3-14-2001

Multilevel data compression techniques for transmission of audio over networks

Craig Chin
Florida International University

DOI: 10.25148/etd.FI14060803

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

MULTILEVEL DATA COMPRESSION TECHNIQUES FOR TRANSMISSION OF AUDIO OVER NETWORKS

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL ENGINEERING

by

Craig Chin

2001

To: Interim Dean Richard Irey
College of Engineering

This thesis, written by Craig Chin, and entitled Multilevel Data Compression Techniques for Transmission of Audio Over Networks, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

Dr. Jean Andrian

Dr. Tadeusz Babij

Dr. Subbarao Wunnava, Major Professor

Date of Defense: March 14, 2001

The thesis of Craig Chin is approved.

Interim Dean Richard Irey
College of Engineering

Dean Douglas Wartzok
Division of Graduate Studies

Florida International University, 2001

DEDICATION

I dedicate this thesis to my father whose encouragement and support was essential to the success of this endeavor, and the memory of my mother, whose open love and nurturing helped me to be a better man.

ACKNOWLEDGMENTS

I would like to thank the members of my committee for their assistance in the completion of this thesis. Their input was most sincerely appreciated. I give special thanks to Dr. SubbaraoWunnava, who sought to guide me through to the completion of my thesis in spite of trying physical ailments. Thanks also to Dr. Jean Andrian and Dr. Tadeusz Babij for the indispensable input they made towards the completion of this thesis. I must also thank Mrs. Pat Brammer who was instrumental in the success of not only my thesis, but every endeavor that I was involved in during my master's program at FIU.

ABSTRACT OF THE THESIS

MULTILEVEL DATA COMPRESSION TECHNIQUES FOR TRANSMISSION OF AUDIO OVER NETWORKS

by

Craig Chin

Florida International University, 2001

Miami, Florida

Professor Subbarao Wunnava, Major Professor

With the spread of the Internet into mainstream society, there has come a demand for the efficient transmission of multimedia information. Accompanying the drive to find more efficient ways of utilizing limited transmission bandwidth is a need to find novel ways of compressing data. This thesis proposed the utilization of transform coding compression techniques for the transmission of audio data across networks. The Discrete Cosine Transform (DCT) and the Discrete Sine Transform (DST) were the primary transforms utilized. This thesis investigated the viability of utilizing individual transforms, as well as, nested modifications of these transforms for compression purposes. These techniques were compared to those already in existence. Viability was determined using objective compression measures. It was found that transform coding techniques gave a useful alternative to the techniques in existence. A voice-over-IP (VOIP) application that utilized one of the transform coding techniques was implemented.

TABLE OF CONTENTS

LIST OF TABLES

TABLE PAGE

LIST OF FIGURES

FIGURE PAGE

Chapter 1 Introduction

1.1 Data Compression Overview

Data compression is the art or science of representing information in a compact form. These compact representations are created by identifying and using structures that exist in data.

There are two classes of data compression techniques: lossless compression and lossy compression. Lossless compression involves no loss of information. This means that the original data can be recovered exactly from the compressed data. Lossy compression involves some loss of information. This implies that the data that has been compressed cannot be used to recover the original data [1].

1.2 Existing Audio Compression Techniques

There are three major classes of audio compression techniques:

1. Differential Encoding
2. Transform Coding
3. Analysis/Synthesis Schemes

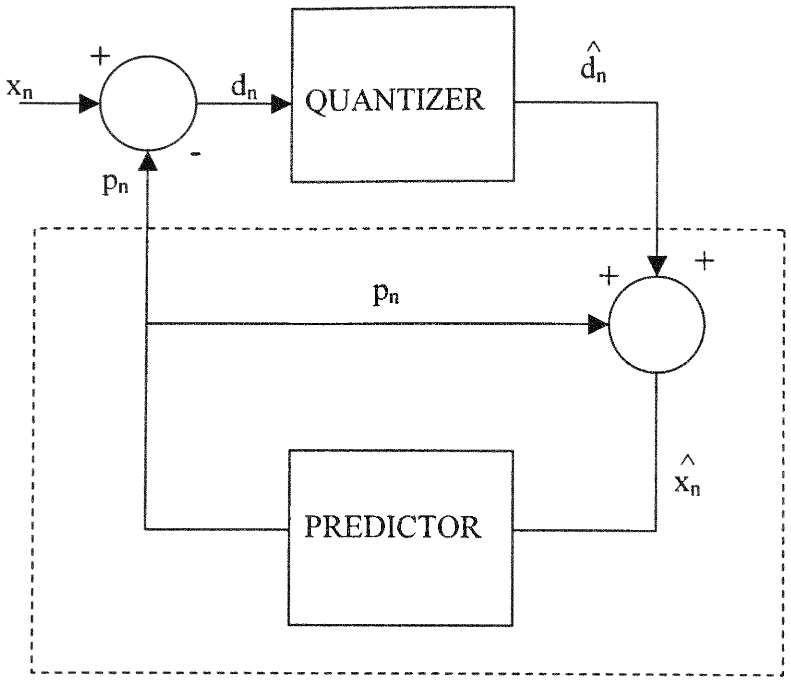
1.2.1 Differential Encoding

Differential Encoding schemes take advantage of the fact that audio sources, especially speech sources, show a high degree of correlation from sample to sample.

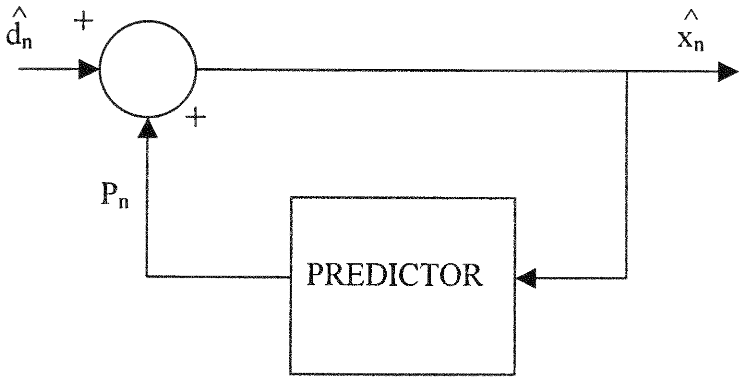
These schemes predict each sample based on past source outputs and only encode and transmit the differences between the prediction and the sample value. The differences are used, because when a source output does not change greatly from sample to sample this means the dynamic range (the variance for uniformly distributed sources) of the differences is smaller than that of the sample output itself. This allows the quantization step size to be smaller for a desired noise level, or quantization noise to be reduced for a given step size. Figure 1.1 shows the basic layout of a differential encoding system. This system is called a Differential Pulse Code Modulation (DPCM) system.

An Adaptive DPCM (ADPCM) system is one in which the quantizer and the predictor are made adaptive. This means that the quantization step size will be altered by the quantizer based on the prevailing local input signal characteristics. For the predictor, it means that the predictor coefficients of the prediction equation are changed by the predictor in accordance with local input signal characteristics.

The International Telecommunications Union (ITU) has published recommendations for a standard ADPCM system, recommendations include: G.721, G.723, and G.726. G.726 supercedes G.723 and G.721. The G.726 recommendation allows for ADPCM systems at rates of 40, 32, 24, and 16 kbps. The recommendation assumes that the speech output is sampled at a rate of 8000 samples per second, so rates of 40, 32, 24, and 16 kbps correspond to 5 bits per sample, 4 bits per sample, 3 bits per sample, and 2 bits per sample respectively. Comparing this to the PCM rate of 8 bits per sample, this would result in compression ratios of 1.6:1, 2:1, 2.67:1, and 4:1 respectively [1].



ENCODER



DECODER

FIGURE 1.1 A DPCM SYSTEM

1.2.2 Transform Coding

Transform coding involves the application of a set of mathematical functions, referred to as a transform, to a sequence of samples so that another sequence of values may be produced. The motivation for using such transforms for data compression applications is that the sequence of transformed values often possess characteristics that make them more amenable to compression coding, that is, the information or energy of the original sequence may be confined to only a few elements of the transformed sequence. This allows us to selectively encode and transmit only those elements of the transformed sequence that possess sufficient information, and hence data compression is accomplished. It should be noted that the transforms used are reversible, that is, the original sequence can be obtained from the transformed sequence by applying the appropriate inverse transform.

We can also analyze the transform process in terms of changes in statistics between the original and transformed sequences. It can be shown that we obtain the greatest amount of compaction when we use a transform that decorrelates the input sequence, that is, the sample-to-sample correlation of the transformed sequence is zero.

There are number of transforms that may be used for compression applications. These include:

1. The Discrete Cosine Transform (DCT)
2. The Discrete Sine Transform (DST)
3. The Fourier Transform
4. Wavelet Transforms

The most popular transform techniques today are the DCT and wavelet transforms. The DCT has been applied to a number of wideband audio coding applications. The applications include the ATRAC algorithm used in the MiniDisc system from Sony, Dolby AC-2 and AC-3 algorithms, the MPEG layer III algorithm [1, 3]. Wavelets have not been generally applied to audio compression techniques, however wavelet decomposition will form the basis of the JPEG 2000 standard for image compression [1].

1.2.3 Analysis/Synthesis Schemes

1.2.3.1 Analysis/Synthesis Schemes Overview

Analysis/Synthesis schemes rely on the availability of a parametric model of the source output generation process. With such a model, the transmitter analyzes the source output, extracts the model parameters, and transmits this information to the receiver. The receiver utilizes the model along with the transmitted parameters to synthesize the source output.

The transmitter does not send a direct representation of the source output samples. It actually sends to the receiver information instructing it how to reconstruct these output samples. This parametric information for some sources can be made significantly smaller in size than a direct representation of the source output samples. Hence analysis/synthesis schemes are useful for data compression applications.

One source that is amenable to modeling is speech. Speech is produced by forcing air through an elastic opening called the vocal cords. This action generates sound, and this sound is modulated into speech as it traverses the vocal tract. This physical process can be modeled into a speech synthesizer as shown in Figure 1.2. In Figure 1.2, the excitation source models sound generation, while the vocal tract filter models the vocal tract.

At the transmitter, analysis of the speech is performed. The speech is divided into a number of segments. For each segment an excitation signal and the parameters of the vocal tract filter are determined. This information is transmitted to the receiver where the excitation signal is synthesized and used to drive the vocal tract filter so that a speech signal may be generated.

The various types of analysis/synthesis schemes are described in the following sections

1.2.3.2 The Channel Vocoder

In the channel vocoder transmitter, the speech input is analyzed at regular intervals by a bank of bandpass filters called analysis filters. The energy output of each filter is estimated and transmitted to the receiver.

In addition to the filter output estimates, the transmitter also makes a decision as to whether the segment of speech being analyzed is voiced, as in the case of the sounds /a/ /e/ /o/, or unvoiced as in the case of sounds /s/ /f/. Voiced sounds tend to have a psuedoperiodic structure, and the period of the fundamental harmonic of such a sound is called the pitch period. So the transmitter also forms an estimate of the pitch period, which is sent to the receiver.

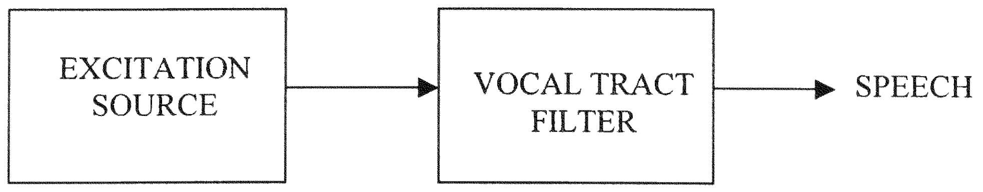


FIGURE 1.2 ANALYSIS/SYNTHESIS BLOCK DIAGRAM

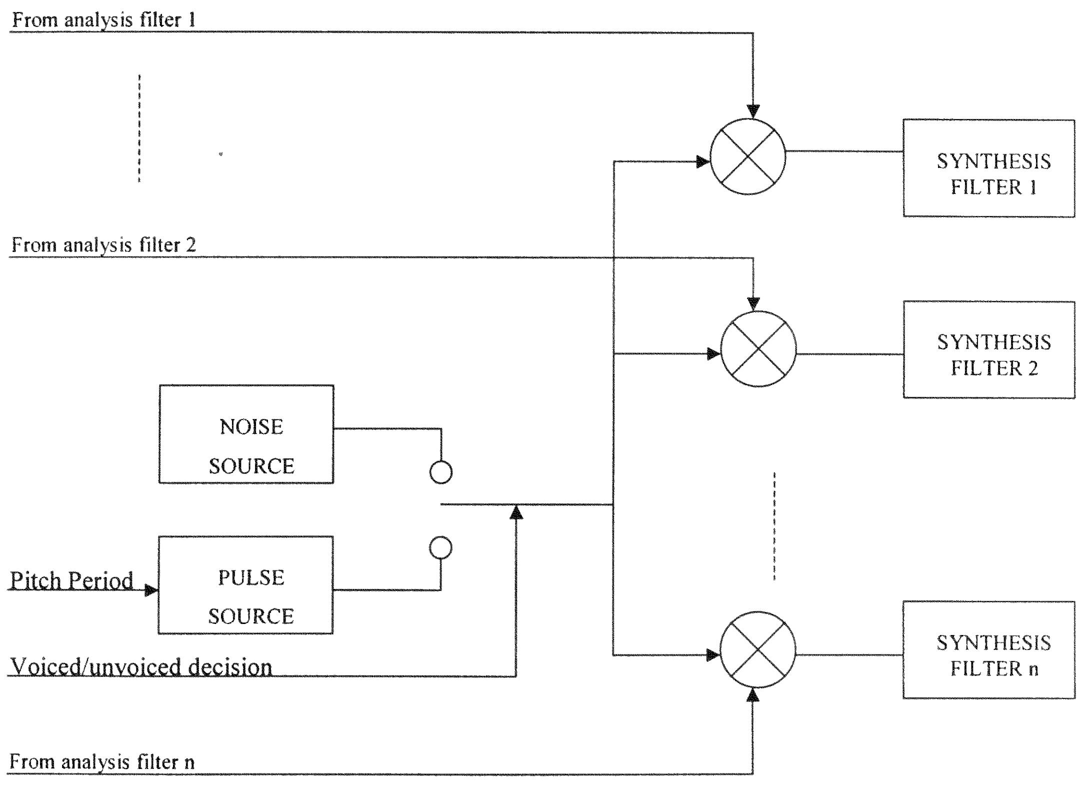


FIGURE 1.3 THE CHANNEL VOCODER RECEIVER

At the receiver, the vocal tract filter is implemented by a bank of bandpass filters called synthesis filters. Based on whether a speech segment is deemed to be voiced or unvoiced, either a periodic pulse generator or a pseudo-noise source is used as the input to the synthesis filter bank. The period of the pulse generator is acquired from the pitch period estimate from the transmitter, and the input is scaled by the energy estimate at the output of the analysis filters. See Figure 1.3.

The channel vocoder is the predecessor to several of the analysis/synthesis speech compression techniques used today, but is presently not one of the popular techniques in use [1].

1.2.3.3 The Linear Predictive Coder (LPC)

In a linear predictive coder the vocal tract is modeled by a single linear filter, whose output y_n is related to the input ε_n by the following equation:

$$y_n = \sum_{i=1}^{M} b_i y_{n-i} + G\varepsilon_n$$

Where G is the called the "gain" of the filter.

A block diagram of an LPC filter is shown in Figure 1.4.

As shown in the diagram, the input to the vocal tract filter is either a noise source or a periodic pulse source, based on whether the speech segment was reckoned to be voiced or unvoiced at the transmitter. The transmitter will also send the pitch period, if the segment is voiced, and the vocal tract filter parameters.

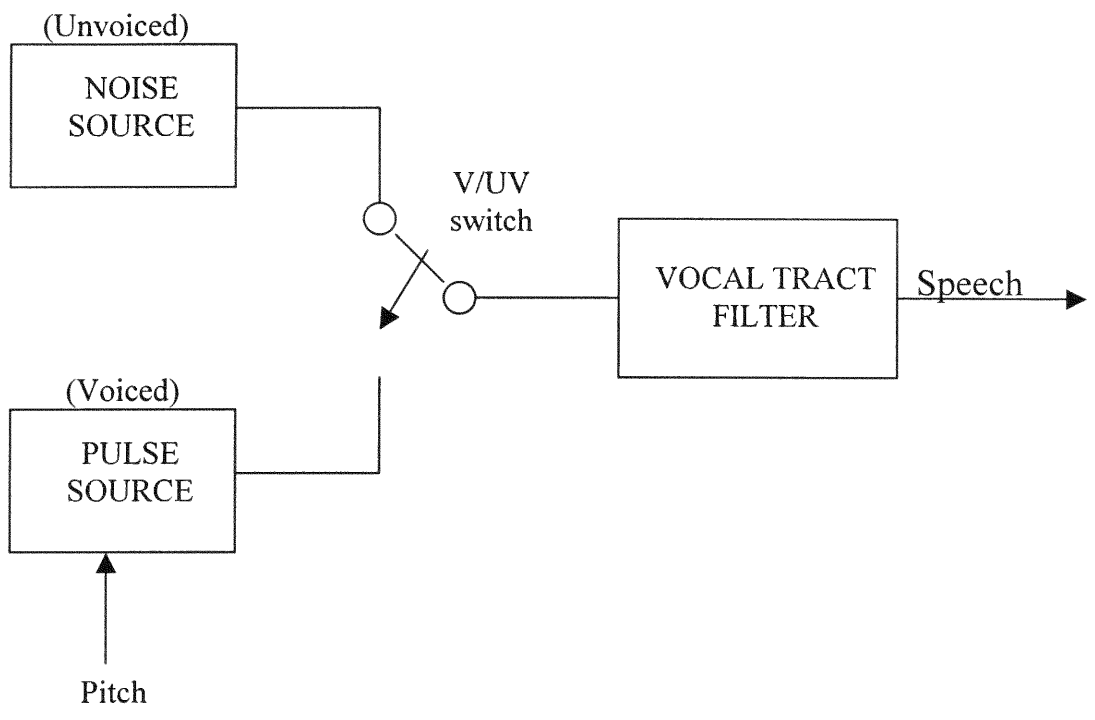


FIGURE 1.4 AN LPC RECEIVER

An LPC scheme is used in the 2.4 kilobit U.S. Government standard, Federal Standard 1015, for secure voice communication, known informally as LPC-10 [1, 3].

1.2.3.4 Code Excited Linear Prediction (CELP)

One of the drawbacks of an LPC system is that regardless of how accurate a pitch period estimate is, the use of a periodic pulse excitation that consists of a single pulse per pitch period leads to a "buzzy twang". A CELP system seeks to improve on an LPC system by using a codebook of excitation signals instead of the periodic pulse and psuedonoise generators.

At the transmitter, the vocal tract parameters are obtained using LPC analysis. The transmitter then excites the filter with entries from the codebook. The difference the original speech segment and the synthesized speech segment is determined by using a perceptual weighting filter. The codebook entry that generates the minimum average weighted error is declared the best match. The index of the best match entry is sent to the receiver along with the parameters for the vocal tract filter.

CELP algorithms are used for compression purposes in the U.S. Government Standard, FS 1016, as a 4.8 kbps coder, and in the ITU G.728 standard as a low-delay 16 kbps coder [1, 3]. An algorithm closely related to CELP algorithms, called the Regular Pulse Excitation with Long-Term Prediction (RPE-LTP) algorithm was adopted as a standard for digital cellular telephony by the Group Speciale Mobile (GSM) subcommittee of the European Telecommunications Standards Institute (ETSI) at a rate of 13 kbps [1, 3].

1.2.3.5 Sinusoidal Coders

Sinusoidal coders form excitation signals from a sum of sine waves of different amplitudes, frequencies, and phases. The excitation signal is of the form:

$$e_n = \sum_{l=1}^{L} a_l \cos(n\omega_l + \psi_l)$$

Where the number of sinusoids L required for each frame depends on the contents of the frame.

The vocal tract filter is a linear system; this means that its output has the same frequency as its input though they differ in amplitude and phase. This implies that the synthesized speech of a sinusoidal coder is of the form.

$$s_n = \sum_{l=1}^{L} A_l \cos(n\omega_l + \phi_l)$$

From this we can see that the number of parameters required to represent an excitation sequence is the same as the number of parameters required to represent synthesized speech. Therefore, sinusoidal coders directly estimate the parameters required to synthesize the speech and transmit these to the receiver.

The two most popular sinusoidal coding techniques today are the Sinusoidal Transform Coder (STC) and the Multiband Excitation coder (MBE). A version of the MBE coder, called the improved MBE (IMBE) coder has been adopted by Inmarsat as a standard for satellite voice communications, and by the Association of Police Communications Officers (APCO) as the standard for law enforcement [1, 3].

1.2.3.6 Mixed Excitation Linear Prediction (MELP)

The MELP algorithm uses the same LPC filter to model the vocal tract. It however uses multiband mixed excitation for its excitation signal. The mixed excitation consists of a filter signal from a noise generator plus a contribution that depends directly on the input signal. A block diagram of an MELP system is shown in Figure 1.5.

The MELP coder was selected to be the new federal standard for speech coding at 2.4 kbps by the Defense Department Voice Processing Consortium (DDVPC) [1].

1.3 Motivation for Research

The scope of the IP telephony industry is expected to increase significantly over the next decade. The International Data Corporation (IDC) predicts that global IP telephony will grow from an estimated 2.6 billion minutes in 1999 to more than 88 billion minutes in 2003. It also predicts that global revenues will increase from $600 million to $13.4 billion over the same period [2]. This growth is being stimulated by the savings that can be made by using the Internet to bypass regular telephone networks, and by a push in the multimedia industry to provide integrated voice, video and data services over one network. The technology does suffer presently from having a lower sound quality than that provided by the Public Switched Telephone Network (PSTN). IP telephony

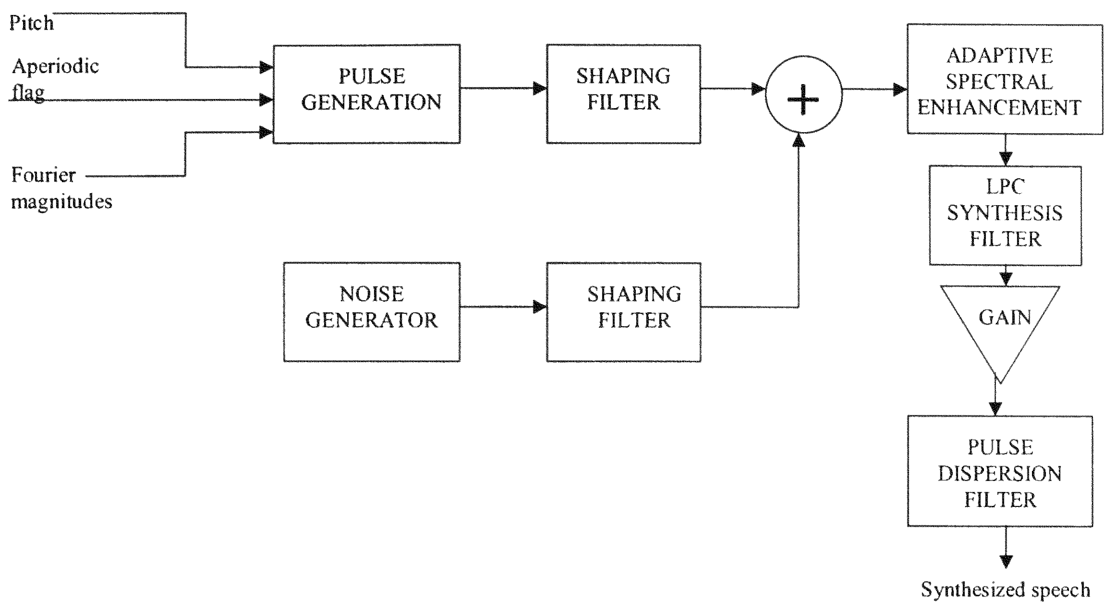


FIGURE 1.5 BLOCK DIAGRAM OF MELP DECODER

also suffers from excessive delays in full-duplex applications and from unreliable connections. Since this technology still requires improvement, there are opportunities for research and new innovations in this field.

The majority of voice-over-IP applications today utilize analysis/synthesis schemes to implement the data compression necessary to render such applications functional. This thesis will investigate the applicability of multilevel transform coding techniques for the implementation of a voice-over-IP application. It will be determined whether or not these methods of data compression can provide comparable or even superior compression performance to those presently in use.

The thesis will also seek to develop a rudimentary voice-over-IP application, and investigate some of the obstacles that are encountered in the development of such an application. This application can form the framework for future research and development of voice-over-IP technology.

This thesis will also provide a fundamental understanding of the variety of compression techniques that are utilized in the area of audio/voice compression, with a special emphasis on transform coding techniques. It can therefore serve as an introductory document to novices in the field of audio/voice compression techniques.

The thesis will also provide an introduction to the compression performance measures used to provide measurable evaluations of compression performance techniques. It will also provide a means of determining these measures in the instance that these compression techniques are applied to raw voice data files, that is, a software application was written to determine these measures for such files (eval.exe).

Chapter 2 Compression Performance Measures

2.1 Compression Performance Measures Overview

In order to determine whether one compression algorithm is more efficient than another algorithm we must specify some measures of performance that will allow us to evaluate individual algorithms and have them compared to each other. Such performance measures are described in this section. The performance measures described here will seek to quantify the amount of compression, and how closely the reconstruction (system output) resembles the original (system input).

For lossy compression it is necessary that we ascertain the difference between the reconstructed data and the original, in order for us to determine the efficiency of the compression algorithm. The difference between the original and the reconstruction is called distortion [1].

With regards to speech compression algorithms there are two groups of methods for assessing speech quality: subjective assessment, and objective assessment. Subjective assessment methods seek to determine the speech quality of a system output by utilizing the observations of human listeners. Objective assessment methods seek to predict speech quality based on objective measures of the physical parameters and properties of a speech signal.

Since the ultimate judge of speech quality is the human listener, subjective assessment methods are generally deemed to be more accurate. The major disadvantage of subjective speech assessment is that in order to obtain subjective speech quality data with good reliability and reproducibility large investments in terms of technical equipment and manpower are required [6]. Commonly used subjective test methods are Absolute Category Methods, Degradation Category Methods, Detectability Methods, Comparison Category Methods, and Threshold Methods. One widely used method is the direct evaluation of the speech quality by an Absolute Category Rating (ACR). In this method the subject is presented with short groups of unrelated sentences that were passed through a system under test. Typically the subject's task is to rate his or her impression on a five-point scale with absolute categories [6]. For example these categories may be classified as: bad, poor, fair, good, and excellent [7]. An estimate of the quality is then obtained from the arithmetic mean of the responses of all subjects. This quantity is called the mean opinion score (MOS) [6].

Objective assessment methods often show considerable deviation from subjective assessment methods and are generally considered to be less accurate. However they are often more easily implemented and cheaper as well. Most objective assessment methods are based on some arithmetic manipulation of the Signal-to-Noise Ratio (SNR) [6]. The method of choice for speech quality assessment for this thesis is the SNR, because of its simplicity and ease of implementation.

2.2 Compression Ratio

The compression ratio is the ratio of the number of bits required to represent the data before compression to the number of bits required to represent the data after compression [1].

Example: A 40 kilobyte image file is compressed into a 10 kilobyte file. This yields a compression ratio of 4:1.

2.3 Rate

Rate is simply the average number of bits required to represent a single sample for the compressed representation [1].

Example: For the example mentioned previously, if we assume that the original image file used 8 bits/pixel, then the rate for the compressed file would be 2 bits/pixel.

2.4 Mean-Squared Error (MSE)

The mean-squared error is one of the primary measures used to determine the average distortion between the original and reconstructed data sequences. It is given by the following equation [1].

$$\sigma^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n - y_n)^2$$

Where σ^2 = mse

x_n = source output sequence

y_n = reconstructed sequence

N = number of samples in sequence

2.5 Signal-To-Noise Ratio (SNR)

The signal-to-noise ratio gives a means of measuring the distortion relative to the input signal. The SNR is the ratio of the average squared value of the source output and the mse. It is given by [1]:

$$SNR = \frac{\sigma_x^2}{\sigma_d^2}$$

Where σ_x^2 = average squared value of the source output

σ_d^2 = mse

The SNR in decibels is given by [1]:

$$SNR(dB) = 10\log_{10}\frac{\sigma_x^2}{\sigma_d^2}$$

2.6 Peak-Signal-To-Noise Ratio (PSNR)

The peak-signal-to-noise ratio gives the size of error relative to the peak value of the signal. The PSNR is given by [1]:

$$PSNR(dB) = 10\log_{10}\frac{\sigma_{peak}^2}{\sigma_d^2}$$

Where x_{peak} = peak value of the signal

2.7 Practical Examples

This section seeks to detail some real-world examples in which the performance measures described previously are used to verify the quality of existing speech compression algorithms.

2.7.1 ADPCM Systems

The first set of ADPCM system evaluation information was obtained from [4]. This reference source contains two SNR versus bit rate plots for three forms of coders: logarithmic PCM, ADPCM (with adaptive quantizer and a simple first-order predictor), and Adaptive Delta Modulation (ADM). The results were obtained from computer simulations that used as coder input, a bandpass-filtered male utterance of "A lathe is a big tool." Two bandwidths were considered: 200 – 3200 Hz and 200 – 2400 Hz. The ADPCM system utilized a sampling frequency of 6.6 kHz.

