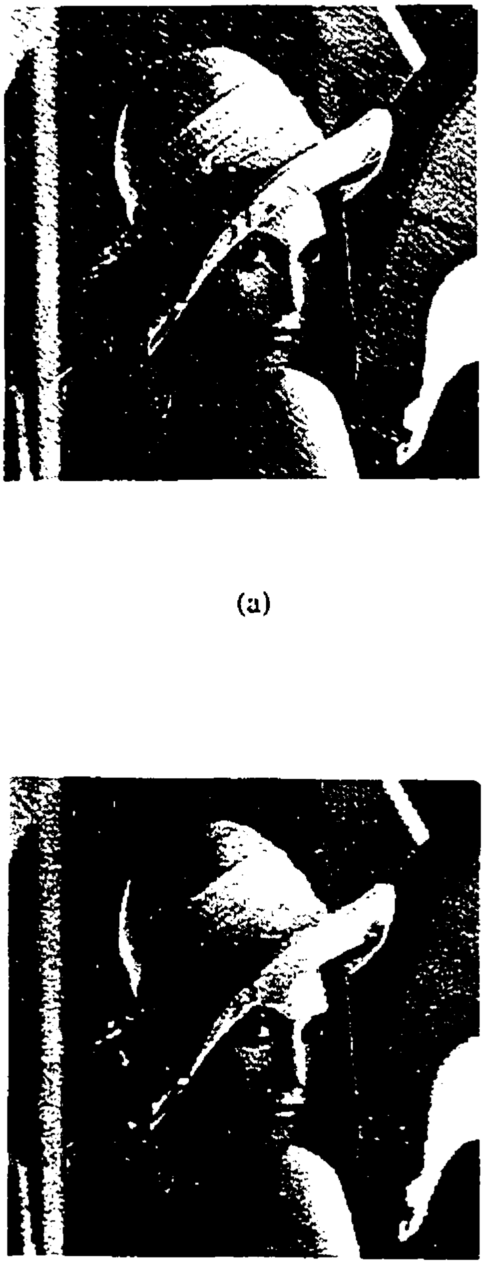
(b)

Figure 4.14: (a) Original LENA image with 256x256 pixels and 8 bits/pixel.
(b) Compressed LENA image with 1.05 bits/pixel and quantization scaling factor = 0.75.

(a)

(b)

Figure 4.15: (a) Original LENA image with 256x256 pixels and 8 bits/pixel.
(b) Compressed LENA image with 0.88 bits/pixel and quantization scaling factor = 1.0.



(a)

(b)

Figure 4.16: (a) Original LENA image with 256x256 pixels and 8 bits/pixel.
(b) Compressed LENA image with 0.58 bits/pixel and quantization scaling factor = 2.0.

(a)

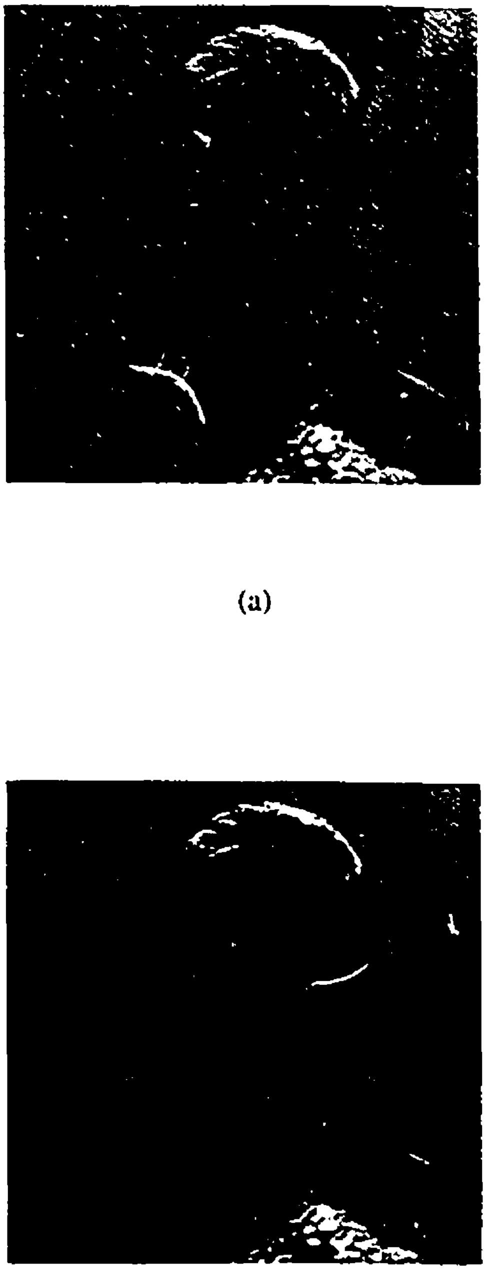
(b)

Figure 4.17: (a) Original LENA image with 256x256 pixels and 8 bits/pixel.
(b) Compressed LENA image with 0.46 bits/pixel and quantization scaling factor = 3.0.

(a)

(b)

Figure 4.18: (a) Original LENA image with 256x256 pixels and 8 bits/pixel.
(b) Compressed LENA image with 0.39 bits/pixel and quantization scaling factor = 4.0.



(a)

(b)

Figure 4.19: (a) Original GIRL image with 256x256 pixels and 8 bits/pixel.
(b) Compressed GIRL image with 1 bit/pixel and quantization scaling factor = 0.5.

(a)

(b)

Figure 4.20: (a) Original GIRL image with 256x256 pixels and 8 bits/pixel.
(b) Compressed GIRL image with 0.8 bits/pixel and quantization scaling factor = 0.75.

(a)

(b)

Figure 4.21: (a) Original GIRL image with 256x256 pixels and 8 bits/pixel.
(b) Compressed LENA image with 0.67 bits/pixel and quantization scaling factor = 1.0.

(a)

(b)

Figure 4.22: (a) Original GIRL image with 256x256 pixels and 8 bits/pixel.
(b) Compressed GIRL image with 0.45 bits/pixel and quantization scaling factor = 2.0.

(a)

(b)

Figure 4.23: (a) Original GIRL image with 256x256 pixels and 8 bits/pixel.
(b) Compressed GIRL image with 0.36 bits/pixel and quantization scaling factor = 3.0.

(a)

(b)

Figure 4.24: (a) Original GIRL image with 256x256 pixels and 8 bits/pixel.
(b) Compressed GIRL image with 0.31 bits/pixel and quantization scaling factor = 4.0.

4.4 Colour Images

Colour images are represented by three bands at each pixel location (i,j) in the two dimensional matrix that represents the colour image. Red corresponds to R(i,j), Green to G(i,j), and Blue to B(i,j). Experimental results indicate that the human visual system (HVS) is more sensitive to certain wavelengths compared to others. For example, the HVS is more sensitive to green colour compared to blue colour. As a result, the blue colour can be compressed more than green [47]. In order to compress colour images, a linear transformation is needed. This reduces the correlation between the R,G, and B bands and produces one achromatic channel (such as luminance), and two chromatic channels (such as chrominance). Since HVS is more sensitive to variations in the luminance channel, high fidelity encoding with large errors is allowed in the chrominance channels. The transformation from the R,G, and B bands into luminance and chrominance channels is achieved through the following set of equations:

$$\begin{pmatrix} Y(i,j) = 0.2990\ R(i,j) + 0.5870\ G(i,j) + 0.1140\ B(i,j) \\ Cb(i,j) = -0.1687\ R(i,j) - 0.3312\ G(i,j) + 0.500\ B(i,j) \\ Cr(i,j) = 0.5000\ R(i,j) - 0.4186\ G(i,j) - 0.0813\ B(i,j) \end{pmatrix} . \quad (4.8)$$

where Y denotes the luminance channels, and Cb and Cr are the chrominance channels. The R,G, and B bands are recovered from the luminance and chrominance channels through the following set of equations:

$$\left\{\begin{array}{lll} R(i,j) = Y(i,j) & & -1.40200\ Cr(i,j) \\ G(i,j) = Y(i,j) & -0.34414\ Cb(i,j) & -0.71414\ Cr(i,j) \\ B(i,j) = Y(i,j) & -1.77200\ Cb(i,j) & \end{array}\right) \ldots\ldots(4.9)$$

4.4.1 JPEG Compression Algorithm for Colour Images

The JPEG compression algorithm for colour images is expansion of the algorithm used for monochrome images. Since monochrome images are represented by a single band, while colour images are represented by three bands, the algorithm for monochrome images is used to compress the three channels (one luminance and two chrominance) separately. The JPEG baseline algorithm for colour image compression was implemented to compress a colour image of 256x256 pixels and 24 bits/pixel and the source code in "C" is included in Appendix C. The scaling matrix used to quantize the luminance channel is the same matrix used to quantize monochrome images and is shown in Figure (4.4), while the chrominance scaling matrix is shown in Figure (4.25). The Huffman code Tables A1 through A4 were used for coding the three channels separately.

$$\begin{bmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix}$$

Figure 4.25: Scaling matrix for chrominance channels.

The obtained scaling factors along with the compression ratios and bit rates are shown in Table (4.3). The time required for the coding process was 147 seconds and the decoding process required only 126 seconds. Figures 4.26 through 4.30 show the original and compressed images corresponding to the scaling factors and bit rates of Table (4.3).

Scaling Factor	Compression Ratio	Bits/pixel
0.5	12	2
1	18.5	1.3
2	26.4	0.9
4	34.75	0.7
6	39.5	0.6

Table 4.3 : The effect of scaling the JPEG quantization matrices on the compression ratio and bit rate of a colour image (256x256x24 bits/pixel).

luminance colour

(a)

luminance colour

(b)

Figure 4.26: (a) Original image with 256x256 pixels and 24 bits/pixel.
(b) Compressed image with 2 bits/pixel and scaling factor = 0.5 .

luminance colour

(a)

luminance colour

(b)

Figure 4.27: (a) Original image with 256x256 pixels and 24 bits/pixel.
(b) Compressed image with 1.3 bits/pixel and scaling factor = 1 .

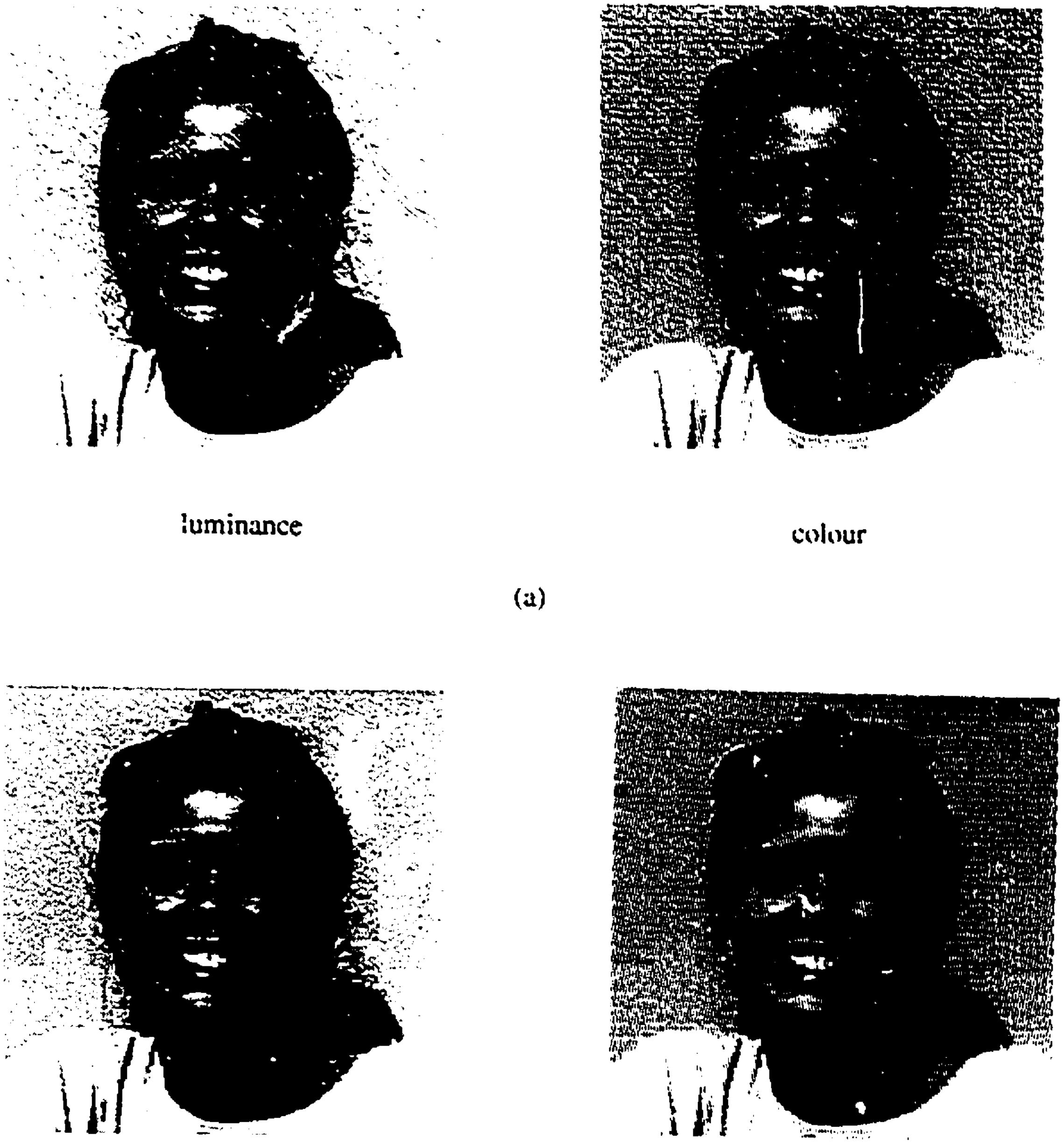
luminance colour

(a)

luminance colour

(b)

Figure 4.28: (a) Original image with 256x256 pixels and 24 bits/pixel.
(b) Compressed image with 0.9 bits/pixel and scaling factor = 2 .



luminance

colour

(a)

luminance

colour

(b)

Figure 4.29: (a) Original image with 256x256 pixels and 24 bits/pixel.
(b) Compressed image with 0.7 bits/pixel and scaling factor = 4 .

luminance colour

(a)

luminance colour

(b)

Figure 4.30: (a) Original image with 256x256 pixels
and 24 bits/pixel.
(b) Compressed image with 0.6 bits/pixel
and scaling factor = 6 .

