1

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

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.

MOTION COMPENSATED VIDEO COMPRESSION USING BLOCK TRUNCATION CODING AS A BASIS

by

Kulvir Kumar Kapoor, Ph.D.

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Systems and Computer Engineering

Carleton University, Ottawa, Ontario

November 9, 1990

©Kulvir Kumar Kapoor, 1990

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

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.

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 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.

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 0-315-68926-9

Canada

Carleton University
Ottawa, Canada K1S 5B6

Thesis contains black & white
photographs &/or explanatory tables
which when microfilmed may lose their
significance. The hardcopy of the
thesis is available upon request from
Carleton University Library.

Acceptance

The undersigned recommend to the Faculty of Graduate Studies and Research acceptance of the thesis, "*Motion Compensated Video Compression using Block Truncation Coding as a basis*", submitted by Kulvir Kumar Kapoor, Ph.D., in partial fulfillment of the requirements for the degree of Master of Science.

Thesis Supervisor

Chairman, Department of Systems
and Computer Engineering,
Carleton University

December 21, 1990

Abstract

The work described in this thesis pays special attention to the aspect of motion estimation and prediction in compression of a sequence of images. Efforts have been made to develop an efficient hybrid coding algorithm using Block Truncation Coding (BTC) as a basis for minimizing computational complexity in the motion search algorithm. The BTC coding scheme as outlined by Healy and Mitchell [1] together with the Adaptive Discrete Cosine Transform (ADCT) Coding algorithm as used by Wu [2] and the recent introduction of hardware implementation of motion estimation algorithms in the image coding environment, have motivated the author to re-explore BTC and utilize it to develop a motion estimation and compensation algorithm. The bit-maps created by BTC are used for the fast and simple prediction of motion vectors in the motion search algorithm and ADCT provides an added component of compression.

The first part of the thesis gives an account of various digital image coding techniques that have evolved over the past many years, and many of which are still in use and are being modified or updated from time to time. Special attention has been devoted to Block Truncation Coding and motion-compensated predictive coding algorithms.

Motion Compensation based on BTC is described in detail in the next two chapters. The scheme combines the simplicity of computation and edge preservation aspects of BTC and the high fidelity and high compression of ADCT with simple motion detection and prediction through exclusive-or operations on the bit maps of the current and previous image frames in the sequence. The overall complexity of the coding scheme developed is much less than that of most existing algorithms, while the fidelity performance is competitive. Simulation studies clearly demonstrate the good quality of reproduction of the predicted image. The ease of computation, good quality of reconstructed image, and sufficiently high level of compression achieved, make the entire motion compensation scheme described in the thesis attractive.

Acknowledgements

I take this opportunity to express my feelings of gratitude and indebtedness to my thesis supervisor, Professor David C. Coll for his guidance and supervision during the entire duration of this research. Thanks are due to Dr. S.A. Mahmoud, Chairman, Department of Systems and Computer Engineering, and Dr. A.U.H. Sheikh, Associate Chairman for Graduate Studies, for their continued interest and support in this work.

I would also like to thank Dr. Yiyang Wu for permitting me to use the DCT component of his BTC-VQ-DCT coding scheme in his thesis to achieve additional image compression. I am also grateful to the faculty and all the staff members in the department, who imparted useful technical courses which acted as a foundation towards writing this thesis in the field of computer engineering which is so different from Physics in which I had spent my previous academic career. Thanks are due to Naren Mehta for his timely assistance during problems faced with Publisher word processing software on SUN/UNIX environment, and to Vivienne Gilchrist for her support pertaining to administrative matters.

I would like to record my sincere thanks to Dr. R.L. Clarke and Dr. M.K. Sundaresan who encouraged me to undertake advanced studies in the field of evolving systems and computer engineering.

Financial assistance in the form of scholarship/research grant/teaching assistantship is gratefully acknowledged.

Finally, I would like to dedicate this thesis to my wife, Sushma, my three children, and my mother.

Table of Contents

Acceptance	ii
Abstract	iii
Acknowledgements	iv
List of Figures	ix
List of Tables	xi
CHAPTER 1 INTRODUCTION	1
1.1 Motivation	1
1.2 Background	2
1.3 Objectives and Methodology	4
1.4 List of Abbreviations and Notations	7
CHAPTER 2 IMAGE DATA COMPRESSION AND MOTION	
COMPENSATION : A REVIEW OF TECHNIQUES	9
2.1 Introduction	9
2.2 Spatial Domain Methods	10
2.2.1 Pulse Code Modulation (PCM)	10
2.2.1.1 Dither Techniques	11
2.2.1.2 Halftoning	11
2.2.1.3 Statistical Coding (Entropy Coding)	11
2.2.2 Predictive Coding	12
2.2.2.1 Delta Modulation (DM)	13
2.2.2.2 Differential Pulse Code Modulation (DPCM)	14
2.2.2.3 Adaptive Techniques	16
2.2.2.4 Constant Area Quantization	17
2.2.2.5 Interpolative Coding	18
2.2.2.6 Tree Encoding	19
2.2.2.7 Run-Length Coding (RLC)	19
2.2.2.8 Laplacian Pyramid Coding	20
2.2.2.9 Block Truncation Coding (BTC)	21

2.3	Transform Coding	25
2.3.1	Theory	26
2.3.2	Recursive Block Coding	28
2.3.3	Two-Source Coding	29
2.3.4	Adaptive Transform Coding	29
2.4	Hybrid Transform/Predictive Coding	31
2.5	Vector Quantization Coding	32
2.6	Interframe Coding and Motion Compensation	34
2.6.1	Frame-Repeating	35
2.6.2	Resolution Exchange	35
2.6.3	Conditional Replenishment	36
2.6.4	Motion Compensation	37
2.6.4.1	Pel Recursive & Block Matching Algorithms (PRA and BMA)	37
2.6.4.2	Block Matching Algorithms (BMA)	38
2.7	Summary	43
CHAPTER 3	MOTION COMPENSATED BTC-DCT HYBRID VIDEO	
	CODING	47
3.1	Introduction	47
3.2	Motivation of BTC as a basis for motion detection and compensation	48
3.3	Description of Motion Compensated BTC-DCT Hybrid Video Coding Algorithm	49
3.3.1	Phase I: Prefiltering	50
3.3.2	Phase II: BTC Coding of Current and Previous Frames.	52
3.3.3	Phase III: Adaptive DCT Encoding of High- and Low-Mean Sub-Images.	54
3.3.4	Phase IV: Motion Compensation Algorithm using Bit-Maps (BTCMCP421)	54
3.4	Comparison of BTCMCP421 Weight Function with other Mismatch Functions	62

CHAPTER 4	SIMULATION, RESULTS AND DISCUSSION	64
4.1	Image Sequences	64
4.2	Hardware Environment	66
4.3	Applications Software	66
4.4	Simulation Tests & Experimental Results	69
4.4.1	First Set: Image Sequence 'SELF1'	69
4.4.2	Second Set: Image Sequence 'TESTPAT'	80
4.4.3	Third Set: Image Sequence 'SELF2'	83
4.4.4	Third Set without pre-filtering: 'SELF2' image sequence	87
4.5	Exhaustive Search on 'SELF2' Image Sequence	92
4.6	Histogram of Motion Vector Data	95
4.7	Image Data Compression Ratio Evaluation	98
4.7.1	Single Frame 2-D BTC, No DCT Coding of High- & Low-mean sub-images, and No Motion Compensation	98
4.7.2	Interframe 2-D BTC, with Motion Compensation, and No DCT Coding of high- and Low-mean sub-images	99
4.7.3	Single Frame 2-D BTC, DCT Coding of High- and Low-mean sub-images, No Motion Compensation	100
4.7.4	Interframe 2-D BTC with Motion Compensation and DCT Coding of High- and Low-mean sub-images	100
4.8	Summary of Simulation Results	102
4.9	Discussion	103
4.9.1	NMSE Performance	103
4.9.2	Psycho-visual criteria for Error Evaluation	103
4.9.3	Image Data Compression	104
4.9.4	Postfiltering	105
4.9.5	Histograms of High-, Low-mean and High-Low difference sub-images	106
4.9.6	Run Length Coding of Bit-maps	106
4.9.7	Worst Case Scenario	107
4.9.8	Compression & High/Low Mean Sub-images	108

CHAPTER 5	CONCLUSIONS	109
References		112
Appendix A	PROGRAM LISTINGS IN 'C'	121
A.1	Source Listing of BTCMCP421.C	121
A.2	BMP44CMP.C/BMP44DEC.C Code	135
A.3	KBTC44CV.C/KBTC44DV.C Code	140
A.4	KBINDIX.C/KABSDIF.C Code	145
A.5	NMSERROR.C Code	147
A.6	WBCOUNTS.C Code	149
A.7	ENRFILE.C Code	150
A.8	HLDA44CV.C/HLDA44DV.C Code	154
A.9	BMP128V2.C Code	160
A.10	DISPLAY3.C/DISPLAY4.C Code	163
A.11	AVERAGE2.C Code	166
A.12	EXH44CON.C Code	167
A.13	KSNAP123.C Code	174
A.14	MCPCOLOR.C Code	175
A.15	MEDFILTR.C Code	176
A.16	BMPRLCOD.C/BMPRLDEC.C Code	181
A.17	ENTPYPIX.C Code	185
A.18	BMP4.BAT/MCP4.BAT Batch Files	187

List of Figures

Figure 3.1:	3×3 convolution process for low pass filtering.	51
Figure 3.2:	4×4 pixel block data before and after BTC.	53
Figure 3.3:	Block Diagram of BTC based Motion Compensated Algorithm.	56
Figure 3.4:	Illustration of First Stage (up to 4 pixels) of Three Stage Search algorithm.	59
Figure 3.5:	The three stage search procedure.	60
Figure 4.1:	'SELF1' image sequence used in simulation of BTC based Motion Compensation algorithm. (a) Prefiltered previous frame. (b) Prefiltered current frame. (c) Bit-map of previous frame. (d) Bit-map of current frame. (e) Exclusive-or difference image created from (c) and (d). (f) Exclusive-or difference image created from predicted bit-map and bit-map of current frame.	71
Figure :	4.1 (Continued.) (g) Absolute difference of gray level current and previous frames. (h) Absolute difference of predicted image and current frame. (i) Bit-map predicted from bit-map of previous frame. (j) Predicted image from previous frame. (k) Direct BTC coding of current frame. (l) Predicted image using Exhaustive search algorithm.	72
Figure 4.2:	Comparison of current image, Pic2 with predicted images obtained from Motion Compensation and direct BTC algorithms in 'SELF1, and SELF2' image sequences.. . . .	73
Figure 4.3:	Illustration of Display utility algorithms to display multiple images simultaneously. (a) High-mean, Low-mean and high-low difference 128×128 images displayed simultaneously. (b) High-mean, Low-mean, high-low difference 128×128 images along with 512×512 Bit-map of previous frame interpolated to size 128×128. (c) Histograms of gray levels of High-mean and high-low difference images.	78

Figure 4.4:	(a) Another illustration of 'Display'utility algorithm. High-mean, Low-mean, high-low difference and interpolated bit-map of previous frame shown superimposed on 512×512 bit-map of previous frame. (b) Histogram of gray levels of previous frame.	79
Figure 4.5:	'TESTPAT' image sequence simulation. (a) Prefiltered previous frame. (b) Prefiltered current frame. (c) Bit-map (a). (d) Bit-map of (b). (e) XOR difference image created from (c) and (d). (f) XOR difference image created from predicted binary image and bit-map of current frame.	82
Figure :	4.5 (Continued) (g) Absolute difference of gray level current and previous frames. (h) Absolute difference of predicted image and current frame. (i) Direct BTC coding of previous frame. (j) Predicted image.	83
Figure 4.7:	'SELF2' image sequence used in simulation of BTC based Motion Compensation algorithm. (a) Prefiltered previous frame. (b) Prefiltered current frame. (c) Bit-map of previous frame. (d) Bit-map of current frame. (e) Exclusive-or difference image created from (c) and (d). (f) Exclusive-or difference image created from predicted binary image and bit-map of current frame.	88
Figure :	4.6 (Continued) (g) Absolute difference of gray level current and previous frames. (h) Absolute difference of predicted image and current frame. (i) Bit-map predicted from bit-map of previous frame. (j) Predicted image from previous frame.	89
Figure 4.7:	'SELF2' image sequence without prefiltering. (a) Bit-map of previous frame. (b) Bit-map of current frame. (c) XOR difference image created from (a) and (b). (d) Predicted image.	90
Figure 4.8:	Vector Histogram for 'SELF1' image sequence.	96
Figure 4.9:	Vector Histogram for 'SELF2' image sequence.	96
Figure 4.10:	Vector Histogram for 'TESTPAT' image sequence.	97
Figure 4.11:	Vector Histogram for 'SELF2' image sequence without prefiltering.	97

Figure 4.12: Entropy (H in bits/pixel) measurements on 128X128 gray-level sub-images and their difference sub-images. The results are compared with entropy of original 512X512 gray-level images of 'SELF1' image sequence. 106a

List of Tables

Table 2.1	Bandwidth Reduction Ratios for various Image Coding Techniques.	45
Table	2.1 (Continued.)	46
Table 4.1	NMSE performance of simulation tests on 'SELF1' image sequence.	76
Table 4.2	White pixels count in gray level, and bit-map image differences in 'SELF1' simulation.	77
Table 4.3	Run Length Statistics of Motion Vector file in 'SELF1' simulation.	80
Table 4.4	NMSE performance of simulation tests on 'TESTPAT' image sequence.	84
Table 4.5	White pixels count in gray level, and bit-map image differences in 'TESTPAT' simulation.	84
Table 4.6	Run Length Statistics of Motion Vector data in 'TESTPAT' simulation.	85
Table 4.7	NMSE performance of simulation test on 'SELF2' image sequence.	86
Table 4.8	White pixels count in gray level, and bit-map image differences in 'SELF2' simulation.	86
Table 4.9	Run length statistics of Motion Vector data in 'SELF2' simulation.	87
Table 4.10	NMSE performance of simulation test on 'SELF2' image sequence without pre-filtering.	91
Table 4.11	White pixels count in gray level, and bit-maps image differences in 'SELF2' simulation without pre-filtering.	91
Table 4.12	Run length statistics of Motion Vector data in 'SELF2' simulation without pre-filtering.	92
Table 4.13	NMSE performance of simulation test on 'SELF1' image sequence using Conventional Exhaustive Search.	93
Table 4.14	White pixels count in gray level image differences in 'SELF1' simulation using Conventional Exhaustive Search.	94
Table 4.15	Summary of simulation tests on different image sequences.	102

Chapter 1

INTRODUCTION

1.1 Motivation

Research on bandwidth compression for transmitting the video component in different applications such as television, teleconferencing and computed tomography in medical imaging etc., has been a challenge to scientists and engineers because of the problems posed. It has been demonstrated beyond all doubts that digital image coding using motion compensated prediction results in better reconstructed picture quality and more compression. With rapid advances in VLSI technology, real-time motion estimation processor chips have started entering the market for hardware implementation of image coding with motion compensation. To make processing on hardware chips still faster, continued research is needed in software algorithm developments. This thesis is a step towards achieving this goal with the least amount of computational complexity, yet with reasonably acceptable picture quality of the reconstructed image.

1.2 Background

A large number of digital image transmission and storage applications such as teleconferencing, videotelephony, television and satellite image transmission, medical imaging for computer assisted tomography (e.g. X-ray Computed Tomography and Magnetic Resonance Imaging) and angiocardiology, contain images of moving objects. The motion captured in such a multiple sequence of images includes translation and rotation of objects with respect to the video camera. In many of such applications, one is confronted with data sets containing very large numbers of pixels. Existing data communication networks simply do not have enough bandwidth to carry image data traffic economically. For instance, today's state-of-the-art digital network — the Integrated Services Digital Network (ISDN) — which integrates voice and data services into one digital network, has a basic channel rate of 64 kbits/sec. To transmit an image of size 1024 by 1024 pixels with 8 bits/pixel gray level values over a basic ISDN channel would require about 130 seconds [2], which is too long indeed for an on-line user. Thus, image data compression systems to increase the network or system throughput, as well as to save storage space, are an inescapable necessity.

A fundamental goal of data compression is to reduce the bit rate for transmission or data storage while maintaining an acceptable fidelity or image quality. Numerous bandwidth compression techniques have been developed, such as differential pulse code modulation (DPCM), transform coding, hybrid coding, and adaptive versions of these techniques, in response to the growth of image-processing methods [3–18]. These techniques usually exploit the characteristics of the human psycho-visual system as well as statistical redundancies in the image data to reduce the bit rate.

Digital image-data compression methods can be classified into two basically different categories; namely, *predictive coding* and *transform coding*. Another classification is

intraframe, exploiting spatial redundancy, and *interframe*, exploiting both spatial and temporal redundancies. *Interframe* methods achieve higher compression than *intraframe* techniques, but *interframe* methods require storage of one or more frames. It is not yet clearly established which combination of methods best stands the test of motion and scene change. A technique called *conditional replenishment (CR)* uses the statistical fact that information is present in a scene only when motion occurs and only in selected areas of the frame [14].

Many investigators have shown that motion compensation can improve the data compression ratio by a factor of 1.5 to 2.0 compared with simple interframe coding without motion compensation [19-36]. Instead of simply transmitting interframe differences, motion is estimated for a pixel or a block of pixels, and differences are taken with respect to the displaced location. The entropy of the displaced frame difference is much below that of frame differencing, indicating the higher quality of compression achieved by compensation for motion.

Although motion compensation technology is still in an early stage of development and efforts must be made to bring it into actual application, the effect of motion compensation has been confirmed through investigations and adaptation in some actual systems. The expected performance gain can be utilized in many ways and will certainly expand the application field. With motion compensation, picture quality at 1.5 Mb/s, for instance, can be significantly improved so that better picture quality can be obtained for a wider range of pictures. In situations such as teleconferencing scenes, picture resolution loss by subsampling or field repeating is avoided if motion compensation is incorporated in some way. Jerkiness caused by a large amount of motion decreases; even camera panning and zooming can be handled without jerkiness. If one accepts the present 1.5 Mb/s picture quality, the bit rate can be reduced to even less than 1 Mb/s with motion compensation techniques.

1.3 Objectives and Methodology

Presently, computational complexity is one major bottle-neck in the implementation of image data compression using motion compensation. The technique presented in this work is an attempt to reduce this bottleneck and at the same time maintain an acceptable compromise between compression ratio, fidelity and computational complexity. The preliminary work on interframe Block Truncation Coding (BTC) scheme applied to moving imagery described by Healy and Mitchell [1] in 1981, and the recent introduction of hardware implementation of motion estimation processors on a VLSI chip, acted as a catalyst to achieve this goal through the BTC scheme. The algorithm discussed in this thesis has the advantages of simple computation and edge preservation of BTC together with an efficient motion compensated interframe coding algorithm involving fast computations on bit-maps for detecting optimum motion trajectories.

The search algorithm phase of this technique utilizes a sequential, nonoverlapping, block-by-block, three-stage displacement search using block size of 4×4 . At the end of the third stage of displacement search, the motion vector (vertical and horizontal position co-ordinates) for each motion block is obtained and the information about all motion vectors in the image is run-length encoded (RLC) for transmission in compressed form.

The simplicity of computation in the present algorithm arises from the initial step of BTC which truncates the input images in each two-frame sequence (current and previous frames) into two bit-maps (one for each frame) and four reduced-sized gray level images (two for each frame) which are composed of "high average" and "low average" values from each block. Only the bit-maps are used in the subsequent motion compensation search algorithm. The exclusive-or bit operation on these bit-maps yields a binary difference bit-map. The motion areas may be depicted as areas of bright pixels (i.e., bit value 1) while static areas remain dark (i.e. bit value 0). The motion search

algorithm becomes effective only on regions of bright pixels. Thus the entire image frame is segmented into active or inactive blocks and motion search applied to active blocks only. The active block in the current frame is compared with eight blocks in the previous frame in the first stage search in an area of size 12×12 (i.e., 4 pixels away in eight different positions in the previous frame located symmetrically with respect to the motion block position in the current frame). The comparison is made by the simplest conceivable process of exclusive-oring (XOR) the pixels in the current block with those in the search blocks in the previous frame. The number of pixels that differ in any block to block comparison gives the *distance* function (also referred to as distortion or weight function) for that comparison. The whole purpose of the three stage searches is to minimize this *distance* function. The block of the previous frame yielding the smallest value for this function during exclusive-ored comparison process is the best match. If the comparison gives zero value, the search ends; otherwise the search is extended to a second stage by 2 pixels thus covering a search area of size 16×16 . If the *distance* function does not attain zero value during this stage, the third stage search comes into play extending the search to 1 further pixel thus covering a search area of size 18×18 . Thus in at the most 24 searches, minimum mismatch (i.e. minimum *distance* function value) is established and one obtains the motion vector components for the current block with respect to the previous frame block.

The above algorithm is obviously attractive in terms of ease of computational complexity as it involves simply exclusive-or bit operations followed by a few add operations.

Simulation tests carried out on test patterns and image sequences have demonstrated that the search errors (prediction errors) associated with motion vector estimation are so small and isolated that the reconstructed image (current image frame reproduced from previous frame) is of good quality in the psycho-visual subjective evaluation sense. The transmission of prediction error data could be done but may be ignored so as to avoid

the additional overhead in the process of image compression.

During the BTC phase, high- and low- average gray level values were created for each block. These may be compressed using adaptive Discrete Cosine Transform coding (DCT) as described in the recently submitted Ph.D. dissertation of Wu[2]

The entire scheme of motion compensated image compression , involving BTC, Run Length Coding and adaptive DCT, is thus an hybrid coding technique involving interframe spatial domain coding as well as transform coding. The adaptive DCT gives an added component of bit-rate reduction to the motion compensation scheme using BTC as its basis. The coding scheme demonstrates high fidelity, with no perceptible degradation, and very low computational complexity. A compression ratio of about 20:1 has been attained in transmission of current frame in an image sequence, which is twice that achieved by the BTC-VQ-DCT hybrid scheme described by Wu [2] for still pictures. However, taking into consideration the previous frame in the sequence, the overall compression ratio is about 12:1 which is still three times that obtained from direct BTC.

The most salient feature of original contribution made in this research can be summed up as follows: it provides a means of simplifying computations in the search algorithm by making use of bit operations on bit-maps generated from the BTC encoding.

1.4 List of Abbreviations and Notations

A brief description of abbreviations and notations used in this thesis is given in the following list.

SELF1, TESTPAT, SELF2	Three image sequences used in experimentation
SELF1A	Previous frame of image sequence SELF1
SELF1B	Current frame of image sequence SELF1
TESTPATA	Previous frame of image sequence TESTPAT
TESTPATB	Current frame of image sequence TESTPAT
SELF2A	Previous frame of image sequence SELF2
SELF2B	Current frame of image sequence SELF2
hi, high-	High-mean 128X128 subimage
lo, low-	Low-mean 128X128 subimage
histo	Distribution of gray levels in gray level image
bpin	512X512 bit-map interpolated to 128X128
bin1/bmp1	Bit-map of previous frame, pic1
Hi1,Hi2	High-mean subimages of prev. & curr. frames
Lo1,Lo2	Low-mean subimages of prev. & curr. frames
HVS	Human Visual System
BTCMCP	BTC based Motion Compensation Prediction
BTCMCP421	BTC based three stage motion estimation and compensation algorithm in 4-2-1 pixels steps

Continued

Pic1, pic1	Previous frame of image sequence
Pic2, pic2	Current frame of image sequence
Bin1, bin1	Bit-map of Pic1
Bin2, bin2	Bit-map of Pic2
Bindif, bindif	Exclusive-or difference (XOR) of bin1 and bin2
predB, prebin	Motion compensated bit-map
predR, prepic	Motion compensated gray level image
btc1, btc2	Gray level images of pic1 & pic2 by direct BTC
PredEx	Gray level predicted image by exhaustive search
errorB	XOR difference between predB and bin2
errR1	Absolute difference between pic1 and pic2
errR2	Absolute difference between predR and pic2
absdif	Absolute difference between pic1 and pic2
predpic2	Predicted pic2 image from pic1
MSE	Mean square error
NMSE	Normalized mean square error
TSE	Total square error
MAD	Minimum absolute difference
MAE	Minimum absolute error
H	Entropy

Chapter 2

IMAGE DATA COMPRESSION AND MOTION COMPENSATION : A REVIEW OF TECHNIQUES

2.1 Introduction

Images contain huge amounts of information and require very large bandwidths for transmission. This translates into costly communication channels for transmission and extensive hardware requirements for storage and manipulation of images. The goal of data compression of images is to reduce transmission and storage costs while preserving good quality. Pixels are highly correlated and thus, by removing this redundancy, bandwidth reductions of large ratios can be achieved. Predictive techniques and orthogonal transformations take advantage of the high correlation between neighboring pixels. For moving images, data are also highly correlated from frame to frame, and several techniques can be used that avoid full-resolution transmission of highly redundant successive images.

Digital image coding methods applicable to still and moving images and to the encoding of graphics, may be classified into three general categories: Spatial Domain Methods, Transform Coding and Hybrid Coding.

Coding in the spatial domain involves the direct manipulation of the sampled image data to remove inherent redundancies. Transform Coding involves an energy preserving mapping of the original image data such that maximum information is preserved in a

minimum number of samples or coefficients. Hybrid Coding employs a combination of both techniques, usually transform coding followed by predictive coding of coefficients. Vector Quantization is a technique for data compression, which, based on Shannon's rate distortion theory [40], directly removes redundancies between groups of pixels, i.e., vectors. Vector Quantization can be implemented in either spatial or transform domains.

2.2 Spatial Domain Methods

In this section we consider digital coding techniques that operate on the data in the spatial domain rather than the transform, or generalized frequency domain. We further restrict this discussion to within-frame, or *intraframe*, schemes and defer between-frame, or *interframe*, schemes until a later section. Methods of pulse code modulation, dither, half-toning, statistical coding, delta modulation, differential pulse code modulation, Laplacian pyramid coding, block truncation coding and vector quantization are described.

2.2.1 Pulse Code Modulation (PCM)

The Pulse Code Modulation (PCM) method operates in the spatial domain, is lossy, and is a pixel-by-pixel technique. In a PCM scheme, the incoming continuous video signal is sampled, usually at the Nyquist rate, and then quantized [10,16]. Hence, it is nothing more than a digital representation of the original analog signal.

Although the eye's sensitivity to quantizing distortion decreases with increasing luminance level [17], the high gamma of the display tube complements this logarithmic response of the eye and greatly reduces the effect. Consequently, in analog-to-digital conversion of TV images, a uniform quantizer is normally used [16].

The quantizer usually has N levels, where N is a power of 2 ($N = 2^B$, say), where each sample is represented by a fixed-length binary code word having B bits. The number

of bits needed to code a pixel using PCM depends on the type of image. Commonly, 8 bits are sufficient for monochrome broadcast or video-conferencing images, whereas medical images may require 10 to 12 bits to obtain sufficient amplitude resolution.

2.2.1.1 Dither Techniques

One way to reduce image data is to limit the number of bits per pixel. Unfortunately, however, for typical TV images, as the number of bits is reduced below five or six, a contouring effect becomes visible in the low detail or flat regions [16]. In a method called pseudorandom noise quantization, these contours are broken up by adding a small amount of broadband, pseudorandom, uniformly distributed noise, called dither, to the luminance samples prior to quantization and subtracting it at the receiver. The dither causes the pixels to oscillate about the original quantization levels, which removes the highly visible contours. The proper amount of added dither will reduce contouring while maintaining resolution; usually it will be equal to the least significant bit of the quantizer. With dither techniques, images subjectively equivalent to 6- or 7-bit PCM can be represented by 3 or 4 bits/pixel [41].

2.2.1.2 Halftoning

Dither techniques are also used to generate halftone images from gray level images [42]. Halftone images are binary images that give a gray-scale rendition and are widely used in the printing industry. To each pixel of the original image, often over-sampled, a dither sample is added, and the resulting image is quantized by a one-bit quantizer to produce an array of black and white dots. Compared with a one-bit quantized image, the one-bit halftone has a better visual appearance. The gray level rendition in half-tones is due to local spatial averaging performed by the eye.

2.2.1.3 Statistical Coding (Entropy Coding)

For signals having nonuniform probability density functions or histograms, it is possible to use statistical coding, i.e., a variable-length code such as the Huffman code,

which assigns shorter code words to those levels that are more likely to occur [43]. The long-term histogram for TV images is approximately uniform, although the short-term statistics are highly non-stationary. Consequently, statistical coding is not used in PCM coding schemes for images.

From Shannon's first, or lossless coding, theorem [40], the average bit rate R for encoding a source can be as small as, but no less than, the source entropy H . Huffman [43] introduced a practical procedure to implement the Shannon coding theorem for encoding a statistically independent source. Huffman coding results in a variable length code in which the number of bits for a source sample, i.e., pixel, varies inversely with the corresponding source probability.

The design of a Huffman code depends upon the source statistics. If source statistics change, a new code is required to ensure the minimum bit rate. In adaptive Huffman coding, the code set is modified based upon frequency information derived from the most recent samples, e.g., Lempel-Ziv-Welch algorithm [11,12,44].

2.2.2 Predictive Coding

In PCM, successive inputs to the quantizer are treated independently, so there is no exploitation of the significant redundancy present in images. However, once the signal has been digitized, it is relatively straightforward to exploit this redundancy in the spatial domain by use of predictive coding.

The philosophy underlying predictive coding is to remove the dependency between successive samples and to quantize only the new information, or *innovations*. For any pixel, the innovation is the difference between the value of that pixel and its optimum predicted value, computed from previously coded pixels. For images whose intensity distribution is non-Gaussian, the optimum mean-square (MSE) predictor is non-linear, although linear predictors are often used because their design and implementation are easier [16,45].

The predictive quantization technique is classified according to the number of quantization levels used. Delta Modulation (DM), the simplest implementation, uses only two quantization levels; and Differential Pulse Code modulation (DPCM), the more general case, uses more than two levels. In Adaptive Pulse Code Modulation the coder is modified to match the local statistics of the image.

2.2.2.1 Delta Modulation (DM)

In Delta Modulation (DM), the predictor is a one-step delay function, and a one-bit quantizer is used to achieve a one-bit representation of the signal [46]. An important aspect of this scheme is that it does not require sampling of the input signal. The predictor simply performs integration of the quantizer output signal, which is a sequence of binary pulses. The reconstruction filter of the delta modulator, which is a simple integrator, is unstable. In the presence of transmission errors, the receiver output can accumulate large levels of error. The prediction filter can be stabilized by attenuating the predicted value by a constant $\gamma < 1$ (called the leak factor). The primary limitations of delta modulation are slope overload, granularity noise, and instability in the presence of transmission errors. Slope overload occurs when there is a large jump or discontinuity in the signal, to which the quantizer can respond only in several delta steps. Granularity noise is the steplike nature of the output, which cannot be constant, when the input signal is almost constant. Both of these errors can be compensated to a certain extent by low-pass filtering the input and output signals. Slope overload can also be reduced by increasing the sampling rate, but that will reduce the achievable compression. An alternative for reducing granularity while retaining simplicity is to go to a tri-state delta modulation scheme, in which the quantizer has three levels. Most of the pixels are found to be in the "level" or "0" state; thus, run-length coding and statistical coding are appropriate and yield rates around 1 bit/pixel [47].

Other variations of delta modulation, such as adaptive DM with variable step size, can further improve its performance [48]. However, the increase in complexity has to be measured against the more general case of DPCM coding and other techniques for

data compression. In fact, a large number of the ills of DM can be overcome by DPCM, thereby making it often more attractive than DM for data compression.

2.2.2.2 Differential Pulse Code Modulation (DPCM)

Differential Pulse Code Modulation (DPCM), a predictive coding scheme, is based on the fact that the gray level values of adjacent pixels in most images are highly correlated. That is, if a pixel at (x_{i-1}) is a certain gray level, then the adjacent pixel at (x_i) is likely to have a similar value. Whereas the gray level may range over 256 values (8 bits/pixel), most differences between adjacent pixel values in natural scene images lie within a range of about 20 gray level values [9]. DPCM makes use of this property in the following manner. We observe a pixel x_{i-1} and based on this observed value we predict the value of the next pixel x_i . Let $\hat{x}_i$ be the predicted value of x_i , and let us subtract this value from the actual value x_i to obtain the difference $d_i = x_i - \hat{x}_i$. Assuming that the estimates are reasonably accurate, the difference $x_i - \hat{x}_i$ will, on the average, be significantly smaller in magnitude than the magnitude of the pixel x_i . Consequently, fewer quantization bins and fewer bits are required to code the sequence of differences than would be required to code the sequence of picture elements.

