To the Graduate Council:

I am submitting herewith a thesis written by Montanez Altamese Wade, entitled "Numerical Study of Linear Predictive Coding for the Compression of Gray Scale Images." I have examined the final copy of this for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

L. Montgomery Smith

L. Montgomery Smith, Major Professor

We have read this thesis
and recommend its acceptance:

Bruce Bosma

Roy S Joseph

Accepted for the Council:

Lew Minkel

Associate Vice Chancellor
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the library shall make it available to borrowers under rules of the library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Montana A. Wade

Date December 1993

Numerical Study of Linear Predictive Coding for the Compression of Gray Scale Images

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee Knoxville

Montanez Altamese Wade

December 1993

DEDICATION

This thesis is dedicated to my parents

Mr. Fred Sullivan Wade

and

Mrs. Valencia Joyce Wade

for their unwavering love and support

during my matriculation.

ACKNOWLEDGEMENTS

The author wishes to express her appreciation to her advisor and major professor, Dr. L. Montgomery Smith, for his guidance, patience, suggestions and advice during the course of this graduate program. The author would also like to acknowledge the technical assistance of Dr. Bruce W. Bomar and Dr. Roy D. Joseph, both members of the thesis committee.

Special thanks to Dr. Wesley L. Harris for encouraging me to pursue my technical and academic endeavors. I would also like to thank the state of Tennessee for its support during the course of my study.

I would like to thank my family: Fred, Valencia, and Camille Wade for their faith, encouragement and support during my studies. I would like to thank my church family and friends in Tullahoma and Massachusetts for their prayers.

Lastly, but most importantly I would like to thank God for all that he has done for without him my dream would not have become a reality.

ABSTRACT

The subject of this thesis is a system for the lossless compression of data in the format of 512 X 512 8-bit gray-scale images. Linear predictive coding cascaded with Huffman coding (LPC_HUFF) is applied to four representative images to determine if the storage requirements could be lowered by reducing the statistical redundancy in the images.

The images were compressed by filtering each image with its optimal prediction coefficients followed by Huffman encoding of the filtered image. The statistical properties of the filtered images were studied by examining the following statistical parameters: minimum and maximum values, histogram, standard deviation, and entropy. To measure the amount of data reduction, the size of the compressed image was compared to the size of the original images. The decompression stage of the LPC_HUFF system was implemented to verify that the system being studied compressed images without error.

Using an FIR predictive filter the optimal predictor coefficients were found to be nearly identical especially for lower order filters. The transient response caused outlying values to appear in the histograms of the filtered images as the length of the impulse response increased; consequently, the magnitude of the minimum and maximum values, and the entropy increased with filter order. Since the entropy results show that a simple differencing approach was optimal, the images were filtered using a first order filter to reduce

the redundancy. The compression ratio for the data files of the four representative images ranged between 0.5429 and 0.7619.

TABLE OF CONTENTS

LIST OF TABLES

TABLES **PAGE**

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The increased use of computers has greatly increased the volume of transmitted and stored data; thus, causing transmission and storage costs to also rise. Data compression is used in both business and medical data processing because of the cost savings it offered in the large volume of data manipulated in those applications[1][2]. Likewise, both communication and storage costs are reduced by compressing data before it is stored or transmitted. Effectively reducing the amount of data, increases the capacity of a communication network.

An increase in the transmission of image data has occurred in the past few years through the use of video-conference services, and facsimile transmission of newspapers and printed material. Since satellites provide a continuous stream of weather photographs and earth-resource pictures, efficient coding of pictures for these applications will significantly decrease transmission cost[3]. Because of their wide application, data compression schemes play an important role in all forms of digital image processing.

The amount of information used to represent a image is tremendous. A simple black-and-white still picture represented by a 512 X 512 array of pixels using 8-bits per pixel requires 2.1×10^6 bits of information. For example, a

television camera scanning such a scene at 30 frames per second generates information at a rate of 6.3×10^7 bits/s. Three-color channels require about three times this rate, and High-Definition Television (HDTV) requirements are even greater[4]. Thus, it is important to consider techniques for representing the information in the image with fewer bits[1][5]. Image compression reduces the volume of information needed to represent a digital image by removing redundancy from the data; thus transmission time is reduced and storage space is conserved [1][5-7].

Numerous applications require an error-free compression method[1]. Typically the archiving of medical or legal documents is required by law to use error-free compression techniques[1]. A "lossless" or error-free method removes or at least reduces redundancy in an image, while retaining all the information of the image. However, by reinserting the redundancy, the exact image is reconstructed[1][7]. Since redundancy reduction is always a reversible process, there is considerable interest in this technique. Many data compression schemes utilize linear predictive coding (LPC) to exploit the redundancy and correlation properties of the data[8-10].

Predictive coding was one of the earliest compression techniques used to reduce the amount of redundancy in a data set. The key idea underlying predictive coding is that successive signal samples tend to be correlated. Rather than

encode each successive sample, the difference between an input sample and some prediction of it is coded[8][9]. This difference has a smaller amplitude than the individual samples of the input, and, hence, can be represented by fewer bits than the original sample[4][8]. If the prediction is optimal, the differences resemble a white-noise sequence; that is, successive differences are spatially uncorrelated[4][10]. The difference is coded with either fixed- or variable-length code word, and stored with the predictor coefficients. Historically, most of the work in predictive coding has been based on linear functions. In linear predictive coding (LPC) the predictor is a linear function of other sample values[8][10-11]. LPC is a simple technique commonly used to reduce interpixel redundancies in images.

Linear predictive coding is used in many applications; including compression of speech, telemetry, and video data[4][7][8]. Likewise, predictive coding is used to compress color television signals[4-5], in archiving images, and to compress images used by remotely guided vehicles (RGV)[5-7][9][12-13]. Linear prediction is also used in videotelephony applications[6][14].

The study presented in this thesis implemented and evaluated a lossless data compression technique on four representative 512 X 512 8-bit gray scale images. The compression technique implemented was lossless linear predictive coding followed by Huffman coding, a combination

referred to as LPC_HUFF. Using the same predictive coefficients and the decoded differences, the original signal is recovered during the decoding process. Compression occurs in linear predictive coding since typically the variance and the entropy of the difference sample is less than the variance and entropy of the original signal[1][7]. The coded differences are stored using Huffman coding because it yields a near optimal number of bits for a given set of prediction error probabilities when the symbols are coded individually[1][7].

A numerical study of the LPC_HUFF method was performed for this thesis. Four representative 512 X 512 8-bit gray scale images supplied as demonstration images by the Data Translation Corporation of Marlboro, Massachusetts were selected for study. The steps involved in the research were:

* Optimal prediction coefficients calculated
* Image data filtered and quantized
* Computed statistical properties of the data (filtered image)
* Encoded data
* Images decompressed

First for the purpose of analysis, the image matrix was converted to vector form by row scanning and then stringing the elements together in a long **N** (the number of pixels in the image) vector to calculate the prediction coefficients.

During the filtering the process, the initial conditions were set to zero at the beginning of each line. It was assumed that the prediction errors had a Gaussian distribution with zero mean. To verify this assumption and gain further information about the statistical properties of the filtered images several statistical parameters were examined: minimum and maximum values, histogram, standard deviation, and entropy. Using the statistical information, the filtered images were Huffman encoded. The compression ratio was measured to examine the relative success or failure of the LPC_HUFF method. Finally, the images were decompressed to verify that the LPC_HUFF system is a lossless process.

Several results were obtained using the LPC_HUFF method. The optimal predictor coefficients were determined to be very nearly the same especially for lower order filters for the four representative images compressed in this study. The transient response caused outlying values to appear in the histograms of the filtered images as the length of the impulse response increased. Consequently, the magnitude of the minimum and maximum values increased, and the entropy increased in all but one case with an increase in filter order. The result indicated that a simple differencing is the optimal filtering technique for compressing the four representative images. The compression method implemented achieved a compression ratio ranging between 0.5429 and 0.7619.

Chapter II deals with the theoretical background of the LPC_HUFF method. The first section reviews the compression process. The second section presents the source encoder by discussing linear predictive coding and how its optimal coefficients are determined, and entropy and its significance in Huffman coding. This section concludes with a discussion on Huffman coding and its implementation. Chapter II concludes with the restoration of the original image from a compressed file to verify that the compression step successfully compresses the representative images without error. The first stage in the decompression block covers the Huffman decoding procedure used to retrieve the prediction errors. The last point discussed is retrieval of the original image from the filtered file using an inverse filter with the same filter coefficients as the encoding process.

Chapter III discusses the algorithms written in the C programming language to implement the data compression technique covered in Chapter II. Four programs were written to implement the compression system being studied: **LNR_PC**, **HISTO**, **HUFF_COMP**, and **HUFF_EXPAND**. The **LNR_PC** program produces an image filtered with the predictor coefficients. The **HISTO** program performs a statistical analysis on the filtered image data file to calculate the minimum and maximum values, standard deviation and the entropy of the filtered image. Using the standard deviation, the probability distribution function of each image is calculated. The

results of the **HISTO** program are used by **HUFF_COMP** to implement the Huffman encoding technique. After encoding a filtered image with the Huffman coding technique, the compression ratio is calculated. The last section of Chapter III is the implementation of the decompression operation using the **HUFF_EXPAND** and **LNR_PC** programs. The **HUFF_EXPAND** program implements the Huffman decoding process. The inverse filter step performs the reverse of the filtering operation; the *Inv_filter* routine in the **LNR_PC** program implements this step.

Results of the data compression technique are presented in Chapter IV. The information presented includes the prediction coefficients, the statistical parameters of each filtered image and the compression ratio achieved using the LPC_HUFF method. Finally Chapter V presents the conclusions and recommendations for future research in the project's area.

CHAPTER II

BACKGROUND

1. COMPRESSION SYSTEM MODEL

The principal goal in the design of an image coding system is to reduce the image transmission rate requirements subject to some image quality constraint. One way to accomplish this goal is to reduce the statistical redundancy of an image source[7]. Image compression results from reduction in the amount of signal space that must be allocated to an individual image[7]. Its purpose is to represent information in a minimal amount of space[15].

A compression system can be modeled as in Figure 1 by the sequence of two stages: an encoder and a decoder. An input image, F(x,y), is fed into the encoder which removes source redundancies by creating a new encoded image, f(x,y). The decoder takes a encoded image, f(x,y) and reconstructs the original image, F(x,y). Since the compression method being modeled is lossless, the restored image must be identical to the original image.

For this study, the format of the data is a 2-D array with a one line of header information followed by 512 lines of data with 512 pixels in each line. The 2-D array (513 X 512 pixels) is treated as 1-D array (262656 samples) with rows appended successively. This chapter discusses the

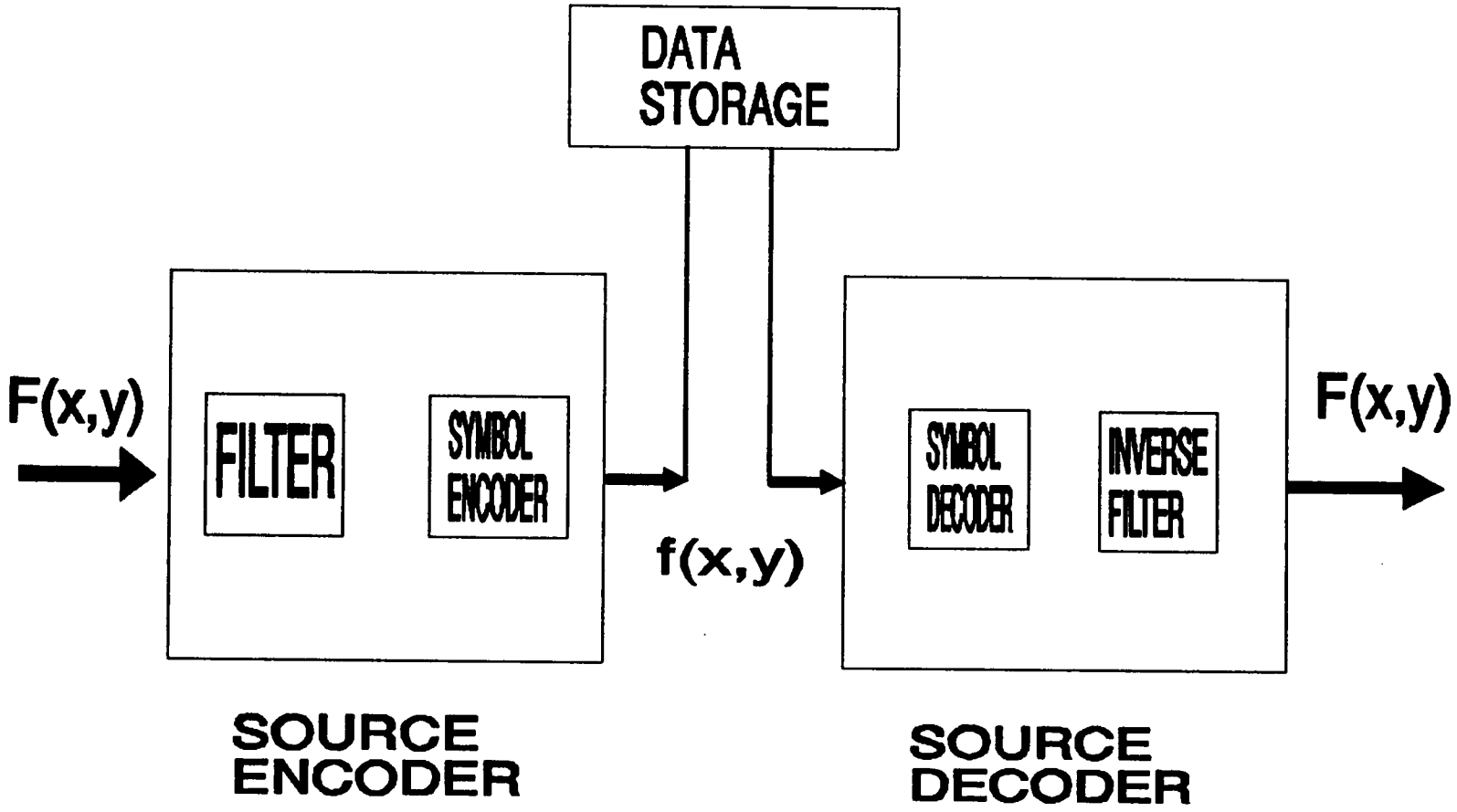

Figure 1 Compression System Model

implementation of the LPC_HUFF compression system.

2. SOURCE ENCODER

In this system the source encoder is responsible for reducing or eliminating interpixel redundancies in the input image[1][11]. This stage performs the compression operation. Normally, the approach as illustrated in Figure 1 can be modeled as a sequence of two operations: a filter and a symbol encoder.

2.1 Filter

The filter maps the input data from the pixel domain into a second domain to reduce interpixel redundancies in the input image[1]. For efficient coding, fewer bits should be required to code the mapped data. The technique selected to perform the mapping operation was linear predictive coding (LPC). This technique eliminates interpixel redundancies of closely spaced pixels by extracting and coding only the *new information* in each pixel[1]. The new information is defined as the difference between the actual and predicted value of that pixel[1].