CHAPTER FIVE

CONCLUSIONS AND FUTURE WORK

5.1 Summary and Conclusions

This work consisted of two main parts. The first part was a survey of the best algorithms that can be used to implement a general purpose lossy compression scheme with emphasis on the quantization techniques. The second part was the implementation of a proposed lossy compression scheme for standardization by the JPEG group (this scheme has been accepted as a standard by the CCITT in November of 1992).

Two main quantization methods were adopted. The first one was based on histogram modification of the DCT coefficients to obtain a more flattened distribution, and then applying uniform quantization to the modified data. This technique obtained a bit rate of 4 bits/pixel. In the second method, the dynamic range of the DCT coefficients was reduced by dividing each DCT coefficient by the standard deviation and applying uniform quantization to the modified data. The obtained bit rate was 3 bits/pixel and the image quality was better compared to the histogram equalization method.

The JPEG standard algorithm for compressing monochrome and colour images was discussed and a software implementation was developed. In this implementation, a new and fast decoding technique was developed. This method was based on modifying the concepts of the transition table, used widely in compiler design, to decode a binary string into variable length codes.

The JPEG compression scheme yields very high compression ratios. A bit rate of approximately 1 bit/pixel was obtained with monochrome and colour images, while still maintaining a high level of image quality. A key advantage in using the JPEG scheme is that a variable compression ratio can be obtained by simply changing the quantization matrix scaling factor. Two sets of images (256x256 pixels and 8 bits/pixel) were used as example images to illustrate the effect of varying this scaling factor. Good image quality was obtained with a bit rate of 1 bit/pixel. The time required to compress the monochrome image was approximately 42 seconds, while the decompressing process required around 44 seconds.

The JPEG colour image compression scheme is an expansion of the baseline algorithm used for monochrome images. It is accomplished by generating the luminance (one channel) and chrominance (two channels) and compressing the three channels separately. Due to the low sensitivity of the HVS to variations in chrominance, higher compression in these channels is possible. The colour image used was 256x256 pixels with 24 bits/pixel and a bit rate of 1.3 bit/pixel was obtained with good image quality. The time needed for the compression process was approximately 147 seconds, while decompression required about 126 seconds. Human observation was the only testing method used to determine the quality of the images recovered after compression.

5.2 Suggestions for Future Work

In order to complement the JPEG compression scheme, a method for dynamic quantization matrix generation could be developed. If static quantization matrices are used, compression ratios will vary for different images. Thus, a method which adaptively adjusts the quantization matrices for specific images will result in more consistent compression ratios. This approach could use statistical methods to extract the quantization matrix from image features. Coding time will increase with this method; however, this will be an acceptable trade-off for most applications.

An alternate route that could be taken to improve the performance of the JPEG quantization method would be to generate a codebook that contains different quantization matrices and are optimized for different image sets. These matrices could be generated empirically and the codebook should be saved at the coder and decoder side. During the compression process the indices of the used matrices will be sent in the header of the file. These indices could then be used during the decompressing process to retrieve the quantization matrices.

REFERENCES

[1] E. Hamilton, ***JPEG Still Picture Compression Algorithm***, A Technical Report (JPEG DIS), Oct. 15/1991.

[2] R.C. Gonzalez and P. Wintz, ***Digital Image Processing***, Addison-Wesly Publishing Company Inc. (1977).

[3] M. Rabbani and P. W. Jones, ***Digital Image Compression Techniques***, Eastman Kodak (1991).

[4] N. Abramson, ***Information Theory and Coding***, McGraw-Hill, NY, 1963.

[5] D. A. Huffman, "A Method for the Construction of Minimum Redundancy," *Proceeding of the IRE*, Vol. 40, pp. 1098-1101, Sep. 1952.

[6] Glen G. Langdon, Jr., " An Introduction to Arithmetic Coding," *IBM Jou. Res. Develop.*, Vol. 28, No. 2, Mar. 1982.

[7] Jack Capon, " A Probabilistic Model for Run-Length Coding of Pictures," *IRE Trans. Infor. Theory*, pp. 157-163, Dec. 1959.

[8] W. K. Pratt, " Karhunen-Loeve Transform Coding of Images," *Proc. IEEE Int. Symp. Inf. Theory*, 1970.

[9] N. Ahmed, T. Natarajan, and K. R. Rao, " Discrete Cosine Transform," *IEEE Trans. on Computers*, pp. 90-93, Jan. 1974.

[10] A. N. Netravali and J. O. Limb, " Picture Coding," *Proc. IEEE*, Vol. 68, No. 3, pp. 366-406, Mar. 1980.

[11] W. W. Peterson, T. G. Birdsall, and W. C. Fix, " The Theory of Signal Detectability," *IRE Trans. Info. Theory*, Vol. IT-4, pp. 171-212, 1954.

[12] J. L. Fleiss, ***Statistical Methods for Rates and Proportions***, John Wiley & Sons, NY, 1981.

[13] M. F. Quinn, " Relation of Observer Agreement to Accuracy According to a Two-Receiver Signal Detection Model of Diagnosis," *Medical Decision Making*, Vol. 9, No. 3, pp.196-206, Jul-Sep 1989.

[14] G. Wallace, R. Vision, and H. Poulsen, " Subjective Testing Results for Still Picture Compression Algorithm for International Standardization," *Globcom 88*, Vol. CH2535-3/88, pp. 1022-1027, 1988.

[15] Chen Ji, " The Effect of Data Compression on Image Quality in High Resolution Digital Chest Radiography," *Ph.D Dissertation*, University of Michigan, 1991.

[16] M. Lu and C. Chen, " A Huffman-Type Code Generator with Order-N Complexity," *IEEE Trans. ASSP*, Vol. 38, No. 9, pp. 1619-1626, Sep. 1990.

[17] Stephen J.P. Todd, Glen G. Langdon, Jr., and G. Nigel N. Martin, " A General Fixed Rate Arithmetic Coding for Constrained Channels," *IBM Jou. Res. Develop.*, Vol. 27, pp. 107-115, 1983.

[18] H. Meyr, H.G. Robdolsky, and T.S. Huang, " Optimum Run Length Codes," *IEEE Trans. on Communications*, Vol. COM-22, No. 6, pp. 826-835, June 1974.

[19] J. Ziv and A. Lempel, " A Universal Algorithm for Sequential Data Compression ," *IEEE Trans. on Info. Theory*, Vol. IT-23, No. 3, pp. 337-343, May 1977.

[20] R.N. Warburton, P.D. Fisher, J. Nosil, G.W. Brauer, W.J. Lawrence, and G.W. Ritchie, " Digital Diagnostic Imaging with a Comprehensive PACS: Hypothetical Economic Evaluation at a Large Community Hospital," *Journal of Digital Imaging*, Vol. 3, No. 2, pp. 101- 107, May 1990.

[21] H.B. Kekre and J.K. Solank, " Comparative Performance of Various Trigonometric Unitary Transforms for Transform Image Coding," *Int. J. Electronics*, Vol. 44, No. 3, pp. 305-315, 1978.

[22] A. Habibi an P. A. Wintz, "Image Coding by Linear Transformation and Block Quantization," *IEEE Trans. on Communication Tech.*, Vol. COM-19, pp. 50-62, Feb. 1971.

[23] K. R. Rao and P. Yip, ***Discrete Cosine Transform, Algorithms, Advantages, Applications***, Academic Press Inc., CA 1990.

[24] H.S. Hou, " A Fast Recursive Algorithm for Computing the Discrete Cosine Transform," *IEEE Trans. on ASSP*, pp. 988-991, 1989.

[25] A. K. Jain, " A Fast Karhunen-Loeve Transform for a Class of Random Processes," *IEEE Trans. on Communications*, Vol. COM-24, pp. 1023-1029, Sep. 1976.

[26] M. Goldberg, P. R. Boucher, and S. Shlien, " Image Compression Using Adaptive Vector Quantization," *IEEE Trans. on Comm.*, Vol. COM-34, No. 2, pp. 180-187, Feb. 1986.

[27] J. W. Mark and T. D. Todd, "A Nonuniform Sampling Approach to Data Compression," *IEEE Trans. on Communications*, Vol. COM-29, No. 1, pp. 24-32, Jan. 1981.

[28] A. Habibi, " Comparison of nth-Order DPCM Encoder with Linear Transformations and Block Quantization Techniques," *IEEE Trans. on Communications Tech.*, Vol. COM-19(6), pp. 948-956, 1971.

[29] A. N. Netravali, " On Quantizers for DPCM Coding of Picture Signals," *IEEE on Inf. Theory*, Vol. IT-23(3), pp. 360-370, 1977.

[30] Y. Linde, A. Buzo, and R. M. Gray, " An Algorithm for Vector Quantizer Design," *IEEE Trans. on Communications*, Vol. COM-28, No.1, pp. 84-95, Jan. 1980.

[31] J. Max, " Quantization for Minimum Distortion," *IRE Trans. on Information Theory*, Vol. IT-6, No. 1, pp. 7-12, 1960.

[32] E. J. Delp and O. R. Mitchell, " Image Compression Using Block Truncation Method," *IEEE Trans. on Communications*, Vol COM-27, No. 9, pp. 1335-1341, Sep. 1979.

[33] W. Mendenhall, ***Introduction to Probability and Statistics***, 5th Edition Duxburg Press, Massachusettes, 1979.

[34] H. Lohscheller, " A Subjectively Adapted Image Communication System," *IEEE Trans. on Comm.*, Vol. COM-32, pp. 1316-1322, Dec. 1984.

[35] A. Tscher and R. Cox, " An Adaptive Transform Coding Algorithm," *IEEE Trans. on Communications*, pp. 47.20-47.25., 1976.

[36] C. N. Fischer and R. J. Leblance, Jr., ***CRAFTING A COMPILER***, Benjamin/Cummings Publ. Co., 1988.

Appendix A

Huffman Code Tables

Category	category range	Huffman codes
0	0	0 0
1	-1,1	0 1 0
2	-3,-2,2,3	0 1 1
3	-7,...,-4,4,...,7	1 0 0
4	-15,...,-8,8,...,15	1 0 1
5	-31,...,-16,16,...,31	1 1 0
6	-63,...,-32,32,...,63	1 1 1 0
7	-127,...,-64,64,...,127	1 1 1 1 0
8	-255,...,-128,128,...,255	1 1 1 1 1 0
9	-511,...,-256,256,...,511	1 1 1 1 1 1 0
10	-1023,...,-512,512,...,1023	1 1 1 1 1 1 1 0
11	-2047,...,-1024,1024,...,2047	1 1 1 1 1 1 1 1 0

Table A1:Huffman codes table for luminance and monochrome DC difference.

Category	category range	Huffman codes
0	0	0 0
1	-1,1	0 1
2	-3,-2,2,3	1 0
3	-7,...,-4,4,...,7	1 1 0
4	-15,...,-8,8,...,15	1 1 1 0
5	-31,...,-16,16,...,31	1 1 1 1 0
6	-63,...,-32,32,...,63	1 1 1 1 1 0
7	-127,...,-64,64,...,127	1 1 1 1 1 1 0
8	-255,...,-128,128,...,255	1 1 1 1 1 1 1 0
9	-511,...,-256,256,...,511	1 1 1 1 1 1 1 1 0
10	-1023,...,-512,512,...,1023	1 1 1 1 1 1 1 1 1 0
11	-2047,...,-1024,1024,...,2047	1 1 1 1 1 1 1 1 1 1 0

Table A2:Huffman codes table for chrominance DC difference.

zero run	category	Huffman codes
0	0	1 0 1 0 (END OF BLOCK EOB)
0	1	0 0
0	2	0 1
.	.	.
.	.	.
0	10	1 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1
1	1	1 1 0 0
1	2	1 1 0 1 1
.	.	.
.	.	.
1	10	1 1 1 1 1 1 1 1 1 0 0 0 1 0 0 0
2	1	1 1 1 0 0
2	2	1 1 1 1 1 0 0 1
.	.	.
.	.	.
2	10	1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0
3	1	1 1 1 0 1 0
3	2	1 1 1 1 1 0 1 1 1
.	.	.
.	.	.
3	10	1 1 1 1 1 1 1 1 1 0 0 1 0 1 0 1
4	1	1 1 1 0 1 1
4	2	1 1 1 1 1 1 1 0 0 0
4	3	1 1 1 1 1 1 1 1 1 0 0 1 0 1 1 0
.	.	.
.	.	.
4	10	1 1 1 1 1 1 1 1 1 0 0 1 1 1 0 1

Table A3 :Huffman codes table for luminance and monochrome AC coefficients (first segment).