The problem is to estimate x_i given that we know x_{i-1} . The linear estimator that results in the least mean square estimation error,

$$E\{(x_i - \hat{x}_i)^2\}, \quad (1)$$

is given by

$$\hat{x}_i = \rho x_{i-1} + (1 - \rho)m \quad (2)$$

where m is the mean (average) gray level and ρ is the normalized correlation between the adjacent picture elements, that is,

$$\rho = \frac{E\{x_i x_{i-1}\}}{E\{x_i^2\}} \quad (3)$$

The estimate $\hat{x}_i$ in Eq. (2) can be interpreted as a weighted average of the preceding pixel x_{i-1} and the mean of x_i . The weights depend on the correlation coefficient ρ . When the pixel values are highly correlated ρ approaches 1 and $(1-\rho)$ approaches 0, in which case the estimate is based primarily on the value of x_{i-1} . When the pixel values are not very correlated the opposite condition holds and the estimate is based primarily on the mean value. For properly sampled images, ρ lies typically between 0.85 and 0.95.

It is easy to show that the variance of the difference

$$d_i = x_i - \hat{x}_i \quad (4)$$

is given by

$$\sigma_{d_i}^2 = (1 - \rho^2) \sigma_{x_i}^2 \quad (5)$$

where $\sigma_{x_i}^2$ is the variance of x_i . It can also be shown that the d_i are uncorrelated; i.e., the mapping from the x_i to the d_i produces uncorrelated coefficients.

The remaining problems are to quantize and code the differences d_i . The principal components of a DPCM system are, thus, predictor and quantizer. In a typical system, the predictor is placed in the feedback loop around the quantizer so that the quantizer noise at a given step is fed back to the quantizer input at the next step. This has a stabilizing effect that prevents accumulation of errors and dc drift in the reconstructed signal [16].

For digital data it is possible to design a distortionless or reversible predictive coder for which the predictor is not inside the feedback loop. This may be termed open-loop DPCM. A pixel is predicted from the past samples, but the predicted value is rounded to an integer. Then the prediction error also takes integer values and can be coded by a Huffman variable-length, or entropy, coder. At the receiver, the Huffman code is decoded, and the integer prediction error is added to the integer prediction (formed just as in the coder) to reproduce the original image exactly [16].

The quantizer is often designed to limit the three types of degradation associated with DPCM. These are granularity, slope overload, and edge busyness. If the inner levels are too coarse, then granularity is visible in flat regions of the image. Slope overload is

caused by high-contrast edges when there is a prediction error that exceeds the range, or extreme levels, of the quantizer and manifests itself as a blurred edge. Edge busyness is caused by less-sharp edges when the prediction error is within the range of the quantizer but when the reproduced image is at different quantization levels on adjacent scan lines.

Quantizers can be designed on a statistical basis, or they can take advantage of psycho-visual phenomena. The optimum quantizer for a given number of levels is the Lloyd-Max quantizer, which minimizes the mean square quantization error [49]. For a one-dimensional first-order predictor for which the prediction error sequence is modeled as a Laplacian density, the Max quantizer is too companded, that is, the inner quantization steps are too small whereas the outer levels are too coarse, resulting in edge busyness. For this reason, there have been a number of attempts, and some success, to incorporate psycho-visual phenomenon and minimize a subjectively weighted mean square error [50]. To date, there is no recognized best way to incorporate such criteria or even to decide which criterion should be used, thus, at present such quantizers tend to be optimized iteratively after subjective analysis of the resulting picture quality.

The improvement in mean square error performance to be gained by two-dimensional processing is dependent on the image statistics and may be fairly small unless the lines are highly correlated. The prediction gain characteristics show early saturation of performance with increasing predictor orders; the best gain is found with third order methods [5]. High order two-dimensional predictors are used not for better signal to noise ratio or mean square error (MSE) [51] but for better subjective performance [52] and for better tolerance to transmission error effects [53].

The advantages of DPCM are high fidelity, computational simplicity, low memory access, and robustness to transmission errors. The major drawback is that the compression ratio is not very high [2].

2.2.2.3 Adaptive Techniques

DPCM coding is quite sensitive to changes in input data statistics [54]. Due to the variation, or nonstationarity, of the image statistics, both within and between frames, it

is impossible to design a single predictor or quantizer that is always optimum. Adaptive techniques use a range of quantizing characteristics and/or predictors from which a current optimum is selected according to a decision rule based on local image properties. Adaptive techniques generally fall into one of the two classes: an adaptive predictor with a fixed quantizer, or an adaptive quantizer with a fixed predictor [5]. In the adaptive approaches, the parameters of the predictors or the quantizers, are continuously modified to keep up with the changing statistics of the input data.

Any adaptive scheme should limit the amount of extra information to be transmitted, because the ultimate aim is data compression. This is achieved by using only previously coded pixels to determine the mode of operation of the adaptive coder. In the absence of transmission errors, the receiver can keep track of decisions made at the transmitter using the received pixels, which are identical to those used at the transmitter. However, in the presence of transmission errors, a poorly designed adaptation rule could prove catastrophic if the receiver were to lose track of the transmitter operations [5,16].

Adaptive predictors provide a much improved subjective image quality, especially at the edges [55]. A popular technique is to use a number of predictors, each of which performs well if the image is highly correlated in a certain direction. The direction of maximum correlation is computed from previously coded pixels, and the corresponding predictor is chosen.

Adaptive quantization schemes segment the image into different regions according to spatial detail, or activity, and different quantizer characteristics are used for each activity class. The flat regions are quantized more finely than edges or detailed areas. This scheme takes advantage of the fact that noise visibility decreases with increased activity. It has been found that the advantages of adaptation increase as the number of activity classes is increased up to four but that there is little improvement thereafter [56].

2.2.2.4 Constant Area Quantization

Constant Area Quantization (CAQ) [57] is a one-dimensional spatial differencing technique that performs well down to approximately 1 bit/pixel for typical applications

in remotely piloted vehicles (RPV) [9]. It exploits the psycho-visual property by which the eye sees more detail in high-contrast regions than in low-contrast ones. It is similar to a tri-state DM technique except that it uses a variable step size, which is determined by the separation of nonzero levels of pulses. The product of the step size can be calculated at the receiver without any overload. Data compression is thus achieved by transmitting lower resolution, fewer pulses, in the low contrast areas.

2.2.2.5 Interpolative Coding

Interpolation is a pixel-by-pixel (with some knowledge of neighboring pixels), lossy, spatial domain method that utilizes image area reduction. Interpolative coding selects a subset of pixels for transmission by some subsampling formula and during reconstruction determines the missing pixels by interpolation on the transmitted one [58,59].

The fixed version of interpolation is similar to subsampling. The quality of the result is a function of the original pixel correlation, the number and type of samples decimated, and the method of interpolation (which may be simple or complex). Most schemes use either a weighted average or higher degree polynomials but linear interpolation is quite effective and little is to be gained by using higher order polynomials [60].

Adaptive interpolation can be done in three different ways. First, the choice of pixels used to compute the interpolated pixel may be varied. The construction of the interpolation formula, used to compute the interpolated pixel, may be altered. Finally, the method may be based on evaluation of the interpolation error to determine what to transmit. In this last case an error threshold may be set and if the actual error is less than the threshold then fewer points will be transmitted. The aim is to keep the fewest points required to maintain the error less than the threshold. The original criterion for error was mean square error [60] but more sophisticated criteria related to the Human Visual System (HVS) have been used [61]. An adaptive two-dimensional interpolation has been used further reducing the number of pixels transmitted [62].

Transmission errors tend to be localized and depend on the number of terms and weighting in the interpolation formula. The performance of interpolation for the number

of bits per pixel for a given image quality is slightly better than non-adaptive DPCM and transform coding but not as good as adaptive DPCM and transform coding [63].

2.2.2.6 Tree Encoding

Tree Encoding [64] is a delayed encoding technique that effectively causes the quantizing level representing a given pixel to be based on future pixels. It has been shown that two-dimensional tree encoding offers potential advantages over two-dimensional DPCM by reducing the distortion introduced by compression and also by increasing error tolerance. In conventional DPCM, each pixel is quantized to minimize the distortion in that pixel, which can result in unnecessarily large distortion for some future pixel. In delayed encoding, a more global measure of distortion is used, and each pixel is quantized to minimize the distortion in a group of pixels.

2.2.2.7 Run-Length Coding (RLC)

Run-Length Coding (RLC) [9] is one of the most extensively studied and simplest methods of one-dimensional compression. Run-Length coding operates pixel-by-pixel in the spatial domain in an effort to reduce image area and the scheme is theoretically information lossless. A run is a succession of pixels having the same gray level. The gray level and the run-length must be transmitted. There are two types of run-length coding, end-on, in which the absolute location of the first pixel in a run is sent as well as the run-length and gray level; and simple run-length, where the next pixel position is deduced from where the previous run ended. The former method has poorer compression owing to the added overhead of pixel position, but can keep any channel errors localized. The second technique is more efficient but is particularly susceptible to transmission noise, and the results of errors can be catastrophic. Error protection is essential.

The codes used to represent the runs are determined by the distribution of the runs. Fewer bits are assigned to shorter runs than to longer ones. Some examples are the CCITT code for document coding [65] and the DIFAX code for weather chart coding [66].

The work presented in this thesis also makes use of this scheme for the transmission of motion vectors data (indicating motion of various regions in the image sequence) to reconstruct the current image from the previous image with respect to which it has moved. Run-length coding of the motion vector data is found to be most efficient in this scheme owing to a relatively large proportion of the image scene being static, i.e., each of the blocks in this region are represented with $(0,0)$ as the components of the motion vector.

2.2.2.8 Laplacian Pyramid Coding

Laplacian Pyramid Coding scheme operates in a fashion that replicates some features of the HVS [63,67]. First, the image is low-pass filtered using a weighted convolving function of a roughly Gaussian profile to produce a so-called "Gaussian" image. This Gaussian image is then subtracted from the original to produce the "Laplacian" image, a difference image with little pixel-to-pixel correlation. This Laplacian image has lower variance and entropy since most pixel gray levels are tightly and symmetrically distributed about the zero level [67]. It is called the Laplacian image since it corresponds very closely to the result of applying a Laplacian operator (∇^2) to the original image. Compression is achieved owing to the lower entropy in the Laplacian image, and by coarsely quantizing it. In order to reconstruct the original image, both the Gaussian and Laplacian images must be added together, which appears to permit no compression. However, the Gaussian image may be represented at a lower sample density since it is a low pass version of the original image. Therefore, in practice, the Gaussian image is subsampled and both the subsampled Gaussian and the Laplacian image must be transmitted. At reconstruction the Gaussian image is expanded again to its original area by interpolation before addition to the Laplacian image. The above steps are then applied repeatedly to each successive low pass image to form a pyramid of Laplacian image planes, each one smaller than the previous. An area reduction of a factor of four occurs at each step up the pyramid. The pyramid production stops when the final Gaussian shrinks to the size of the convolving filter (typically 5×5 pixels). What is transmitted is this smallest Gaussian image plus all

the other Laplacian planes. The Laplacian image, being a difference image, possesses decorrelated pixel properties similar to those of predictive coding methods.

Reconstruction is performed by interpolating smaller images to the next larger one in the pyramid, adding in the Laplacian plane at that level, and repeating these operations until the final image is produced. The Laplacian planes can be viewed as a series of bandpassed images. By judicious choice of quantization levels of the Laplacian planes, the characteristics of the HVS can be used to reduce the bit-rate in those planes; lower planes represent higher spatial frequencies which the HVS cannot perceive, and thus they can be coded with fewer levels. These pixels will contribute little to the final image since they will have a low sample density.

The advantages of Laplacian Pyramid Coding include conformity with the HVS, access to quasi-bandpass images possibly useful for pattern recognition, and easy progressive transmission of images with increasingly better resolution. The disadvantages of this coding algorithm are computational complexity, large memory requirement, and transmission error sensitivity.

2.2.2.9 Block Truncation Coding (BTC)

Block Truncation Coding (BTC) is a coding method that was originally proposed by Delp and Mitchell [68] for coding images in the spatial domain. Its performance and simplicity make this coding method very attractive for real-time image transmission. Healy and Mitchell [1] used this technique in their preliminary investigations of interframe coding of moving imagery.

In BTC, the image is divided into $n \times n$ pixel blocks (typically $n = 4$), each of whose pixels are individually quantized to two levels, Y_0 and Y_1 , such that the block sample mean μ and variance σ^2 are preserved. If $X_1, X_2, \dots, X_N$ are the original sampled values within a given block, then the sample mean is defined as

$$\mu = \frac{1}{N} \sum_{i=1}^N X_i = \bar{X} \quad (6)$$

and the sample variance is defined as

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^N X_i^2 - \mu^2 = \overline{X^2} - \bar{X}^2 \quad (7)$$

It has been shown that the first two sample moments $\bar{X}$ and $\overline{X^2}$ can be preserved using quantizer levels

$$Y_0 = \mu - \sigma \sqrt{q/p} \quad (8)$$

and

$$Y_1 = \mu + \sigma \sqrt{p/q} \quad (9)$$

where q and p are the number of pixels above and below the sample mean, respectively. Pixels below the sample mean are quantized to level Y_0 and the others to Y_1 . Since the first two sample moments are preserved, Eq. (7) shows that the sample variance is also preserved. For each block, μ , σ , and a bit plane (one bit per pixel to select quantizer level Y_0 or Y_1) are normally transmitted. Using 4×4 pixel blocks and eight bits each for μ and σ , the data rate is 2 bits/pixel (including 16 bits for the bit plane).

The Absolute Moment BTC technique described in [69] quantizes the pixels in each block into high-mean and low-mean, which will preserve the Mean Absolute Moment or minimize the Mean Absolute Error in each block. The sample mean defined in Eq.(1) is used as the threshold for the 1-bit quantizer. If

$$X_i \leq \bar{X} \text{ or } X_i > \bar{X}, \quad (10)$$

$\bar{X}_L$ or $\bar{X}_H$, respectively, is assigned to the quantizer levels, where

$$\bar{X}_L = \frac{1}{N-q} \sum_{X_i \leq \bar{X}} X_i \quad (11)$$

and

$$\bar{X}_H = \frac{1}{q} \sum_{X_i > \bar{X}} X_i \quad (12)$$

and q is the number of pixels having value higher than $\bar{X}$. Thus, the outputs of the BTC for each block include two numbers which specify the high-mean, $\bar{X}_H$, and low-mean, $\bar{X}_L$, and a bit-map which specifies the state, $\bar{X}_H$ or $\bar{X}_L$, of each output pixel. This algorithm has the potential of low computational complexity with no multiplications involved. Advantages of BTC include computational simplicity and robust behavior in the presence of channel noise. Additionally, the adaptive one-bit quantizer preserves the dominant intensity variations at the expense of secondary changes which are less perceptible to the Human Visual System (HVS) [63,69,70].

Halverson et al [71] generalized BTC to include a family of moment preserving quantizers with the potential of improving performance. It has been proved that the selection of the quantizer threshold and levels can be generalized for much larger classes of pairs and triples of sample moments, respectively. Performance improvements were demonstrated from the standpoint of peak signal-to-noise ratio. However, the application of the BTC algorithm for the higher moments case is often hampered by overflow problems and increased complexity.

BTC may be adapted to local image statistics by partitioning the 512×512 pixel image into blocks of 32×32 pixels and applying the BTC using a two-level moment preserving quantizer to each block [69,118]. All required parameters such as overall average, high-mean, low-mean and maximum of deviation of pixel values from high-mean and low-mean in the block are computed. If the maximum deviation within that block is higher than the specified threshold, then a recursive call is made to divide the block into sub-blocks of 16×16 pixels and the process is repeated for 8×8 , 4×4 , 2×2 pixel blocks if necessary. If the threshold level still cannot be satisfied, then raw data are saved.

A major advantage of any BTC scheme is that the coding algorithm is very simple and decoding is trivial. The one-bit quantizer tends to enhance edges by preserving dominant intensity variations although at the expense of minor secondary changes which are much less perceptible to the human visual system. The BTC technique effectively preserves abrupt edges with no blurring. However, the characteristics of any particular

block may be different from those surrounding it, so that, coding parameters will be different from block to block. This can lead to discernible block type error. Hence, a two-level quantizer is simply not able to code a highly busy image with a satisfactory fidelity [72]. To achieve high fidelity, Wu and Coll [73,74] have implemented a multi-level quantizer and a mini-max quantization error criterion [75] to allocate the quantizer levels which will obtain a close replica of the original image and which reflects the pixel statistical distribution within each pixel block.

The Absolute Moment BTC non-adaptive scheme involving a two-level quantizer has been used as a precursor to the development and implementation of motion compensation predictive coding algorithm described in this thesis. Block truncation is a simple and effective technique for image data compression that preserves more detail than simple average-and-subsample decimation procedures. The bit-maps resulting from BTC of two consecutive frames in an image sequence are utilized in the process of motion estimation and consequent motion vector data generation. This motion vector data together with high- and low-mean subimages — which are also created during BTC — are transmitted, thereby achieving much more bandwidth compression than could be obtained by direct BTC without motion compensation.

2.3 Transform Coding

An alternative to predictive coding (spatial domain coding) is transform coding, which is a special case of block quantization, in which a vector or a block of N samples is quantized. An optimum block quantizer could be defined as one that minimizes the average distortion of the quantized elements of the vector for a given number of quantization levels. In predictive coding, the successive inputs to the quantizer are decorrelated by a nonlinear recursive filter. In Transform Coding, all the samples are first decorrelated jointly and then quantized. The optimal decorrelation filter that minimizes the mean square distortion in the reconstructed signal turns out to be a linear noncausal filter (as opposed to the causal-predictive filter) known as the Karhunen-Loeve (KL) transform [16]. In practice, the KL transform is substituted by a suboptimal but fast unitary transform [4,9,7,17]. The use of KL transform (KLT) in practical applications is limited because of its computational complexity. Amongst a variety of suboptimal transforms, the Discrete Cosine Transform (DCT) gained much interest for its superior energy compaction property [76]. In fact, an interesting property of the DCT is that it can be obtained directly from a Fast Fourier Transform (FFT) algorithm.

Transform coding achieves high compression compared with predictive methods. Generally, any distortion due to quantization and transmission errors is distributed, during the inverse transformation, over the entire image [15]. Visually this is less objectionable than the distortion in predictive coding, which is distributed locally at the source [77]. Moreover, transform coding is not sensitive to fluctuations in the statistics of the data. From an implementation point of view, predictive coding has generally a much lower level of complexity, in terms of both memory requirements and number of operations to be performed, but some fast transforms may perform as well with the same number of operations per pixel [9,10,76]. However, with the rapidly decreasing cost of digital hardware and computer memory, the hardware complexity of transform coders may not remain a disadvantage any longer.

2.3.1 Theory

An optimal orthogonal transform should completely decorrelate the transform coefficients and, therefore, minimize the geometric mean of the variance of the transform coefficients [78]. It can be shown [78] that minimizing the geometric mean of the variance of the transform coefficients corresponds to minimizing the reconstruction error variance in transform coding.

Suppose a random vector u with zero mean and covariance R is linearly transformed by an $N \times N$ matrix A to provide a (complex) vector v such that its components $\{v(k)\}$ are mutually uncorrelated. Each component $v(k)$ is quantized independently. The output vector v^* is linearly transformed by a matrix B to yield a vector u^* . It is desired to find the optimal decorrelation matrix A , the reconstruction matrix B , and the optimum quantizers such that the overall average mean square distortion, (also referred to as average quantization error variance)

$$D = \frac{1}{N} E \left\{ \sum_{k=1}^N [u(k) - u^*(k)]^2 \right\}, \quad (13)$$

is minimized. The solution of this problem requires that

$$B = A^{-1}, \quad (14)$$

that A be the KL transform of u (i.e., rows of A are the orthogonal eigenvectors of R), and that the quantizers be optimum mean square (i.e., Lloyd-Max quantizers). Since the rows of A are orthonormal, it follows that

$$A A^{*T} = I \quad (15)$$

or

$$A^{-1} = A^{*T} \quad (16)$$

that is, A is unitary. There are other unitary transforms that also have the energy packing tendency. Hence, the dynamic ranges or, equivalently, the variances of transform

coefficients are unequal. For a given quantizer, the compression performance of a transform is completely determined by the distribution of the variances of the transform coefficients. The KL transform distributes them in the most efficient way. It has the property that, for any $n \leq N$, it packs the maximum average energy into some n samples of v . Although the KL transform is optimal, it is often difficult to compute and has no fast algorithm associated with it [16]. A typical implementation of the KLT requires $2N^3$ multiplications and additions, where N is the transform block size. For stationary random sequences there are many fast unitary transforms that approach the energy packing efficiency of the KLT. Examples are the Discrete Fourier Transform (DFT), Discrete Cosine Transform (DCT), Discrete Sine Transform (DST), Walsh-Hadamard Transform (WHT), Harr Transform (HT), and the Slant Transform (ST) [9,7,76]. These transforms have been shown to be members of a large family of transforms [76] all of which have a performance equivalent to that of the KLT as the size N of the data vector goes to infinity. For first-order stationary Markov processes, the Cosine Transform matrix defined in [79] as

$$C_{i,j} = 1/\sqrt{N}, \quad i = 1, 1 \leq j \leq N \quad (17)$$

and

$$C_{i,j} = \sqrt{2/N} \cos[(2j-1)(i-1)\pi/2N], \quad 2 \leq i \leq N, 1 \leq j \leq N \quad (18)$$

has been shown to perform very much like the KLT when the correlation parameter r lies in the interval $(0.5,1)$, even when N is small. The transformation $y = C x$ can be computed by means of the FFT in $O(N \log_2 N)$ operations. These properties have made the Cosine Transform a useful substitute for the KL transform in image processing, because many images can be modeled by low-order Markov processes with high interpixel correlation.

Non-sinusoidal unitary transforms such as Hadamard, Haar (which are square-wave transforms), and the Slant Transform are also used, because they are computationally faster than the FFT-based fast sinusoidal transforms. Therefore, in transform coding practice, the KL transform is substituted by a suitable fast unitary transform.

In two dimensions, one considers the class of transformations of an $N \times M$ image U given by

$$V = A_N U A_M^T \quad (19)$$

where A_N is an $N \times N$ fast unitary transform. This is called a separable transform, because it can be realized by a column-wise transformation ($A_N U$) followed by a row-wise transformation.

To make Transform Coding practical, the given image is divided into small rectangular blocks (typically 8×8) and each block is coded independently. For example, if a 256×256 image is coded in 8×8 blocks rather than a single 256×256 block, then the storage requirement is reduced by a factor of 256, and the speed is improved by a factor of two for a transform that requires $O(N \log_2 N)$ operations for an $N \times 1$ vector.

2.3.2 Recursive Block Coding

In conventional block-by-block transform coding, each block is processed independently. Therefore, the block size has to be large enough (typically from 16×16 to 64×64) so that interblock redundancy is minimal, although, a large block size will increase the complexity of the hardware. Moreover, at low bit rates (typically less than 1 bit/pixel), the block boundaries start becoming visibly objectionable.

In recursive block coding, redundancy between successive blocks is removed to achieve additional compression while maintaining small block size (8×8 or less). The redundancy between successive blocks can be modeled explicitly using the block boundaries by means of a theory of noncausal representations [80,81]. In addition to reduced transform size requirement, such encoders also tend to minimize the block boundary effects.

2.3.3 Two-Source Coding

Here an image is considered as a composition of smooth and flat areas that can be represented by low spatial frequencies and edges represented by high spatial frequencies [16]. For encoding, the given image is filtered to separate its low- and high- frequency components. The low-frequency component is transform coded. The addresses of the edges obtained from the high-frequency component are coded separately. At the receiver, the low-frequency component is recovered by a transform decoder to which synthesized edges called synthetic highs are added at the locations of the edges. A synthetic high is a signal that is proportional to the second derivative (or difference) of a step function. In another two-source coding scheme [82] the high frequency component is taken to be the sharp changes in the local mean of the image. The low frequency component is the difference between the original image and the sharp changes. Another method [83] is based on the use of certain stochastic image models to achieve the two-source decomposition.

2.3.4 Adaptive Transform Coding

Performance of Transform Coding schemes can be improved substantially by adapting them to changes in image statistics [16]. There are essentially three types of adaptation that could be made: adaptation of transform, adaptation of bit allocation, and adaptation of quantizer levels.

Theoretically, a change in statistics of the data could require all of these adaptations. Adaptation of the transform basis vectors is the most difficult and expensive because , ideally, one should find a new set of KL basis vectors for any change in the statistical parameters. A method of this type was considered by Tasto and Wintz [84]. From as practical standpoint, if one knows the range and type of statistical variations, one can choose a single transform that would be least sensitive to such changes.

A more practical and perhaps more effective method is to adapt the bit assignment to changes in statistics [16]. For example, one could classify an image block into one of several categories and allocate a larger number of bits to blocks that have larger activity and fewer bits to those having lower activity. This results in a variable average rate from block to block but gives a better utilization of the total number of bits over the entire ensemble of image blocks.

Another adaptive scheme is to allocate bits to image blocks so that each block has the same distortion [85]. This results in a uniform degradation of the image and appears less objectionable to the eye. In adaptive quantization schemes, the bit allocation is kept the same but the quantizer levels are adjusted according to changes in the variances of the transform domain samples.

A number of Discrete Cosine Transform algorithms which are adaptive in nature [86–89] have been implemented. Among them, Chen and Pratt's [88] seems to be most successful one. All these adaptive algorithms are based on the observation that images are composed of objects which are in turn a combination of edges and areas of varying degrees of detail. Within each block the degree of picture detail changes drastically. An efficient encoding of these blocks requires more bits for areas of high detail and fewer bits for areas of low detail. This results in a variable average bit rate from block to block, but gives a better utilization of the total bits over the entire ensemble of image blocks [2]. A buffer and some buffer control schemes must be used to transmit the encoder outputs over a fixed rate channel.

The DCT phase, described in the recent work of Wu and Coll [73,74] on BTC-VQ-DCT Hybrid Coding of digital images, is primarily an adaptive Discrete Cosine Transform technique, and the same has been utilized in one of the stages in the present work that concerns transmission of high- and low-mean sub-images generated from BTC scheme applied on current and previous frames of the image sequence.

2.4 Hybrid Transform/Predictive Coding

Hybrid coding refers to techniques that combine transform and predictive coding techniques [6,16]. The Hybrid Coding scheme described by Habibi [6], combines the concepts of transform coding and DPCM (predictive coding) into a system that achieves essentially the same performance as two-dimensional transform coding, but it is easier to implement. The scheme uses one-dimensional transform encoding in the horizontal direction and DPCM in the vertical direction.

In Hybrid Coding schemes, typically, the image is unitarily transformed in one of its dimensions to decorrelate the samples in that direction. Each transform coefficient sequence is then coded independently by a one-dimensional predictive technique such as DPCM in the other direction. This technique combines the advantages of hardware simplicity of DPCM and the robust performance of transform coding [15,90]. The hardware complexity of this method is that of a one-dimensional transform coder and at most N DPCM channels, where N is the size of the transform basis vectors. The number of DPCM channels is significantly less than N , because many elements of the transformed vector are allocated zero bits and, therefore, are not transmitted at all.

A Hybrid Coding scheme could be adapted to an image whose spatial statistics vary slowly, by updating the parameters of the model. Adaptive schemes should be chosen that offer reasonable trade-offs between performance and complexity of the coder [16]. For a fixed predictor in the feedback loop of a DPCM channel, the variance of the prediction error will fluctuate with changes in the image statistics. A simple method is to update the variance of the prediction error at each step and use it to adjust the spacing of the quantizer levels in each DPCM channel.

Another adaptation is based on a classification method in which each image column is classified as belonging to one of k predetermined classes that are fixed according to the activity in that image column, as measured by its variance. Quantization bits are allocated according to their dynamic activity. The classification information is communicated by

sending an extra $\log k$ bits per column. Experimentally, it has been found that the compression increases by a factor of almost two for the adaptive hybrid techniques as against the nonadaptive hybrid algorithms.

2.5 Vector Quantization Coding

Shannon's rate-distortion theory states that it is always possible to achieve, in theory, a better performance by coding vectors instead of scalars, even though the source is memoryless [60,91,92]. However, this theory does not provide a constructive design technique for vector coders. Vector quantization design techniques and various vector quantization algorithms have only been developed in recent years for speech and image coding applications. [34,93–98]. In Vector Quantization (VQ), the first step is to decompose the input image into a set of vectors. A subset of vectors is then chosen as the training sequence from which the codebook is generated, normally, using the generalized Lloyd clustering algorithm proposed by Linde, Buzo, and Gray (LBG) [99]. The LBG algorithm generates, in an iterative manner, a locally optimum codebook of size, N , from a given initial codebook, refining the codebook at each iteration until the distortion is within acceptable limits. In the quantization process, for each input vector, the codebook is searched to obtain the closest matching codeword. Compression is achieved by transmitting the index (label) of the codeword. Reconstruction of images can be implemented by simple table look-up techniques where the label is used as an address to a table containing the codewords.

In VQ, a fixed or universal codebook is generated which minimizes the long-term average distortion of coding training set. If the individual images have different statistics, they may not be represented well [100]. An alternative strategy would be, therefore, to design a codebook which more closely matches the local frame statistics. A number of inter- and intra-image adaptive techniques have been proposed in the literature. For

example, an inter-frame adaptive technique (VQIAC) presented by Goldberg et al [101] is based on the design of a new codebook for each image by using the input image vectors themselves as the training set. Gersho and Yano [102] developed a similar algorithm in which the adaptation involves changing only a small part of the codebook rather than the entire codebook. The algorithm monitors the distortion for each input vector and if it is larger than a predetermined threshold, that input vector is added to the codebook, as a new vector. Yeh [103] has presented a codevector replacement scheme incorporating adaptive overhead transmission for both still and image sequences. The fixed overhead information contains the replaced codevector which represents the cluster of vectors better than the codevector generated from the training images. Goldberg and Sun [101] have proposed a replenishment technique for image sequences. Here, the changes in the successive frames are tracked by replenishing the labels and the codebook. The coding performance may be improved further by employing intra-image adaptive techniques where, the codebook is adapted to local regions of the image. For example, Goldberg and Boucher [95] have proposed a technique where a small codebook is designed for different portions of the image. Aravind and Gersho [104] proposed the use of a state transition function F to predict the state of each input vector from the previous reproduction vectors. The state variable, s , identifies which subcodebook in the codec's supercodebook to use to encode the input vector.

In all the techniques described above, the mean square error criterion has been used to find the closest codeword for a given input vector. This measure does not guarantee that the closest codeword for a given input vector is within a prespecified bound. This can result in large errors which give rise to artifacts. Panchanathan and Goldberg [100] have proposed an intra-image adaptive VQ technique which employs a criteria that minimizes the maximum error. Here, the codebook is generated on the fly from the input vectors to be coded. In this technique, using the mini-max error criterion for VQ, the receiver tracks the transmitter without requiring any additional overhead. Simulations using this algorithm have demonstrated excellent coding performance and have confirmed the absence of large errors when compared to the VQIAC technique.

It can be seen that the key issues in the VQ Coding scheme are selecting the training sequence and codebook generation. An optimal codebook should completely reflect the statistics of the input vectors. A codebook generated by the LBG algorithm is optimized for the particular training sequence, chosen as the representative of the input image source. VQ technique achieves significant performance advantage over scalar quantization and decoding procedure can be realized by a simple look-up table (LUT) operation. However, the codebook design and quantization procedures involve immense computation.

Wu and Coll [73,74] in their recent work have implemented a hybrid BTC-VQ-DCT Coding scheme for still images in which the second phase utilizes the concepts of Vector Quantization for encoding the bit-map generated as a consequence of first phase of BTC applied on the input image. The codebook was generated from a group of training images that included standard images, satellite images, and radar images. A 64-K word encoding-LUT was built from the codebook through an exhaustive-search, based on minimum Hamming distance and the moments preserving scheme.