Tables 2.1 and 2.2 were obtained by extrapolation from these plots at bit rates that are of significance in present-day applications. The compression ratio values were obtained by comparing the bits/sample values obtained for the ADPCM system to the standard 8 bits/sample value used for PCM coders.

The second set of ADPCM system evaluation information was obtained from [5]. This reference source introduces an "improved system for speech digitization using adaptive differential pulse-code modulation (ADPCM)". The system uses an adaptive predictor, an adaptive quantizer, and a variable length, source coding scheme. The system is called the "residual encoder".

BIT RATE (kbps)	BITS / SAMPLE (BIT RATE / 6600)	COMPRESSION RATIO	SNR (dB)
16	2.42	3.3	11.9
24	3.63	2.2	17.3
32	4.85	1.65	23.2
40	6.06	1.32	28.3

TABLE 2.1 ADPCM Compression Performance Data (Bandwidth 200 – 3200 Hz)

TABLE 2.2 ADPCM Compression Performance Data (Bandwidth 200 – 2400 Hz)

BIT RATE (kbps)	BITS / SAMPLE (BIT RATE / 6600)	COMPRESSION RATIO	SNR (dB)
16	2.42	3.3	14.2
24	3.63	2.2	21.3
32	4.85	1.65	28.8

TABLE 2.3 ADPCM "Residual Encoder" Compression Performance Data (Bandwidth 0 – 2900 Hz)

BIT RATE (kbps)	BITS / SAMPLE (BIT RATE / 6400)	COMPRESSION RATIO	SNR (dB)
16	2.5	3.2	17.8
24	3.75	2.13	23.8
32	5	1.6	26.2

The SNR performance of the "residual encoder" is detailed in a SNR versus bit rate plot. The SNR for the residual encoder was measured using the following three sentences.

1) "Cats and dogs hate each other" (male speaker).
2) "Move the vat over the hot fire" (male speaker).
3) "The pipe began to rust while new" (female speaker).

The input data was low pass filtered at 2900 Hz and sampled at 6400 samples /s.

The table of values was obtained by extrapolation from the plot at bit rates that are of significance. The compression ratio values were obtained by comparing the bits/sample values obtained for the residual encoder system to the standard 8 bits/sample value used for PCM coders.

2.7.2 CELP Systems

The CELP system under performance analysis can be found in [10]. It is a variable-bit-rate low-delay CELP (VBR-LD-CELP) coder that operates at a nominal rate of 16 kbps, but has bit rate range of 24 kbps to 12.8 kbps. The performance of the algorithm was tested using three male and three female speech files, each of about 3 seconds in length. The segmental SNR (refer to [7]) was the performance measure used to evaluate the algorithm for both the 16kbps and 24 kbps modes. The values obtained are shown in Table 2.4.

Table 2.4 Simulation Results for Variable-Bit-Rate Low-Delay CELP (VBR-LD-CELP) Coder

Speech File	**Seg. SNR (dB)**		
	16 kbps	24 kbps	Gain
Male-1	17.0	21.4	4.4
Male-2	15.7	20.7	5.0
Male-3	15.2	19.5	4.3
Female-1	19.4	23.6	4.2
Female-2	21.2	25.7	4.5
Female-3	20.9	24.6	3.7
Average	18.2	22.6	4.4

Chapter 3 Transform Coding Compression Techniques

3.1 Linear Transforms

A one-dimensional linear transform may be expressed as shown below.

$$\theta_n = \sum_{i=0}^{N-1} x_i a_{n,i}$$

Where θ_n = transformed sequence

x_n = original sequence

This transform is called the forward transform.

The original sequence may be recovered using the inverse transform.

$$x_n = \sum_{i=0}^{N-1} \theta_i b_{n,i}$$

These 1-D transforms may be expressed in matrix format as shown below.

$$\theta = Ax$$

$$x = B\theta$$

Where **A** and **B** are NxN matrices.

The forward and inverse matrices are inverses of each other, that is, $\mathbf{AB} = \mathbf{BA} = \mathbf{I}$. The transforms that will be discussed will all be orthonormal transforms. An orthonormal transform has the property that the inverse of the matrix is simply its transpose, that is, $\mathbf{B} = \mathbf{A}^{-1} = \mathbf{A}^{T}$.

3.2 The Significance of Decorrelation in Transform Coding Schemes

The key factor for a transform coding technique being able to perform data compression applications is the ability of the technique to decorrelate the input signal. What follows is an intuitive discussion of the process of decorrelation that will serve to better explain how such a process can aid in data compression applications [8].

Suppose we desire to transmit a sinusoid across a medium. The signal can be transmitted as a sampled waveform, with each sample being sent in a sequential manner. The greater the number of sampled values transmitted will result in a better reconstruction at the receiver end. However, it is known that all that is required to reconstruct a deterministic sinusoid is its magnitude, phase, frequency, starting time, and the fact that it is a sinusoid. Thus, assuming no degradation by the medium, only five parameters need to be transmitted to reconstruct the signal at the receiver. From an information theoretic point of view, the sampled values of the sinusoid are highly correlated, and the information content of the transmitted signal is low. On the other hand, the five parameters of magnitude, phase, frequency, starting time, and shape are completely uncorrelated, and have the same amount of information content as the sampled values. It is the goal of transform coding techniques to represent a correlated input signal as completely uncorrelated pieces of information, thereby reducing the number of pieces of information required to represent the corresponding signal.

3.3 Transforms of Interest

3.3.1 Karhunen-Loeve Transform

The Karhunen-Loeve Transform (KLT) is a data-dependent transform that adjusts to the changing characteristics of a non-stationary source output. The rows of the discrete Karhunen-Loeve transform consist of the eigenvectors of the auto-covariance matrix. The auto-covariance matrix for a random process X is a matrix whose (i,j)th element $[R]_{i,j}$ is given by [1]:

$$[R]_{i,j} = E\left[X_n X_{n+|i-j|}\right]$$

The procedure for obtaining the KLT is described in the following paragraphs [8]. We seek to obtain the best representation of a given random signal with regards to mean-square error (MSE). Consider N sample points of a zero mean random vector x given by:

$$\vec{x} = \{x(0), x(1), \ldots, x(N-1)\}^T$$

If $\{\Phi_i\}$ is a set of linearly independent vectors spanning the N-dimensional vector space, vector x can be expanded in terms of Φ_i's:

$$\vec{x} = \sum_{i=0}^{N-1} X_i \vec{\Phi}_i$$

where X_i are the coefficients of expansion given by

$$X_i = <\vec{x}, \vec{\Phi}_i> / <\vec{\Phi}_i, \vec{\Phi}_i> \qquad \text{For } i = 0,1, \ldots, N-1.$$

And <•, •> denotes the inner product.

The vector x can be represented by the N coefficients X_i, as well as, the N signal values x(i). If only the first D coefficients (D < N) are significantly different from zero, then the vector x can be well represented by the D coefficients instead of N coefficients in the $\{\Phi_i\}$space. This means that data compression and bandwidth reduction are possible.

The truncated representation of vector x is given by:

$$\tilde{x} = \sum_{i=0}^{D-1} X_i \vec{\Phi}_i$$

The MSE with regard to the truncated version is given by:

$$\varepsilon = E\left[(\vec{x} - \tilde{x})^2\right]$$

$$\varepsilon = E\left[\left\langle \sum_{i=D}^{N-1} X_i \vec{\Phi}_i, \sum_{i=D}^{N-1} X_i \vec{\Phi}_i \right\rangle\right]$$

Where E is the expectation operator.

For orthonormal basis functions we have

$$< \vec{\Phi}_i, \vec{\Phi}_k >= \delta_{i,k}$$

If vector x is real

$$\varepsilon = E\left[\sum_{i=D}^{N-1} |X_i|^2\right]$$

$$\varepsilon = E\left[\sum_{i=D}^{N-1} |< \vec{x}, \vec{\Phi}_i >|^2\right]$$

This equation can be reduced further to

$$\varepsilon = E\left[\sum_{i=D}^{N-1} \vec{\Phi}_i^T \vec{x}\vec{x}^T \vec{\Phi}_i\right]$$

$$\varepsilon = \sum_{i=D}^{N-1} \vec{\Phi}_i^T E\left[\vec{x}\vec{x}^T\right] \vec{\Phi}_i$$

It should be noted that auto-covariance matrix is given by:

$$[A] = E\lfloor xx^T \rfloor$$

If one seeks to minimize the MSE by the proper choice of basis functions, subject to the orthonormality condition mentioned previously, one obtains the following:

$$\left(\frac{\delta}{\delta \vec{\Phi}_i}\right)\left\{\varepsilon - \mu_i < \vec{\Phi}_i, \vec{\Phi}_i >\right\} = 0$$

This leads to the following characteristic equation

$$([A] - \mu_i [I_N])\vec{\Phi}_i = 0 \qquad \text{For i} = 0, 1, \ldots, \text{N-1}$$

Where μ_i is the Lagrange multiplier introduced for the purpose of satisfying the constraint, and $[I_N]$ is the NxN identity matrix. From the characteristic equation we see that the minimization of the MSE involves solving for the corresponding eigenvalues. The set of basis vectors obtained will diagonalize the auto-covariance matrix [A]. Let $[\phi]$ be the set of eigenvectors given by:

$$[\phi] = [\vec{\Phi}_0, \vec{\Phi}_1, \ldots, \vec{\Phi}_{N-1}]$$

Then

$$[\phi]^{-1}[A][\phi] = diag[\mu_0, \mu_1, \ldots, \mu_{N-1}]$$

Based on the forgoing, the MSE due to truncation is given by:

$$\varepsilon = \sum_{i=D}^{N-1} \mu_i$$

and we note that ε is minimized by ranking the eigenvalues μ_i in descending order.

The set of basis vectors $\{\Phi_i\}$ form the bases for the KLT expansion, and a matrix consisting of N such eigenvectors constitute the KLT matrix.

The following properties make the KLT an optimal transform [8].

1) It completely decorrelates the signal in the transform domain.
2) It minimizes the MSE in bandwidth reduction or data compression.
3) It contains the most variance (energy) in the fewest number of transform coefficients.
4) It minimizes the total entropy of the sequence.

If the source output is non-stationary, its auto-covariance function changes with time. This means that the auto-covariance matrix changes with time, and the KLT will have to be recomputed. For a transform of reasonable size, these computations are significant. Also, since the auto-covariance is calculated from the source output, this information is not available at the receiver. Therefore, the incurred overhead in transmission, combined with significant computations generally negate the advantages gained by the optimal KLT, and render this transform impractical [1].

3.3.2 Discrete Cosine Transform

The Discrete Cosine Transform (DCT) is a non-data dependent transform that provides energy compaction performance that approaches the KLT for strongly correlated sources [1, 8]. The rows of an NxN DCT-III matrix C are given by [1]:

$$[C]_{i,j} = \begin{cases} \sqrt{\frac{1}{N}}\cos\frac{(2j+1)i\pi}{2N} & i = 0, j = 0,1,\ldots,N-1 \\ \sqrt{\frac{2}{N}}\cos\frac{(2j+1)i\pi}{2N} & i = 1,2,\ldots,N-1, j = 0,1,\ldots,N-1 \end{cases}$$

The DCT can be derived from the Discrete Fourier Transform (DFT). It provides better compression performance than the DFT. The reason for this is that in order to obtain the DFT coefficients of an N point sequence we assume the sequence to be periodic with a period N. The effect of this is shown in the Figure 3.1.

This representation introduces sharp discontinuities at the beginning and then end of each periodic sequence. The DFT represents these sharp discontinuities by having non-zero high frequency coefficients. Since these discontinuities occur only at the beginning and end of the N point sequence, their effect needs to be canceled out within the sequence. To do so, the DFT adjusts the other frequency coefficients accordingly. When the high frequency coefficients are removed during the compression process, the coefficients used to compensate for these high frequency coefficients introduce additional distortion in the reconstructed sequence.

The DCT can be obtained using the DFT by mirroring the N-point sequence to obtain a 2N-point sequence. The DCT is first N points of the 2N-point DFT. As shown in Figure 3.2 this 2N-point DFT does not possess any sharp discontinuities at the edges.

3.3.3 Discrete Sine Transform

The Discrete Sine Transform (DST) is a non-data dependent transform that provides energy compaction performance close to the KLT when the source does not have a high degree of correlation [1, 8]. The elements of an NxN DST-IV transform matrix are given by:

$$[S]_{i,j} = \left\{ \sqrt{\frac{2}{N+1}} \sin \frac{\pi(i+1)(j+1)}{N+1} \right. \qquad i, j = 0,1,\ldots,N-1$$

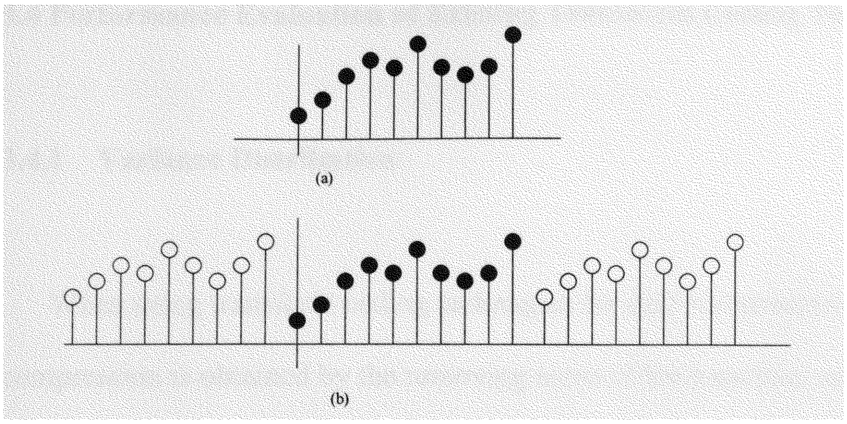


FIGURE 3.1 TAKING THE DISCRETE FOURIER TRANSFORM OF A SEQUENCE

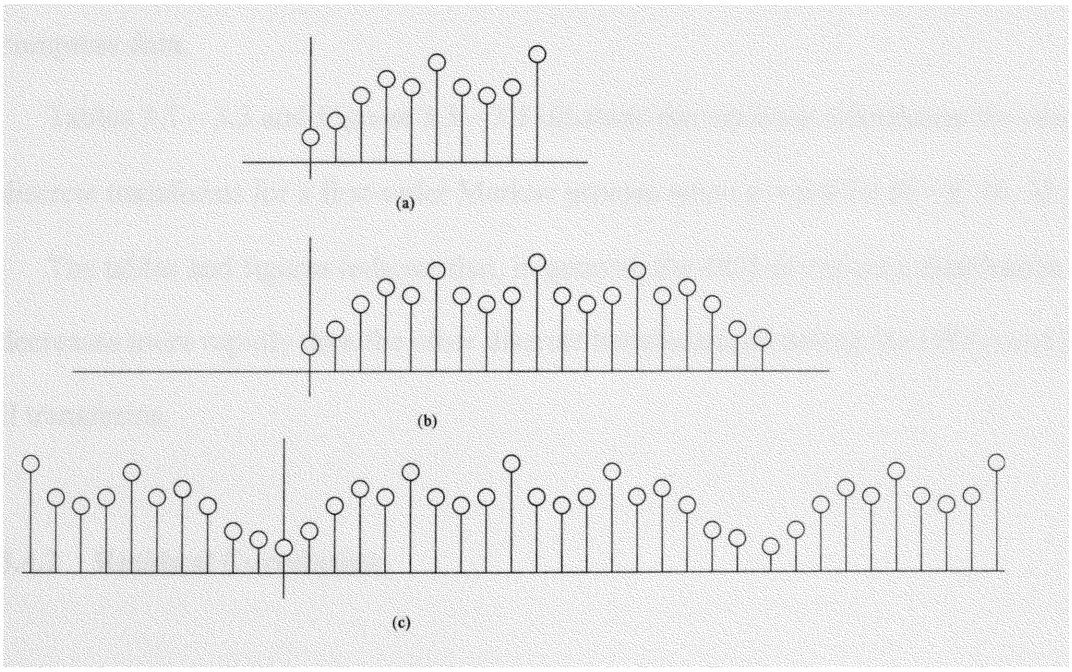


FIGURE 3.2 TAKING THE DISCRETE COSINE TRANSFORM OF A SEQUENCE

3.4 Performance Evaluation of Existing Transform Coding Techniques

3.4.1 Variance Distribution

When using transform coding techniques for data compression, some of the compression is obtained by the removing some of the transform coefficients from the transformed sequence. It is desirable to have only a few coefficients with large variances (energy) with the remaining transform coefficients possessing small variances. This will allow the coefficients with small variances to be removed without a large MSE occurring between the original and reconstructed signals. Therefore the variance distribution among the transform coefficients is indicative of how well the transform will be able to compress data.

Tables 3.1 – 3.3 and Figures 3.3 – 3.5 illustrate the variance distribution for various discrete transforms for a first-order Markov process when $\rho = 0.9$ for N = 8, 16, 32 [8].

The tables and figures indicate that, in general, the DCT-II variance distribution decreases more rapidly than the other discrete transforms, including the DST-I and DST-II transforms.

3.4.2 Residual Correlation

Another method of determining the performance of a discrete transform is determine how well it can decorrelate a given sequence. The optimal KLT transform completely decorrelates a Markov-I sequence. Other discrete, suboptimal transforms can be judged

TABLE 3.1 Variance Distribution For a First-order Markov Process Defined by $\rho = 0.9$ and N = 8. I is Ith Transform Coefficient.

I	WHT	ST	DCT-II	DST-I	DST-II	CMT	DFT	DLT
1	6.185	6.185	6.185	5.76	5.384	6.185	6.185	6.185
2	0.863	0.989	1.006	0.931	0.823	0.992	0.585	0.989
3	0.305	0.345	0.346	0.661	0.74	0.346	0.585	0.332
4	0.246	0.146	0.166	0.222	0.246	0.153	0.175	0.173
5	0.105	0.105	0.105	0.197	0.333	0.1	0.175	0.113
6	0.104	0.104	0.076	0.093	0.145	0.105	0.103	0.083
7	0.103	0.063	0.062	0.08	0.24	0.057	0.103	0.067
8	0.088	0.063	0.055	0.056	0.088	0.062	0.088	0.057

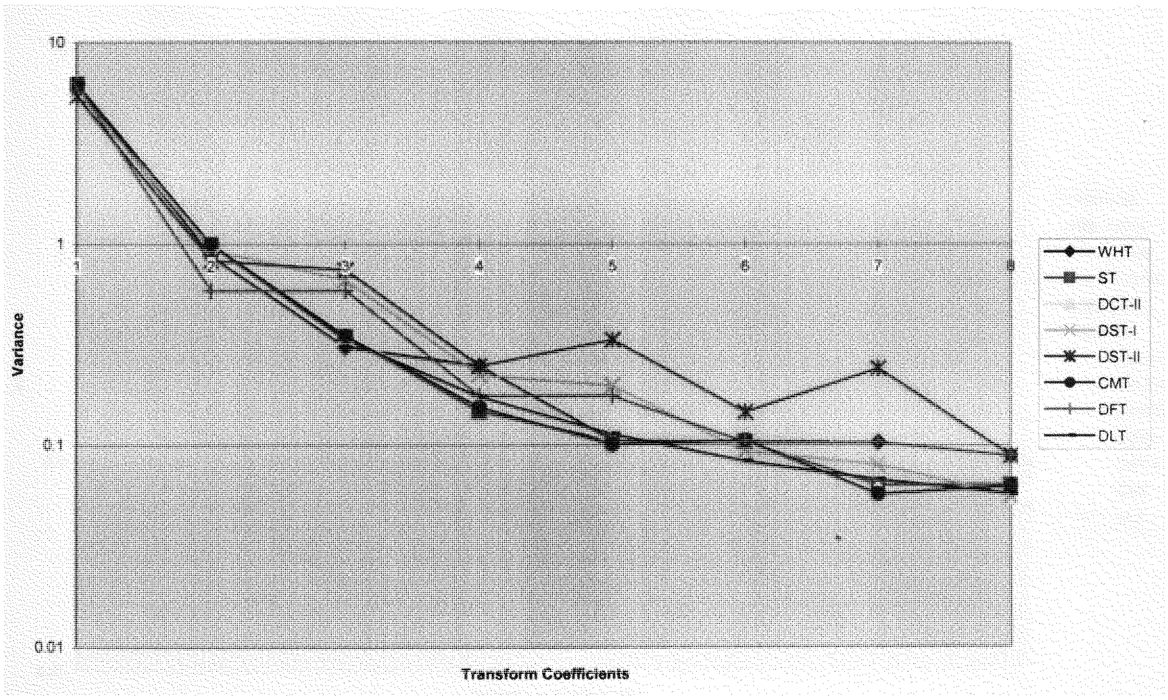


Figure 3.3 Variance Distribution for Various Discrete Transforms for N = 8 and $\rho = 0.9$

TABLE 3.2 Variance Distribution For a First-order Markov Process Defined by $\rho = 0.9$ and $N = 16$. I is Ith Transform Coefficient.

I	HT	CHT	WHT	DCT-II	DFT	ST	(SHT)1	$(HHT)_1$	$(HHT)_2$	DST-I	DST-II	CMT	DLT
1	9.8346	9.8346	9.8346	9.8346	9.8346	9.8346	9.8346	9.8346	9.8346	9.218	8.914	9.835	9.8346
2	2.5464	2.5364	2.536	2.9328	1.8342	2.8536	2.7765	2.5364	2.5364	2.64	2.458	2.885	2.8532
3	0.8638	0.8635	1.02	1.2108	1.8342	1.1963	1.0208	1.0209	1.0209	1.468	1.433	1.196	1.1356
4	0.8638	0.8635	0.706	0.5814	0.5189	0.461	0.467	0.7061	0.7061	0.709	0.689	0.489	0.5912
5	0.2755	0.2755	0.307	0.3482	0.5189	0.3468	0.3092	0.2946	0.3066	0.531	0.561	0.348	0.3656
6	0.2755	0.2755	0.303	0.2314	0.2502	0.3424	0.3031	0.2946	0.3031	0.314	0.332	0.283	0.2523
7	0.2755	0.2755	0.283	0.1664	0.2502	0.1461	0.2837	0.2562	0.2864	0.263	0.312	0.157	0.1875
8	0.2755	0.2755	0.206	0.1294	0.1553	0.146	0.2059	0.2562	0.2059	0.174	0.206	0.125	0.1471
9	0.1	0.1	0.105	0.1046	0.1553	0.1047	0.1042	0.1024	0.1038	0.153	0.211	0.101	0.1202
10	0.1	0.1	0.105	0.0876	0.1126	0.1044	0.1042	0.1024	0.1038	0.11	0.149	0.105	0.1014
11	0.1	0.1	0.104	0.076	0.1126	0.1044	0.1034	0.1024	0.1034	0.099	0.162	0.105	0.0878
12	0.1	0.1	0.104	0.0676	0.0913	0.0631	0.1034	0.1024	0.1034	0.078	0.121	0.105	0.0775
13	0.1	0.1	0.103	0.0616	0.0913	0.0631	0.101	0.0976	0.1013	0.071	0.138	0.062	0.0695
14	0.1	0.1	0.102	0.0574	0.0811	0.0631	0.101	0.0976	0.1013	0.061	0.107	0.079	0.0634
15	0.1	0.1	0.098	0.0548	0.0811	0.0631	0.0913	0.0976	0.0913	0.057	0.128	0.057	0.0585
16	0.1	0.1	0.078	0.0532	0.078	0.0631	0.0913	0.0976	0.0913	0.054	0.073	0.07	0.0544

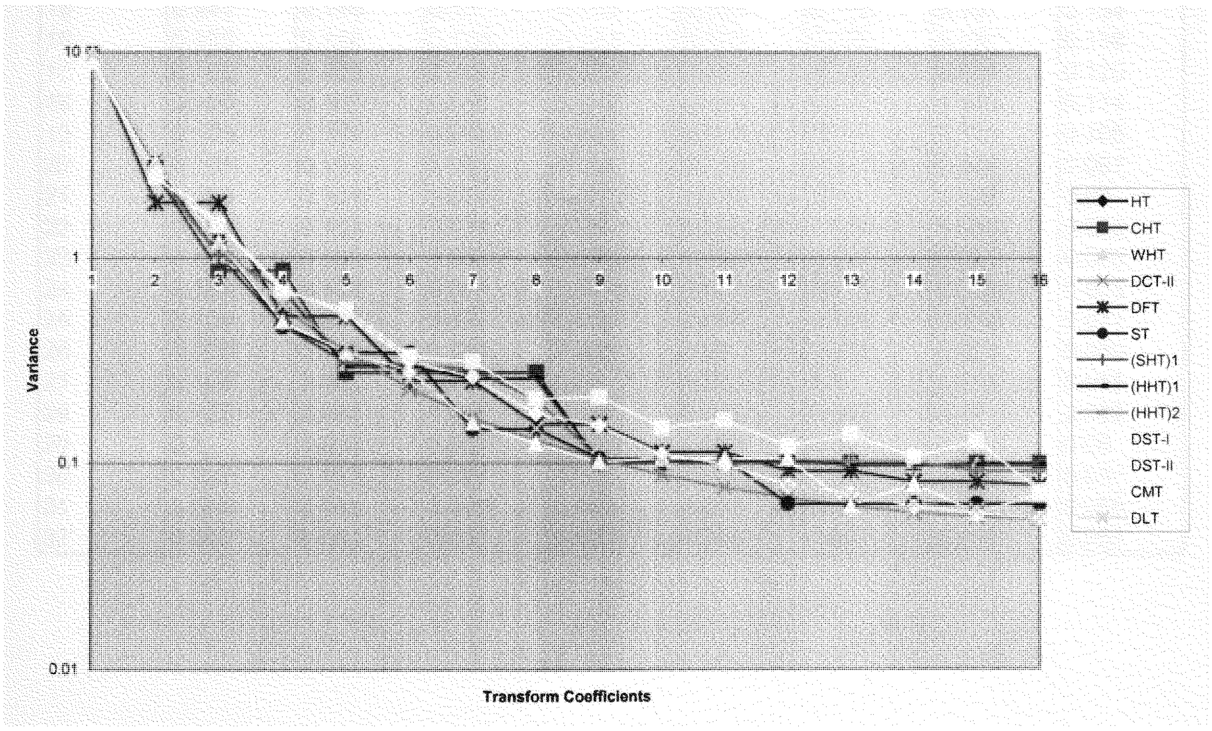


Figure 3.4 Variance Distribution for Various Discrete Transforms for $N = 16$ and $\rho = 0.9$

TABLE 3.3 Variance Distribution For a First-order Markov Process Defined by $\rho = 0.9$ and $N = 32$. I is Ith Transform Coefficient.