Figure 2 illustrates the implementation of the filter stage where I_n is the intensity of the input pixel, $\hat{I}_n$ is the predicted value of the intensity of the input pixel, and ϵ_n is the difference or prediction error. The predictor estimates the current input pixel based on a linear combination of the

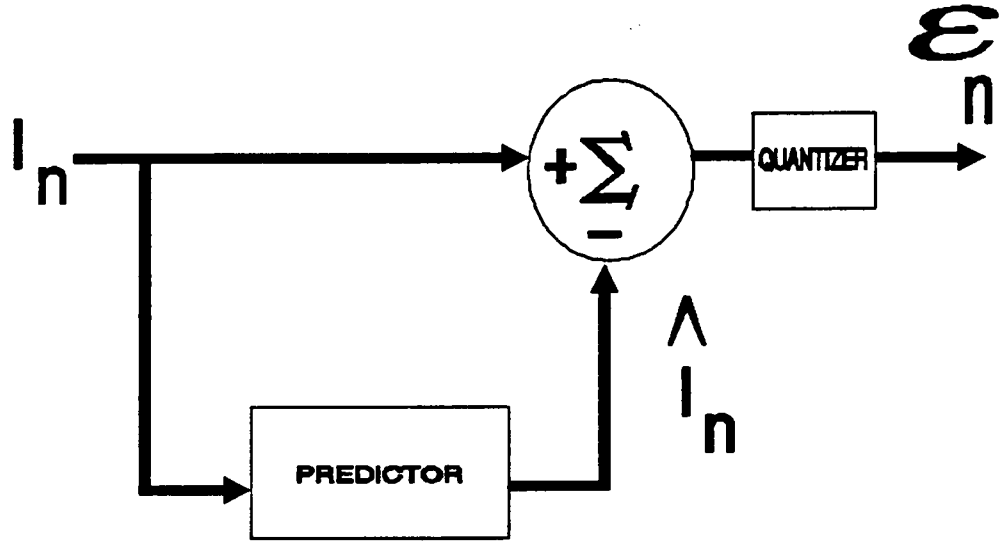

Figure 2 Filter Implementation

previous *M* pixels. I_n is estimated by

$$\hat{I}_n = \sum_{i=1}^{M} q(a_i\ I_{n-i}) \qquad (1)$$

where *M* is the order of the linear predictor, and a_n are the predictor coefficients. In this study the 1-D linear prediction $\hat{I}_n$ is a function of the previous *M* pixels on the current line alone. When the filter is implemented, the initial conditions are set to zero. Thus, pixel values

$$I_{n-i} = 0 \qquad (2)$$

when $n-i < 0$.

As each pixel of the input image, I_n, is introduced to the

encoder, the predictor generates the anticipated value of the pixel. The output of the predictor is rounded to the nearest integer as represented by the q(.) operator. The rounded estimate, $\hat{I}_n$, is then subtracted from the original pixel I_n to form the difference

$$\epsilon_n = I_n - \hat{I}_n \tag{3}$$

where ϵ_n is the n^{th} calculated prediction error. For practical implementation these operations may be carried out by an FIR filter as

$$\begin{aligned} \epsilon_n &= I_n - \sum_{i=1}^{M} q(h_i\ I_{n-i}) \\ &= \sum_{i=0}^{M} q(h_i\ I_{n-i}) \end{aligned} \tag{4}$$

where $h_0=1$ *and* $h_i= -a_i$ *for* $i > 0$.

For efficient coding, optimal coefficients should be used. The classical technique for selecting the optimum predictor is to use the predictor that minimizes the mean square value of the of the average error term[10]. Initially the classical least squares approach was attempted to determine the optimal coefficients. Using an approach similar to the method in [16], the covariance matrix was found to be ill-conditioned. Thus, an alternative method was selected for optimizing the coefficients. The optimal coefficients were calculated using an eigenvector approach and a least squares

technique commonly used in spectral estimation[17-19]. The best estimate of I_n is that value of $\hat{I}_n$ for which the mean-square error is minimum.

Using vector notation, the convolution in (3) can be written as

$$\epsilon = Ih \tag{5}$$

where the vector notation of ϵ, I and h are defined in the following manner

$$\begin{bmatrix} \epsilon_o \\ \epsilon_1 \\ \epsilon_2 \\ \cdot \\ \cdot \\ \cdot \\ \epsilon_{N-1} \end{bmatrix} = \begin{bmatrix} I_o & 0 & 0 & \cdot & \cdot & \cdot & 0 \\ I_1 & I_0 & 0 & \cdot & \cdot & \cdot & 0 \\ I_2 & I_1 & I_0 & \cdot & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & \cdot & & \cdot & & \cdot \\ \cdot & \cdot & \cdot & & & \cdot & \cdot \\ I_{N-1} & I_{N-2} & \cdot & \cdot & \cdot & I_{N-2-M} & I_{N-1-M} \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ \cdot \\ \cdot \\ \cdot \\ h_{M-1} \\ h_M \end{bmatrix} \tag{6}$$

where N is the total number of data samples (in this case 262656).

To find the optimum predictor the mean square value

$$\epsilon^T\epsilon \tag{7}$$

is minimized subject to the constraint that

$$h^Th \neq 0. \tag{8}$$

This error can be minimized by the methods of LaGrange

multipliers and equating the derivative of the sum to zero

$$\frac{\partial}{\partial \boldsymbol{h}} (\boldsymbol{h}^T I^T I \boldsymbol{h} - \lambda \boldsymbol{h}^T \boldsymbol{h}) = 0 \qquad (9)$$

Differentiating (9) with respect to the coefficients and equating the derivative to zero yields

$$(I^T I)\boldsymbol{h} = \lambda \boldsymbol{h}. \qquad (10)$$

Thus, for an input image the optimal coefficients that minimize the mean square value of the output (7) depend only on the autocorrelation of the pixels in the image. The result in (10) implies that $\mathbf{h}$ is an eigenvector of I^TI; therefore the quantity in (7) is minimized by choosing the eigenvector with the smallest eigenvalue. The eigenvectors and eigenvalues of the covariance matrix I^TI can be found by Jacobi diagonalization. The predictor coefficients can be calculated from (9) as given in (3) and are normalized so that

$$\boldsymbol{h}_0 = 1. \qquad (11)$$

The output of the filtering step is a file containing prediction errors.

2.2 Information Theory Techniques

Using information theory techniques, the amount of information contained in the filtered image can be calculated. The generation of output data can be modeled as a probabilistic process[20]. Based on this assumption, the information content of a message (prediction error value) is

defined as

$$I(M_k) = \log_2 \frac{1}{P_k} = -\log_2 P_k \qquad (12)$$

where M_k is the k^{th} random message with the probability of occurrence P_k and I(.) is the self information of the message M_k.

Hence, the lower the probability of a message, the larger its information content[4][11]. The self-information content defines the number of bits the message will require in order to be coded. Averaging the information content for all possible M_k source messages likely to happen yields

$$H = -\sum_k P_k \log_2 P_k \ \ bits/message \qquad (13)$$

which is called the entropy of the source. Entropy is a measure of the average amount of information contained in the data[17]. The more probable the message, the lower its entropy. Using (13) and the source probabilities estimated from occurrences in a filtered data file, the entropy of the file can be determined, and then used to calculate a lower bound on the compressed file size.

Upon completing the symbol encoder step, the input image has been processed to remove interpixel redundancies. The output of the source encoder is a compressed file. The compression ratio (CMPR) measures the compression performance of a data reduction technique. It provides the degree of data reduction and is defined as

$$Compression\ Ratio = \frac{Size\ of\ Compressed\ File}{Size\ of\ Original\ File} \quad (14)$$

where *the file size is expressed in bytes.*
The lower the compression ratio falls the more effective the compression technique employed.

2.3 Symbol Encoder

In the second stage of the source encoding process, the symbol encoder creates either a fixed- or a variable-length code word to represent each mapped output value. For each input to the coder, ϵ_n, the coder outputs a binary word whose value depends on the value of the input word. Since some prediction errors occur more often than others, greater efficiency can be achieved by using a variable-length code and assigning the shortest code words to the most likely inputs and longer code words to the least likely inputs[1]. The symbol encoder block was implemented with Huffman coding. This technique was selected because it generates a variable length code with an average word length that is very close to the entropy of the difference[1][20].

Huffman coding is one of several techniques whereby data are encoded based upon their probability of occurrence. Data compression is achieved by reducing the number of bits used to represent the higher probability messages and increasing the number of bits representing lower probability messages. Huffman coding creates variable-length codes that are an

integral number of bits. The result is an optimal code whose average word length is less than or equal to the average length of all other uniquely decodable codes for a given set of input probabilities[1][7]. In practice, this technique has often achieved average code word lengths nearly that predicted by the entropy.

Huffman coding has several attributes. First it is an instantaneous uniquely decodable block code[1], it is so named because each source symbol is mapped into a fixed sequence of bits[1]. The descriptor "instantaneous property" means that each word in a message is decoded without referencing succeeding words. Another characteristic of Huffman coding is that it has unique prefix attributes; consequently, no short code word is duplicated as the beginning of a longer code word[1][23]. This unique prefix characteristic makes the code instantaneously decodable. The unique prefix property means that a code word can be decoded one way for a given set of source probabilities[1][24]. Huffman coding is typically represented with a tree structure whereby new nodes are built from the bottom up, starting with the leaves of the tree and working progressively closer to the root[24]. Each node has either two children or none.

Expressed graphically, Huffman's algorithm takes as input a list of weights and constructs a full binary tree. Using the probabilities of the input data, the Huffman process constructs an encoding tree. The leaf nodes of the tree are

comprised of the source data symbols. Each node has a weight w_i which is simply the frequency of the symbol's appearance in the data. This encoding tree is built following these steps:

* Arrange the source probabilities P_k in descending order.
* Form a list of available nodes . The initial list consists of the given input probabilities.
* Locate the two nodes with the lowest weights.
* Form a parent node from these two sibling nodes. Assign a weight equal to the sum of the two siblings nodes.
* Add the parent node to the list of available nodes and delete the two sibling nodes from the list.
* Assign one side of the nodal path a 0 and the other a 1 bit. The assignment of 0 or a 1 bit is arbitrary but must be consistent.
* The previous steps are repeated until only one available node is left. This last available node is then designated the root node of the tree.

The resulting encoding tree has leaves of varying distance from the root. The leaves that represent the symbols with the highest probabilities are closest to the root, while those with the lowest probabilities are farthest away. For more details on the implementation of the Huffman coding technique, the reader is referred to [11][16][21][25].

The following example illustrates the Huffman algorithm outlined above for the source probabilities P_k in Table 1.

Table I Source Probabilities

Messages	Probabilities
M_1	0.25
M_2	0.20
M_3	0.15
M_4	0.12
M_5	0.10
M_6	0.10
M_7	0.08

The Huffman coding algorithm starts by placing the input probabilities in descending order on the leaf nodes of the tree. Initially, the list of probabilities for the available nodes is comprised of the source probabilities: {0.25, 0.20, 0.15, 0.12, 0.10, 0.10, 0.08}. In this Huffman process the higher the probability node will be assigned a bit value of 1 while a 0 is assigned to the lower probability node. Beginning at the bottom of the tree, nodes M_7 and M_6, the leaf nodes with the two smallest probabilities are selected. The probabilities for these two nodes are summed and 0.18, the composite frequency (the sum of nodes M_7 and M_6) forms a parent node. The node for M_7 is assigned a bit value of 1 and the node for M_6 is assigned a value of 0. The probabilities for M_7 and M_6 are deleted from the list of probabilities for the available nodes, and the probability for the new parent node is added to the list. The list is then comprised of the

following values: {0.25, 0.20, 0.18, 0.15, 0.12, 0.10}. Next, the probabilities for M_5 and M_4 are summed to form a parent node with a probability of 0.22, and their probabilities are deleted from the list. The node for M_5 is assigned a value of 1 and the node for M_4 is assigned a value of 0. The probability of 0.22 is then added to the list. The node probability list is then comprised of the following values: {0.25, 0.22, 0.20, 0.18, 0.15}. This process is repeated until only one node is left with probability of 1. This last available node is designated the root node.

To determine the code for a given symbol, the bits are read, traversing the path from the root node to the leaf node of the tree. Figure 3 illustrates the tree generated by this example. For example, using the tree M_7, is encoded as 1110 and M_4 is coded as 011. The codes created by the Huffman tree are listed in Table 2.

Table II Huffman Codes

Message	Code
M_1	10
M_2	00
M_3	110
M_4	011
M_5	010
M_6	1111
M_7	1110

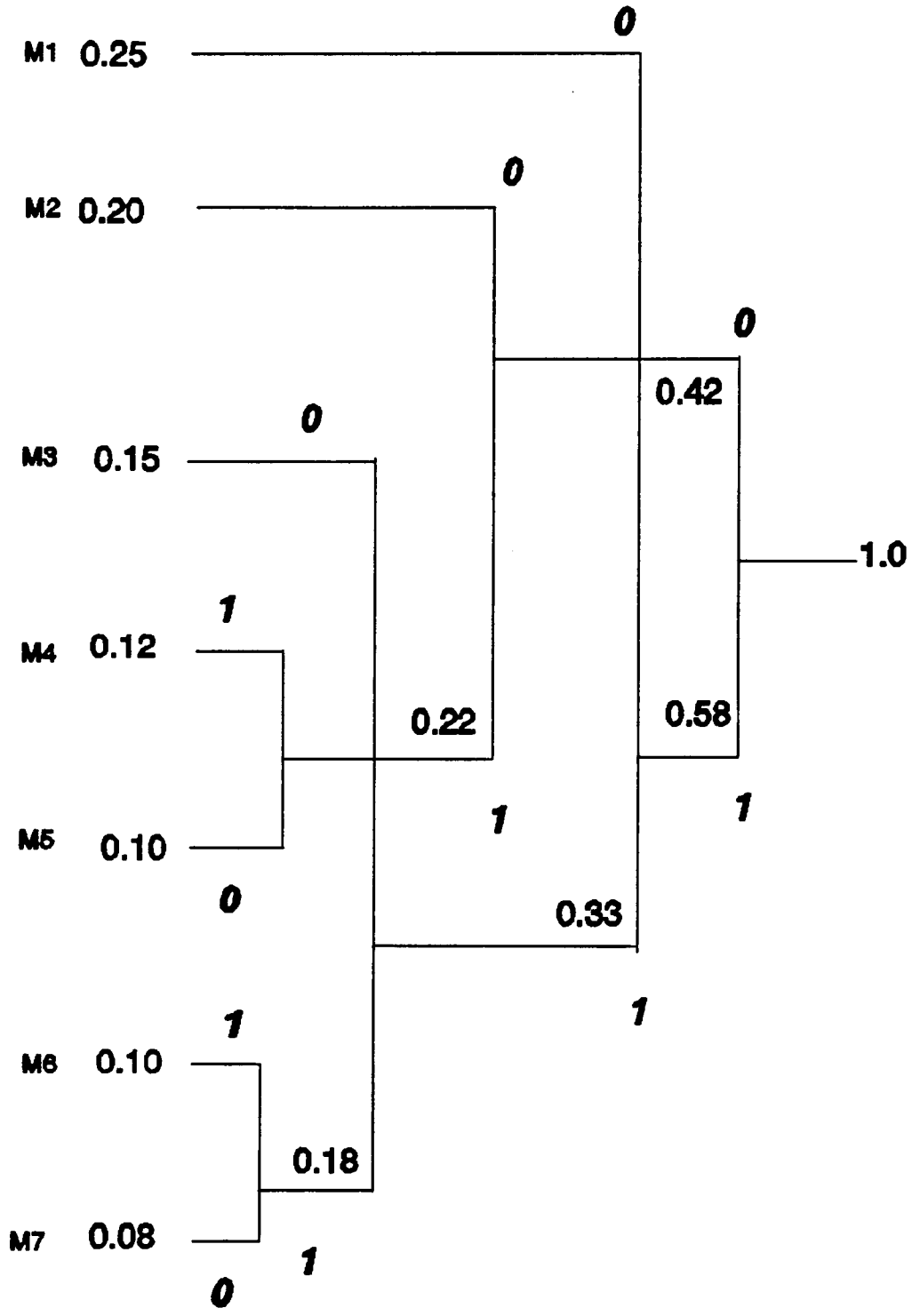

Figure 3 Huffman Tree

3. SOURCE DECODER

The second stage in the compression system in Figure 1 is the source decoder block, and implements the decompression process. The source decoder involves a symbol decoder and the inverse filter[1]. These blocks perform in reverse order the inverse operations of the symbol encoder and filter blocks in the source encoder stage.

3.1 Symbol Decoder

The symbol decoder block is implemented using the Huffman decoding technique. The decoder uses the same tree as the encoder, but the coding process works in reverse. The encoding tree is built using the same technique outlined previously in section 2.3. Commencing at the root of the tree, the decoder reads a bit and transverses the tree. The code travels to either a 0 branch or to 1 branch, depending upon the bit received by the decoder. The code progresses branch to branch until a leaf node is encountered. The symbol at the leaf node is the decoded character. The Huffman decoding technique maps a given code word, c_k, to a message M_k corresponding to a prediction error. The symbol decoder reconstructs ϵ_n from the received variable-length code word; the output of the symbol decoder is a file of decoded prediction errors.