zero run	category	Huffman codes
5	1	1111010
5	2	11111110111
5	3	1111111110011110
.	.	.
.	.	.
5	10	1111111110100101
6	1	1111011
6	2	111111110110
.	.	.
.	.	.
6	10	1111111110101101
7	1	11111010
7	2	111111110111
.	.	.
.	.	.
7	10	1111111110110101
8	1	111111000
8	2	111111111000000
.	.	.
.	.	.
8	10	1111111110111101
9	1	111111001
9	2	1111111110111110
9	3	1111111110111111
.	.	.
.	.	.
9	10	1111111111000110

Table A3 : Huffman codes table for luminance and monochrome AC coefficients(second segment).

zero run	category	Huffman codes
10	1	111111010
10	2	1111111111000111
10	3	1111111111001000
.	.	.
.	.	.
10	10	1111111111001111
11	1	1111111001
11	2	1111111111010000
.	.	.
.	.	.
11	10	1111111111011000
12	1	1111111010
12	2	1111111111011001
.	.	.
.	.	.
12	10	1111111111100001
13	1	11111111000
13	2	1111111111100010
.	.	.
.	.	.
13	10	1111111111101010
14	1	1111111111101011
14	2	1111111111101100
14	3	1111111111101101
.	.	.
.	.	.
14	10	1111111111110100

Table A3 : Huffman codes table for luminance and monochrome AC coefficients(third segment).

zero run	category	Huffman codes
15	0	1 1 1 1 1 1 1 1 0 0 1 (ZRL)
15	1	1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1
15	2	1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0
15	3	1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1
15	4	1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
15	5	1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1
15	6	1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0
15	7	1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1
15	8	1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0
15	9	1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
15	10	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
*******	*******	********************************
******	*******	*******************************
*****	*******	******************************
****	*******	*****************************
***	*******	***************************
**	*******	*************************
*	*******	************************
	*******	***********************
	******	*********************
	*****	********************
	****	*******************
	***	******************
	**	*****************
	*	****************
		***************
		*************

Table A3 :Huffman codes table for luminance and monochrome AC coefficients(fourth segment).

zero run	category	Huffman codes
0	0	0 0 (END OF BLOCK EOB)
0	1	0 1
0	2	1 0 0
.	.	.
.	.	.
0	10	1 1 1 1 1 1 1 1 0 1 0 0
1	1	1 0 1 1
1	2	1 1 1 0 0 1
.	.	.
.	.	.
1	10	1 1 1 1 1 1 1 1 1 0 0 0 1 0 1 1
2	1	1 1 0 1 0
2	2	1 1 1 1 0 1 1 1
.	.	.
.	.	.
2	10	1 1 1 1 1 1 1 1 1 0 0 1 0 0 0 0
3	1	1 1 0 1 1
3	2	1 1 1 1 1 0 1 1 1
.	.	.
.	.	.
3	10	1 1 1 1 1 1 1 1 1 0 0 1 0 1 1 0
4	1	1 1 1 0 1 0
4	2	1 1 1 1 1 0 1 1 0
4	3	1 1 1 1 1 1 1 1 1 0 0 1 0 1 1 1
.	.	.
.	.	.
4	10	1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 0

Table A4 : Huffman codes table for chrominance monochrome AC coefficients(first segment).

zero run	category	Huffman codes
5	1	111011
5	2	1111111001
5	3	1111111110011111
.	.	.
.	.	.
5	10	1111111110100110
6	1	1111001
6	2	111111110111
.	.	.
.	.	.
6	10	1111111110101110
7	1	1111010
7	2	11111111000
.	.	.
.	.	.
7	10	1111111110110110
8	1	11111001
8	2	1111111110110111
.	.	.
.	.	.
8	10	1111111110111111
9	1	111110111
9	2	1111111111000000
9	3	1111111111000001
.	.	.
.	.	.
9	10	1111111111001000

Table A4 : Huffman codes table for chrominance monochrome AC coefficients(second segment).

zero run	category	Huffman codes
10	1	1 1 1 1 1 1 0 0 0
10	2	1 1 1 1 1 1 1 1 1 1 0 0 1 0 0 1
10	3	1 1 1 1 1 1 1 1 1 1 0 0 1 0 1 0
.	.	.
.	.	.
10	10	1 1 1 1 1 1 1 1 1 1 0 1 0 0 0 1
11	1	1 1 1 1 1 1 0 0 1
11	2	1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 0
.	.	.
.	.	.
11	10	1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0
12	1	1 1 1 1 1 1 0 1 0
12	2	1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1
.	.	.
.	.	.
12	10	1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1
13	1	1 1 1 1 1 1 1 1 0 0 1
13	2	1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0
.	.	.
.	.	.
13	10	1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 0
14	1	1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
14	2	1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1
14	3	1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0
.	.	.
.	.	.
14	10	1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1

Table A4 : Huffman codes table for chrominance monochrome AC coefficients(third segment).

zero run	category	Huffman codes
15	0	1111111010 (ZRL)
15	1	111111111000011
15	2	1111111111110110
15	3	1111111111110111
15	4	1111111111111000
15	5	1111111111111001
15	6	1111111111111010
15	7	1111111111111011
15	8	1111111111111100
15	9	1111111111111101
15	10	1111111111111110
*******	*******	********************************
******	*******	*******************************
*****	*******	******************************
****	*******	*****************************
***	*******	***************************
**	*******	*************************
*	*******	************************
	*******	***********************
	******	*********************
	*****	********************
	****	*******************
	***	******************
	**	*****************
	*	****************
		***************
		*************

Table A4 : Huffman codes table for chrominance monochrome AC coefficients(fourth segment).

Appendix B

JPEG Compression Algorithm For Monochrome Images

Programs Listings

```
/*************************************************************************
**************************************************************************
** The following program is implementation of JPEG baseline coder      **
** for monochrome images explained in section 4.1, the coder consist of: **
**              1. blocking                                            **
**              2. 8x8 2-D DCT                                         **
**              3. Dividing by JPEG scaling matrix                     **
**              4. Run Length coding                                   **
**              5. Huffman coding                                      **
**************************************************************************
*************************************************************************/
#include<stdio.h>
#include<math.h>
#include<float.h>
#include<stdlib.h>
#include<io.h>
#include<malloc.h>
#define ROWS  8
short image_width, scalor, n=8, blocks;

double AB[ROWS][ROWS];
double AS[ROWS][ROWS];
double XC[ROWS][ROWS];

void CODE_LIST(); /* Subprogram that contains the Huffman codes */

void HUFF();      /* Subprogram for coding each block into variable
                     length codes */
void dct1 (short n, short m, float *xr, float *xrs, float *CX, unsigned int *L);
void butterfly (short n, short m, float *xrs, float *CX);
void bit_reverse (unsigned int *L, short n, short m, float *xrs);
void subtract (short n, short m, float *xrs);

main()
{
  float *xr,*xrs,*temp,*CX,*yro;
  unsigned char *yr,kk;
  char *yr_sign;
  short i,j,k,m,index,l,le,le1,d1,d2,c,z,mark;
  long loc,loco;
  double nsq, pi=3.14159;
  unsigned int MASK,C,A,*L;
  char file_name[15];
  FILE *in_file,*fp;
```

```
clrscr();
printf(" Enter the name of the input file ->");
scanf("%s",file_name);
/*************************************
   Open original image file
**************************************/
   if( (in_file = fopen(file_name,"rb")) == NULL)
        {
                 printf(" Can't open input file \n");
                 exit(1);
        }

   printf(" enter the scalor factor for JPEG matrix\n");
   scanf("%d",&scalor);

   nsq=(double)filelength(fileno(in_file));
   image_width=sqrt(nsq);
   blocks=(short)image_width/n;
   m=(int)(log10((double)n)/log10((double)2.0)+0.5);

     kk=1;
          for(i=0;i<m;i++)
             kk<<=1;
   if(kk != n)
       printf(" Length of input file has to be multiples of 2\n");
/****************************************************
  Open output file to write the quantized DCT to it
*****************************************************/
  if (( fp=fopen("out3.img","w")) == NULL)
        {
                 printf(" Can't open output file \n");
                 exit(1);
        }
/****************************************
 allocating memory for xr ,xrs ,yr,CX
 ****************************************/
 xr=(float *)malloc((n+1)*sizeof(float));
 xrs=(float *)malloc((n+1)*sizeof(float));
 temp=(float *)malloc((n+1)*(n+1)*sizeof(float));
 yr=(unsigned char *)malloc((n+1)*(n+1)*sizeof(unsigned char));
 yr_sign=(char *)malloc((n+1)*(n+1)*sizeof(char));
 yro=(float *)malloc((n+1)*(n+1)*sizeof(float));
 CX=(float *)malloc((n+1)*sizeof(float));
 L=(unsigned int *)malloc((n+1)*sizeof(unsigned int));
```

```
/*********************************************
 generating Look Up Table for cosine factors
 *********************************************/
 k=0;
 for(l = 1;l <= m;l++)
   {
     le=pow(2,m+1-l);
     le1=le/2;
     for(j = 0;j < le1; j++)
       {
        CX[j+k]=cos(pi*(4.0*j+1)/(2.0*le));
       }
       k=k+le1;
  }
/*********************************************
  generating Look Up Table for bit reversing
 **********************************************/
  for(k=0;k < n;k++)
    {
     MASK=1;
     C=0;
     for(i=0,j=m-1;i < m;i++,j--)
        {
         A=(k&MASK) >>i;
         A<<=j;
         C |=A;
         MASK=MASK << 1;
        }
      L[k]=C;
     }
/*************************************
 JPEG quantization matrix
 *************************************/
AS[0][0]=16.0;AS[0][1]=11.0;AS[0][2]=10.0;AS[0][3]=17.0;AS[0][4]=24.0;
AS[0][5]=40.0;AS[0][6]=51.0;AS[0][7]=61.0;
AS[1][0]=12.0;AS[1][1]=12.0;AS[1][2]=14.0;AS[1][3]=19.0;AS[1][4]=26.0;
AS[1][5]=58.0;AS[1][6]=60.0;AS[1][7]=55.0;
AS[2][0]=14.0;AS[2][1]=13.0;AS[2][2]=16.0;AS[2][3]=24.0;AS[2][4]=40.0;
AS[2][5]=57.0;AS[2][6]=69.0;AS[2][7]=56.0;
AS[3][0]=14.0;AS[3][1]=17.0;AS[3][2]=22.0;AS[3][3]=29.0;AS[3][4]=51.0;
AS[3][5]=87.0;AS[3][6]=80.0;AS[3][7]=62.0;
AS[4][0]=18.0;AS[4][1]=22.0;AS[4][2]=37.0;AS[4][3]=56.0;AS[4][4]=68.0;
AS[4][5]=109.0;AS[4][6]=103.0;AS[4][7]=77.0;
AS[5][0]=24.0;AS[5][1]=35.0;AS[5][2]=55.0;AS[5][3]=64.0;AS[5][4]=81.0;
```

```
AS[5][5]=104.0;AS[5][6]=113.0;AS[5][7]=92.0;
AS[6][0]=49.0;AS[6][1]=64.0;AS[6][2]=78.0;AS[6][3]=87.0;AS[6][4]=103.0;
AS[6][5]=121.0;AS[6][6]=120.0;AS[6][7]=101.0;
AS[7][0]=72.0;AS[7][1]=92.0;AS[7][2]=95.0;AS[7][3]=98.0;AS[7][4]=112.0;
AS[7][5]=100.0;AS[7][6]=103.0;AS[7][7]=99.0;

for(i=0;i < n;i++)
 {
 for(j=0;j < n;j++)
   {
    AS[i][j]=(double)scalor*0.5*AS[i][j];
   }
 }
/***************************************************************************
Start blocking the image and process each block through DCT, Quantization
***************************************************************************/
for(d1=0;d1 < blocks ;d1++)
  {
    for(d2=0;d2 < image_width ;d2=d2+n)
       {
              /* Reading one block of 8x8 pixels from the image file */

              loc=(long)d1*(long)image_width*(long)n+(long)d2;
              for(i=0;i < n;i++)
              {
                     fseek(in_file,loc,SEEK_SET);
                     fread(yr+i*n,n*sizeof(unsigned char),1,in_file);
                     loc=loc+image_width;
              }
              /*level shifting the unsigned pixels to signed pixels*/
              for(i=0;i<n*n;i++)
                     yr_sign[i]=yr[i]-128;
/*************************************************************
   evaluating the 2-D DCT from 1-D DCT using row column method
  *************************************************************/
/* 1.transforming the rows of each block */

for(i=0; i < n;i++)
 {
       for(j=0;j<n;j++)
       {
              xr[j]=(float)(yr_sign[i*n+j]);
       }
```

```
          /* calling 1-D DCT function*/

         dct1(n,m,xr,xrs,CX,L);

         for(j=0;j < n;j++)
          {
                temp[i*n+j]=xrs[j];
          }

   }/* i loop */

 /* 2. Transforming the columns of the semi-transformed matrix resulted from 1 */

 for(i=0; i < n;i++)
  {
         for(j=0;j<n;j++)
          {
             xr[j]=temp[j*n+i];
          }
          /* calling 1-D DCT function*/

         dct1(n,m,xr,xrs,CX,L);

         for(j=0;j < n;j++)
          {
                yro[j*n+i]=xrs[j];
          }

  }/* i loop */
 /************************************************
  Dividing the DCT coefficients by scaling factors
  and rearrange in two dimensional form
 ************************************************/
 for(i=0;i < n;i++)
  {
         for(j=0;j < n;j++)
          {
                if(i==0 && j==0)
                       AB[i][j]=yro[i*n+j]/(8.0);
                else
                       AB[i][j]=yro[i*n+j]/(4.0);
          }
  }
```

```
/*****************************************************
  Divide the transformed block by JPEG scaling matrix
  and ensure proper rounding
*****************************************************/
for(i=0;i < n;i++)
 {
        for(j=0;j < n;j++)
         {
                if(((int)AS[i][j]%2) == 0)
                   XC[i][j]=(AB[i][j]/AS[i][j])+0.5;
                else
                   XC[i][j]=(AB[i][j]/AS[i][j])+(AS[i][j]-1.0)/(2.0*AS[i][j]);
         }
 }
 /*****************************************************************
 Zigzag reordering of the quantized DCT coefficients from 2-D to 1-D,
 rounded by taking truncating, and finally written to output file
 *****************************************************************/
for(i=0;i < n;i++)
 {
        mark=pow((double)-1,(double)i);
        for(c=i;c>=0;c--)
         {
                if(mark < 0)
                        fprintf(fp,"%d\n",(int)XC[c][i-c]);
                else
                        fprintf(fp,"%d\n",(int)XC[i-c][c]);
         }
 }
for(i=n;i < 2*n;i++)
 {
        mark=pow((double)-1,(double)i);
        for(c=n;c> i-n+1;c--)
         {
                if(mark < 0)
                        fprintf(fp,"%d\n",(int)XC[c-1][i-c+1]);
                else
                        fprintf(fp,"%d\n",(int)XC[i-c+1][c-1]);

         }
 }

  }/*end of rowwise blocking*/
 }/*end of columnwise blocking*/
```

```
/* closing input and output files */

fclose(in_file):
fclose(fp):

/* freeing the dynamic allocated arrays */

free(xr):
free(xrs):
free(temp):
free(yr):
free(yro):
free(yr_sign):
free(CX):
free(L):

/******************************************
  Calling subprogram for Huffman codes tables
************************************************/

  CODE_LIST():

/**********************************
  Calling subprogram for Run Length coding
  and Huffman coding
********************************************/

 HUFF():

 }/*main*/

/*********************************************
  End of main program
**********************************
********************
****************
************
********
*****
**
*
```

```
/***************************************************************
  The following function implements a fast 1-D DCT algorithm
   using Vector Radix approach (flowgraph figure 3.3)
****************************************************************/

void dct1(short n,short m,float *xr,float *xrs,float *CX,unsigned int *L)
 {
  short i;
  float n_fl,fm;

/*-------------------------------------
  rearranging the order of input sequence
-------------------------------------*/
for(i=0; i < n/2 ;i++)
 {
        xrs[i]=xr[2*i];
        xrs[(n-i-1)]=xr[2*i+1];
 }

/*-------------------------------------------------------------
  calling butterfly function to evaluate butterflies at each stage
  -----------------------------------------------------------*/

  butterfly(n,m,xrs,CX);

/*-------------------------------
  calling bit reversing function
  ------------------------------*/


  bit_reverse(L,n,m,xrs);

/*---------------------------------------
   calling function to substract odd components
-------------------------------------------*/
   subtract (n,m,xrs);


 }/* end of dct1 function*/
/*******************************
*******************************/
```

```
/*********************************
   1. butterfly function
**********************************/
void butterfly(short n,short m,float *xrs,float *CX)
 {
  int l,i,j,ip,k;
  float u,tr,trr,xrr,le,le1;
  char ch;

  k=0;
  for(l = 1;l <= m;l++)
   {
        le=pow(2,m+1-l);
        le1=le/2;
        for(j = 0;j < le1; j++)
         {
                if(l != m)
                        u=2.0*CX[j+k];
                else
                        u=CX[j+k];

                for(i = j;i < n; i=i+le)
                 {
                        ip=i+le1;

                        /* the even part */

                        tr=xrs[i]+xrs[ip];

                        /*the odd part*/

                        xrr=xrs[i]-xrs[ip];
                        xrs[ip]=xrr*u;
                        xrs[i]=tr;
                 }

         }
         k=k+le1;
   }

  }/* end of butterfly function*/
```

```
/**********************************
    2.bit reversing function
**********************************/
void bit_reverse(unsigned int *L,short n,short m,float *xrs)
{
  unsigned int j,k,i;
  unsigned int *flag;
  float buff;

  flag=(unsigned int *)malloc((n+1)*sizeof(unsigned int));
  for(k=0 ; k < n;k++)
        flag[k]=0;

  for(i=0 ; i < n;i++)
    {
        if(L[i] != i && flag[i]==0)
         {
                buff=xrs[i];
                xrs[i]=xrs[L[i]];
                xrs[L[i]]=buff;
                flag[L[i]]=1;
         }
    }
free(flag);

 }/*end of bit_reverse*/
/*-----------------------------------------
3.substracting the odd components function
-----------------------------------------*/
void subtract(short n,short m,float *xrs)
{
   short i,j,k,index,l,st,st2,st4,st5,st6,st7;
   float st3;

   for(l=1; l < m; l++)
    {
                st2=pow(2,l-1);
                st3=0.5*st2;

                for(i=0; i < st3 ;i++)
                 {
                        if(l==1)
                                xrs[st2]=0.5*xrs[st2];
```