I	WHT	ST	DCT-II	DST-I	DST-II	CMT	DFT	DLT
1	13.568	13.569	13.569	13.244	13.08	13.568	13.568	13.568
2	6.101	6.59	6.847	6.352	6.157	6.611	4.938	6.59
3	3.128	3.658	3.72	3.46	3.352	3.659	4.938	3.423
4	1.965	1.561	2.023	2.048	1.983	1.602	1.613	1.986
5	1.046	1.23	1.244	1.386	1.355	1.229	1.613	1.273
6	0.995	1.163	0.832	0.96	0.943	0.916	0.77	0.881
7	0.867	0.464	0.596	0.729	0.726	0.511	0.77	0.646
8	0.545	0.46	0.448	0.5551	0.553	0.359	0.452	0.496
9	0.308	0.349	0.35	0.446	0.456	0.35	0.452	0.394
10	0.305	0.345	0.282	0.356	0.367	0.299	0.3	0.321
11	0.303	0.343	0.233	0.3	0.318	0.298	0.3	0.268
12	0.303	0.342	0.197	0.249	0.266	0.286	0.217	0.229
13	0.293	0.146	0.169	0.216	0.238	0.273	0.217	0.198
14	0.28	0.146	0.147	0.184	0.204	0.165	0.167	0.173
15	0.247	0.146	0.13	0.163	0.187	0.158	0.167	0.154
16	0.165	0.146	0.116	0.142	0.165	0.155	0.135	0.138
17	0.105	0.105	0.105	0.128	0.154	0.129	0.135	0.125
18	0.105	0.105	0.096	0.114	0.138	0.116	0.113	0.114
19	0.104	0.104	0.088	0.104	0.131	0.116	0.113	0.105
20	0.104	0.104	0.081	0.094	0.119	0.11	0.098	0.097
21	0.104	0.104	0.076	0.087	0.115	0.11	0.098	0.09
22	0.104	0.104	0.071	0.08	0.106	0.108	0.087	0.085
23	0.104	0.104	0.068	0.075	0.104	0.107	0.087	0.079
24	0.104	0.104	0.064	0.07	0.097	0.105	0.079	0.075
25	0.104	0.063	0.062	0.066	0.096	0.105	0.079	0.071
26	0.103	0.063	0.059	0.062	0.09	0.104	0.074	0.068
27	0.103	0.063	0.057	0.06	0.09	0.1	0.074	0.065
28	0.102	0.063	0.056	0.057	0.086	0.082	0.07	0.062
29	0.1	0.063	0.055	0.056	0.087	0.08	0.07	0.06
30	0.097	0.063	0.054	0.054	0.084	0.073	0.068	0.057
31	0.088	0.063	0.053	0.053	0.085	0.062	0.068	0.055
32	0.068	0.063	0.053	0.053	0.068	0.057	0.068	0.053

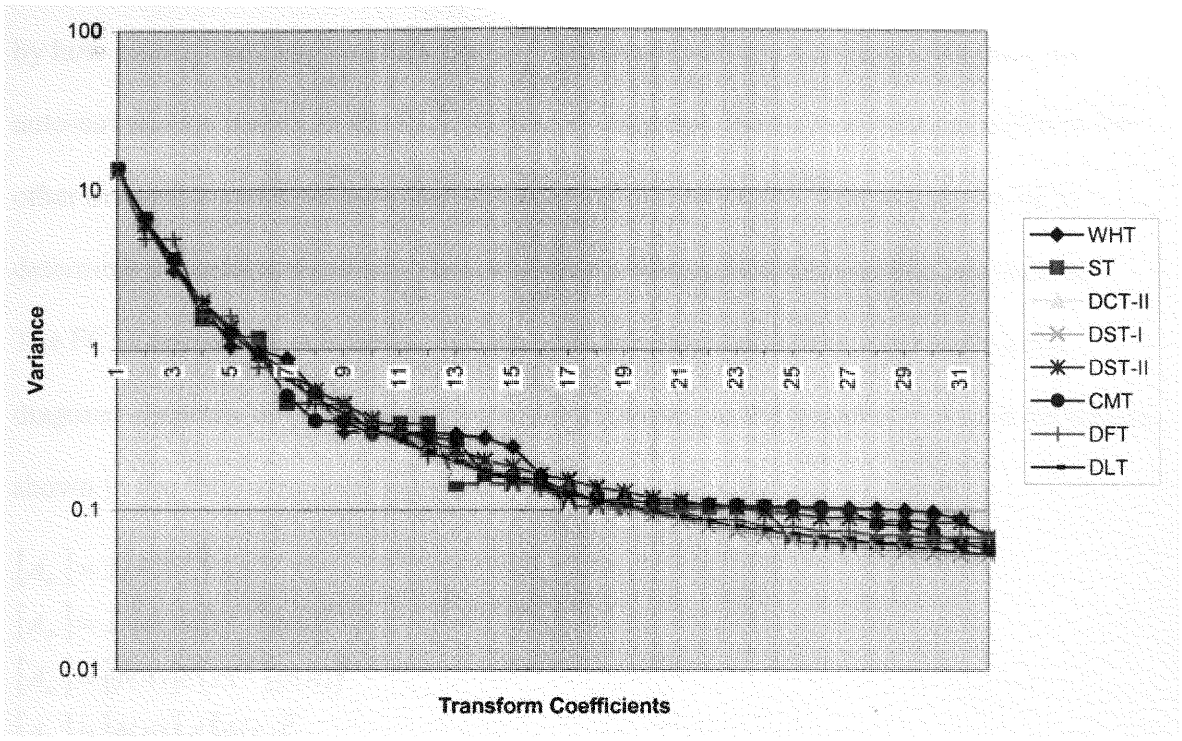


Figure 3.5 Variance Distribution for Various Discrete Transforms for N = 32 and $\rho = 0.9$

by how closely they approximate the KLT in decorrelating a sequence. Whereas the auto-covariance matrix in the KLT domain is diagonal, the auto-covariance matrices for other discrete transforms have nonzero off-diagonal elements. The measure used to determine decorrelation ability measures the Fractional Correlation (FC) left undone. The FC is developed as follows. Let $[A_{tr}]_{jj}$ be the diagonal matrix representing the diagonal elements of $[A_{tr}]$, the auto-covariance matrix in the transform domain defined as shown in the following expression.

$$\begin{aligned}
[A_{tr}] &= E\left[\vec{X}\vec{X}^T\right] \\
[A_{tr}] &= E\left([\phi(n)]\vec{x}\vec{x}^T[\phi(n)]^T\right) \\
[A_{tr}] &= [\phi(n)]E\left(\vec{x}\vec{x}^T\right)[\phi(n)]^T \\
[A_{tr}] &= [\phi(n)][A][\phi(n)]^T
\end{aligned}$$

Where [A] is the auto-covariance matrix in the time domain.

An auto-covariance matrix [A'] can be obtained by inverse transforming $[A_{tr}]_{jj}$ such that

$$[A'] = [\phi(n)]^T[A_{tr}]_{jj}[\phi(n)]$$

The FC left undone by the transform is then defined as

$$FC = \frac{\left|[A]-[A']\right|^2}{\left|[A]-[I_N]\right|^2}$$

Where $|\;|^2$ denotes the Hilbert-Schmidt weak norm [8]. For the KLT, [A] = [A'], and thus FC is zero.

Tables 3.4 – 3.5 and Figures 3.6 – 3.8 show FC versus ρ (the correlation coefficient for Markov-I processes), for N = 8, 16, and 32, and for a variety of discrete transforms.

From the tables and figures it can be seen that, in general, the DCT-II (and hence all the DCT's) is superior to all the other transforms based on the degree of correlation. For

TABLE 3.4 Residual Correlation Left Undone for a First-Order Markov Process N = 8.

rho	WHT	DFT	ST	DCT-II	CMT	DST-I	DST-II	DLT
0.1	24.583	12.61	21.39	9.012	13.2	0.135	10.066	22.2
0.2	23.365	12.93	18.64	8.422	12.5	0.405	10.684	19.4
0.3	21.49	13.44	16.03	7.881	11.6	0.929	11.551	16.7
0.4	19.1	13.93	13.44	7.25	10.3	1.751	12.698	14
0.5	16.24	13.99	10.81	6.424	8.7	2.966	14.239	11.2
0.6	12.86	12.96	8.1	5.307	6.8	4.742	16.363	8.3
0.7	8.97	10.31	5.34	3.85	4.7	7.323	19.339	5.5
0.8	4.86	6.18	2.75	2.166	2.5	11.006	23.479	2.8
0.9	1.425	1.95	0.077	0.661	0.74	16.008	28.991	0.8

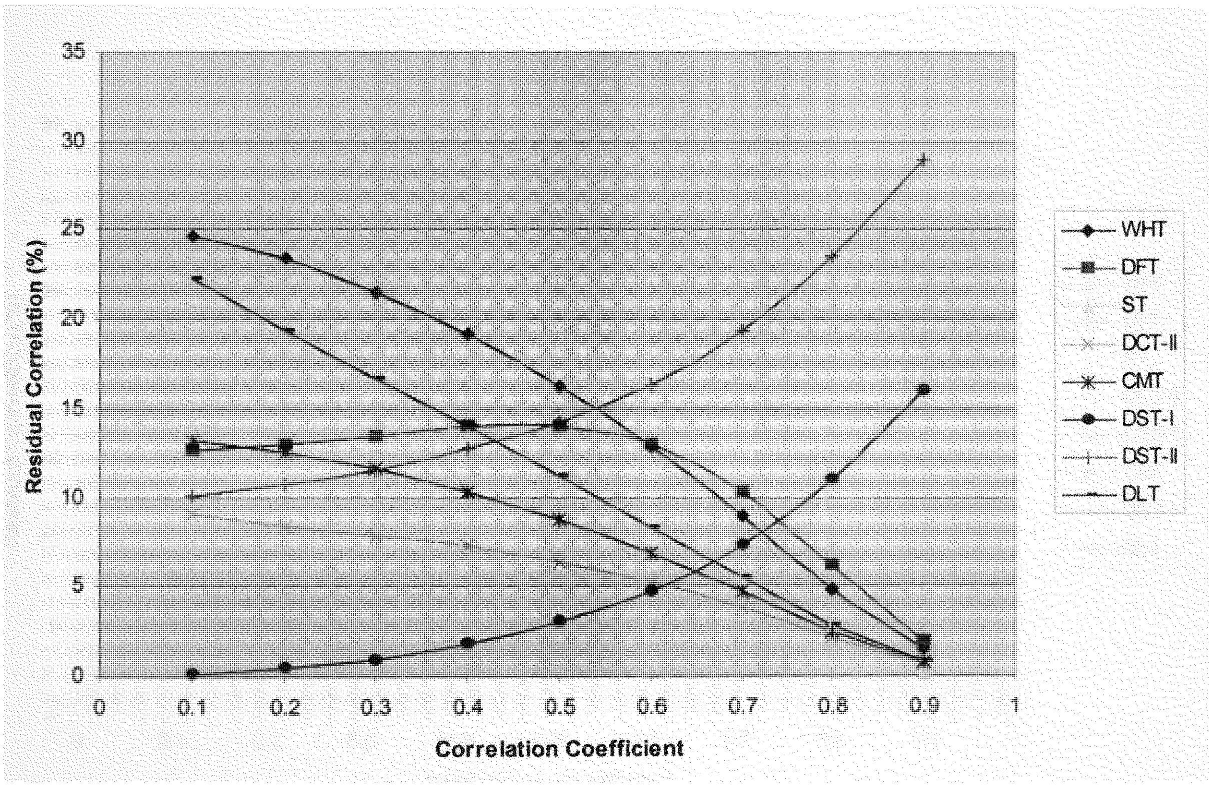


Figure 3.6 Residual Correlation Versus the Correlation Coefficient ρ for N = 8.

TABLE 3.5 Residual Correlation Left Undone for a First-Order Markov Process

N = 16.

rho	WHT	DFT	ST	DCT-II	DST-II	CMT	DST-I	DLT
0.1	28.8	6.31	25.1	5.392	5.68	17.2	0.136	26.19
0.2	27.7	6.49	22.54	5.321	5.05	16.4	0.251	23.77
0.3	26.04	6.82	20.08	5.315	6.35	15.1	0.55	21.27
0.4	23.97	7.34	17.67	5.347	6.93	13.6	1.038	18.77
0.5	21.54	8.13	15.25	5.377	7.78	11.9	1.787	16.2
0.6	18.69	9.24	12.69	5.328	9.06	10.1	2.954	13.45
0.7	15.2	10.37	9.84	5.006	11.11	8	4.865	10.34
0.8	10.56	9.96	6.4	3.99	14.76	5.5	8.28	6.63
0.9	4.3	5.23	2.41	1.83	21.84	2.2	14.962	2.45

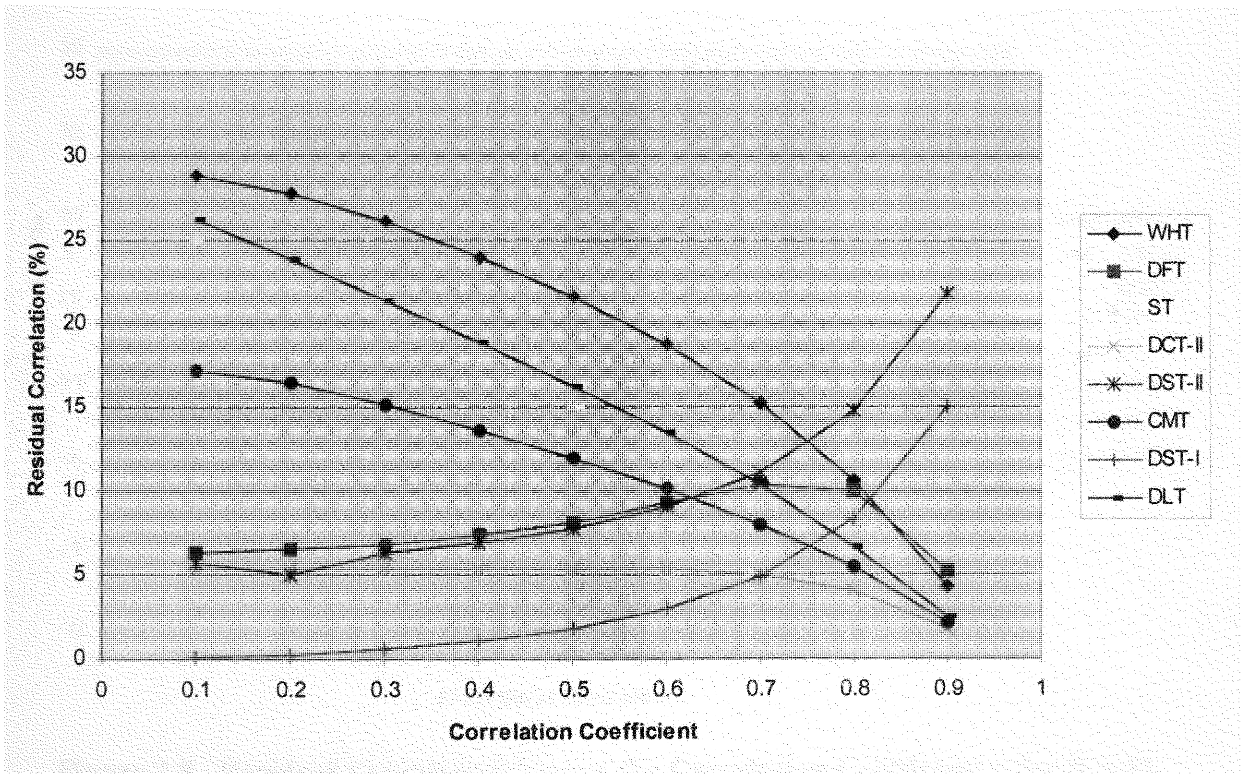


Figure 3.7 Residual Correlation Versus the Correlation Coefficient ρ for N = 16.

TABLE 3.6 Residual Correlation Left Undone for a First-Order Markov Process N = 32.

rho	WHT	DST-II	DCT-II	DFT	ST	CMT	DST-I	DLT
0.1	30.9	2.96	2.88	3.16	27.06	24.4	0.058	28.6
0.2	29.9	3.11	2.94	3.25	24.67	23.4	0.131	26.1
0.3	28.4	3.31	3.03	3.42	22.34	22	0.294	23.7
0.4	26.5	3.59	3.17	3.7	20.09	20.2	0.558	21.7
0.5	24.4	4.03	3.38	4.12	17.88	18.1	0.968	19.1
0.6	22	4.71	3.67	4.79	15.67	15.7	1.618	16.7
0.7	19.2	5.86	4.03	5.92	13.2	13.1	2.723	14.1
0.8	15.6	8.09	4.3	7.89	10.21	10	4.876	10.8
0.9	9.3	13.79	3.44	8.72	5.567	5.6	10.403	5.8

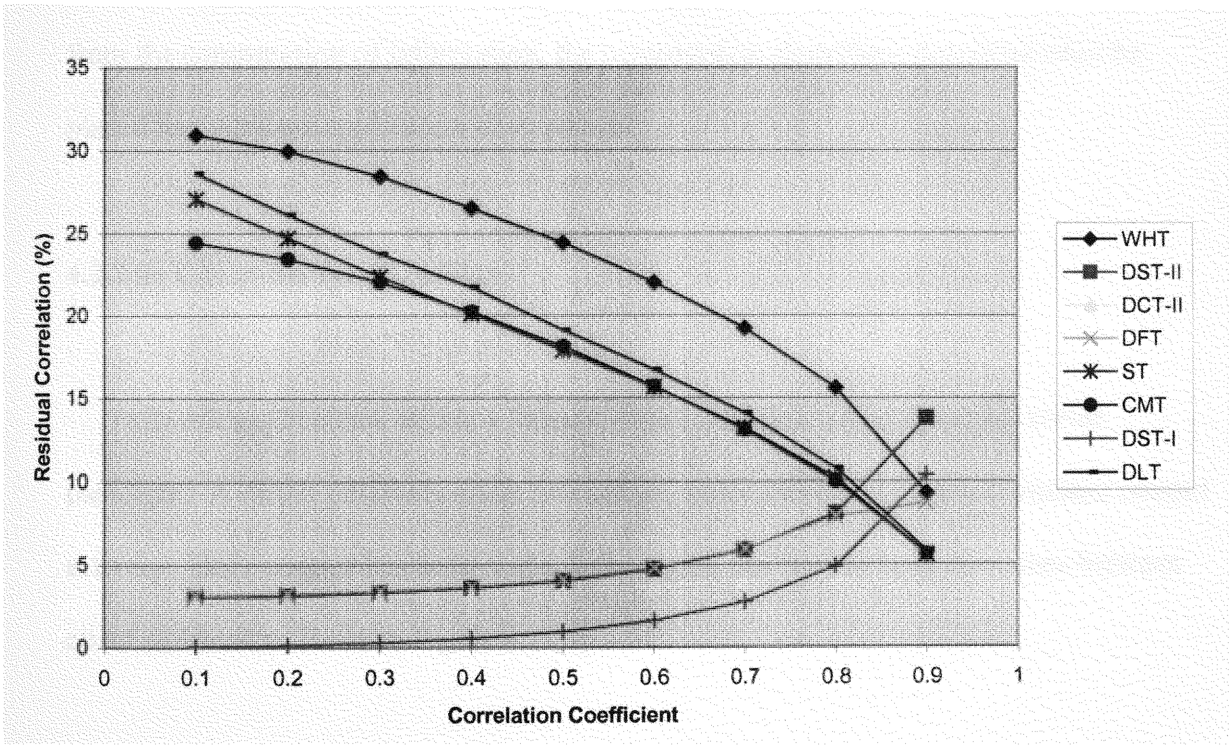


Figure 3.8 Residual Correlation Versus the Correlation Coefficient ρ for N = 32.

Chapter 4 Network Applications

4.1 Introduction

Network applications occupy the top layer of a network protocol stack. Although an Internet system provides basic communication of information from computer-to-computer, application programs are required to define the format in which the information will be displayed and the mechanisms users have to select or access the information. To take advantage of the reduction in Internet bandwidth usage that results from the compression of information, the compression algorithms described must be placed in a network application environment.

4.2 The Client-Server Paradigm

Network applications utilize a form of communication called the client-server paradigm. A server passively waits for contact, while a client application initiates contact actively. Once contact has been made, information can flow in both directions between client and server. Refer to Figure 4.1 [9].

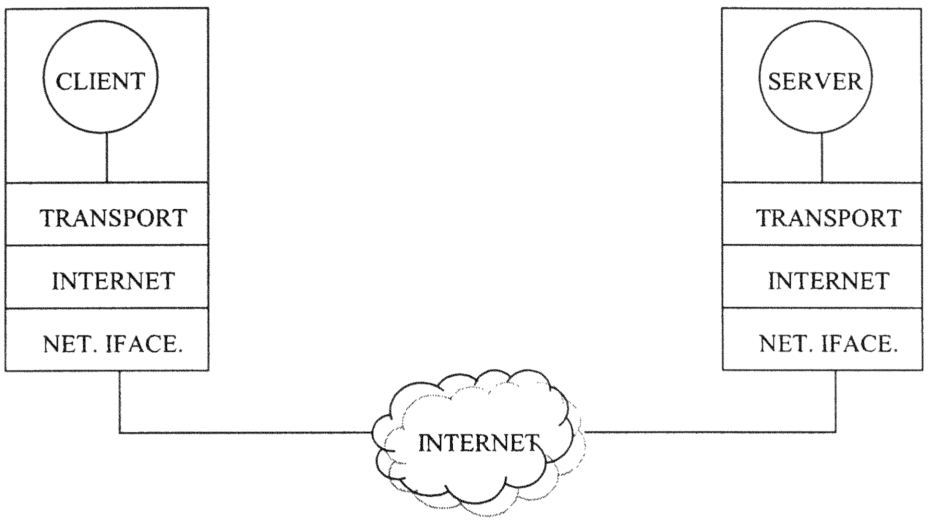


FIGURE 4.1 CLIENT-SERVER INTERACTION

It is possible to have multiple server applications being run on a single computer at the same time. This functionality is made possible by the fact that transport protocols assign unique identifiers to each service. In TCP, 16-bit integer values called protocol port numbers are used. Clients and servers specify the service identifiers, and the protocol software uses the identifiers to route the incoming request to the appropriate server.

It is also possible to have multiple copies of the same service running on a single computer at the same time. This is made possible by the use of service identifiers for each client as well as for each service. The server machine uses the combination of client and server identifiers to choose the correct copy of the concurrent server.

4.3 The Socket Interface

In order for an application program to obtain Internet functionality, a set of procedures must be put in place to specify how an application program can interact with the transport protocol software. The interface between an application program and the communications protocols of an operating system is called an Application Program Interface or API. It is in the API that the set of procedures defining application/transport protocol interaction is specified. The de facto standard is the socket API [9].

In the socket API, communication between client and server must occur through a socket. Sockets are integrated with I/O. This means that an application communicates through a socket in a manner similar to how an application transfers data to or from a file, that is, the open-read-write-close paradigm is used.

When an application creates (opens) a socket, the procedure used returns a descriptor (an integer) to identify the socket. This descriptor is passed as an argument when the application calls procedures to perform data transfers. Finally, when the application is finished with its required data transfers, it calls a close procedure to close the socket

4.4 Socket Procedures [9]

The socket procedure creates a socket and returns an integer descriptor. The procedure format is shown below.

$descriptor = socket(protofamily, type, protocol)$

Argument protofamily specifies the protocol family to be used with the socket. Argument type specifies the type of communication to be used with the socket. For example, this argument could take the value SOCK_STREAM for connection-oriented communication or SOCK_DGRAM for connectionless-oriented communication. Argument protocol specifies the particular transport protocol used with the socket.

The close procedure terminates the use of a socket. It takes the following form.

$close(socket)$

Argument socket is the descriptor of the socket to be closed. If the socket is using a connection-oriented protocol, close terminates the connection before closing the socket.

A server application uses the bind procedure to supply the protocol port number at which the server will wait for contact. The procedure has the following format.

$bind(socket, localaddr, addrlen)$

Argument socket is the descriptor of a socket created but is not yet bound. Argument localaddr is a structure that specifies the local address to be assigned to the socket. This structure contains data members that specify the IP address of the local computer, as well as, the protocol port number. The argument addrlen is an integer that specifies the length of the address.

The listen procedure is used by the server application to instruct the operating system to put the socket into a passive mode in order that the socket may wait for contact from client applications. The procedure takes the following form.

$listen(socket, queuesize)$

Argument socket is the descriptor that has been created and bound, and argument queuesize specifies the length of a socket's request queue.

If a server uses connection-oriented transport it must call the accept procedure in order to accept the next connection request. If a request is already in the queue, then accept returns immediately. If no requests have arrived, the system blocks the server until a client seeks to form a connection. The procedure takes the following format.

$newsock = accept(socket, caddress, caddresslen)$

Argument socket is the descriptor of the socket that the server has created and bound. Argument caddress is the address of a structure of type sockaddr, and caddresslen is a pointer to an integer. The accept procedure fills the sockaddr structure pointed to by caddress with the address of the client that seeks to form a connection, and sets caddresslen to the length of this address. The accept procedure also creates a new socket for the connection and returns a descriptor for this socket to the caller. The server uses this socket to communicate with the client and closes the socket when finished.

Meanwhile the server's original socket remains unchanged; after it finishes communicating with the client, the server uses the original socket to accept the next connection from a client.

It should be noted that for connectionless-oriented communication, a server would need only to call the socket and bind procedures to be ready to accept messages. However, for connection-oriented communication, the socket, bind, listen, and accept procedures must be used.

The connect procedure is used by clients to establish a connection to a specific server. The form is shown below.

$connect = (socket, saddress, saddresslen)$

Argument socket is the descriptor of the socket to be used by the client to make the connection. Argument saddress is a sockaddr structure that specifies the server's address and protocol number, and saddresslen is the length of the server's address in octets.

If a socket is connected, the send procedure can be used to send data. The send procedure takes the following format.

$send(socket, data, length, flags)$

Argument socket is the descriptor of the socket to use. Argument data is the address in memory of the data to send. Argument length is an integer that specifies the number of octets of data, and argument flags contains bits that request special options.

Procedure sendto allows a client and a server to send data using an unconnected socket. The procedure possesses the following format.

$sendto(socket, data, length, flags, destaddress, addresslen)$

The first four arguments correspond to the four arguments of the send procedure. Argument destaddress specifies the address of a destination, and argument addrlen gives the length of that address.

The sendmsg procedure performs the same operation as the sendto procedure, but abbreviates the arguments by defining a structure. The procedure has the following format.

sendmsg(socket, msgstruct, flags)

Argument msgstruct is a structure that contains information about the destination address, the address length, the message to be sent, and the length of the message. The other two arguments are the same as those described previously.

A client and server each need to receive data sent by the other. The recv procedure can be used to receive data from a connected socket. The procedure has the following format.

recv(socket, buffer, length, flags)

Argument socket is the descriptor of the socket from which the data is to be received. Argument buffer is the address in memory in which the received data will be stored. Argument length specifies the size of the buffer in octets, and argument flags allows the caller to control details.

With unconnected sockets, you are able to receive data from an arbitrary set of clients. In such cases, it is required that the system returns the address of the sender in addition to the data sent. The recvfrom procedure can be used to receive both the sender's address and the message. The procedure has the following format.

recvfrom(socket, buffer, length, flags, sndraddr, saddrlen)

The first four arguments correspond to the four arguments of the recv procedure. Arguments sndraddr and saddrlen are used to determine the sender's IP address.

The recvmsg procedure performs the same operation as the recvfrom procedure, but abbreviates the arguments by defining a structure. The procedure has the following format.

$recvmsg(socket, msgstruct, flags)$

Argument msgstruct gives the address of a structure that contains information about the sender's address, the address length, an address to store the incoming message, and the length of the message. The other two arguments are the same as those described previously.

Chapter 5 Compression Performance Analysis

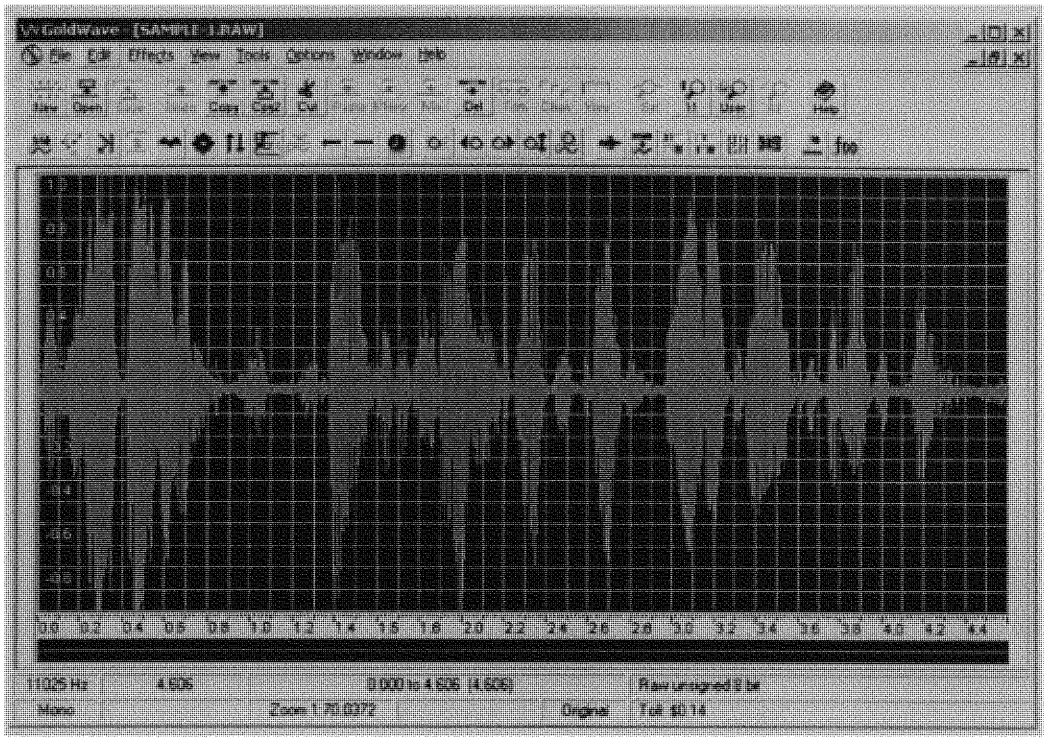
5.1 Performance Analysis Overview

To determine the efficiency of the transform coding compression techniques, these techniques, along with some lossless and alternative lossy compression techniques, were applied to a data set consisting of six audio sample files.

These audio sample files varied in length from one to seven seconds and possessed divergent statistical characteristics. The six sample files are shown in the Figures 5.1 – 5.6. Each audio file was sampled at a rate of 11 kHz. Eight bits were used to represent each sample, which equates to a magnitude range of –128 to 127.