2.6 Interframe Coding and Motion Compensation

Interframe Coding techniques exploit the redundancy that is present between successive frames of a moving sequence of images. This redundancy, or similarity, occurs in regions where there is relatively little motion between frames. The sources of motion are object motion and the associated uncovering of background areas and camera motion, panning, and zooming. Much of the research on interframe image coding has been done recently and is primarily based on spatial-temporal domain techniques. A brief description of frame-repeating, resolution exchange, and conditional replenishment coding schemes is given followed by a detailed description of motion compensation techniques which form the basis for the work presented in this thesis.

2.6.1 Frame-Repeating

In broadcast TV systems, the picture is displayed at a rate of at least 50 frames/sec to prevent flickering of the display. However, a rate of 30 frames/sec is adequate, and often more than adequate, for displaying motion. This can be likened to the high spatial resolution specification, which is required only in regions containing great detail. [16].

The most widely used technique for obtaining a transmission rate below the flicker frequency is line interface, whereby each frame is divided into two interleaved fields (odd and even), each containing half the number of lines present in a frame. During a 1/60th second interval only one field is transmitted, so that each pixel is displayed once every 1/30th second. This is half the flicker rate, but the eye hardly perceives this 30 Hz flicker, because it temporarily smooths the displayed signal. Attempts to reduce the data rate by frame-repeating the interlaced pictures do not produce quality moving images.

An alternative approach is selective replenishment, which requires a frame memory at the receiver only. The picture information is transmitted at a reduced rate according to a fixed, predetermined updating algorithm. At the receiver, any nonupdated data are replaced by corresponding data from previous frames. This technique is simple and produces good quality pictures during periods of little or no movement. However, for more rapid motion, the sampling pattern becomes visible at the edges of the moving regions [105].

2.6.2 Resolution Exchange

It is well known that scenes containing little or no movement can be transmitted and reproduced with good quality using frame-repeating techniques. Furthermore, changing areas of an image can be represented with reduced amplitude and spatial resolution when compared with stationary areas [106]. This resolution exchange phenomenon can be used in straight-forward coding schemes to produce good quality images at a constant data

rate of 2 to 2.5 bits/pixel. One such scheme segments the image into stationary and moving areas by thresholding the value of the frame-difference sequence. In stationary areas, frame differences are transmitted for every other pixel, and the remaining pixels are repeated from the previous frame. In moving areas, 2:1 horizontal subsampling is used with intervening elements restored by interpolating along the lines. The main distortion caused by this system occurs at sharp edges moving with moderate speed.

2.6.3 Conditional Replenishment

In this scheme, the image is segmented into stationary and moving areas, and data are transmitted only within the moving areas. This is a more complex technique that requires a frame memory at both the transmitter and the receiver and a buffer at the transmitter to smooth the data, which are generated at a variable rate according to the amount of motion present in the scene. However, it offers much greater compression than selective, or unconditional, replenishment techniques for a comparable picture quality [16]. A typical interframe predictive coding algorithm using conditional replenishment (CR) requires the following sequence of steps: (i) Segmentation of moving and stationary areas; (ii) Prediction of a pixel in a given area from pixels in the previous frame(s) and from pixels in the given frame; (iii) Spatial and temporal resolution exchange used in bit allocation; (iv) Temporal filtering to reduce jerkiness of motion in the reconstructed image; (v) Buffer control.

It has been shown that sophisticated prefiltering of the source video prior to encoding can decrease the noticeable visual degradations due to mode changes (e.g., subsampling) in a conditional replenishment coder.

2.6.4 Motion Compensation

A recent improvement over Conditional Replenishment is Motion Compensation, which adaptively varies the predictor, usually according to an estimate of the translational motion of objects [14,19–21,26,29–31,35–39,108–111]. If the estimate is accurate, the prediction is improved by use of appropriately spatially displaced pixels from the previous frame. For typical moving image sequences such as video-conference images quantized to 8 bits/pixel, Motion Compensation might save 2[^]-100% of the total bit rate required by a Conditional Replenishment coder. The saving clearly depends on the amount of motion and the accuracy of the motion estimation algorithm. Conditional Replenishment (CR) uses the statistical fact that information is present in a scene only when motion occurs and only in selected areas of the frame [112]. The technique is suitable for applications where motion in the scene typically does not exceed 15% in a frame interval [112]. Such is usually the case for most videoconference codecs. Instead of simply transmitting interframe differences, motion is estimated for a pixel or a block of pixels, and differences are taken with respect to the displaced location. The entropy of the displaced frame difference (DFD) is much below that of frame differencing, indicating the higher quality of compression achieved by compensating for motion.

2.6.4.1 Pel Recursive & Block Matching Algorithms (PRA and BMA)

Motion compensated prediction schemes can be divided basically into two types: (i) Pel Recursive algorithms (PRA) or method of differentials; and (ii) Block Matching algorithms (BMA) or pattern matching algorithms.

The PRA techniques start with an initial estimate of the motion parameters and revise these estimates at predetermined rates. For the most part, the motion is estimated for every *pel* (pixel). The PRA techniques developed up to now seem to be complex for applications like videoconferencing implementation. On the other hand, BMA techniques estimate motion vectors for a block of *pels*, say of size ($M \times N$) in the current frame in relation to a reference (previous) frame. A spatially centered search area of size

$(M+2p) \times (N+2p)$ is chosen in the reference frame. In this set up there are $(2p+1)^2$ possible blocks in the search area that can be a match to the $(M \times N)$ block. The best match typically is sought by maximizing the cross-correlation or minimizing the mean square error. This simple pattern matching technique assumes translational motion for the blocks. BMA handled in this fashion still is expensive in computational terms, but the number of search points can be lower.

In general, simulations have shown that coding using Motion Compensation has resulted in better reconstructed picture quality and more compression. Such results lead to the claim that motion compensation is the best way to achieve optimum image compression [110]. Compensating for motion is a natural extension of currently available interframe codecs that suffer from tearing or jerkiness problems in motion conditions. Motion Compensation has started becoming part of what is already being called the full motion videoconferencing codec. Motion Compensation is a good solution to Conditional Replenishment codecs that suffer from buffer overflow and a resultant loss of tracking in motion conditions. Simulations have further shown that PRA can track complex motion such as rotation, and translation better than BMA. This is because, on an individual basis, *pel* motion in the interframe interval can be viewed as strictly translatory in nature. However, from the viewpoint of developing hardware for real-time implementation, it is felt that BMA lend themselves to easier implementation. It is believed that block algorithms, while not providing the performance offered by PRA, do, indeed, perform well enough to warrant more facile hardware design.

The motion compensated interframe predictive coding techniques, whether based on PRA or BMA, have almost doubled the data compression ratio compared with conventional interframe Conditional Replenishment coding schemes; and extensive studies and developments in this direction will continue to be conducted to improve the data compression ratio greatly.

2.6.4.2 Block Matching Algorithms (BMA)

The variety of Block Matching Algorithms (BMA) for motion compensated inter-

frame predictive coding that have evolved over the past few years, have primarily described ways to reduce the number of searches [14] that should be carried out if a thorough search is used in finding the motion vector in a BMA [14]. It may be recalled that $(2p+1)^2$ searches are needed for thorough detection of the best matching block, where p is the maximum displacement. Each of these searches involves computation of mean square error (MSE) or mean of absolute error (MAE) over 4×4 , 8×8 or 16×16 displaced frame difference (DFD) blocks. This is a costly proposition, and as such is not suitable for motion estimation in real time situations. Koga et al [109] proposed a way to solve this problem in their coarse-fine 3-stage search. In every stage, the center point and eight other points at 45° steps are searched. At every stage the point at which the minimum distortion occurs becomes the center for the next stage. With a reduction in spacing between points in every stage, estimate of the displacement is found in 24 searches in comparison to 169 that would have been required by brute force.

Jain and Jain [107] provide another approach to the motion estimation problem. It is an extension of the well known 1-D binary search algorithm [112] in data structures. An assumption that the covariance function of images is a nonincreasing function simplified the problem to a great extent. Srinivasan [113] proved by examples that the distortion function for various displacements in the search area behaved as though it increased gradually away from the minimum. The 2D-logarithmic search required searching 13 to 21 points for a maximum displacement of $p = 5$. Mean Square Error (MSE) measure was used, thus still requiring quite a number of multiplications to implement the algorithm, though typically 10 dB improvement was registered in comparison to plain frame differencing [114].

Jain and Jain [107] have described an interframe hybrid scheme which makes use of the Discrete Cosine Transform (DCT). An important result in their scheme is that of frame skipping and resultant interpolation along the motion trajectory. At the receiver skipped frames are filled using the motion vectors in an interpolation with improved results. A significant 10 dB improvement is reported over interpolation along the temporal axis.

Kappagantula and Rao [108] have described an algorithm using Mean Absolute Error

(MAE) measure as a mismatch function, and their scheme requires far fewer positions, typically, 7, to be searched. It uses a combination of NEC's technique [109] and the 2D-logarithmic search with a threshold value to exit the search at every stage. Though the computations are reduced greatly, the results do not seem to be very encouraging, particularly for bigger block sizes. Apparently, frequent thresholding causes this method to take premature exits.

The mismatch function used in the above method is the Mean Absolute Error measure which computes the mean of the absolute differences between the two sub-blocks in issue (current frame subblock and the subblock of the previous frame) and can be written in the form:

$$M(p, q) = \frac{1}{mn} \sum_{k=1}^m \sum_{l=1}^n [u(k, l) - u_R(k + p, l + q)] \quad (20)$$

where $u(k, l)$ denotes the location of the *pel* in the current frame subblock, i.e., row k and column l and $u(k + p, l + q)$ denotes the *pel* in the subblock being searched in the previous frame at search position $(k + p, l + q)$. The computational complexity of this measure is significantly less than the MSE or correlation measure.

The algorithm proposed by Kappagantula and Rao can be briefly described as follows:

(1) The block characterized by its upper left corner *pel* (i, j) is compared with the corresponding block in the previous frame. If $M(0, 0) < T$, where T is a threshold function (say 4 out of 255), then the block is classified as an unchanged or stationary block and the search ends there. Otherwise go to step 2.

(2a) The next *pel* positions searched are $(i-4, j)$, $(i, j+4)$, $(i+4, j)$, and $(i, j-4)$. If the minimum from these searches is $M(0, 0)$ go to step 3. If this is not true and a minimum is found for the searches in step 2a, then this value is compared against T . If the value is T , the search ends there. Otherwise go to step 2b.

(2b) Assume that from the previous step the *pel* position that had the minimum mismatch was at $(i-4, j)$. The next positions of search are $(i-4, j-4)$ and $(i-4, j+4)$ respectively. The test for a minimum and the threshold are performed again. If the test is true the search stops. Otherwise go to step 3a.

(3a) This is similar to step 2a except that the spacing of the search positions is now reduced by $1/2$.

(3b) This is similar to step 2b.

(4a) The spacing is now reduced further by $1/2$.

(4b) This is the final possible step of the search.

The number of searches in the above algorithm can vary between 1 and 19. This number in the normal sense means 1–19 sequential computations. However, the positions searched in the previous frame are largely independent of one another. This suggests that parallel processing elements can be used to speed up the entire search considerably. This is also warranted by the fact that each search step involves at least two and up to four computations from which a decision has to be taken. Hence, these can be performed simultaneously.

With the average number of search points as 7 (compared to 121 by brute force in exhaustive search.), Srinivasan and Rao [26] adopted an optimization procedure. Having in view that the distortion (mismatch) function behaved as a convex function [26], they suggested that the key to solving this problem is by optimization algorithms, which already exist in the literature [116]. They proposed an advanced search method using conjugate directions. However, for a limited search area, a simpler technique called one-at-a-time search (OTS) seems to be very effective. With a 94% reduction in the number of computations, and still using Mean Absolute Error (MAE) as the search measure, the results recorded in this paper are very promising. With the number of points searched less than half those required by the 2D-logarithmic search and close performance, this algorithm seems to be effectively implementable, typically, in videoconferencing codecs. There is at least one additional feature that can simplify this algorithm further without affecting the performance very much. It is a threshold for the distortion function value at the center of the search area to exit the search. Extensions or other optimization algorithms are possible too.

Lin and Kwatra [37] have described a modification to Koga's algorithm [109]

based on the amount of activity in the subblocks containing the DFD. No information is transmitted if no activity is found. Like an address information in Conditional Replenishment (CR), this needs a flag to set the activity or inactivity indicator. With a study on three different block sizes, an 8×8 block was finally adopted in their scheme as an optimum for BMA implementation. A bigger block size assumes constant motion over four times (for a 16×16) this area, which may not be too realistic an assumption. However, Jain and Jain [108] did use 16×16 blocks. A smaller one (e.g., 4×4) can create problems with the increased number of motion vectors to be transmitted [14].

It may be appropriate to add a few words about the choice of mismatch (distortion) functions for search algorithms. Selecting the best matching block in the search area by Cross Correlation function is a technique that has been widely used in "area correlation". Minimizing the Mean Square Error (MSE) should give equally good results. However, both methods require an enormous number of multiplications. Mean Absolute Error (MAE) is a measure that requires no multiplications [110]. It has been investigated and compared against MSE in the work described by Srinivasan and Rao [26] with encouraging results [113]. These results suggest that MAE is a potential candidate for hardware implementation. Motivated by these findings, the motion search algorithm developed in this thesis uses MAE as the basis and may be referred to as modified MAE involving exclusive-or (XOR) bit-operations on bit-maps as representative of absolute difference measurement operations. Replacing arithmetic add operations in calculating MAE by exclusive-or bit operations in obtaining modified MAE as the measure for mismatch function evidently leads to a great amount of computational simplicity.

2.7 Summary

In this chapter, different digital image coding algorithms have been reviewed with particular emphasis on Block Truncation Coding and Interframe Motion Compensation Predictive Coding schemes using block matching techniques. The adaptive Discrete Cosine Transform coding technique has also been described in greater length.

Transform Coding schemes achieve higher compression compared with predictive methods (Spatial Coding). Generally, any distortion due to quantization and transmission errors is distributed, during the inverse transformation, over the entire image. Visually, this is less objectionable than the distortion in predictive coding, which is distributed locally at the source.

Block Truncation Coding (BTC) has generally a much lower level of computational complexity, and edge preserving ability, and single pixel resolution, but only a medium compression ratio. Vector Quantization provides high compression ratio; fast, LUT-based, decoding, and competitive subjective performance; but the encoding procedure is quite time consuming. Adaptive Discrete Cosine Transform Coding (DCT) has high compression ratio especially at low bit rates, and good mean square error performance, but both encoding and decoding processes are computationally complex. All these coding algorithms are robust in the presence of transmission errors, which implies high reliability on real channels.

It has been proved beyond doubt that coding using Motion Compensation results in better reconstructed picture quality and more compression. Compensation for motion is a natural extension of currently available interframe codecs that suffer from tearing or jerkiness problems in motion conditions. Motion compensated prediction schemes based on block matching algorithms (BMA) have an edge over the pel recursive algorithms (PRA) from the viewpoint of ease and computational simplicity and developing hardware for real time implementation. The BMA schemes for Motion Compensation, while not providing the performance offered by PRA, do perform well enough to warrant the trade-

off in picture resolution with more facile hardware design.

Since image bandwidth compression was the prime consideration of this chapter, a chart has been prepared that compares bandwidth reduction ratios for the various techniques studied. This information is shown in Table 2.1 [2,10].

The Absolute Moment BTC non-adaptive scheme involving a two-level quantizer has been used as a precursor to the development and implementation of motion compensated predictive coding algorithm described in this thesis. Block truncation is a simple and effective technique for image data compression that preserves more detail than simple average-and-subsample decimation procedures. The bit-maps resulting from BTC of two consecutive frames in an image sequence are utilized in the process of motion estimation and consequent motion vector data generation. This motion vector data together with high- and low-mean subimages — which are also created during BTC — are transmitted, thereby achieving much more bandwidth compression than could be obtained by direct BTC without motion compensation.

The motion search algorithm developed in this thesis uses MAE as the basis and may be referred to as modified MAE involving *exclusive-or (XOR)* bit-operations on bit-maps as representative of absolute difference measurement operations. Replacing arithmetic *add* operations in calculating MAE by *XOR* bit-operations in obtaining modified MAE as the measure for mismatch function evidently leads to a great amount of computational simplicity.

The work presented in this thesis also makes use of Run Length Coding scheme for the transmission of motion vector data (indicating motion of various regions in the image sequence) to reconstruct the current image from the previous image with respect to which it has moved. Run-length coding of the motion vector data is found to be most efficient in this scheme owing to a relatively large proportion of the image scene being static, i.e., each of the blocks in this region are represented with $(0,0)$ as the components of the motion vector. Run Length Coding scheme can also be implemented on the bit-map of the previous frame in an image sequence to yield additional compression.

Table 2.1 Bandwidth Reduction Ratios for various Image Coding Techniques.

Technique	Typical Bandwidth Reduction Ratio (Image Compression)	Average bit-rate (bits/pixel)	Comments
<u>Spatial Coding</u>			
Block Truncation Coding (BTC)	4 : 1	1 - 2	Very simple implementation
Laplacian Pyramid Coding	4 : 1	1.5 - 2	Simple computation
Intraframe DPCM	4 : 1	2 - 3	Simple implementation
Intraframe Adaptive DPCM	4 : 1	1 - 2	Simple implementation
Interframe DPCM	6 : 1	1 - 1.5	Simple implementation
Interframe Adaptive DPCM	8 : 1	0.5 - 1	Simple implementation
Vector Quantization	6 : 1	0.4 - 1.5	Extremely complex
<u>Transform Coding</u>			
Intraframe	6 : 1	1 : 1.5	Highly complex
Intraframe Adaptive	8 : 1	0.5 - 1	Highly complex
Interframe	8 : 1	0.5 - 1	Highly complex
Interframe Adaptive	16 : 1	0.1 - 0.5	Extremely complex

Table 2.1 (Continued.)

Technique	Bandwidth Reduction Ratio (Image Compression)	Average bit-rate (bits/pixel)	Comments
<u>Hybrid Coding</u>			
Intraframe	4 : 1	1 - 2	Complex
Intraframe Adaptive	6 : 1	0.5 - 1.5	Complex
Interframe	8 : 1	0.5 - 1	Complex
Interframe Adaptive	16 : 1	0.3 - 0.5	Complex
Interframe with BTC based Motion Compensation	20 : 1	0.2 - 0.3	Simple computation

Chapter 3

MOTION COMPENSATED BTC-DCT HYBRID VIDEO CODING

3.1 Introduction

This chapter describes a motion compensated predictive coding algorithm using BTC as a basis for detecting motion and then, motion compensation and DCT for achieving an added component of compression to supplement the compression obtained with motion compensation. It may, therefore, seem appropriate to call the complete scheme as Motion Compensated BTC-DCT Hybrid Video Coding.

The method combines the simple computation and edge preservation properties of BTC, the high fidelity and high compression ratio of adaptive DCT, and the simple operations involving only exclusive-oring (XOR) of the bit-maps of two consecutive image frames, obtained by BTC, for detecting and compensating for motion in the sequence of images. The input image sequence first undergoes BTC encoding, which codes the input images (previous and current frames) into bit-maps, and four reduced-sized gray level sub-images (two each for previous and current frames), which are composed of the "high-mean" and "low-mean" values from each block. The BTC bit-maps are exclusive-ored to segment the images into areas of activity (motion) and non-activity (stationary surroundings, environment etc.). Obviously, exclusive-oring on bit-

maps is by far the fastest and the simplest means for motion detection in the consecutive image frames. The stationary structures cancel out almost completely, while the displaced structures produce considerable contrast, which for bit-maps with bi-levels typically set to 0 and 255, for display, show as bright white regions when displayed, i.e., gray level of 255. Some of the stationary structures, however, may produce residual derivative outlines in the exclusive-ored difference image due to imperfect registration between the two frames. The motion compensation algorithm operates on the two bit-maps using typically the 4×4 block matching algorithm, which is developed on lines similar to most existing BMA schemes, but introduces the outstanding factor of computational simplicity by using exclusive-or operations.

The overall computational complexity of BTC-DCT based motion compensation predictive coding described in this thesis is much less than existing algorithms while the fidelity performance is competitive. The hybrid scheme, as such, has strong edge preserving ability because of the implementation of BTC as a precursor. The reconstruction of the current frame is obtained from the immediate previous frame by applying the motion vectors generated by the motion compensation algorithm and combining this information with the "high-mean" and low-mean" sub-images which are transmitted in the most efficient compression scheme using adaptive DCT as described by Wu [2].

3.2 Motivation of BTC as a basis for motion detection and compensation

There are three criteria that may be used to evaluate the performance of an image compression algorithm: computational complexity, compression ratio, and fidelity. The fastest computations, moderate compression ratio, and psycho-visually (subjectively) acceptable quality of the reconstructed image using motion compensation were the main considerations in developing and implementing the entire scheme described in the

present work. The overall computational complexity of our Motion Compensated BTC-DCT Hybrid Video Coding scheme is much less than the existing motion compensation algorithms (see Section 3.3), while the fidelity performance is comparable as assessed on the psycho-visual subjective criterion.

3.3 Description of Motion Compensated BTC-DCT Hybrid Video Coding Algorithm

The BTC based Motion compensated BTC-DCT Hybrid Video Coding algorithm is developed in the following four phases:

- **Phase I: Prefiltering Phase.** Prefiltering of the input sequence of frames with a low pass filter is required in view of the accuracy of the displacement estimates being normally influenced by noise in the images and sensitivity of operations on bit-maps. The main source of noise in the input images is due to Video Camera noise. The impact of this source is greatly reduced by a digital low pass filter applied on the input image sequence before using BTC for bit-map generation.
- **Phase II: BTC Coding of Current and Previous Frames.** The BTC of the current and previous smoothed image frames results in compressing blocks of the original 512×512 gray level images into two “bit-maps” and four reduced-sized gray level sub-images (two for each image). The sub-images comprise the “high-mean” and low-mean” values of each BTC block. The primary compression ratio of this phase is 4:1 for 4×4 blocks.
- **Phase III: Adaptive DCT Coding of High- and Low-Mean Sub-images.** Adaptive DCT Coding is implemented to compress the 128×128 size “high-mean” and low-mean” sub-images in the most efficient way. It removes both inter- and intra-block correlations among high- and low-mean pixel values. A compression ratio of 3:1 may be achieved on transmission of high- and low- mean subimages by this algorithm.

This scheme is directly adopted from Wu [2] and no effort is made to introduce any change in this algorithm.

- **Phase IV: Motion Compensation using Bit-Maps.** The motion compensation algorithm is developed and implemented on the bit-maps of the current and previous frames in the image sequence. The steps involved are: (i) generating an XOR difference image called *bindif* from the bit-maps, *bin1* & *bin2*, of current and previous frames. (ii) searching the *bindif* bit-map using 4×4 blocks to isolate motion areas from static areas, the regions containing a large number of 1's (i.e., areas of significant difference) corresponding to displacement between the frames. (iii) For every motion block detected above, to carry out a limited, but effective and efficient, search to find the best match between the current block and a block in the search region of the previous frame. The search is carried out in three stages: the first stage covers a region of 4 pixels in the previous frame with respect to the current block position in eight symmetrical directions around the current position co-ordinates. If perfect match is not found, the search is extended to a second stage of 2 more pixels, which, in the case of imperfect match, is extended to 1 more pixel. The entire search process involves merely exclusive-or operations on current and previous blocks with some trivial counting operations.

3.3.1 Phase I: Prefiltering

As earlier pointed out, noise in the video signal will reduce the efficiency of the coder using any interframe motion compensated video scheme. In other words, it will not only unnecessarily increase the generated bit rate but also degrade the accuracy of motion prediction. Therefore, the use of noise reduction techniques prior to coding of the video signal is highly desirable.

The low pass filter operation used in this study is a 3×3 convolution process where every pixel in the image frame has its adjacent neighbors multiplied by the corresponding

values in the 3×3 kernel shown in Fig.3.1(b). These values are then added, normalized, and used to replace the pixel's original value. Fig.3.1(a) illustrates the basic principle.

Figure 3.1 3×3 convolution process for low pass filtering.

The pixel value for the pixel at the position P5 is given by

$$P5 = \frac{k1P1 + k2P2 + k3P3 +k9P9}{k1 + k2 + k3 +k9} \quad (21)$$

If required, further reduction in noise can be achieved by averaging over two consecutive images in the image sequence, on pixel by pixel basis, before applying the 3×3 convolution operation. In the averaging process over two images, the stationary component of the image sequence is unchanged, whereas the noise pattern, different from one image to the next, builds up more slowly in the pixel by pixel sum. Averaging over two images, for instance, increases the signal-to-noise power ratio by a factor of 2 at all points in the image [9]. The signal-to-noise ratio, which is the square root of the power ratio, therefore, goes up by $\sqrt{2}$ i.e., the square root of the number of images over which averaging is carried out.

In our implementation of motion compensation on sequence of images, the above approach of trying to achieve additional reduction in noise must be applied with reservation, since large motion between frames will not justify averaging even between two adjacent

frames without apparent degradation in perception of the final reconstructed image from the previous image using final motion compensation phase. In initial preliminary trials, however, the video camera in use was very old and gave extremely noisy image sequences and in those situations, we did make use of the above procedure of averaging over two frames as a means of additional noise reduction.

3.3.2 Phase II: BTC Coding of Current and Previous Frames.

From the description of the BTC scheme given in the previous chapter, it is noticed that the scheme is essentially a 1-bit adaptive moment-preserving quantization process, which preserves certain statistical moments of small blocks of the input image in the quantized output. The original BTC algorithm [68] preserves the mean and standard deviation in each block. Other quantization schemes result in less mean square error (MSE) [71], mean absolute error (MAE) [69], and minimax error [75].

The absolute moment BTC described in [69] quantizes the pixels in each block into "high-mean" and "low-mean", which will preserve the mean absolute moment or minimize the mean absolute error in each block. This algorithm has the potential of low computational complexity with no multiplications involved.

Each of the input images is first partitioned into non-overlapping $n \times n$ pixel blocks in which there will likely be fairly high pixel-to-pixel correlation. In each pixel block, the block mean given by

$$\mu = \frac{1}{N} \sum_{i=1}^N X_i = \bar{X} \quad (22)$$

is used as the threshold for the 1-bit quantizer, where X_i is the value of i th pixel in the block and N is the number of pixels in a block, i.e., $n \times n$. If

$$X_i \leq \bar{X} \quad \text{or} \quad X_i > \bar{X}, \quad (23)$$

$\overline{X}_L$ or $\overline{X}_H$, respectively, is assigned to the quantizer output levels, where

$$\overline{X}_L = \frac{1}{N-q} \sum_{X_i \leq \overline{X}} X_i, \quad (24)$$

and

$$\overline{X}_H = \frac{1}{q} \sum_{X_i > \overline{X}} X_i, \quad (25)$$

and q is the number of pixels having value higher than $\overline{X}$. Hence, the outputs of the BTC for each block include two numbers which specify the "high-mean", $\overline{X}_H$, and "low-mean", $\overline{X}_L$; and a bit-map which specifies the state, $\overline{X}_H$ or $\overline{X}_L$, of each output pixel.

Figure 3.2 shows a pixel block data before and after BTC.

116	121	126	133	0	0	0	0	130	130	130	130
126	133	142	151	0	0	0	1	130	130	130	165
142	151	160	170	0	1	1	1	130	165	165	165
160	169	176	183	1	1	1	1	165	165	165	165

(a) Original data

(b) Bit map

(c) Decoded data

Threshold: $\overline{X} = 147$, High mean: $\overline{X}_H = 165$, Low mean: $\overline{X}_L = 130$.

Figure 3.2 4×4 pixel block data before and after BTC.

In our implementation of BTC phase, 4×4 block size is used and each pixel is represented by an 8-bit gray-shade value. The resulting bit rate is

$$\frac{(2 \times 8) + (4 \times 4)}{4 \times 4} = 2 \text{ bits/pixel} \quad (26)$$

Thus, the compression ratio of Phase II is 4:1.

During this phase, we transmit the bit-map (after compression) of previous frame, the "high-mean" and "low-mean" sub-images of the previous frame (after adaptive DCT

compression) and only the "high-mean" and "low-mean" (or high-low difference) sub-images of the current frame (using adaptive DCT compression), ignoring completely the transmission of the bit-map corresponding to the current frame since it has to be reconstructed from the previous frame bit-map after applying the subsequent motion compensation phase of the algorithm which leads to transmission of the motion vectors for each motion block. As would be seen later in the description, the bit-rate for transmitting motion vectors data is much less than transmitting the bit-map of the current frame.

In Phase II, bit-maps are thus formed for previous and current frames in the image sequence, and they are available in the buffers for applying the motion compensation phase for reconstructing the current frame from the previous frame without the bit-map of current frame being transmitted.

3.3.3 Phase III: Adaptive DCT Encoding of High- and Low-Mean Sub-Images.

The high- and low- mean sub-images created in Phase II for each of the current and previous frames are coded using adaptive DCT as described by Wu [2] and transmitted together with the compressed bit map of the previous frame. Details on implementing adaptive DCT compression scheme for high- and low-mean sub-images can be seen in Wu's recent Ph.D. thesis [2]. No amendment or modification was found necessary in this scheme. This hybrid scheme leads to a data requirement of about 1.3 bits/pixel.

3.3.4 Phase IV: Motion Compensation Algorithm using Bit-Maps (BTCMCP421)

The motion compensation algorithm, using bit-maps created during Phase II of BTC encoding, developed in this thesis, involves minimal amount of computational complexity

and is found to be effective for estimating the motion vectors in a video sequence. The algorithm utilizes a search scheme based on the bit-maps of two consecutive frames in the image sequence created during Phase II which employs 4×4 block BTC for bit-map generation. A block diagram of BTC based Motion Compensated Codec is shown in Fig. 3.3.

Figure 3.3 (a) Motion Compensated Interframe Codec. (b)

Block Diagram of BTC based Motion Compensated Algorithm.

(b)

Figure 3.3 (Continued.)

As a first step in this phase, a difference bit-map (herein called *bindif*) is created by exclusive-oring the bit-maps of current and previous frames (called *bin1* and *bin2*, respectively). Areas with a high density of 1's in the difference bit-map, *bindif* — which appear as bright regions when the bit-map is displayed — are representative of the extent of motion between the current and previous frames. The identification of motion blocks is initiated by looking at the *bindif* bit-map using 4×4 blocks to identify the number of bright (i.e., non-zero) pixels (bit value of 1 represented by intensity value of 255 for display simulation). The stationary regions do not result in bright pixels and are thus easily segmented from the motion areas where bright pixels are detected. Even a single bright pixel occurring in a block causes that block to be treated as a motion block. The subsequent search algorithm is implemented only for motion blocks. The role of the *bindif* bit-map is limited to this stage of discriminating between stationary and motion blocks. Each stationary block is identified with the motion vector $(0,0)$ as the horizontal and vertical co-ordinates and eliminated from the subsequent search to follow.

Once a motion block has been detected, each 4×4 block in the bit-map of the current frame (i.e., *bin2*) is compared with the 4×4 blocks within a search area of size $(M+2p, N+2q)$ in the previous frame (see Fig. 3.4), and the best match is found based on the simple criterion of minimizing the *distance function*, *distance* being defined, in the Hamming sense, as the number of non-zero pixels obtained by exclusive-oring the block of the current bit-map, *bin2*, with a block in the bit-map, *bin1*, in the search area of the previous frame surrounding the position of the block in the current frame. The scheme involves exclusive-or (XOR) bit operations only and some trivial additions to calculate the *distance function*, and is thus computationally the easiest and the quickest means for locating the best match. To estimate the interframe motion, conventional exhaustive search algorithms usually require $(2p+1)^2$ number of weights to be compared for each motion block in the current bit-map. In our study, $M \times N$ is a 4×4 block, and the search area in the previous frame extends up to 7 pixels around the position of the motion block in the current bit-map., *bin2*, i.e., $p=q=7$.

To simplify and cut down the number of search positions in the previous frame,

the conventional exhaustive search approach has been modified to the so-called Three-Stage Motion Search which is developed on lines similar to those described by numerous researchers in the past [14].

The three-stage motion search method is used in this study, because, it does provide a good compromise between the number of computations and the range effectiveness as compared to searching every possible location within the motion search range. The three-stage motion search method is briefly illustrated below:

Let $B = (M \times N)$ block of the present frame data.

and let $R = (M + HMV \times 2) \times (N + VMV \times 2)$ size motion search range data of the previous frame centered on block B ;

where $M = \#$ of rows in a block;

$N = \#$ of columns in a block;

$HMV =$ maximum horizontal displacement for a block;

$VMV =$ maximum vertical displacement for a block.