3.2 Inverse Filter

The last step in the decoder is the inverse filter. This filter recovers the original pixel values of the image from the file of prediction errors. Implementation of the inverse filter is illustrated in Figure 4. I_n is the restored pixel, $\hat{I}_n$ is the predicted pixel, and ϵ_n is the prediction

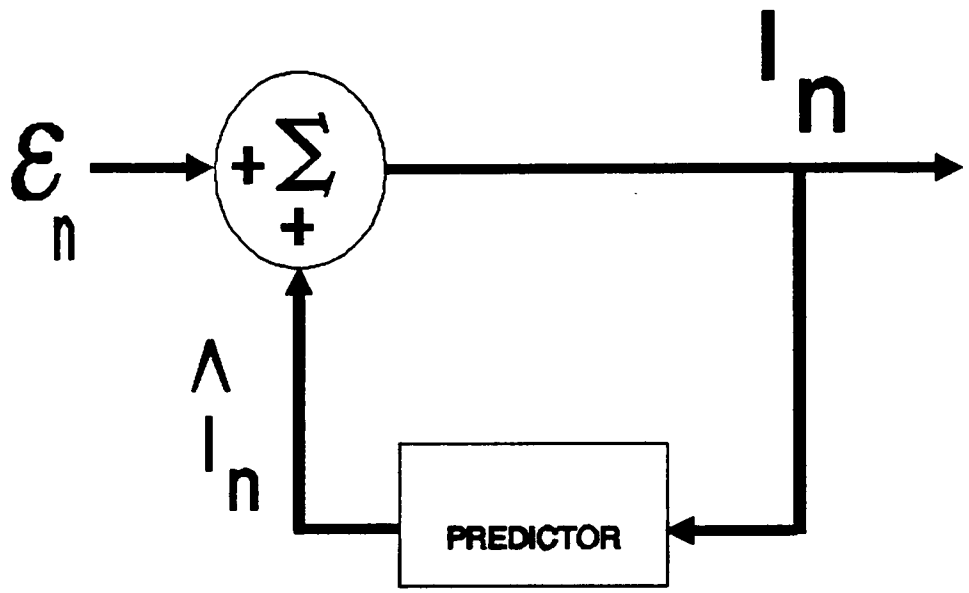

Figure 4 Inverse Filter Implementation

error from the Huffman decoder. The predictor in the inverse filter is the same one used in the filter section of the encoder stage. The input pixel value, I_n is reconstructed by

$$I_n = \epsilon_n + \hat{I}_n \qquad (15)$$

where $\hat{I}_n$ is given in (1). The inverse filter uses the same

predictive coefficients as the source encoder to realize an all-pole recursive system. The inverse filter receives a decoded prediction error and adds it to the predicted value of the pixel. For practical implementation, the operations in (15) can be carried out by

$$I_n = \epsilon_n + \sum_{i=1}^{M} q(h_i I_{n-i}) \qquad (16)$$

where $h_0=1$ and $h_i=-a_i$ for $i > 0$, and I_n is the restored pixel. Once the mapping is complete, the output will be the restored image file.

CHAPTER III

IMPLEMENTATION OF LPC-HUFF COMPRESSION SYSTEM

1. LPC_HUFF COMPRESSION SYSTEM

This thesis examined the problem of reducing the amount of space required to store an image while retaining all the information contained in an image. A numerical study of the LPC_HUFF method was applied to four representative 512 X 512 8-bit gray-scale images supplied by Data Translation. The subjects of the four images are listed below.

Image #1 - Statue of Liberty

Image #2 - earth

Image #3 - apple and orange

Image #4 - one dollar bill

This chapter explains the programs written to implement the encoder and decoder blocks in Figure 1. Several C programs were developed to implement the two step process outlined in Figure 1: **LNR_PC** (Appendix A), **HISTO** (Appendix B), **HUFF_COMP** (Appendix C), and **HUFF_EXPAND** (Appendix D). The *Filter* function of the **LNR_PC** program implements the linear prediction filtering technique described earlier. **HISTO** calculates the maximum and minimum values, the standard deviation, and the entropy of each filtered image. The **HUFF_COMP** program generates the Huffman code words for the prediction errors produced by the **LNR_PC** program. After the

code words are generated, **HIST_COMP** encodes the filtered file using the code words; the resulting file is a compressed image. In addition, the **HISTO_COMP** program and the *Inv_filter* subroutine of the **LNR_PC** program were written for the decompression step. **HUFF_COMP** decodes the prediction errors from a Huffman encoded file. *Inv_filter* recovers the original image file from a linear predictive filtered file.

2. SOURCE ENCODER BLOCK

The first stage in the LPC_HUFF method was designed to implement the source encoder (Figure 1) using the linear predictive filtering technique described in Chapter II. The **LNR_PC** program was written to achieve this objective. The steps involved in filtering an image are

* calculate optimum predictor coefficients
* calculate prediction errors using linear predictive filter

The filter block in Figure 2 was implemented with three functions: *Covar*, *Jacobi*, and *Filter*.

The *Covar* and *Jacobi* functions execute the numerical calculations needed to determine the optimum prediction coefficients. The coefficients were calculated using eigenanalysis and a least squares technique commonly used in spectral estimation[17-19]. The *Covar* function treats the image matrix as a 1-D vector by row scanning the matrix and appending the rows successively, and calculates the

autocorrelation matrix of the source image. The *Jacobi* function performs Jacobi diagonalization of the matrix generated by *Covar* to determine the eigenvalues; the minimum eigenvalue identifies the eigenvector with the optimum predictor coefficients. This function is a modified version of the Jacobi diagonalization routine contained in Numerical Recipes[26]. The eigenvalues of the autocorrelation matrix

$$A = I^T I \tag{17}$$

are returned in the diagonal elements of the original covariance matrix, and the corresponding eigenvectors are stored as the columns of the V matrix. After the eigenvalues are found, the program locates the smallest eigenvalue; identifying this column to select for the eigenvector. The segment of code from the *Jacobi* function finds the minimum eigenvalue from the diagonal elements of the *A* matrix and selects the appropriate eigenvector from the *V* matrix.

```
Min=A[0][0];
DOFOR(i,order) {
     DOFOR(i,order) {
          sum=A[i][i];
         Min= min(sum,Min);
          if(Min==sum)
               JCOL=i;
     }
}
```

The selected vector is normalized and stored in the *H* array. The following lines normalize the coefficients so that the lead coefficient is unity.

```
DOFOR(i,order) {
     H[i]=V[i][JCOL]/V[0][JCOL];
     }
```

The eigenvector selected and normalized yields the optimum predictor coefficients. Results for the four representative images used in the numerical study are discussed in the next chapter.

2.1 Filter Implementation - LPC

During the first stage of the compression system, a representative 512 X 512 8-bit image is filtered using LPC. The *Filter* function filters an image using a predictor with coefficients from the Jacobi-diagonalization routine. First, a pixel value, I_n, is read from a data file. Next, the prediction error is calculated using (3). The resulting value is stored in the variable *y* in the *Filter* function. This operation is implemented in the *Filter* function with the following lines of codes.

```
while(fin.read((unsigned char *) &imp,sizeof(imp))){
     for(int i=0;i<512;i++) {
          sum=0;
          for(int j=0;j<order+1;j++) {
               diff=imp[i-j]*A_COEFF[j];
               sum += int(diff);
          }
          y[i]=sum;
     }
```

In a practical implementation of the filtering process, the prediction errors are calculated one at a time. To implement the process the initial conditions are set to zero for each line of the image. Once a line of pixels (512 samples) is filtered, the results are written to an output

file.

In the lines of code the original pixels of the image, I_n, are stored in the *imp* array and the coefficients are stored in *A_COEFF* array. When index j is greater than i,

$$I_{i-j} = 0 \qquad (18)$$

because of the initial conditions. The i^{th} prediction error is stored in y_i. After each line of errors is generated, the y array is written to a data file. The output of this program is a filtered image comprised of the generated prediction errors. The filtered file is then sent to the second stage of the compression process.

2.2 Histogram and Entropy Calculations

As illustrated in Figure 1 the second step Huffman codes the prediction errors in the filtered file. Since statistical redundancy in an image can be reduced or eliminated by taking advantage of an image's statistical properties, the histogram, the maximum and minimum values, and the standard deviation were examined to study the statistical properties of the filtered image. The entropy was also calculated to determine the theoretical compression achievable. **HISTO** performs all these functions. **HISTO** consists of two subroutines: *Histogram* and *Entropy*.

The main program calls the *Histogram* function and scans through the image data to determine the frequency count of

each pixel value between -32768 and 32767. This function divides the range into 8 divisions so that 8192 values are being tracked at one time. A prediction error is stored in the variable *pred_err*. If *pred_err* is within the range of numbers being evaluated, then the frequency count is incremented by one and stored in the array *freq_cnt* at the location defined by *index*. The sum of frequency counts is stored in the variable *sum*. This process is implemented in the *Histogram* function by the following code. These results are presented in Chapter III.

```
while(fin.read((char *)&pred_err,sizeof(int))) {
     index = pred_err-start;
     if((pred_err >= start) && (pred_err <= end)) {
          freq_cnt[index]+=1;
          sum+=freq_cnt[index];
     }
}
```

This program also identifies the prediction errors with the largest and smallest frequency values. This information is stored in the variables *MAX* and *MIN*. After the frequency counts for a given range are determined, the values are written to a data file.

After the frequency counts are calculated, the *Histogram* function calls *Entropy*. Using the frequency counts and the sum of frequency counts, the line of code below calculates the histogram of a filtered image with the result stored in *p_histo*. The probability of the i^{th} prediction error is

p_histo=double(freq_count)/(double(freq_sum)).

The probabilities calculated are used by the *Entropy* function

to calculate the standard deviation. From the histogram information the entropy is calculated using (13). The self-information of the current prediction error is stored in the variable *result* and the entropy of the filtered file is stored in the variable *H*. The entropy as defined in (13) was implemented with the following lines of code.

```
result = log10(p_histo)/(log10(2));
H+=(-1)*(p_histo)*result;
```

The standard deviation of the prediction errors was also determined in this function to further analyze the statistical properties of the prediction errors. Since the data (the filtered image) was assumed to have a Gaussian distribution with zero mean (an assumption verified by the results presented in Chapter IV), the standard deviation can be calculated by the square root of the second order moment by

$$SD = \sqrt{\sum_{k=1}^{N} (X_k)^2 P_k} \,, \tag{19}$$

where X_k is the k^{th} prediction error value and P_k is the probability of that error in the image. The standard deviation was calculated using the following lines of code

```
while(fin.read((char *)&pixel_val,sizeof(int)))
     {
     fin.read((uchar*)&freq_count,sizeof(double));
     var+=double(pixel_val*pixel_val)*p_hist;
     }
SD=sqrt(var);
```

In this segment of the code *pixel_val* is the k^{th} prediction error and *p_histo* is the probability of that error. The entropy, standard deviation, and the maximum and minimum values of the prediction errors were generated for each filtered image. These results are presented in Chapter IV.

2.3 Symbol Encoder - Huffman Encoding

The second step in the encoding process is the symbol encoder. This step was implemented by the **HUFF_COMP** program. The Huffman coding process can be divided into four stages.

* Calculation of source probabilities
* Creation of the Huffman Tree
* Generation of Huffman Codes
* Encoding of source file using Huffman Codes

As noted earlier Huffman coding constructs an encoding tree to generate code words. In constructing the tree, it is only necessary to keep links from each node to its father and an indication of whether each node is a left son or right son; the information used to construct the tree is stored in the variable *nodes* which is a NODE structure. The NODE structure has four fields: *father*, *sontype*, *freq*, and *nextroot*. The *father* field is a pointer to the father node of the current node. If the current node is the root node, its father field is nil. The value of *sontype* is one of the constants *l* or *r*, depending on whether the node is a left son or right son; this

field determines whether the node represents a 0 bit or 1 bit. The *freq* field stores the frequency of occurrence of the symbol represented by that node. The *nextroot* field identifies which node the current node is linked to. The Huffman algorithm takes source probabilities as inputs and generates Huffman codes.

The distribution of the prediction errors was assumed to be zero-mean Gaussian as done in [16]. The validity of this assumption is verified by the results presented in Chapter IV. If the data is zero-mean Gaussian, the standard deviation is the only parameter needed to generate the source probabilities. Thus, eliminating the need to store the source probabilities or the code words.

The source probabilities are generated by

$$P_k = C1 \exp(-C2) \tag{20}$$

where C1 and C2 are defined as followed:

$$C1 = \frac{1}{\sigma\sqrt{2\pi}}$$
$$C2 = \frac{k^2}{2\sigma^2} . \tag{21}$$

These probabilities are stored in *freq* field of *nodes* (a NODE structure). The source probabilities for k values from -2.5σ (rounded to nearest integer) to 2.5σ (rounded to the nearest integer) are generated in increasing order. Thus N source probabilities are generated where N is

5σ. (22)

Code words are only generated for values within the range defined by $(-2.5\sigma, 2.5\sigma)$. When a prediction error that is encountered is outside the range $(-2.5\sigma, 2.5\sigma)$, then the code word for 2.5σ is used followed by the actual prediction error.

The following lines of code created the leaf nodes of the Huffman tree by computing the Gaussian probabilities of the prediction errors with standard deviation *SD*.

```
c1=1.0/(sqrt(2*M_PI)*SD);
c2=1.0/(2*SD*SD);
j=-N/2;
while(j<0) {
     Prob=((j*j*c2));
     Prob=c1*exp(-Prob);
     nodes[i].freq=Prob;
     nodes[i+1].freq=Prob;
     sum+=2*Prob;
     j++;
     i+=2;
}
nodes[N].freq=c1;
sum+=c1;
```

The sum of the probabilities is accumulated in *sum*. The sum of the discrete probabilities is normalized exactly to unity by dividing the probabilities by *sum*. This step is achieved using the following lines of codes.

```
for(j=1;j<=N;j++)
     nodes[j].freq/=sum;
```

After the source probabilities are calculated, the Huffman tree is constructed. The encoding tree is built according to the process outlined in Chapter II and using the following code.

```
for(int m=N+1;m<2*N;m++) {
```

```
n1=firstroot;
n2=nodes[n1].nextroot;
nodes[m].freq=nodes[n1].freq+nodes[n2].freq;
nodes[n1].father=m;
nodes[n1].stype ='l';
nodes[n2].father=m;
nodes[n2].stype='r';
```

Since the probabilities were generated in increasing order, the first one generated will have the lowest probability and the second one will have the second lowest probability. In the program, *n1* is designated the node with lowest probability and *n2* is the node with the second lowest probability. These two nodes are summed to form node *m*, the index of the new node being created. This sum, the probability of the newly created node *m*, is stored in the variable *nodes[m].freq*. Node *m* is the parent node of nodes *n1* and *n2*, and this information is stored in the father field of the *nodes* structure. The stype field designates a sibling node as a left son or right son.

After a node is created, the program uses variables *frq*, *firstroot*, and *found* to position the probability of node *m* in the probability list of available nodes. This sorting routine was implemented with the following lines of code.

```
frq=nodes[m].freq;
firstroot=nodes[n2].nextroot;
kk=0;
found=FALSE;
j=firstroot;
while((j!=0) && !(found)) {
     if(nodes[j].freq >= frq)
          found = TRUE;

     else {
          kk=j;
          j=nodes[j].nextroot;
     }
}
```

```
        if(kk==0)
             firstroot = m;
        else
             nodes[kk].nextroot = m;
        nodes[m].nextroot = j;
    }
```

This routine determines if the probability of the newly created node is less than the next probability in the probability list of available nodes. Each time a new node is created the probability list of available nodes is updated and reordered so that the values will be in increasing order. These steps are repeated until the root node is reached.

Once the Huffman tree, is constructed the next step is to generate the Huffman code words. A structure called CODE is used to store code words and related information. CODE has four fields: *wrd*, *lbits*, *index*, and *inval*. The *wrd* field is an array containing the Huffman code. The variable *lbits* stores the number of bits in a code word and *index* tells which prediction error the code word represents. The *inval* field holds the integer representation of a code word. The algorithm starts at the leaf node, reads the bit associated with that node, and this information is retrieved from the *stype* field of variable nodes. The bit retrieved is temporarily stored in *IBTMP* and then permanently stored in the *wrd* field of the CODE structure. Next the father node of the current node is called and the bit in its *stype* field is read. This process is continued until the root node is reached.