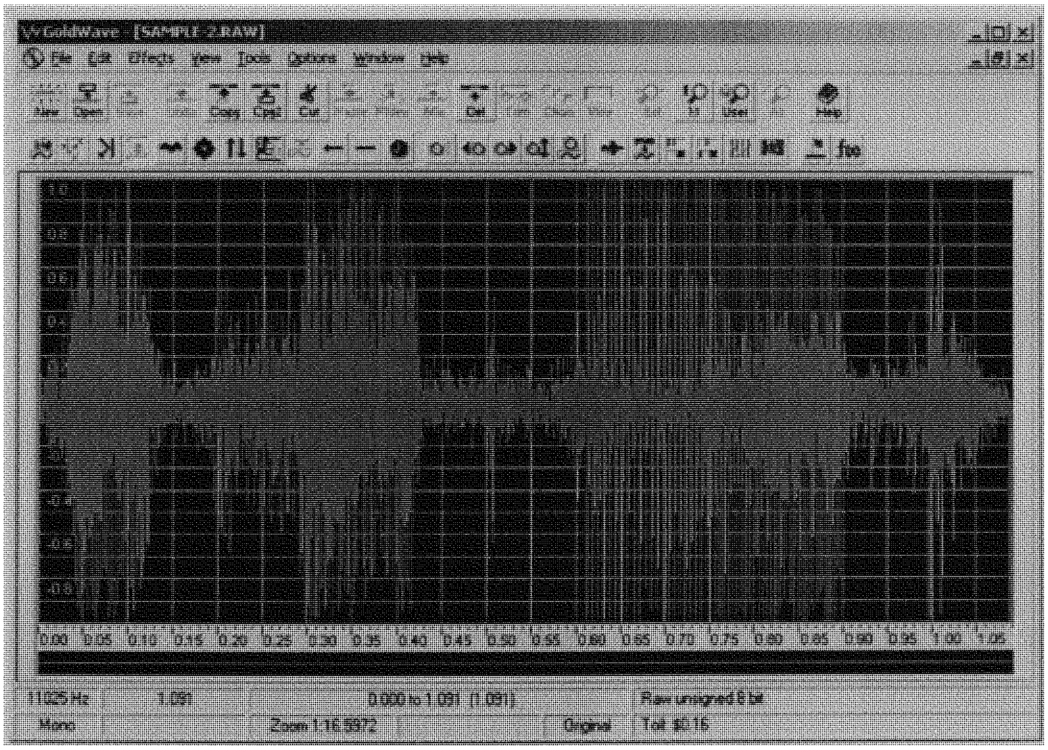
The audio file SAMPLE-1.RAW contains the phrase, spoken by a male speaker, "If you are following the book and find that a song is not recorded, merely turn to the next page." The audio file SAMPLE-2.RAW contains the phrase, spoken by a male speaker, "What's up doc!" The audio file SAMPLE-3.RAW contains a section of a rock song with a male vocalist. The section contains the lyrics, "Born to be wild!" The audio file SAMPLE-4.RAW contains the phrase, spoken by a young female speaker, "Good night Katie." The audio file SAMPLE-5.RAW contains the phrase, spoken by a young male speaker, "Now here's something we hope you'll really like!" The audio file SAMPLE-6.RAW contains the phrase, spoken by a male speaker, "And now for something completely different."

The following algorithms were used in the analysis process:



11025 Hz 4.606 0.000 to 4.606 (4.606) Raw unsigned 8 bit
Mono Zoom 1:70.0372 Original Tot: $0.14

FIGURE 5.1 SAMPLE-1



11025 Hz 1.091 0.000 to 1.091 (1.091) Raw unsigned 8 bit
Mono Zoom 1:16.5972 Original Tot: $0.16

FIGURE 5.2 SAMPLE-2

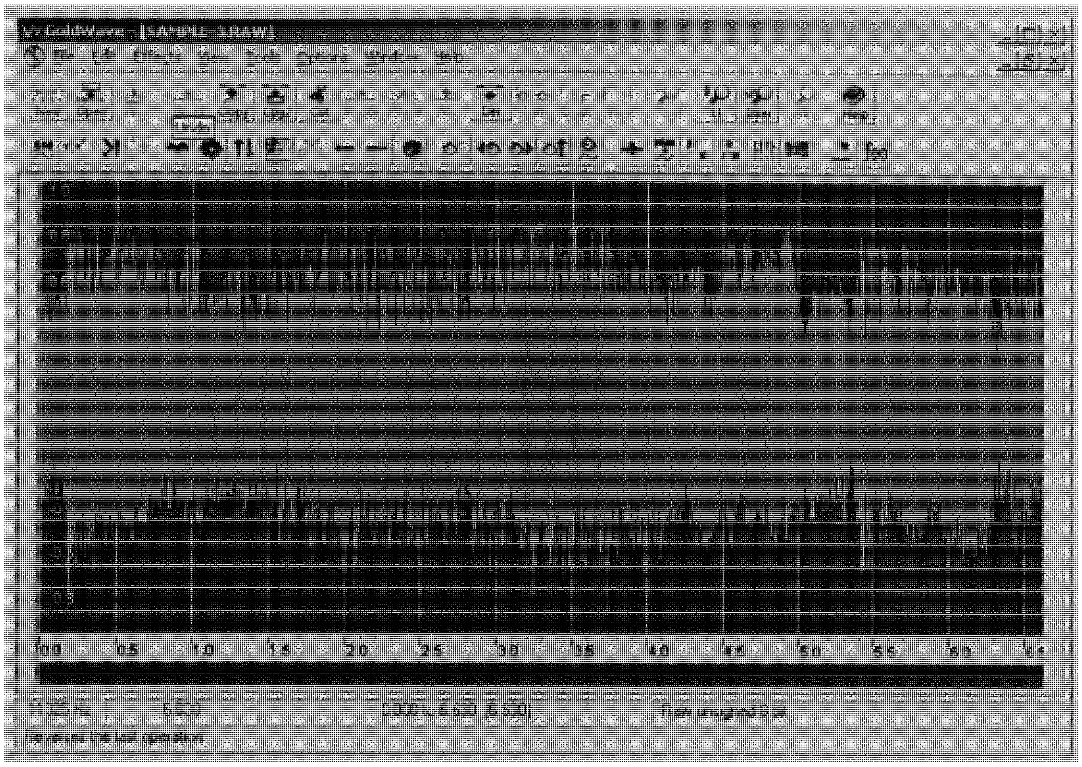

GoldWave - [SAMPLE-3.RAW]
File Edit Effects View Tools Options Window Help
New Open Undo Copy Cpy2 Cut Del 1:1 User Help
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0
11025 Hz | 6.630 | 0.000 to 6.630 (6.630) | Raw unsigned 8 bit
Reverses the last operation

FIGURE 5.3 SAMPLE-3

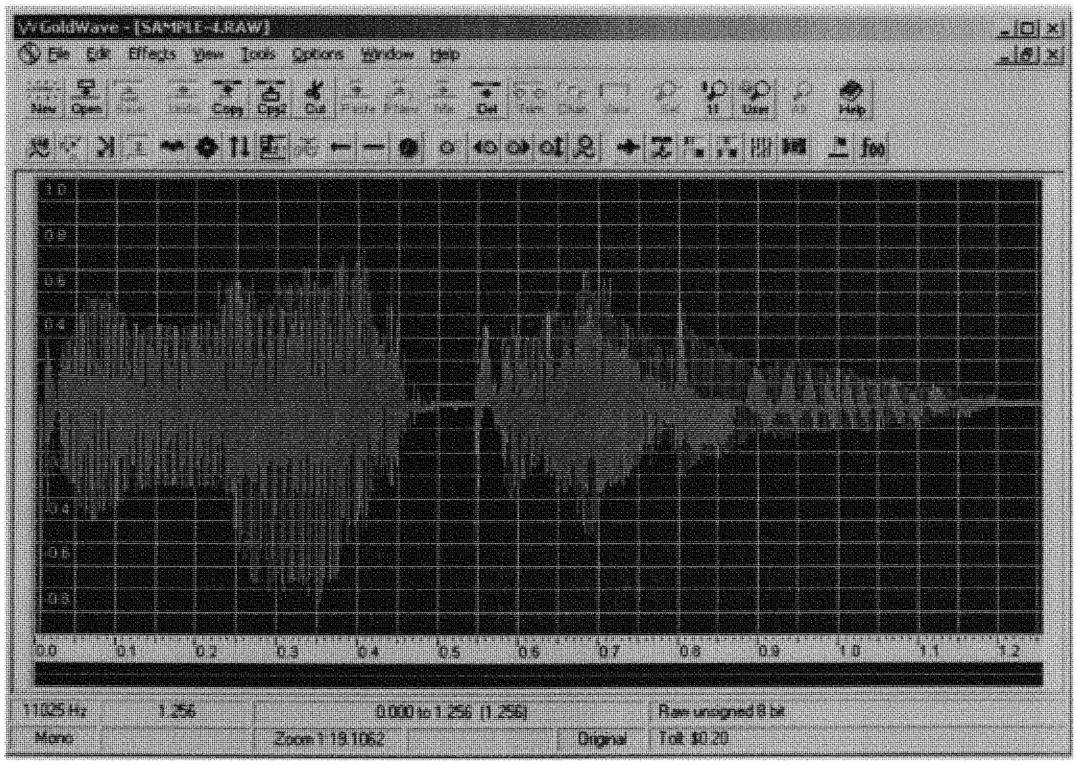
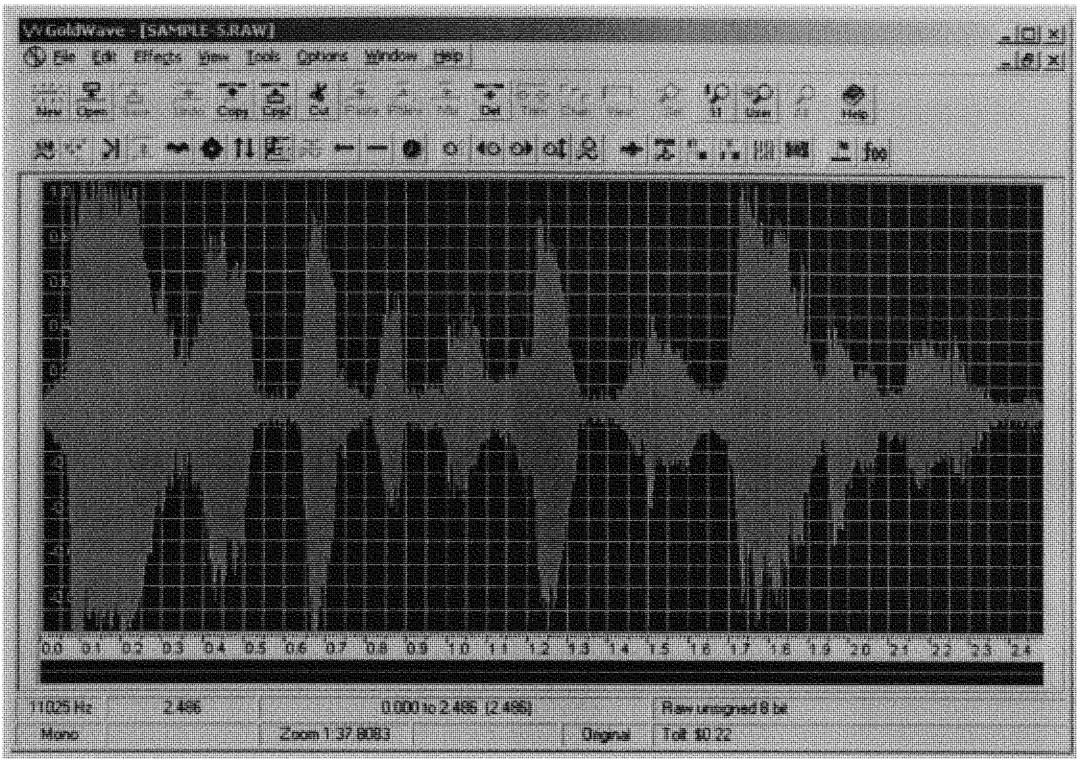
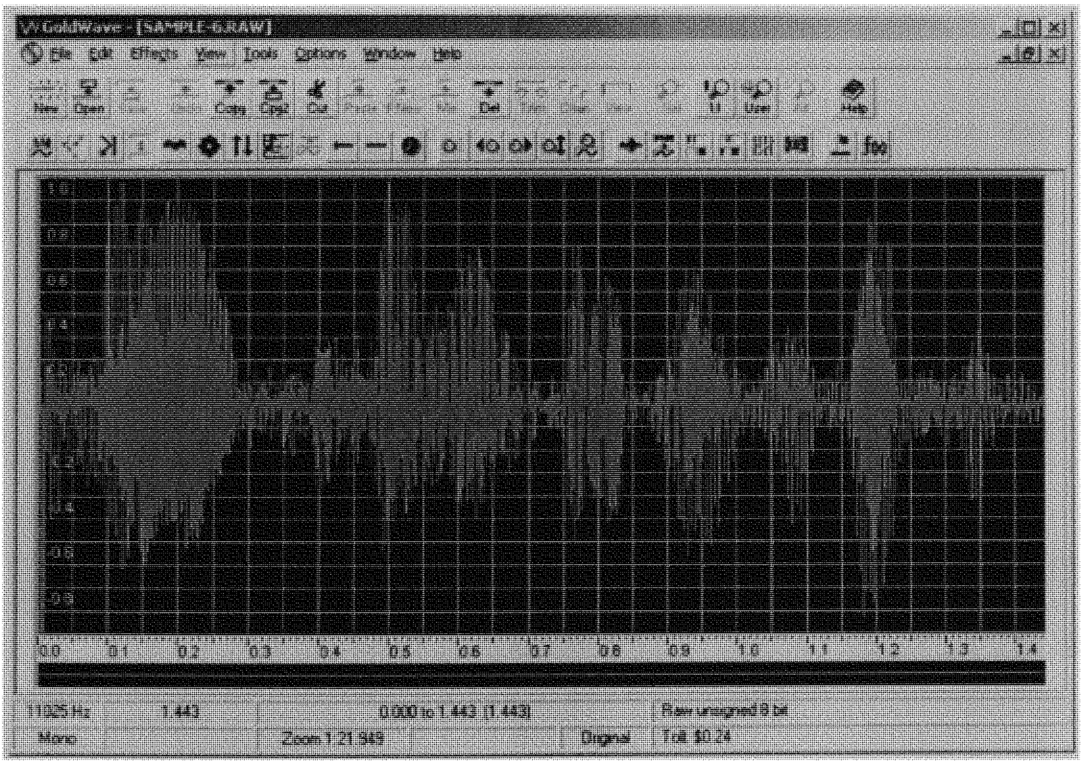

GoldWave - [SAMPLE-4.RAW]
File Edit Effects View Tools Options Window Help
New Open Copy Cpy2 Cut Del 1:1 User Help
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2
11025 Hz | 1.256 | 0.000 to 1.256 (1.256) | Raw unsigned 8 bit
Mono | Zoom 1:18.1062 | Original | Tolt $0.20

FIGURE 5.4 SAMPLE-4


GoldWave - [SAMPLE-5.RAW]
File Edit Effects View Tools Options Window Help
New Open Copy Cop2 Cut Del Help
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4

11025 Hz 2.486 0.000 to 2.486 (2.486) Raw unsigned 8 bit
Mono Zoom 1:37.8083 Original Toll $0.22

FIGURE 5.5 SAMPLE-5


GoldWave - [SAMPLE-6.RAW]
File Edit Effects View Tools Options Window Help
New Open Copy Cop2 Cut Del Help
1.0
0.8
0.6
0.4
0.2
0.4
0.6
0.8
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4

11025 Hz 1.443 0.000 to 1.443 (1.443) Raw unsigned 8 bit
Mono Zoom 1:21.849 Original Toll $0.24

FIGURE 5.6 SAMPLE-6

i. Huffman Coding – This is an entropy-based lossless coding algorithm. It applies variable length codes to a set of data symbols based on the frequency of occurrence of the data symbols. The more frequently a symbol occurs the shorter the code. Refer to file huffm2.cpp in Appendix 1 and [11] for a detailed explanation of the algorithm.

ii. Arithmetic Coding – This is an entropy-based lossless coding scheme. It seeks to generate a unique tag or identifier for a sequence of data symbols. The code used for the tag is of variable length, where the length of the code is dependent on the frequency of occurrence of the sequence. The more frequent a sequence, the shorter the code. Refer to file arith.cpp in Appendix 1 and [11] for a detailed explanation of the algorithm.

iii. Silence Compression – This lossy compression technique accomplishes compression by seeking to replace sections of the audio of relative silence, with a code that indicates the length of these sections. Upon reconstruction, these sections of relative silence will be replaced by sections of absolute silence. Sections of relative silence are determined by a threshold. If any sample magnitude falls below this threshold, it is deemed a "silent" sample. Refer to file silencem.cpp in Appendix 1 and [11] for a detailed explanation of the algorithm.

iv. Compand Compression – the compand compression technique is a lossy compression scheme that achieves compression by reducing the number of bits used to represent a sample magnitude range. It utilizes a logarithmic transfer function to map the range of magnitude values into the reduced number of bits in an efficient manner. For the purpose of this performance analysis, four bits were used to represent each eight-bit sample, giving a compression ratio of 2:1. Refer to file compandM.cpp in Appendix 1 and [11] for a detailed explanation of the algorithm.

v. DCT Compression – This compression technique utilized the Discrete Cosine Transform (DCT) matrix as the means of performing compression. A 16x16 transform matrix was applied to the audio files that were segmented into 16-sample vectors. The resulting vectors of transform coefficients were quantized and encoded using a variable-length encoding scheme. Refer to file oneddct.cpp in Appendix 1 and [11] for a detailed explanation of the DCT algorithm.

vi. DST Compression – This compression technique utilized the Discrete Sine Transform (DST) matrix as the means of performing compression. A 16x16 transform matrix was applied to the audio files that were segmented into 16-sample vectors. The resulting vectors of transform coefficients were quantized and encoded using a variable-length encoding scheme. Refer to file oneddst.cpp in Appendix 1 for a detailed explanation of the DST algorithm.

vii. Multilevel DCT Compression – These compression techniques utilize multiple multiplications by a 16x16 DCT matrix to produce transform coefficients. The multiplications ranged from 2 to 5. The audio files were segmented into 16 sample vectors, and the resulting vectors of transform coefficients were quantized and encoded using a variable-length encoding scheme. Refer to file multidct.cpp in Appendix 1 for a detailed explanation of the multilevel DCT algorithm.

viii. DCT/DST Compression – This compression technique utilizes a 16x16 DCT matrix and 16x16 DST matrix to produce its transform coefficients. Each 16 sample input vector is multiplied by the DCT matrix then the DST matrix to produce a 16 element vector of transform coefficients. The resulting vectors are quantized and encoded using a variable-length encoding scheme. Refer to file dct_dst.cpp in Appendix 1 for a detailed explanation of the DCT/DST algorithm.

The algorithms were analyzed using the following performance measures:

i. Compression Ratio

ii. Rate

iii. Mean-Squared Error

iv. Signal-to-Noise Ratio

5.2 Results

In the performance analysis, two sets of investigations were made. In the first set of investigations, the one-level DCT compression algorithm was compared to the Huffman, Arithmetic, Compand, Silence, and DST compression algorithms. In the second set of investigations the one-level DCT compression algorithm was compared to the multilevel DCT compression algorithms, and the DCT/DST compression algorithm.

The performance analysis results were taken and are displayed in Tables 5.1 – 5.14, as well as, in Figures 5.7. - 5.15. The table values for Tables 5.1 – 5.6 and 5.8 – 5.13 were obtained by compressing and decompressing each audio file sample using the compression algorithms described previously, and using a software application, eval.exe to take the required performance measures. The program eval.cpp, as well as, the auxiliary program perfmeas.cpp is listed in Appendix 2. Tables 5.1 – 5.6 correspond to investigation set #1 and tables 5.8. - 5.13 correspond to investigation set #2. The values of Table 5.7 were obtained by taking the mean of all the values obtained for tables 5.1 – 5.6 for the corresponding performance measure and compression method. The values of Table 5.14 were obtained by taking the mean of all the values obtained for tables 5.8 – 5.13 for the corresponding performance measure and compression method.

Figures 5.7 – 5.11 display graphically all the results obtained in tables 5.1 – 5.6. Figure 5.7 displays the compression ratio performance over the six audio file samples of the Huffman, Arithmetic, Silence, Compand, DST, and DCT algorithms. Figure 5.8 displays the rate performance, and Figure 5.9 displays the mse performance over the same audio file samples for the same algorithms. Since both Huffman and Arithmetic algorithms are lossless, their SNR values are infinite. Therefore, their plots are not shown in Figures 5.10 and 5.11 that display the SNR performance over the six audio file samples for the remaining four algorithms. Figure 5.11 only includes the Compand, DST, and DCT algorithms in order to get a more detailed comparison of these algorithms.

Figures 5.12 – 5.15 display graphically all the results obtained in tables 5.8 – 5.13. Figure 5.12 displays the compression ratio performance over the six audio file samples of the one-level DCT (DCT1), the two-level DCT (DCT2), the three-level DCT (DCT3), the four-level DCT (DCT4), the five-level DCT (DCT5), and the DCT/DST algorithm. Figure 5.13 displays the rate performance, Figure 5.14 displays the mse performance, and

TABLE 5.1 PERFORMANCE MEASURE RESULTS TAKEN FOR FILE SAMPLE-1.RAW USING THE SPECIFIED COMPRESSION ALGORITHMS

PERFORMANCE MEASURES	COMPRESSION METHODS					
	HUFFMAN	ARITHMETIC	SILENCE	COMPAND	DST	DCT
COMPRESSION RATIO	1.33997	1.34491	1.34441	1.99957	3.41174	3.85756
RATE (bits/sample)	5.97026	5.94836	5.95059	4.00087	2.34484	2.07385
MEAN-SQUARED ERROR	0	0	1.34843	15.4832	7.58806	5.30778
SIGNAL-TO-NOISE RATIO	infinity	infinity	12648.7	1101.57	2247.72	3213.37

TABLE 5.2 PERFORMANCE MEASURE RESULTS TAKEN FOR FILE SAMPLE-2.RAW USING THE SPECIFIED COMPRESSION ALGORITHMS

PERFORMANCE MEASURES	COMPRESSION METHODS					
	HUFFMAN	ARITHMETIC	SILENCE	COMPAND	DST	DCT
COMPRESSION RATIO	1.41282	1.42033	1.03066	1.99817	1.83991	1.85266
RATE (bits/sample)	5.66243	5.63251	7.76199	4.00366	4.34804	4.31813
MEAN-SQUARED ERROR	0	0	0.434223	16.5156	24.1842	24.2506
SIGNAL-TO-NOISE RATIO	infinity	infinity	38681.8	1017.01	694.523	692.621

TABLE 5.3 PERFORMANCE MEASURE RESULTS TAKEN FOR FILE SAMPLE-3.RAW USING THE SPECIFIED COMPRESSION ALGORITHMS

PERFORMANCE MEASURES	COMPRESSION METHODS					
	HUFFMAN	ARITHMETIC	SILENCE	COMPAND	DST	DCT
COMPRESSION RATIO	1.19929	1.20576	1.00026	1.9997	1.74692	1.77122
RATE (bits/sample)	6.67059	6.6348	7.99792	4.0006	4.5795	4.51667
MEAN-SQUARED ERROR	0	0	0.00889302	16.0757	19.9405	19.766
SIGNAL-TO-NOISE RATIO	infinity	infinity	1982400	1096.65	884.104	891.911

TABLE 5.4 PERFORMANCE MEASURE RESULTS TAKEN FOR FILE SAMPLE-4.RAW USING THE SPECIFIED COMPRESSION ALGORITHMS

PERFORMANCE MEASURES	COMPRESSION METHODS					
	HUFFMAN	ARITHMETIC	SILENCE	COMPAND	DST	DCT
COMPRESSION RATIO	1.36151	1.36959	1.26364	1.99856	2.71182	3.12404
RATE (bits/sample)	5.87583	5.84118	6.33093	4.00289	2.95004	2.56079
MEAN-SQUARED ERROR	0	0	1.18712	15.4665	11.2064	9.38673
SIGNAL-TO-NOISE RATIO	infinity	infinity	14468.8	1110.53	1532.64	1829.81

TABLE 5.5 PERFORMANCE MEASURE RESULTS TAKEN FOR FILE SAMPLE-5.RAW USING THE SPECIFIED COMPRESSION ALGORITHMS

PERFORMANCE MEASURES	COMPRESSION METHODS					
	HUFFMAN	ARITHMETIC	SILENCE	COMPAND	DST	DCT
COMPRESSION RATIO	1.45764	1.46379	1.19882	1.9992	2.47772	2.52032
RATE (bits/sample)	5.48831	5.46525	6.67323	4.00161	3.22878	3.1742
MEAN-SQUARED ERROR	0	0	1.60636	16.9614	15.2916	14.8024
SIGNAL-TO-NOISE RATIO	infinity	infinity	10499.8	994.403	1102.99	1139.44

TABLE 5.6 PERFORMANCE MEASURE RESULTS TAKEN FOR FILE SAMPLE-6.RAW USING THE SPECIFIED COMPRESSION ALGORITHMS

PERFORMANCE MEASURES	COMPRESSION METHODS					
	HUFFMAN	ARITHMETIC	SILENCE	COMPAND	DST	DCT
COMPRESSION RATIO	1.21187	1.21678	1.093	1.99862	2.52307	2.73937
RATE (bits/sample)	6.6014	6.57475	7.3193	4.00277	3.17074	2.92038
MEAN-SQUARED ERROR	0	0	0.575567	14.3754	12.3538	11.1662
SIGNAL-TO-NOISE RATIO	infinity	infinity	29236	1170.56	1362.11	1506.99

TABLE 5.7 PERFORMANCE MEASURE RESULTS TAKING THE MEAN OF THE RESULTS OF ALL AUDIO FILES USING THE SPECIFIED COMPRESSION ALGORITHMS

PERFORMANCE MEASURES	COMPRESSION METHODS					
	HUFFMAN	ARITHMETIC	SILENCE	COMPAND	DST	DCT
COMPRESSION RATIO	1.573456667	1.580825	1.354935	2.33217	2.864816667	3.064248333
RATE (bits/sample)	6.959521667	6.927016667	8.117865	4.669001667	3.97512	3.789703333
MEAN-SQUARED ERROR	0	0	1.127825503	18.63986667	17.64269333	16.58035167
SIGNAL-TO-NOISE RATIO	infinity	infinity	349739.15	1247.521	1487.846167	1735.597