(continued)

```
                    else
                     {
                            st=st2+2*i;
                            xrs[st]=0.5*xrs[st];
                            st++;
                            xrs[st]=0.5*xrs[st];
                     }
              }/* i loop */
   }/* l loop*/


   for(l=1;l < m;l++)
    {
              st4=pow(2,m-l);
              st5=pow(2,l);
              st6=pow(2,m-1-l);
              st7=0.5*st5;

              if( l==m-1 )
               {
                     for(k=1; k < n/2;k++)
                      {
                            index=2*k;
                            xrs[index+1]=xrs[index+1]-xrs[index-1];
                      }
               }
              else
               {
                     for(i=1; i < st5;i++)
                      {
                            for(k=3*st6+(i-1)*st4; k < n/st7+(i-1)*st4;k=k+2)
                             {
                                   xrs[k]=xrs[k]-xrs[k-st4];
                                   xrs[k+1]=xrs[k+1]-xrs[k+1-st4];
                             }
                       }
               }

    }/* l loop */

 }/* end of subtract function*/
```

```
/**************************************************************************
***************************************************************************
** This Subprogram contains the Huffman codes tables ********************** **
** A1 and A3 for DC and AC categories              ************************** **
***************************************************************************
***************************************************************************
#include<stdio.h>
unsigned int dc_code_word[13];
unsigned char dc_code_word_length[13];
unsigned int ac_code_word[17][12];
unsigned char ac_code_word_length[17][12];
unsigned int grey_ac[250];

CODE_LIST()
{
 int i,j,k;
/****************************************
 Huffman codes table A1 (DC categories)
****************************************/
dc_code_word[0]=0;
dc_code_word[1]=2;
dc_code_word[2]=6;
dc_code_word[3]=1;
dc_code_word[4]=5;
dc_code_word[5]=3;
dc_code_word[6]=7;
dc_code_word[7]=15;
dc_code_word[8]=31;
dc_code_word[9]=63;
dc_code_word[10]=127;
dc_code_word[11]=255;

dc_code_word_length[0]=2;
dc_code_word_length[1]=3;
dc_code_word_length[2]=3;
dc_code_word_length[3]=3;
dc_code_word_length[4]=3;
dc_code_word_length[5]=3;
dc_code_word_length[6]=4;
dc_code_word_length[7]=5;
dc_code_word_length[8]=6;
dc_code_word_length[9]=7;
dc_code_word_length[10]=8;
dc_code_word_length[11]=9;
```

```
/*****************************************************************
 Converting the two indices of AC huffman codes table into one index that can be used
for decoding process
  ****************************************************************/
grey_ac[0]=1;
k=1;
for(i=0;i < 15;i++)
 {
  for(j=1;j<=10;j++)
   {
        grey_ac[k]=i*11+j+1;/*  all  indicies  are  shifted  by  one  to  be
                                              used in transition table*/
        k++;
   }
 }

i=15;
  for(j=0;j<=10;j++)
   {
        grey_ac[k]=i*11+j+1;/* all indicies are shifted by one*/
        k++;
   }

/*********************************
  Huffman codes table A3
********************************/

ac_code_word[0][0]=5;/*EOB*/

ac_code_word[0][1]=0;
ac_code_word[0][2]=2;
ac_code_word[0][3]=1;
ac_code_word[0][4]=13;
ac_code_word[0][5]=11;
ac_code_word[0][6]=15;
ac_code_word[0][7]=31;
ac_code_word[0][8]=447;
ac_code_word[0][9]=16895;
ac_code_word[0][10]=49663;
```

```
ac_code_word[1][1]=3;
ac_code_word[1][2]=27;
ac_code_word[1][3]=79;
ac_code_word[1][4]=223;
ac_code_word[1][5]=895;
ac_code_word[1][6]=8703;
ac_code_word[1][7]=41471;
ac_code_word[1][8]=25087;
ac_code_word[1][9]=57855;
ac_code_word[1][10]=4607;

ac_code_word[2][1]=7;
ac_code_word[2][2]=159;
ac_code_word[2][3]=959;
ac_code_word[2][4]=767;
ac_code_word[2][5]=37375;
ac_code_word[2][6]=20991;
ac_code_word[2][7]=53759;
ac_code_word[2][8]=12799;
ac_code_word[2][9]=45567;
ac_code_word[2][10]=29183;

ac_code_word[3][1]=23;
ac_code_word[3][2]=479;
ac_code_word[3][3]=2815;
ac_code_word[3][4]=61951;
ac_code_word[3][5]=2559;
ac_code_word[3][6]=35327;
ac_code_word[3][7]=18943;
ac_code_word[3][8]=51711;
ac_code_word[3][9]=10751;
ac_code_word[3][10]=43519;

ac_code_word[4][1]=55;
ac_code_word[4][2]=127;
ac_code_word[4][3]=27135;
ac_code_word[4][4]=59903;
ac_code_word[4][5]=6655;
ac_code_word[4][6]=39423;
```

```
ac_code_word[4][7]=23039;
ac_code_word[4][8]=55807;
ac_code_word[4][9]=14847;
ac_code_word[4][10]=47615;

ac_code_word[5][1]=47;
ac_code_word[5][2]=1919;
ac_code_word[5][3]=31231;
ac_code_word[5][4]=63999;
ac_code_word[5][5]=1535;
ac_code_word[5][6]=34303;
ac_code_word[5][7]=17919;
ac_code_word[5][8]=50687;
ac_code_word[5][9]=9727;
ac_code_word[5][10]=42495;

ac_code_word[6][1]=111;
ac_code_word[6][2]=1791;
ac_code_word[6][3]=26111;
ac_code_word[6][4]=58879;
ac_code_word[6][5]=5631;
ac_code_word[6][6]=38399;
ac_code_word[6][7]=22015;
ac_code_word[6][8]=54783;
ac_code_word[6][9]=13823;
ac_code_word[6][10]=46591;

ac_code_word[7][1]=95;
ac_code_word[7][2]=3839;
ac_code_word[7][3]=30207;
ac_code_word[7][4]=62975;
ac_code_word[7][5]=3583;
ac_code_word[7][6]=36351;
ac_code_word[7][7]=19967;
ac_code_word[7][8]=52735;
ac_code_word[7][9]=11775;
ac_code_word[7][10]=44543;

ac_code_word[8][1]=63;
ac_code_word[8][2]=511;
ac_code_word[8][3]=28159;
```

```
ac_code_word[8][4]=60927;
ac_code_word[8][5]=7679;
ac_code_word[8][6]=40447;
ac_code_word[8][7]=24063;
ac_code_word[8][8]=56831;
ac_code_word[8][9]=15871;
ac_code_word[8][10]=48639;

ac_code_word[9][1]=319;
ac_code_word[9][2]=32255;
ac_code_word[9][3]=65023;
ac_code_word[9][4]=1023;
ac_code_word[9][5]=33791;
ac_code_word[9][6]=17407;
ac_code_word[9][7]=50175;
ac_code_word[9][8]=9215;
ac_code_word[9][9]=41983;
ac_code_word[9][10]=25599;

ac_code_word[10][1]=191;
ac_code_word[10][2]=58367;
ac_code_word[10][3]=5119;
ac_code_word[10][4]=37887;
ac_code_word[10][5]=21503;
ac_code_word[10][6]=54271;
ac_code_word[10][7]=13311;
ac_code_word[10][8]=46079;
ac_code_word[10][9]=29695;
ac_code_word[10][10]=62463;

ac_code_word[11][1]=639;
ac_code_word[11][2]=3071;
ac_code_word[11][3]=35839;
ac_code_word[11][4]=19455;
ac_code_word[11][5]=52223;
ac_code_word[11][6]=11263;
ac_code_word[11][7]=44031;
ac_code_word[11][8]=27647;
ac_code_word[11][9]=60415;
ac_code_word[11][10]=7167;
```

```
ac_code_word[12][1]=383;
ac_code_word[12][2]=39935;
ac_code_word[12][3]=23551;
ac_code_word[12][4]=56319;
ac_code_word[12][5]=15359;
ac_code_word[12][6]=48127;
ac_code_word[12][7]=31743;
ac_code_word[12][8]=64511;
ac_code_word[12][9]=2047;
ac_code_word[12][10]=34815;

ac_code_word[13][1]=255;
ac_code_word[13][2]=18431;
ac_code_word[13][3]=51199;
ac_code_word[13][4]=10239;
ac_code_word[13][5]=43007;
ac_code_word[13][6]=26623;
ac_code_word[13][7]=59391;
ac_code_word[13][8]=6143;
ac_code_word[13][9]=38911;
ac_code_word[13][10]=22527;

ac_code_word[14][1]=55295;
ac_code_word[14][2]=14335;
ac_code_word[14][3]=47103;
ac_code_word[14][4]=30719;
ac_code_word[14][5]=63487;
ac_code_word[14][6]=4095;
ac_code_word[14][7]=36863;
ac_code_word[14][8]=20479;
ac_code_word[14][9]=53247;
ac_code_word[14][10]=12287;

ac_code_word[15][0]=1279/*ZRL*/;

ac_code_word[15][1]=45055;
ac_code_word[15][2]=28671;
ac_code_word[15][3]=61439;
ac_code_word[15][4]=8191;
ac_code_word[15][5]=40959;
ac_code_word[15][6]=24575;
```

```
ac_code_word[15][7]=57343;
ac_code_word[15][8]=16383;
ac_code_word[15][9]=49151;
ac_code_word[15][10]=32767;

/******************
 AC code word lengths
************************/

ac_code_word_length[0][0]=4;/*EOB*/



ac_code_word_length[0][1]=2;
ac_code_word_length[0][2]=2;
ac_code_word_length[0][3]=3;
ac_code_word_length[0][4]=4;
ac_code_word_length[0][5]=5;
ac_code_word_length[0][6]=7;
ac_code_word_length[0][7]=8;
ac_code_word_length[0][8]=10;
ac_code_word_length[0][9]=16;
ac_code_word_length[0][10]=16;

ac_code_word_length[1][1]=4;
ac_code_word_length[1][2]=5;
ac_code_word_length[1][3]=7;
ac_code_word_length[1][4]=9;
ac_code_word_length[1][5]=11;
ac_code_word_length[1][6]=16;
ac_code_word_length[1][7]=16;
ac_code_word_length[1][8]=16;
ac_code_word_length[1][9]=16;
ac_code_word_length[1][10]=16;

ac_code_word_length[2][1]=5;
ac_code_word_length[2][2]=8;
ac_code_word_length[2][3]=10;
ac_code_word_length[2][4]=12;
ac_code_word_length[2][5]=16;
ac_code_word_length[2][6]=16;
```

```
ac_code_word_length[2][7]=16;
ac_code_word_length[2][8]=16;
ac_code_word_length[2][9]=16;
ac_code_word_length[2][10]=16;

ac_code_word_length[3][1]=6;
ac_code_word_length[3][2]=9;
ac_code_word_length[3][3]=12;
ac_code_word_length[3][4]=16;
ac_code_word_length[3][5]=16;
ac_code_word_length[3][6]=16;
ac_code_word_length[3][7]=16;
ac_code_word_length[3][8]=16;
ac_code_word_length[3][9]=16;
ac_code_word_length[3][10]=16;

ac_code_word_length[4][1]=6;
ac_code_word_length[4][2]=10;
ac_code_word_length[4][3]=16;
ac_code_word_length[4][4]=16;
ac_code_word_length[4][5]=16;
ac_code_word_length[4][6]=16;
ac_code_word_length[4][7]=16;
ac_code_word_length[4][8]=16;
ac_code_word_length[4][9]=16;
ac_code_word_length[4][10]=16;

ac_code_word_length[5][1]=7;
ac_code_word_length[5][2]=11;
ac_code_word_length[5][3]=16;
ac_code_word_length[5][4]=16;
ac_code_word_length[5][5]=16;
ac_code_word_length[5][6]=16;
ac_code_word_length[5][7]=16;
ac_code_word_length[5][8]=16;
ac_code_word_length[5][9]=16;
ac_code_word_length[5][10]=16;

ac_code_word_length[6][1]=7;
ac_code_word_length[6][2]=12;
ac_code_word_length[6][3]=16;
```

```
ac_code_word_length[6][4]=16;
ac_code_word_length[6][5]=16;
ac_code_word_length[6][6]=16;
ac_code_word_length[6][7]=16;
ac_code_word_length[6][8]=16;
ac_code_word_length[6][9]=16;
ac_code_word_length[6][10]=16;

ac_code_word_length[7][1]=8;
ac_code_word_length[7][2]=12;
ac_code_word_length[7][3]=16;
ac_code_word_length[7][4]=16;
ac_code_word_length[7][5]=16;
ac_code_word_length[7][6]=16;
ac_code_word_length[7][7]=16;
ac_code_word_length[7][8]=16;
ac_code_word_length[7][9]=16;
ac_code_word_length[7][10]=16;

ac_code_word_length[8][1]=9;
ac_code_word_length[8][2]=15;
ac_code_word_length[8][3]=16;
ac_code_word_length[8][4]=16;
ac_code_word_length[8][5]=16;
ac_code_word_length[8][6]=16;
ac_code_word_length[8][7]=16;
ac_code_word_length[8][8]=16;
ac_code_word_length[8][9]=16;
ac_code_word_length[8][10]=16;

ac_code_word_length[9][1]=9;
ac_code_word_length[9][2]=16;
ac_code_word_length[9][3]=16;
ac_code_word_length[9][4]=16;
ac_code_word_length[9][5]=16;
ac_code_word_length[9][6]=16;
ac_code_word_length[9][7]=16;
ac_code_word_length[9][8]=16;
ac_code_word_length[9][9]=16;
ac_code_word_length[9][10]=16;
```

```
ac_code_word_length[10][1]=9;
ac_code_word_length[10][2]=16;
ac_code_word_length[10][3]=16;
ac_code_word_length[10][4]=16;
ac_code_word_length[10][5]=16;
ac_code_word_length[10][6]=16;
ac_code_word_length[10][7]=16;
ac_code_word_length[10][8]=16;
ac_code_word_length[10][9]=16;
ac_code_word_length[10][10]=16;

ac_code_word_length[11][1]=10;
ac_code_word_length[11][2]=16;
ac_code_word_length[11][3]=16;
ac_code_word_length[11][4]=16;
ac_code_word_length[11][5]=16;
ac_code_word_length[11][6]=16;
ac_code_word_length[11][7]=16;
ac_code_word_length[11][8]=16;
ac_code_word_length[11][9]=16;
ac_code_word_length[11][10]=16;

ac_code_word_length[12][1]=10;
ac_code_word_length[12][2]=16;
ac_code_word_length[12][3]=16;
ac_code_word_length[12][4]=16;
ac_code_word_length[12][5]=16;
ac_code_word_length[12][6]=16;
ac_code_word_length[12][7]=16;
ac_code_word_length[12][8]=16;
ac_code_word_length[12][9]=16;
ac_code_word_length[12][10]=16;

ac_code_word_length[13][1]=11;
ac_code_word_length[13][2]=16;
ac_code_word_length[13][3]=16;
ac_code_word_length[13][4]=16;
ac_code_word_length[13][5]=16;
ac_code_word_length[13][6]=16;
ac_code_word_length[13][7]=16;
ac_code_word_length[13][8]=16;
```

```
ac_code_word_length[13][9]=16;
ac_code_word_length[13][10]=16;

ac_code_word_length[14][1]=16;
ac_code_word_length[14][2]=16;
ac_code_word_length[14][3]=16;
ac_code_word_length[14][4]=16;
ac_code_word_length[14][5]=16;
ac_code_word_length[14][6]=16;
ac_code_word_length[14][7]=16;
ac_code_word_length[14][8]=16;
ac_code_word_length[14][9]=16;
ac_code_word_length[14][10]=16;

ac_code_word_length[15][0]=11;/*ZRL*/

ac_code_word_length[15][1]=16;
ac_code_word_length[15][2]=16;
ac_code_word_length[15][3]=16;
ac_code_word_length[15][4]=16;
ac_code_word_length[15][5]=16;
ac_code_word_length[15][6]=16;
ac_code_word_length[15][7]=16;
ac_code_word_length[15][8]=16;


ac_code_word_length[15][9]=16;
ac_code_word_length[15][10]=16;

}/*end of CODE_LIST*/
```

```
/***********************************************************************
************************************************************************
** The following subprogram reads the quantized DCT coefficients ***************
** as blocks of 8x8, calculates the zero run lengths preceding  *************** **
** the non-zero AC coefficients, and uses the Huffman codes     ****************
** in CODE_LIST subprogram to encode each block.                ***************
************************************************************************
*********************************************************************** */
#include<stdio.h>
#include<math.h>
#include<float.h>
#include<errno.h>
#include<stdlib.h>
#include<io.h>
#include<malloc.h>

extern short n,image_width,blocks,scalor;
extern unsigned int dc_code_word[13];
extern unsigned char dc_code_word_length[13];
extern unsigned int ac_code_word[17][12];