The optimum motion trajectory of each stage is given by the minimum value of distance function, $d(i, j)$, $i = 1, 2, 3$, where for the first stage, $d(1, j)$ is defined as

$$d(1, j) = \sum_{m=1}^M \sum_{n=1}^N f[b(m, n) - r(m + p, n + q)] \quad (27)$$

where b, r denote elements of matrices B and R respectively,

$j = 1, 2, 3, \dots, 9$

$p, q = -4, 0, +4$

$f =$ a positive measuring function, and here f is defined to be $f[\cdot] = [\text{sum of non-zero pixels on exclusive-or-ing current block with previous block}]$. This is the "weight" of the XOR difference 2D vector which represents the Hamming distance between the blocks.

The distance, $d(1, j)$ between blocks B and R is determined by shifting B 4 pixels around the center C_1 to each of 8 possible locations in the previous frame. The location which results in minimum $d(1, j)$ is referred to as the first-stage motion trajectory and

selected as the new center C_2 for the second stage motion search. The procedure for first-stage motion search is illustrated in Figs 3.4, and Fig. 3.5 illustrates the three stage search steps.

Figure 3.4 Illustration of First Stage (up to 4 pixels) of Three Stage Search algorithm.

Figure 3.5 The three stage search procedure.

Similarly, the second stage motion detection is accomplished by moving B around C_2 with a radius of 2 pixels ($p, q = -2, 0, +2$). The location at which the *distance*, $d(2, j)$, is minimum, is referred to as the second stage motion trajectory and chosen to be the center C_3 , for the third stage motion detection. At the third stage, block B is moved around C_3 with the radius of 1 pixel ($p, q = -1, 0, +1$), and the location where *distance* $d(3, j)$ is minimum, is determined to be the motion vector with vertical and horizontal position co-ordinates p & q , obtained after completion of third stage search.

Once the motion vector has been determined, the corresponding block B at that location may be further segmented into active and not so active subblocks for further bit-rate reduction as described by Lin and Kwatra [37]. This additional step, however, was not taken in our study, since it would lead only to a marginal improvement in bit-rate reduction but only at the cost of setting up further threshold criteria and increasing the number of computations.

The above algorithm may be referred to as the BTCMCP421 Search Algorithm since it involves creation of bit-maps, *bin1* and *bin2*, using BTC, followed by a motion detection search in three stages of 4 to 2 to 1 pixels.

To determine the optimum motion trajectory for each block, two major concerns are: the choice of block size and the range of motion search. Test results from 8×8 , 16×16 , and 32×32 block sizes, lead to apparent blockiness in the final reconstructed image which is inherent in the BTC algorithm when applied to large size blocks. In view of this limitation, block size of 4×4 is chosen in our study as the optimum.

The motion search range (*MSR*) expressed as

$$MSR = (M + HMV \times 2) \times (N + VMV \times 2)$$

shows that a larger search range will result in smaller motion search errors. However, it is impractical to do such a search since this requires an excessive number of computations. Ranges between 18×20 to 28×32 have been investigated by various workers [14] and the range of 18×18 is chosen for implementation in our study. We have also tested the results with the BTCMCP321 three stage search algorithm incorporating a search area of 16×16 . The results from the two alternatives were comparable. However, in view of the recent hardware implementation of 23×23 search range (using 8×8 blocks) on a VLSI motion estimation chip [117] — which is equivalent to 19×19 search range using 4×4 blocks — it was thought appropriate to stick to the BTCMCP421 algorithm.

3.4 Comparison of BTCMCP421 Weight Function with other Mismatch Functions

The evaluation of the weight function (also referred to as *distance* function or *distortion* function), $d(i,j)$ used in our BTCMCP421 search algorithm has much less computational complexity as compared to mismatch functions investigated in the past. The most popular mismatch functions described in literature [14,108,109] are the following.

- Normalized point correlation measure.
- Normalized mean square error (NMSE) and
- Mean absolute difference (MAD)

The normalized point correlation measure computes the statistical correlation between the current frame subblock and the subblock of the previous frame and is by far the most accurate registration measure. It is defined as

$$M(p, q) = \frac{\frac{1}{MN} \sum_{k=1}^M \sum_{l=1}^N [u(k, l)u_R(k + p, l + q)]}{\frac{1}{MN} \left[\sum_{k=1}^M \sum_{l=1}^N u^2(k, l)u^2(k + p, l + q) \right]^{\frac{1}{2}}} \quad (28)$$

where $u(k,l)$ denotes the location of the pixel in the current frame subblock, i.e., row k and column l , and $u_R(k+p,l+q)$ denotes the pixel in the subblock being searched in the previous frame at search position $(k+p, l+q)$.

The normalized mean square error (NMSE) measure computes the mean squared error (MSE) between the two subblocks, the current and the previous. This is a nonlinear measure and is supposed to track image covariance models well. It is defined as

$$M(p, q) = \frac{\sum_{k=1}^M \sum_{l=1}^N [u(k, l) - u_R(k + p, l + q)]^2}{\sum_{k=1}^M \sum_{l=1}^N [u^2(k, l)]} \quad (29)$$

The mean absolute difference (MAD) measure simply computes the mean of the absolute differences between the two subblocks in issue and is by far the simplest of the three measures. It is defined as

$$M(p, q) = \frac{1}{MN} \sum_{k=1}^M \sum_{l=1}^N |u(k, l) - u_R(k + p, l + q)| \quad (30)$$

The distortion measure $d(i, j)$, called *distance* (or *weight*), in our search algorithm, is simply the outcome of exclusive-or bit operations on bit-maps of current and previous frames. Evaluation of this *distance* function, obviously, is simpler than the simplest mismatch function, i.e., mean absolute difference (MAD), defined above. The introduction of the *distance* function as a distortion measure yields significant savings in the computational complexity of the search algorithm, as the only operations involved are exclusive-or bit operation on bits (0 & 1) followed by trivial adding of 1's. The computational complexity of the *distance* function as a distortion measure is thus drastically less than the NMSE or correlation measure and significantly less than the mean absolute difference (MAD); and, therefore, relatively simple arithmetic hardware can be used.

It may be worthwhile to note the following justification for equivalence between *distortion* measure and *distance* (or *weight*). If the binary data has values of +1 and 0, then the correlation between the blocks (Eq. 28) may be expressed as

$$M(p, q) = \frac{\text{No. of similar pixels} - \text{No. of Dissimilar Pixels}}{\text{Total Number of Pixels}} \quad (31)$$

The Hamming distance, D , is the number of dissimilar pixels; thus the correlation between binary blocks is given by

$$M(p, q) = \frac{S - D}{N} \quad (32)$$

Thus, for binary data, measuring the *distance* is equivalent to measuring the correlation.

Chapter 4

SIMULATION, RESULTS AND DISCUSSION

4.1 Image Sequences

The BTC based motion estimation and compensation algorithm, BTCMCP421, developed in the present work, was tested using three different sets of picture sequences. The first set, entitled *SELF1*, was acquired by taking three consecutive snapshots, *SELF1A*, *SELF1B* and *SELF1C*, of the author, in a sitting posture introducing intentional marginal displacement between the three frames by slight bodily movement. The time interval between one snapshot and the next was approximately 0.5 sec. Two of these three snapshots were selected as a pair comprising previous and current frames for implementation. The pair was chosen for this purpose on the basis of inspection of the three images on the video monitor and estimating that the motion between the frames falls within the range of 7 pixels (*Pic1* and *Pic2* in Fig. 4.1).

The second set, entitled '*TESTPAT*', was a pair of images created by generating a pattern of rectangular objects on the video screen (*Pic1* and *Pic2* in Fig. 4.5). The second image in this pair comprised the same rectangular objects as in the first image, but displaced with respect to their counterparts in the first image by precisely known

amounts, varying from 0 to 7 pixels, in eight different directions. The directions of displacement included horizontal-left, horizontal-right, vertical-up, vertical-down and four diagonal directions, left-up, left-down, right-up and right-down. This pair of images was essentially a set of bi-level images with pixel gray levels of 0 or 255 only. In view of the unique features as stated, this set may be called the '*TESTPAT*' set. Since the pixels in this set have only two levels of intensity, 0 or 255, the BTC algorithm phase results in bit-maps of previous and current frames, that are exactly identical to the original images. Thus the BTC phase can be bypassed while carrying out testing on this image sequence. The validity and effectiveness of the motion estimation and compensation algorithm can thus be verified without the introduction of carried-over errors, if any, due to the BTC phase.

The third set for testing was created in a fashion similar to the first set with the exception that in this case the author continued speaking during the period of snap shots, thus simulating a sort of teleconferencing environment, where there is focus with a zoom lens on one individual trying to explain his viewpoint to the others assembled in the videoconference (*Pic1* and *Pic2* in Fig. 4.6). Again, two of the three images in this sequence, entitled '*SELF2*', were chosen as a pair of previous *SELF2A* and current *SELF2B* frames to carry out simulation tests on this sequence.

4.2 Hardware Environment

The entire applications software development including the Block Truncation Coding algorithm, the Motion Estimation and Compensation algorithm, and other allied algorithms, required for simulation studies, was done in the 'C' language and their implementation carried out on a Tandy 3000HD (IBM PC/AT-compatible) microcomputer [115,116], equipped with an 80286 CPU, 80287 numeric co-processor and a Matrox PIP-512 frame grabber imaging board. The total memory available on the system consisted of 640 K of main memory, 2 MB of extended memory, and 40 MB of Winchester disk storage. The frame grabber-digitiser board had a resolution of 512×512 pixels with eight bits per pixel and provided the IBM PC/AT with the capability for various manipulations on video signals. The system was equipped with an ELMO B/W TV Camera, model CCD SE302, for image acquisition, and a Panasonic Display Monitor, model WV-5410.

The compiler used in the system was version 2.0 of *Turbo C* with its integrated environment, extensive library, and a powerful source-level debugger. The 'Large' C memory model option of the *Turbo C*, based on 32-bit pointers, was utilized in the development and implementation of applications software.

4.3 Applications Software

The typical applications software developed in 'C' language and utilized for simulations, tests and performance studies, may be listed briefly as follows.

- **I. Block Truncation Coding (BTC) algorithm using 4×4 blocks.** The algorithm consists of block truncation coding/decoding capability of video images. The most salient feature of this algorithm from the point of view of motion estimation and compensation is in the creation of 512×512 bit-maps from 8-bit gray level 512×512 images. In addition high- and low-mean, 128×128 subimages are created for ultimate

reconstruction of the final image. BTC algorithms using 8×8 , 16×16 and 32×32 block sizes were also written and implemented. However the BTC algorithm using 4×4 block size was chosen for final simulation tests.

- **II. Motion Estimation and Compensation algorithm using bit-maps and their exclusive-or difference bit-map as a basis for motion detection.** The final BTCMCP421 algorithm evolved as a consequence of preliminary investigations and development of algorithms for one-stage motion detection up to 3 pixels and up to 4 pixels, three-stage motion detection up to 6 pixels in 3-2-1 steps and finally the three-stage motion detection up to 7 pixels in 4-2-1 steps.
- **III. Conventional Exhaustive Search based Motion Estimation and Compensation algorithm.** This algorithm was developed along lines suggested by numerous workers in the past. This algorithm was deployed only for the sole purpose of comparison of the results obtained with the BTC based Motion Estimation and Compensation algorithm.
- **IV. Exclusive-Or Difference and Absolute Difference algorithms.** The Exclusive-or (XOR) Difference algorithm is utilized for creating the difference bit-map, *hindif*, from the bit-maps, *bin1* and *bin2*, of the previous and current image frames. The algorithm plays its role also in the performance evaluation of the predicted image. The Absolute Difference algorithm is used to create a difference gray level image from two 8-bit gray level images, the extent of relatively bright areas in the difference images indicating motion between the two images. This algorithm is made use of only as a crude test in the performance evaluation of the motion compensated reconstructed image.
- **V. White Pixel Count algorithm.** The number of non-zero pixels in the difference image is counted which gives information on the closeness of the reconstructed image

to the current image being predicted from the previous image frame.

- **VI. Normalized Mean Square Error (NMSE) Algorithm.** This algorithm tests the performance of our final BTCMCP421 algorithm using the mean square error (MSE) distortion function criteria.
- **VII. Low Pass Filter algorithm.** This algorithm is used to carry out preliminary smoothing of the original images before they are subjected to ETC based Motion Estimation and Compensation algorithm.
- **VIII. Run Length Code Statistics.** This algorithm measures the run-length statistics of the motion vector data and results in compression of Motion-Vector Data obtained in the Motion Estimation and Compensation algorithm.
- **IX. Display of Multiple Images.** Algorithms were written for displaying three or four 128×128 images simultaneously on the video screen. An interpolation algorithm was also written to transform a 512×512 bit-map into a 128×128 bit-map. This interpolated bit-map, together with 128×128 high-, low-, and high-low- mean sub-images could thus be displayed on the video monitor simultaneously.
- **X. Other allied algorithms.** Many other algorithms, such as Averaging over Multiple image frames in a sequence, Thresholding to get rid off background intensity values from relatively bright subject image, Run Length Coding/Decoding of bilevel images such as encountered in bit-maps created from BTC algorithms, Displaying B/W images in four-level color on the high resolution screen of the Color Monitor, model CM-1, which is an integrated part of Tandy Microcomputer, were also developed and implemented in preliminary investigations.

4.4 Simulation Tests & Experimental Results

4.4.1 First Set: Image Sequence 'SELF1'

Figures 4.1 and 4.2 show the results of simulations of the BTCMCP421 algorithm on the pair of images, *SELF1A* and *SELF1B*, in the image sequence entitled 'SELF1'. Figure 4.1(a), denoted by *Pic1* is the low-pass filtered previous frame, *SELF1A*, and Fig. 4.1(b), denoted by *Pic2*, is the low-pass filtered current frame, *SELF1B*. The purpose of the algorithm is to reproduce or reconstruct *Pic2* (current frame) from *Pic1* (previous frame), compensating for motion detected between the frames on the basis of operations on bit-maps generated by the BTC algorithm. The operations involved to obtain the reconstructed or predicted image and carry out performance evaluation tests on the predicted image are outlined as follows.

- (i) BTC encoding of *Pic1* results in its bit-map shown as *bin1* in Fig. 4.1(c).
- (ii) BTC encoding of *Pic2* results in its bit-map shown as *bin2* in Fig. 4.1(d).
- (iii) The exclusive-or difference of *bin1* and *bin2* gives the exclusive-or difference image, shown as *bindif* in Fig. 4.1(e), and highlights motion areas.
- (iv) The BTCMCP421 algorithm utilizes *bindif* to segment the entire region into Motion Blocks and Static Blocks. For each 4×4 block identified as a motion block, a three-stage search is initiated using the current motion block in *bin2* and finding its best match in *bin1* by comparing it with eight neighboring blocks in *bin1* in each stage.
- (v) The Motion Vector corresponding to each motion block, ('0,0' vector components given to static blocks), as estimated by the three-stage search in 4-2-1 steps above,

is applied to the previous bit-map, *binI*, to yield the 'predicted bit-map'. This is shown in Fig.4.1(i) as *PredB*.

- The BTC encoding steps (i) and (ii) yielded high- and low-mean subimages of previous and current frames. The high- and low-mean sub-images corresponding to the current frame are used to transform the predicted bit-map, *PredB*, into the predicted gray level image. This is shown in Fig. 4.1(j) as *PredR*. This is the final outcome of the algorithm which was intended to obtain a reconstructed current frame from the bit-map information of the previous frame and the high- and low-mean subimages of the current frame by utilizing motion vector data.

Figure 4.1 'SELF1' image sequence used in simulation of BTC based Motion Compensation algorithm. (a) Prefiltered previous frame. (b) Prefiltered current frame. (c) Bit-map of previous frame. (d) Bit-map of current frame. (e) Exclusive—or difference image created from (c) and (d). (f) Exclusive-or difference image created from predicted bit-map and bit-map of current frame.

Figure 4.1 (Continued.) (g) Absolute difference of gray level current and previous frames. (h) Absolute difference of predicted image and current frame. (i) Bit-map predicted from bit-map of previous frame. (j) Predicted image from previous frame. (k) Direct BTC coding of current frame. (l) Predicted image using Exhaustive search algorithm.

Figure 4.2 Comparison of current image, *Pic2* with predicted images obtained from Motion Compensation and direct BTC algorithms in '*SELF1*, and *SELF2*' image sequences..

The pictures shown in Figs. 4.2 (b) and (c) show the comparison of the reconstructed current frames — created as a result of BTC based Motion Compensation and direct BTC respectively, on the image sequence '*SELF1*' — with the original prefiltered current frame, *Pic2* shown in Fig. 4.2 (a). Similar comparison is made in Figs. 4.2 (d), (e) and (f) for the image sequence '*SELF2*', simulations on which are described in a later section (Sec. 4.4.3).

The pictures shown in Figs. 4.1 (f), (g), (h) and (k), are generated from the point of view of performance evaluation on the predicted image and involve the following steps.

- The exclusive-or difference of the predicted bit-map, *PredB* of Fig. 4.1(i), with the bit-map of current frame, *bin2* of Fig. 4.1(d), results in the formation of the exclusive-or difference image shown in Fig. 4.1(f) as *errorB*. This is the error in prediction of the bit-map of the current frame. This gives an estimate of the closeness of the bit-map predicted from *bin1* to the bit-map *bin2*. The number of non-zero (bright with intensity value 255) pixels in the exclusive-or difference image is representative of the extent of motion compensation achieved. The lower this number, the better is the prediction. This number is compared with the number of whites in the *bindif* image of Fig. 4.1(c).
- The absolute-difference of gray levels between the predicted image, *PredR* of Fig. 4.1(j) and the true image, *Pic2* of Fig. 4.1(b), gives a rough estimate of the prediction error in terms of the extent of relatively bright pixels left over in this absolute-difference image. In case of perfect matching, there should be complete cancellation and entire image would be black, i.e., filled with black pixels only. This difference image, shown in Fig. 4.1(h) as *errR2*, is compared with the absolute-difference image *errR1* shown in Fig. 4.1(g), of *Pic1* and *Pic2* gray level absolute differences.
- The image reconstructed from the BTCMCP421 algorithm is compared with the image obtained by the BTC encoding/decoding algorithm alone. For instance, the

image 'BTC2' shown in Fig. 4.1(k), created using the BTC encoding/decoding algorithm, may be compared with the motion compensated predicted image *PredR* shown in Fig. 4.1(j).

- The reconstructed image, shown in Fig. 4.1(j) as *PredR*, is compared with the actual image of interest, shown in Fig. 4.1(b) as *Pic2*. The performance evaluation criteria used is subjective psycho-visual assessment. The prediction is either excellent, or good or unsatisfactory and annoying.
- One commonly used quantitative or objective measure for performance evaluation of compression algorithms is the Normalized Mean Square Error (NMSE). The Normalized Mean Square Error Evaluation algorithm is utilized to obtain the numerical value of the NMSE between the predicted image, *PredR* of Fig. 4.1(j), and true image, *Pic2* of Fig. 4.1(b). This value is compared with the NMSE value obtained between *Pic1* and *Pic2*, which signifies the extent of motion between previous and current frames. Total Squared Error (TSE) as well as Mean Square Error (MSE) are also evaluated during the execution of the NMSE parameter. However, these can be used for comparing the results from those systems which have the same dynamic range and when processing the same image, since MSE and TSE are absolute quantities rather than relative quantities. The definition of these objective measures are given below.

$$NMSE = \frac{\sum (x - \hat{x})^2}{\sum x^2} = \frac{TotalErrorEnergy}{TotalSignalEnergy} \quad (33)$$

or

$$NMSE(\%) = \frac{\sum (x - \hat{x})^2}{\sum x^2} \times 100 \quad (34)$$

where x is the original pixel intensity value, $\hat{x}$ is the reconstructed value of x , and summation is carried over all the pixels in the image.

$$MSE = \frac{\sum (x - \hat{x})^2}{N \times M} \quad (35)$$

where $N \times M$ is total number of pixels processed in the image.

$$TSE = \sum (x - \hat{x})^2 = TotalErrorEnergy \quad (36)$$

where x is the original pixel value and $\hat{x}$ is the reconstructed value of x , summation being over the total number of pixels in the image.

The experimental results obtained as a result of the steps followed above are shown in Table 4.1 and Table 4.2.

Table 4.1 NMSE performance of simulation tests on 'SELF1' image sequence.

Parameters for NMSE evaluation	Pic2 vs Pic1	Pic2 vs BTC2	Pic2 vs PredR
Mean Gray level	154 154	154 154	154 154
Total Square Error(TSE)	6.38×10^6	4.27×10^6	6.01×10^6
Mean Square Error	259.51	17.39	24.47
Ave Energy/pixel	25080.42	24834.83	24834.83
NMSE(%)	1.035	0.070	0.098

Table 4.2 White pixels count in gray level, and bit-map image differences in 'SELF1' simulation.

	Pic2 - Pic1 (abs. diff.)	PredR-Pic2 (abs. diff.)	bin2-bin1 (xor diff.) bindif	PredB-bin2 (xor diff.)	PredR-btc2 (abs. diff.)
Number of White pixels	49646 (20%)	15696 (6%)	32307 (13%)	6610 (3%)	7075 (3%)
Number of Black pixels	196114	230064	213453	239150	238685

In addition to the above steps taken for obtaining the motion compensated predicted image, display algorithms were used to demonstrate three or four pictures on the TV Monitor screen simultaneously. These are shown in Figs. 4.3(a), (b), and (c), and in Fig. 4.4(a); all these being relevant to the original images *Pic1* and *Pic2* of image sequence *SELF1A*. The first two images in Fig. 4.3(a) and Fig. 4.3(b) are the high- and low-mean subimages created as a result of BTC algorithm. The third image in these figures is the difference between high- and low-mean subimages. The fourth image in Fig. 4.3(b) is the interpolated image from bit-map, *bin1*, which was created by using the Decimation algorithm for converting 512×512 bit-map into a 128×128 bit-map. The 512×512 bit-map is also shown in Fig. 4.4(a). Images in Fig. 4.4(a) are another example of depicting multiple images simultaneously on the TV Monitor screen. We have also shown the histogram of gray level values in a typical 512×512 gray level picture in Fig. 4.4(b).

Figure 4.3 Illustration of *Display* utility algorithms to display multiple images simultaneously. (a) High-mean, Low-mean and high-low difference 128×128 images displayed simultaneously. (b) High-mean, Low-mean, high-low difference 128×128 images along with 512×512 Bit-map of previous frame interpolated to size 128×128 . (c) Histograms of gray levels of High-mean and high-low difference images.

Figure 4.4 (a) Another illustration of 'Display'utility algorithm. High-mean, Low-mean, high-low difference and interpolated bit-map of previous frame shown superimposed on 512x512 bit-map of previous frame. (b) Histogram of gray levels of previous frame.

Finally, the Run Length Statistics algorithm was utilized to obtain the entropy of run lengths in the Motion Vector data generated by the Motion Estimation and Compensation algorithm. All components of motion vectors with non-zero values were treated as 1's in calculating the run lengths. The results are shown in Table 4.3. The information

Table 4.3 Run Length Statistics of Motion Vector file in '*SELF1*' simulation.

Number of NON-ZERO run lengths	3128
Number of ZERO run lengths	3134
Entropy of NON-ZERO value data	2.728
Entropy of ZERO value data	2.335
Average Entropy	2.531

obtained from run length statistics, i.e., entropy of ZERO and NON-ZERO run lengths, is utilized in evaluating the overall bit rate required for transmission of coded image as will be shown later in this chapter (sub-section 4.7.2).

4.4.2 Second Set: Image Sequence 'TESTPAT'

The second data set, entitled "*TESTPAT*", involved a pair of images, *TESTPATA*, and *TESTPATB*, in the artificially generated test pattern sequence in which the relative motion of different structures between the two images is known precisely. The relative motion varied from 0 to 7 pixels in eight different symmetrical directions. The previous and current frames are again designated as *Pic1* and *Pic2*, and shown in Figs 4.5(a) and (b),

respectively. These are the bi-level pictures with pixel values of either 0 or 255 (255 treated as '1' during execution of algorithm).

The operations involved in encoding, using the '*TESTPAT*' set, are exactly identical to those listed for the first set, entitled '*SELF1*'. However, it may be noted that the bit-maps, *bin1*, and *bin2*, generated in this case are nothing but the original pictures *Pic1* and *Pic2* in view of the bi-level pixel intensity values of *Pic1* and *Pic2*. The BTC phase, which generates the bit-maps, may therefore be completely by-passed from the point of view of the motion search. The exclusive-or difference between *Pic1* and *Pic2* gives the same difference image, *bindif*, as would have been obtained from *bin1* and *bin2*. In short, to create the *bindif* image, one may treat *Pic1* and *Pic2* as *bin1* and *bin2* images for further processing.

Since the BTC algorithm can be bypassed in this simulation, the final predicted image obtained in the simulation test will indicate the precision and efficiency of the Motion Estimation and Compensation algorithm without propagation of errors, if any, resulting from the BTC phase.

One additional point worth noting in this ideal set is that there is no requirement for the prefiltering stage for smoothing the images prior to the BTC based motion search. Each pixel in the two images of the pair are given specific values of 0 or 255 and no random or systematic noise is present in the originals.

The results of performance evaluation on '*TESTPAT*' image sequence are shown in Table 4.4, Table 4.5, and Table 4.6, and discussed in section 4.9.

Resolution Test Chart Labels:

- 1.0
- 1.1
- 1.25
- 1.4
- 1.6
- 1.8
- 2.0
- 2.2
- 2.5
- 2.8
- 3.2
- 3.6
- 4.0
- 4.5
- 5.0
- 5.6
- 6.3
- 7.1
- 8.0
- 9.0
- 10

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

Figure 4.5 "TESTPAT" image sequence simulation. (a) Prefiltered previous frame. (b) Prefiltered current frame. (c) Bit-map (a). (d) Bit-map of (b). (e) XOR difference image created from (c) and (d). (f) XOR difference image created from predicted binary image and bit-map of current frame.

(g)

(h)

(i)

(j)

Figure 4.5 (Continued) (g) Absolute difference of gray level current and previous frames. (h) Absolute difference of predicted image and current frame. (i) Direct BTC coding of current frame. (j) Predicted image.

Table 4.4 NMSE performance of simulation tests on 'TESTPAT' image sequence.

Parameters for NMSE evaluation	Pic2 vs Pic1	PredR vs Pic2
Mean Gray level	74 74	74 72
Total Square Error(TSE)	1.18×10^9	1.14×10^8
Mean Square Error	4804.91	465.15
Ave Energy/pixel	19058.23	19058.23
NMSE(%)	25.212	2.441

Table 4.5 White pixels count in gray level, and bit-map image differences in 'TESTPAT' simulation.

	Pic2 - Pic1 (abs. diff.)	PredR-Pic2 (abs. diff.)	bin2-bin1 (xor diff.) bindif	PredB-bin2 (xor diff.)
Number of White pixels	18160 (7.4%)	1758 (0.7%)	18160 (7.4%)	1366 (0.6%)
Number of Black pixels	227600	244002	227600	244394

Table 4.6 Run Length Statistics of Motion Vector data in 'TESTPAT' simulation.

Number of NON-ZERO run lengths	698
Number of ZERO run lengths	706
Entropy of NON-ZERO value data	2.106
Entropy of ZERO value data	3.339
Average Entropy	2.726

4.4.3 Third Set: Image Sequence 'SELF2'

The operations involved in this test, using *SELF2A* and *SELF2B* as previous and current frames of the image sequence 'SELF2', are identical to those described in the first set pertaining to 'SELF1' sequence. The results of simulation are shown in Table 4.7, Table 4.8, and Table 4.9 and discussed in Sec. 4.9.

Table 4.7 NMSE performance of simulation test on 'SELF2' image sequence.

Parameters for NMSE evaluation	Pic2 vs Pic1	Pic2 vs BTC2	PredR vs Pic2
Mean Gray level	110 118	118 118	118 118
Total Square Error(TSE)	5.13×10^6	1.55×10^6	2.28×10^6
Mean Square Error	20.88	6.31	9.26
Ave Energy/pixel	14779.75	14729.75	14730.79
NMSE(%)	0.141	0.043	0.063

Table 4.8 White pixels count in gray level, and bit-map image differences in 'SELF2' simulation.

	Pic2 - Pic1 (abs. diff.)	PredR-Pic2 (abs. diff.)	bin2-bin1 (xor diff.) bindif	PredB-bin2 (xor diff.)	PredR-btc2 (abs. diff.)
Number of White pixels	10837 (4.4%)	5459 (2.2%)	9608 (5.8%)	5117 (1.5%)	3735 (0.2%)
Number of Black pixels	234923	240301	236152	241983	242025

Table 4.9 Run length statistics of Motion Vector data in '*SELF2*' simulation.

Number of NON-ZERO run lengths	2465
Number of ZERO run lengths	2471
Entropy of NON-ZERO value data	2.301
Entropy of ZERO value data	3.352
Average Entropy	2.823

4.4.4 Third Set without pre-filtering: '*SELF2*' image sequence

The image sequence '*SELF2*' of the third set was used again for additional simulation tests, but in this procedure, the initial stage of pre-filtering the original images, *SELF2A* and *SELF2B*, was bypassed. The pair of previous and current pictures have thus not been smoothed to reduce noise, and these pictures directly undergo BTC operations for bit-map creation, followed by subsequent Motion Estimation and Compensation operations. Figures 4.7 (a), (b), (c), and (d) show the display of results of simulation. Other results of performance are shown in Tables 4.10, 4.11 and 4.12.

Figure 4.6 'SELF2' image sequence used in simulation of BTC based Motion Compensation algorithm. (a) Prefiltered previous frame. (b) Prefiltered current frame. (c) Bit-map of previous frame. (d) Bit-map of current frame. (e) Exclusive-or difference image created from (c) and (d). (f) Exclusive-or difference image created from predicted binary image and bit-map of current frame.

Figure 4.6 (Continued) (g) Absolute difference of gray level current and previous frames. (h) Absolute difference of predicted image and current frame. (i) Bit-map predicted from bit-map of previous frame. (j) Predicted image from previous frame.

Figure 4.7 'SELF2' image sequence without prefiltering. (a) Bit-map of previous frame. (b) Bit-map of current frame. (c) XOR difference image created from (a) and (b). (d) Predicted image.

Table 4.10 NMSE performance of simulation test on 'SELF2' image sequence without pre-filtering.

Parameters for NMSE evaluation	Pic2 vs Pic1	Pic2 vs BTC2	PredR vs Pic2
Mean Gray level	119 119	118 118	118 119
Total Square Error(TSE)	6.85×10^6	2.82×10^6	3.72×10^6
Mean Square Error	27.87	11.48	15.12
Ave Energy/pixel	14905.04	14838.66	14838.66
NMSE(%)	0.187	0.077	0.102

Table 4.11 White pixels count in gray level, and bit-maps image differences in 'SELF2' simulation without pre-filtering.

	Pic2 - Pic1 (abs. diff.)	PredR-Pic2 (abs. diff.)	bin2-bin1 (xor diff.) bindif	PredB-bin2 (xor diff.)	PredR-btc2 (abs. diff.)
Number of White pixels	13740 (5.6%)	8409 (3.4%)	18887 (7.7%)	8230 (3.3%)	8175 (3.3%)
Number of Black pixels	232020	237351	226873	237530	237585

Table 4.12 Run length statistics of Motion Vector data in '*SELF2*' simulation without pre-filtering.

Number of NON-ZERO run lengths	5396
Number of ZERO run lengths	5402
Entropy of NON-ZERO value data	2.559
Entropy of ZERO value data	2.647
Average Entropy	2.603

4.5 Exhaustive Search on '*SELF2*' Image Sequence

For comparative evaluation purposes, an Exhaustive Search algorithm was developed and this algorithm was used to carry out motion estimation and compensation on the first, image sequence, '*SELF1*'. This algorithm is exhaustive in the sense that for each current block, 225 blocks in the previous frame at all displacement positions, resulting from combinations of -7 to $+7$ pixels in the horizontal direction and -7 to $+7$ pixels in the vertical direction, are searched to find the best match. The optimum match is found by minimizing the sum of the absolute differences of individual pixel gray level values in each block comparison.

The results of exhaustive search and resulting compensation are shown in Tables 4.13 and 4.14. The time taken for execution of the exhaustive search algorithm is almost ten times that taken by our limited search algorithm, BTCMCP421. This is to be expected, since the limited search algorithm searches at most 24 blocks to find the optimum match.