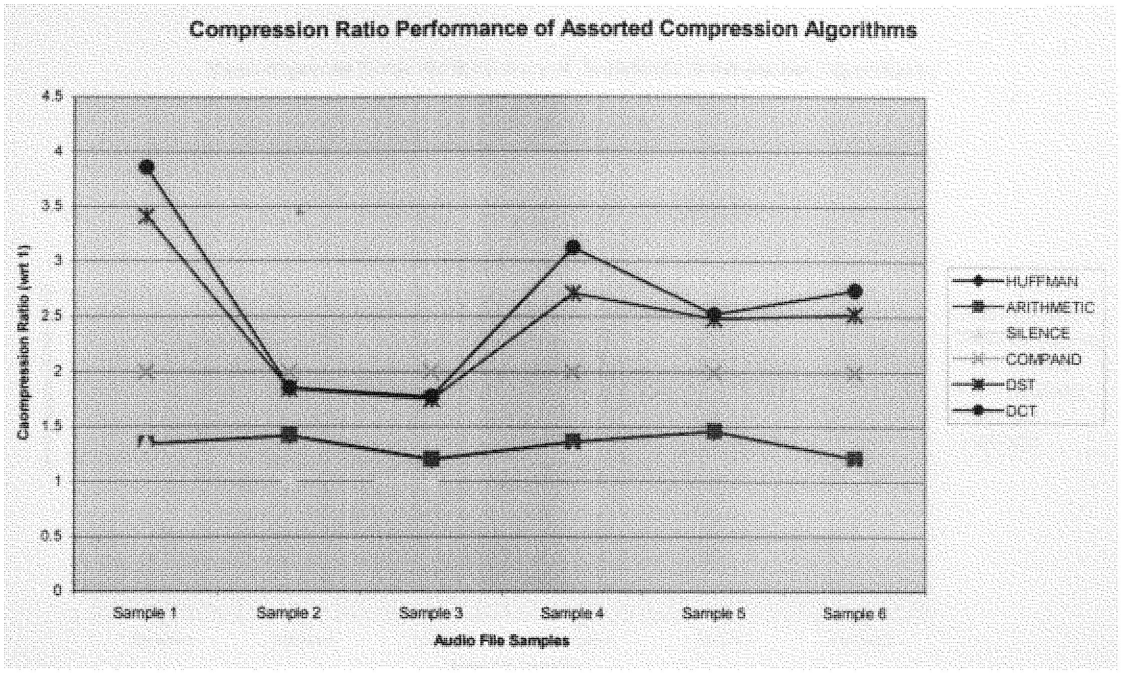


FIGURE 5.7 Compression Ratio Performance of Specified Compression Methods

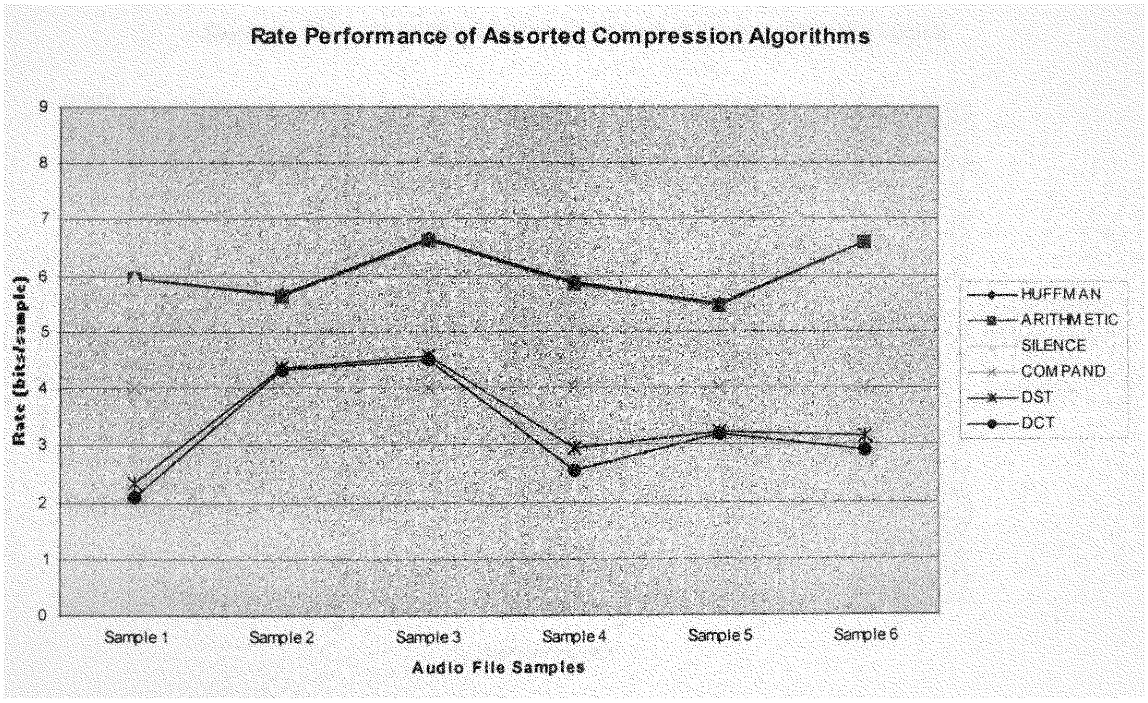


FIGURE 5.8 Rate Performance of the Specified Compression Methods

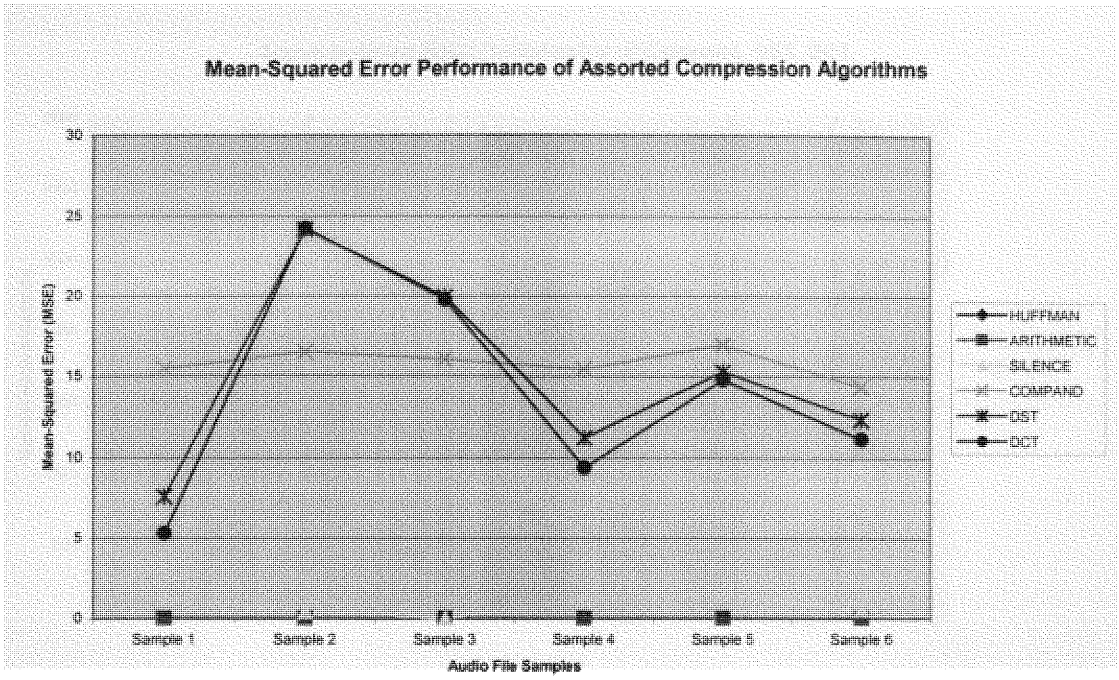


FIGURE 5.9 MSE Performance for the Specified Compression Methods

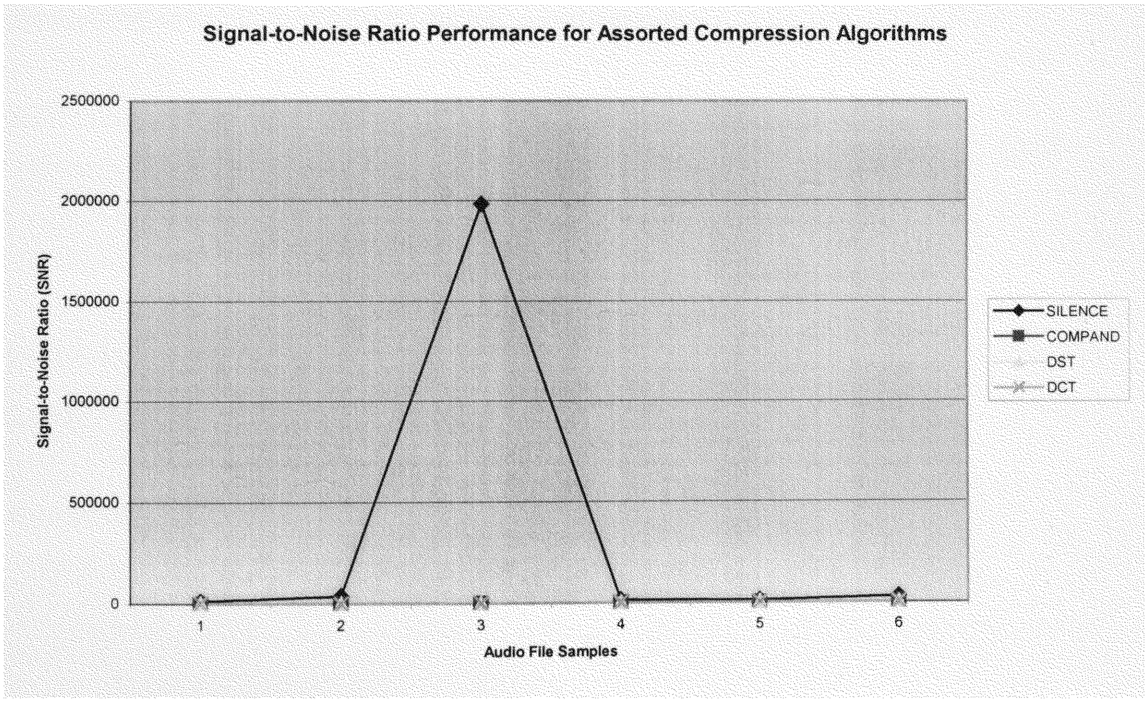


FIGURE 5.10 SNR Performance of the Specified Compression Methods

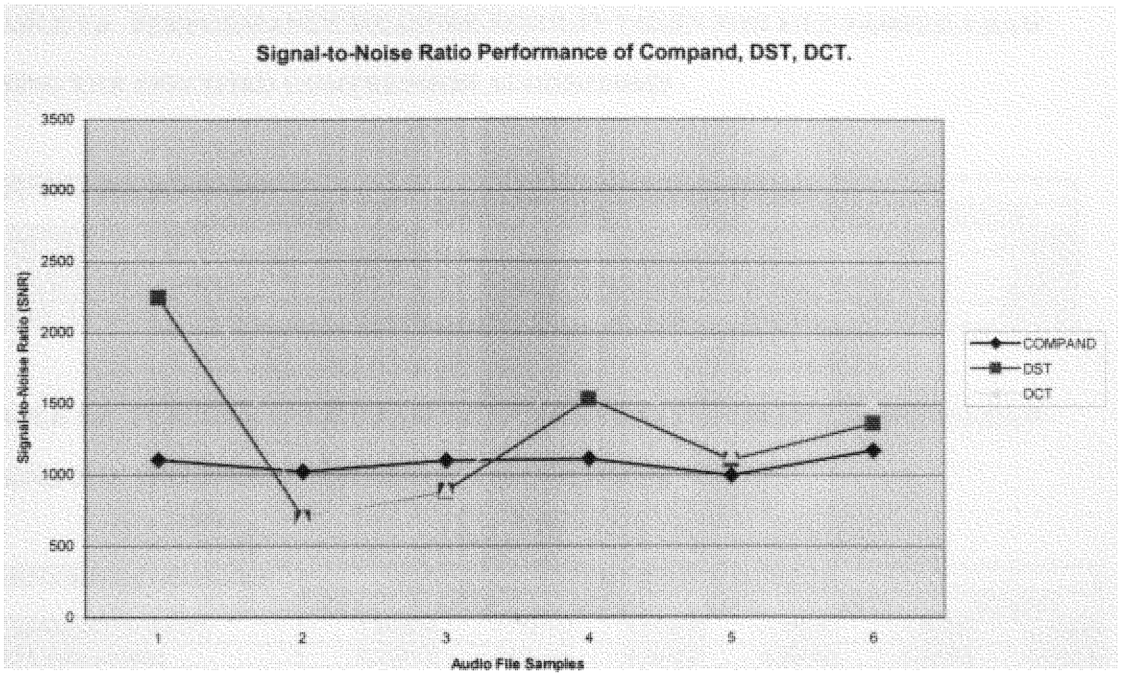


FIGURE 5.11 SNR Performance of the Compand, DST, and DCT Compression Methods (Magnified plot)

TABLE 5.8 PERFORMANCE MEASURE RESULTS TAKEN FOR FILE SAMPLE-1.RAW USING THE SPECIFIED COMPRESSION ALGORITHMS

PERFORMANCE MEASURES	COMPRESSION METHODS					
	DCT1	DCT2	DCT3	DCT4	DCT5	DCT/DST
COMPRESSION RATIO	3.85756	2.57594	3.56155	2.55649	3.3514	2.70407
RATE (bits/sample)	2.07385	3.10566	2.4621	3.12929	2.38707	2.9585
MEAN-SQUARED ERROR	5.30778	18.8946	6.14197	17.3596	9.71887	20.0778
SIGNAL-TO-NOISE RATIO	3213.37	902.685	2776.94	982.502	1754.92	849.487

TABLE 5.9 PERFORMANCE MEASURE RESULTS TAKEN FOR FILE SAMPLE-2.RAW USING THE SPECIFIED COMPRESSION ALGORITHMS

PERFORMANCE MEASURES	COMPRESSION METHODS					
	DCT1	DCT2	DCT3	DCT4	DCT5	DCT/DST
COMPRESSION RATIO	1.85266	1.77635	1.81493	1.79597	1.8027	1.76826
RATE (bits/sample)	4.31813	4.50362	4.40788	4.45442	4.4378	4.52423
MEAN-SQUARED ERROR	24.2506	26.5782	25.319	27.0481	25.2894	26.2636
SIGNAL-TO-NOISE RATIO	692.621	631.965	665.684	620.986	664.173	639.535

TABLE 5.10 PERFORMANCE MEASURE RESULTS TAKEN FOR FILE SAMPLE-3.RAW USING THE SPECIFIED COMPRESSION ALGORITHMS

PERFORMANCE MEASURES	COMPRESSION METHODS					
	DCT1	DCT2	DCT3	DCT4	DCT5	DCT/DST
COMPRESSION RATIO	1.77122	1.43352	1.7191	1.43888	1.66639	1.443
RATE (bits/sample)	4.51667	5.58066	4.6536	5.55986	4.80081	5.54399
MEAN-SQUARED ERROR	19.766	29.5741	21.1091	29.0772	22.1382	30.1246
SIGNAL-TO-NOISE RATIO	891.911	596.113	835.161	606.301	796.339	585.219

TABLE 5.11 PERFORMANCE MEASURE RESULTS TAKEN FOR FILE SAMPLE-4.RAW USING THE SPECIFIED COMPRESSION ALGORITHMS

PERFORMANCE MEASURES	COMPRESSION METHODS					
	DCT1	DCT2	DCT3	DCT4	DCT5	DCT/DST
COMPRESSION RATIO	3.12404	2.05428	2.90094	2.05611	2.75497	2.13075
RATE (bits/sample)	2.56079	3.89431	2.75772	3.89085	2.90384	3.75455
MEAN-SQUARED ERROR	9.38673	22.886	9.88074	21.4742	10.5998	23.7412
SIGNAL-TO-NOISE RATIO	1829.81	750.5	1738.33	799.84	1620.41	723.466

TABLE 5.12 PERFORMANCE MEASURE RESULTS TAKEN FOR FILE SAMPLE-5.RAW USING THE SPECIFIED COMPRESSION ALGORITHMS

PERFORMANCE MEASURES	COMPRESSION METHODS					
	DCT1	DCT2	DCT3	DCT4	DCT5	DCT/DST
COMPRESSION RATIO	2.52032	1.97614	2.43221	1.97557	2.35187	1.9863
RATE (bits/sample)	3.1742	4.0483	3.28919	4.04947	3.40155	4.02758
MEAN-SQUARED ERROR	14.8024	23.5312	15.2325	23.2627	15.8139	23.7062
SIGNAL-TO-NOISE RATIO	1139.44	716.771	1107.27	725.042	1066.56	711.478

TABLE 5.13 PERFORMANCE MEASURE RESULTS TAKEN FOR FILE SAMPLE-6.RAW USING THE SPECIFIED COMPRESSION ALGORITHMS

PERFORMANCE MEASURES	COMPRESSION METHODS					
	DCT1	DCT2	DCT3	DCT4	DCT5	DCT/DST
COMPRESSION RATIO	2.73937	2.00795	2.59465	2.00517	2.46293	2.03101
RATE (bits/sample)	2.92038	3.98416	3.08327	3.98969	3.24816	3.93892
MEAN-SQUARED ERROR	11.1662	23.968	11.8047	22.9675	20.3434	24.798
SIGNAL-TO-NOISE RATIO	1506.99	702.072	1425.47	732.656	827.16	678.575

TABLE 5.14 PERFORMANCE MEASURE RESULTS TAKING THE MEAN OF THE RESULTS OF ALL AUDIO FILES USING THE SPECIFIED COMPRESSION ALGORITHMS

PERFORMANCE MEASURES	COMPRESSION METHODS					
	DCT1	DCT2	DCT3	DCT4	DCT5	DCT/DST
COMPRESSION RATIO	2.644195	1.970696667	2.503896667	1.971365	2.398376667	2.010565
RATE (bits/sample)	3.26067	4.186118333	3.442293333	4.17893	3.529871667	4.124628333
MEAN-SQUARED ERROR	14.113285	24.23868333	14.91466833	23.53155	17.31726167	24.78523333
SIGNAL-TO-NOISE RATIO	1545.690333	716.6843333	1424.809167	744.5545	1121.593667	697.96

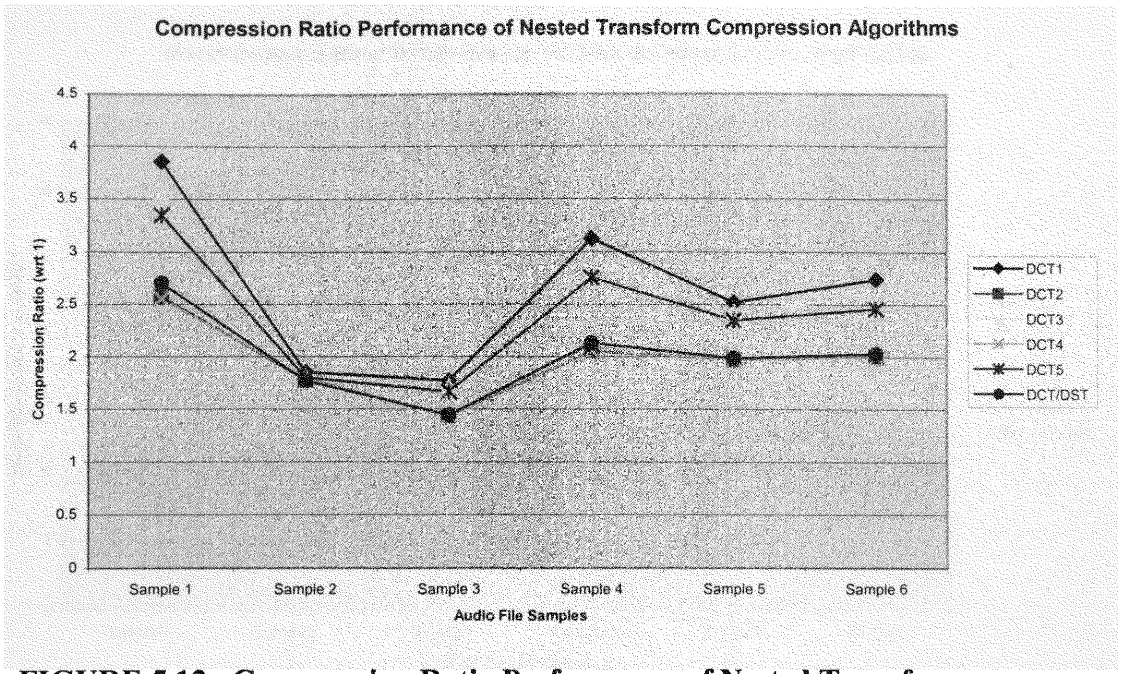


FIGURE 5.12 Compression Ratio Performance of Nested Transform Compression Methods

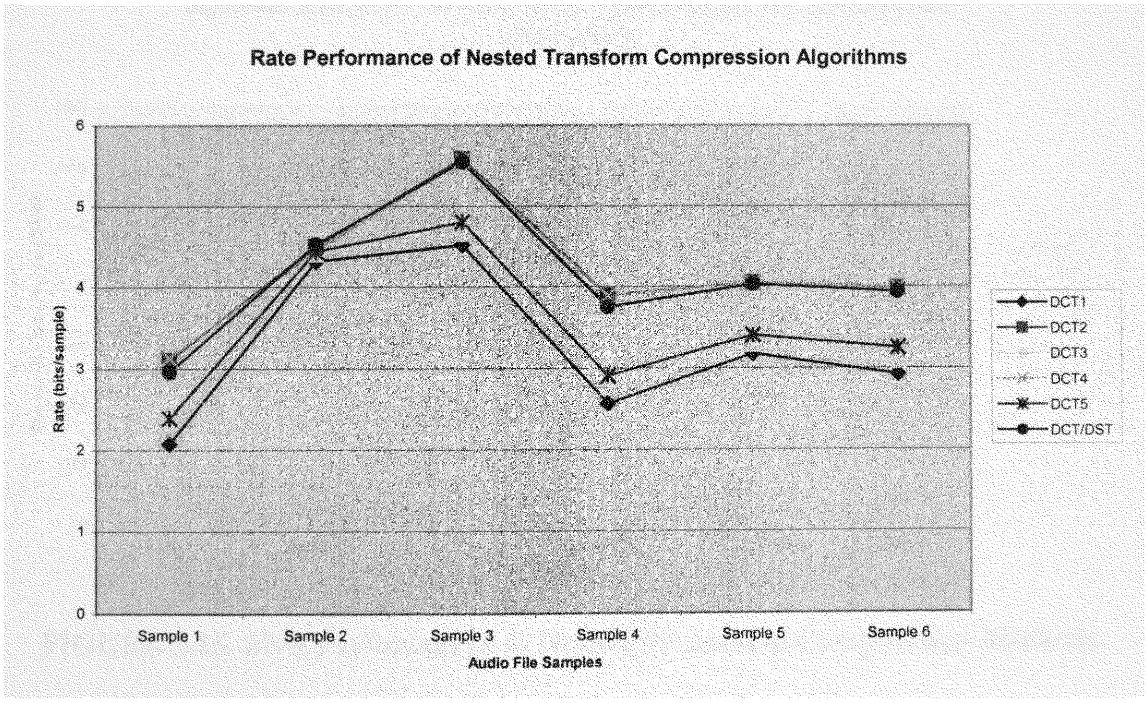


FIGURE 5.13 Rate Performance of Nested Transform Compression Methods

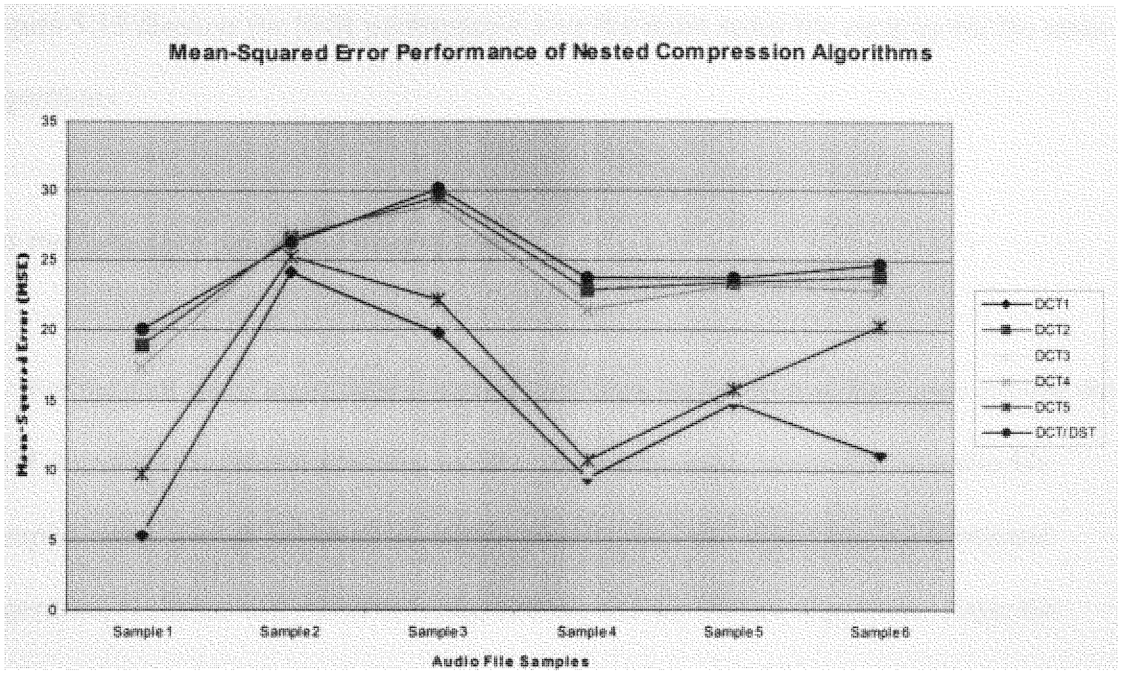


FIGURE 5.14 MSE Performance of Nested Transform Compression Methods

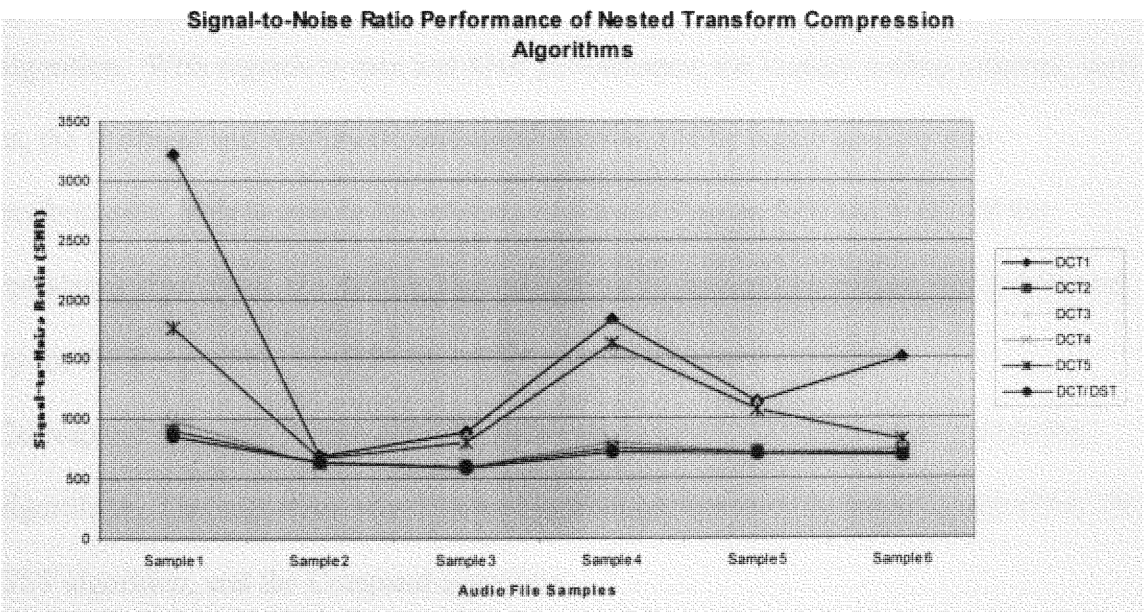


FIGURE 5.15 SNR Performance of Nested Transform Compression Methods

Figure 5.15 displays the SNR performance over the same audio file samples for the same algorithms.

5.3 Performance Analysis Observations

For the first set of algorithms it can be seen from the tables and the Figure 5.8 that the DCT algorithm in general provides larger compression ratio values for the audio file samples used. This is followed in performance by the DST algorithm, the Compand algorithm, Huffman and Arithmetic algorithms, and the Silence algorithm. From the tables and Figure 5.9 we observe that the DCT algorithm produces the lowest rate values for the audio file samples used. This is followed in performance by the DST algorithm, the Compand algorithm, the Huffman and Arithmetic algorithms, and the Silence algorithm. With regards to mse and SNR performance, the lossless coding schemes, such as, the Huffman and Arithmetic coding schemes would by nature give the best performance. This is seen in the relevant tables and Figures 5.10 and 5.11. Of the remaining algorithms, the Silence algorithm gave the lowest mse values, this is followed by the DCT algorithm, the DST algorithm and the Compand algorithm. Correspondingly, the Silence algorithm gave the largest SNR values of the remaining algorithms. This algorithm was followed in SNR performance by the DCT algorithm, the DST algorithm, and the Compand algorithm.

For the second set of algorithms it can be seen from the tables and the Figure 5.12 that the one-level DCT algorithm provides larger compression ratio values for the audio file samples used. This is followed in performance by the DCT3 algorithm, the DCT5 algorithm, the DCT/DST algorithm, the DCT2 algorithm, and the DCT4 algorithm. From the tables and Figure 5.13 we observe that the DCT1 algorithm produces the lowest rate values for the audio file samples used. This is followed in performance by the DCT3 algorithm, the DCT5 algorithm, the DCT/DST algorithm, the DCT2 algorithm, and the DCT4 algorithm. Figure 5.14 reveals that, in terms of mse performance, the DCT1 algorithm gave the lowest mse values. The DCT1 algorithm is followed by the DCT3 algorithm, the DCT5 algorithm, the DCT4 algorithm, the DCT2 algorithm, and the DCT/DST algorithm. Correspondingly, Figure 5.15 reveals that, in terms of SNR performance, the DCT1 algorithm gave the highest SNR values. The DCT1 algorithm is followed by the DCT3 algorithm, the DCT5 algorithm, the DCT4 algorithm, the DCT2 algorithm, and the DCT/DST algorithm.

5.4 Assessment of Performance Analysis Results

In general, we desire a compression algorithm that provides a high compression ratio, low rate, low mse, and high SNR. Of the compression measures used, the compression ratio measure should be weighted with greater value, because poorer distortion performance does not always translate into poorer intelligibility. Based on these criteria the DCT algorithm is the most effective, followed by the DCT3 algorithm, the DCT5 algorithm, and then the DST algorithm.

Chapter 6 Application Implementation

6.1 Application Overview

In order to demonstrate the applicability of the transform coding algorithms for the transmission of audio data across networks one of the algorithms was selected to compress/expand audio data used in a voice-over-IP application. Based on the compression performance analysis results, the one-level DCT algorithm was chosen as the compression algorithm to be used in the voice-over-IP application. The reasons for this choice are as follows:

i. The DCT1 algorithm provides the greatest amount of compression when compared to the other algorithms.

ii. It provides adequate distortion performance as determined by listening to the outputs of the DCT1 compression/expansion process.

The voice-over-IP application allows a user to communicate by voice to another user on a remote computer. The mode of communication is half-duplex. The structure of this VOIP application is displayed in Figure 6.1. As can be seen from the figure, the application possesses both hardware and software components. The hardware components consist of a microphone for audio input, speakers for audio output, and a sound card that acts as a transducer, converting the audio signals into 8 bit, mono, unsigned PCM samples and vice versa.