extern unsigned char ac_code_word_length[17][12];

int run[65];
unsigned char zero_length[65];
unsigned char ac_category_values[65];
int AC_BLOCK_VALUES[65];
int DC_VALUE,PREVIOUS_DC,AC_VALUE,VALUE,number_of_ac_values,K_index;
unsigned char dc_category,ac_category,category,code_length,code_buffer,flag;
unsigned int code_track,code_print,d1,d2;
char bit_count,mark;
FILE *fp4,*fp5;
void return_category_dc();/* function that returns DC category*/
void return_category_ac();/* function that returns AC category*/
void code_write();        /* function that writes the Huffman codes
                             to output file */
HUFF()
{
  int temp;
  short length,j,sign,sum;
  short m,i,r,index,l,b,line;
  char buffer;
```

```
   /* open a file that contains the DCT quantized coefficients */

   if( (fp5 = fopen("out3.img","r")) == NULL)
    {
              printf(" Can't open file 'out3.img'\n");
              exit(1);
    }
   /* open output file to write the compressed data*/

   if (( fp4=fopen("jpeg.img","wb")) == NULL)
    {
              printf(" Can't open output file 'jpeg.img'\n");
              exit(1);
    }
 /*******************************************************
  writting the image_width and the matrix scaling factor
  to the header of the ouput file
 *******************************************************/
 code_track=(unsigned int)image_width;
 code_buffer=(unsigned char )code_track;
 fputc(code_buffer,fp4);
 code_track >>=8;
 code_buffer=(unsigned char)(code_track);
 fputc(code_buffer,fp4);
 fputc((unsigned char)scalor,fp4);
 /**************************************************************
 Reading the quantized DCT coefficients as blocks of 8x8, calculating the zeros length
 preceding the nonzero AC coeff.
 **************************************************************/
 PREVIOUS_DC=0;
 for(d1=0;d1 < blocks ;d1++)
   {
     for(d2=0;d2 < image_width ;d2=d2+n)
        {
              flag=0;
              for(i=0;i<64;i++)
               {
                 fscanf(fp5,"%d",&temp);
                 run[i] = temp;
                 zero_length[i]=0;
               }
```

```
/* applying 1st order DPCM on DC coefficients*/
                DC_VALUE=run[0]-PREVIOUS_DC;

                PREVIOUS_DC=run[0];

        /*get the category for the dc value*/

            return_category_dc();

          /*write the code for dc values categories.*/

           code_track=dc_code_word[dc_category];
           code_length=dc_code_word_length[dc_category];
           VALUE=DC_VALUE;
           category=dc_category;
           K_index=255;
           code_write();

           /*1.calculating  the  AC  categories  and  the  length  of
                zeros preceding the nonzero AC coeff.****/

            K_index=0;
            for(j=1;j < 64;j++)
             {
                        if(run[j]==0)
                         {
                               zero_length[K_index]++;
                               if(zero_length[K_index]==16)/* ZRL */
                                {
                                      zero_length[K_index]=15;
                                      AC_BLOCK_VALUES[K_index]=0;
                                      zero_length[K_index+1]=1;
                                      K_index++;
                                }
                         }
               else
                  {
                     AC_BLOCK_VALUES[K_index]=run[j];
                        K_index++;
                  }
             }/* j loop */
```

```
                number_of_ac_values=K_index:
                for(i = K_index-1: i >= 0:i--)
                  {
                          if(zero_length[i]==15)
                                number_of_ac_values--:
                          else
                                break:
                  }
                zero_length[i+1]=0:/* EOB */
                AC_BLOCK_VALUES[i+1]=0:/* EOB */

                /*2.Getting the categories of AC values*/

        for(K_index=0:K_index<=number_of_ac_values: K_index++)
                  {
                          if(AC_BLOCK_VALUES[K_index]==0)
                                ac_category_values[K_index]=0:
                          else
                           {
                          AC_VALUE=AC_BLOCK_VALUES[K_index]:
                          return_category_ac():
                          ac_category_values[K_index]=ac_category:
                           }
                  }
        /*3.writting the code for ac values categories.*/

        flag=1:
        for( K_index=0:K_index <= number_of_ac_values: K_index++)
          {
              VALUE=AC_BLOCK_VALUES[K_index]:
              code_track = ac_code_word[zero_length[K_index]]
                                     [ac_category_values[K_index]]:
              code_length   =   ac_code_word_length[zero_length[K_index]]
                                     [ac_category_values[K_index]]:
              category=ac_category_values[K_index]:

              code_write():
          }
  }/*end of column blocking*/
}/*end of row blocking*/
```

```
/* closing the files */
fclose(fp5);
fclose(fp4);
}/*end of main program*/
/************************************
  The following function returns
  the category for the DC value
************************************/
void return_category_dc()
 {
 if(DC_VALUE ==0)
      dc_category=0;

 else if(DC_VALUE ==1 || DC_VALUE ==-1)
      dc_category=1;

 else if((DC_VALUE >= 2 && DC_VALUE <= 3) || (DC_VALUE >=-3 &&
                                            DC_VALUE <= -2))
      dc_category=2;

 else if((DC_VALUE >=4 && DC_VALUE <= 7) || (DC_VALUE >=-7 &&
                                            DC_VALUE <= -4))
      dc_category=3;

 else if((DC_VALUE >=8 && DC_VALUE <= 15) || (DC_VALUE >=-15 &&
                                            DC_VALUE <= -8))
      dc_category=4;

 else if((DC_VALUE >=16 && DC_VALUE <= 31) || (DC_VALUE >=-31 &&
                                            DC_VALUE <= -16))
      dc_category=5;

 else if((DC_VALUE >=32 && DC_VALUE <= 63) || (DC_VALUE >=-63 &&
                                            DC_VALUE <= -32))
      dc_category=6;

 else if((DC_VALUE >=64 && DC_VALUE <= 127) || (DC_VALUE >=-127 &&
                                                 DC_VALUE <= -64))
      dc_category=7;
```

```
else if((DC_VALUE >=128 && DC_VALUE <= 255) || (DC_VALUE >=-255 &&
                                                   DC_VALUE <= -128))
   dc_category=8;

else if((DC_VALUE >=256 && DC_VALUE <= 511) || (DC_VALUE >=-511 &&
                                                   DC_VALUE <= -256))
   dc_category=9;

else if((DC_VALUE >=512 && DC_VALUE <= 1023) || (DC_VALUE >=-1023 &&
                                                   DC_VALUE <= -512))
   dc_category=10;

else if((DC_VALUE >=1024 && DC_VALUE <= 2047) || (DC_VALUE >=-2047 &&
                                                   DC_VALUE <= -1024))
   dc_category=11;

}/*end of return_category_dc*/
/************************************
 the following function returns
 the category for the ac value
************************************/
void return_category_ac()
{
  if( AC_VALUE == 1 || AC_VALUE == -1 )
    ac_category=1;

  else if((AC_VALUE >=2 && AC_VALUE <= 3) || (AC_VALUE >=-3 &&
                                               AC_VALUE <= -2))
    ac_category=2;

  else if((AC_VALUE >=4 && AC_VALUE <= 7) || (AC_VALUE >=-7 &&
                                               AC_VALUE <= -4))
    ac_category=3;

  else if((AC_VALUE >=8 && AC_VALUE <= 15) || (AC_VALUE >=-15 &&
                                               AC_VALUE <= -8))
    ac_category=4;

  else if((AC_VALUE >=16 && AC_VALUE <= 31) || (AC_VALUE >=-31 &&
                                               AC_VALUE <= -16))
    ac_category=5;
```

```
  else if((AC_VALUE >=32 && AC_VALUE <= 63) || (AC_VALUE >=-63 &&
                                                AC_VALUE <= -32))
    ac_category=6;

  else if((AC_VALUE >=64 && AC_VALUE <= 127) || (AC_VALUE >=-127 &&
                                                AC_VALUE <= -64))
    ac_category=7;

  else if((AC_VALUE >=128 && AC_VALUE <= 255) || (AC_VALUE >=-255 &&
                                                AC_VALUE <= -128))
    ac_category=8;

  else if((AC_VALUE >=256 && AC_VALUE <= 511) || (AC_VALUE >=-511 &&
                                                AC_VALUE <= -256))
    ac_category=9;

  else if((AC_VALUE >=512 && AC_VALUE <= 1023) || (AC_VALUE >=-1023 &&
                                                AC_VALUE <= -512))
    ac_category=10;
 }/*end of return_category_ac*/
/****************************************
 code writting function for DC values
******************************************/
void code_write()
 {
  char length_count;
  short length,j,i,track,sign;

            /*1. writting the code worde to the output buffer*/
                 if(d1==0  &&  d2==0  &&  flag==0)
                  {
                         bit_count=0;
                         code_buffer=0;
                  }
                 length_count=0;
                 while(length_count < code_length)
                   {
                         if((code_track%2)==0)
                                 code_buffer=code_buffer | 0;
                         else
                                 code_buffer=code_buffer | 128;
```

```
                code_track >>= 1;
                bit_count++;
                length_count++;

                if(bit_count != 8)
                        code_buffer>>=1;

                mark=0;
                if(bit_count==8)
                  {
                        fputc(code_buffer,fp4);
                        code_buffer=0;
                        bit_count=0;
                        mark=1;
                }
          }/* while loop*/

/*2. Masking the sign bit */

        if(category != 0)
         {
                if(VALUE < 0)
                        code_buffer=code_buffer | 128;
                if(VALUE > 0)
                        code_buffer=code_buffer |  0 ;

                bit_count++;

                if(bit_count != 8)
                        code_buffer>>=1;

                mark=0;
                if(bit_count==8)
                 {
                        fputc(code_buffer,fp4);
                        code_buffer=0;
                        bit_count=0;
                        mark=1;
                 }
         }/*if category*/
```

```
                /*3.writting the actual value to the output buffer*/
                if(category != 0)
                  {
                        code_track=abs(VALUE);
                        length_count=0;
                        while(length_count < category)
                          {
                                if((code_track%2)==0)
                                        code_buffer=code_buffer | 0;
                                else
                                        code_buffer=code_buffer | 128;

                                code_track >>= 1;
                                bit_count++;
                                length_count++;

                                if(bit_count != 8)
                                        code_buffer>>=1;

                                mark=0;
                                if(bit_count==8)
                                  {
                                        fputc(code_buffer,fp4);
                                        code_buffer=0;
                                        bit_count=0;
                                        mark=1;
                                  }
                        }/* while loop*/
                  }/*if category*/
                        if(K_index==number_of_ac_values)
                          {
                                if(mark==0)
                              {
                                        code_buffer>>=(7-bit_count);
                                        fputc(code_buffer,fp4);
                                        code_buffer=0;
                                        bit_count=0;
                                        mark==1;
                                  }
                          }
        }/**end of code_write function***/
```

```
/******************************************************************** **
**********************************************************************
**The following program implements the JPEG baseline  *************************
**decoder which includes:                             **************************
**          1. Huffman decoding                       ****************************
**          2. Run Length decoding                    ****************************
**          3. Multiplication by JPEG scaling matrix ************************ **
**          4. 8x8 IDCT                               *************************** **
**********************************************************************
*********************************************************************/
#include<stdio.h>
#include<math.h>
#include<float.h>
#include<errno.h>
#include<stdlib.h>
#include<io.h>
#include<malloc.h>
#include<memory.h>
#define ROWS 8

extern unsigned int dc_code_word[13];
extern unsigned char dc_code_word_length[13];
extern unsigned int ac_code_word[17][12];
extern unsigned char ac_code_word_length[17][12];
extern unsigned int grey_ac[250];

unsigned char n,blocks,select,scalor;