In this program the information pertaining to the code words is stored in the variable *cd* (a CODE structure). The

process outlined for generating the code words was implemented in the following lines of code.

```
for(i=1;i<N+1;i++)
     {
     j=i;
     J=0;
     L=0;
     while((nodes[j].father !=0) && (L < 16)) {

          if(nodes[j].stype =='l') IBTMP[J]=0;
          if(nodes[j].stype =='r') IBTMP[J]=1;
          L++;
          j=nodes[j].father;
          J++;
     }
     if(nodes[i].index<0)
          indx=N/2+abs(nodes[i].index);
     else
          indx=nodes[i].index;
     cd[indx].Lbits=L;
     cd[indx].index=index;
     for(int I=0;I<L;I++)
          cd[indx].wrd[I]=IBTMP[I];
     L=0;
}
```

After the integer representation of each code word is stored in the *inval* field of *cd*, the filtered file is encoded with the variable length Huffman code words. When a prediction error is read from the filtered file, it is stored in *pred_err* and then its position in the *cd* structure is determined by the *index* calculation. Using the value in *index* the Huffman code word representing *pred_err* is located. The following fragment of code implements the Huffman encoding process.

```
while(!(fin.eof())) {
     fin.read((char *)&pred_err,sizeof(int));
     if(pred_err<0)
          index = N/2 + abs(pred_err);
     else
```

```
            index=pred_err;

        /* Encoding of prediction errors for
           the range (-2.5σ,2.5σ) */

           if((pred_err > -N/2) && (pred_err < N/2)) {
              Len_bits=cd[index].Lbits;
             X=IOUT(cd[index].inval,Len_bits,0,X);
             num_bits+=Len_bits;
        }
```

If the prediction error is in the range

$$(-2.5\sigma, 2.5\sigma) \qquad (23)$$

the Huffman code for the retrieved prediction error is selected, and the *IOUT* function (Appendix C) is called. Using the selected code word which is in the *inval* field of the variable *cd*, and *L*, the number of bits in the code word, *IOUT* writes the code word to an output file. Prediction errors outside the range are also coded in the while loop listed above. However, when the prediction error is outside the range defined in (23), the code word for the prediction error for 2.5σ is used and the *IOUT* function is called and writes out the prediction error for 2.5σ. The actual 16-bit prediction error is also written to the file using the *IOUT* function. The fragment of code used to encode prediction errors outside the range (23) is shown below.

```
if((pred_err <= -N/2) || (pred_err >= N/2)) {
     L=cd[N/2].Lbits;
     X=IOUT(cd[N/2].inval,L,0,X);
     L+=16;
     X=IOUT(pred_err,16,0,X);
     num_bits+=L;
     }
```

The second stage is completed when all the prediction errors in the filtered file are Huffman coded. The output of this final stage is a compressed file of the original image.

3. SOURCE DECODER BLOCK

The second stage in the LPC_HUFF method was designed to implement the source decoder in Figure 1. The source decoder was implemented to verify that the LPC_HUFF method is lossless. There were two routines written to implement the decoder which is responsible for decompressing an image.

3.1 Symbol Decoder Implementation

The first block, the symbol decoder, was implemented using the **HUFF_EXPAND** (Appendix D) program. The Huffman decoding process can be divided into three stages.

* Calculation of source probabilities
* Creation of the Huffman Tree
* Decoding of source file

As noted earlier, Huffman coding constructs an encoding tree to generate code words. The same tree that was generated during the encoding process is constructed during the decoding process. In constructing the tree during the decoding process, it is only necessary to keep links from each node to its sibling nodes since the tree is only traveled from root to the leaf. The information used to construct the tree is

stored in the variable *nodes* (a NODE structure). *Nodes* uses the same fields as previously discussed, but instead of a *father* field there is a *lson* and *rson* field. The *rson* field is a pointer to the current node's right sibling and lson is a pointer to the left sibling. The other fields (*sontype*, *freq*, and *nextroot*) used have the same meanings as defined in the **HUFF_COMP** program.

The encoding tree is built in the same manner as explained in the **HUFF_COMP** program. First using (20) and the standard deviation (stored uncoded at the beginning of the compressed file) the source probabilities are generated. Next the Huffman tree is created in the same manner as discussed previously for the symbol encoder. Using the source probabilities the symbol decoder constructs the identical tree as the encoder.

Once the tree is constructed, the Huffman decoding routine is invoked. The decoding function uses the *IN* function (Appendix D) to retrieve bits from the input file. *IN* retrieves code words a byte at a time and then reads *NBITS*, a requested number of bits, from that word. *IN* returns the current bit or bits read into the *x* variable. To decode a word the tree is traversed from the root node to a leaf node by moving down one branch for each bit retrieved. If the bit is a 1, the branch to the right son is taken and if the bit is a 0, the branch of the left son is taken. This process was implemented in the following lines of code.

```
while(NP>N) {
  x=IN(1,x);
     pos=x.pos;
     inval= x.inval;

     //****Select appropriate son
     if(x.inval==0)
          NP= nodes[NP].lson;
     else
          NP= nodes[NP].rson;
     }
```

In the above lines of code, this process is continued until the node pointer (*NP*) is equal to the root node of the tree. When the node pointer (*NP*) is less than or equal to root then a symbol representing a leaf node has been encountered. Next, the prediction error associated with the encountered code word is determined by examining the value in the *index* field of the nodes array. Then the code determines if the prediction error is outside or inside the range $(-2.5\sigma, 2.5\sigma)$. If *index* is equal to 2.5σ, then 16 more bits are retrieved from the input file. The retrieved bits stored in the variable *x.inval* constitute the actual prediction error. If *index* is not equal to 2.5σ, then the prediction error is equal to the value residing in *index*. These steps were implemented with the following lines of code.

```
index = nodes[NP].index;
     if(index==N/2) {

     x=IN(16,x);
          pred_err = x.inval;
     }
if(index !=N/2)
     pred_err=index;
ft.write((char *)&pred_err,sizeof(int));
NP=m-1;
```

Each decoded prediction error is stored in *pred_err* and written out to data file. After the prediction error is written to a file, then *NP* is reset to m-1. This process is continued until all the bits in the file have been decoded. The result of the **HUFF_EXPAND** is a file composed of decoded prediction errors.

3.2 Inverse Filter Implementation

After completing the first step of the decoding process, the filtered file is fed to the second and final block of the source decoder (Figure 1). The second block implemented is the inverse filter. This block was implemented with the *Inv_filter* function in the **LNR_PC** program. The *Inv_filter* subroutine takes a filtered image and recovers the original image. This function implements the process defined in (15) using the same filter coefficients as the source encoder to recover a pixel, I_n. To calculate a pixel value, a prediction error is read from an input file and then added to the calculated estimate of that pixel. For practical implementation the inverse filter step was implemented using (16) in the following lines of code.

```
fin.read((char *)&Y,sizeof(Y));
     for(int i=0;i<5\;i++) {
     sum=0;
     for(int j=1;j<order+1;j++) {
          if((i-j)<0)
               IM_X=0;
          else
               IM_X=X[i-j];
     pred =  float(IM_X)*A_COEFF[j];
```

```
            sum+=int(pred);
            }
            X[i]=int(Y[i]-sum);
        }
```

A line of coded prediction errors is read into the *Y* array. Using the prediction error stored in Y_i, the predictor coefficients in the *A_COEFF* array, and the calculated predicted value for the i^{th} term the restored pixel is calculated; the result is stored in the variable x_i. After a line of pixels is calculated, the *x* array is written to an output file. Upon completing the previously discussed steps, the *Inv_filter* function yields a restored image from the filtered data.

CHAPTER IV

RESULTS

The data compression system discussed in earlier chapters was implemented with the computer programs described in Chapter III. Four demo images from Data Translation were compressed and restored to examine the degree of compression achieved by the LPC_HUFF method. The predictor coefficients, the entropy, and the statistical properties for each image were studied. The compression ratio as defined in (14) for the four representative 512 X 512 8-bit gray scale images was determined.

The first step in the compression system was to determine the predictor coefficients for a given filter order. Tables III-VIII show the predictor coefficients calculated by **LNR_PC** program for filter orders one to six. The results show that the coefficients are nearly the same especially for lower order filters.

Using the coefficients from Tables III-VIII, the four images were filtered using the linear predictive coding technique. The entropy and standard deviation were calculated for the filtered images using (13) and (19); the maximum and minimum values were also calculated. The results from the **HISTO** program are recorded in Tables IX-XII. The

Table III First Order Coefficients

IMAGE 1	IMAGE 2	IMAGE 3	IMAGE 4
1.0	1.0	1.0	1.0
-1.0	-1.0	-1.0	-1.0

Table IV Second Order Coefficients

IMAGE 1	IMAGE 2	IMAGE 3	IMAGE 4
1.0	1.0	1.0	1.0
-1.993398	-1.998163	-1.999673	-1.995629
0.999993	1.000022	1.000055	0.999996

Table V Third Order Coefficients

IMAGE 1	IMAGE 2	IMAGE 3	IMAGE 4
1.0	1.0	1.0	1.0
-2.327604	-2.474115	-2.231204	-2.217172
2.327588	2.474145	2.231163	2.217176
-0.999984	-1.000031	-0.999958	-1.00004

Table VI Fourth Order Coefficients

IMAGE 1	IMAGE 2	IMAGE 3	IMAGE 4
1.0	1.0	1.0	1.0
-2.840482	-2.607832	-2.376825	-2.8256
3.684351	3.217314	2.753825	3.653012
-2.840522	-2.607876	-2.376923	-2.825602
1.000050	1.000063	1.00027	0.999992

Table VII Fifth Order Coefficients

IMAGE 1	IMAGE 2	IMAGE 3	IMAGE 4
1.0	1.0	1.0	1.0
-3.086557	-2.892426	-2.541631	-2.868306
4.776444	4.068151	3.292157	4.363214
-4.7767	-4.068181	-3.2922	-4.363147
3.087029	2.892605	2.542004	2.868174
-1.000217	-1.000149	-1.000329	-0.999933

Table VIII Sixth Order Coefficients

IMAGE 1	IMAGE 2	IMAGE 3	IMAGE 4
1.0	1.0	1.0	1.0
-3.35112	-3.080246	-2.541631	-3.206948
5.779202	4.853637	3.551715	5.368147
-6.8540333	-5.545303	-3.9555392	-6.321101
5.780296	4.85136	3.551318	5.36802
-3.352375	-3.079507	-2.57307	-3.206857
1.000527	0.999635	0.999675	0.999953

Table IX Statistical Parameters for Image 1

Order	Min	Max	σ	Entropy
1	-188	194	13.597199	4.740608
2	-381	194	12.552493	4.860043
3	-444	445	10.209376	4.87131
4	-541	705	11.399358	5.154249
5	-899	912	13.601923	5.469657
6	-1295	1100	17.581282	5.900029

Table X Statistical Parameters for Image 2

Order	Min	Max	σ	Entropy
1	-188	236	8.033687	4.646937
2	-381	236	7.014855	4.693917
3	-473	472	6.660317	4.726762
4	-496	615	7.566772	4.853733
5	-766	775	9.114641	5.157519
6	-1047	922	11.117703	5.528665

Table XI Statistical Parameters for Image 3

Order	Min	Max	σ	Entropy
1	-188	194	5.810222	3.614082
2	-381	194	6.190578	3.963563
3	-426	428	6.748961	4.155462
4	-452	526	7.533533	4.36786
5	-616	627	9.19483	4.689556
6	-743	674	10.882581	4.960429

Table XII Statistical Parameters for Image 4

Order	Min	Max	σ	Entropy
1	-188	194	11.765783	5.083556
2	-382	194	11.587214	5.201215
3	-423	424	9.145866	4.902832
4	-539	699	11.320539	5.178499
5	-820	833	12.984048	5.322666
6	-1194	1020	16.209242	5.739165

tables show these statistical parameters as function of the filter order for the four filtered images. The entropy result in this case indicates that when applying the LPC_HUFF technique to the four images the entropy increases with an increase in the filter order except for one case.

This trend is caused by outlying values that occur in the histograms of the filtered images due to the zero initial conditions assumed in the row-by-row processing. The standard deviation follows the same trend as the entropy, and the maximum and minimum values also increase with an increase in filter order. As indicated in Tables III-VIII the magnitudes of the prediction coefficients increase with increases in filter order. Since the magnitudes increase with filter order, initial sample values in each row of the output also increase. Thus, the magnitude of the statistical parameters increases as the length of the impulse responses increases. The result is illustrated in the histograms in Figures 5-8. Figures 5-8 illustrate the outlying values caused by the transient response.

The prediction errors were assumed to have a Gaussian distribution with zero mean. To verify this assumption the histograms for the filtered images were calculated. This trend was exhibited for all the filtered images. Figures 5-8 demonstrate the Gaussian nature of the prediction errors as well as the large outlying values discussed previously. These figures show the histogram results produced by the **HISTO**