Because of the unduly high time complexity of the Exhaustive Search algorithm, it was not considered for simulation on other sets of image sequences.

Figure 4.1(l) shows the predicted image, *PredEx*, reconstructed from the previous image *SELFIA* using exhaustive search.

Table 4.13 NMSE performance of simulation test on
'SELF1' image sequence using Conventional Exhaustive Search.

Parameters for NMSE evaluation	Pic2 vs Pic1	Pic2 vs BTC2	PredR vs Pic2
Mean Gray level	154 154	154 154	154 154
Total Square Error(TSE)	6.38×10^6	3.99×10^6	4.58×10^6
Mean Square Error	259.51	16.26	18.65
Ave Energy/pixel	25080.42	24830.59	24865.60
NMSE(%)	1.035	0.065	0.075

Table 4.14 White pixels count in gray level image differences
in '*SELF1*' simulation using Conventional Exhaustive Search.

	$ \text{Pic2} - \text{Pic1} $	$ \text{PredR} - \text{Pic2} $
Number of White pixels	49646 (20.2%)	12926 (5.3%)
Number of Black pixels	196114	232834

It may be noted that the exhaustive search approach does not use bit-maps for estimation and compensation but operates directly on gray level previous and current frames.

4.6 Histogram of Motion Vector Data

Motion Vector data obtained in simulation tests carried out on different image sequences was collected and Figs. 4.8, 4.9, 4.10 and 4.11 show histograms of motion vectors which are detected for the '*SELF1*', '*SELF2*', '*TESTPAT*' and '*SELF2 (without prefiltering)*' image sequences respectively. The x- and y-axis of the grid of the 3-dimensional plots represents the range of displacements from -7 to 0 to $+7$ pixels along each axis and the heights represent the number of motion blocks for each (x,y) displacement position on the grid. The total number of Motion Blocks observed in '*SELF1*', '*TESTPAT*', '*SELF2*' and '*SELF2*' without prefiltering simulations were 6187, 1450, 3934 and 9142 respectively. The distribution of these Motion Block numbers is depicted by the vertical heights in the 3-dimensional isometric plots.

It may be worth pointing out that the above 3-D plots are obtained from the ASCII files containing only the displacement vectors of each motion block. On the other hand, Run Length Coding of motion vector data for transmission is carried out on binary files containing vector data for Motion Blocks and Static Blocks in the same file, which will, thus, be composed of a significantly large number of $(0,0)$ vector positions.

SELF1:x- & y-displacement range -7 to 0 to +7
 Figure 4.8 Vector Histogram for 'SELF1' image sequence.

SELF2:x- & y-displacement range -7 to 0 to +7
 Figure 4.9 Vector Histogram for 'SELF2' image sequence.

TESTPAT:x- & y-displacement range -7 to 0 to +7
 Figure 4.10 Vector Histogram for 'TESTPAT' image sequence.

SELF2 (No Filtering):x- & y-displacement range -7 to 0 to +7
 Figure 4.11 Vector Histogram for 'SELF2' image sequence without prefiltering.

4.7 Image Data Compression Ratio Evaluation

4.7.1 Single Frame 2-D BTC, No DCT Coding of High- & Low-mean sub-images, and No Motion Compensation

In a normal 2-D BTC coding scheme, the reconstruction of a 8-bit gray level image, using 4×4 pixel blocks, requires transmission of a 512×512 bit-map and 8-bit values of high- and low-mean for each 4×4 block, there being 128×128 blocks in the gray level image. Using 8 bits for the high- and low-mean values, one requires

$$2 \times (128 \times 128) \times 8 = 262144 \quad (37)$$

bits. The bit-map requires

$$512 \times 512 = 262144 \quad (38)$$

bits. Thus

$$262144 + 262144 = 524288 \quad (39)$$

bits are needed to transmit the current frame. In terms of bit rate per pixel, one obtains

$$\frac{524288}{512 \times 512} = 2 \text{ bits/pixel.} \quad (40)$$

The image data compression ratio with respect to the 8-bit gray level per pixel of the current frame is, therefore, given by

$$4 : 1. \quad (41)$$

4.7.2 Interframe 2-D BTC, with Motion Compensation, and No DCT Coding of high- and Low-mean sub-images

This is the case that directly concerns our Motion Estimation and Compensation algorithm and for simplicity we ignore the DCT Coding of the high- and low-mean sub-images. As in the above case, the transmission of the high- and low-mean subimages of current frame requires 262144 bits. However, instead of transmitting the bit-map of current frame, one transmits only the Motion Vector data. The entropy of this data for ZERO values as calculated, for instance, in the first set relating to '*SELF1*' image sequence, is 2.335 and there are 3134 ZERO run lengths (see Table 4.3). Further there are 3128 NON-ZERO run lengths and 4 bits are needed for each non-zero motion vector value which may range from -7 to $+7$. One, therefore, requires at least

$$3128 \times 4 + 3134 \times 2.335 = 19830 \quad (42)$$

bits to represent the bit-map. Therefore,

$$262144 + 19830 = 281974 \quad (43)$$

bits are needed to reconstruct the current frame. The bit rate is, therefore, given by

$$\frac{281974}{512 \times 512} = 1.08 \text{ bits/pixel.} \quad (44)$$

The compression ratio is, therefore, 7.4 : 1.

In the second set, '*TESTPAT*', the entropy for ZERO run lengths, as seen from Table 4.6, is 3.339 and the bit rate is, therefore given by

$$\frac{(698 \times 4 + 706 \times 3.339) + 262144}{512 \times 512} = 1.02 \text{ bits/pixel.} \quad (45)$$

The compression ratio is, therefore, 7.8:1 approximately.

Similarly, in third set, '*SELF2*' image sequence, the entropy for ZERO run lengths being 3.352, the bit rate is given by

$$\frac{(2465 \times 4 + 2471 \times 3.352) + 262144}{512 \times 512} = 1.07 \text{ bits/pixel.} \quad (46)$$

The compression ratio is, therefore,

$$7.5 : 1. \quad (47)$$

4.7.3 Single Frame 2-D BTC, DCT Coding of High- and Low-mean sub-images, No Motion Compensation

The DCT Coding phase, as developed in Ph.D. thesis of Wu [2], indicates that about 2.5 bits are needed to code each high- and low-mean value. There being 128×128 blocks, and for each block one high- and one low-mean value, one requires

$$128 \times 128 \times 2.5 \times 2 = 81920 \quad (48)$$

bits to code the high- and low-mean sub-images. The 512×512 bit-map of the current frame needs 262144 bits. Thus the bit rate is given by

$$\frac{262144 + 81920}{512 \times 512} = 1.31 \text{ bits/pixel.} \quad (49)$$

The compression ratio is, therefore,

$$6 : 1, \quad (50)$$

when DCT coding the high- and low-mean sub-images only.

4.7.4 Interframe 2-D BTC with Motion Compensation and DCT Coding of High- and Low-mean sub-images

This is the situation that pertains to the entire development of Motion Estimation and Compensation algorithm based on BTC as described in this thesis. Taking into account

the entropy values for the Motion Vector data and DCT coding of high- and low-mean sub-images, the bit rate for each of the sets used in our simulation studies, can be easily calculated. In the first set, involving '*SELF1*' image sequence, for instance, one needs

$$(3128 \times 4 + 3134 \times 2.335) = 19830 \quad (51)$$

bits for Motion Vector data, and

$$128 \times 128 \times 2.5 \times 2 = 81920 \quad (52)$$

bits for high- and low-mean sub-images. Thus the bit rate is given by

$$\frac{19830 + 81920}{512 \times 512} = 0.39 \text{ bits/pixel.} \quad (53)$$

The compression ratio is, therefore,

$$20.6 : 1. \quad (54)$$

The bit rate for second and third sets can be similarly calculated and the results are

$$0.33 \text{ bits/pixel} \quad (55)$$

in second set, '*TESTPAT*' image sequence, and

$$0.38 \text{ bits/pixel} \quad (56)$$

in the third set, '*SELF2*' image sequence. The corresponding compression ratios are

$$24 : 1 \text{ and } 21 : 1 \quad (57)$$

respectively.

4.8 Summary of Simulation Results

Table 4.15 summarizes salient results of simulation tests carried out on different sets of image sequences described in this chapter.

Table 4.15 Summary of simulation tests on different image sequences.

IMAGE SEQUENCE	NMSE Pic2 vs Pic1	NMSE PredR vs Pic2	White pixel count bin2 - bin1 bindif	White pixel count PredB - bin2	Compression Ratio & bit-rate (bits/pixel)
SELF1	1.035	0.098	32307 (13%)	6610 (3%)	20.6 : 1 (0.39)
TESTPAT	25.212	2.441	18160 (7.4%)	1366 (0.6%)	24.0 : 1 (0.33)
SELF2	0.141	0.063	9608 (5.8%)	3777 (1.5%)	21.0 : 1 (0.38)
SELF2 without prefiltering	0.187	0.102	18887 (7.7%)	8230 (3.3%)	-
SELF1 Exhaustive search	1.035	0.075	-	-	-

4.9 Discussion

4.9.1 NMSE Performance

The NMSE performance for 512×512 'SELF1', 'TESTPAT', and 'SELF2' image sequences are summarized in columns 2 and 3 of Table 4.15. For 'SELF1' image sequence, the reduction in NMSE is by a factor of more than 10, whereas for 'SELF2' image sequence, the reduction factor is only about 3. This variation from one sequence to the other may be attributed to the observation that in 'SELF2' image sequence, the extent of displacement regions is much smaller as compared to 'SELF1' image sequence, as can be seen from their respective motion vector histograms shown in Figs. 4.8 and 4.9. In the case of 'SELF2' image sequence without prefiltering, the reduction factor is even less than 2. This is owing to the fact that a very large proportion of the static region in the image sequence manifests itself as motion area as evaluated from their bit-maps.

The results of NMSE performance for 'SELF1' image sequence have been compared both using our limited search motion compensation algorithm as well as conventional exhaustive search algorithm. The exhaustive search algorithm results only in a marginal improvement in NMSE reduction by a factor of 14 as against slightly more than 10 with our limited search. This gain in performance, however, does not warranted using exhaustive search since the time for execution of exhaustive search is almost 10 times that for limited search.

4.9.2 Psycho-visual criteria for Error Evaluation

Comparison of columns 4 and 5 in Table 4.15 gives an estimate of the extent of motion compensation achieved by the BTC based motion compensation algorithm. As can be seen, the white pixel content, signifying the degree of relative motion between frames,

is reduced by a factor of 4 in the '*SELF1*' and '*SELF2*' image sequences simulation. On the other hand, for the '*TESTPAT*' simulation, reduction is by a factor of more than 12. In the case of the '*SELF2*' image sequence without prefiltering, the reduction factor is just about 2. This observation stresses the need for prefiltering before applying motion compensation.

4.9.3 Image Data Compression

The last column in Table 4.15 lists the amount of data compression achieved with BTC based motion compensation algorithm used on different image sequences. As can be seen, the range of compression ratio is from 20 to 25 which is about 5 times the compression from direct BTC algorithm.

It may be noted that the smaller the extent of motion region, the greater the compression ratio. For instance, in the '*TESTPAT*' image sequence, there were only 1450 motion blocks detected, and we have achieved a compression ratio of 24:1. On the other hand, in the '*SELF1*' image sequence simulation, 6187 motion blocks were observed, and this yielded a compression ratio of only 21:1.

Compared with the compression ratio of *4:1* achievable with the direct BTC algorithm, the ratio of about *20:1*, from the BTC based motion compensation algorithm, implies that one obtains a lowering of bit-rate transmission from 2 bits/pixel from direct BTC to about *0.4 bits/pixel* from BTC based motion compensation algorithm. This is a significant improvement, indeed.

The above bit rates only refer to the current frame in an image sequence. The previous frame, which achieves a compression ratio of 6:1 (see sec 4.7.3) using BTC combined with DCT of high- and low-mean subimages, still needs 1.33 bits/pixel. By carrying out run length coding of bit-map of the previous frame in an image sequence, it was observed that the bit-map could be compressed to about 0.6 bits/pixel instead of 1 bit/pixel (see section A.16 of Appendix A for run length coding algorithm on bit-maps). **Taking this**

into account, the bit rate for transmitting the previous frame is then reduced from *1.33 bits/pixel* to about *0.91 bits/pixel*, i.e., a compression ratio of about *8.8:1*. Thus the overall compression ratio for the previous and current frames is about *12:1* which corresponds to a bit rate of *0.67 bits/pixel* for each frame.

4.9.4 Postfiltering

On careful examination of a reconstructed image on a high resolution TV monitor, one notices a small amount of blockiness persisting all over the displayed picture. This appearance is expected as the entire scheme of motion estimation and compensation is based on BTC using 4×4 blocks. The blockiness becomes more pronounced if larger size blocks were used. To remove this apparent blockiness in reconstructed image, one could apply a 2×2 low pass (LP) filter to the reconstructed image. This postfiltering phase is exactly similar to the prefiltering phase that is used in smoothing raw previous and current frames in the image sequences under study.

4.9.5 Histograms of High-, Low-mean and High-Low difference sub-images

From histograms of the pixel gray level values for High-mean (or Low-mean) and High-Low difference sub-images shown in Fig. 4.3(c), it is noted that gray level values are distributed in a broad range of about 75–255 in High-mean (or Low-mean) sub-images, whereas their distribution is narrowed down to 0–48, peaked about zero gray level, in the High-Low difference image. The High-Low difference image, thus, has a much lower entropy than the High-mean (or Low-mean) sub-image. The High-Low difference image can thus be coded more efficiently, requiring fewer bits, than the High-mean (or Low-mean) sub-image. Instead of coding High-mean and Low-mean sub-images, it is, therefore, preferable, to code High-mean (or Low-mean) sub-image and High-Low difference sub-image. Obviously the Low-mean (or High-mean) sub-image can always be recovered by the subtracting High-Low difference sub-image from the High-mean (or Low-mean) sub-image. This approach was adopted in adaptive DCT coding of high- and low-mean sub-images.

In the present situation, since we are dealing with a sequence of images, we can make use of temporal differences in high- and low-mean sub-images to achieve further reduction in bit-rate. We can create difference high-mean sub-image by taking interframe difference between high-mean sub-images of current and previous frames and similarly for difference low-mean sub-image. The distribution of gray levels in these temporal difference sub-images is obviously expected to be very narrow, peaked about zero gray level. There is thus a significant reduction in the entropy of these difference sub-images (see Figure 4.12) and adaptive DCT coding will result in further reduction in bit-rate.

4.9.6 Run Length Coding of Bit-maps

A 512×512 bit-map requires 1 bit/pixel for transmission. To obtain the predicted

Figure 4.12 Entropy (H in bits/pixel) measurements on 128X128 gray-level sub-images and their difference sub-images. The results are compared with entropy of original 512X512 gray-level images of 'SELF' image sequence.

bit-map, *predB*, for the current frame from the bit-map, *bin1*, of the previous frame, one, therefore, needs 32768 bytes (into which bits are compressed) for transmission of the *bin1* bit-map. It was considered worthwhile to explore the possibility of achieving reduction in this bit rate by employing Run Length Coding on bit-maps. For instance, in the case of prefiltered '*SELF1*' image sequence, run length coding of *bin1* bit-map required only 21011 bytes as measured by the Run Length Coding algorithm performed on the bit-map binary file. The algorithm is given in appendix A as BMPRLCOD.C/BMPRLDEC.C source code. Thus a saving in bit rate from 1 bit/pixel to about 0.6 bits/pixel is achieved by run length encoding the bit-map. It may, however, be pointed out that run length coding of a noisy bit-map, such as the one created in '*SELF1*' image sequence without prefiltering, does not lead to this reduction in bit rate. This is to be expected as is evident from the *bindif* bit-map displayed in Fig. 4.7(c) showing a very large number of white pixels even in the static regions in the image sequence.

4.9.7 Worst Case Scenario

The worst case scenario of an image sequence occurs when the relative displacement between the current and previous frames is such that, in the motion search algorithm, every 4×4 block in the current frame is found to be a motion block. In this situation, we obviously encounter 16384 motion blocks and there is a motion vector with components (*x*,*y*) associated with each block. Run length encoding of the motion vector data is of no practical utility for such a data. One, therefore, has to transmit 32768 displacement values, each requiring 4 bits, to represent displacement between -7 to $+7$ pixels. Thus 131072 bits are needed for the motion vector data. Taking into account the 81920 bits required for high- and low-mean subimages of the current frame, one, therefore, needs 212992 bits to reconstruct the predicted image. The bit rate is then given by

$$\frac{212992}{512 \times 512} = .8 \text{ bits/pixel.} \quad (58)$$

The corresponding compression ratio is therefore 10:1.

It may be noted that even in this worst case situation, motion compensation gives more data compression than may be attained without motion compensation. **It, therefore, goes without saying that motion compensation is the best way of coding.**

4.9.8 Compression & High/Low Mean Sub-images

In all simulation tests carried out in this work, we have obtained the predicted image from high- and low-mean subimages of **the current frame** and **motion vector data** from the bit-map of the previous frame. Transmission of prediction error data was found unnecessary in view of the good quality reconstruction obtained without it. Compression ratio of the order of **20:1** has been achieved without sacrificing image quality.

To be most optimistic, we explored the possibility of achieving the highest possible compression by reconstructing the image utilizing high- and low-mean sub-images of the previous frame (or even only the first frame in a sequence of frames). Then, the reconstruction of the current image would, ideally, involve only the transmission of motion vector data. In a typical image sequence, say, '*SELF1*', only 19830 bits are required for coding the motion vector data. Thus the bit rate could be as low as

$$\frac{19830}{512 \times 512} = 0.08 \text{ bits/pixel.} \quad (59)$$

The compression ratio is, thus, **100:1**. Unfortunately such an ideal situation is far from practical reality. On exploring this possibility it was observed that the predicted image was very annoying to the eye due to the appearance of undesirable sharp square patches at various positions in the image. These were apparently due to significant changes in the high- and low-mean sub-images which are not communicated since the motion vector data only describes changes in the bit-map. The predicted image being unacceptable to the Human Visual System, no efforts were made to carry out any objective evaluation, such as NMSE measurement. The distortion in the predicted image can possibly be overcome by transmitting information describing changes in high- and low-mean sub-images in motion blocks.

Chapter 5

CONCLUSIONS

Although motion estimation and compensation technology is still in a developing stage, and efforts are being made to bring it into actual application giving precise and optimum results, the effect of motion compensation approach based on BTC has clearly been confirmed through varied simulation studies carried out in this thesis. The performance gain achieved in terms of data compression without much degradation in image quality will certainly expand the application field for the BTC based motion estimation and compensation algorithm described here. The concepts described and implemented on test image sequences and their simplicity of computation demonstrate the potential of the BTC based motion compensated algorithm for a wider range of pictures including TV scenes, videoconferencing, and dynamic studies in medical imaging.

Most of the motion compensation coding algorithms available presently are quite sophisticated and involve a fair amount of computational complexity. However, the BTC based Motion Estimation and Compensation algorithm involves the least amount of computational complexity, and yet is capable of yielding good quality reconstructed images together with more data compression relative to direct BTC, both of which are the basic goals of any image coding algorithm.

One limitation of the Motion Estimation and Compensation algorithm based on BTC, which became so obvious in our preliminary studies followed by simulation tests, is the sensitivity of this technique to the noise present in pictures. It was absolutely essential to prefilter the images with a low-pass filter before they were subjected to the BTC

algorithm followed by Motion Estimation and Compensation. The bit-maps of previous and current frames, when exclusive-ored to produce the exclusive-or difference image, should ideally yield bright pixels only in the regions of motion between the two frames and should show dark pixels in static surroundings. If the pictures are not prefiltered, the above situation does not occur and even the stationary regions show up as a large number of bright pixels in the exclusive-or difference image created from the bit-maps.

The above limitation may be overcome by use of seeking still better methods for taking image sequences and using better quality video cameras. This observation is also based on some initial trials at an early stage of this work when we used an old TV camera that was introducing a tremendous amount of random noise as evident from bit-maps. There is, however, a lower bound on this approach, since statistical Gaussian noise inherent in images cannot be eliminated altogether. It may only be reduced by smoothing operations using low-pass filters. This, in turn, results in loss of resolution to some extent and slightly enhanced computational complexity. Such a situation may impose limitations, for instance, when medical applications require observing a small cold spot in the presence of a large hot surrounding region in a dynamic study being acquired by a medical imaging system.

In our simulation studies on different image sequences, we have collected and made subjective and objective estimates of the prediction errors in the reconstructed image. The prediction errors, however, have not been made use of in updating the reconstructed image. This was intentionally done so as to avoid the extra overhead of coding and transmitting the prediction error data. However, for precise reconstruction, this aspect may be given due consideration, although at the cost of added overhead and resulting loss in data compression.

In spite of the limitations pointed out above, it may be concluded that Motion Estimation and Compensation based on BTC is a fascinating and convenient approach towards image coding using motion compensation and is very attractive in view of the simplicity of computational complexity and improved performance as compared to coding without motion compensation. The algorithm achieves its simplicity of computational

complexity due to bit-operations on bit-map representations of gray level images, results in good quality reconstructed images as viewed on a psycho-visual criteria basis, and provides the added compression which is so important in image communication when bandwidth reduction on communication channels is always the requirement in view of the enormous bandwidth associated with video signals.

Another attractive feature of BTC based Motion Estimation and Compensation algorithm is the possibility of simpler and more inexpensive hardware implementation on a chip. This conclusion has, once again, its roots on motion detection achieved in this algorithm through bit-map operations.

To sum up, a new algorithm has been developed and presented for motion estimation and prediction/compensation. It is computationally simpler and easy to implement in hardware. **The algorithm achieves significant data compression with a compression ratio on the order of about 20:1 on the current frame in an image sequence compared to the compression ratio of 4:1 achievable with direct BTC involving no motion compensation; and yet without significant loss in image quality.** However, taking into consideration the previous frame in the sequence, the overall compression ratio is about **12:1**. A simple coder based on this algorithm can find wide range of applications including videoconferencing with various video sequences, TV sequences with large motions, and dynamic medical imaging sequences.

References

- [1] D. Healy and O. Mitchell, "Digital video bandwidth compression using block truncation coding," *IEEE Trans. Commun.*, vol. COM-29, pp. 1809–1817, Dec. 1981.
- [2] Y. Wu, *Multi-stage Hybrid Coding of Digital Images*. PhD thesis, Carleton University, Ottawa, Canada, 1990.
- [3] A. Netravali and J. Limb, "Picture coding: a review," *Proc. IEEE*, vol. 68, pp. 366–406, March 1980.
- [4] W. Pratt, *Image Transmission Techniques*. Academic Press, 1982.
- [5] A. Habibi, "Survey of adaptive coding techniques," *IEEE Trans. Commun.*, vol. COM-25, pp. 1275–1284, Nov. 1977.
- [6] A. Habibi, "Hybrid coding of pictorial data," *IEEE Trans. Commun.*, May 1974.
- [7] W. Pratt, *Digital Image Processing*. John Wiley and sons, 1978.
- [8] J. Roese et al, "Interframe cosine transform image coding," *IEEE Trans. Commun.*, vol. COM-25, pp. 1329–1338, Nov. 1977.
- [9] R. Gonzalez and P. Wintz, *Digital Image Processing*. Addison-Wesley, 1987.
- [10] R. Stafford, *Digital Television*. John Wiley & sons, 1980.
- [11] J. Storer, *Data Compression: Methods and Theory*. Computer Science Press, 1988.
- [12] J. Welch, "A technique for high performance data compression," *IEEE Computer*, vol. 17, June 1984.
- [13] P. Wintz, "Transform picture coding," *Proc. IEEE*, vol. 60, July 1972.
- [14] K. Rao and R. Srinivasn(cds.), *Teleconferencing*. Van Nostrand Reinhold Co., 1985.
- [15] A. Jain, "Image data compression: A review," *Proc. IEEE*, vol. 69, pp. 349–389, March 1981.

- [16] M. Ekstrom(ed.), *Digital Image Processing Techniques*. Academic Press, 1984.
- [17] E. Hall, *Computer Image Processing and Recognition*. Academic Press, 1979.
- [18] T. Huang and O. Tretiak(eds.), *Picture Bandwidth Compression*. Gordon and Breach, 1972.
- [19] A. Netravali and J. Robbins, "Motion compensated television coding: Part 1," *BSTJ*, vol. 58, pp. 631-670, March 1979.
- [20] A. Netravali and J. Stuller, "Motion compensated transform coding," *BSTJ*, vol. 58, Sept. 1979.
- [21] Y. Ninomiya and Y. Ohsuka, "A motion compensated interframe coding scheme for television pictures," *IEEE Trans. Commun.*, vol. COM-30, pp. 201-211, 1982.
- [22] K. Prabhu and A. Netravali, "Pel-recursive motion compensated color coding," *Proc. ICC82, Philadelphia, PA*, pp. 2G.8-1-2G.8.5, June 1982.
- [23] J. Robbins and A. Netravali, "Interframe television coding using movement compensation," *Proc. ICC79, Boston, MA*, pp. 23.4.1-23.4.5, June 1979.
- [24] F. Rocca, "Television bandwidth compression utilizing frame-to-frame correlation and movement compensation," *Symposium on Picture Bandwidth Compression, April 1969, Gordon and Breach*, 1972.
- [25] S. Sabri, "Movement compensated interframe prediction for NTSC color TV signal," tech. rep., B.N.R.L., Montreal, Canada, 1982.
- [26] R. Srinivasan and K. Rao, "Predictive coding based on efficient motion estimation," in *Teleconferencing edited by K.R. Rao and R. Srinivasan*, Van Nostrand Reinhold Co., 1985.
- [27] J. Stuller and A. Netravali, "Transform domain motion estimation," *BSTJ*, vol. 58, Sept. 1979.
- [28] H. Bergmann, "Displacement estimation based on the correlation of image segments," *IEEE Int'l Conf. on Electronic Image Processing, York, U.K.*, pp. 215-217, July 1982.

- [29] S. Brofferio, C. Cafforio, M. Piacentini, and T. Ishiguro, "Motion compensation in teleconference," in *Teleconferencing edited by K.R. Rao and R. Srinivasan*, Van Nostrand Reinhold Co., 1985.
- [30] C. Cafforio and F. Rocca, "Tracking moving objects in television images," *Signal Processing, Vol. 1*, pp. 133-140, 1979.
- [31] C. Cafforio and F. Rocca, "Methods for measuring small displacements of television images," *IEEE Trans. Theory*, vol. IT-22, pp. 573-579, 1979.
- [32] P. DeBaldo, "Compression technologies and techniques," in *Teleconferencing edited by K.R. Rao and R. Srinivasan*, Van Nostrand Reinhold Co., 1985.
- [33] E. Dubois and S. Sabri, "Video teleconferencing: squeezing the signal pays off," in *Teleconferencing edited by K.R. Rao and R. Srinivasan*, Van Nostrand Reinhold Co., 1985.
- [34] S. Sabri, K. Cuffling, and B. Prasada, "Coding of video signals at 50 kb/s using motion compensation techniques," in *Teleconferencing, edited by K.R. Rao and R. Srinivasan*, Van Nostrand Reinhold Co., 1985.
- [35] T. Koga et al, "A 1.5 mb/s interframe codec with motion compensation," *Proc. ICC83, Boston, MA*, pp. D8.7.1-D8.7.5, June 20-22, 1983.
- [36] T. Koga et al, "Motion compensated interframe coding for video conferencing," *Proc. NTC81, New Orleans, LA*, pp. G5.3.1-G5.3.5, Nov. 29-Dec. 3, 1981.
- [37] C. Lin and S. Kwatra, "Motion compensated interframe color image coding," in *Teleconferencing, edited by K.R. Rao and R. Srinivasan*, Van Nostrand Reinhold Co., 1985.
- [38] J. Limb and J. Murphy, "Estimating the velocity of moving images in television signals," *Computer Graphics and Image Processing*, vol. 4, pp. 311-327, 1975.
- [39] R. Natarajan and K. Rao, "Design of a 64 Kbps coder for teleconferencing," in *Teleconferencing, edited by K.R. Rao and R. Srinivasan*, Van Nostrand Reinhold Co., 1985.
- [40] C. Shannon, "Coding theorem for a discrete source with a fidelity criterion," *IRE National*

Convention Record, Part 4, pp. 142–163, 1959.

- [41] J. Thompson, "A 36-Mbit/s television coder employing pseudorandom quantization," *IEEE Trans. Commun. Technol.*, vol. COM-19, pp. 872–879, Dec. 1971.
- [42] P. Roetling, "Halftone method with edge enhancement and moire suppression," *J. Opt. Soc. Am.*, vol. 66, pp. 985–989, 1976.
- [43] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, pp. 1098–1101, Sept. 1952.
- [44] A. Netravali and B. Haskell, *Digital Picture Representation and Compression*. Plenum Press, 1988.
- [45] C. Cutler, "Differential quantization for commutation signals," tech. rep., U.S. Patent 2, 605, 361, July 29, 1952.
- [46] R. Steele, *Delta Modulation System*. John Wiley & Sons, 1975.
- [47] I. Paz, G. Collins, and B. Batson, "A tri-state delta modulator for run-length encoding of video," *Proc. National Telecomm. Conf., Dallas, Texas*, vol. 1, pp. 6.3–1 — 6.3–6, Nov. 1976.
- [48] T. Lei, N. Scheinberg, and D. Schilling, "Adaptive delta modulation systems for video encoding," *IEEE Trans. Commun.*, vol. COM-25, pp. 1302–1314, Nov. 1977.
- [49] J. Max, "Quantization for minimum distortion," *IRE Trans. Infor. Theory*, vol. IT-6, pp. 7–12, March 1960.
- [50] A. Netravali, "On quantizers for DPCM coding of picture signals," *IEEE Trans. Infor. Theory*, vol. IT-23, pp. 360–370, May 1977.
- [51] J. O'Neal, "Predictive quantizing differential pulse code modulation for transmission of television signal," *Bell Syst. Tech. J.*, pp. 689–722, May 1976.
- [52] D. Connor, R. Pease, and W. Scholes, "Television coding using two-dimensional spatial prediction," *Bell Syst. Tech. J.*, pp. 1049–1061, March 1971.
- [53] P. Pirsch, "Stability conditions for DPCM coders," *IEEE Trans. Commun.*, vol. COM-

25, pp. 1275-1284, Nov. 1977.

- [54] D. Coll and Y. Wu, "High fidelity compression of composite meteorological imagery," *Proc. 14th Biennial Symposium on Communications, Kingston, Ont., Canada, May 1988*.
- [55] W. Zschunke, "DPCM picture coding with adaptive prediction," *IEEE Trans. Commun.*, vol. COM-25, pp. 1295-1302, Nov. 1977.
- [56] A. Netravali and B. Prasada, "Adaptive quantization of picture signals using spatial masking," *Proc. IEEE*, vol. 65, pp. 536-548, April 1977.
- [57] J. Arnold and M. Cavenor, "Improvements to the CAQ bandwidth compression scheme," *IEEE Trans. Commun.*, vol. COM-29, pp. 1818-1822, Dec. 1981.
- [58] D. Hochman, H. Katzman, and D. Weber, "Application of redundancy reduction to television bandwidth compression," *Proc. IEEE*, vol. 55, pp. 263-266, March 1967.
- [59] C. Kurtman, "Redundancy reduction - a practical method of data compression," *Proc. IEEE*, vol. 55, pp. 253-263, March 1967.
- [60] L. Davisson, "Rate-distortion theory and application," *Proc. IEEE*, pp. 156-164, July 1972.
- [61] A. Netravali, "Interpolative picture coding using a subjective criterion," *IEEE Trans. Commun.*, vol. COM-25, pp. 503-508, May 1977.
- [62] L. Ehrman, "Analysis of some redundancy removal bandwidth compression technique," *Proc. IEEE*, vol. 55, pp. 278-287, March 1967.
- [63] A. Brown, "Compression and bilevel display of digital meteorological satellite imagery," Master's thesis, Carleton University, Ottawa, Canada, 1986.
- [64] J. Modestino and V. Bhaskaren, "Robust two-dimensional tree encoding of images," *IEEE Trans. Commun.*, vol. COM-29, pp. 1786-1798, Dec. 1981.
- [65] CCITT, *Terminal equipment and television for facsimile series, Series T Recommendations, Part II*. 1980.