unsigned char cod_buffer,bit_count,k_index,k;
unsigned int image_width,new_state,index,state,head;
int *table_dc[25];
int *table_ac[600];
unsigned char grey_dc[13];
unsigned char zero_length[65];
int AC_BLOCK_VALUES[65];
int run[65];
unsigned int category_values[65];
unsigned int ac_category_values[65];
unsigned int dc_category,ac_category;
int AC_VALUE,DC_VALUE,past_dc;
unsigned int DC_BUFFER,AC_BUFFER;
FILE *in_file,*huff_fp;
```

```
void open_files();
void memory_alloc();
void transition_table_dc(); /* function to generate transition
                                table for DC categories */

void transition_table_ac(); /* function to generate transition
                                table for AC indices*/
generate_ac_table(i,j);
void return_zerolength_and_ac_cat(int i);
void output();
void get_dc_value();
void get_ac_value(int i);
void close_files();
void IQUZ();
void CODE_LIST();
main()
{
  int i;
  unsigned char temp,code_buffer1,code_buffer2;
  unsigned int code_track;

/*******************************************
  Open compressed file and get the image width
  and JPEG matrix scaling factor from header
*******************************************/
  open_files();

  code_buffer1=fgetc(in_file);
  code_buffer2=fgetc(in_file);

  code_track=(unsigned int)code_buffer2;
  code_track <<= 8;
  code_track=code_track | code_buffer1;
  image_width=code_track;
  scalor=(int)fgetc(in_file);

  n=8;
  blocks=image_width/n;
  past_dc=0;
```

```
  /* Calling the subprogram for Huffman codes*/

       CODE_LIST( ): /*external function*/
  /* Allocating memory */

       memory_alloc( ):

  /*The following function generates transition table for dc values*/

  for(i=0:i<12:i++)
    grey_dc[i]=i+1:/* index 0 will not work in transition table*/

  transition_table_dc():

  /*The following function generates transition table for ac value*/

  transition_table_ac():

  /* The following function decodes the compressed blocks by aid
       of DC and AC transition tables*/

  output():

 /* closing files function*/

  close_files():

 /* freeing some allocated memory*/
  for(i=0:i<25:i++)
     free(*(table_dc+i)):

  for(i=0:i<600:i++)
     free(*(table_ac+i)):
/**************************************
 Calling subprogram for dequantization
 and IDCT
***************************************/
IQUZ():

}/*end of main program*/
/***************************/
```

```
/**********************
  open files function
***********************/
void open_files()
{
  if( (in_file = fopen("jpeg.img","rb")) == NULL)
   {
        printf(" Can't open file `jpeg.img'\n");
        exit(1);
   }
   if((huff_fp= fopen("huff.img","w")) == NULL)
   {
        printf(" Can't open file `huff.img'\n");
        exit(1);
   }
}/*end of open function*/
/**************************
 memory allocation function
****************************/
void memory_alloc()
{
   int i,j;

for(i=0;i<25;i++)
  {
        *(table_dc+i)=(int *)malloc(2*sizeof(int));
        if(*(table_dc+i)==NULL)
         {
                printf("insufficient memory for table_dc\n");
                exit(1);
         }
  }
for(i=0;i<600;i++)
  {
        *(table_ac+i)=(int *)malloc(2*sizeof(int));
        if(*(table_ac+i)==NULL)
         {
                printf("insufficient memory for table_ac\n");
                exit(1);
         }
  }
```

```
/* initializing the transition tables to blanks */
for(i=0;i < 25;i++)
  {
        for(j=0;j <= 1;j++)
                table_dc[i][j]=-1000;
  }
for(i=0;i < 600;i++)
  {
        for(j=0;j <= 1;j++)
                table_ac[i][j]=-1000;
  }
}/*end of memory allocation function*/
/*********************************************************
 function to generate transition table  for DC categories
*********************************************************/
void transition_table_dc()
{
  unsigned  int cod_track;
  short i,j;


new_state=1;

for(i = 0; i < 12; i++)
 {
        state=0;
        cod_track=dc_code_word[i];

        for(k=1;k <= dc_code_word_length[i]; k++)
         {
                if((cod_track%2)==0)
                        head=0;
                else
                        head=1;

                if(table_dc[state][head]== -1000)
                 {
                        table_dc[state][head]=new_state;
                        state=new_state;
                        new_state++;
                 }
```

```
                else
                        state=table_dc[state][head];

                        cod_track>>=1;
        }/* k loop*/

        table_dc[state][head]=-grey_dc[i];

 }

}/*end of dc transition table function*/

/*********************************************************
 function to generate transition table for AC categories
**********************************************************/
void transition_table_ac()
{
  int i,j;

  new_state=1;
  i=0;
  j=0;
  index=0;

  generate_ac_table(i,j);

for(i=0;i < 15;i++)
 {
        for(j=1;j < 11;j++)
                generate_ac_table(i,j);

 }

i=15;
for(j=0;j < 11;j++)
        generate_ac_table(i,j);

}/*end of ac transition table function*/
```

```
/*************************************************************
 The following function decodes the compressed blocks
 into zero-length and AC and DC values
 *************************************************************/

void output()
{

 int i,j,l,d1,d2;
 unsigned int cod_track;

 for(d1=0;d1 < blocks ;d1++)
  {
    for(d2=0; d2 < image_width ;d2=d2+n)
      {
              DC_BUFFER=0;
              AC_BUFFER=0;
              state=0;
              i=0;
              k_index=0;

           /* get a byte from the file */
              cod_buffer=fgetc(in_file);

              bit_count=0;
              while(bit_count < 8)
                {
                     if((cod_buffer%2)==0)  /* check the LSB */
                            head=0;
                     else
                            head=1;

                     if(i == 0)
                            new_state=table_dc[state][head];
                     else
                            new_state=table_ac[state][head];


                     state=new_state;
```

```
if(i == 0)  /* decode the first byte as DC value*/
 {
        if(table_dc[state][head] <= 0 && table_dc[state][head] !=
                                                              -1000)
         {
                get_dc_value();
                state=0;
                i++;
         }/*if table */
        else if(table_dc[state][head]==-1000)
         {
                printf(" error in dc code generation\n");
                exit(1);
         }
 }/*end of if i==0 */
if(i != 0)
 {
        if(table_ac[state][head] < 0 && table_ac[state][head] !=
                                                              -1000)
         {
                grey_ac[i]=(unsigned int)abs(table_ac[state][head]);
                /*getting the length of zeros and the
                corresponding category*/

                return_zerolength_and_ac_cat(i);

                /* checking for End Of Block */

                if(ac_category_values[k_index]==0   &&
                        zero_length[k_index]==0 )
                  {
                        AC_BLOCK_VALUES[k_index]=0;
                        goto EOB;
                  }
                /* checking if zero_length is greater than 15*/
                if(ac_category_values[k_index]==0   &&
                        zero_length[k_index]==15)
                 {
                        AC_BLOCK_VALUES[k_index]=0;
                        goto ZERO_AC;
                 }
```

```
                                        /* decode the AC value */
                                        get_ac_value(i);

    ZERO_AC:                            state=0;
                                        AC_BUFFER=0;
                                        i++;
                                        k_index++;
                                   }/* if table < 0  */

                                  else if(table_ac[state][head]==-1000)
                                   {
                                        printf(" error in AC code generation\n");
                                        exit(1);
                                   }
                             }/*end of i != 0 */

                             cod_buffer >>=1;
                             bit_count++;
                             if(bit_count==8)
                              {
                                   cod_buffer=fgetc(in_file);
                                   bit_count=0;
                              }
                         }/*while bit count loop*/
/****************************************************************
  getting the locations of AC values using the lengths of zeros
 ****************************************************************/
EOB:         k_index=0;
             run[0]=DC_VALUE;
             l=1;
          for(;;)
             {
             if(zero_length[k_index]==0 && AC_BLOCK_VALUES[k_index]==0)
                   break;

                   if(zero_length[k_index]==0)
                     {
                        run[l]=AC_BLOCK_VALUES[k_index];
                        l++;
                        goto stage;
                     }
```

```
                    for(i=0; i < zero_length[k_index];i++)
                      {
                            run[l]=0;
                            l++;
                      }

                    if(zero_length[k_index] != 15 && zero_length[k_index] != 0)
                     {
                            run[l]=AC_BLOCK_VALUES[k_index];
                            l++;
                     }


stage:              k_index++;

                 }/*for;; loop*/
                /* adding zeros to complete 64 elements */

                for(i=l;i<n*n;i++)
                        run[i]=0;

/*****************************************
   writting the block values to the o/p file
*****************************************/
for(i=0;i < n*n;i++)
  {
        if(i==0)
         {
                /* Inverse DPCM to get the DC coefficient*/
                run[i]=run[i]+past_dc;
                past_dc=run[i];
         }
        fprintf(huff_fp,"%d\n",run[i]);
  }


 }/*end of rowwise blocking*/
}/*end of columnwise blocking*/

}/*end of output function*/
```

```
/*********************************************
  function to get the dc value from the tables
  using the category ,sign bit and and next bits
**********************************************/
void get_dc_value()
{
  int  sign;

        dc_category=(unsigned char)abs(table_dc[state][head]);
        dc_category--;
        if(dc_category==0)
         {
                DC_VALUE=0;
                goto  end;
         }
      /*getting the sign*/

        cod_buffer >>=1;
        bit_count++;

       if(bit_count==8)
         {
                cod_buffer=fgetc(in_file);
                bit_count=0;
         }
        if((cod_buffer%2)==0)
                sign=0;
        else
                sign=1;
        /*getting the actual value of dc*/

        for(k=0;k < dc_category;k++)
         {
                cod_buffer >>=1;
                bit_count++;

                if(bit_count==8)
                 {
                        cod_buffer=fgetc(in_file);
                        bit_count=0;
                 }
```

```
            if((cod_buffer%2)==0)
                    DC_BUFFER=DC_BUFFER | 0;
            else
                    DC_BUFFER=DC_BUFFER | 32768;

            DC_BUFFER >>= 1;
        }/*k loop*/

        DC_BUFFER >>= (15-dc_category);

        if(sign==0)
                DC_VALUE=(int)DC_BUFFER;
        else
                DC_VALUE=(int)(-DC_BUFFER);

end:    sign=0;

}/*end of get_dc_value function*/
/****************************************************
  function that returns the zero length and category
****************************************************/
void return_zerolength_and_ac_cat(int i)
 {

        ac_category_values[k_index]=(grey_ac[i]-1)%11;
        zero_length[k_index]=((grey_ac[i]-1)-ac_category_values[k_index])/11;

 }/*end of return_zerolength_ac_category function*/

/**************************************
  function to get the actual AC value
**************************************/
void get_ac_value(int i)
{
 int sign,l;

        /*getting the sign*/

        cod_buffer >>=1;
        bit_count++;
```

```
        if(bit_count==8)
         {
                cod_buffer=fgetc(in_file);
                bit_count=0;
         }

        if((cod_buffer%2)==0)
                sign=0;
        else
                sign=1;

   /*getting the actual AC value */

        for(l=0;l < ac_category_values[k_index];l++)
          {
                cod_buffer >>=1;
                bit_count++;
                if(bit_count==8)
                 {
                        cod_buffer=fgetc(in_file);
                        bit_count=0;
                 }

                if((cod_buffer%2)==0)
                        AC_BUFFER=AC_BUFFER | 0;
                else
                        AC_BUFFER=AC_BUFFER | 32768;

                AC_BUFFER >>=1;
          }/*l loop*/

        AC_BUFFER=AC_BUFFER >> (15-ac_category_values[k_index]);

        if(sign==0)
                AC_BLOCK_VALUES[k_index]=(int)AC_BUFFER;
        else
                AC_BLOCK_VALUES[k_index]=(int)(-AC_BUFFER);

}/*end of get_ac_value*/
```

```
/**********************************************************
  function that generates transition table for ac values