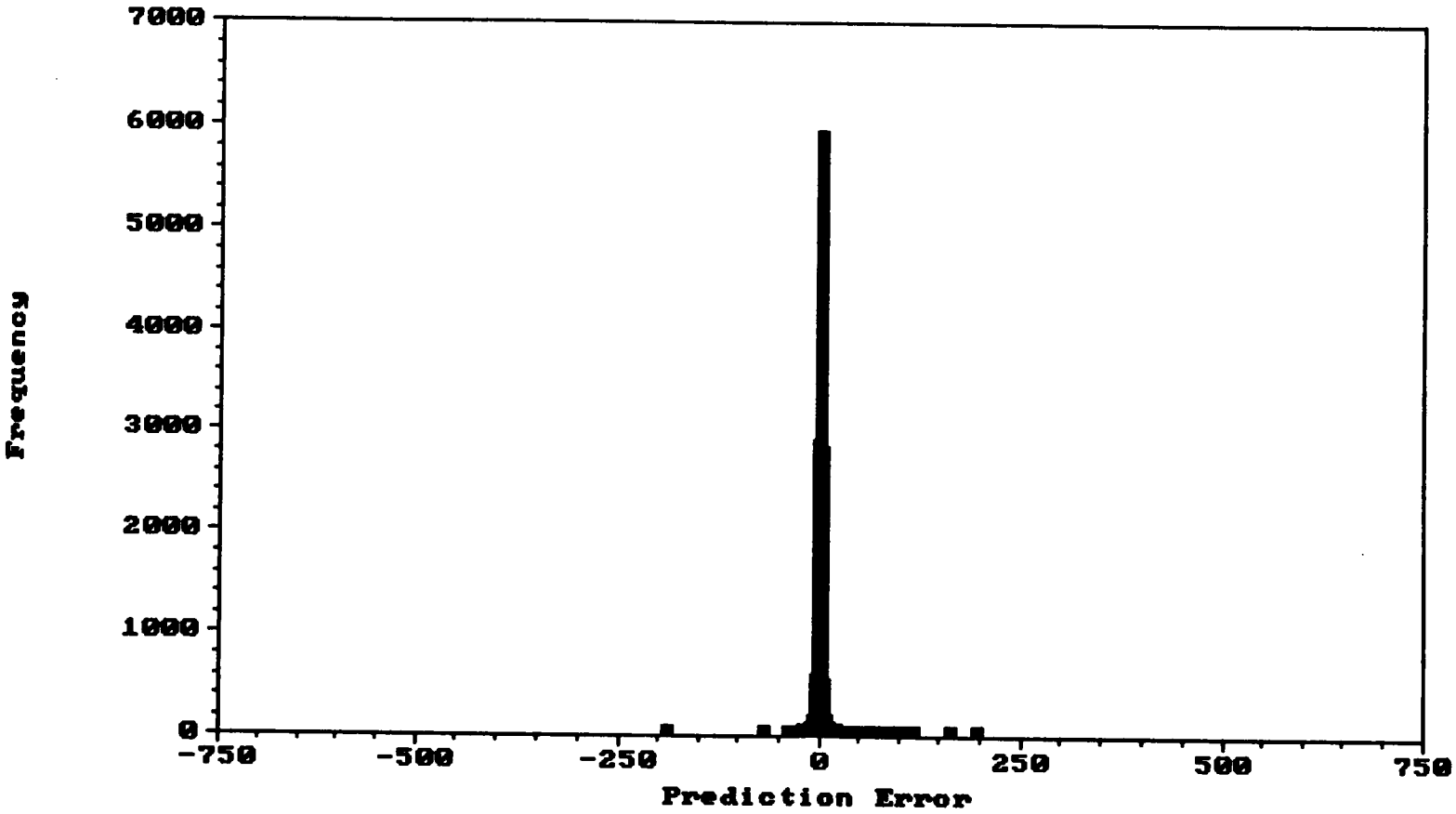

Figure 5 Histogram For Filtered Data - First Order

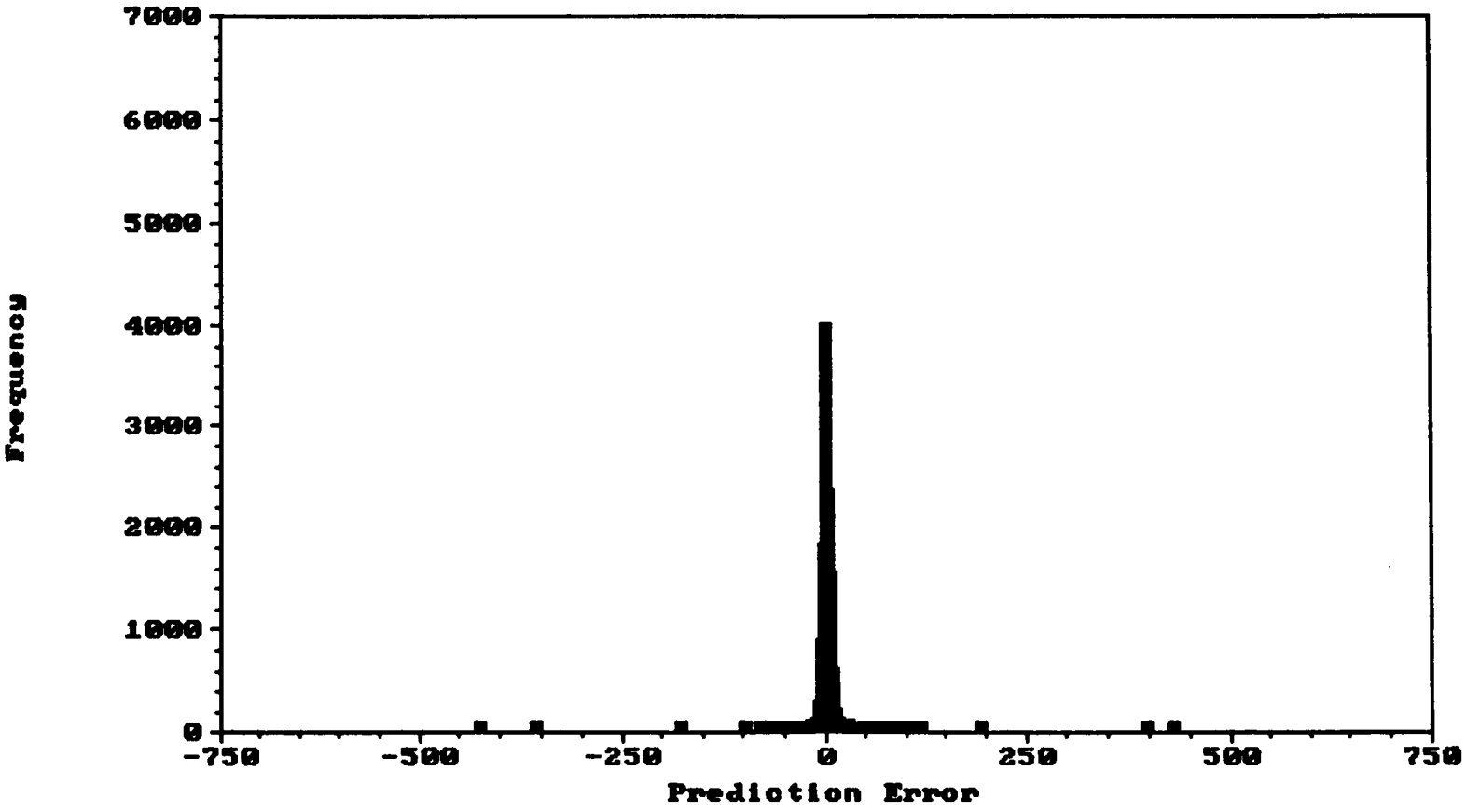

Figure 6 Histogram For Filtered Data - Third Order

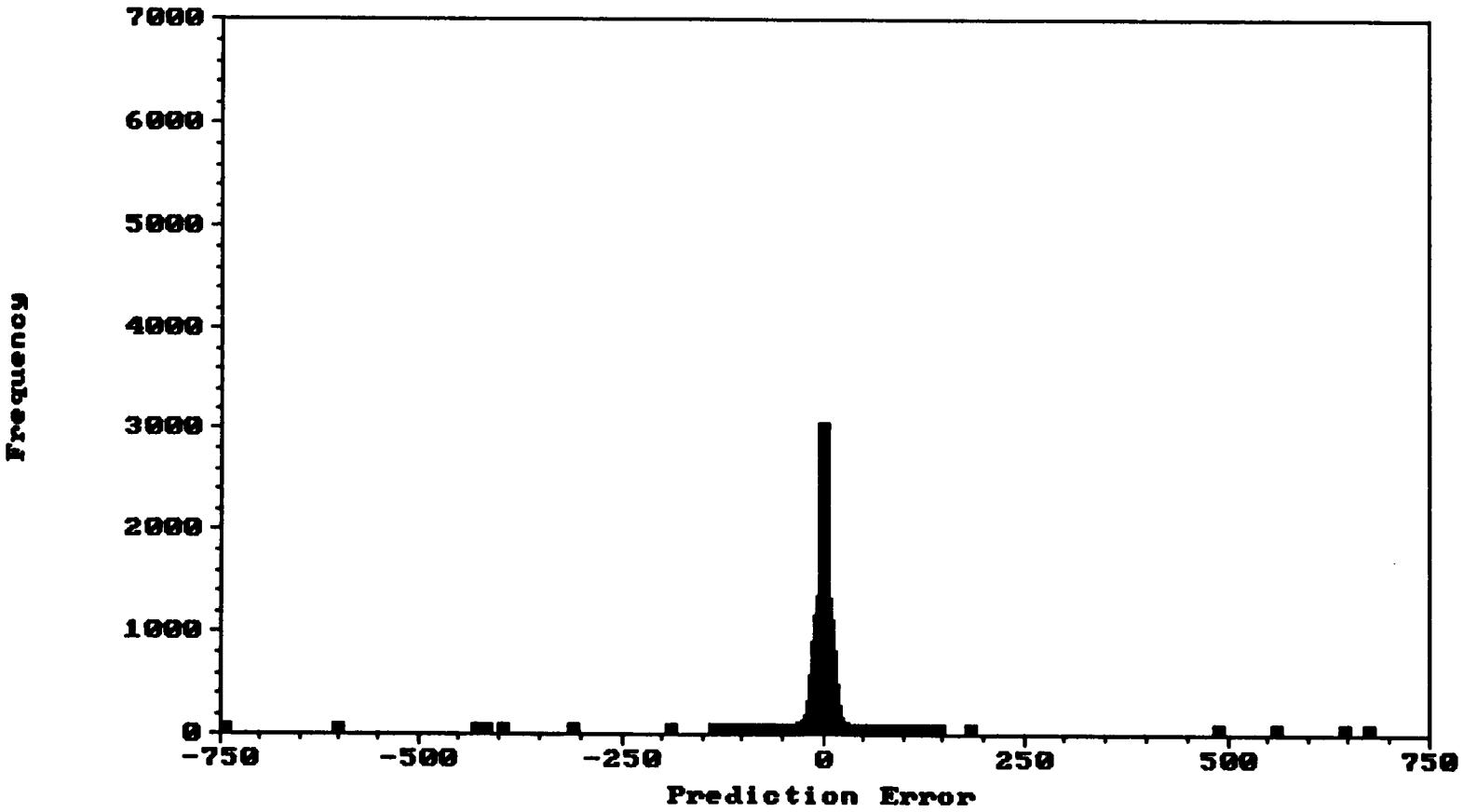

Figure 7 Histogram For Filtered Data - Sixth Order

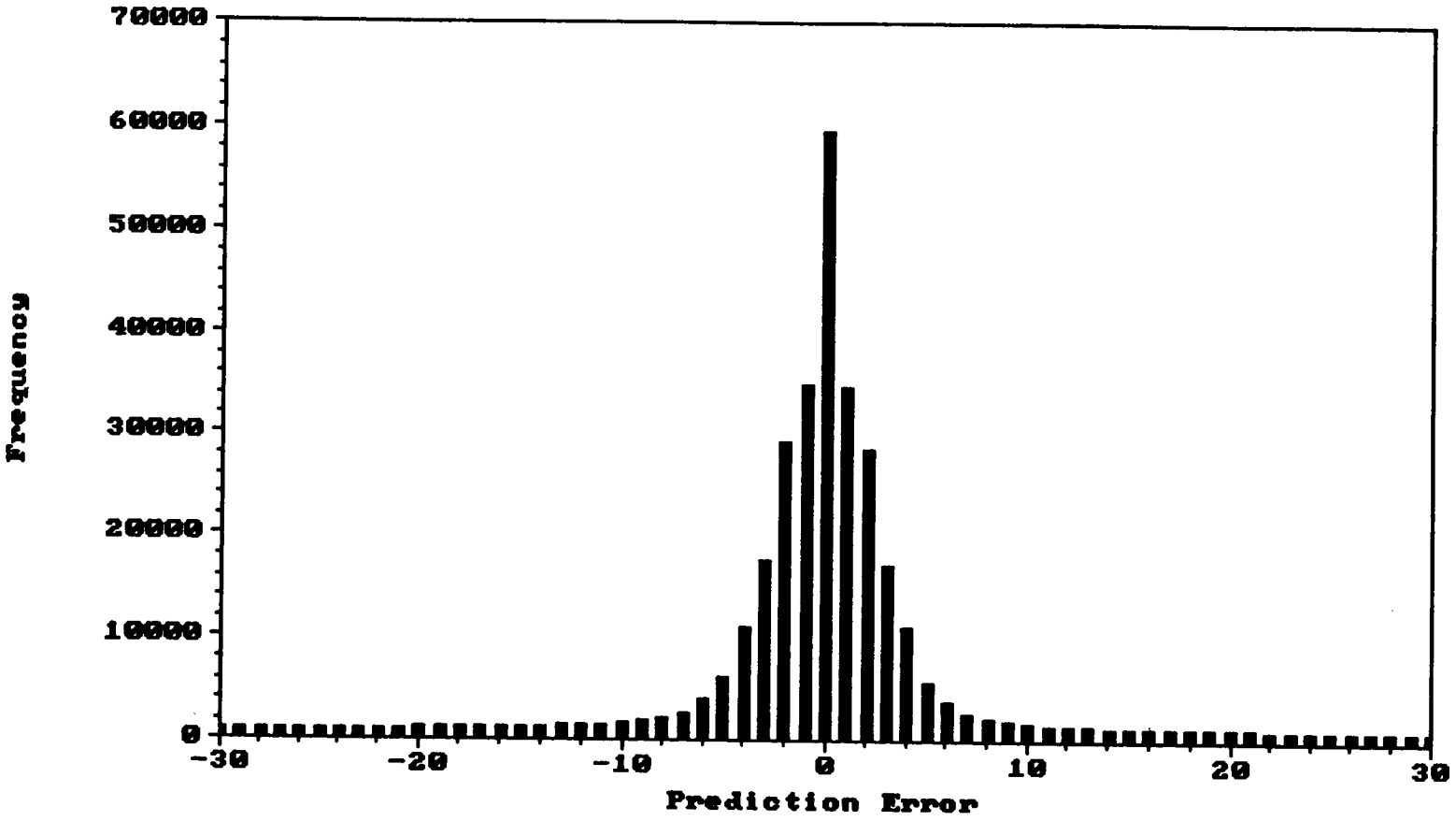

Figure 8 Histogram For First Order - Closeup View

program for Image 3. Since the result is similar for all the filtered images, only the results for demo Image 3 are shown. Figures 5-7 show the histograms for filter orders 1, 3, and 6. The frequency values are scaled by 10% and translated by 100 so that the very small levels can be viewed. Since a closeup of the histogram is similar for all the filtered images, only the result for M=1 for Image 3 is shown in Figure 8. To view the histograms close up, the x axis is defined only for the prediction errors in the range -30 to 30.

Once the images were filtered using LPC, the resulting files were encoded using the Huffman technique. When a image was applied to the steps defined in the encoder block (Figure 1), a compressed image was produced. Next the compression ratio was calculated for the compressed images. Table XIII shows the results for the four compressed images. Decompression of these data files was also carried out to verify the ability of this method to reconstruct the original images without loss.

Table XIII Compression Ratio for Compressed Images

Image#	CMPR
1	.7619
2	.7547
3	.5429
4	.7617

CHAPTER V

SUMMARY AND CONCLUSION

A numerical study of an error-free image compression method for reducing statistical redundancy in an image was performed in this study. Four 512 X 512 8-bit representative image were processed by the compression system illustrated in Figure 1.

The compression system studied used linear prediction cascaded with Huffman coding (LPC_HUFF method) to compress the images. Four programs were written to implement the two block system: **LNR_PC, HISTO, HUFF_COMP**, and **HUFF_EXPAND**. The histogram and the statistical properties of each filtered image was calculated and examined. Further information about the filtered file was gained by calculating the entropy to examine the theoretical compression achievable.

The coefficients were found to be nearly identical especially for lower order filters. Initially it was assumed that the prediction errors (the filtered image) had a zero-mean. Since the histogram results indicated that the prediction errors had a Gaussian distribution, the standard deviation was calculated to generate the source probabilities. The entropy increased with increases in filter order except for one case. The increase in magnitude of coefficients as the length of the response increased caused an increase in the

magnitudes of the statistical parameters studied resulting in the presence of outlying values in the histograms due to the transient response. Since the entropy results in these cases revealed that a simple differencing approach was optimal, it was decided to use a first order filter in the compression process. After generating the Huffman code words for the prediction errors, the filtered files were Huffman coded resulting in a compressed image.

The second stage of the compression system was responsible for recovering the original image from a compressed image. The decoder block like the encoder block was implemented as a two step process (performing the inverse functions of the encoder block). The symbol decoder implemented the Huffman decoding operation and the inverse filter recovered the original pixel values from a filtered file.

Using LPC and Huffman coding statistical redundancy in the image was reduced to achieve oppression. The system was successfully implemented thereby achieving error-free compression. The compression system studied reduced file storage requirements; the degree of data reduction (1 - compression ratio) ranged from 22% to 46% for the four images compressed.

The need for image compression will continue to rise as the use of image data becomes commonplace. The LPC_HUFF method was studied to see if the cascaded method would

compress image data error-free. Possible improvements may be achieved using this method by using a different method of linear prediction. A 1-D predictor with the last M pixels from the previous line could be used. The scanning of the image would go left-to-right then right-to-left alternately. Thus, in this design only the first row would have zero initial conditions. Another approach is to examine the effectiveness of a 2-D predictor. In 2-D prediction the prediction would be a function of the previous pixels in a left-right, top-to-bottom scan of an image[4]. Further work in the area of image compression will be required because the source of the data determines which method offers the best balance between the transmission and storage requirements and the minimal amount of data required to make a decision.

REFERENCES

REFERENCES

[1] R. C. Gonzalez and P. Wintz, *Digital Image Processing*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1992.

[2] D. A. Lelewer and D. S. Hirschberg, "Data Compression," ACM Computer Surveys, vol. 19, No. 3, pp. 262-296, September 1987.

[3] W. F. Schreiber, *Fundamentals of Electronic Imaging Systems*. New York: Springer-Verlag, 1986.

[4] R, A. Haddad and T. W. Parsons, *Digital Signal Processing: Theory, Applications and Hardware*. New York: Computer Science Press, 1991.

[5] A. N. Netravali and J. O. Limb, "Picture Coding: A Review," Proc. of the IEEE, vol. 68, No. 3, pp. 366-406, March 1980.

[6] A. K. Jain, "Image Data Compression:A Review," Proc. of the IEEE, vol. 69, No. 3, pp. 349-389, March 1991.

[7] T. J. Lynch, *Data Compression Techniques and Applications*. New York: Van Nostrand Reinhold, 1985.

[8] J. Makhoul, "Linear Prediction: A Tutorial Review," Proc. of the IEEE, vol. 63, No. 4, pp. 561-580, April 1975.

[9] P.L Poehler and J. Choi, "Linear Predictive Coding of Imagery for Data Compression Applications," IEEE ICASSP 83, vol. 3, pp. 1240-1243, 1983.