- [66] ALDEN, *Coded digital facsimile for weather charts*. Jan. 1983.
- [67] P. Burt and E. Adelson, "Laplacian pyramid as a compact image code," *IEEE Trans. Commun.*, vol. COM-31, pp. 532-540, April 1983.
- [68] E. Delp and O. Mitchell, "Image compression using block truncation coding," *IEEE Trans. Commun.*, vol. COM-27, pp. 1336-1342, Sept. 1979.
- [69] M. Lema and O. Mitchell, "Absolute moment block truncation coding application to colour images," *IEEE Trans. Commun.*, vol. COM-32, pp. 1148-1157, Oct. 1984.
- [70] H. Ko and C. Lee, "Real time implementation of block truncation coding for picture data compression," *Proc. ICASSP 87*, 1987.
- [71] D. Halverson et al, "A generalized algorithm for image compression," *IEEE Trans. Acoust. Speech and Signal Proc.*, vol. ASSP-32, pp. 664-668, June 1984.
- [72] J. Raby, A. Brunswick, and C. Cliper, "Advantage in block truncation coding with applications to digital terrain elevation data," *Proc. IEEE Conf. NACOM*, 1986.
- [73] Y. Wu and D. Coll, "BTC-VQ-DCT hybrid coding of digital images." submitted to *IEEE Trans. Commun.*
- [74] Y. Wu and D. Coll, "Multi-level block truncation coding for high fidelity compression of composite imagery," in *Proc. Canadian Conference on Electrical and Computer Engineering, Vancouver, B.C., Canada*, 1988.
- [75] Y. Wu and D. Coll, "Multi-level block truncation coding using a mini-max error criterion for high fidelity compression of digital images," *Proc. 1990 Picture Coding Symposium, Cambridge, Mass., USA*, 1990.
- [76] A. Jain, "A sinusoidal family of unitary transforms," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. PAMI-1, pp. 356-365, Oct. 1979.
- [77] N. Nill, "A visual model weight cosine transform for image compression and quality assessment," *IEEE Trans. Commun.*, vol. COM-33, pp. 551-557, June 1985.
- [78] N. Jayant and P. Noll, *Digital Coding of Waveforms — Principles and Applications to Speech and Video*. Prentice-Hall, 1984.

- [79] N. Ahmed, T. Natarajan, and K. Rao, "Discrete cosine transform," *IEEE Trans. on Computers*, pp. 90–93, Jan. 1974.
- [80] A. Jain, "A fast Karhunen-Loeve transform for a class of stochastic processes," *IEEE Trans. Commun.*, vol. COM-24, pp. 1023–1029, Sept. 1976.
- [81] A. Jain, "Advances in mathematical methods for image processing," *Proc. IEEE*, vol. 69, pp. 502–528, March 1981.
- [82] J. Yan and D. Sakrison, "Encoding of images on a two component source model," *IEEE Trans. Commun.*, vol. COM-25, pp. 1315–1322, Nov. 1977.
- [83] S. Wang, *Applications of Stochastic Models for Image Data Compression*. PhD thesis, State Univ. of New York, Buffalo, USA, 1979.
- [84] M. Tasto and P. Wintz, "Image coding by adaptive block quantization," *IEEE Trans. Commun. Technol.*, vol. COM-19, pp. 956–972, Dec. 1971.
- [85] V. Algazi and D. Sakrison, "Encoding of a counting rate source with orthogonal functions," in *Computer Processing in Communications*, pp. 85–100, Polytechnic Institute of Brooklyn, N.Y., USA, 1969.
- [86] M. Navasimha and A. Peterson, "On the computation of the discrete cosine transform," *IEEE Trans. Commun.*, vol. COM-26, pp. 934–936, June 1978.
- [87] W. Chen, C. Smith, and S. Fralick, "A fast computational algorithm for the discrete cosine transform," *IEEE Trans. Commun.*, vol. COM-25, pp. 1004–1009, Sept. 1977.
- [88] W. Chen and W. Pratt, "Scene adaptive codes," *IEEE Trans. Commun.*, vol. COM-32, pp. 225–232, March 1984.
- [89] Y. Wu, "Data compression of meteorological satellite photo imagery for transmission," Master's thesis, Carleton University, Ottawa, Canada, 1986.
- [90] A. Habibi, "Compression of nth order DPCM encoder with linear transformation and block quantization techniques," *IEEE Trans. Commun.*, pp. 948–956, Dec. 1971.
- [91] C. Shannon, "A mathematical theory of communications," *Bell Sys. Tech. J.*, vol. 27, pp. 379–423, 623–656, 1948.

- [92] T. Berger, *Rate Distortion Theory*. Prentice-Hall, 1971.
- [93] R. Gray, "Vector quantization," *IEEE ASSP Mag.*, pp. 4-29, 1984.
- [94] N. Nasrabadi and R. King, "Image coding using vector quantization: a review," *IEEE Trans. Commun.*, vol. COM-36, pp. 957-971, Aug. 1988.
- [95] M. Goldberg, P. Boucher, and S. Shlien, "Image compression using adaptive vector quantization," *IEEE Trans. Commun.*, vol. COM-34, pp. 180-187, Feb. 1986.
- [96] A. Gersho and V. Cuperman, "Vector quantization: a pattern-making technique for speech coding," *IEEE Commun. Mag.*, pp. 15-21, Dec. 1983.
- [97] A. Gersho and B. Ramamurthi, "Image coding using vector quantization," *Proc. IEEE ICASSP'82, San Diego, CA, USA*, pp. 428-431, May 1982.
- [98] J. Makhoul, S. Roucos, and H. Gish, "Vector quantization in speech coding," *Proc. IEEE*, vol. COM-73, pp. 1551-1588, Nov. 1985.
- [99] Y. Linde, A. Buzo, and R. Gray, "An algorithm for vector quantizer design," *IEEE Trans. Commun.*, vol. COM-28, pp. 84-95, Jan. 1980.
- [100] S. Panchanathan, *Algorithms and Architectures for Image Coding using Vector Quantization*. PhD thesis, Univ. of Ottawa, Ottawa, Canada, 1989.
- [101] M. Goldberg and H. Sun, "Image sequence coding using vector quantization," *IEEE Trans. Commun.*, vol. COM-34, pp. 703-710, July 1986.
- [102] A. Gersho and M. Yano, "Adaptive vector quantization by progressive code vector replacement," *Proc. IEEE ICASSP'85, Tampa, FL, USA*, vol. 1, March 1985.
- [103] C. Yeh, "Color-image compression using adaptive binary-tree vector quantization with codebook replenishment," *Proc. IEEE ICASSP'87, Dallas, Texas, USA*, April 1987.
- [104] R. Arvind and A. Gersho, "Image compression based on vector quantization with finite memory," *Optical Eng. J.*, July 1987.
- [105] F. Mounts, "Low-resolution TV: An experimental digital system for evaluating bandwidth-reduction techniques," *Bell Syst. Tech. J.*, vol. 46, pp. 167-198, Jan. 1967.

- [106] R. Pease and J. Limb, "Exchange of spatial and temporal resolution in television coding," *Bell Syst. Tech. J.*, vol. 50, pp. 191-200, Jan. 1971.
- [107] J. Jain and A. Jain, "Displacement measurement and its application in interframe coding," *IEEE Trans. Commun.*, vol. COM-29, pp. 1799-1808, Dec. 1981.
- [108] S. Kappagoda and K. Rao, "Motion compensated predictive coding," in *Teleconferencing*, edited by K.R. Rao and R. Srinivasan, Van Nostrand Reinhold Co., 1985.
- [109] T. Koga et al, "Motion-compensated interframe coding for video conferencing," in *Teleconferencing*, edited by K.R. Rao and R. Srinivasan, Van Nostrand Reinhold Co., 1985.
- [110] T. Ishiguru and K. Iinuma, "Television bandwidth compression transmission by motion-compensated interframe coding," in *Teleconferencing*, edited by K.R. Rao and R. Srinivasan, Van Nostrand Reinhold Co., 1985.
- [111] F. Mounts, "A video encoding system with conditional picture element replenishment," *Bell Syst. Tech. J.*, vol. 48, pp. 2545-2554, 1969.
- [112] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*. Computer Science Press, Potomac, Md., USA, 1976.
- [113] R. Srinivasan, *Motion Compensated Hybrid CMT Coder for Videoconferencing*. PhD thesis, Univ. of Texas at Arlington, Texas, USA, 1984.
- [114] J. Jain and A. Jain, "Interframe adaptive data compression techniques for images," tech. rep., Signal and Image Processing Lab., Univ. of California, CA, USA, 1979.
- [115] MATROX Electronics Systems Ltd., Dorval, Que., Canada, *PIP-512/1024A Hardware Manual; Video Dig Board for the IBM PC, XT and AT*, 238-a50-02/0 ed., March 1986.
- [116] MATROX Electronics Systems Ltd., Dorval, Que., Canada, *MS DOS Software Library for the PIP-512/Video Digitiser*, 238-a57-01/3 ed., Feb. 1986.
- [117] SGS-Thomson Microelectronics, *STV3220 Motion Estimation Chip*, USA.
- [118] D. Coll, *personal communication*.

Appendix A

PROGRAM LISTINGS IN 'C'

A.1 Source Listing of BTCMCP421.C

```

/*BTCMCP421.C Code renamed as MCPFNL44.C Program*/
/*MCPFNL44.C PROGRAM FOR MOTION COMPENSATED PREDICTED IMAGE*/
/*Date Written:14 JULY 1990                      Kulvir Kumar Kapoor*/
/*****
#include <stdio.h>          /*standaed i/o*/
#include <math.h>           /*math definitions*/

#define ROW                2048
#define ROWADD             7168 /*fourteen more rows for previous frame*/
#define ROWOFF7            3584 /*seven rows*/
#define FILEOFFSET         -7168L /*equal to minus ROWADD*/
#define FILEOFFSET0        -2048L /*Lo back off extra 7 rows during first*/
    /**loop will lead to 21 rows back off**/
    /**in 1st loop which is not possible.***/
#define FILEOFFSET1        -1536L /*deficit 3 rows taken care off in 2nd loop*/

#define THRES              100
#define WTHRES             0/*2*/ /*to begin 1st stage search to 4 pixels*/
#define E0                 4/*step for first stage search*/
#define E1                 2 /*to extend to 2nd stage search up to 6 pixels*/
#define E2                 1 /*to extend to 3rd stage search up to 7 pixels*/
#define WTHRES1            0/*1*/ /*to extend to 2nd stage search up to 6 pixels*/
#define WTHRES2            0/*1*/ /*to extend to 3rd stage search up to 7 pixels*/

unsigned char inbufcv1[ROW+ROWADD];
unsigned char inbufcv2[ROW];blockcv2[4][4];
unsigned char inbufdec1[ROW+ROWADD],inbufdec2[ROW],inbufdif[ROW];
unsigned char blockdec1[4][4],blockdec2[4][4],blockdif[4][4];
unsigned char blockmc[4][4],blockdeclo[4][4];

```

```

unsigned char rowmc[2048],rowdeclo[2048];
unsigned int bitsmc[4][4];
unsigned int bufcountw[16];
unsigned int hicount,locount;
unsigned int hicount2,locount2;
unsigned long avg,hiavg,loavg;
unsigned long avg2,hiavg2,loavg2;

unsigned int countw;
unsigned int weightage;
int mvectory,mvectorx;
FILE *fincv1,*findec1,*findec2,*findif;
FILE *fincv2;
FILE *foutblock,*foutcountw,*fhistocountw,*foutvector,*foutwt,*foutmcpred;
FILE *foutdec1;
FILE *foutbinvec;
char outbufvec[256];
unsigned int outvecptr;

unsigned char in_matrix[120][128];
unsigned char out_matrix[120][128];

void getblockd(n)
unsigned char n;
{
    unsigned int i,j;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            blockdif[i][j] = inbufdif[512*i+4*n+j];
        }
    }
}

void codediff(c)
unsigned char c;
{
    unsigned int i,j;
    countw = 0;
    for(i=0;i<c;i++)
    {
        for(j=0;j<c;j++)
        {
            if (blockdif[i][j]==1)
            {
                countw++;
            }
        }
    }
    /* printf("countwf:%d\r",countw);*/
}

```

```

void getblockdec2(n) /*current frame block*/
unsigned char n;
{
    unsigned int i, j;
    for(i=0; i<4; i++)
    {
        for(j=0; j<4; j++)
        {
            blockdec2[i][j] = inbufdec2[512*i+4*n+j];
        }
    }
}

void getblockcv2(n) /*current frame block*/
unsigned char n;
{
    unsigned int i, j;
    for(i=0; i<4; i++)
    {
        for(j=0; j<4; j++)
        {
            blockcv2[i][j] = inbufcv2[512*i+4*n+j];
        }
    }
}

void getblockdec1(r,n,p,q) /*previous frame block*/
unsigned char r,n;
int p,q;
{
    unsigned int i, j;
    switch(r)
    {
        default:
            for(i=0; i<4; i++)
            {
                for(j=0; j<4; j++)
                {
                    if((4*n+j+q)<512)
                    {
                        blockdec1[i][j] = inbufdec1[512*(i-p)+ROWOFF7+4*n+(j+q)];
                    }
                    else
                    {
                        blockdec1[i][j] = inbufdec1[512*(i-p)+ROWOFF7+4*n+j];
                    }
                }
            }
            break;
    }
}

```

```

void blocksearch()
{
    unsigned int i, j;
    unsigned char pixel;
    weightage = 0;
    for (i=0; i<4; i++)
    {
        for (j=0; j<4; j++)
        {
            pixel = (blockdec1[i][j] ^ blockdec2[i][j]);
            if (pixel==1)
            {
                weightage++;
            }
        }
    }
    /* printf("weightsearch:%d\n", weightage); */
}

void getblockmc(r, n, p, q) /* motion compensated predicted block */
unsigned char r;
unsigned char n;
int p, q;
{
    unsigned int i, j;
    switch(r)
    {
        default:
            for (i=0; i<4; i++)
            {
                for (j=0; j<4; j++)
                {
                    if ((4*n+j+q)<512)
                        blockmc[i][j]=inbufcv1[512*(i-p)+ROWOFF7+4*n+(j+q)];
                    else
                        blockmc[i][j]=inbufcv1[512*(i-p)+ROWOFF7+4*n+j];
                }
            }
            break;
    }
}

void codemcp(c)
unsigned int c;
{
    unsigned int pixel;
    unsigned int i, j;
    avg = 0;
    for (i=0; i<c; i++)
    {
        for (j=0; j<c; j++)

```

```

        {
            avg += blockmc[i][j];
        }
    }
    avg /= (c*c);
    hcount = locount = loavg = hiavg = 0;
    for(i=0; i<c; i++)
    {
        for(j=0; j<c; j++)
        {
            pixel = blockmc[i][j];
            if(pixel>avg)
            {
                bitsmc[i][j] = 1;
                hcount += 1;
                hiavg += pixel;
            }
            else
            {
                bitsmc[i][j] = 0;
                locount += 1;
                loavg += pixel;
            }
        }
    }
    switch(locount)
    {
        case 0:
            hiavg >>= 4;
            loavg = hiavg;
            break;
        case 16: /*corresponds to hcount==0*/
/*    for(i=0; i<4; i++)
    {
        for(j=0; j<4; j++)
        {
            blockmc[i][j] += dithermask[i][j];
        }
    }
    for(i=0; i<4; i++)
    {
        for(j=0; j<4; j++)
        {
            pixel = blockmc[i][j];
            if(pixel>avg)
            {
                bitsmc[i][j] = 1;
                hcount += 1;
                hiavg += pixel;
            }
            else
            {

```



```

        bit-smc[i][j] = 0;
        locount += 1;
        loavg += pixel;
    }
}
} /*
loavg >>= 4;
hiavg = loavg;
break;
default:
    loavg /= locount;
    hiavg /= hicount;
    break;
} /*switch*/
} /*codemcp*/

void codemcpmod(c)
unsigned int c;
{
    unsigned int pixel;
    unsigned int i,j;
    avg2 = 0;
    for(i=0;i<c;i++)
    {
        for(j=0;j<c;j++)
        {
            avg2 += blockcv2[i][j];
        }
    }
    avg2 /= (c*c);
    hicount2 = locount2 = loavg2 = hiavg2 = 0;
    for(i=0;i<c;i++)
    {
        for(j=0;j<c;j++)
        {
            pixel = blockcv2[i][j];
            if(pixel>avg2)
            {
                hicount2 += 1;
                hiavg2 += pixel;
            }
            else
            {
                locount2 += 1;
                loavg2 += pixel;
            }
        }
    }
    switch(locount2)
    {
        case 0:
            hiavg2 >>= 4;

```

```

    loavg2 = hiavg2;
    break;
    case 16:          /*corresponds to hicount--0*/
/*   for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            blockmc[i][j] += dithermask[i][j];
        }
    }
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            pixel = blockmc[i][j];
            if(pixel>avg)
            {
                bitsmc[i][j] = 1;
                hicount += 1;
                hiavg += pixel;
            }
            else
            {
                bitsmc[i][j] = 0;
                locount += 1;
                loavg += pixel;
            }
        }
    }
    */
    loavg2 >>= 4;
    hiavg2 = loavg2;
    break;
    default:
    loavg2 /= locount2;
    hiavg2 /= hicount2;
    break;
    }/*switch*/
}/*codemcpmod*/

```

```

void getrowmc(number)
unsigned char number;
{
    unsigned int i,j;
    unsigned char pixel;
    for (i=0;i<4;i++)
    {
        for (j=0;j<4;j++)
        {
/*   printf("bitsmcb:%d\t",bitsmc[i][j]);*/
            if(bitsmc[i][j]==1)
                bitsmc[i][j]=hiavg2;
        }
    }
}

```

```

        else
            bitsmc[i][j]=loavg2;
/*    printf("bitsmca:%d\t",bitsmc[i][j]);*/
    rowmc[512*i+4*number+j] = bitsmc[i][j];
    }
}
}

void getblockdeclo(r,n,p,q) /* motion compensated predicted dec2 block*/
unsigned char r;
unsigned char n;
int p,q;
{
    unsigned int i,j;
    unsigned char pixel;
    switch(r)
    {
        default:
            for(i=0;i<4;i++)
            {
                for(j=0;j<4;j++)
                {
                    if((4*n+j+q<512)
                    pixel=blockdeclo[i][j]=inbufdec1[512*(i-p)+ROWOFF7+4*n+(j+q)];
                    else
                    pixel=blockdeclo[i][j]=inbufdec1[512*(i-p)+ROWOFF7+4*n+j];
                    if(pixel) blockdeclo[i][j]=255;
                    else blockdeclo[i][j]=0;
                }
            }
            break;
    }
}

void getrowdeclo(number)
unsigned char number;
{
    unsigned int i,j;
    for (i=0;i<4;i++)
    {
        for (j=0;j<4;j++)
        {
            rowdeclo[4*number+512*i+j] = blockdeclo[i][j];
        }
    }
}

void append(what)
char what;
{
    if(outvecptr<255)

```

```

    {
        outbufvec[outvecptr]=what;
        outvecptr +=1;
    }
else
    {
        fwrite(outbufvec,1,256,foutbinvec);
        outvecptr = 0;
    }
}

main(argc,argv)
int argc;
char *argv[];
{
    unsigned char row;
    unsigned int i;
    unsigned int il;
    unsigned char jl;
    int p,q;
    int tempwt;
    int tempv0,temph0,tempv1,temph1,tempv2,temph2;
    int tempver,temphor;
    unsigned char number;
    unsigned int blockcount;
    unsigned int motionblock;
    if (argc<=6)
    {
        puts("You must specify more than seven arguments!");
        exit(1);
    }
    if ((fincv1=fopen(argv[1],"rb")) == NULL)
    {
        puts("fopencv1 failed.\n");
        exit(1);
    }
    if ((fincv2=fopen(argv[2],"rb")) == NULL)
    {
        puts("fopencv2 failed.\n");
        exit(1);
    }

    if ((findec1=fopen(argv[3],"rb")) == NULL)
    {
        puts("fopendec1 failed.\n");
        exit(1);
    }
    if ((findec2=fopen(argv[4],"rb")) == NULL)
    {
        puts("fopendec2 failed.\n");
        exit(1);
    }
}

```

```

if ((fIndif=fopen(argv[5],"rb")) == NULL)
{
    puts("fopendif failed.\n");
    exit(1);
}

if ((foutblock=fopen(argv[6],"w")) == NULL) /*stores non-static blocks*/
{
    puts("error opening output text file for non-static blocks.\n");
    exit(1);
}
if ((foutcountw=fopen(argv[7],"w")) == NULL) /*stores countw */
{
    puts("error opening output text file for countw in non-static blocks.\n");
    exit(1);
}
if ((fhistocountw=fopen(argv[8],"w")) == NULL)
{
    puts("error opening output text file for countw histogram.\n");
    exit(1);
}
if ((foutvector=fopen(argv[9],"w")) == NULL) /*stores motion vectors*/
{
    puts("error opening output text file for motion vectors.\n");
    exit(1);
}
if ((foutwt=fopen(argv[10],"w")) == NULL) /*stores weightage factors*/
{
    puts("error opening output text file for weightages.\n");
    exit(1);
}
if ((foutbinvec=fopen(argv[11],"wb"), -NULL)/*stores p & q in bin file*/
{
    puts("error opening output binary file for displacement vectors.\n");
    exit(1);
}
if ((foutmcpred=fopen(argv[12],"wb")) == NULL) /* mc predicted image*/
{
    puts("error opening output binary file for mc predicted image.\n");
    exit(1);
}
if ((foutdec1=fopen(argv[13],"wb")) == NULL) /* mc predicted image*/
{
    puts("error opening output binary file for dec2 predicted image.\n");
    exit(1);
}
for(i=0;i<16;i++)
{
    bufcountw[i] = 0;
}
blockcount = 0;
for (row=0; row<128;row++) /*each row is a group of 4 rows*/

```

```

{
    number = 0;
    fread(inbufcv1, 1, ROW+ROWADD, fincv1);
    fread(inbufcv2, 1, ROW, fincv2);
    fread(inbufdec1, 1, ROW+ROWADD, findec1);
    fread(inbufdec2, 1, ROW, findec2);
    fread(inbufdif, 1, ROW, findif);
    for (i1=0; i1<9216; i1++) /*ROW+ROWADD #of pixels*/
    {
        if (inbufdec1[i1]>THRES)
inbufdec1[i1] = 1;
        else
inbufdec1[i1] = 0;
    }
    for (i1=0; i1<2048; i1++) /*ROW #of pixels*/
    {
        if (inbufdec2[i1]>THRES)
inbufdec2[i1] = 1;
        else
inbufdec2[i1] = 0;
        if (inbufdif[i1]>THRES)
inbufdif[i1] = 1;
        else
inbufdif[i1] = 0;
    }
    for(j1=0; j1<128; j1++)
    {
        mvectory = 0;
        mvectorx = 0;
        weightage = 0;
        tempwt = 256;
        tempv0=temph0=tempv1=temph1=tempv2=temph2=0;
        tempver=temphor=0;
        hiavg = 0;
        loavg = 0;

        getblockd(j1); /*xor difference image*/
        codediff(4);
/*      printf("countw:%d\n",countw);*/

        if (countw>WTHRES)
/*begin with 1st stage search and extend up to 3rd stage*/
        {
            motionblock = blockco :t;
            fprintf(foutblock, " %d\t", motionblock);
            fprintf(foutcountw, " %d\t", countw);
            bufcountw[countw]++;

            getblockdec2(j1); /*current image frame*/

            for (p=-E0; p<=E0; p += E0)

```

```

for (q=-E0; q<=E0; q += E0)
{
    if ((p==0) && (q==0))
    {
        ;
    }
    else
    {
        getblockdecl(row,jl,p, q);
        blocksearch();
        if (weightage < tempwt)
        {
            tempwt = weightage;
            tempv0 = p;
            temph0 = q;
        }
    }
}
)/*for*/
if ((tempv0!=0) || (temph0!=0))
{
    tempver = tempv0;
    temphor = temph0;
}
/*      printf("tempwt0:%d\t",tempwt);*/
if (tempwt > WTHRES1)
/*****
/*****
/*SECOND STAGE SEARCH TO 2 FURTHER PIXELS*/
/*****
/*****
{
for (p=tempver-2; p<=tempver+2; p += 2)
{
    for (q=temphor-2; q<=temphor+2; q += 2)
    {
        if ((p==tempver) && (q==temphor))
        {
            ;
        }
        else
        {
            getblockdecl(row,jl,p,q);
            blocksearch();
            if (weightage < tempwt)
            {
                tempwt = weightage;
                tempv1 = p;
                temph1 = q;
            }
        }
    }
}
}

```

```

    /*for*/
        if((tempv1!=0)|| (temph1!=0))
        {
            tempver = tempv1;
            temphor = temph1;
        }
    }
    /*    printf("tempwt1:%d\t",tempwt);*/
    if(tempwt > WTHRES2)

        /*****
        *****/
    /*THIRD STAGE SEARCH TO 1 FURTHER PIXEL*/

        /*****
        *****/
    {
for (p=tempver-1; p<=tempver+1; p += 1)
    {
        for (q=temphor-1; q<=temphor+1; q += 1)
        {
            if((p==tempver)&&(q==temphor))
            {
                ;
            }
            else
            {
                getblockdecl(row,j1,p,q);
                blocksearch();
                if(weightage < tempwt)
                {
                    tempwt = weightage;
                    tempv2 = p;
                    temph2 = q;
                }
            }
        }
    }
    /*for*/
        if((tempv2!=0)|| (temph2!=0))
        {
            tempver = tempv2;
            temphor = temph2;
        }
    }/*third stage searches ends*/
    /* printf("y2:%d\t x2:%d\n",tempver,temphor);*/

        /*****
        *****/
    /*END OF THREE STAGE SEARCH*/

        /*****
        *****/

```



```

/*      printf("tempwt2:%d\n",tempwt);*/
      fprintf(foutwt,"%d\t",tempwt);
      mvectory = tempver;
      mvectorx = temphor;
      fprintf(foutvector,"%d %d\t",mvectory,mvectorx);
/*      printf("y:%d\t x:%d\n",mvectory,mvectorx);*/
    }
    else
    {
      mvectory=0;
      mvectorx=0;
/*      printf("y:%d\t x:%d\n",mvectory,mvectorx);*/
    }
    getblockmc(row,j1,mvectory,mvectorx);
    codemcp(4);
    getblockcv2(j1);
    codemcpmod(4);
    getrowmc(number);
    getblockdeclo(row,j1,mvectory,mvectorx);
    getrowdeclo(number);
    append((char)mvectory);
    append((char)mvectorx);
    number++;
    if(number == 128)
    {
      fwrite(rowmc,1,2048,foutmcpred);
      fwrite(rowdeclo,1,2048,foutdec1);
      number = 0;
    }
    blockcount++;
  }/*for*/
/*  if(row==0||row==1||row==2||row==3||row==4)
    printf("row:%d %ld\n",row,ftell(findecl));*/
    if(row==0)
    {
      fseek(findecl,FILEOFFSET+FILEOFFSET0,SEEK_CUR);
      fseek(fincv1,FILEOFFSET+FILEOFFSET0,SEEK_CUR);
    }
    else if (row==1)
    {
      fseek(findecl,FILEOFFSET+FILEOFFSET1,SEEK_CUR);
      fseek(fincv1,FILEOFFSET+FILEOFFSET1,SEEK_CUR);
    }
    else
    {
      fseek(findecl,FILEOFFSET,SEEK_CUR);
      fseek(fincv1,FILEOFFSET,SEEK_CUR);
    }
/*  if(row==0||row==1||row==2||row==3||row==4)
    printf("row:%d %ld\n",row,ftell(findecl));*/
  }/*for*/
  for (i=0;i<16;i++)

```

```

(
    fprintf(fhistocountw," %d\t",bufcountw[i]);
)
fclose(fincv1);
fclose(findecl);
fclose(findecl2);
fclose(findif);
fclose(foutblock);
fclose(foutcountw);
fclose(fhistocountw);
fclose(foutvector);
fclose(foutwt);
fclose(foutbinvec);
fclose(foutmcpred);
fclose(foutdecl);
system("cls");
}/*main*/

```

A.2 BMP44CMP.C/BMP44DEC.C Code

```

/*BMP44CMP.C PROGRAM: BTC coding: 4 by 4 */
#include <stdio.h>
#include <math.h>
unsigned char buffer[2048],block[4][4],outbuff[2048];
unsigned int bits[4][4],outbuffptr;
unsigned long avg;
FILE *fin,*fout,*favg;

void readrow(buffer,fin)
unsigned char buffer[];
FILE *fin;
{
    fread(buffer,1,2048,fin);
}

void getblock(n)
unsigned int n;
{
    unsigned int i,j;
    for (i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            block[i][j] = buffer[512*i+4*n+j];/*&248;*/ /*strip off 3 LSBs*/
        }
    }
}

```

```

    }
}/*getblock*/

void code(a,b,c)
unsigned int a,b,c;
{
    unsigned int pixel;
    unsigned int i,j;
    avg = 0;
    for(i=0;i<c;i++)
    {
        for(j=0;j<c;j++)
        {
            avg += block[i+a][j+b];
        }
    }
    avg /= (c*c);
    for(i=0;i<c;i++)
    {
        for(j=0;j<c;j++)
        {
            pixel = block[i+a][j+b];
/* if(pixel > avg-4)*/ /*&&>32 is meaningful only for dark backgd.*/
            if (pixel > avg-3)
            {
                bits[i][j] = 1;
            }
            else
            {
                bits[i][j] = 0;
            }
        }
    }
/*    fprintf(favg," %d\t", avg);*/
} /*code*/

void append(what)
unsigned char what;
{
    if(outbuffptr<2048)
    {
        outbuff[outbuffptr]=what;
        outbuffptr +=1;
    }
    else
    {
        fwrite(outbuff,1,2048,fout);
        outbuffptr = 0;
        outbuff[outbuffptr++]=what;
    }
} /*append*/

```

```

void sendcode()
{
    unsigned char byte;
    byte = bits[0][0]*128 + bits[0][1]*64 + bits[0][2]*32
           + bits[0][3]*16 + bits[1][0]*8 + bits[1][1]*4
           + bits[1][2]*2 + bits[1][3];
    append(byte);
    byte = bits[2][0]*128 + bits[2][1]*64 + bits[2][2]*32
           + bits[2][3]*16 + bits[3][0]*8 + bits[3][1]*4
           + bits[3][2]*2 + bits[3][3];
    append(byte);
}
/*sendcode*/

/*main()*/
main(argc,argv)
int argc;
char *argv[];
{
    unsigned int i1,j1,i;
    /*char iname[30],oname[30], oname2[30];*/
    /*printf("Enter input file name: ");*/
    /*gets(iname);*/
    /*printf("Enter output file name: ");*/
    /*gets(oname);*/
    /*printf("Enter file name to store block averages: ");*/
    /*gets(oname2);*/
    /*iname = "mspi.pic";*/
    /*oname = "mspi.comp"; */
    /*fin = fopen(iname, "rb");
    fout = fopen(oname, "wb");
    favg = fopen(oname2, "w");
    if((fin==NULL)|(fout==NULL))
        printf("fopen failed \n");*/
    if(argc<2)
    {
        puts("You must specify more than one arguments!");
        exit(1);
    }
    if((fin=fopen(argv[1],"rb")==NULL)
    {
        puts("fopen failed.\n");
        exit(1);
    }
    if((fout=fopen(argv[2],"wb")==NULL)
    {
        puts("fout failed.\n");
        exit(1);
    }
    outbuffptr = 0;
    for(i1=0;i1<120;i1++)
    {

```

```

        readrow(buffer, fin);
        for(jl=0; jl<128; jl++)
        {
            getblock(jl);
            code(0,0,4);
/*      append(4); */
            sendcode();
        } /*for jl*/
        } /*for il*/
        fwrite(outbuff, 1, outbuffptr, fout);
        fclose(fin);
        fclose(fout);
        fclose(favg);
    } /*main*/

```

```

/*BMP44DEC.C PROGRAM*/
/* decode for 4 BY 4 BTC */
#include <stdio.h>
unsigned char block[4][4];
FILE *fin, *fout;
char *iname, *oname;
unsigned char row[2048];
unsigned char mask[8]= {128,64,32,16,8,4,2,1};

```

```

decode4()
{
    unsigned int i, j;
    unsigned char bits[2], byte;
    fread(bits, 1, 2, fin);
    for(i=0; i<2; i++)
    {
        byte = bits[i];
        for(j=0; j<8; j++)
        {
            if((mask[j]&byte) == 0)
            {
                block[2*i+j/4][j%4] = 0;
            }
            else
            {
                block[2*i+j/4][j%4] = 255;
            }
        }
    }
}

appendblock(number)
unsigned int number;
{
    unsigned int i, j;