***********************************************************/
void generate_ac_table(int i,int j)
{
        int state,head;
        unsigned  int cod_track;

        state=0;
        cod_track=ac_code_word[i][j];

        for(k=1; k <= ac_code_word_length[i][j];k++)
          {
                if((cod_track%2)==0)
                        head=0;
                else
                        head=1;

                if(table_ac[state][head]== -1000)
                  {
                        table_ac[state][head]=new_state;
                        state=new_state;
                        new_state++;
                  }
                else
                        state=table_ac[state][head];

                cod_track>>=1;

          }/*k loop */

        table_ac[state][head]=-grey_ac[index];
        index++;

 }/*end of generate_ac_table*/
/*  closing files function */
void close_files()
{
        fclose(in_file);
        fclose(huff_fp);
}
```

```
/***************************************************************************
****************************************************************************
** This subprogram implements dequantization by multiplying by   *************
** JPEG quantization matrix, get the original block pixels by IDCT.**************
** and unblocking to get the image back.                    ******************
****************************************************************************
***************************************************************************/

#include<stdio.h>
#include<math.h>
#include<float.h>
#include<errno.h>
#include<stdlib.h>
#include<io.h>
#include<malloc.h>
#define ROWS 8

extern unsigned char n,blocks,scalor;
extern unsigned int image_width;

double AS[ROWS][ROWS];
double zig[ROWS][ROWS];
double XX[ROWS][ROWS];
void dct1(short n,short m,float *xr,float *xrs,float *CX,unsigned int *L);
void butterfly(short n,short m,float *xrs,float *CX);
void bit_reverse(unsigned int *L,short n,short m,float *xrs);
void addition(short n,short m,float *xrs);

IQUZ()
{
   float *xr,*xrs,*temp,*CX,buffer1,*yr;
   unsigned char *yro;
   char *yro_sign;
   unsigned int *L,buffer,count;
   short m,i,j,r,k,index,l,le,le1,temp1;
   short d1,d2,z,c,mark;
   long loc,loco;
   unsigned int MASK,C,A;
   double sum;
   float pi=3.14159265358979;
   FILE *in_file,*fp;
```

```
  /* open file to read the quantized coefficients*/

  if( (in_file = fopen("huff.img","r")) == NULL)
  {
       printf(" Can't open file'huff.img' \n");
       exit(1);
  }

  m=(int)(log10((double)n)/log10((double)2.0)+0.5);

  if( (fp= fopen("back.img","wb")) == NULL)
  {
       printf(" Can't open file'back.img' \n");
       exit(1);
  }
/**********************
 allocating memory
 **********************/
 xr=(float *)malloc((n+1)*sizeof(float));
 xrs=(float *)malloc((n+1)*sizeof(float));
 temp=(float *)malloc((n+1)*(n+1)*sizeof(float));
 yr=(float *)malloc((n+1)*(n+1)*sizeof(float));
 CX=(float *)malloc(n*sizeof(float));
 yro=(unsigned char *)malloc((n+1)*(n+1)*sizeof(unsigned char));
 yro_sign=( char *)malloc((n+1)*(n+1)*sizeof( char));
 L=(unsigned int *)malloc(n*sizeof(unsigned int));
/****************************************
  generating LUT for cosine factors
 ****************************************/
  k=0;
  for(l = m;l >= 1;l--)
  {
      le=pow(2,m+1-l);
      le1=le/2;
      for(j = 0;j < le1; j++)
        {
          CX[j+k]=cos(pi*(4.0*j+1)/(2.0*le));
        }
        k=k+le1;
  }
```

```
/****************************************
  generating LUT for bit reversing
****************************************/
 for(k=0:k < n:k++)
    {
      MASK=1:
      C=0:
      for(i=0,j=m-1:i < m:i++,j--)
         {
           A=(k&MASK) >>i:
           A<<=j:
           C |=A:
           MASK=MASK << 1:
         }
        L[k]=C:
     }
/******************************
 JPEG  scaling matrix
********************************/
AS[1][1]=16.0:AS[1][2]=11.0:AS[1][3]=10.0:AS[1][4]=17.0:AS[1][5]=24.0:
AS[1][6]=40.0:AS[1][7]=51.0:AS[1][8]=61.0:
AS[2][1]=12.0:AS[2][2]=12.0:AS[2][3]=14.0:AS[2][4]=19.0:AS[2][5]=26.0:
AS[2][6]=58.0:AS[2][7]=60.0:AS[2][8]=55.0:
AS[3][1]=14.0:AS[3][2]=13.0:AS[3][3]=16.0:AS[3][4]=24.0:AS[3][5]=40.0:
AS[3][6]=57.0:AS[3][7]=69.0:AS[3][8]=56.0:
AS[4][1]=14.0:AS[4][2]=17.0:AS[4][3]=22.0:AS[4][4]=29.0:AS[4][5]=51.0:
AS[4][6]=87.0:AS[4][7]=80.0:AS[4][8]=62.0:
AS[5][1]=18.0:AS[5][2]=22.0:AS[5][3]=37.0:AS[5][4]=56.0:AS[5][5]=68.0:
AS[5][6]=109.0:AS[5][7]=103.0:AS[5][8]=77.0:
AS[6][1]=24.0:AS[6][2]=35.0:AS[6][3]=55.0:AS[6][4]=64.0:AS[6][5]=81.0:
AS[6][6]=104.0:AS[6][7]=113.0:AS[6][8]=92.0:
AS[7][1]=49.0:AS[7][2]=64.0:AS[7][3]=78.0:AS[7][4]=87.0:AS[7][5]=103.0:
AS[7][6]=121.0:AS[7][7]=120.0:AS[7][8]=101.0:
AS[8][1]=72.0:AS[8][2]=92.0:AS[8][3]=95.0:AS[8][4]=98.0:AS[8][5]=112.0:
AS[8][6]=100.0:AS[8][7]=103.0:AS[8][8]=99.0:

for(i=1:i <= n:i++)
 {
        for(j=1:j <= n:j++)
               AS[i-1][j-1]=scalor*0.5*AS[i][j]:
 }
```

```
/**************************************************************************
 Reading the input file as blocks of 8x8, multiplying each by quantization
matrix, taking 8x8 IDCT , and put the blocks together to perform the final image.
**************************************************************************/
for(d1=0;d1 < blocks ;d1++)
  {
     for(d2=0;d2 < image_width ; d2=d2+n)
        {
           /* 1. reading blocks of 8x8 */
              for(i=0;i < n*n;i++)
                 {
                    fscanf(in_file,"%d",&temp1);
                    yr[i]=(float)temp1;
                 }
           /* zigzag reordering from 1-D to 2-D*/
              k=0;
              for(i=0;i < n;i++)
                {
                     mark=pow((double)-1,(double)i);
                     for(c=i;c>=0;c--)
                       {
                          if(mark < 0)
                               zig[c][i-c]=(double)yr[k];
                          else
                               zig[i-c][c]=(double)yr[k];
                          k++;
                       }
                }
              for(i=n;i < 2*n;i++)
                {
                     mark=pow((double)-1,(double)i);
                     for(c=n;c > i-n+1;c--)
                       {
                          if(mark < 0)
                               zig[c-1][i-c+1]=(double)yr[k];
                          else
                               zig[i-c+1][c-1]=(double)yr[k];
                          k++;
                       }
                }
```

```
for(i=0;i <  n;i++)
  {
        for(j=0;j < n;j++)
          {
                XX[i][j]=zig[i][j]*AS[i][j];

                if(i==0 && j==0)
                        yr[i*n+j]=(float)(XX[i][j]*(double)8.0);
                else
                        yr[i*n+j]=(float)(XX[i][j]*(double)4.0);

          }
  }
/*************************************
  evaluating the 2-D IDCT from
  1-D IDCT using row column method
*************************************/
/* 1. row wise IDCT */
for(i=0; i < n;i++)
  {
        for(j=0;j<n;j++)
          {
                xrs[j]=(float)yr[i*n+j];
          }
        /* calling 1-D IDCT function*/

        dct1(n,m,xr,xrs,CX,L);

        for(j=0;j < n;j++)
          {
                temp[i*n+j]=xr[j];
          }

  }
/* 2. column wise dct*/
for(i=0; i < n;i++)
  {
        for(j=0;j<n;j++)
          {
                xrs[j]=temp[j*n+i];
          }
```

```
          /* calling 1-D DCT function*/

               dct1(n,m,xr,xrs,CX,L);

               for(j=0;j < n;j++)
                 {
                       yro_sign[j*n+i]=(char)(xr[j]);
                 }

         }/* i loop */

       /* level shifting the signed pixels to unsigned pixels*/
          for(i=0;i < n*n;i++){
               yro[i]=yro_sign[i]+128;
       /* make sure the pixel values are not exceeding 0-255 range */
               if(yro[i] > 255)
                 yro[i]=255;
               else if(yro[i] < 0)
                 yro[i]=0;
          }

       /******************************************
       writting the recovered pixels to o/p file
       as blocks of 8x8 pixels.
       ******************************************/
       loco=(long)d1*(long)image_width*(long)n+(long)d2;
       for(i=0;i<n;i++)
         {
               fseek(fp,loco,SEEK_SET);
               fwrite(yro+i*n,n*sizeof(unsigned char),1,fp);
               loco=loco+image_width;
         }

 }/*end of rowwise blocking*/
}/*end of columnwise blocking*/
/* closing input and output files*/
        fclose(in_file);
        fclose(fp);
}/*end of main program*/
/***************************
*********************/
```

```
/*****************************************************
  The following function implements the inverse 1-D DCT
  ( flowgraph figure 3.3 by going backwards)
 *****************************************************/
void dct1(short n,short m,float *xr,float *xrs,float *CX,unsigned int *L)
 {
  short i;
  float n_fl,fm;
  char ch;

  /*-------------------------------------------
    calling function to add the odd components
  ---------------------------------------------*/

      addition (n,m,xrs);

  /*---------------------------------
    calling bit reversing function
   -------------------------------*/

      bit_reverse(L,n,m,xrs);

  /*------------------------------------------------------------
   calling butterfly function to evaluate butterflies at each stage
  ------------------------------------------------------------*/

      butterfly(n,m,xrs,CX);

  /*-------------------------------------
   rearranging the order of input sequence
  -------------------------------------*/
      for(i=0; i < n/2 ;i++)
        {
              xr[2*i]=xrs[i];
              xr[2*i+1]=xrs[n-i-1];
        }
}/* end of dct1 function*/
/******************************
****************************
***********************/
```

```
/*-----------------------------------------
    3. butterfly function
-----------------------------------------*/
void butterfly(short n.short m.float *xrs.float *CX)
 {
   int l.i.j.ip.k:
   float u.tr.trr.xrr.le.le1:
   char ch:

        k=0:
        for(l = m:l >= 1:l--)
          {
                le=pow(2.m+1-l):
                le1=le/2:
                for(j = 0:j < le1: j++)
                  {
                        if(l != m)
                                u=2.0*CX[j+k]:
                        else
                                u=CX[j+k]:

                        for(i = j:i < n: i=i+le)
                          {
                                ip=i+le1:
                                xrs[ip]=xrs[ip]/u:

                                /* the even part */

                                tr=0.5*(xrs[i]+xrs[ip]):

                                /*the odd part*/

                                xrr=0.5*(xrs[i]-xrs[ip]):
                                xrs[ip]=xrr:
                                xrs[i]=tr:
                          }/* i loop */

                  }/* j loop */
                k=k+le1:
          }/* l loop */
  }/* end of butterfly function*/
```

```
/*------------------------
     2.bit reversing function
------------------------*/
void bit_reverse(unsigned int *L,short n,short m,float *xrs)
{
 unsigned int j,k,i;
 unsigned int *flag;
 float buff;

     /* allocating memory for flag */
       flag=(unsigned int *)calloc(n*sizeof(unsigned int));


       for(i=0;i<n;i++)
         {
              if(L[i] != i && flag[i]==0)
                {
                     buff=xrs[i];
                     xrs[i]=xrs[L[i]];
                     xrs[L[i]]=buff;
                     flag[L[i]]=1;
                }
         }

       free(flag);
 }/*end of bit_reverse*/
/*----------------------------------------
  1.adding the odd components function
----------------------------------------*/
void addition(short n,short m,float *xrs)
{

 short i,j,k,count,index,l,st,st2,st4,st5,st6,st7;
 float st3;

       for(l=1;l < m;l++)
         {
             st4=pow(2,m);
             st5=pow(2,l-2);
             st6=pow(2,l);
             st7=pow(2,m-l)-1;
```

```
                if( l==1 )
                  {
                        for(k=n/2-1; k >= 1;k--)
                          {
                                index=2*k;
                                xrs[index+1]=xrs[index+1]+xrs[index-1];
                          }
                  }
                else
                  {
                        for(i=1; i <= st5 ;i++)
                          {
                                k=st4-2-2*(i-1);
                                for(count=1;count <= st7;count++)
                                  {
                                        xrs[k]=xrs[k]+xrs[k-st6];
                                        xrs[k+1]=xrs[k+1]+xrs[k+1-st6];
                                        k=k-st6;
                                  }
                          }
                  }
          }/* l loop */

        for(l=1;l < m; l++)
          {
                st2=pow(2,l-1);
                st3=0.5*st2;
                for(i=0; i < st3 ;i++)
                  {
                        if(l==1)
                                xrs[st2]=2*xrs[st2];
                        else
                          {
                                st=st2+2*i;
                                xrs[st]=2*xrs[st];
                                st++;
                                xrs[st]=2*xrs[st];
                          }
                  }
          }
}/* end of subtract function*/
```