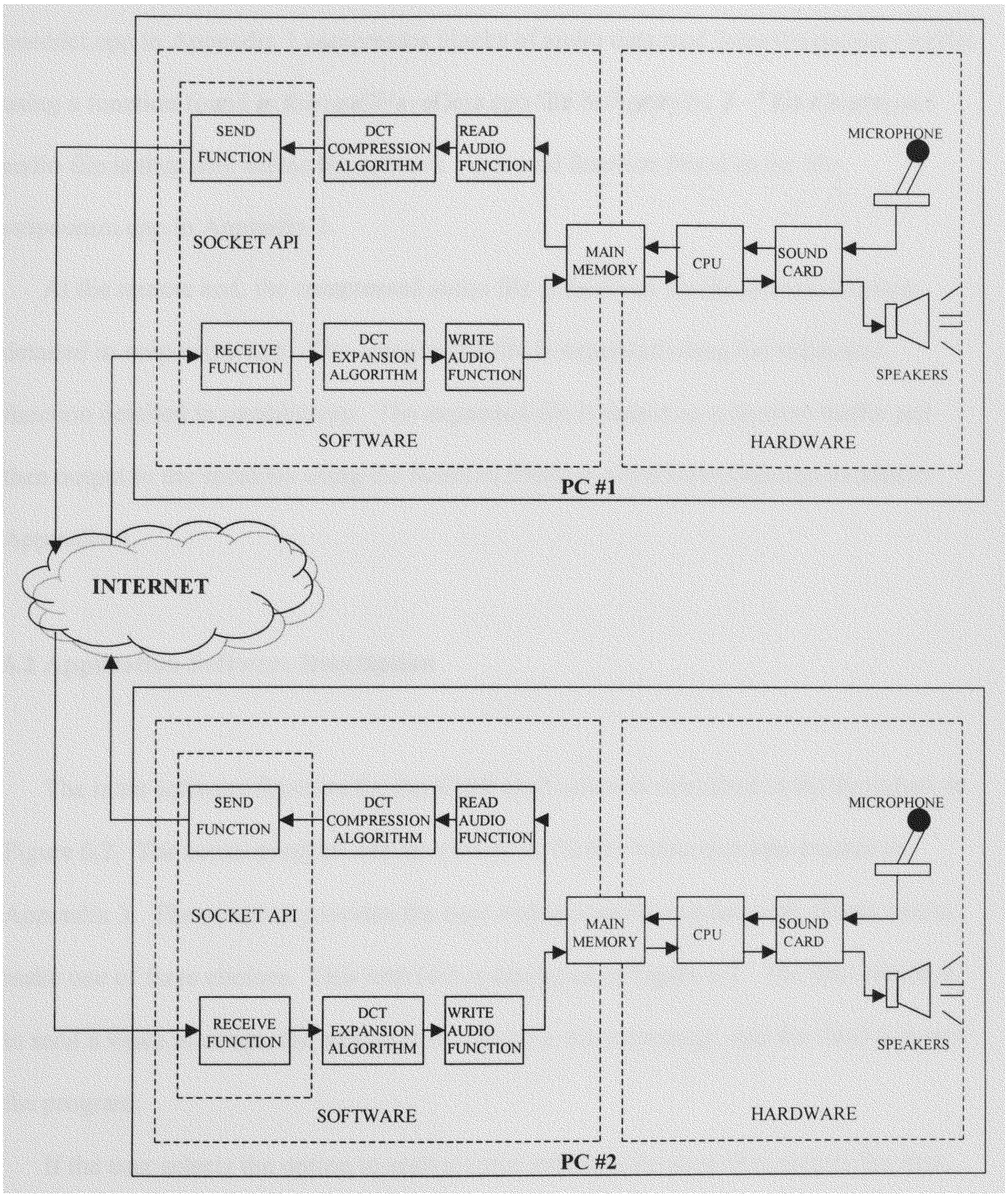


FIGURE 6.1 APPLICATION STRUCTURAL DIAGRAM

Incoming samples are stored in a memory buffer. The compression function found in oneddct.cpp in Appendix 1 compresses blocks of audio data read from the memory buffer using a function found in the readWaveData.cpp file in Appendix 3. This compressed audio file is then sent across the network via a send function found in the file voipcomm.cpp in Appendix 3.

At the remote end, the compressed audio file is received via the receive function detailed in voipcomm.cpp. The compressed file is expanded using the expansion function detailed in oneddct.cpp. The expanded file is output to a memory buffer and then output to the speakers using the function found in writeWaveData.cpp located in Appendix 3.

6.2 Application Software Description

The main software function for the VOIP application is described in the flowchart in Figure 6.2. The actual program can be viewed in the file voipcomm.cpp located in Appendix 3. The program provides the user with a console interface that allows him to make one of three choices. This interface is displayed in Figure 6.3. The first choice is to send a voice message, the second is to receive a voice message, and the third is to exit the program.

If the user selects the option to send a voice message, he must first specify the host name of the computer he desires to communicate with. Provided the user of the remote computer has his application prepared to receive voice messages; a connection is made between the two applications using the socket API functions. The local user is then

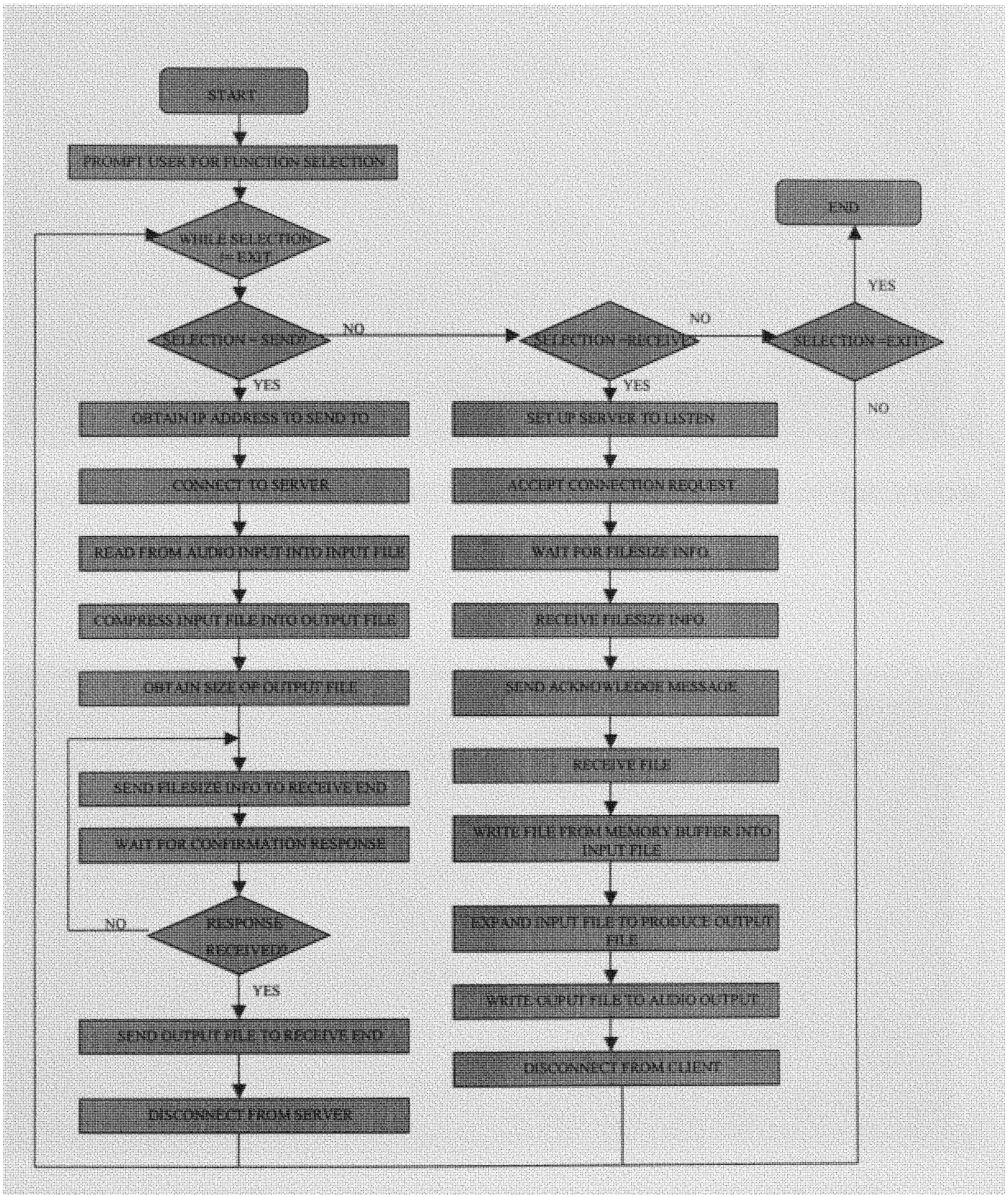
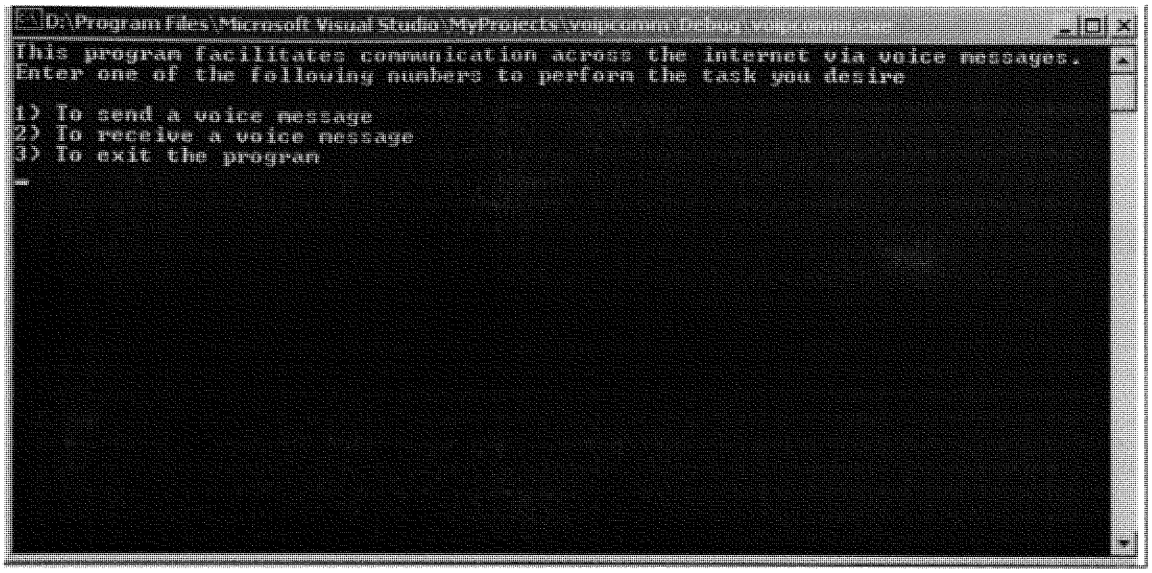

START
PROMPT USER FOR FUNCTION SELECTION
WHILE SELECTION != EXIT
SELECTION = SEND?
NO
YES
OBTAIN IP ADDRESS TO SEND TO
CONNECT TO SERVER
READ FROM AUDIO INPUT INTO INPUT FILE
COMPRESS INPUT FILE INTO OUTPUT FILE
OBTAIN SIZE OF OUTPUT FILE
SEND FILESIZE INFO TO RECEIVE END
WAIT FOR CONFIRMATION RESPONSE
RESPONSE RECEIVED?
NO
YES
SEND OUTPUT FILE TO RECEIVE END
DISCONNECT FROM SERVER
SELECTION = RECEIVE?
NO
YES
SET UP SERVER TO LISTEN
ACCEPT CONNECTION REQUEST
WAIT FOR FILESIZE INFO
RECEIVE FILESIZE INFO
SEND ACKNOWLEDGE MESSAGE
RECEIVE FILE
WRITE FILE FROM MEMORY BUFFER INTO INPUT FILE
EXPAND INPUT FILE TO PRODUCE OUTPUT FILE
WRITE OUPUT FILE TO AUDIO OUTPUT
DISCONNECT FROM CLIENT
SELECTION = EXIT?
YES
NO
END

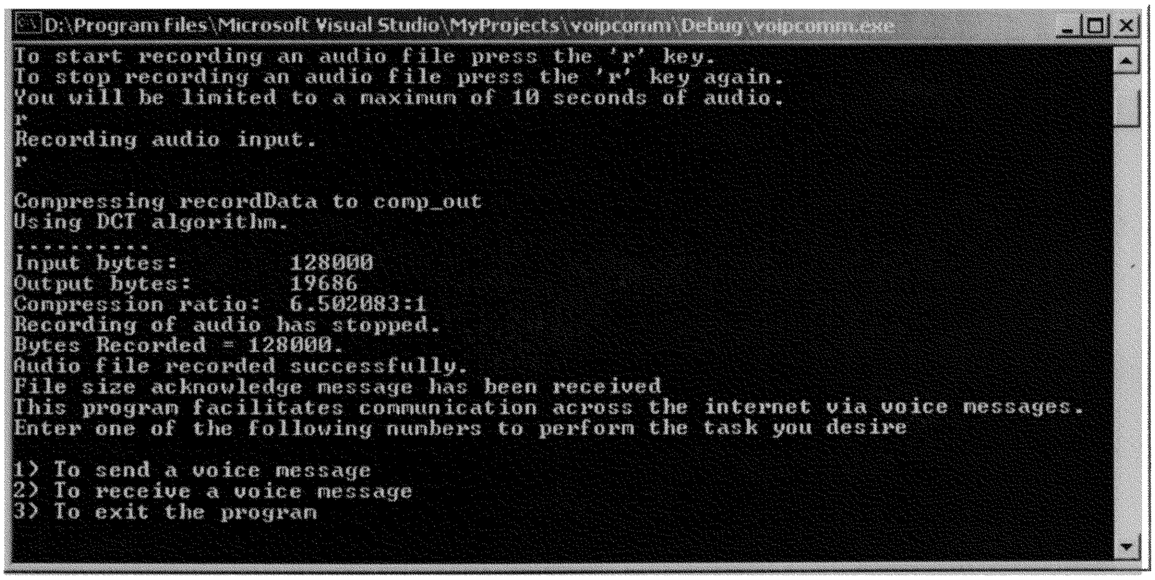
FIGURE 6.2 VOICE-OVER-IP APPLICATION FLOWCHART


D:\Program Files\Microsoft Visual Studio\MyProjects\voipcomm\Debug\voipcomm.exe

```
This program facilitates communication across the internet via voice messages.
Enter one of the following numbers to perform the task you desire

1) To send a voice message
2) To receive a voice message
3) To exit the program
_
```

FIGURE 6.3 CONSOLE MAIN MENU INTERFACE FOR VOIP APPLICATION


D:\Program Files\Microsoft Visual Studio\MyProjects\voipcomm\Debug\voipcomm.exe

```
To start recording an audio file press the 'r' key.
To stop recording an audio file press the 'r' key again.
You will be limited to a maximum of 10 seconds of audio.
r
Recording audio input.
r

Compressing recordData to comp_out
Using DCT algorithm.
..........
Input bytes:        128000
Output bytes:       19686
Compression ratio:  6.502083:1
Recording of audio has stopped.
Bytes Recorded = 128000.
Audio file recorded successfully.
File size acknowledge message has been received
This program facilitates communication across the internet via voice messages.
Enter one of the following numbers to perform the task you desire

1) To send a voice message
2) To receive a voice message
3) To exit the program
```

FIGURE 6.4 CONSOLE AUDIO RECORDING INTERFACE FOR VOIP APPLICATION

given the opportunity to record up to ten seconds of audio input from the microphone (refer to Figure 6.4). Once he has recorded his message, the message is compressed, and the size of the compressed audio file is obtained and is sent to the remote end. Upon receiving an acknowledgement message from the remote end, the compressed message is sent and the application returns to the main menu of options.

If the user selects the option to receive a voice message, the application will enter a waiting mode until it is contacted by a remote application. Once a connection request has been received and accepted from a remote application, the local application receives the file size information of the incoming file and sends an acknowledgement message. The application then waits again to receive the incoming compressed audio message. Once received, the compressed message is written from the memory buffer to a file. This file is expanded and then output to the speakers. At this point the application returns to the main menu of options.

The compression function is described in more detail in the flowchart displayed in Figure 6.5. The compression function begins by opening the input and output files used in the compression process. The quantization vector and the DCT matrix are then initialized. The audio samples are then read in to fill a vector. Once a vector is filled, a forward DCT transform is performed to produce a corresponding vector of coefficients. These coefficients are then quantized and encoded before they are stored in the compression output file. This process is continued until we reach the end of the input file. At this point the transform is applied to any remaining samples, if there are any, and these are quantized, encoded and stored in the output file, along with an end-of-file code. The input and output files are then closed and the function returns.

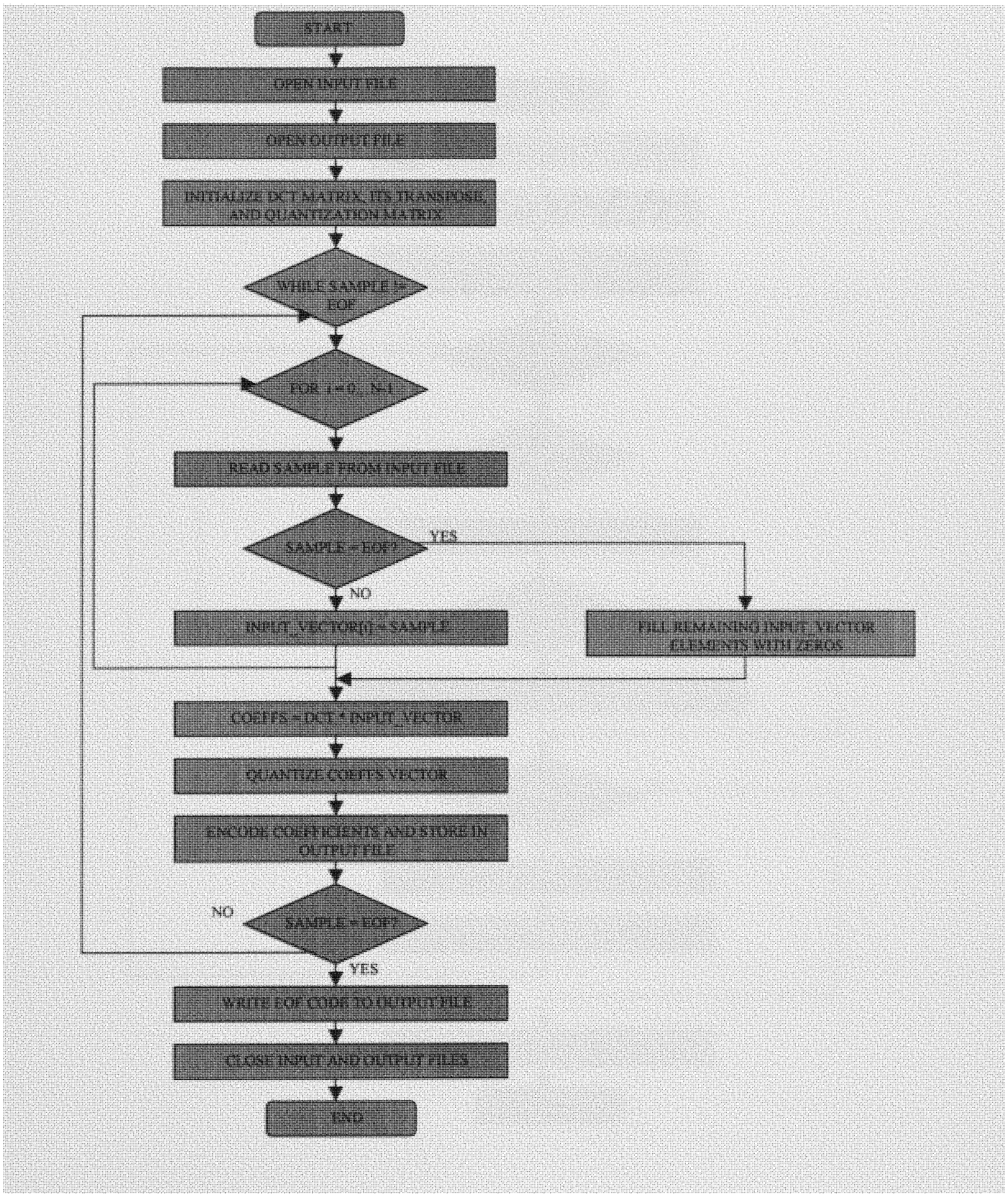

START
OPEN INPUT FILE
OPEN OUTPUT FILE
INITIALIZE DCT MATRIX, ITS TRANSPOSE, AND QUANTIZATION MATRIX
WHILE SAMPLE != EOF
FOR I = 0.. N-1
READ SAMPLE FROM INPUT FILE
SAMPLE = EOF?
YES
NO
INPUT_VECTOR[I] = SAMPLE
FILL REMAINING INPUT_VECTOR ELEMENTS WITH ZEROS
COEFFS = DCT * INPUT_VECTOR
QUANTIZE COEFFS VECTOR
ENCODE COEFFICIENTS AND STORE IN OUTPUT FILE
NO
SAMPLE = EOF?
YES
WRITE EOF CODE TO OUTPUT FILE
CLOSE INPUT AND OUTPUT FILES
END

FIGURE 6.5 DCT COMPRESSION FLOWCHART

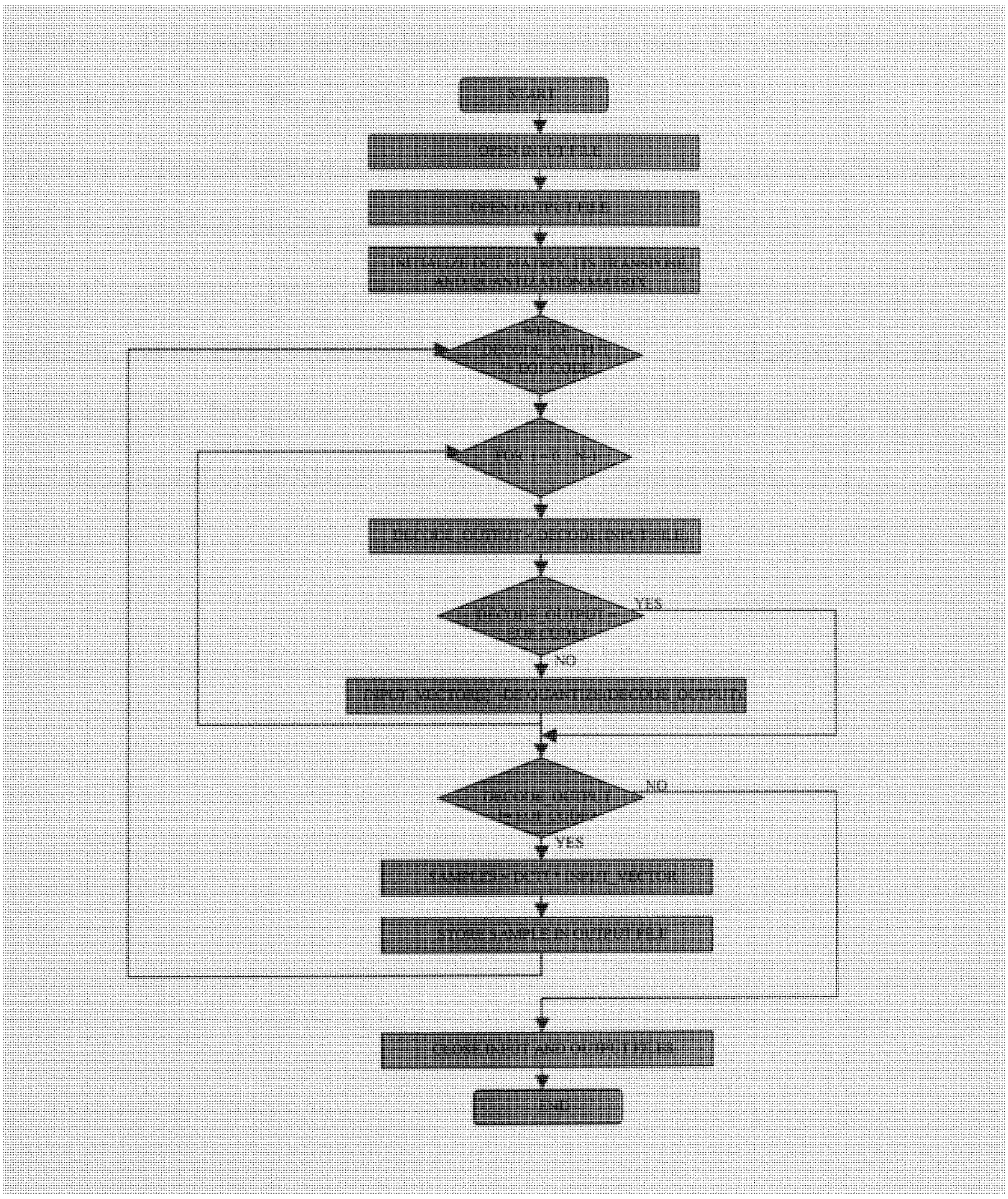

START
OPEN INPUT FILE
OPEN OUTPUT FILE
INITIALIZE DCT MATRIX, ITS TRANSPOSE, AND QUANTIZATION MATRIX
WHILE DECODE_OUTPUT != EOF CODE
FOR i = 0...N-1
DECODE_OUTPUT = DECODE(INPUT FILE)
DECODE_OUTPUT = EOF CODE?
YES
NO
INPUT_VECTOR[i] = DE-QUANTIZE(DECODE_OUTPUT)
DECODE_OUTPUT != EOF CODE?
NO
YES
SAMPLES = DCT * INPUT_VECTOR
STORE SAMPLE IN OUTPUT FILE
CLOSE INPUT AND OUTPUT FILES
END

FIGURE 6.6 DCT EXPANSION FLOWCHART

The expansion function is described in more detail in the flowchart displayed in Figure 6.6. The expansion function begins by opening the input and output files used in the expansion process. The quantization and the DCT transpose matrix are then initialized. The coefficients are then obtained from the input file by decoding the input file. The input file is decoded until a vector can be filled with coefficients. The resultant vector of coefficients is then dequantized. This vector then undergoes an inverse DCT transform, and the output vector consisting of reconstructed audio samples is then stored in the output file. This process is continued until we reach the end-of-file code. At this point the input and output files are then closed and the function returns.

Chapter 7 Conclusions and Recommendations

7.1 General Conclusions

By comparing the multilevel Discrete Cosine transforms to the one-level DCT, it was observed that for this data set of audio samples, the one-level DCT is more effective. It was also observed that the DCT3 and DCT5 algorithms produce a better compression performance than the one-level DST for this data set. This may suggest that these transforms may provide an alternative method of optimal compression for an audio signal with different statistical characteristics from those used. This in turn would lead to a possible compression application in which these transforms, along with, the one-level DCT and DST, among others, can form a set of bases, that may be applied to short time periods of the audio signal. The actual transform that is chosen is selected based on which transform gives the best compression/distortion performance for that particular section of the audio signal. Such an application would be potentially be more effective in compression performance than a fixed DCT algorithm due to the non-stationary nature of audio signals. One of the disadvantages of the multilevel DCT algorithms are that by their very nature they will involve more coding time when compared to the one-level transforms. This makes their implementation in real-time applications prohibitive.

With regards to the voice-over-IP application developed. The one-level DCT algorithm has shown itself to be a suitable coding algorithm for VOIP applications. It produces an average compression ratio of 2.6, which is equivalent to a bit rate of 24.6 kbps for the standard 64 kbps PCM input. This is a significant reduction in the bandwidth utilized. The coding time for the algorithm was consistently less than 1 second for <= 10 seconds of recorded audio. The VOIP application can also be developed in the following ways:

1. An application with full duplex capability can be developed.
2. The interface for the application can be upgraded to a more user-friendly Graphic User Interface (GUI)
3. The application can be expanded to provide more communication services to the user. These include e-mail, voicemail, live-chat, and VOIP via the network.

7.2 Contributions of Thesis Work

1. Thesis research has provided 2 multilevel compression algorithms that give compression performance comparable to the one-level DCT. This statement is supported by the average Compression Ratio and average Signal-to-Noise Ratio values obtained in the performance results section, and are summarized in Table 7.1.

   As can be seen from the table, the two multilevel DCT algorithms provide intermediate compression/distortion performance, that is, less than the one-level DCT algorithm, but greater than the one-level DST algorithm.

2. This thesis has also contributed a rudimentary half-duplex voice-over-IP (VOIP) software application, which can serve as the foundation for further VOIP application research and development in the future. The application allows a

computer user to send/receive up to 10 seconds of audio information to/from a remote computer user. Bandwidth reduction of the audio information is accomplished by using one of the transform coding compression techniques.

3. The thesis research has also resulted in a compression performance analysis software application (eval.exe). This application allows you to measure compression ratio, rate, entropy, mean-squared-error (mse), and Signal-to-Noise Ratio (SNR), once the user has supplied the appropriate compression/expansion file information.

Table 7.1 Average Compression Ratio/SNR Performance of Selected Transform Coding Algorithms

Algorithm	Avg. Compression Ratio	Avg. SNR
One-level DCT	2.644	1545.690
Three-level DCT	2.504	1424.809
Five-level DCT	2.398	1121.594
One-level DST	2.452	1304.015

References

[1] K. Sayhood, *Introduction to Data Compression*, San Francisco: Morgan Kaufmann, 2000.

[2] J. Stojaspal, *Wake-up Call. E-Europe – A Time Special Report,* Time-Europe (http://www.time.com/time/europe/specials/eeurope/field/telecoms.html.)

[3] A. Gersho, *Advances in Speech and Audio Compression*, Proceedings of IEEE, vol. 82, no. 6, pp. 900 – 918, June 1994.

[4] N. S. Jayant, *Digital Coding of Speech Waveforms: PCM, DPCM, and DM Quantizers,* Proceedings of IEEE, vol. 62, no. 5, pp. 611 – 632, May 1974.

[5] D. L. Cohn and J.L. Melsa, *The Residual Encoder – An Improved ADPCM System for Speech Digitization,* IEEE Transactions on Communications, vol. 23, no. 9, pp. 935 – 941, September 1975.