```

```

/* printf("append block %d \n",number); */
for(i=0;i<4;i++)
{
    for(j=0;j<4;j++)
    {
        row[4*number + 512*i + j] = block[i][j];
    }
}
}

/* decode 4 x 4 block */
/*main()*/
main(argc,argv)
int argc;
char *argv[];
{
    unsigned int byte,number,count;
    int i;
    /*char iname[30], oname[30];*/
    /*printf("Enter input file name: ");
    gets(iname);
    printf("Enter output file name: ");
    gets(oname);*/
    /*
    iname = "mspi.cmp";
    oname = "mspi.dec";*/
    number=0;
    count = 0;
    /*fin = fopen(iname, "rb");
    fout = fopen(oname, "wb");
    if ((fin==NULL) || (fout==NULL))
        printf("fopen failed \n");*/
    if(argc<2)
    {
        puts("You must specify more than one arguments!");
        exit(1);
    }
    if((fin=fopen(argv[1],"rb"))==NULL)
    {
        puts("fopen failed.\n");
        exit(1);
    }
    if((fout=fopen(argv[2],"wb"))==NULL)
    {
        puts("fout failed.\n");
        exit(1);
    }
    /*while(((int)byte = (int)fgetc(fin)) != EOF)*/
    for(i=0;i<16384;i++)
    {
        ++count;

```

```

/*      printf("Block: %d\r",count);*/
      decode4();
      appendblock(number);
      number +=1;
      if(number == 128)
      {
        fwrite(row,1,2048,fout);
        number =0;
      }
    }

    fclose(fin);
    fclose(fout);
  }/*main*/

```

A.3 KBTC44CV.C/KBTC44DV.C Code

```

/*KBTC44CV.C CODING PROGRAM USING 4 by 4 BLOCKS*/
/*Gray scale 512X512 images*/
#include <stdio.h>
#include <math.h>
unsigned char buffer[2048],block[4][4],outbuff[2048];
unsigned int bits[4][4],outbuffptr;
unsigned int hicount,locount;
unsigned long avg,hiavg,loavg;
FILE *fin,*fout;

void readrow(buffer,fin)
unsigned char buffer[];
FILE *fin;
{
  fread(buffer,1,2048,fin);
}

void getblock(n)
unsigned int n;
{
  unsigned int i,j;
  for (i=0;i<4;i++)
  {
    for(j=0;j<4;j++)
    {
      block[i][j] = buffer[512*i+4*n+j];
    }
  }
}

```

```

}

void code(a,b,c)
unsigned int a,b,c;
{
    unsigned int pixel;
    unsigned int i,j;
    avg = 0;
    for(i=0;i<c;i++)
    {
        for(j=0;j<c;j++)
        {
            avg += block[i+a][j+b];
        }
    }
    avg /= (c*c);
    hcount = locount = loavg = hiavg = 0;
    for(i=0;i<c;i++)
    {
        for(j=0;j<c;j++)
        {
            pixel = block[i+a][j+b];
            if(pixel>avg)
            {
                bits[i][j] = 1;
                hcount += 1;
                hiavg += pixel;
            }
            else
            {
                bits[i][j] = 0;
                locount += 1;
                loavg += pixel;
            }
        }
    }
    if(hcount == 0)
    {
        hiavg = 0;
    }
    else
    {
        hiavg /= hcount;
    }
    if(locount == 0)
    {
        loavg = 0;
    }
    else
    {
        loavg /= locount;
    }
}

```



```

    }
    void append(what)
    unsigned char what;
    {
    if(outbuffptr<2048)
    {
        outbuff[outbuffptr]=what;
        outbuffptr +=1;
    }
    else
    {
        fwrite(outbuff,1,2048,fout);
        outbuffptr = 0;
        outbuff[outbuffptr]=what;
        outbuffptr++;
    }
    }

    void sendcode(n)
    unsigned int n;
    {
    unsigned char byte;
        byte = bits[0][0]*128 + bits[0][1]*64 + bits[0][2]*32
              + bits[0][3]*16 + bits[1][0]*8 + bits[1][1]*4
              + bits[1][2]*2 + bits[1][3];
        append(byte);
        byte = bits[2][0]*128 + bits[2][1]*64 + bits[2][2]*32
              + bits[2][3]*16 + bits[3][0]*8 + bits[3][1]*4
              + bits[3][2]*2 + bits[3][3];
        append(byte);
        append(hiavg);
        append(loavg);
    }

    main()
    {
    unsigned int i1,j1,i;
    char iname[30],oname[30];
    printf("Enter input file name: ");
    gets(iname);
    printf("Enter output file name: ");
    gets(oname);
    /*iname = "mspi.pic";
    oname = "mspi.cmp"; */
    fin = fopen(iname, "rb");
    fout = fopen(oname, "wb");
    if((fin==NULL)||(fout==NULL))
        printf("fopen failed \n");
    else
    {
        outbuffptr = 0;
        for(i1=0;i1<128;i1++)

```

```

    {
        readrow(buffer,fin);
/*      printf("coding row %d \n",i1);*/
        for(j1=0;j1<128;j1++)
        {
            getblock(j1);
            code(0,0,4);
/*      append(4);*/
            sendcode(4);
        }
    }
    fwrite(outbuff,1,outbuffptr,fout);
    fclose(fin);
    fclose(fout);
}

/* decode for 4 BY 4 BTC */
#include <stdio.h>
unsigned char block[4][4];
FILE *fin,*fout;
char *iname,*oname;
unsigned char hiavg,loavg,row[2048];
unsigned char mask[8]= {128,64,32,16,8,4,2,1};

decode4()
{
    unsigned int i,j;
    unsigned char bits[4],byte;
    fread(bits,1,4,fin);
    hiavg = bits[2];
    loavg = bits[3];
    for(i=0;i<2;i++)
    {
        {
            byte = bits[i];
            for(j=0;j<8;j++)
            {
                if((mask[j]&byte) == 0)
                {
                    block[2*i+j/4][j%4] = loavg;
                }
                else
                {
                    block[2*i+j/4][j%4] = hiavg;
                }
            }
        }
    }
}

appendblock(number)
unsigned int number;
{

```

```

unsigned int i,j;

/* printf("append block %d \n",number); */
for(i=0;i<4;i++)
{
    for(j=0;j<4;j++)
    {
        row[4*number + 512*i + j] = block[i][j];
    }
}
}

/* decode 4 x 4 block */
main()
{
    unsigned int byte,number,count;
    int i;
    char iname[30], oname[30];
    printf("Enter input file name: ");
    gets(iname);
    printf("Enter output file name: ");
    gets(oname);
    /*
    iname = "mspi.cmp";
    oname = "mspi.dec";*/
    number=0;
    count = 0;
    fin = fopen(iname, "rb");
    fout = fopen(oname, "wb");
    if ((fin==NULL)||(fout==NULL))
        printf("fopen failed \n");
    else {
        /* while(((int)byte = (int)fgetc(fin)) !=EOF)*/
        for(i=0;i<16384;i++)
        {
            ++count;
            /* printf("Block: %d\r",count);*/
            decode4();
            appendblock(number);
            number ++;
            if(number == 128)
            {
                fwrite(row,1,2048,fout);
                number =0;
            }
        }
        fclose(fin);
        fclose(fout);
    }
}

```

A.4 KBINDIX.C/KABSDIF.C Code

```

/*KBINDIFFX.C :creates exclusive-ored diff image*/
/* from 512*512 bit-map images. */
#include <stdio.h>
#include <fcntl.h>

unsigned char inbyte1[16][512],inbyte2[16][512],outbyte[16][512];

main(argc,argv)
int argc;
char *argv[];
{
    int file1,file2,file3,iostatus,counter;
    int temp;
    int x,y;
    FILE *fin1, *fin2, *fdif;

    if(argc<3)
    {
        puts("You must specify more than two arguments!");
        exit(1);
    }
    if((fin1=fopen(argv[1],"rb"))==NULL)
    {
        puts("fopen1 failed.\n");
        exit(1);
    }
    if((fin2=fopen(argv[2],"rb"))==NULL)
    {
        puts("fopen2 failed.\n");
        exit(1);
    }
    if((fdif=fopen(argv[3],"wb"))==NULL)
    {
        puts("fout failed.\n");
        exit(1);
    }
    file1 = fileno(fin1);
    file2 = fileno(fin2);
    file3 = fileno(fdif);

    counter=0;
    iostatus=8192;
    while(iostatus==8192&&counter<=496)
    {
        read(file1,inbyte1,8192);
        iostatus = read(file2,inbyte2,8192);
    }
}

```

```

printf("Working on line # %d\r",counter);
for(y=0;y<16;y++)
{
    for(x=0;x<512;x++)
    {
        temp = inbyte2[y][x] ^ inbyte1[y][x]; /* xor */
        outbyte[y][x] = (unsigned char)temp;
    }
    counter += 16;
    write(file3,outbyte,8192);
} /* while */
fclose(fin1);
fclose(fin2);
fclose(fdif);
} /*main */

```

```

/*KABSDIFF.C: creates absolute-difference image*/
/* from 512*512 gray level images*/
#include <stdio.h>
#include <fcntl.h>
#include <math.h>

```

```

unsigned char inbyte1[16][512],inbyte2[16][512],outbyte[16][512];

```

```

main(argc,argv)
int argc;
char *argv[];
{
    int file1,file2,file3,iostatus,counter;
    int temp;
    int x,y;
    FILE *fin1, *fin2, *fdif;

    if(argc<3)
    {
        puts("You must specify more than two arguments!");
        exit(1);
    }
    if((fin1=fopen(argv[1],"rb"))==NULL)
    {
        puts("fopen1 failed.\n");
        exit(1);
    }
    if((fin2=fopen(argv[2],"rb"))==NULL)
    {
        puts("fopen2 failed.\n");
        exit(1);
    }
}

```

```

if((fdif=fopen(argv[3],"wb"))!=NULL)
{
    puts("fout failed.\n");
    exit(1);
}
file1 = fileno(fin1);
file2 = fileno(fin2);
file3 = fileno(fdif);

counter=0;
iostatus=8192;
while(iostatus==8192&&counter<=496) read(file1,inbyte1,8192);
    iostatus = read(file2,inbyte2,8192);
    printf("Working on line # %d\r",counter);
    for(y=0;y<16;y++)
    {
        for(x=0;x<512;x++)
        {
            temp = (abs)(inbyte2[y][x] - inbyte1[y][x]); /*absolute difference*/
            outbyte[y][x] = (unsigned char)temp;
        }
        counter += 16;
        write(file3,outbyte,8192);
    } /* while */
    fclose(fin1);
    fclose(fin2);
    fclose(fdif);
} /*main */

```

A.5 NMSError.C Code

```

/*NMSError.C program for calculating*/
/* normalized mean square error (NMSE) between two images*/
/*****

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

#define ROW 16384
#define TOTALPIX 512L*480L
unsigned char inbuff1[ROW],inbuff2[ROW];

```

```

FILE *fin1,*fin2;

main(argc,argv)
int argc;
char *argv[];
{
    unsigned int i;
    unsigned int rows;
    int diff;
    unsigned long average1,average2;
    double merror,energytemp;
    float nmerror,nmsetemp,energy;
    if (argc<=2)
    {
        puts("You must specify more than one arguments!");
        exit(1);
    }
    if ((fin1=fopen(argv[1],"rb")) == NULL)
    {
        puts("fopen1 failed.\n");
        exit(1);
    }
    if ((fin2=fopen(argv[2],"rb")) == NULL)
    {
        puts("fopen2 failed.\n");
        exit(1);
    }
    average1 = 0L;
    average2 = 0L;
    merror = 0.0;
    energytemp = 0.0;
    energy = 0.0;
    nmsetemp = 0.0;
    nmerror = 0.0;
    for (rows=0; rows<30;rows++)/*each row comprises 16 rows*/
    {
        fread(inbuff1, 1, ROW, fin1);
        fread(inbuff2, 1, ROW, fin2);
        for (i=0; i < ROW; i++)
        {
            average1 += (unsigned long)inbuff1[i];
            average2 += (unsigned long)inbuff2[i];
        }
    }
    average1 /= TOTALPIX;
    average2 /= TOTALPIX;
    printf("avq1 %ld avq2 %ld\n",average1,average2);
    fseek(fin1,0L,SEEK_SET);
    fseek(fin2,0L,SEEK_SET);
    for (rows=0; rows<30;rows++)
    {
        fread(inbuff1, 1, ROW, fin1);

```

```

fread(inbuff2, 1, ROW, fin2);
for (i=0; i < ROW; i++)
{
    diff = ((int)inbuff1[i]-(int)inbuff2[i]);
    merror += pow((double)diff,2.0);
    energytemp += pow((double)inbuff1[i],2.0);
}
nmsetemp = merror/(float)(TOTALPIX);
energy = energytemp/(float)(TOTALPIX);
printf("avg.energy/pixel= %f\n",energy);
nmerror = nmsetemp/energy;
printf("mse=%f nmsetemp=%f nmse=%f %f\n",merror,nmsetemp,nmerror*100);
fclose(fin1);
fclose(fin2);
}/*main*/

```

A.6 WBCOUNTS.C Code

```

/*Program to count the total number of bright pixels in a */
/*512x512 image./

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

#define ROW 512
#define THRES 10 /*254*/
unsigned char inbuff[ROW];
FILE *fin;

main(argc,argv)
int argc;
char *argv[];
{
    unsigned int i;
    unsigned int rows;
    unsigned long countw, countb;
    if ((fin=fopen(argv[1],"rb")) == NULL)
    {
        puts("fopen failed.\n");
        exit(1);
    }
    countw = 0;
    countb = 0;

```



```

for (rows=0; rows<480;rows++)
{
    fread(inbuff, 1, ROW, fin);
    for (i=0; i < 512; i++)
    {
        if (inbuff[i] >= THRES)
            countw++;
        else
            countb++;
    }
}
printf("#of whites:%ld\n", countw);
printf("#of blacks:%ld\n", countb);
fclose(fin);
}/*main*/

```

A.7 ENRFILE.C Code

```

/*ENRFILE.C PROGRAM*/
/*Run Length Coding program for binary block motion vector file.*/
/*****

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

#define ROW 4096
#define THRES 0
#define REALPROG 100
char inbuff[ROW];
unsigned int rlwbuff[ROW], rlbuff[ROW];
float prob[ROW], probb[ROW];
float entropy,entropyb;
FILE *fin, *fout1, *fout2, *fout3, *fout4, *fout5, *fout6;

main(argc,argv)
int argc;
char *argv[];
{
    unsigned int rows;
    unsigned int i,j,k;
    unsigned int countw, countb;
    char continuation;
    unsigned int x;
    char pflag;

```

```

float sum, sumb;
if (argc<=3)
{
    puts("You must specify more than six arguments!");
    exit(1);
}
if ((fin=fopen(argv[1],"rb")) == NULL)
{
    puts("fopen failed.\n");
    exit(1);
}
if ((fout1=fopen(argv[2],"wb")) == NULL)
{
    puts("error opening output1 file");
    exit(1);
}
if ((fout2=fopen(argv[3],"wb")) == NULL)
{
    puts("error opening output2 file");
    exit(1);
}
if ((fout3=fopen(argv[4],"w")) == NULL)
{
    puts("error opening output3 file");
    exit(1);
}
if ((fout4=fopen(argv[5],"w")) == NULL)
{
    puts("error opening output4 file");
    exit(1);
}
if ((fout5=fopen(argv[6],"w")) == NULL)
{
    puts("error opening output5 file");
    exit(1);
}
if ((fout6=fopen(argv[7],"w")) == NULL)
{
    puts("error opening output6 file");
    exit(1);
}
for (j=0; j<ROW; j++)
{
    rlwbuff[j] = 0;
    rlbbuff[j] = 0;
}
for (rows=0; rows<8;rows++) /*32768 bytes in block motion vector file*/
{
    #ifdef REALPROG
    fread(inbuff, 1, ROW, fin);
    for (i=0; i < ROW; i++)
    {

```

```

        if (inbuff[i] == THRES)
inbuff[i] = 0;
        else
inbuff[i] = 1;
    }
    #else
        for (i=0;i<140;i++) inbuff[i] =1;
        for (i=140;i<300;i++) inbuff[i] = 0;
        inbuff[300] =1;
        inbuff[301] = 0;
        inbuff[302] = 1;
        for (i=303;i<429;i++) inbuff[i] = 1;
        for (i=429;i<512;i++) inbuff[i] = 0;
    #endif*/
    continuation = 0;
    for (x=0; x < ROW; x++)
    {
        if (!continuation)
    {
        pflag = inbuff[x];
        if (pflag) {
            countw = 1;
        }
        else {
            countb = 1;
        }
        continuation = 1;
        continue;
    }/*continuation if*/
    if (inbuff[x] != inbuff[x-1])
    {
        if (pflag)
        {
            rlwbuff[countw]++;
        }
        else
        {
            rlbuff[countb]++;
        }
        pflag = inbuff[x];    /* get next byte */
        if (pflag)
        {
            countw = 1; /* note that continuation will be TRUE */
        }
        else
        {
            countb = 1;
        }
    } /* discontinuity sensed */
    else
    {
        if (pflag) countw++;
    }

```

```

        else countb++;
    }
    /*for*/
    if (pflag) /* do any leftovers... */
    {
        rlwbuff[countw]++;
    }
    else
    {
        rlbbuff[countb]++;
    }
#ifdef REALPROG
    for (i=0; i < ROW; i++)
    {
        /* if (rlwbuff[i] !=0) printf ("white[%d] = %d\n", i, rlwbuff[i]);*/
        /* if (rlbbuff[i] !=0) printf ("black[%d] = %d\n", i, rlbbuff[i]);*/
    }
    break; /* ----->>> ... */
#endif
    /*for all rows do... */
    if(!fwrite(rlwbuff,2,ROW,fout1))
    {
        puts("error writing output1 data");
        exit(0);
    }
    if(!fwrite(rlbbuff,2,ROW,fout2))
    {
        puts("error writing output2 data");
        exit(0);
    }

    do
    {
        for (k=0; k<ROW; k++)
        {
            fprintf(fout3,"%4d",rlwbuff[k]);
            fprintf(fout4,"%4d",rlbbuff[k]);
            /* printf("%4d", rlwbuff[k]);*/
        }
        while (feof(fin) != 0);

        /*RLC Histo statistics is now worked out*/
        sum = 0.0;
        sumb = 0.0;
        entropy = 0.0;
        entropyb = 0.0;
        for (k=0;k<ROW;k++)
        {
            sum = sum + (float)rlwbuff[k];
            sumb = sumb + (float)rlbbuff[k];
        }
        printf("\n%f %f\n",sum,sumb);

```

```

    for (k=0;k<ROW;k++)
    {
        prob[k] = (float)rlwbuff[k] / (float)sum;
        probb[k] = (float)rlbbuff[k] / (float)sumb;
        fprintf(fout5,"%%.6f\t",prob[k]);
        fprintf(fout6,"%%.6f\t",probb[k]);
        printf("%%.6f %%.6f\n",prob[k],probb[k]);
        if(prob[k] !=0.0) entropy += (prob[k]*log(prob[k]));
        if(probb[k] !=0.0) entropyb += (probb[k]*log(probb[k]));
    }/*for*/
    entropy = -entropy/log(2.0);
    entropyb = -entropyb/log(2.0);
    printf("\nEnterpy of whites = %f\n",entropy);
    printf("Entropy of blacks = %f\n",entropyb);
/*End of RLC Histo statistics*/

fclose(fin);
fclose(fout1);
fclose(fout2);
fclose(fout3);
fclose(fout4);
fclose(fout5);
fclose(fout6);
}/*main*/

```

A.8 HLDA44CV.C/HLDA44DV.C Code

```

/*HLDA44CV.C PROGRAM*/
/* BTC coding: 4 by 4;creates 128x128 size High-mean, Low-mean,*/
/*High-low difference, and Average sub-images which can be*/
/*displayed simultaneously, one next to the other,on the TV*/
/*monitor using DISPLA3 or DISPLAY4 utility algorithms*/
#include <stdio.h>
#include <math.h>
unsigned char buffer[2048],block[4][4];
unsigned char outbuffh[2048],outbuffl[2048],outbuffhl[2048];
unsigned char outbuffa[2048];
unsigned int bits[4][4],outbuffptrh,outbuffptrl;
unsigned int outbuffptrhl,outbuffptra;
unsigned int hlicount,locount;
unsigned int avg,hiavg,loavg,hilodiff;
FILE *fin,*fouth,*foutl,*fouthl,*fouta;

void readrow(buffer,fin)
unsigned char buffer[];

```

```

FILE *fin;
{
fread(buffer,1,2048,fin);
}

void getblock(n)
unsigned int n;
{
unsigned int i,j;
for (i=0;i<4;i++)
{
for(j=0;j<4;j++)
{
block[i][j] = buffer[512*i+4*n+j] ;
}
}
}

void code(a,b,c)
unsigned int a,b,c;
{
unsigned int pixel;
unsigned int i,j;
avg = 0;
for(i=0;i<c;i++)
{
for(j=0;j<c;j++)
{
avg += block[i+a][j+b];
}
}
avg /= (c*c);
hicount = locount = loavg = hiavg = 0;
for(i=0;i<c;i++)
{
for(j=0;j<c;j++)
{
pixel = block[i+a][j+b];
if(pixel>=avg)
{
bits[i][j] = 1;
hicount += 1;
hiavg += pixel;
}
else
{
bits[i][j] = 0;
locount += 1;
loavg += pixel;
}
}
}
}/*for*/
}/*for*/

```

```
switch(loccount)
{
    case 0:
        hiavg >>=4;
        loavg = hiavg;
        break;

    case 16:
        loavg >>=4;
        hiavg = loavg;
        break;

    default:
        loavg /= locount;
        hiavg /= hicount;
        break;
}/*switch*/

hilodiff = hiavg - loavg;
}
void appendh(what)
unsigned char what;
{
    if(outbuffptrh<2048)
    {
        outbuffh[outbuffptrh]=what;
        outbuffptrh +=1;
    }
    else
    {
        fwrite(outbuffh,1,2048,fouth);
        outbuffptrh = 0;
        outbuffh[outbuffptrh++]=what;
    }
}

void appendl(what)
unsigned char what;
{
    if(outbuffptrl<2048)
    {
        outbuffl[outbuffptrl]=what;
        outbuffptrl +=1;
    }
    else
    {
        fwrite(outbuffl,1,2048,foutl);
        outbuffptrl = 0;
        outbuffl[outbuffptrl++]=what;
    }
}
void appendhl(what)
```

```

unsigned char what;
{
if(outbuffptrhl<2048)
{
outbuffhl[outbuffptrhl]=what;
outbuffptrhl +=1;
}
else
{
fwrite(outbuffhl,1,2048,fouthl);
outbuffptrhl = 0;
outbuffhl[outbuffptrhl++]=what;
}
}
void appenda(what)
unsigned char what;
{
if(outbuffptrh<2048)
{
outbuffa[outbuffptrh]=what;
outbuffptrh +=1;
}
else
{
fwrite(outbuffa,1,2048,fouta);
outbuffptrh = 0;
outbuffa[outbuffptrh++]=what;
}
}
void sendcode()
{

appendh(hiavg);
appendl(loavg);
appendhl(hilodiff);
appenda(avg);
}

main()
{
unsigned int il,jl,i;
char iname[30],onameh[30],onamel[30],onamehl[30],onamea[30];
printf("Enter input file name: ");
gets(iname);
printf("Enter hiavg file name: ");
gets(onameh);
printf("Enter loavg file name: ");
gets(onamel);
printf("Enter hilodiff file name: ");
gets(onamehl);
printf("Enter avg file name: ");
gets(onamea);

```



```

/*iname = "mspl.pic";
oname = "mspl.cmp"; */
fin = fopen(iname, "rb");
fouth = fopen(onameh, "wb");
foutl = fopen(onamel, "wb");
fouthl = fopen(onamehl, "wb");
fouta = fopen(onamea, "wb");
if((fin==NULL)|(fouth==NULL)|(foutl==NULL)|\
    (fouthl==NULL)|(fouta==NULL))
    printf("fopen failed \n");
else
{
    for(il=0;il<120;il++) /*128 in lieu of 120 may be used if*/
/*PIP's 'get' program is to be used to display images created here*/
    {
        readrow(buffer,fin);
/*      printf("coding row %d \n",il);*/
        for(jl=0;jl<128;jl++)
        {
            getblock(jl);
            code(0,0,4);
/*      append(4);          */
            sendcode();
        }/*for jl*/
        }/*for il*/
        }/*else*/
        fwrite(outbuffh,1,outbuffptrh,fouth);
        fwrite(outbuffl,1,outbuffptrl,foutl);
        fwrite(outbuffhl,1,outbuffptrhl,fouthl);
        fwrite(outbuffa,1,outbuffptra,fouta);
        fclose(fin);
        fclose(fouth);
        fclose(foutl);
        fclose(fouta);
    }/*main*/

```

```
/*HLLDA44DV.C PROGRAM*/
```

```
/******
```

```
#include <stdio.h>
```

```
unsigned char block[128][128];
```

```
unsigned char row[512];
```

```
FILE *fin,*fout;
```

```
char *iname, *oname;

decode4()
{
    unsigned int i, j;
    fread(block, 1, 16384, fin);
    for (i=0; i<128; i++)
    {
        for (j=0; j<128; j++)
        {
            row[j] = block[i][j];
        }
        for (j=128; j<512; j++)
        {
            row[j] = 0;
        }
        fwrite(row, 1, 512, fout);
    }
}

/* decode 4 x 4 block */
main()
{
    unsigned int count, byte;
    char iname[30], oname[30];
    printf("Enter input file name: ");
    gets(iname);
    printf("Enter output file name: ");
    gets(oname);
```

```

fin = fopen(lname, "rb");
fout = fopen(oname, "wb");
if ((fin==NULL)|(fout==NULL))
    printf("fopen failed \n");
else
{
    count = 0;
while((((int)byte =(int)fgetc(fin)) !=EOF) && (count <1))
    for (count=0;count<1;count++)
    {
        decode4();
        count++;
    }/*while*/
}/*else*/
fclose(fin);
fclose(fout);
}/*main*/

```

A.9 BMP128V2.C Code

```

/*BMP128C2.C PROGRAM: Interpolates 512x512 Bit-map image into*/
/*128x128 Bit-map image, which in turn can be displayed on TV*/
/*Monitor along with High-mean,Low-mean, High-Low difference*/
/*sub-images, simultaneously by using DISPLAY3 or DISPLAY4*/
/*utility programs*/
/*****/

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

#define ROW 2048

```

```

#define BMP      128
#define THRES    125

unsigned char inbuff[ROW], outbuff[BMP], block[4][4];
unsigned int outbuffptr;
unsigned char pixelvalue;
FILE *fin, *fout;

void getblock(n)
unsigned int n;
{
    unsigned int i, j;
    for(i=0; i<4; i++)
    {
        for(j=0; j<4; j++)
        {
            block[i][j] = inbuff[512*i+4*n+j];
        }
    }
}
/*getblock*/

void code(c)
unsigned int c;
{
    unsigned int i, j;
    unsigned int countw = 0;
    unsigned int countb = 0;
    for(i=0; i<c; i++)
    {
        for (j=0; j<c; j++)
        {
            if (block[i][j]==1)
            {
                countw++;
            }
            else
            {
                countb++;
            }
        }
    }
    /*for*/
    /*for*/
    if (countw>countb)
    {
        pixelvalue = 250;
    }
    else
    {
        pixelvalue = 0;
    }
}
/*code*/

void append(what)

```

```

unsigned char what;
{
    if(outbuffptr<128)
    {
        outbuff[outbuffptr] = what;
        outbuffptr +=1;
    }
    else
    {
        fwrite(outbuff,1,128,fout);
        outbuffptr = 0;
    }
}
/*append*/

main(argc,argv)
int argc;
char *argv[];
{
    unsigned int row;
    unsigned int i1,j1;
    unsigned char outptr;
    unsigned char countw, countb;
    if (argc<=1)
    {
        puts("You must specify more than one arguments!");
        exit(1);
    }
    if ((fin=fopen(argv[1],"rb")) == NULL)
    {
        puts("fopen failed.\n");
        exit(1);
    }
    if ((fout=fopen(argv[2],"wb")) == NULL)
    {
        puts("error opening output file");
        exit(1);
    }
    for (row=0; row<128;row++) /*each row comprising group of 4 rows*/
    {
        fread(inbuff, 1, ROW, fin);
        for (i1=0; i1<2048; i1++)
        {
            if (inbuff[i1]>THRES) inbuff[i1] = 1;
            else inbuff[i1] =0;
        }
        for (j1=0; j1<128; j1++)
        {
            getblock(j1);
            code(4);
/*      printf("%d\t",pixelvalue); */
            append(pixelvalue);
        }
    }
}

```

```

    }/*for*/
    fwrite(outbuff,1,outbuffptr,fout);
    fclose(fin);
    fclose(fout);
}/*main*/

```

A.10 DISPLAY3.C/DISPLAY4.C Code

```

/*DISPLAY3.C PROGRAM*/
/* decode for 4 BY 4 BTC */
#include <stdio.h>
unsigned char blockh[128][128],blockl[128][128];
unsigned char blockhl[128][128];/*No more buffer space for avg file*/
unsigned char row[512];
FILE *finh,*finl,*finhl,*fout;

decode4()
{
    unsigned int i,j;
    fread(blockh,1,16384,finh);
    fread(blockl,1,16384,finl);
    fread(blockhl,1,16384,finhl);
    for (i=0;i<128;i++)
    {
        for (j=0;j<128;j++)
            row[j] = blockh[i][j];
        for (j=128;j<256;j++)
            row[j] = blockl[i][j-128];
        for (j=256;j<384;j++)
            row[j] = blockhl[i][j-256];
        for (j=384;j<512;j++)
            row[j] = 0;
        fwrite(row,1,512,fout);
    }/*for*/
}
/* decode 4 x 4 block */
main()
{
    unsigned int count,byte;
    char inameh[30],inamell[30];

```

```

char inameh[30],oname[30];
printf("Enter hiavg file name: ");
gets(inameh);
printf("Enter loavg file name: ");
gets(inamel);
printf("Enter hilodiff or coded bmp file name: ");
gets(inamehl);
printf("Enter output file name: ");
gets(oname);
finh = fopen(inameh, "rb");
finl = fopen(inamel, "rb");
finhl = fopen(inamehl, "rb");
fout = fopen(oname, "wb");
if ((finh==NULL)|(finhl==NULL)|(finhl==NULL)|(fout==NULL))
    printf("fopen failed \n");
else
{
    decode4();
}/*else*/
fclose(finh);
fclose(finl);
fclose(finhl);
fclose(fout);
}/*main*/

```

```

/*DISPLAY4.C PROGRAM*/
/*****
#include <stdio.h>
unsigned char blockh[15][128],blockl[15][128];
unsigned char blockhl[15][128],blocka[15][128];
unsigned char row[7680];
unsigned char blockhe[8][128],blockle[8][128];
unsigned char blockhle[8][128],blockae[8][128];
unsigned char rowe[4096];
FILE *finh,*finl,*finhl,*fina,*fout;

main()
{
    unsigned int i,j,k,x,y;
    unsigned int count;
    char inameh[30],inamel[30];
    char inamehl[30],inamea[30],oname[30];
    printf("Enter hiavg file name: ");
    gets(inameh);
    printf("Enter loavg file name: ");
    gets(inamel);
    printf("Enter hilodiff file name: ");
    gets(inamehl);
    printf("Enter avg or coded bmp file name: ");
    gets(inamea);
    printf("Enter output file name: ");