[10] H. Kobayaski, and L. R. Bahl, "Image Data Compression I: Prediction Algorithms," IBM J. Res. Develop., vol. 18, No. 172, pp. 164-171, 1974.

[11] P. Elias, "Predictive Coding," IRE Trans. on Inform. Theory, vol. IT-1, pp.16-33, March 1955.

[12] C. G. Boncelet, J. R. Cobbs, and A. R. Moser, "Error Free Compression of Medical X-ray Image," Proc. of SPIE, vol. 1001, pp. 269-276, 1988.

[13] N. S. Jayant and Peter Noll, *Digital Coding of Waveforms*. Englewood Cliffs, NJ: Prentice Hall, 1984.

[14] P. H. Ang, P. A. Ruetz, and D. Auld, "Video Compression Makes Big Gains," IEEE Spectrum, vol. 10, pp.16-19,

October 1991.

[15] R. N. Williams, *Adaptive Data Compression*. Boston, MA: Kluwer Academic Publishers, 1991.

[16] O. A. Adeyemi, "Analysis and Comparison of Three Techniques for the Compression of Digital Instrumentation Data." Master's Thesis, Dept. of Elec. Engr., The Univ. of Tenn., TN, 1990.

[17] S. M. Kay, *Modern Spectral Estimation: Theory & Application*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[18] S. L. Marple, Jr., *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice Hall, 1987.

[19] S. M. Kay and S. L. Marple, Jr., "Spectrum Analysis - A Modern Perspective," Proc. of the IEEE, vol. 69, No. 3, pp. 349-389, November 1981.

[20] R. G. Gallager, "Variations on a Theme by Huffman," IEEE Trans. on Info. Theory, vol. IT-24,8, pp. 668-674, November 1978.

[21] J. Amsterdam, "Data Compression With Huffman Coding," Byte Magazine, pp. 99-108, May 1986.

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

[23] G. Held, *Data Compression*. New York: John Wiley & Sons,Inc., 1991.

[24] S. Apiki, "Lossless Data Compression," Byte Magazine, pp.99-108, May 1986.

[25] A.M. Tenenbaum and M.J. Augenstein, *Data Structures Using Pascal*. Englewood Cliffs, NJ: Prentice Hall, 1981.

[26] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C*. New York: Cambridge University Press, 1988.

[27] M. Bricklin, "The Word," Heart & Soul Magazine, p. 96, Winter 1993.

APPENDICES

APPENDIX A

```
/*************************************************************
 *                                                           *
 *   Program Name: LNR_PC                                    *
 *                                                           *
 *     Input:        IMAGE FILE - 512 X 512 8-bit            *
 *     Output:       LPC CODED FILE                          *
 *                                                           *
 *     Input:        LPC CODED FILE                          *
 *     Output:       IMAGE FILE - 512 X 512 8-bit            *
 *                                                           *
 *   Subroutines:  Covar, Jacobi, Filter                     *
 *                 Inv_Filter                                *
 *                                                           *
 *   This program encodes and decodes an image using         *
 *   a M order linear predictor.  It is executed by          *
 *                                                           *
 *            LNR_PC file1 file2 order file3                 *
 *                                                           *
 *   where file1 is the source image, file2 is the           *
 *   output file(filtered image),  order is the filter       *
 *   order, and file3 is used to store the predictor         *
 *   coefficients.                                           *
 *                                                           *
 *************************************************************


 ************************************************
 *                    FUNCTIONS                 *
 *   Covar - calculates covariance matrix       *
 *               from Image Data                *
 *   Jacobi - performs Jacobi                   *
 *            Diagonalization of the            *
 *            covariance matrix and             *
 *            calculates optimal predictior     *
 *            coefficients.                     *
 *   Filter - filters image using LPC           *
 *   Inv_filter - decodes  LPC compressed file  *
 *                                              *
 ************************************************
```

```
 ************************************************
 *                                              *
 *              Variables                       *
 *                                              *
 *    infile, outfile - input file,output files *
 *    coef_fil - I/O file for coefficients      *
 *    order - filter order                      *
 *    filemde - function selector               *
 *                                              *
 ************************************************

##########################################################/
//**** HEADER FILES****
#include <fstream.h>
#include <iostream.h>
#include <stdlib.h>
#include <limits.h>
#include <float.h>
#include <io.h>
#include <stdio.h>
#include <ctype.h>
#include <conio.h>
#include <math.h>
#include "dofor.h"
#include <iomanip.h>
#include <dos.h>

//****FUNCTION DECLARATIONS****
int Covar(char *infile,int order);
int Jacobi(char *infile,int order);
int Filter(char *infile, char *outfile,int order) ;
int Inv_filter(char *infile,char *outfile);


template <class T> T min(T x, T y)
    {
    return (x < y) ? x : y;
    }




int main() {

     //****VARIABLES****
     char *infile,*outfile, filemde, *coef_fil;
     int order;

     clrscr();
     order=atoi(_argv[3]);
```

```
      if(order<1) {
           cerr<<"order less than 1\n";
           return 1;
      }
      infile=_argv[1];
      outfile=_argv[2];
      coef_fil=_argv[4];

      if(_argc < 5) {
         cout<<"USAGE:"<<"LNR_PC <INFILE> <OUTFILE>
               <ORDER> <COEFFICIENT_FILE>\n";
         return 1;
      }

      //****CALCULATION OF COVARIANCE MATRIX****
      cout<<"enter C to calculate covariance matrix\n";
      cin>>filemde;
      if(toupper(filemde)=='C')
           Covar(infile,order);



      //****FILTERING OF DATA****
      cout<<"enter F to filter data\n";
      cin>>filemde;
      if(toupper(filemde)=='F')
           Filter(infile,outfile,order);



      //****INVERSE FILTERING OF DATA****
      cout<<"enter I to retrieve image
             from filtered file\n";
      cin>>filemde;
      if(toupper(filemde)=='I') {
           cout<<"enter input file\n";
           cin>>infile;
           Inv_filter(infile,"huff_expand");
      }

return 0;
}

/############################################################
/############################################################
```

```
/*****************************************************
 *                  **** FUNCTIONS ****               *
 *****************************************************



 ************************************************
 *              COVAR  FUNCTION                 *
 *                                              *
 * This  function generates the covariance      *
 * matrix for the image data.  The results      *
 * are stored in the phi matrix.                *
 *                                              *
 ************************************************


 ************************************************
 *                                              *
 *              Variables                       *
 *                                              *
 *    im_pel - source pixel(a pixel from the    *
 *             original image)                  *
 *    N - total number of pixels in the image   *
 *    phi - covariance matrix                   *
 *    f,i,j,k,sum - scratch variables           *
 *                                              *
 ************************************************/

int  Covar(char *infile,int order)

     {
     typedef unsigned char uchar;

     //****VARIABLES****
     char filemde;
     float phi[30][30];
     int i,j;
     long N;
     uchar im_pel;
     unsigned int f[30];
     unsigned long k, sum;

     //****IO_SETUP****
     ifstream fin (_argv[1],ios::app|ios::nocreate);
     ofstream fout (_argv[4],ios::trunc|ios::binary);
     if(!fin) {
          cerr<<_argv[1] <<" can not be opened file for
            input.";
          return(1);
     }
     N=262656;
     //****INITIALIZATION OF VARIABLES****
```

```
        for(int i=0;i<30;i++) {
             for(int j=0;j<30;j++)
                  phi[i][j]=0;
             f[i]=0;
        }

        //****CALCULATION OF COVARIANCE MATRIX****
        for(k=0;k<N;k++) {
             fin.read((uchar *)&im_pel,sizeof im_pel);
             f[0]=im_pel;
             for(i=0;i< int(order)+1;i++) {
                  for(j=i;j<order+1;j++)
                       phi[i][j]+= float(f[i]*f[j]);
             }
             sum=0;
             for(i=0;i< order+1;i++)
                  sum+=(f[i]*f[i]);
                  for(i=0;i<order;i++)
                       f[order-i]=f[(order-i)-1];
        }
        for(i=0;i<order+1;i++) {
             for(j=0;j<order+1;j++)
                  phi[j][i]=phi[i][j];
        }

        for(i=0;i<order+1;i++) {
             for(j=0;j<order+1;j++)
                fout.write((uchar *)&phi[i][j],sizeof( float));
        }

        jacobi(_argv[4],order);
        return 0;
   }


    //#########################################################
```

```
/************************************************
 *                   JACOBI FUNCTION            *
 *                                              *
 * This  function performs Jacobi               *
 * diagonalization of a N by N real symmetric   *
 * matrix A.  The eigenvalues of A are returned *
 * in the diagonal elements  and the            *
 * corresponding  eigenvectors in the columns of*
 * of the V matrix.                             *
 *                                              *
 ************************************************


 ************************************************
 *                                              *
 *              Variables                       *
 *                                              *
 *    A - eigenvalues                           *
 *    H - predictor coefficients                *
 *    order - filter order                      *
 *    V - eigenvector                           *
 *    C,IROW,JCOL,Min,sum - scratch variables   *
 *    i,j,DELTA,R,R_MAX,S,T - scratch variables *
 *                                              *
 ************************************************/



int  jacobi(char *ifile,int order) {

     //****VARIABLES****
     float DELTA = 0.0,T,Min,sum;
     float V[30][30],A[30][30],H[30];
     float R,C,S,R_MAX;
     int i,j,IROW,JCOL,order;

     //****IO_SETUP****
     ifstream fin (ifile,ios::binary|ios::nocreate);
     if(!fin) {
      cerr<<ifile <<" can not be opened file for input.";
     return(1);
     }
     ofstream fout ("W_JOEFF",ios::app|ios::binary);
     if(!fout) {
      cerr<<"W_JOEFF can not be opened file for input.";
     return(1);
     }

     ofstream ft("J_COEFF",ios::app|ios::trunc|ios::binary);

     //****INITIALIZATION OF VARIABLES****
     order+=1;
```

```
 DOFOR(i,30){
      DOFOR(j,30) {
           V[i][j]=0.0;
           A[i][j]=0.0;
      }
 }

 //****READING PREDICTOR COEFFICIENTS
 DOFOR(i,order) {
      DOFOR(j,order) {
           fin.read((uchar *) &A[i][j],sizeof(float));
      }
 }

 /****FIND CONVERGENCE VALUE AND****
  ****INITIALIZE V TO IDENTITY MATRIX****/
 DOFOR(i,order){
      DOFOR(j,order){
           DELTA+=A[i][j]*A[i][j];
           V[i][j]=0.0;
      }
      V[i][i]=1.0;
 }

DELTA = (1e-7*sqrt(DELTA))/float(order);
R_MAX=DELTA;

//****FIND LARGEST OFF-DIAGONAL ELEMENT****
while ( R_MAX >= DELTA) {
      R_MAX=0.0;
      DOFOR(i,order){
           DOBYYY(j,i+1,order) {
           T=fabs(A[i][j]);
           if(T<=R_MAX) goto again;
           R_MAX=T;
           IROW =i;
           JCOL = j;

      again:
      }
 }

 /* IF THAT VALUE IS GREATER THAN THE CONVERGENCE
     FIND C AND S TO ELIMINATE THAT ELEMENT */
 if(R_MAX > DELTA) {
      R = (0.5*(A[IROW][IROW] -
        A[JCOL][JCOL])/A[IROW][JCOL]);
      T = R + sqrt((1 + R*R));
      C = 1.0/sqrt((1.0 + T*T));
      S = (C*T);
```

```
            //****PRE-MULTIPLY A*****
            DOFOR(j,order) {
               R=A[IROW][j];
               T= A[JCOL][j];
               A[IROW][j]= C*R - S*T;
               A[JCOL][j]= C*T + S*R;
             }

             //****POST-MULTIPLY A AND V*****
             DOFOR(i,order){
               R = A[i][IROW];
               T = A[i][JCOL];
               A[i][IROW] = C*R - S*T;
               A[i][JCOL] = C*T + S*R;
               R = V[i][IROW];
               T = V[i][JCOL];
               V[i][IROW] = C*R - S*T;
               V[i][JCOL] = C*T + S*R;
             }
      }
     }

      //****FINDING A MINIMUM EIGENVALUE****
      Min=A[0][0];
      DOFOR(i,order) {
           DOFOR(i,order) {
           sum=A[i][i];
           Min= min(sum,Min);
           if(Min==sum)  JCOL=i;
           }
      }

      //****SELECTION OF EIGENVECTOR****
      DOFOR(i,order) {
           H[i]=V[i][JCOL]/V[0][JCOL];
           cout <<"H is "<<H[i]<<"\n";
           fout<<H("<<i<<") IS "<<H[i]<<"\n";
           ft.write((char *) &H[i],sizeof(float));
      }
      fout<<"\n";
      return 0;
}
//##########################################################
```

```
/************************************************
 *              FILTER FUNCTION                 *
 *                                              *
 * This  function implements linear predictive  *
 * filtering technique.                         *
 *                                              *
 ************************************************



 ************************************************
 *                                              *
 *              Variables                       *
 *                                              *
 *    A_COEFF - predictor coefficients          *
 *    imp - image pixel                         *
 *    order - filter order                      *
 *    y - prediction error                      *
 *    diff,sum - scratch variables              *
 *                                              *
 ************************************************/

int Filter(char *infile,char *outfile,int order)
     {

     //*****VARIABLES****
     float A_COEFF[30],diff;
     int  y[512], sum=0.0;;
     unsigned char imp[512];


     //****INITIALIZATION OF VARIABLES****
     for(int i=0;i< 512;i++)
          y[i]=0;

     //****IO_SETUP****
     ifstream fin ("J_COEFF",ios::binary|ios::nocreate);
     if(!fin) {
      cerr<<infile <<" can not be opened file for input.";
      return(1);
     }
     ofstream fout(_argv[2],ios::app
             |ios::binary|ios::trunc);
     order+=1;
```

```
        //****READING PREDICTOR COEFFICIENTS
        DOFOR(i,order) {
             DOFOR(j,order) {
                fin.read((uchar *) &A_COEFF[i][j],sizeof(float));
             }
        }

        if(fin.eof()) {
             fin.clear();
             fin.seek(0,ios::beg);
        }
        fin.open(_argv[1]);


        //****LINEAR PREDICTIVE FILTERING****
        while(fin.read((unsigned char *) &imp,sizeof(imp))){
             for(int i=0;i<512;i++) {
                  sum=0;
                  for(int j=0;j<size+1;j++) {
                       diff=imp[i-j]*A_COEFF[j];
                       sum += int(diff);
                  }
                  y[i]=sum;
             }

              fout.write((char *) &y,sizeof(y));
        }
        return 0;
}
//#########################################################
```

```
/************************************************
 *              INV_FILTER FUNCTION             *
 *                                              *
 * This  function implements linear predictive  *
 * decoding technique.                          *
 *                                              *
 ************************************************


 ************************************************
 *                                              *
 *              Variables                       *
 *                                              *
 *    A_COEFF - predictor coefficients          *
 *    order - filter order                      *
 *    X - resorted image pixel - I              *
 *    Y - prediction error                      *
 *    i,j,k,sum,IM_X - scratch variables        *
 *    pred - scratch variables                  *
 *                                              *
 ************************************************/

int Inv_filter( char *infile,char *outfile)
    {

     //****IO_SETUP****
     ifstream fin ("J_COEFF",ios::binary|ios::nocreate);
     if(!fin) {
          cerr<<"J_COEFF" can not be opened file for input.";
          return(1);
     }

     ofstream fout(_argv[2],ios::app
     |ios::binary|ios::trunc);



     //****VARIABLES****
     float sum;
     float A_COEFF[30], pred;
     int Y[512],order,i,j;
     order=atoi(_argv[3])+1;
     unsigned char X[512],IM_X;


     //****READING PREDICTOR COEFFICIENTS
     DOFOR(i,order) {
          DOFOR(j,order) {
               fin.read((uchar *) &A[i][j],sizeof(float));
          }
     }
```

```
        //****OPENING OF FILTERED FILE FOR READING
        if(fin.eof()) {
            fin.clear();
            fin.seek(0,ios::beg);
        }
        fin.open(_argv[1]);
            if(!fin) {
         cerr<<_argv[1]  <<" can not be opened file for input.";
            return(1);

        //****LINEAR PREDICTIVE DECODING****
        for(int k=0;k<513;k++) {
            fin.read((char *)&Y,sizeof(Y));
            for(int i=0;i<512;i++) {
                sum=0;
                for(int j=1;j<order+1;j++) {
                    if((i-j)<0)
                        IM_X=0;
                    else
                        IM_X=X[i-j];
                    pred =  float(IM_X)*A_COEFF[j];
                    sum+=int(pred);
                }
                X[i]=int(Y[i]-sum);
            }

            fout.write((unsigned char *) &X,sizeof(X));
        }
        return 0;
}

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

APPENDIX B