[6] M. Hansen, *Assessment and prediction of speech transmission quality with an auditory processing model,* Ph.D. Dissertation, University of Oldenburg, Oldenburg, Germany, 1998. (http://www.bis.uni-oldenburg.de/dissertation/hanass98/inhalt.html)

[7] J. D. Gibson, *Adaptive Prediction in Speech Differential Encoding Systems,* Proceedings of IEEE, vol. 68, no. 6, pp. 488 – 525, June 1980.

[8] K. R. Rao and P. Yip, *Discrete Cosine Transform: Algorithms, Advantages, Applications*, San Diego: Academic Press, 1990.

[9] D. E. Comer, *Computer Networks and Internets*, Upper Saddle River, New Jersey: Prentice Hall, 1999.

[10] Y. Be'ery, Z. Shpiro, T. Simchony, L. Shatz, and J. Piasetzky, *Advances in Speech Coding: An Efficient Variable-Bit-Rate Low-Delay CELP (VBR-LD-CELP) Coder,* Boston: Kluwer Academic Publishers, 1991.

[11] M. Nelson and J. L. Gailly, *The Data Compression Book,* New York: M&T Books, 1996.

APPENDIX

SOURCE CODE

Appendix 1 – Compression Algorithm C++ files

```
// Program:        DCT Compression/Expansion Functions
// Filename:       oneddct.cpp
// Description:    This module supplies audio compression/expansion
//                 functions that utilize a 1-D DCT as the means of
//                 compression.
// Author:         Craig Chin
// Date :          01/29/01
// Notes:          The code is a modification of the source code DCT.C obtained
//                 from "The Data Compression Book", by M. Nelson and J.L.
//                 Gailly.  This file must be linked to the files errhand.cpp, bitio.cpp,
//                 and utility_func.cpp for proper functionality.

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "bitio.h"
#include "errhand.h"
#include "utility_func.h"

// N defines the vector length, and the DCT matrix dimensions.

#define N 16

// This macro is used to ensure the correct rounding of integer values.

#define ROUND(a) (((a)<0) ? (int)((a) - 0.5) : (int)((a) + 0.5))

// Auxiliary Function Prototypes.

void Initialize(int quality);
int     InputCode(BIT_FILE *input);
void OutputCode(BIT_FILE *output_file, int code);
void WriteTformData(BIT_FILE *output_file, int output_vector[N]);
void WriteAudioData(FILE *output, unsigned char output_vector[N]);
void FowardDCT(unsigned char input[N], int output[N]);
void InverseDCT(int input[N], unsigned char output[N]);
void WriteEOFCode(BIT_FILE *output_file);

// These functions perform 1-D DCT compression and expansion

void OneDDCTC(void);
void OneDDCTE(void);

// These funtions use a combination of the 1-D DCT transform and Adaptive
// Huffman coding to produce compression.

void Dct1dAhuffC(void);
void Dct1dAhuffE(void);

// These functions are used to link the DCT algorithm to the Adaptive Huffman
// algorithm in order to obtain DCT-Adaptive Huffman compression/expansion
// functions.  Refer to file ahuff.cpp.

void PipeAHuffC(char inFileName[], char outFileName[]);
void PipeAHuffE(char inFileName[], char outFileName[]);

// These functions are used to compress and expand files for a client-server
// application.

void send_dctC(char inFileName[], char outFileName[]);
void recv_dctE(char inFileName[], char outFileName[]);

// Global data used at various places in the program.

double C[N][N];
double Ct[N][N];
int InputRunLength;
int OutputRunLength;
int Quantum[N];

// The initialization routine has the job of setting up the cosine
// transform matrix, as well as its transpose matrix.  These two matrices
// are used to caculate the DCT and its inverse.  In addition, the
// quantization matrix is set up based on the quality parameter that is
// input.  The two run-length variables are set to zero.

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

        for(i = 0; i < N; i++)
                Quantum[i] = 1 + ((1 + i) * quality);

        OutputRunLength = 0;
        InputRunLength = 0;

        for(j = 0; j < N; j++)
        {
                C[0][j] = 1.0/sqrt((double)N);
                Ct[j][0] = C[0][j];
        }
        for (i = 1; i < N; i++)
                for(j = 0; j < N; j++)
                {
                        C[i][j] = sqrt(2.0/N)*
                                        cos(pi*(2*j+1)*i/(2.0*N));
                        Ct[j][i] = C[i][j];
                }
}

// This routine reads in a transform code from a compressed file.  In
// general the code consists of two components, a bit count, and an
// encoded value.  There is also a code to signal the end of the compressed
// file.  The bit count is encoded as a prefix code with the following
// binary values:
//
//              Number of Bits                          Binary Code
//                      0                               00
//                      1                               010
//                      2                               011
//                      3                               10000
//                      4                               10001
//                      5                               10010
//                      6                               10011
//                      7                               10100
//                      8                               10101
//                      9                               10110
//                      10                              10111
//
// The code value used to signal the End-Of-File is 11111.
//
// A bit count of zero is followed by a four-bit number telling how many
// zeros are in the encoded run.  A value of one through ten indicates a
// code value follows, which takes up that many bits.  The encoding of
// values into this system has the following characteristics:
//
//              Bit Count                               Amplitudes
//                      1                               -1, 1
//                      2                               -3 to -2, 2 to 3
//                    3                                    -7 to -4, 4 to 7
//                    4                                    -15 to -8, 8 to 15
//                    5                                     -31 to -16, 16 to 31
//                    6                                     -63 to -32, 32 to 64
//                    7                                    -127 to -64, 64 to 127
//                    8                                    -255 to -128, 128 to 255
//                    9                                    -511 to -256, 256 to 511
//                    10                                   -1023 to -512, 512 to 1023

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

  if(InputRunLength > 0)
  {
        InputRunLength--;
    return(0);
  }
  bit_count = (int)InputBits(input_file, 2);
  if (bit_count == 0)
  {
        InputRunLength = (int)InputBits(input_file, 4);
    return(0);
  }
  if (bit_count == 1)
  {
        bit_count = (int) InputBits(input_file, 1) + 1;
    result = (int) InputBits(input_file, bit_count);
       if (result & (1 << (bit_count - 1)))
       return (result);
       return(result - (1 << bit_count) + 1);
  }
  if (bit_count == 2)
  {
         bit_count = (int) InputBits(input_file, 3) + (bit_count << 3) - 13;
     result = (int) InputBits(input_file, bit_count);
       if (result & (1 << (bit_count - 1)))
       return (result);
       return(result - (1 << bit_count) + 1);
  }
  else
  {
       bit_count = (int) InputBits(input_file, 3) + (bit_count << 3);
    if (bit_count == 31)
```

```
            return (-12375);
        else
        {
            printf("There has been an error in input file termination.\n");
                    return(0);
             }
    }
            return(-1);
}

// This routine outputs a code to the compressed transform file.  For specs
// on the code format, see the comments that accompany InputCode().

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

  if(code == 0)
  {
         OutputRunLength++;
    return;
  }

  if (OutputRunLength != 0)
  {
        while (OutputRunLength > 0)
    {
      OutputBits(output_file, 0L, 2);
        if (OutputRunLength <= 16)
      {
        OutputBits(output_file, (unsigned long)(OutputRunLength - 1), 4);
        OutputRunLength = 0;
      }
      else
      {
        OutputBits(output_file, 15L, 4);
        OutputRunLength -= 16;
      }
    }
  }

  if (code < 0)
        abs_code = -code;
  else
        abs_code = code;
  top_of_range = 1;
  bit_count = 1;
  while (abs_code > top_of_range)
  {
        bit_count++;
    top_of_range = ((top_of_range + 1) * 2) - 1;
  }
  if (bit_count < 3)
        OutputBits(output_file, (unsigned long)(bit_count + 1), 3);
  else
        OutputBits(output_file, (unsigned long)(bit_count + 13), 5);
  if (code > 0)
        OutputBits(output_file, (unsigned long)code, bit_count);
  else
        OutputBits(output_file, (unsigned long)(code + top_of_range), bit_count);
}

// This routine takes transform data, quantizes it, and outputs the code.

void WriteTformData(BIT_FILE *output_file, int output_vector[N])
{
  int i;
  double result;

  for(i = 0; i < N; i++)
  {
        result = output_vector[i]/Quantum[i];
    OutputCode(output_file, ROUND(result));
  }
}

// This routine writes an End-of-File prefix code to compressed file.

void WriteEOFCode(BIT_FILE *output_file)
{
        OutputBits(output_file, 31L , 5);
}

// This routine writes out the audio data to raw format file.

void WriteAudioData(FILE *output, unsigned char output_vector[N])
{
        int i;
  for(i = 0; i < N; i++)
        putc(output_vector[i], output);
}

// The Foward DCT routine implements the matrix function:
//      DCT = C * input_vector.

void FowardDCT(unsigned char input[N], int output[N])
{
  double temp;
  int i;
  int k;

//MatrixMultiply(output, C, input)
  for (i = 0; i < N; i++)
  {
        temp = 0.0;
    for (k = 0; k < N; k++)
        temp += C[i][k] * ((int)input[k] - 128);
    output[i] = ROUND(temp);
  }
}

// The inverse DCT routine implements the matrix function:
//      output_vector = Ct * DCT

void InverseDCT(int DCT[N], unsigned char output[N])
{
  double temp;
  int i;
  int k;

//MatrixMultiply(output, Ct, DCT)
  for (i = 0; i < N; i++)
  {
        temp = 0.0;
    for (k = 0; k < N; k++)
        temp += Ct[i][k] * DCT[k];
    temp += 128.0;
    if (temp < 0)
        output[i] = 0;
    else if (temp > 255)
        output[i] = 255;
    else
        output[i] = (unsigned char) ROUND(temp);
  }
}


// This is a 1-D DCT compression function.  The compression function requires
// a user input, the quality value, ranging from 0 - 25.

void OneDDCTC(void)
{
        char inFileName[80];
        char outFileName[80];
        FILE *input;
        BIT_FILE *output;
        int i;
        int c;
        unsigned char input_vector[N];
        int output_vector [N];
        int quality;
        char *CompressionName = "DCT algorithm";
        time_t first, second;
        double time_diff;

        // Obtain and locate the input and output files for compression

        cout << "Enter the input filename.\n";
        cin >> inFileName;
        input = fopen(inFileName, "rb");
        if (input == NULL)
                fatal_error("Error opening %s for input\n", inFileName);
        cout << "Enter the output filename.\n";
        cin >> outFileName;
        output = OpenOutputBitFile(outFileName);
        if (output == NULL)
                fatal_error("Error opening %s for output\n", outFileName);

        // Obtain quality factor

        cout << "You are required to enter a quality factor that determines\n";
        cout << "the degree of quantization.  Quality values range from 0-25\n";
        cout << "Please enter the value you desire.\n";
        cin >> quality;
        if (quality < 0 || quality > 25)
        {
                fatal_error("Illegal quality factor of %d\n", quality);
                quality = 3;
                printf("Using quality factor of %d\n", quality);
        }
        printf("\nCompressing %s to %s\n", inFileName, outFileName);
        printf("Using %s.\n", CompressionName);

first = time(NULL);

// Initialize quantization and DCT matrix values and
// write quality value to compression output

Initialize(quality);
OutputBits(output, (unsigned long) quality, 8);

do
{
                // fill input vector with audio file input

                for (i = 0; i < N; i++)
                {
                c = getc(input);

                        // if file size is not a multiple of N fill remaining elements with
                        // zeros.

                        if (c == EOF)
                        {
                for (;i < N; i++)
                                        input_vector[i] = 0;
                                break;
                        }
                        else
                                input_vector[i] = (unsigned char) c;

                }

                // Perform DCT transform and write the data to compression output file

                FowardDCT(input_vector, output_vector);
                WriteTformData(output, output_vector);
                if (c ==EOF)
                WriteEOFCode(output);
        }while(c != EOF);

        // Calculate time for completion of compression algorithm.
        second = time(NULL);
        time_diff = difftime(second, first);
        cout << "\nTime for algorithm completion = " << time_diff << " seconds.\n";

        // Close Input and Output files.  Display Compression Information

        CloseOutputBitFile(output);
        fclose(input);
        print_ratios(inFileName, outFileName);
}

// This is a 1-D DCT expansion function.  The expansion function reads in
// the compressed data from the DCT file, then writes out the decompressed
// audio file.

void OneDDCTE(void)
{
        char inFileName[80];
        char outFileName[80];
        BIT_FILE *input;
        FILE *output;
        int i;
        int input_vector[N];
        unsigned char output_vector[N];
        int quality;
        char *CompressionName = "DCT algorithm";
        int EOFTest;
        time_t first, second;
        double time_diff;

        // Obtain and locate the input and output files for expansion

        cout << "Enter the input filename.\n";
        cin >> inFileName;
        input = OpenInputBitFile(inFileName);
        if (input == NULL)
                fatal_error("Error opening %s for input\n", inFileName);
        cout << "Enter the output filename.\n";
        cin >> outFileName;
        output = fopen(outFileName, "wb");
        if (output == NULL)
                fatal_error("Error opening %s for output\n", outFileName);
        printf("\nExpanding %s to %s\n", inFileName, outFileName);
        printf("Using %s.\n", CompressionName);
        first = time(NULL);

        // Read quality value from the compressed file and initialize
        // quantization and DCT matrix values.

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

        // Decode compressed input, perform inverse transform, and write
        // audio data to expansion output file

        do
        {
                for (i = 0; i < N; i++)
                {
                        EOFTest = InputCode(input);
                        if (EOFTest == -12375)
                                break;
                        else
                        input_vector[i] = EOFTest * Quantum[i];
                }
                if (EOFTest != -12375)
                {
                InverseDCT(input_vector, output_vector);
                WriteAudioData(output, output_vector);
                }
        }while(EOFTest != -12375);

        // Calculate time for completion of expansion algorithm.

        second = time(NULL);
        time_diff = difftime(second, first);
        cout << "\nTime for algorithm completion = " << time_diff << " seconds.\n";

        // Close Input and Output files.

        CloseInputBitFile(input);
        fclose(output);
        putc('\n', stdout);
}

// This is a DCT/Adaptive Huffman compression function.  The compression
// function utilizes a 1-D DCT compression algorithm followed by an Adaptive
// Huffman algorithm to obtain a compressed output.  The function definition
// of the adaptive Huffman algorithm is found in file ahuff.cpp.  The
// program requires a user input, the quality value, ranging from 0 - 25.

void Dct1dAhuffC(void)
{
        char inFileName[80], outFileName[80];
        FILE *input;
        BIT_FILE *interfile;
        int i, c, quality, output_vector [N];
        unsigned char input_vector[N];
        char *CompressionName = "1-D DCT - Adaptive Huffman algorithm";
        time_t first, second;
        double time_diff;

        // Obtain input and output filenames.  Locate input and interim files.

        cout << "Enter the input filename.\n";
        cin >> inFileName;
        input = fopen(inFileName, "rb");
        if (input == NULL)
                fatal_error("Error opening %s for input\n", inFileName);
        cout << "Enter the output filename.\n";
        cin >> outFileName;
                interfile = OpenOutputBitFile("interfile");
        if (interfile == NULL)
                fatal_error("Error opening %s for output\n", "interfile");

        // Obtain quality factor

        cout << "You are required to enter a quality factor that determines\n";
        cout << "the degree of quantization.  Quality values range from 0-25\n";
        cout << "Please enter the value you desire.\n";
        cin >> quality;
        if (quality < 0 || quality > 25)
        {
                fatal_error("Illegal quality factor of %d\n", quality);
                quality = 3;
                printf("Using quality factor of %d\n", quality);
        }
        printf("\nCompressing %s to %s\n", inFileName, outFileName);
        printf("Using %s.\n", CompressionName);

        first = time(NULL);

        // Initialize quantization and DCT matrix values and
// write quality value to compression output

Initialize(quality);
OutputBits(interfile, (unsigned long) quality, 8);

do
{
        // fill input vector with audio file input

        for (i = 0; i < N; i++)
        {
        c = getc(input);

                // if file size is not a multiple of N fill remaining elements with
                // zeros.

                if (c == EOF)
                {
        for (;i < N; i++)
                                input_vector[i] = 0;
                        break;
                }
                else
                        input_vector[i] = (unsigned char) c;
        }

        // Perform DCT transform and write the data to interim file

        FowardDCT(input_vector, output_vector);
        WriteTformData(interfile, output_vector);
        if (c ==EOF)
        WriteEOFCode(interfile);
}while(c != EOF);

//Close Input and Interim files.

CloseOutputBitFile(interfile);
fclose(input);

//Call Adaptive Huffman Compression Algorithm

PipeAHuffC("interfile", outFileName);

// Calculate time for completion of compression algorithm.
```

```
        second = time(NULL);
        time_diff = difftime(second, first);
        cout << "\nTime for algorithm completion = " << time_diff << " seconds.\n";

        //Display Compression Information

        print_ratios(inFileName, outFileName);
}

// This is a DCT/Adaptive Huffman expansion function. The function reads in
// the compressed data and decompresses it using an adaptive Huffman
// algorithm followed by a 1-D DCT algorithm.  The decompressed output is
// written to an audio file.  The function definition of the adaptive
// Huffman algorithm is found in file ahuff.cpp.

void Dct1dAhuffE(void)
{
        char inFileName[80], outFileName[80];
        BIT_FILE *interfile;
        FILE *output;
        int i, input_vector[N], quality, EOFTest;
        unsigned char output_vector[N];
        char *CompressionName = "1-D DCT - Adaptive Huffman algorithm";
        time_t first, second;
        double time_diff;

        // Obtain input and output filenames.  Locate output file.

        cout << "Enter the input filename.\n";
        cin >> inFileName;
        cout << "Enter the output filename.\n";
        cin >> outFileName;
        output = fopen(outFileName, "wb");
        if (output == NULL)
                fatal_error("Error opening %s for output\n", outFileName);
        printf("\nExpanding %s to %s\n", inFileName, outFileName);
        printf("Using %s.\n", CompressionName);

        first = time(NULL);

        // expand input file using the adaptive Huffman algorithm

        PipeAHuffE(inFileName, "interfile");

        // open Adaptive Huffman output (interim file) for use in DCT algorithm
        interfile = OpenInputBitFile("interfile");
        if (interfile == NULL)
                fatal_error("Error opening %s for input\n", "interfile");

        // Read quality value from the compressed file and initialize
        // quantization and DCT matrix values.

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

        // Decode compressed input, perform inverse transform, and write
        // audio data to expansion output file

        do
        {
                for (i = 0; i < N; i++)
                {
                        EOFTest = InputCode(interfile);
                        if (EOFTest == -12375)
                                break;
                        else
                        input_vector[i] = EOFTest * Quantum[i];
                }
                if (EOFTest != -12375)
                {
                InverseDCT(input_vector, output_vector);
                WriteAudioData(output, output_vector);
                }
        }while(EOFTest != -12375);

        // Calculate time for completion of expansion algorithm.

        second = time(NULL);
        time_diff = difftime(second, first);
        cout << "\nTime for algorithm completion = " << time_diff << " seconds.\n";

        // Close Input and Output files.

        CloseInputBitFile(interfile);
        fclose(output);
        putc('\n', stdout);
}
```

```
// This is the compression routine used to compress the file to be sent
// to a remote computer.  Compression is obtained by using a 1-D DCT
// algorithm with a fixed quality value of 1.

void send_dctC(char inFileName[], char outFileName[])
{
	FILE *input;
	BIT_FILE *output;
	int i;
	int c;
	unsigned char input_vector[N];
	int output_vector [N];
	const int quality = 1;
	char *CompressionName = "DCT algorithm";

	// Obtain and locate the input and output files for compression

	input = fopen(inFileName, "rb");
	if (input == NULL)
		fatal_error("Error opening %s for input\n", inFileName);
	output = OpenOutputBitFile(outFileName);
	if (output == NULL)
		fatal_error("Error opening %s for output\n", outFileName);
	printf("\nCompressing %s to %s\n", inFileName, outFileName);
	printf("Using %s.\n", CompressionName);

	// Initialize quantization and DCT matrix values

	Initialize(quality);

	do
	{
		// fill input vector with audio file input

		for (i = 0; i < N; i++)
		{
		c = getc(input);

			// if file size is not a multiple of N fill remaining elements with
			// zeros.

			if (c == EOF)
			{
		for (;i < N; i++)
					input_vector[i] = 0;
```

```
                        break;
                    }
                    else
                        input_vector[i] = (unsigned char) c;
            }

            // Perform DCT transform and write the data to compression output file

            FowardDCT(input_vector, output_vector);
        WriteTformData(output, output_vector);
            if (c ==EOF)
            WriteEOFCode(output);
        }while(c != EOF);

        // Close Input and Output files.

        CloseOutputBitFile(output);
        fclose(input);

        // Display Compression Information

        print_ratios(inFileName, outFileName);
}


// This is the expansion routine used to expand the compressed file
// sent from the remote computer.  Expansion is obtained by using a 1-D DCT
// algorithm with a fixed quality value of 1.

void recv_dctE(char inFileName[], char outFileName[])
{
        BIT_FILE *input;
        FILE *output;
        int i;
        int EOFTest;
        int input_vector[N];
        unsigned char output_vector[N];
        const int quality = 1;
        char *CompressionName = "DCT algorithm";

        // Obtain and locate the input and output files for expansion

        input = OpenInputBitFile(inFileName);
        if (input == NULL)
            fatal_error("Error opening %s for input\n", inFileName);
        output = fopen(outFileName, "wb");
        if (output == NULL)
                fatal_error("Error opening %s for output\n", outFileName);
        printf("\nExpanding %s to %s\n", inFileName, outFileName);
        printf("Using %s.\n", CompressionName);
        printf("\rUsing quality factor of %d\n", quality);

        // Initialize quantization and DCT matrix values.

        Initialize(quality);

        // Decode compressed input, perform inverse transform, and write
        // audio data to expansion output file

        do
        {
                for (i = 0; i < N; i++)
                {
                        EOFTest = InputCode(input);
                        if (EOFTest == -12375)
                                break;
                        else
                        input_vector[i] = EOFTest * Quantum[i];
                }
                if (EOFTest != -12375)
                {
                InverseDCT(input_vector, output_vector);
                WriteAudioData(output, output_vector);
                }
        }while(EOFTest != -12375);

        // Close Input and Output files.

        CloseInputBitFile(input);
        fclose(output);
        putc('\n', stdout);
}


// Program:        DST Compression/Expansion Functions
// Filename:       oneddst.cpp
// Description:    This module supplies audio compression/expansion functions that
//                 utilize a 1-D DST as the means of compression.
// Author:         Craig Chin
// Date :          02/06/01
// Notes:          The code is a modification of the source code DCT.C obtained
//                 from "The Data Compression Book", by M. Nelson and J.L.
//                 Gailly.  This file must be linked to the files oneddct.cpp,
//                 errhand.cpp, bitio.cpp, and utility_func.cpp for proper
//                 functionality.

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "bitio.h"
#include "errhand.h"
#include "utility_func.h"


// N defines the vector length, and the DST matrix dimensions

#define N 16

// This macro is used to ensure the correct rounding of integer values

#define ROUND(a) (((a)<0) ? (int)((a) - 0.5) : (int)((a) + 0.5))

// Auxiliary Function Prototypes.  All unspecified function definitions
// may be  found in the file ONEDDCT.CPP

void SInitialize(int quality);
int InputCode(BIT_FILE *input);
void OutputCode(BIT_FILE *output_file, int code);
void WriteTformData(BIT_FILE *output_file, int output_vector[N]);
void WriteAudioData(FILE *output, unsigned char output_vector[N]);
void FowardDST(unsigned char input[N], int output[N]);
void InverseDST(int input[N], unsigned char output[N]);
void WriteEOFCode(BIT_FILE *output_file);

// Global data used at various places in this program. (Some are defined
//in ONEDDCT.CPP

double S[N][N];
double St[N][N];
extern int InputRunLength;
extern int OutputRunLength;
extern int Quantum[N];
```

```
// The initialization routine has the job of setting up the sine
// transform matrix, as well as its transpose matrix.  These two matrices
// are used to caculate the DST and its inverse.  In addition, the
// quantization matrix is set up based on the quality parameter that is
// input.  The two run-length variables are set to zero.

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

        for(i = 0; i < N; i++)
                Quantum[i] = 1 + ((1 + i) * quality);

        OutputRunLength = 0;
        InputRunLength = 0;

        for (i = 0; i < N; i++)
                for(j = 0; j < N; j++)
                {
                        S[i][j] = sqrt(2.0/(double)(N+1))*
                                                    sin(pi*(i+1)*(j+1)/(double)(N+1));
                        St[j][i] = S[i][j];
                }
}

// The Foward DST routine implements the matrix function:
//
//              DST = S * input_vector

void FowardDST(unsigned char input[N], int output[N])
{
        double temp;
        int i;
        int k;

  // MatrixMultiply(output, S, input)

        for (i = 0; i < N; i++)
        {
                temp = 0.0;
                for (k = 0; k < N; k++)
                temp += S[i][k] * ((int)input[k] - 128);
                output[i] = ROUND(temp);
        }
}

// The inverse DCT routine implements the matrix function:
//
//              output_vector = St * DST

void InverseDST(int DST[N], unsigned char output[N])
{
  double temp;
  int i;
  int k;

  // MatrixMultiply(output, St, DST)

  for (i = 0; i < N; i++)
  {
        temp = 0.0;
    for (k = 0; k < N; k++)
        temp += St[i][k] * DST[k];
    temp += 128.0;
    if (temp < 0)
        output[i] = 0;
    else if (temp > 255)
        output[i] = 255;
    else
        output[i] = (unsigned char) ROUND(temp);
  }
}


// This is a 1-D DST compression function.  The compression function requires
// a user input, the quality value, ranging from 0 - 25.

void OneDDSTC(void)
{
        char inFileName[80];
        char outFileName[80];
        FILE *input;
        BIT_FILE *output;
        int i, c, quality;
        unsigned char input_vector[N];
        int output_vector [N];
        char *CompressionName = "DCT algorithm";
        time_t first, second;
```

```
double time_diff;

// Obtain and locate the input and output files for compression

cout << "Enter the input filename.\n";
cin >> inFileName;
input = fopen(inFileName, "rb");
if (input == NULL)
        fatal_error("Error opening %s for input\n", inFileName);
cout << "Enter the output filename.\n";
cin >> outFileName;
output = OpenOutputBitFile(outFileName);
if (output == NULL)
        fatal_error("Error opening %s for output\n", outFileName);

// Obtain quality factor

cout << "You are required to enter a quality factor that determines\n";
cout << "the degree of quantization.  Quality values range from 0-25\n";
cout << "Please enter the value you desire.\n";
cin >> quality;
if (quality < 0 || quality > 25)
{
        fatal_error("Illegal quality factor of %d\n", quality);
        quality = 3;
        printf("Using quality factor of %d\n", quality);
}
printf("\nCompressing %s to %s\n", inFileName, outFileName);
printf("Using %s.\n", CompressionName);

first = time(NULL);

// Initialize quantization and DST matrix values and
// write quality value to compression output

SInitialize(quality);
OutputBits(output, (unsigned long) quality, 8);

do
{
        // fill input vector with audio file input

        for (i = 0; i < N; i++)
        {
        c = getc(input);
```

```
                // if file size is not a multiple of N fill remaining elements with
                // zeros.

                if (c == EOF)
                {
            for (;i < N; i++)
                                input_vector[i] = 0;
                        break;
                }
                else
                        input_vector[i] = (unsigned char) c;
            }
            FowardDST(input_vector, output_vector);
            WriteTformData(output, output_vector);
            if (c ==EOF)
            WriteEOFCode(output);
        }while(c != EOF);


        second = time(NULL);
        time_diff = difftime(second, first);
        cout << "\nTime for algorithm completion = " << time_diff << " seconds.\n";

        // Close Input and Output files.  Display Compression Information

        CloseOutputBitFile(output);
        fclose(input);
        print_ratios(inFileName, outFileName);
}

// The expansion routine reads in the compressed data from the DST file,
// then writes out the decompressed audio file.

void OneDDSTE(void)
{
        char inFileName[80];
        char outFileName[80];
        BIT_FILE *input;
        FILE *output;
        int i, quality, EOFTest, input_vector[N] ;
        unsigned char output_vector[N];
        char *CompressionName = "DCT algorithm";
        time_t first, second;
        double time_diff;
```

```
// Obtain and locate the input and output files for expansion

cout << "Enter the input filename.\n";
cin >> inFileName;
input = OpenInputBitFile(inFileName);
if (input == NULL)
        fatal_error("Error opening %s for input\n", inFileName);
cout << "Enter the output filename.\n";
cin >> outFileName;
output = fopen(outFileName, "wb");
if (output == NULL)
        fatal_error( "Error opening %s for output\n", outFileName);
printf("\nExpanding %s to %s\n", inFileName, outFileName);
printf("Using %s.\n", CompressionName);

first = time(NULL);

// Read quality value from the compressed file and initialize
// quantization and DCT matrix values.

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

// Decode compressed input, perform inverse transform, and write
// audio data to expansion output file

do
{
        for (i = 0; i < N; i++)
        {
                EOFTest = InputCode(input);
                if (EOFTest == -12375)
                        break;
                else
                input_vector[i] = EOFTest * Quantum[i];
        }
        if (EOFTest != -12375)
        {
        InverseDST(input_vector, output_vector);
        WriteAudioData(output, output_vector);
        }
}while(EOFTest != -12375);
```

```
        // Calculate time for completion of expansion algorithm.

        second = time(NULL);