Appendix C

JPEG Compression Algorithm For Colour Images Programs Listings

```
/******************************************************************************
*******************************************************************************
*** The following program is the implementation of the first part   ***************
*** of the of JPEG baseline coder for colour images size of 256x256   ************
*** and 24 bits/pixel in R-G-B binary format.  In this program the     **************
*** R-G-B channels are extracted and one luminance and two            ****************
*** chrominance channels are generated using the set of equations  ****************
*** (4.8). Each channel is compressed separately using the programs  ***************
*** listing in Appendix B.                                         ********************
*******************************************************************************
******************************************************************************/
#include<stdio.h>
#include<math.h>
#include<float.h>
#include<errno.h>
#include<stdlib.h>
#include<malloc.h>
#define ROWS  8

short image_width,n,m,blocks,scalor,component,d1,d2;
unsigned char kk;

static unsigned char yr[195];
static unsigned char yr_R[65];
static unsigned char yr_G[65];
static unsigned char yr_B[65];

float *yr_Y,*yr_I,*yr_Q;
FILE *in_file,*fp_Y,*fp_I,*fp_Q;

void open_file();
void extract_R_G_B(); /* function to get R-G-B from image*/
void generate_Y_Cb_Cr();/* function to generate luminance
                              and chrominance channels*/
void DCT();             /* External function to convert each channel
                             into DCT domain and apply quantization on DCT coeff.*/
void CODE_LIST(): /* External function that contains the Huffman code tables for
                             luminance and chrominance channels */
void HUFF();            /* External function to encode each channel to variable length
                             codes*/
main()
{
        short i,j,k;
        long loc,loco;
```

```
/* open files function */

   open_file();

/******************
allocating memory
*******************/
yr_Y=(float *)malloc((n+1)*(n+1)*sizeof(float));
yr_I=(float *)malloc((n+1)*(n+1)*sizeof(float));
yr_Q=(float *)malloc((n+1)*(n+1)*sizeof(float));

/*******************************
Dividing the image into blocks
*********************************/
for(d1=0;d1 < blocks ;d1++)
  {
  for(d2=0;d2 < 3*image_width ;d2=d2+3*n)
        {
                loc=(long)d1*(long)(3*image_width)*(long)n+(long)(d2);
                for(i=0;i < n;i++)
                  {
                        fseek(in_file,loc,SEEK_SET);
                        fread(yr+i*3*n,3*n*sizeof(unsigned char),1,in_file);
                        loc=loc+(long)(3*image_width);
                  }

/**************************
  function to extract R-G-B
***************************

extract_R_G_B();

/************************************************
function to generate the luminance and chrominance
*************************************************/

generate_Y_Cb_Cr();

   }/*end of rowwise blocking*/
 }/*end of columnwise blocking*/
```

```
        /***************
        closing files
        ***************/
        fclose(in_file):
        fclose(fp_Y):
        fclose(fp_I):
        fclose(fp_Q):

        /****************************************
        Compressing Y. Cb. Cr separately
        ****************************************/

        printf("enter the scaling factor \n"):
        scanf("%d".&scalor):

        for(component=1:component <= 3:component++)
          {

                DCT():

                CODE_LIST():

                HUFF():

          }

        return(-1):
}/*end of main program*/
/********************************
********************************/
        /*****************************************
          function to open input and output files
        *****************************************/
        void open_file()
         {

        int i:
        double nsq;
        char file_name[15]:

        printf(" Enter the input file name ->"):
        scanf("%s".file_name):
```

```
if( (in_file = fopen(file_name ,"rb"))= NULL)
  {
        printf(" Can't open input file \n");
        exit(1);
  }

n=8;
image_width=256;
blocks=image_width/n;

m=(int)(log10((double)n)/log10((double)2.0)+0.5);

if (( fp_Y=fopen("Y.img","w")) == NULL)
  {
        printf(" Can't open output file \n");
        exit(1);
  }

if (( fp_I=fopen("Cb.img","w")) == NULL)
  {
        printf(" Can't open output file \n");
        exit(1);
  }

if (( fp_Q=fopen("Cr.img","w")) == NULL)
  {
        printf(" Can't open output file \n");
        exit(1);
  }

}/* end of open_file function*/
/****************************
Function for extracting R_G_B
*********************************/
void extract_R_G_B()
 {
int i,k,l;

   k=0;
   for(i=0;i < n*n ;i++)
     {

        yr_R[i]=(unsigned char)(yr[k]);
        k++;
```

(continued)

```
        yr_G[i]=(unsigned char)(yr[k]);
        k++;

        yr_B[i]=(unsigned char)(yr[k]);
        k++;

     }/* i-loop*/

}/*R_G_B function*/

/*****************************
Function for generating Y_Cb_Cr
*******************************/
void generate_Y_Cb_Cr()
  {
  int i;

  for(i=0;i < n*n;i++)
    {

        yr_Y[i]=0.299*(float)yr_R[i]+0.587*(float)yr_G[i]+0.114*(float)yr_B[i];

        fprintf(fp_Y,"%f\n",yr_Y[i]);

        yr_I[i]=-0.168*(float)yr_R[i]-0.331*(float)yr_G[i]+0.5*(float)yr_B[i]+128;

        fprintf(fp_I,"%f\n",yr_I[i]);

        yr_Q[i]=0.5*(float)yr_R[i]-0.4186*(float)yr_G[i]-0.0813*(float)yr_B[i]+128;

        fprintf(fp_Q,"%f\n",yr_Q[i]);

    }

}/*end of function Y_Cb_Cr*/
```

```
/*****************************************************************************
******************************************************************************
*** The following program is the implementation of the last part    ************* **
*** of the JPEG baseline decoder for colour images size of 256x256    ********** **
*** and 24 bits/pixel in R-G-B binary format.  In this program the    **************
*** reconstructed Y-Cb-Cr are used to generate R-G-B channels         *************
*** using the set of equations (4.9). The programs in Appendix B      ***********  **
*** that decodes the monochrome image are used prior to this program  ********* **
*** to decode Y-Cb-Cr.                                               ****************
******************************************************************************
**************************************************************************** **/

/********************************
 function for generating R_G_B
*********************************/
void generate_R_G_B()
{
 double buffer_R,buffer_G,buffer_B;
 int i;

 for(i=0;i < n*n;i++)
     {
       /* ensuring that rounding errors will not cause
       the values to exceed the range 0-255 */

       buffer_R=yr_Y[i]+(double)1.402*(yr_Q[i]-128.0);

       if( buffer_R > 255.0)
          yr_R[i]=255;
       else if( buffer_R < 0.0)
          yr_R[i]=0;
       else if( buffer_R <= 255  && buffer_R >=0)
          yr_R[i]=(unsigned char) buffer_R;


       buffer_G=yr_Y[i]-(double)0.344*(yr_I[i]-128.0)-(double)0.714*(yr_Q[i]-128.0);

       if( buffer_G > 255.0)
          yr_G[i]=255;
       else if( buffer_G < 0.0)
          yr_G[i]=0;
       else if( buffer_G <= 255  && buffer_G >=0)
          yr_G[i]=(unsigned char) buffer_G;
```

```
        buffer_B=yr_Y[i]+(double)1.772*(yr_I[i]-128.0);

        if( buffer_B > 255.0)
            yr_B[i]=255;
        else if( buffer_B < 0.0)
            yr_B[i]=0;
        else if( buffer_B <= 255  && buffer_B >=0)
            yr_B[i]=(unsigned char) buffer_B;
      }
} /***end of function generate_R_G_B******/
/******************************
   function for mixing R_G_B
******************************/
void mix_R_G_B()
{
  unsigned int color_buffer,i,k;

  k=0;
  for(i=0;i < n*n; i++)
  {
    yro[k]=(unsigned char)yr_R[i];
    k++;
    yro[k]=(unsigned char)yr_G[i];
    k++;
    yro[k]=(unsigned char)yr_B[i];
    k++;
  }
}/* end of function mix_R_G_B*/
/**************************************************
function for writting the transformed data to a file
**************************************************/
void write_to_file()
 {
  unsigned long int loco;
  int i;

    loco=(long)d1*(long)(3*image_width)*(long)n+(long)(d2);
    for(i=0;i < n;i++)
           {
           fseek(outfile,loco,SEEK_SET);
           fwrite(yro+i*3*n,3*n*sizeof(unsigned char),1,outfile);
           loco=loco+(long)(3*image_width);
           }
 }/* end of write_to_file function*/
```

Vita Auctoris

Napiluon Shlimon was born in Mosul. Iraq on May 12. 1962. He received his B.A.Sc. degree in 1985 in Electrical Engineering from the University of Baghdad. Baghdad. Iraq. Currently he is a candidate for the M.A.Sc. degree in Electrical Engineering at the University of Windsor and is expected to graduate in June 1993. His research interest is in the area of digital image processing and software developments.