```
/*************************************************************
 *                                                           *
 *    Program Name: HISTO                                    *
 *                                                           *
 *      Input:          Filtered Image(Encoded using LPC)    *
 *                                                           *
 *      Output:         Maximum and Minimum Values,          *
 *                      Histogram, Standard Deviation, and   *
 *                      the Entropy of a Filtered Image      *
 *                                                           *
 *      Subroutines:    Histogram, Entropy                   *
 *                                                           *
 *    This program generates the histogram, the standard     *
 *    deviation,entropy and minimum and maximum values of    *
 *    filtered file.  It is executed by                      *
 *                                                           *
 *              HISTO file1 file2                            *
 *                                                           *
 *    where file1 is the input file(filtered file), and      *
 *    file2 is the output file(histogram file).              *
 *                                                           *
 *************************************************************/

//****HEADER FILES*****
#include <fstream.h>
#include <iostream.h>
#include <stdlib.h>
#include <limits.h>
#include <float.h>
#include <io.h>
#include <stdio.h>
#include <new.h>
#include <ctype.h>
#include <conio.h>
#include <math.h>
#include <dos.h>

//****FUNCTION DECLARATIONS****
int Histogram(char *input_file, char  *obfile);
int Entropy(long sum);


template <class T> T max(T x, T y)
    {
    return (x > y) ? x : y;
    }
template <class T> T min(T x, T y)
    {
    return (x < y) ? x : y;
```

```
    }
typedef uchar unsigned char;


main()
    {
     if(_argc<3) {
          textcolor(YELLOW );
          cprintf(" USAGE: HISTO <input_file>
                 <b_output file>\r\n");
          return(1);
     }

     Histogram(_argv[1],_argv[2]);
     return 0;
     }
//#########################################################
```

```
/*******************************************************
 *                                                     *
 *                      FUNCTIONS                      *
 *                                                     *
 *******************************************************/


 ************************************************
 *                HISTOGRAM FUNCTION            *
 *                                              *
 * The function generates the frequency counts  *
 * of the filtered file(LPC encoded) and the    *
 * min and max values.                          *
 *                                              *
 ************************************************



 ************************************************
 *              Variables                       *
 *                                              *
 *    freq_cnt - frequency count                *
 *    freq_sum - total frequency count          *
 *    inp_file - LPC filtered file              *
 *    o_file - Binary output file               *
 *    pred_err - prediction error               *
 *    SD - standard deviation                   *
 *    start, end - range of frequency values    *
 *                 being counted                *
 *    MAX,MIN,i,index - scratch variables       *
 *                                              *
 ************************************************/


int Histogram(char *inp_file,char *o_file) {

     //********IO SETUP********
     ifstream fin(inp_file,ios::binary|ios::nocreate);
     if(!fin) {
          cerr<<inp_file<<" can not be opened file for
                          input.\n";
          return(1);
     }

     ofstream fotb(o_file,ios::binary|ios::trunc);
     ofstream f_t("plot.dat",ios::trunc);      //PLOT FILE
```

```
//****VARIABLES****
float SD=0,pred_err;
int start,end=0,index;
long freq_sum =0;
long MAX=0,MIN = 0;
unsigned freq_cnt[8192];
start=-32768;


//****CALCULATION OF MAX AND MIN VALUES****
while(fin.read((char *)&pred_err,sizeof(pred_err)))
     {
     MAX=max(long(pred_err),MAX);
     MIN=min(long(pred_err),MIN);
}

if(fin.eof()) {
     fin.clear();
     fin.seek(0,ios::beg);
}


//****FREQUENCY CALCULATION****
while(end!=INT_MAX) {
      //**** INITIALIZATION OF FREQ_CNT ARRAY*****
     for(int i=0;i<8192;i++)
          freq_cnt[i]=0;
     end=start+8191;

     while(fin.read((char *)&pred_err,sizeof(int))){
          index = pred_err-start;
          if((pred_err >= start) && (pred_err <= end))
               {
               freq_cnt[index]+=1;
               freq_sum+=freq_cnt[index];
               }

     }


     //****OUTPUT OF DATA ****
     for(int i=0;i<8192;i++) {
          if(freq_cnt[i]!=0) {
               index=start+i;
               fotb.write((char *)&index,sizeof(int));
               fotb.write((uchar*)&freq_cnt[i],
                    sizeof(int));
          }
     }

     if(fin.eof())
          fin.clear();
```

```
            start=end+1;
        }



        //****CLOSING OF DATA FILES DATA ****
        fotb.close();
        fin.close();


        cout<<"MAX: "<<MAX<<"\n";
        cout<<"MIN: "<<MIN<<"\n";

        //****CALL TO ENTROPY FUNCTION****
        SD=Entropy(sum);

        return 0;
}
//#############################################################
```

```
/*************************************************************
 *                      ENTROPY FUNCTION                     *
 *                                                           *
 *   This function calculates the entropy and the            *
 *   standard deviation of the filtered file                 *
 *                                                           *
 *************************************************************


 ************************************************
 *                                              *
 *                 Variables                    *
 *                                              *
 *   freq_count - frequency count               *
 *   freq_sum - sum of frequency count          *
 *   H - entropy                                *
 *   p_histo - probability of prediction error  *
 *   SD - standard deviation                    *
 *   var - variance                             *
 *   pixel_val - pixel values -32768 to +32767  *
 *   result - scratch variables                 *
 *                                              *
 ************************************************/


int Entropy(long freq_sum)
     {

     //****IO SETUP****
     ifstream fin(_argv[3],ios::binary|ios::nocreate);
     if(!fin)
          {
          cerr<<in_file<< " can not be opened file for
        input.";
          return(1);
          }
     ofstream fout(filen,ios::nocreate|ios::app);
     if(!fout)
          {
          cerr<<filen<< " can not be opened file for
                        input.\n";
          return(1);
          }


     //****VARIABLES****
     double p_histo,H=0,result=0,var;
     float SD=0;
     int pixel_val;
     unsigned freq_count;
```

```
      //****ENTROPY and  STANDARD DEVIATION CALCULATIONS****
      while( fin.read((char *)&pixel_val,sizeof(int))) {

            fin.read((uchar *)&freq_count,sizeof(freq_count));
            p_histo=double(freq_count)/(double(freq_sum));

            result = log10(p_histo)/(log10(2));
            H+=(-1)*(p_histo)*result;
            var+=(double(pixel_val)*double(pixel_val)*p_hist);
            }

      SD=sqrt(var);
      cout< <"SD is "<<sqrt(SD)<<"\n";
      cout<<"H is "<<H<<"\n";
      return SD;
}

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

APPENDIX C

```
/*************************************************************
 *                                                           *
 *     Program Name:  HUFF_COMP                              *
 *                                                           *
 *      Input:        Filter Image(Encoded using LPC)        *
 *                                                           *
 *      Output:       Huffman Coded File                     *
 *                                                           *
 *      Subroutines: IBSET,IBTEST,IOUT                       *
 *                                                           *
 *                                                           *
 *************************************************************
```

PROGRAM DESCRIPTION

```
*************************************************************
```

The HUFF_COMP program compresses a file using Huffman coding. This program is executed by typing:

HUFF_COMP file1 file2

where file1 is the input file(a filtered image) and file2 is the output file. The HUFF_COMP program generates a Huffman code tree for coding prediction errors that are Gaussian distributed with zero mean and standard deviation SD. The tree is then traversed from each leaf to the root node to generate the code words for the prediction errors. This program generates **N(5σ)** code words for prediction errors between **-N/2(-2.5σ)** to **N/2(2.5σ)**. Three structures are used to store information in this program. The NODE structure stores all the information required to describe the Huffman tree. The Huffman code words reside in the CODE structure. The B_OUT structure tracks the current word being dumped to the output file.

Some of this routine is similar to the PASCAL program for Huffman coding given in: A. M. Tenenbaum and M. J. Augenstein, Data Structures Using PASCAL. Second edition. Englewood Cliffs, NJ: Prentice-Hall. 1986, pp.322-324.

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

```
/*****************************************************
 *                                                   *
 *              Variables                            *
 *                                                   *
 *     C1,C2 - constants for probabilities           *
 *              calculation                          *
 *     cd - structure containing Huffman code        *
 *          words                                    *
 *     M_PI - C constant for π                       *
 *     N - # of Huffman code words                   *
 *     nodes - Huffman tree                          *
 *     n1 - node with lowest probability             *
 *     n2 - node with second lowest probability      *
 *     num_bits - # of bits in compressed file       *
 *     pred_err - prediction error                   *
 *     sum - sum of probabilities                    *
 *     SD - standard deviation                       *
 *     X - structure for writing out code words      *
 *     count, i,found,j,IBTMP - scratch variables    *
 *     ICODE,Len_wrd          - scratch variables    *
 *                                                   *
 *****************************************************

*************************************************************/
//****HEADER FILES****
#include <iostream.h>
#include <dos.h>
#include <math.h>
#include <stdlib.h>
#include <fstream.h>

        /* The B_OUT structure contains the information
        required to write a code word.  Inval holds the
        word being generated and count tells which bit is
        being processed. */

        struct B_OUT
             {
              int inval;
              int count;
             };


        /* The CODE structure contains the code words.
        Lbits holds the number of bits in a code word.  Wrd
        contains the code generated by the Huffman code.
        Index tells which prediction error the code word
        represents.  This is used because C array's can
        only have positive indices.  Inval contains the 16-
        bit representation of the code  word stored in the
```

```
array wrd.  The first N/2 locations of the CODE
structure contains the positive prediction
errors;word[0] gives the code word for a
prediction error of magnitude 0, word[1] gives the
code word for a prediction error of magnitude 1,
etc.  Negative prediction errors are stored in
locations N/2+i for i=1 to N/2; word[N/2+1] gives
the code word for the prediction error of magnitude
-1, word[N/2+2] gives the code word for a
prediction error of magnitude -2, etc. */

struct CODE
      {
       int wrd[17];
       unsigned Lbits;
       int index;
       int inval;
      };


/* The NODE structure contains the information
used to describe Huffman tree. Freq holds the
frequency of the current node. Father identifies
the parent node of current node. Nextroot
identifies what node the current branch is
connected to. The root node has zero as its next
node. Stype identifies whether a node is a 0
branch or a 1 branch. Index tells which prediction
error the node represents. */

struct NODE
      {
       double freq;
       int father;
       int nextroot;
       unsigned char stype;
       int index;
      };


//FUNCTION DECLARATIONS****
int IBSET(int word, int ITH_BIT);
int IBTEST(int word,int ITH_BIT);
B_OUT IOUT(int inval, int nbits,int iflush, B_OUT X);

//#########################################################


int main()
     {
          if(_argc<3) {
```

```
        cout<<"USAGE:"<<"HUFF_COMP <INFILE>
              <OUTFILE> \n";
    return 1;
    }

    //****I/O_SETUP*****
    ifstream fin(_argv[2],ios::binary);
    ofstream out("codes_out",ios::trunc);

    //****VARIABLES******
    enum {FALSE, TRUE};
    double Prob,sum, frq,SD;
    float c1,c2,PI,var;
    int count,i,I,N;
    int n1=0,n2=0,kk=0,IBTMP[16];
    int j=firstroot,index,indx;
    int found=FALSE,L=0;
    CODE cd[MAXNODE];
    NODE nodes[MAXNODE];

    //****INPUT OF STANDARD DEVIATION****
    cout<<"enter SD:";
    cin>>SD;
    var=SD*SD;
    out<<"SD:"<<SD<<"\n\n";
    fout.write((char *)SD,sizeof(SD));
```

/* Determines N (the number of code words generated during the Huffman process. */

```
    N=5*SD;
    N = (N >= int(N)+.5) ? N+1 :  int(N);
    if(N%2==0) N+=1;
```

/* GENERATION OF SOURCE PROBABILITIES
The following lines of code create the first N tree nodes which are the leaf nodes. Since the prediction errors have a zero-mean Gaussian distribution, the source probabilities can be calculated from the standard deviation. These probabilities are stored in the *freq* section of **nodes**(a NODE structure). C1 and C2 are constants used to calculate source probabilities. The sum of the probabilities is accumulated in **sum**. The probabilities are generated for the range of values **-2.5σ** to **2.5σ**. The sum of the discrete probabilities is normalized exactly unity by dividing the probabilities by sum. */

```
    c1=1.0/(sqrt(2*M_PI)*SD);
    c2=1.0/(2*SD*SD);
    j=-N/2;
```

```
i=1;


while(j<0) {
      Prob=((j*j*c2));
      Prob=c1*exp(-Prob);
      nodes[i].freq=Prob;
      nodes[i+1].freq=Prob;
      sum+=2*Prob;
      j++;
      i+=2;
}
nodes[N].freq=c1;
sum+=c1;

//****NORMALIZATION OF PROBABILITIES****
for(j=1;j<=N;j++)
      nodes[j].freq/=sum;

//****GENERATION OF INDEXES****
j=1;
count=-N/2;

while(j<N)
      {
      nodes[j].index=count;
      nodes[j+1].index=(-count);
      j+=2;
      count++;
}
nodes[N].index=0;

/* INITIALIZATION OF TREE
  OF STRUCTURE */
for(i=1;i<N;i++)
      nodes[i].nextroot=i+1;

for(i=1;i<2*N;i++) {
      nodes[i].father=0;
      nodes[i].stype='0';
}
nodes[N].nextroot=0;
```

/* HUFFMAN TREE
The following lines of code generate the Huffman tree by summing probabilities and sorting the probability list of available nodes. The variable ***m*** designates the current node being created. The information describing the tree is stored in the **nodes** variable. */

```
int firstroot=1;
for(int m=N+1;m<2*N;m++)
     {
     n1=firstroot;
     n2=nodes[n1].nextroot;
     nodes[m].freq=nodes[n1].freq+nodes[n2].freq;
     nodes[n1].father=m;
     nodes[n1].stype ='l';
     nodes[n2].father=m;
     nodes[n2].stype='r';
     frq=nodes[m].freq;
     firstroot=nodes[n2].nextroot;
     kk=0;
     found=FALSE;
     j=firstroot;
     while((j!=0) && !(found)) {
          if(nodes[j].freq >= frq)
               found = TRUE;
          else {
               kk=j;
               j=nodes[j].nextroot;
          }
     }
     if(kk==0)
          firstroot = m;
     else
          nodes[kk].nextroot = m;
     nodes[m].nextroot = j;
}
```

/* GENERATION OF HUFFMAN CODES
The following lines of code traverse the tree to form the code words. The code words are stored in **cd**(a CODE structure). Starting at each leaf node, its father node is found followed by finding the father's father, etc., until the root node is encountered. The bit value of each branch encountered is the value in the stype field of the **nodes** structute. A *l* type assigns the value of zero to a branch and a *r* type assigns a 1 to the branch. The bits generated are temporally stored in IBTMP array and later are permanently stored in the *wrd* field of *cd*. */

```
for(i=1;i<N+1;i++)
     {
     j=i;
     J=0;
     L=0;
while((nodes[j].father !=0) && (L <16)) {

          if(nodes[j].stype =='l')
               IBTMP[J]=0;
          if(nodes[j].stype =='r')
               IBTMP[J]=1;
          L++;
          j=nodes[j].father;
          J++;
     }
     if(nodes[i].index<0)
          indx=N/2+abs(nodes[i].index);
     else
          indx=nodes[i].index;
     cd[indx].Lbits=L;
     cd[indx].index=index;
     for(int I=0;I<L;I++)
          cd[indx].wrd[I]=IBTMP[I];
     L=0;
}
```

/* The following lines of code convert the Huffman code words into their sixteen bit value which is stored in the *inval* field of **cd**. */

```
int ICODE;
for(i=0;i<N;i++) {
     L = cd[i].Lbits;
     ICODE=0;
     for(int K = L;K>0;K--) {
          if(cd[indx].wrd[K-1])
               ICODE = IBSET(ICODE,L-K);
     }
     cd[i].inval=ICODE;
}
```