```

```

gets(online);
finh = fopen(inameh, "rb");
finl = fopen(inamel, "rb");
finhl = fopen(inamehl, "rb");
fina = fopen(inamea, "rb");
fout = fopen(online, "wb");
if ((finh==NULL) || (finhl==NULL) || (finhl==NULL) ||
    (fina==NULL) || (fout == NULL))
    printf("fopen failed \n");
else
{
    count = 0;
    do
    {
        fread(blockh,1,1920,finh);
        fread(blockl,1,1920,finl);
        fread(blockhl,1,1920,finhl);
        fread(blocka,1,1920,fina);
        for (i=0;i<15;i++)
        {
            for (j=0;j<128;j++)
                row[512*i+j] = blockh[i][j];
            for (j=128;j<256;j++)
                row[512*i+j] = blockl[i][j-128];
            for (j=256;j<384;j++)
                row[512*i+j] = blockhl[i][j-256];
            for (j=384;j<512;j++)
                row[512*i+j] = blocka[i][j-384];
        } /*for */
        fwrite(row,1,7680,fout);
        count++;
    } while ( count<8);
    printf("%d\t",ftell(finh));
    printf("%d\t",ftell(finl));
    printf("%d\t",ftell(finhl));
    printf("%d\t",ftell(fina));
    /*each image is 120*128 & we are reading fifteen rows*/
    /*at a time each row containing 512 pixels, 128 from each of 4 images*/
    /*HLDA44CV gives 15360 bytes in 128*120 hiavg/loavg/hilodiff images*/
    for(i=0;i<1;i++)
    {
        fread(blockhe,1,1024,finh);
        fread(blockle,1,1024,finl);
        fread(blockhle,1,1024,finhl);
        fread(blockae,1,1024,fina);
        for (i=0;i<8;i++)
        {
            for (j=0;j<128;j++)
                rowe[512*i+j] = blockhe[i][j];
            for (j=128;j<256;j++)
                rowe[512*i+j] = blockle[i][j-128];
            for (j=256;j<384;j++)

```



```

rowe[512*i+j] = blockhle[i][j-256];
    for (j=384;j<512;j++)
rowe[512*i+j] = blockae[i][j-384];
    /*for */
    fwrite(rowe,1,4096,fout);
    /*for*/
    printf("\n%d\t",ftell(finh));
    printf("%d\t",ftell(finl));
    printf("%d\t",ftell(finhl));
    printf("%d\t",ftell(fina));
    /*else*/
fclose(finh);
fclose(finl);
fclose(finhl);
fclose(fina);
fclose(fout);
}/*main*/

```

A.11 AVERAGE2.C Code

```

/*AVERAGE2.C : creates image by averaging over 2 input images*/
#include <stdio.h>
#include <fcntl.h>
unsigned char p1[16][512],p2[16][512],p3[16][512];
FILE *finl,*fin2,*fout;
main(argc,argv)
char argc;
char *argv[];
{
    int iostatus,temp,counter;
    int x,y;

    if (argc<=3)
    {
        puts("You must specify more than two arguments!");
        exit(1);
    }
    if ((finl=fopen(argv[1],"rb")) == NULL)
    {
        puts("fopen1 failed.\n");
        exit(1);
    }
    if ((fin2=fopen(argv[2],"rb")) == NULL)
    {
        puts("error opening fopen2 file");
    }
}

```

```

    exit(1);
}
if ((fout=fopen(argv[3],"wb")) == NULL)
{
    puts("error opening output file");
    exit(1);
}
counter = 0;
iostatus = 8192;
while(iostatus == 8192)
{
    fread(p1,1,8192,fin1);
    iostatus = fread(p2,1,8192,fin2);
    printf("Working on line # %d\r",counter);
    for (y=0;y<16;y++)
    {
        for (x=0;x<512;x++)
        {
            temp = (p1[y][x]+p2[y][x])/2;
            p3[y][x] = temp;
        }
    }
    counter += 16;
    fwrite(p3,1,8192,fout);
}/*while*/
fclose(fin1);fclose(fin2);
fclose(fout);
}/*main*/

```

A.12 EXH44CON.C Code

```

/*EXH44CON.C PROGRAM FOR MOTION COMPENSATED PREDICTED IMAGE*/
/*9 JULY 1990; WRITTEN BY KULVIR KUMAR KAPOOR*/

/*The Exhaustive search program carries out exhaustive search*/
/*requiring 120 searches for each block in current frame*/
/*being matched with the one in the previous frame.This may be*/
/*compared to at most 24 searches needed in our BTCMCP421.C code*/
/*Moreover the optimum match is found by the conventional approach*/
/*of taking absolute difference of the gray level of each of 16*/
/*pixels in current block with that in the previous block, taking*/
/*the sum of these absolute differences and using this as */
/*weightage function which is to be minimized for optimum match*/

```

```

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

#define ROW      2048
#define ROWADD   7168 /*fourteen more rows for previous frame*/
#define ROWOFF7  3584 /*seven rows*/
#define FILEOFFSET -7168L /*equal to minus ROWADD*/
#define FILEOFFSET0 -2048L /*to back off extra 7 rows during first*/
    /*loop will lead to 21 rows back off*/
    /***in 1st loop which is not possible.***/
#define FILEOFFSET1 -1536L /*deficit 3 rows taken care off in 2nd loop*/

#define WTHRES   0 /*1*/ /*to begin 1st stage search to 4 pixels*/
#define E0       4 /*step for first stage search*/
#define E1       2 /*to extend to 2nd stage search upto 6 pixels*/
#define E2       1 /*to extend to 3rd stage search upto 7 pixels*/
#define WTHRES1  0 /*1*/ /*to extend to 2nd stage search upto 6 pixels*/
#define WTHRES2  0 /*1*/ /*to extend to 3rd stage search upto 7 pixels*/

unsigned char inbufcv1[ROW+ROWADD], inbufcv2[ROW];
unsigned char blockcv1[4][4], blockcv2[4][4];
unsigned char blockmc[4][4];
unsigned char rowmc[ROW];
unsigned int bitsmc[4][4];
unsigned int hicount, locount;
unsigned long avq, hiavq, loavq;

int weightage;
int mvector, mvectorx;
FILE *fincv1, *fincv2;
FILE *foutblock, *foutvector, *foutwt, *foutmcpred;

void getblockcv2(n) /*current frame block*/
unsigned char n;
{
    unsigned int i, j;
    for(i=0; i<4; i++)
    {
        for(j=0; j<4; j++)
        {
            blockcv2[i][j] = inbufcv2[512*i+4*n+j];
        }
    }
}

void getblockcv1(r,n,p,q) /*previous frame block*/
unsigned char r,n;
int p,q;
{
    unsigned int i, j;
    switch(r)
    {

```

```

default:
for(i=0;i<4;i++)
{
for(j=0;j<4;j++)
{
if((4*n+j+q)<512)
{
blockcv1[i][j] = inbufcv1[512*(i-p)+ROWOFF7+4*n+(j+q)];
}
else
{
blockcv1[i][j] = inbufcv1[512*(i-p)+ROWOFF7+4*n+j];
}
}
}
break;
}

void blocksearch()
{
unsigned int i, j;
int pixel;
pixel = 0;
weightage = 0;
for (i=0;i<4;i++)
{
for(j=0;j<4;j++)
{
pixel += (abs)((int)blockcv2[i][j] - (int)blockcv1[i][j]);
}
}
weightage = pixel;
/* printf("weightsearch:%d\n",weightage); */
}

void getblockmc(r,n,p,q) /* motion compensated predicted block */
unsigned char r;
unsigned char n;
int p,q;
{
unsigned int i, j;
switch(r)
{
default:
for(i=0;i<4;i++)
{
for(j=0;j<4;j++)
{
if((4*n+j+q)<512)
blockmc[i][j]=inbufcv1[512*(i-p)+ROWOFF7+4*n+(j+q)];
else

```

```

        blockmc[i][j]=inbufcv1[512*(1-p)+ROWOFF7+4*n+j];
    }
}
break;
}
}

void codemcp(a,b,c)
unsigned int a,b,c;
{
    unsigned int pixel;
    unsigned int i,j;
    avg = 0;
    for(i=0;i<c;i++)
    {
        for(j=0;j<c;j++)
        {
            avg += blockmc[i+a][j+b];
        }
    }
    avg /= (c*c);
    hcount = locount = loavg = hiavg = 0;
    for(i=0;i<c;i++)
    {
        for(j=0;j<c;j++)
        {
            pixel = blockmc[i+a][j+b];
            if(pixel>avg)
            {
                bitismc[i][j] = 1;
                hcount += 1;
                hiavg += pixel;
            }
            else
            {
                bitismc[i][j] = 0;
                locount += 1;
                loavg += pixel;
            }
        }
    }
    switch(locount)
    {
        case 0:
            hiavg >> 4;
            loavg = hiavg;
            break;
        case 16:
            loavg >>= 4;
            hiavg = loavg;
            break;
        default:

```

```

        loavg /= locount;
        hiavg /= hicount;
        break;
    }
}

void getrowmc(number)
unsigned char number;
{
    unsigned int i,j;
    unsigned char pixel;
    for (i=0;i<4;i++)
    {
        for (j=0;j<4;j++)
        {
            /* printf("bitsmcb:%d\t",bitsmc[i][j]); */
            if(bitsmc[i][j]==1)
                bitsmc[i][j]=hiavg;
            else
                bitsmc[i][j]=loavg;
            /* printf("bitsmca:%d\t",bitsmc[i][j]); */
            rowmc[512*i+4*number+j] = bitsmc[i][j];
        }
    }
}

main(argc,argv)
int argc;
char *argv[];
{
    unsigned char row;
    unsigned int il;
    unsigned char jl;
    int p,q;
    int tempwt;
    int tempv,temph;
    int tempver,temphor;
    unsigned int number;
    unsigned int blockcount;
    unsigned int motionblock;
    if (argc<=6)
    {
        puts("You must specify six arguments!");
        exit(1);
    }
    if ((fincvl=fopen(argv[1],"rb")) == NULL)
    {
        puts("fopencvl failed.\n");
        exit(1);
    }
}

```

```

if ((fincv2=fopen(argv[2],"rb")) == NULL)
{
    puts("fopencv2 failed.\n");
    exit(1);
}
if ((foutblock=fopen(argv[3],"w")) == NULL) /*stores non-static blocks*/
{
    puts("error opening output text file for non-static blocks.\n");
    exit(1);
}
if ((foutvector=fopen(argv[4],"w")) == NULL) /*stores motion vectors*/
{
    puts("error opening output text file for motion vectors.\n");
    exit(1);
}
if ((foutwt=fopen(argv[5],"w")) == NULL) /*stores weightage factors*/
{
    puts("error opening output text file for weightages.\n");
    exit(1);
}
if ((foutmcpred=fopen(argv[6],"wb")) == NULL) /* mc predicted image*/
{
    puts("error opening output binary file for mc predicted image.\n");
    exit(1);
}
blockcount = 0;
for (row=0; row<128;row++) /*each row is a group of 4 rows*/
{
    number = 0;
    fread(inbufcv1, 1, ROW+ROWADD, fincv1);
    fread(inbufcv2, 1, ROW, fincv2);

    for (j1=0; j1<128; j1++)
    {
        mvectory = 0;
        mvectorx = 0;
        tempwt = 6999;
        tempv=temph=0;
        tempver=temphor=0;
        hiavg = 0;
        loavg = 0;
        getblockcv1(row, j1, 0, 0);
        getblockcv2(j1);
        blocksearch();
        if (weightage>WTHRES) /*begin with three stage search*/
        {
            motionblock = blockcount;
            fprintf(foutblock, " %d\t", motionblock);
            getblockcv2(j1); /*current image frame*/
            tempwt = 6999;
            for (p=-(E0+E1+E2); p<=E0+E1+E2; p++)

```

```

for (q=-(E0+E1+E2);q<=E0+E1+E2;q++)
{
    getblockcvl(row,jl,p,q);
    blocksearch();
    if(weightage < tempwt)
    {
        tempwt = weightage;
        tempv = p;
        temph = q;
    }
}
/*for*/ /*end of search*/
tempver = tempv;
temphor = temph;
/* printf("tempver:%d\t temphor:%d\n",tempver,temphor); */
/*****
/*****
/*****END OF THREE STAGE SEARCH*****/
/*****
/*****
/*****
/*
    printf("tempwt2:%d\n",tempwt); */
    fprintf(foutwt,"%d\t",tempwt);
    mvectory = tempver;
    mvectorx = temphor;
    fprintf(foutvector,"%d %d\t",mvectory,mvectorx);
/*
    printf("y:%d\t x:%d\n",mvectory,mvectorx); */
}
else
{
    mvectory=0;
    mvectorx=0;
/*
    printf("y:%d\t x:%d\n",mvectory,mvectorx); */
}
getblockmc(row,jl,mvectory,mvectorx);
codemcp(0,0,4);
getrowmc(number);
number++;
if(number == 128)
{
    fwrite(rowmc,1,2048,foutmcpred);
    number = 0;
}
blockcount++;
}/*for*/
/*
if(row==0||row==1||row==2||row==3||row==4)
printf("row:%d %d\n",row,ftell(fincv1)); */
if(row==0)
{
    fseek(fincv1,FILEOFFSET+FILEOFFSETO,SEEK_CUR);
}
else if (row==1)

```



```

        {
            fseek(fincv1, FILEOFFSET+FILEOFFSET1, SEEK_CUR);
        }
        else
        {
            fseek(fincv1, FILEOFFSET, SEEK_CUR);
        }
    }
    /*      if (row==0||row==1||row==2||row==3||row==4)
        printf("row:%d %ld\n", row, ftell(fincv1)); */
    } /*for*/
    fclose(fincv1);
    fclose(fincv2);
    fclose(foutblock);
    fclose(foutvector);
    fclose(foutwt);
    fclose(foutmcpred);
} /*main*/

```

A.13 KSNAP123.C Code

```

/*KSNAP123.C PROGRAM TO TAKE THREE INSTANTANEOUS PICTURES*/
#include <stdio.h>
main()
{
    char *name1, *name2, *name3;
    name1 = "f1.pic";
    name2 = "f2.pic";
    name3 = "f3.pic";
    /* init_pipregs(); */
    /* init_crtc(); */
    /* clrscr(); */
    /*above three statements lead to unexpected interrupt somehow? So not used*/
    /*use pip command: pipint init.pip before using this program*/
    /* delay(100); */
    system("dir c:\\\\ Kapoor\\bmp4x4");
    snapshot();
    save_image(name1);
    snapshot();
    save_image(name2);
    snapshot();
    save_image(name3);
    exit(1);
}

```

A.14 MCPCOLOR.C Code

```

/*MCPCOLOR.C: exploits graphics software capability of Turbo 'C'*/
/*to display 512x512 images in 4 level pseudocolors on*/
/*PC/AT.*/
/*****

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

#define COL    512
#define ROW    120 /*max array size acceptable*/
#define TOTAL COL*ROW
unsigned char inbuff[ROW][COL], workbyte;
/*int xasp, yasp;*/
/*char buffer[80];*/
FILE *fin;

main(argc,argv)
int argc;
char *argv[];
{

    int graphdriver = CGA, graphmode = CGAC2;
    int i,j;
    unsigned int counter;
    initgraph(&graphdriver, &graphmode, "d:\\tc\\turbo");
    if (argc<=1)
    {
        puts("you must specify more than one arguments!");
        exit(1);
    }
    if((fin=fopen(argv[1],"rb")) == NULL)
    {
        puts("fopen failed.\n");
        exit(1);
    }
    counter = 0;
    cleardevice();
    while (counter<4)
    {

```

```

    fread(inbuff,COL,ROW,fin);
    for (i=0;i<57;i++)
    {
        for (j=0;j<320;j++)
        {
            workbyte = inbuff[i*2.133248L][j*1.6L];
            if (workbyte <64){ workbyte = 0;
            putpixel(j,i+57*counter,workbyte);}
            if ((workbyte >=64) && (workbyte< 128)) {workbyte=1;
            putpixel(j,i+57*counter,workbyte);}
            if ((workbyte>=128) &&(workbyte<192)) {workbyte=2;
            putpixel(j,i+57*counter,workbyte);}
            if ((workbyte>=192) &&(workbyte<=255)) {workbyte=3;
            putpixel(j,i+57*counter,workbyte);}
        }
        counter++;
    } /*while*/
    fclose(fin);
    getch();
    closegraph();
}

```

A.15 MEDFILTR.C Code

```

/*MEDFLTR1.C PROGRAM*/
/****MEDIAN FILTER USING 4x4 WINDOW****/
/*10 JULY 1990; written by KULVIR KUMAR KAPOOR*/
/*Median filter may be applied to the Prediction errors binary*/
/*file created in the Motion Estimation/Compensation algorithm*/
/*used to obtain predicted current frame from previous frame*/
/*in an image sequence.Transmission overhead of prediction*/
/*errors is thereby reduced to a great extent.*/
/*Median Filter also finds use in postfiltering the predicted*/
/*image to eliminate blockiness by low pass smoothing operation.*/

/*****/

#include <stdio.h>
#include <pipdrv.h>
#define NC 512
#define NR 480
#define N 3 /*4*/
#define NN 9 /*16*/ /*N TIMES N*/
FILE *fin;

```

```

median1(x,num,xmed)
float x[];
int num;
float *xmed;
{
    int n2,n2p;
    void sort();
    sort(num,x);
    n2p = (n2=num/2)+1;
    *xmed = (num%2 ? x[n2p] : 0.5*(x[n2]+x[n2p]));
}

void sort(n,ra)
int n;
float ra[];
{
    int l,j,ir,i;
    float rra;
    l = (n>>1)+1;
    ir = n;
    for(;;)
    {
        if(l>1)
            rra = ra[--l];
        else
        {
            rra = ra[ir];
            ra[ir] = ra[l];
            if(--ir==1)
            {
                ra[l] = rra;
                return;
            }
        }
        i = l;
        j = l<<1;
        while(j<=ir)
        {
            if(j<ir && ra[j]<ra[j+1]) ++j;
            if(rra<ra[j])
            {
                ra[i]=ra[j];
                j += (i=j);
            }
            else j=ir+1;
        }
        ra[i]=rra;
    }
} /* sort */

/*void vector(x,y,vec)
int x,y,vec[];

```

```

        entropy = -entropy/log(2.0);
        printf("\nEntropy   %f\n",entropy);
/*End of RLC Histo statistics*/

fclose(fin);
fclose(fout1);
fclose(fout2);
fclose(fout3);
}/*main*/

```

A.18 BMP4.BAT/MCP4.BAT Batch Files

```

/*BMP4.BAT FILE*/
del e:err*
del e:pred*
c:bmp44cmp e:pic1 e:cmp1
get e:cmp1
c:bmp44cmp e:pic2 e:cmp2
get e:cmp2
c:bmp44dec e:cmp1 e:bin1
del e:cmp1
get e:bin1
c:bmp44dec e:cmp2 e:bin2
del e:cmp2
get e:bin2
c:kbindifx e:bin1 e:bin2 e:bindif
get e:bindif
c:mcp4

```

```

/*MCP4.BAT FILE*/
get e:bin1
get e:bin2
c:kbindifx bin1 bin2 bindif
get e:pic1
get e:pic2
c:mcpfn144 pic1 pic2 bin1 bin2 bindif f1 f2 f3 mvectors f5 vdata bin predR predB
get e:predR
del e:pic1

```

```

{
int i, j;
for(i=0; i<3; i++)
    for(j=0; j<3; j++){
        vec[3*i+j]=fg_pixr(x-j+1,y-i+1);
    }
}*/

main(argc,argv)
int argc;
char *argv[];
{
    unsigned char inbuff[2048];
    /* float *data,*data1,xmed, block[N][N]; */
    unsigned char block[N][N];
    float data[NN],data1[NN],xmed;
    int i,j,k,x,y;
    long pos;
    if (argc<=1)
    {
        puts("You must specify more than one arguments!");
        exit(1);
    }
    if ((fin=fopen(argv[1],"rb")) == NULL)
    {
        puts("fopen failed.\n");
        exit(1);
    }
    for(y=0; y<NR; y++)
    {
        printf("y=%d\r",y);
        pos=y*512L;
        fseek(fin,pos,SEEK_SET);
        fread(inbuff,1,N*512L,fin);
        if(y>=1)
            for(k=0; k<NC; k++)
            {
                inbuff[k]=pixel(k,y);
            }

        for(x=0; x<NC-N/2; x++)
        {
            for(i=0; i<N; i++)
            {
                for(j=0; j<N; j++)
                {
                    block[i][j] = inbuff[512*i+x+j];
                }
            }
            for(i=0; i<N; i++)
            {
                for(j=0; j<N; j++)

```

```

    {
        data[N*i+j] = (float)block[i][j];
    }
}
for(i=1; i<=NN; i++)
{
    data1[i] = data[i-1];
}

median1(data1, NN, &xmed);

plot(x, y, (int) xmed);
}/*for x*/
}/*for y*/
fclose(fin);
}/*main*/

/*#include <stdio.h>
#include <math.h>

#define ROW      1536
#define BIG      1.0e12
#define AFAC     1.5
#define AMP      1.5
#define N        9

unsigned char inbuf[ROW];
unsigned char outbuf[ROW];
unsigned char block[3][3];
float *xmed, x[9];
unsigned char med;
FILE *fin, *fout;

void median2(x, n, xmed)
float x[];
int n;
float *xmed;
{
    int np, nm, j;
    float xx, xp, xm, sumx, sum, eps, stemp, dum, ap, am, aa, a;
    a = 0.5*(x[8]+x[0]);
    printf("a = %f\t", a);
    eps = fabs(x[8]-x[0]);
    printf("eps: %f\n", eps);
    am = (ap=BIG);
    for(;;)
    {
        sum=sumx=0.0; /*start with one pass through*/
                        /*16 pixels data in block[]*/;
        np=nm=0.0; /*no. of points above current guess, & below it*/
        xm = -(xp=BIG); /*value of the point above and closest to guess*/
                        /*and below and closest*/

```

```

for(j=1; j<=n; j++)
{
  xx = x[j];
  printf("xx:%f\t",xx);
  if(xx!=a) /*omit a zero denom in sums,update diagnostics*/
  {
    if(xx>a)
    {
      ++np;
      if(xx<xp) xp=xx;
      printf("xp:%f\t",xp);
    }
    else if(xx<a)
    {
      ++nm;
      if(xx>xm) xm=xx;
      printf("xm:%f\t",xm);
    }
    sum += dum=1.0/(eps+fabs(xx-a)); /*smoothing const used here*/
    sumx += xx*dum; /*accumulate the sums*/
    printf("sumx:%f\n",sumx);
  }
} /*for j*/
stemp = (sumx/sum)-a;
printf("stemp:%f\n",stemp);
if(np-nm>=2) /*guess too low,make another pass*/
{
  /*with new lower bound*/
  am = a;
  aa = stemp< 0.0 ? xp : xp+stemp*AMP;
  if(aa>ap) aa=0.5*(a+ap); /*new best guess(but no larger*/
  /*than upper bound)& new smoothing factor*/
  eps = AFAC*fabs(aa-a);
  a = aa;
}
else if(nm-np>=2) /*guess too high,make another pass,with*/
{
  /*new upper bound*/
  ap = a;
  aa = stemp>0 ? xm : xm+stemp*AMP; /*a new best guess*/
  /* but no smaller*/
  if(aa<am) aa=0.5*(a+am); /*than LB and a new smoothing factor*/
  eps = AFAC*fabs(aa-a);
  a = aa;
}
else
{
  if(n%2==0) /*got it!*/
  {
    /*for even n median is always an average*/
    *xmed = 0.5*(np==nm ? xp+xm : np > nm ? a+xp : xm+a);
  }
  else /*for odd n median is always one point*/
  {

```



```

        *xmed = np < nm ? a : np>nm ? xp : xm;
    }
    printf("xmed:%f\n",&xmed);
    return;
}
}
}*/

```

A.16 BMPRLCOD.C/BMPRLDEC.C Code

```

/*BMPRLCOD.C:Run Length Coding program for binary 512*512 images*/
/*****

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

#define ROW 512
#define THRES 60
#define REALPROG 100

unsigned char inbuff[ROW], outbuff[ROW];
FILE *fin, *fout, *fprintout;

main(argc,argv)
int argc;
char *argv[];
{
    unsigned int rows;
    unsigned int i;
    unsigned int outptr;
    unsigned char count, continuation;
    unsigned int x;
    unsigned int pflag;
    if (argc<= 1)
    {
        puts("You must specify more than one arguments!");
    }
}

```

```

    exit(1);
}
if ((fin=fopen(argv[1],"rb")) == NULL)
{
    puts("fopen failed.\n");
    exit(1);
}
if ((fout=fopen(argv[2],"wb")) == NULL)
{
    puts("error opening output file");
    exit(1);
}
#ifdef DEBUG
if ((fprintout=fopen(argv[3],"w")) == NULL)
{
    puts("error opening ascii printout file");
    exit(1);
}
#endif
for (rows=0; rows<428;rows++)
{
#ifdef REALPROG
    fread(inbuff, 1, ROW, fin);
    for (i=0; i < 512; i++)
        if (inbuff[i] >= THRES) inbuff[i] = 1;
        else inbuff[i] = 0;
#else
    for(i=0; i < 140; i++) inbuff[i] = 1;
    for (i=140; i < 300;i++) inbuff[i] = 0;
    inbuff[300]=1;
    inbuff[301] = 0;
    inbuff[302] =1;
    for(i=303; i < 429; i++) inbuff[i] = 1;
    for(i=429; i < 512; i++) inbuff[i] = 0;
#endif
    /*run length coding on binary image row follows*/
    outptr = 0;
    continuation = 0;
    for (x=0; x < 512; x++) {
        if (!continuation) {
#ifdef DEBUG
            fprintf(fprintout, "--> Not a continuation-- at x=%d\n",x);
#endif
            pflag = inbuff[x];
            count = 1;
            continuation = 1;
            continue;
        }
        if (inbuff[x] != inbuff[x-1])
            { /* we have a discontinuity */
#ifdef DEBUG
                fprintf(fprintout, "--> Change at x= %d\n", x);

```

```

        fprintf(fprintout,"%ld:%4d\n",pflag, count);
    #endif
    outbuff[outptr++] = count | (pflag << 7);
    pflag = inbuff[x];
    count--; /* note that continuation will be TRUE */
} /* discontinuity sensed */
else {
    count++;
    if (count == 127) {
#ifdef DEBUG
        fprintf(fprintout, "--> count = 127 at x=%d\n",x);
#endif
        outbuff[outptr++] = count | (pflag << 7);
        fprintf(fprintout, "%ld:%4d\n", pflag, count);
        count = 0;
        continuation = 0;
    } /* reached a count of 127 */
} /* same character */
} /* for */
if (count > 0) {
    outbuff[outptr++] = count | (pflag << 7);
#ifdef DEBUG
    fprintf(fprintout, "%ld:%4d\n", pflag, count);
#endif
};
if(!fwrite(outbuff,1,outptr,fout))
{
    puts("error writing output data");
    exit(0);
}
}/*for*/
fclose(fin);
fclose(fout);
fclose(fprintout);
}/*main*/

/*BMPRLDEC.C:program to decode run length coded compressed*/
/*binary image with 0-1 pixel values generated by krlcode.c program*/
/*****/

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

#define HIVAL 255
#define LOVAL 0

unsigned char inbuff[512],outbuff[512];
FILE *fin, *fout;

main(argc,argv)

```

```

int argc;
char *argv[];
{
    unsigned char workbyte;
    unsigned char x;
    unsigned int i,j,count;
    if (argc<=1.
    {
        puts("Must specify more than one arguments!");
        exit(1);
    }
    if ((fin=fopen(argv[1],"rb")) == NULL)
    {
        puts("error opening input file");
        exit(1);
    }
    /* Open output file to store 512*512 image*/
    /*out of compressed image */
    if ((fout=fopen(argv[2],"wb")) == NULL)
    {
        puts("error opening output file");
        exit(1);
    }
    for (;;)
    {
        workbyte = fgetc(fin);
        if (feof(fin))
            break;
        if (workbyte > 127)
        {
            workbyte = workbyte - 128;
            for (i=0;i<workbyte;i++)
                outbuff[i] = HIVAL;
        }
        else {
            for (i=0;i<workbyte;i++)
                outbuff[i] = LOVAL;
        }
        /*if*/
        if(!fwrite(outbuff,1,workbyte,fout))

            puts("error writing output data");
            exit(0);
    }
    /*for all bytes in the file DO..*/
    fclose(fin);
    fclose(fout);
}/*main*/

```

A.17 ENTPYPIX.C Code

```

/*ENTPYPIX.C PROGRAM*/
/* Entropy calculation Coding program for 16K grey level*/
/* hi/lo/hilo/avg compressed images created by BMPHL4CV.C Program*/
/*****/

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

#define PIXELS      15360 /*32 by 512 compressed image*/
#define MAXGLEVEL    256
unsigned char inbuff[PIXELS];
unsigned int outbuff[MAXGLEVEL];
float prob[MAXGLEVEL];
float entropy;
/*unsigned int byte;*/
float sum;
FILE *fin, *fout1, *fout2, *fout3;

main(argc,argv)
int argc;
char *argv[];
{
    unsigned int i, j, k;
    float sum;
    if (argc<=3)
    {
        puts("You must specify four arguments!");
        exit(1);
    }
    if ((fin=fopen(argv[1], "rb")) == NULL)
    {
        puts("fopen failed.\n");
        exit(1);
    }
    if ((fout1=fopen(argv[2], "wb")) == NULL)
    {
        puts("error opening output1 file");
        exit(1);
    }
    if ((fout2=fopen(argv[3], "wb")) == NULL)
    {
        puts("error opening output2 file");
        exit(1);
    }
    if ((fout3=fopen(argv[4], "w")) == NULL)
    {

```

```

    puts("error opening output3 file");
    exit(1);
}
for (j=0; j<60; j++)
{
    outbuff[j] = 0;
}
fread(inbuff, 1, PIXELS, fin);
for (i=0; i<PIXELS; i++)
{
    outbuff[inbuff[i]]++; /*replaces lengthy switch statement*/
    /*or use following two statements*/
/*    byte = inbuff[i];*/
/*    outbuff[byte]++;*/
/*    switch(byte)
    {
        case 0: outbuff[0]++;break;

        case 33: outbuff[33]++;break;
        case 34: outbuff[34]++;break;*/
/*        */
/*        */
/*case 255: outbuff[255]++;break;*/
/*default: break; */

/*    }*/ /*switch*/
    } /*for*/
    if (!fwrite(outbuff, 2, MAXGLEVEL, fout1))
    {
        puts("error writing output1 data");
        exit(0);
    }

    for (k=0; k<MAXGLEVEL; k++)
    {
        fprintf(fout2, "%4d", outbuff[k]);
    }
/*RLC Histo statistics is now worked out*/
    sum = 0.0;
    entropy = 0.0;
    for (k=0; k<MAXGLEVEL; k++)
    {
        sum = sum + (float)outbuff[k];
    }
    printf("\n%f\n", sum);
    for (k=0; k<MAXGLEVEL; k++)
    {
        prob[k] = (float)outbuff[k] / (float)sum;
        fprintf(fout3, "%.10f\t", prob[k]);
        printf("%.10f\t", prob[k]);
        if (prob[k] != 0.0) entropy += (prob[k]*log(prob[k]));
    } /*for*/

```

```

        entropy = -entropy/log(2.0);
        printf("\nEntropy   %f\n",entropy);
/*End of RLC Histo statistics*/

fclose(fin);
fclose(fout1);
fclose(fout2);
fclose(fout3);
}/*main*/

```

A.18 BMP4.BAT/MCP4.BAT Batch Files

```

/*BMP4.BAT FILE*/
del e:err*
del e:pred*
c:bmp44cmp e:pic1 e:cmp1
get e:cmp1
c:brp44cmp e:pic2 e:cmp2
get e:cmp2
c:bmp44dec e:cmp1 e:bin1
del e:cmp1
get e:bin1
c:bmp44dec e:cmp2 e:bin2
del cmp2
get e:bin2
e:kbindifx e:bin1 e:bin2 e:bindif
get e:bindif
e:mcp4

```

```

/*MCP4.BAT FILE*/
get e:bin1
get e:bin2
e:kbindifx bin1 bin2 bindif
get e:pic1
get e:pic2
e:mcp2nl44 pic1 pic2 bin1 bin2 bindif f1 f2 f3 mvectors f5 vdatabin predR predB
get e:predR
del e:pic1

```

```
c:kabsdiff predR pic2 e:errR
get e:errR
c:wbcounst e:errR
c:wbcounst bindif
get e:predB
c:kbindifx predB bi^2 e:errB
get e:errB
c:wbcounst e:errB
c:nmserror e:errB
c:nmserror e:bindif
c:enrlfile vdatadir
del e:errB
del e:errR
c:kbtcc44cv e:pic2 e:cmp2
c:kbtcc44dv e:cmp2 e:btc2
c:mcpcolor e:btc2
c:mcpcolor e:predR
c:mcpcolor e:pic2
```


END

04|03|92

FIN