/* ENCODING OF IMAGE
First a prediction error is read from the input file. If the prediction error is within 5 standard deviation, then the Huffman code word for that prediction error is written to a output file. If the error is outside the range then the code word for 2σ is written to file followed by the actual value of the prediction error written to the output file. */

```
//****IO_SETUP****
```

```
ifstream fin(_argv[1],ios::binary|ios::nocreate);
if(!fin) {
   cerr<<_argv[1]<<"_ will not open\n";
  return(1);
}

//****VARIABLES****
int pred_err,end,index,Len_bits;
unsigned long num_bits=0;
B_OUT X;
X.inval=0;
X.count=0;



while(!(fin.eof())) {
     fin.read((char *)&pred_err,sizeof(int));
     if(pred_err<0)
          index = N/2 + abs(pred_err);
     else
          index=pred_err;

     /* Encoding of prediction errors
        for the range (-2.5σ, 2.5σ) */

     if((pred_err > -N/2) && (pred_err < N/2)) {
          Len_bits=cd[index].Lbits;
          X=IOUT(cd[index].inval,Len_bits,0,X);
          num_bits+=L;
     }


     /* Encoding of prediction errors
        outside the range[-2.5σ, 2.5σ] */

     if((pred_err<= -N/2) || (pred_err >=N/2)) {
          L=cd[N/2].Lbits;
          X=IOUT(cd[N/2].inval,L,0,X);
          L+=16;
          X=IOUT(pred_err,16,0,X);
          num_bits+=L;
     }
}


//**** PRINTS OUTS COMPRESSION RATIO
 cout<<"\nending compressing:\n";
 long total_bits = ( num_bits)/long (8);
 cout<<"Compressed bytes: "<<(num_bits)/long(8);
 cout<<"\n(float(total_bits) /float(262656))*100;
 cout<<" % compressed\n";
```

```
            cout<<"Input:"<<_argv[1];
            cout<<"   Original Size: 262656 \n";
            out<<"Compressed bytes: "<<(num_bits)/long (8);
            out<<"\n"<<(float(total_bits) /float(262656))*100
                <<" % compressed\n";

            return 0;
}
//############################################################
//############################################################
```

```
/*****************************************************
*                  **** FUNCTIONS ****                *
*****************************************************


*************************************************
*                  IBSET FUNCTION                *
*                                                *
* Function IBSET sets the ITH_BIT bit position *
* in WORD.  In MASK only the ITH_BIT portion    *
* contains a 1.                                  *
*                                                *
*************************************************/

     int IBSET(int WORD, int ITH_BIT)
     {
      unsigned int mask = powl(2,ITH_BIT);
      WORD |= mask;
      return  WORD;
     }




//###########################################################


   ************************************************
   *                IBTEST FUNCTION                *
   *                                               *
   * Function IBTEST test the ITH_BIT             *
   * position in WORD.                             *
   *                                               *
   ************************************************/

     int IBTEST(int WORD,int ITH_BIT)
     {
      int val=0;
      unsigned int mask;
      mask = powl(2,ITH_BIT);
      val = (WORD&mask);
      if(val) val =1;
      return val;
     }

//###########################################################
```

```
/************************************************
 *                IOUT FUNCTION                 *
 *                                              *
 * Function IOUT outputs NBITS bits from IVAL   *
 * to an output file. Bits are transferred from *
 * IVAL to byte variable IWRITE least           *
 * significant  bit first.  Whenever 8 bits     *
 * have been transferred to IWRITE, IWRITE is   *
 * written to disk and the bit pointer J        *
 * is reset to bit 0.  IFLUSH is                *
 * normally .FALSE.  Setting  IFLUSH=.TRUE.     *
 * causes any bits remaining in IWRITE to be    *
 * stored.  This should only be  done with the  *
 * last bits to be transferred.                 *
 * The structure B_OUT tracks the current value *
 * in IWRITE and which bit in the word IWRITE   *
 * is being examined.                           *
 *                                              *
 ************************************************/

    B_OUT  IOUT(int INVAL, int NBITS,int iflush,B_OUT X)
    {

    ofstream fout(_argv[2],ios::binary|ios::app);
         int J = X.count;
         int IWRITE ;
         IWRITE=X.inval;
         if(!iflush)
              {
              for(int k=0;k<NBITS;k++) {
                   if(IBTEST(INVAL,k))
                        IWRITE=(IBSET(IWRITE,J));
                   J+=1;
                   if(J==8) {
                        fout.put(IWRITE);
                        J=0;
                        IWRITE=0;
                   }
              }
         }
         else
              if(J!=0) fout.put(IWRITE);
         X.inval=IWRITE;
         X.count=J;
         return X;

    }
//############################################################
//############################################################
```

APPENDIX D

```
/*************************************************************
 *                                                           *
 *     Program Name:  HUFF_EXPAND                            *
 *                                                           *
 *      Input:        Huffman Encoded  File                  *
 *                                                           *
 *      Output:       Decoded File(LPC Filtered)             *
 *                                                           *
 *      Subroutines: Expand,IBSET,IBTEST,IN                  *
 *                                                           *
 *                                                           *
 *************************************************************

                        PROGRAM DESCRIPTION
*************************************************************
```

The HUFF_EXPAND program expands a file compressed by Huffman coding. Called by:

HUFF_EXPAND file1 file2

where file1 is the input compressed file and file2 is the output uncompressed BINARY file. This code generates a Huffman code tree for a source that is Gaussian distributed with zero mean and standard deviation SD. The tree is then traversed from the root to the leaf node to decode the prediction errors. This program generates **N(5σ)** code words for prediction errors between **-N/2(-2.5σ)** to **N/2(2.5σ)**. Two structures are used to store information in this program.The NODE structure stores all the information required to describe the Huffman tree. The B_IN structure tracks the current word being read from the input file.

Some of this routine is similar to the PASCAL program for Huffman coding given in: A. M. Tenenbaum and M. J. Augenstein, Data Structures Using PASCAL. Second edition. Englewood Cliffs, NJ: Prentice-Hall. 1986, pp.322-324.

```
*************************************************************/
//****HEADER FILES****
```

```
#include <iostream.h>
#include <dos.h>
#include <math.h>
#include <stdlib.h>
#include <fstream.h>
#define MAXN  256
#define MAXNODE 2*MAXN-1

        /* The B_IN structure contains the information
        required to retrieve a code word a bit at a time.
        This structure has four fields: c_val, flag, and
        pos.  The c_val field tracks the current byte
        read, the counter field identifies which bit in
        c_val is being examined by  the function.The flag
        field tests whether this the first time the input
        file is being accessed.  The inval field  stores
        the current word being created, and the pos field
        tracks the current location accessed by the file.
        */

        struct B_IN
             {
              char c_val;
              int counter;
              int flag;
              int inval;
              unsigned long pos;
             };


        /* This NODE structure contains the information
        describing the Huffman tree. Freq holds the
        frequency of the current node.  Lson and rson
        identify the sibling nodes of the current node.
        nextroot identifies what node is the current
        branch is connected to. The root node has zero as
        its next node.  Stype identifies whether a node is
        a 0 branch or a 1 branch. Index tells which
        prediction error the node represents.  Nextroot
        identifies the next node the tree branch is
        connected to. */

        struct NODE
             {
              double freq;
              int index;
              int lson;
              int rson;
              int nextroot;
              unsigned char stype;
             };
```

```
//****FUNCTION DECLARATIONS****
int EXPAND();
int IBSET(int word, int bit_num);
int IBTEST(int word,int bit_num);
B_IN IN(int nbits,B_IN in);


//#########################################################

int main()
     {
     if(_argc < 3) {
          cout<<"USAGE:"<<"Expand <INFILE> <OUTFILE> \n";
          return 1;
     }
     EXPAND();
     return 0;
}
/#########################################################
/#########################################################
```

```
/*********************************************************
*                        **** FUNCTIONS****                *
*********************************************************

*********************************************************
*                    EXPAND  FUNCTION                    *
*                                                        *
*   This function takes a Huffman coded file             *
*   and decodes it.                                      *
*                                                        *
*********************************************************


*********************************************************
*                                                       *
*              Variables                                *
*                                                       *
*   C1,C2 - constants for probabilities                 *
*           calculation                                 *
*   m - current node                                    *
*   N - # of Huffman code words                         *
*   nodes - structure for Huffman tree                  *
*   M_PI - C constant for π                             *
*   sum - sum of probabilities                          *
*   n1 - node with lowest probability                   *
*   n2 - node with second lowest probability            *
*   SD - standard deviation                             *
*   pred_err - decoded prediction error                 *
*   x - structure input Code word                       *
*   count, i,found,j - scratch variables                *
*   firstroot,NP,row - scratch variables                *
*                                                       *
*********************************************************/


//****EXPAND FUNCTION****
     int EXPAND() {

          //****I/O_SETUP***
          ifstream fin(_argv[1],ios::binary|ios::nocreate);
          ofstream ft (_argv[2],ios::trunc|ios::binary);


          //****VARIABLES****
          double sum=0,frq,SD,prob;
          enum {FALSE, TRUE};
          float c1,c2;
          int N,n1,n2,pred_err,index,NP,j;
          int firstroot=1,inval,count,i;
          long row, pos=0;
          B_IN x;
```

```
NODE nodes[255];

//****INPUT OF STANDARD DEVIATION****
fin.read((char *)&SD,sizeof(SD))
```

/* Determines N (the number of code words generated during the Huffman process. */

```
double N=5*SD;
N = (N >= int(N)+.5) ? N+1 :  int(N);
if(N%2==0) N+=1;
```

/* GENERATION OF SOURCE PROBABILITIES
This section generates the source probabilities from the standard deviation. This process is done in the same way as described in the HUFF_COMP program These probabilities are assigned to the leaf nodes of the Huffman coding tree. These probabilities are stored *freq* field of the ***nodes*** structure. */

```
c1=1.0/(sqrt(2*M_PI)*SD);
c2=1.0/(SD*SD*2);
int j=-N/2;
i=1;
while( j <0) {
     prob=((j*j*c2));
     prob=c1*exp(-prob);
     nodes[i].freq=prob;
     nodes[i+1].freq=prob;
     sum+=prob*2;
     j++;
     i+=2;
}
nodes[N].freq=c1;
sum+=c1;
//****NORMALIZATION OF PROBABILITIES****
for(j =1;j<=N;j++)
     nodes[j].freq/=sum;
```

```
//****GENERATION OF INDEXES****
j=1;
count=-N/2;
while( j<N)
     {
     nodes[j].index=count;
     nodes[j+1].index=-count;
     j+=2;
     count++;
}
nodes[N].index=0;


 /* INITIALIZATION OF TREE
   OF STRUCTURE */
for(i=1;i<N;i++)
     nodes[i].nextroot=i+1;
for(i=1;i<2*N;i++) {
     nodes[i].lson=0;
     nodes[i].rson=0;
     nodes[i].stype='0';
}
nodes[N].nextroot=0;
```

/* HUFFMAN TREE
The following lines of code generate the Huffman tree by summing probabilities and sorting the probability list of available nodes. The Huffman tree is formed in the same way as described in HUFF_COMP. */

```
j=firstroot;
found= FALSE;
for(int m=N+1;m<2*N;m++) {
n1=firstroot;
n2=nodes[n1].nextroot;
nodes[m].freq=nodes[n1].freq+nodes[n2].freq;
nodes[m].lson=n1;
nodes[m].rson=n2;
frq=nodes[m].freq;
firstroot=nodes[n2].nextroot;
kk=0;
found=FALSE;
j=firstroot;
//****SORTING OF PROBABILITIES****
while((j!=0) && !(found))
     {
     if(nodes[j].freq >= frq)
          found = TRUE;
     else
          {
          kk=j;
          j=nodes[j].nextroot;
```

```
        }
    }
    if(kk==0)
        firstroot = m;
    else
        nodes[kk].nextroot = m;
    nodes[m].nextroot = j;
    }

    //*INITIALIZATION OF x Structure****
    x.inval=0;
    x.counter=0;
    x.pos=0;
    x.flag=0;
```

/* HUFFMAN DECODING ROUTINE
The Huffman tree is traversed from the root node to a leaf node by moving down one branch for each bit input. If the bit is a zero, the branch to the right son is taken and if the bit is 1, the branch to the left son is taken. This process continues until the node pointer,**NP**, is less than or equal to N which identifies a leaf node. If the leaf node pointer is less than N then the prediction error magnitude is equal to value in the *index* field of ***nodes*** pointed to by ***NP***. Otherwise, 16 more bits must be read to obtain the actual prediction error magnitude. At the end of the while loop the decoded prediction error is written to an output file. The node pointer(NP) is reset to the last branch generated in the tree(m-1 where m is 2*N). */

```
    NP=m-1;

    r0w=262656;
```

```
        while(row>0) {
             while(NP>N) {
                  x=IN(1,x);
                  pos=x.pos;
                  inval = x.inval;

                  //****SELECT APPROPRIATE SON****
                  if(x.inval==0)
                       NP= nodes[NP].lson;
                  else
                       NP= nodes[NP].rson;
             }

             index = nodes[NP].index;

             /* If index is equal to 2.5σ read
             16 more bits to retrieve actual
             prediction error. */

             if(index==N/2) {
                  x=IN(16,x);
                  pred_err=x.inval;
             }


             /* If index is less than to 2.5σ the
                decoded prediction error is the
                value in index. */

             if(index !==N/2)
             pred_err=index;

             ft.write((char *)&pred_err,sizeof(int));
             NP=m-1;
             row--;
        }

        cout<<"expansion finished\n";
        return 0;
   }

//#########################################################
```

```
/************************************************
 *                  IBSET FUNCTION              *
 *                                              *
 * Function IBSET sets the ith bit position in  *
 * WORD where ITH_BIT indicates which position  *
 * to set. In MASK only the ITH_BIT portion     *
 * is set      .                                *
 *                                              *
 ************************************************/

     int IBSET(int WORD, int ITH_BIT)
     {
      unsigned int mask = powl(2,ITH_BIT);
      WORD |= mask;
      return  WORD;
     }



/************************************************
 *                  IBTEST FUNCTION             *
 *                                              *
 * Function IBTEST test the ITH_BIT             *
 * position in WORD.                            *
 *                                              *
 ************************************************/

   int IBTEST(int WORD,int ITH_BIT)
    {
     int val=0;
     unsigned int mask;
     mask = powl(2,ITH_BIT);
     val = (WORD&mask);
     if(val) val =1;
     return val;
    }
```

```
/*************************************************
 *                  IN FUNCTION                  *
 *                                               *
 * Function In reads nbits(a specified number of*
 * bits)  into the variable iread from an input  *
 * file. Bits are retrieved byte at a time.      *
 * The structure B_IN tracks the current value   *
 * in IREAD.                                     *
 *                                               *
 *************************************************

   B_IN IN(int nbits, B_IN X)
   {
    ifstream fin(_argv[1],ios::binary);
    int j = X.counter;
    char iread=X.c_val;
    int itest=X.flag,inval=0;

    if(itest==0) {
         fin.seek(X.pos,ios::beg);
         fin.get(iread);
         itest=1;
         X.pos=fin.tellg();
    }

    for(int k=0;k<nbits;k++) {
         if(IBTEST(iread,j))
              inval=(IBSET(inval,k));
         j+=1;
         if(j==8) {
              fin.seek(X.pos,ios::beg);
              fin.get(iread);
              X.pos=fin.tellg();
              j=0;
         }
    }

    X.flag=itest;
    X.inval=inval;
    X.counter= j;
    X.c_val=iread;
    return X;
   }
```

VITA

Montanez Altamese Wade was born on February 18. She attended school in Nashville, Tennessee and graduated from St. Bernard Academy High School in May, 1981. From August 1981, to August 1986, she attended Tennessee State University, Nashville, Tennessee and obtained a Bachelor of Science degree in Electrical Engineering. She held memberships and offices in Eta Kappa Nu, Golden Key, and Alpha Kappa Mu Honor Societies. From 1986 to 1989 she worked at AT&T Bell Laboratories, North Andover, Massachusetts. In 1990, she enrolled in The University of Tennessee Space Institute in Tullahoma, TN. She was awarded the Master of Science degree in Electrical Engineering in December 1993. The author currently holds memberships in The Institute of Electrical and Electronics Engineers, National Technical Association, and American Society of Engineering Education professional societies. She is also a member of Sigma Xi, and Eta Kappa Nu Honor societies.

For now she shares her knowledge with others because education is the passport to the future, for tomorrow belongs to the people who prepare for it today[27].