        time_diff = difftime(second, first);
        cout << "\nTime for algorithm completion = " << time_diff << " seconds.\n";

        // Close Input and Output files.

        CloseInputBitFile(input);
        fclose(output);
        putc('\n', stdout);
}

// Program:          Multi-Level DCT Compression/Expansion Functions
// Filename:         multiDCT.cpp
// Description:      This module implements an audio compression program based on
//                   the 1-D DCT.  It performs the DCT matrix multiplication a
//                   specified number of times for both the compression and expansion
//                   algorithms.
// Author:           Craig Chin
// Date :            02/05/01
// Notes:            The code is a modification of the source code DCT.C obtained
//                   from "The Data Compression Book", by M. Nelson and J.L.
//                   Gailly.  This file must be linked to the files oneddct.cpp,
//                   errhand.cpp, bitio.cpp, and utility_func.cpp for proper
//                   functionality.

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "bitio.h"
#include "errhand.h"
#include "utility_func.h"

// N defines the vector length, and the DCT matrix dimensions

#define N 16

// This macro is used to ensure the correct rounding of integer values

#define ROUND(a) (((a)<0) ? (int)((a) - 0.5) : (int)((a) + 0.5))

// Auxiliary Function Prototypes.  All unavailable function definitions may be
// found in oneddct.cpp

void Initialize(int quality);
int InputCode(BIT_FILE *input);
void OutputCode(BIT_FILE *output_file, int code);
void WriteTformData(BIT_FILE *output_file, int output_vector[N]);
void WriteAudioData(FILE *output, unsigned char output_vector[N]);
void MultiFowardDCT(unsigned char input[N], int output[N], int transform_no);
void MultiInverseDCT(int DCT[N], unsigned char output[N], int transform_no);
void WriteEOFCode(BIT_FILE *output_file);

// These functions implement the multi-level DCT compression/expansion
// algorithms.

void MultiDCTC(void);
void MultiDCTE(void);

// Global data used that are defined in ONEDDCT.CPP

extern double C[N][N];
extern double Ct[N][N];
extern int InputRunLength;
extern int OutputRunLength;
extern int Quantum[N];

// The Foward DCT routine implements the matrix function:
//
//              DCT = C^(transform_no) * input_vector

void MultiFowardDCT(unsigned char input[N], int output[N], int transform_no)
{
        double temp_vector[N][5];
        int i, j, k;

        //Initialize temp_vector matrix

        for (i = 0; i < N; i++)
                for (j = 0; j < 5; j++)
                        temp_vector[i][j] = 0.0;

        //MatrixMultiply(output, C^(transform_no), input)

        for (i = 0; i < N; i++)
                for (k = 0; k < N; k++)
                temp_vector[i][0] += C[i][k] * ((int)input[k] - 128);
        if (transform_no > 1)
                for (j = 0; j < (transform_no - 1); j++)
                for (i = 0; i < N; i++)
                                for (k = 0; k < N; k++)
                                temp_vector[i][j+1] += C[i][k] * temp_vector[k][j];

        for (i = 0; i < N; i++)
        output[i] = ROUND(temp_vector[i][transform_no - 1]);
}

// The inverse DCT routine implements the matrix function:
//
//      output_vector = Ct^(transform_no) * DCT

void MultiInverseDCT(int DCT[N], unsigned char output[N], int transform_no)
{
        double temp_vector[N][5];
        int i, j, k;

        //Initialize temp_vector matrix

        for (i = 0; i < N; i++)
                for (j = 0; j < 5; j++)
                        temp_vector[i][j] = 0.0;

        //MatrixMultiply(output, Ct^(tranform_no), DCT)

        for (i = 0; i < N; i++)
                for (k = 0; k < N; k++)
                temp_vector[i][0] += Ct[i][k] * DCT[k];

        if (transform_no > 1)
                for (j = 0; j < (transform_no - 1); j++)
                for (i = 0; i < N; i++)
                for (k = 0; k < N; k++)
                        temp_vector[i][j+1] += Ct[i][k] * temp_vector[k][j];

        for (i = 0; i < N; i++)
        {
                temp_vector[i][transform_no - 1] += 128.0;
                if (temp_vector[i][transform_no - 1] < 0)
                output[i] = 0;
                else if (temp_vector[i][transform_no - 1] > 255)
                output[i] = 255;
```

```
            else
            output[i] = (unsigned char) ROUND(temp_vector[i][transform_no - 1]);
        }
}

// This the multi-level DCT compression routine.  The compression routine
// requires 2 user inputs, the quality value, ranging from 0 - 25, and the number of
// DCT multiplications to be performed, ranging from 1 to 5.

void MultiDCTC(void)
{
        char inFileName[80];
        char outFileName[80];
        FILE *input;
        BIT_FILE *output;
        unsigned char input_vector[N];
        int output_vector [N];
        int quality, transform_no, i, c;
        char *CompressionName = "Multi-DCT algorithm";
        time_t first, second;
        double time_diff;

        // Obtain and locate the input and output files for compression

        cout << "Enter the input filename.\n";
        cin >> inFileName;
        input = fopen(inFileName, "rb");
        if (input == NULL)
            fatal_error("Error opening %s for input\n", inFileName);
        cout << "Enter the output filename.\n";
        cin >> outFileName;
        output = OpenOutputBitFile(outFileName);
        if (output == NULL)
            fatal_error("Error opening %s for output\n", outFileName);

        // Obtain the quality factor and number of DCT transforms

        cout << "You are required to enter the quality factor that determines\n";
        cout << "the degree of quantization.  Quality values range from 0-25\n";
        cout << "Please enter the value you desire.\n";
        cin >> quality;
        if (quality < 0 || quality > 25)
        {
            fatal_error("Illegal quality factor of %d\n", quality);
            quality = 3;
        printf("Using quality factor of %d\n", quality);
}
cout << "You are also required to enter the number of DCT transforms to\n";
cout << "be performed.  This number ranges from 1 - 5.\n";
cout << "Please enter the value you desire.\n";
cin >> transform_no;
if (transform_no < 1 || transform_no > 5)
{
        fatal_error("Illegal transform number of %d\n", transform_no);
        transform_no = 1;
        printf("Using transform number of %d\n", transform_no);
}

printf( "\nCompressing %s to %s\n", inFileName, outFileName );
printf( "Using %s.\n", CompressionName);

first = time(NULL);

// Initialize quantization and DCT matrix values

Initialize(quality);

// Output quality and transform_no information to compression output

OutputBits(output, (unsigned long) quality, 8);
OutputBits(output, (unsigned long) transform_no, 8);

do
{
        // fill input vector with audio file input

        for (i = 0; i < N; i++)
        {
        c = getc(input);

                // if file size is not a multiple of N fill remaining elements
                // with zeros

                if (c == EOF)
                {
        for (;i < N; i++)
                                input_vector[i] = 0;
                        break;
                }
                else
```

```
                    input_vector[i] = (unsigned char) c;
            }

            // Perform multi-level transform and write the data to compression
            // output file

            MultiFowardDCT(input_vector, output_vector, transform_no);
            WriteTformData(output, output_vector);
            if (c ==EOF)
            WriteEOFCode(output);
        }while(c != EOF);

        // Calculate time for completion of compression algorithm.

        second = time(NULL);
        time_diff = difftime(second, first);
        cout << "\nTime for algorithm completion = " << time_diff << " seconds.\n";

        // Close Input and Output files.  Display Compression Information

        CloseOutputBitFile(output);
        fclose(input);
        print_ratios(inFileName, outFileName);
}

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

void MultiDCTE(void)
{
        char inFileName[80];
        char outFileName[80];
        BIT_FILE *input;
        FILE *output;
        int input_vector[N];
        unsigned char output_vector[N];
        int quality, transform_no, i, EOFTest;
        char *CompressionName = "Multi-DCT algorithm";
        time_t first, second;
        double time_diff;

        //Obtain and locate the input and output files for expansion

        cout << "Enter the input filename.\n";
        cin >> inFileName;
```

```
input = OpenInputBitFile(inFileName);
if (input == NULL)
          fatal_error("Error opening %s for input\n", inFileName);
cout << "Enter the output filename.\n";
cin >> outFileName;
output = fopen(outFileName, "wb");
if (output == NULL)
          fatal_error("Error opening %s for output\n", outFileName);
printf("\nExpanding %s to %s\n", inFileName, outFileName);
printf("Using %s.\n", CompressionName);

first = time(NULL);

// Read quality and tranform_no values from the compressed file
// and initialize quantization and DCT matrix values.

quality = (int)InputBits(input, 8);
transform_no = (int)InputBits(input, 8);
cout << "\nUsing a quality factor of " << quality << "\n";
cout << "And a transform number of " << transform_no << "\n";
Initialize(quality);

// Decode compressed input, perform inverse multi-level transform,
// and write audio data to expansion output file

do
{
          for (i = 0; i < N; i++)
          {
                    EOFTest = InputCode(input);
                    if (EOFTest == -12375)
                              break;
                    else
                    input_vector[i] = EOFTest * Quantum[i];
          }
          if (EOFTest != -12375)
          {
          MultiInverseDCT(input_vector, output_vector, transform_no);
          WriteAudioData(output, output_vector);
          }
}while(EOFTest != -12375);

// Calculate time for completion of expansion algorithm.

second = time(NULL);
```

```
        time_diff = difftime(second, first);
        cout << "\nTime for algorithm completion = " << time_diff << " seconds.\n";

        // Close Input and Output files.

        CloseInputBitFile(input);
        fclose(output);
        putc('\n', stdout);
}

// Program:             DCT/DST Compression/Expansion Functions
// Filename:            DCT_DST.cpp
// Description:         This module implements an audio compression program based on
//                      the 1-D DCT and the 1-D DST.  It performs a DCT matrix
//                      multiplication followed by a DST matrix multiplication for
//                      compression and vice versa for expansion.
// Author:              Craig Chin
// Date :               02/06/01
// Notes:               The code is a modification of the source code DCT.C obtained
//                      from "The Data Compression Book", by M. Nelson and J.L.
//                      Gailly.  This file must be linked to the files oneddct.cpp,
//                      oneddst.cpp, errhand.cpp, bitio.cpp, and utility_func.cpp for proper
//                      functionality.

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "bitio.h"
#include "errhand.h"
#include "utility_func.h"

// N defines the vector length, and the DCT/DST matrix dimensions

#define N 16

// This macro is used to ensure the correct rounding of integer values

#define ROUND(a) (((a)<0) ? (int)((a) - 0.5) : (int)((a) + 0.5))

// Auxiliary Function Prototypes.  All unavailable function definitions may be
// found in oneddct.cpp and oneddst.cpp

void Initialize(int quality);
void InitializeSine(void);
int InputCode(BIT_FILE *input);
void OutputCode(BIT_FILE *output_file, int code);
void WriteTformData(BIT_FILE *output_file, int output_vector[N]);
void WriteAudioData(FILE *output, unsigned char output_vector[N]);
void FowardDCT_DST(unsigned char input[N], int output[N]);
void InverseDCT_DST(int DCT[N], unsigned char output[N]);
void WriteEOFCode(BIT_FILE *output_file);

// These functions implement the DCT/DST compression/expansion algorithms.

void DCT_DSTC(void);
void DCT_DSTE(void);

// Global data used that are defined in oneddct.cpp and oneddst.cpp

extern double C[N][N];
extern double Ct[N][N];
extern double S[N][N];
extern double St[N][N];
extern int InputRunLength;
extern int OutputRunLength;
extern int Quantum[N];

// The initialization routine has the job of setting up the sine
// transform matrix, as well as its transpose matrix.  These two matrices
// are used to caculate the DST and its inverse.

void InitializeSine(void)
{
        int i;
        int j;
        double pi = atan(1.0)*4.0;

        for (i = 0; i < N; i++)
                for(j = 0; j < N; j++)
                {
                        S[i][j] = sqrt(2.0/(double)(N+1))*
                                                sin(pi*(i+1)*(j+1)/(double)(N+1));
                        St[j][i] = S[i][j];
                }
}
```

```
// The Foward DCT_DST routine implements the matrix function:
//
//              output_vector = S * C * input_vector

void FowardDCT_DST(unsigned char input[N], int output[N])
{
        double temp_vector[N][2];
        int i, j;

        // Initialize temp_vector matrix

        for (i = 0; i < N; i++)
                for (j = 0; j < 2; j++)
                        temp_vector[i][j] = 0.0;

        // MatrixMultiply(temp, C, input)

        for (i = 0; i < N; i++)
                for (j = 0; j < N; j++)
                temp_vector[i][0] += C[i][j] * ((int)input[j] - 128);

        // MatrixMultiply(output, S, temp)

        for (i = 0; i < N; i++)
                for (j = 0; j < N; j++)
                temp_vector[i][1] += S[i][j] * temp_vector[j][0];

        for (i = 0; i < N; i++)
        output[i] = ROUND(temp_vector[i][1]);
}

// The inverse DCT_DST routine implements the matrix function:
//
//      output_vector = Ct * St * DCT

void InverseDCT_DST(int DCT[N], unsigned char output[N])
{
        double temp_vector[N][2];
        int i, j;

        // Initialize temp_vector matrix

        for (i = 0; i < N; i++)
                for (j = 0; j < 2; j++)
```

```
                    temp_vector[i][j] = 0.0;

        // MatrixMultiply(temp, St, DCT)

        for (i = 0; i < N; i++)
                for (j = 0; j < N; j++)
                temp_vector[i][0] += St[i][j] * DCT[j];

        // MatrixMultiply(output, Ct, temp)

    for (i = 0; i < N; i++)
                for (j = 0; j < N; j++)
                        temp_vector[i][1] += Ct[i][j] * temp_vector[j][0];

        for (i = 0; i < N; i++)
        {
                temp_vector[i][1] += 128.0;
                if (temp_vector[i][1] < 0)
                output[i] = 0;
                else if (temp_vector[i][1] > 255)
                output[i] = 255;
                else
                output[i] = (unsigned char) ROUND(temp_vector[i][1]);
        }
}

// This the DCT_DST compression routine.  The compression routine
// requires 1 user inputs, the quality value, ranging from 0 - 25.

void DCT_DSTC(void)
{
        char inFileName[80];
        char outFileName[80];
        FILE *input;
        BIT_FILE *output;
        unsigned char input_vector[N];
        int output_vector [N];
        int quality, i, c;
        char *CompressionName = "DCT_DST algorithm";
        time_t first, second;
        double time_diff;

        // Obtain and locate the input and output files for compression

        cout << "Enter the input filename.\n";
```

```
cin >> inFileName;
input = fopen(inFileName, "rb");
if (input == NULL)
        fatal_error("Error opening %s for input\n", inFileName);
cout << "Enter the output filename.\n";
cin >> outFileName;
output = OpenOutputBitFile(outFileName);
if (output == NULL)
        fatal_error("Error opening %s for output\n", outFileName);

// Obtain the quality factor

cout << "You are required to enter the quality factor that determines\n";
cout << "the degree of quantization.  Quality values range from 0-25\n";
cout << "Please enter the value you desire.\n";
cin >> quality;
if (quality < 0 || quality > 25)
{
        fatal_error("Illegal quality factor of %d\n", quality);
        quality = 3;
        printf("Using quality factor of %d\n", quality);
}

printf( "\nCompressing %s to %s\n", inFileName, outFileName );
printf( "Using %s.\n", CompressionName);

first = time(NULL);

// Initialize quantization, DCT, and DST matrix values

Initialize(quality);
InitializeSine();

// Output quality information to compression output

OutputBits(output, (unsigned long) quality, 8);

do
{
        // fill input vector with audio file input

        for (i = 0; i < N; i++)
        {
        c = getc(input);
```

```
                    // if file size is not a multiple of N fill remaining elements
                    // with zeros

                    if (c == EOF)
                    {
               for (;i < N; i++)
                                        input_vector[i] = 0;
                              break;
                    }
                    else
                              input_vector[i] = (unsigned char) c;
               }

               // Perform DCT_DST transform and write the data to compression
               // output file

               FowardDCT_DST(input_vector, output_vector);
               WriteTformData(output, output_vector);
               if (c ==EOF)
               WriteEOFCode(output);
          }while(c != EOF);

          // Calculate time for completion of compression algorithm.

          second = time(NULL);
          time_diff = difftime(second, first);
          cout << "\nTime for algorithm completion = " << time_diff << " seconds.\n";

          // Close Input and Output files.  Display Compression Information

          CloseOutputBitFile(output);
          fclose(input);
          print_ratios(inFileName, outFileName);
}

// The expansion routine reads in the compressed data from the compressed file,
// then writes out the decompressed audio file.

void DCT_DSTE(void)
{
          char inFileName[80];
          char outFileName[80];
          BIT_FILE *input;
          FILE *output;
          int input_vector[N];
```

```
unsigned char output_vector[N];
int quality, i, EOFTest;
char *CompressionName = "DCT_DST algorithm";
time_t first, second;
double time_diff;

// Obtain and locate the input and output files for expansion

cout << "Enter the input filename.\n";
cin >> inFileName;
input = OpenInputBitFile(inFileName);
if (input == NULL)
        fatal_error("Error opening %s for input\n", inFileName);
cout << "Enter the output filename.\n";
cin >> outFileName;
output = fopen(outFileName, "wb");
if (output == NULL)
        fatal_error("Error opening %s for output\n", outFileName);
printf("\nExpanding %s to %s\n", inFileName, outFileName);
printf("Using %s.\n", CompressionName);

first = time(NULL);

// Read quality value from the compressed file and initialize quantization
// and DCT matrix values.

quality = (int)InputBits(input, 8);
cout << "\nUsing a quality factor of " << quality << "\n";
Initialize(quality);
InitializeSine();

// Decode compressed input, perform inverse DCT_DST transform,
// and write audio data to expansion output file

do
{
        for (i = 0; i < N; i++)
        {
                EOFTest = InputCode(input);
                if (EOFTest == -12375)
                        break;
                else
                input_vector[i] = EOFTest * Quantum[i];
        }
        if (EOFTest != -12375)
```

```
                {
                InverseDCT_DST(input_vector, output_vector);
                WriteAudioData(output, output_vector);
                }
        }while(EOFTest != -12375);

        // Calculate time for completion of expansion algorithm.

        second = time(NULL);
        time_diff = difftime(second, first);
        cout << "\nTime for algorithm completion = " << time_diff << " seconds.\n";

        // Close Input and Output files.

        CloseInputBitFile(input);
        fclose(output);
        putc('\n', stdout);
}

// Program:         Compand Compression Functions
// Filename:        compandM.cpp
// Description:     This file contains the function definitions that will allow you
//                  to compress/expand raw audio data files using the compand
//                  algorithm.
// Author:          Craig Chin
// Date :           02/13/01
// Notes:           The code is a modification of source code obtained from "The Data
//                  Compression Book", by M. Nelson and J.L. Gailly. This file must
//                  be linked with errhand.cpp, and bitio.cpp for proper functionality.
//                  The function declarations are found in compand.h

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include "bitio.h"
#include "errhand.h"
#include "compand.h"

// This utility routine is used to determine the size of the input file

long get_file_length(FILE *file);
```

```
#ifndef SEEK_END
#define SEEK_END 2
#endif

#ifndef SEEK_SET
#define SEEK_SET 0
#endif

// The compression routine performs a table lookup on each input byte.
// The first part of the routine builds that table.  After that a
// byte is read in and a compressed value is written out.
// The file length is written out at the very start of the compressed
// file, along with the number of bits used to encode the data.

void CompandC(FILE *input, BIT_FILE *output)
{
        int compress[256];
        int steps;
        int bits;
        int value;
        int i;
        int j;
        int c;


        // The first section of code determines the number of bits to use for
        // output codes, and then writes it out to the compressed file.  The
        // length of the input file is also written out.

        cout << "Enter the number of bits to be used for output code.";
        cout << " The default number is 4.\n";
        cin >> bits;
        printf("Compressing using %d bits per sample...\n", bits);
        steps = (1 << ( bits - 1 ) );
        OutputBits(output, (unsigned long) bits, 8);
        OutputBits(output, (unsigned long) get_file_length(input), 32);

        // The compression table is built here.  Each input code maps to
        // a single output code.  There are "steps" codes to be used in
        // the output space.  This builds an exponential output function.

        for (i = steps ; i > 0; i--)
        {
                value = (int)
                        (128.0 * (pow(2.0, (double) i / steps) - 1.0) + 0.5);
                for (j = value ; j > 0 ; j--)
                {
                        compress[j + 127] = i + steps - 1;
                        compress[128 - j] = steps - i;
                }
        }

        // The actual compression takes place here.

        while (( c = getc(input)) != EOF)
        OutputBits(output, (unsigned long) compress[c], bits);
}


// The expansion routine gets the number of bits per code from the
// compressed file, then builds an expansion table.  Each of the
// "steps" codes expands to a unique eight bit code that lies on
// the exponential encoding curve.


void CompandE(BIT_FILE *input, FILE *output)
{
        int steps;
        int bits;
        int value;
        int last_value;
        int i;
        int c;
        long count;
        int expand[256];

        // First this routine reads in the number of bits, then builds the
        // expansion table.  Once the table is built, expanding the file
        // is simply a matter of performing a table lookup on each code.

        bits = (int) InputBits(input, 8);
        printf("Expanding using %d bits per sample...\n", bits);
        steps = (1 << (bits - 1)) ;
        last_value = 0;
        for (i = 1; i <= steps; i++)
        {
                value = (int)
                        (128.0 * (pow(2.0, (double) i / steps) - 1.0) + 0.5);
                expand[steps + i - 1] = 128 + (value + last_value) / 2;
                expand[steps - i]  = 127 - (value + last_value) / 2;
```

```
			last_value = value;
		}

		// The actual file size is stored at the start of the compressed file.
		// It is read in to determine how many codes need to be expanded.  Once
		// that is done, expansion takes place.

		for (count = InputBits(input, 32) ; count > 0 ; count--)
		{
			c = (int) InputBits(input, bits);
			putc(expand[c], output);
		}

}


// This utility routine is used to determine the size of the input file.

long get_file_length(FILE * file)
{
		long marker;
		long eof_ftell;

		marker = ftell(file);
		fseek(file, 0L, SEEK_END);
		eof_ftell = ftell(file);
		fseek(file, marker, SEEK_SET);
		return(eof_ftell - marker);
}

// Program:		Compand Compression Function Declarations
// Filename:		compand.h
// Description:	This file contains the function declarattions that will allow you
//			to compress/expand raw audio data files using the compand
//			algorithm.
// Author:		Craig Chin
// Date :		02/13/01
// Notes:		The code is a modification of source code obtained from "The Data
//			Compression Book", by M. Nelson and J.L. Gailly.  The function
//			definitions are found in compandM.cpp.

void CompandC(FILE *input, BIT_FILE *output);
void CompandE(BIT_FILE *input, FILE *output);

// Program:		Silence Compression Functions
// Filename:        silenceM.cpp
// Description:     This file contains the function definitions that will allow you
//                  to compress/expand raw audio data files using the silence
//                  algorithm.
// Author:          Craig Chin
// Date :           02/13/01
// Notes:           The code is a modification of source code obtained from "The Data
//                  Compression Book", by M. Nelson and J.L. Gailly.  This file must
//                  be linked with errhand.cpp, and bitio.cpp for proper functionality.
//                  The function declarations are found in silence.h

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "bitio.h"
#include "errhand.h"
#include "silence.h"

// These macros define the parameters used to compress the silent
// sequences.  SILENCE_LIMIT is the maximum size of a signal that can
// be considered silent, in terms of offset from the center point.
// START_THRESHOLD gives the number of consecutive silent codes that
// have to be seen before a run is started.  STOP_THRESHOLD tells how
// many non-silent codes need to be seen before a run is considered to
// be over.  SILENCE_CODE is the special code output to the compressed
// file to indicate that a run has been detected.  SILENCE_CODE is always
// followed by a single byte indicating how many consecutive silence
// bytes are to follow.

#define SILENCE_LIMIT   4
#define START_THRESHOLD 5
#define STOP_THRESHOLD  2
#define SILENCE_CODE    0xff
#define IS_SILENCE( c ) ( (c) > ( 0x7f - SILENCE_LIMIT ) && \
                (c) < ( 0x80 + SILENCE_LIMIT ) )

// BUFFER_SIZE is the size of the look ahead buffer.  BUFFER_MASK is
// the mask applied to a buffer index when performing index math.

#define BUFFER_SIZE 8
#define BUFFER_MASK 7

// Local function prototypes.
```

```
int silence_run(int buffer[], int index);
int end_of_silence(int buffer[], int index);


// The compression routine has to detect when a silence run has started,
// and when it is over.  It does this by keeping up and coming bytes in
// a look ahead buffer.  The buffer along with the current index is passed
// ahead to routines that check to see if a run has started, or if it has
// ended.


void SilenceC(FILE *input, BIT_FILE *output)
{
        int look_ahead[BUFFER_SIZE];
        int index;
        int i;
        int run_length;

        for (i = 0 ; i < BUFFER_SIZE ; i++)
                look_ahead[i] = getc(input);
        index = 0;
        for ( ; ; )
        {
                if (look_ahead[index] == EOF)
                        break;

        // If a run has started the program will sit in a do loop until
        // the run is complete, loading new characters all the while.

                if (silence_run(look_ahead, index))
                {
                        run_length = 0;
                        do
                        {
                                look_ahead[index++] = getc(input);
                                index &= BUFFER_MASK;
                                if (++run_length == 255)
                                {
                                        putc( SILENCE_CODE, output->file );
                                        putc( 255, output->file );
                                        run_length = 0;
                                }
                        } while (!end_of_silence(look_ahead, index));
                        if (run_length > 0)
                        {
```

```
                                        putc(SILENCE_CODE, output->file);
                                        putc(run_length, output->file);
                                }
                        }

                // When a run of silence is over, some plain codes must be output.
                // Any code that accidentally matches the silence code gets silently changed.

                        if (look_ahead[index] == SILENCE_CODE)
                                look_ahead[index]--;
                        putc(look_ahead[index], output->file);
                        look_ahead[index++] = getc(input);
                        index &= BUFFER_MASK;
                }

        }

// The expansion routine has to check for the run code, and when it finds it
// pad out the output file with some silence bytes.

void SilenceE(BIT_FILE *input, FILE *output)
{
        int c;
        int run_count;

        while ((c = getc( input->file)) != EOF)
        {
                if (c == SILENCE_CODE)
                {
                        run_count = getc(input->file);
                        while (run_count-- > 0)
                                putc(0x80, output);
                }
                else
                        putc(c, output);
        }
}

// This support routine checks to see if the look ahead buffer contains
// the start of a run, which by definition is START_THRESHOLD consecutive
// silence characters.


int silence_run(int buffer[], int index)
{
        int i;

        for (i = 0 ; i < START_THRESHOLD ; i++)
                if (!IS_SILENCE(buffer[(index + i) & BUFFER_MASK]))
                        return(0);
        return(1);
}

// This support routine is called during the middle of a run of
// silence.  It checks to see if audio input has reached the end of the
// run.  By definition this occurs when we see STOP_THRESHOLD consecutive
// non-silence characters.

int end_of_silence(int buffer[], int index)
{
        int i;

        for (i = 0 ; i < STOP_THRESHOLD ; i++)
                if (IS_SILENCE(buffer[(index + i) & BUFFER_MASK]))
                        return(0);
        return(1);
}

// Program:          Silence Compression Function Declarations
// Filename:         silence.h
// Description:      This file contains the function declarations that will allow you
//                   to compress/expand raw audio data files using the silence
//                   algorithm.
// Author:           Craig Chin
// Date :            02/13/01
// Notes:            The code is a modification of source code obtained from "The Data
//                   Compression Book", by M. Nelson and J.L. Gailly.  The function
//                   definitions are found in silenceM.cpp

void SilenceC(FILE *input, BIT_FILE *output);
void SilenceE(BIT_FILE *input, FILE *output);

// Program:          Huffman Compression/Expansion Functions
// Filename:         huffm2.cpp
// Description:      This file provides compression/expansion functions that
//                   utilize the Huffman entropy coding algorithm as the means
//                   of compression.
// Author:           Craig Chin
// Date :            02/15/01
// Notes:            The code is a modification of the source code HUFF.C obtained
//                              from "The Data Compression Book", by M. Nelson and J.L.
//                              Gailly. This file must be linked to the files errhand.cpp, bitio.cpp,
//                              and utility_func.cpp for proper functionality.

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>
#include "bitio.h"
#include "errhand.h"
#include "utility_func.h"

// The NODE class is a node in the Huffman decoding tree.  It has a
// count, which is its weight in the tree, and the node numbers of its
// two children.  The saved_count member of the structure is only
// there for debugging purposes, and can be safely taken out at any
// time.  It just holds the intial count for each of the symbols, since
// the count member is continually being modified as the tree grows.

typedef class tree_node
{
        public:
        unsigned int count;
        unsigned int saved_count;
        int child_0;
        int child_1;
} NODE;

// A Huffman tree is set up for decoding, not encoding.  During the
// encoding process the tree is traversed and a table of codes is built
// for the symbols.  The codes are stored in this CODE class.

typedef class Code
{
        public:
        unsigned int code;
        int code_bits;
} CODE;

// The special EOS symbol is 256, the first available symbol after all
// of the possible bytes.  When decoding, reading this symbol indicates
// that all of the data has been read in.
```

```
#define END_OF_STREAM 256

// Local function prototypes.

void count_bytes(FILE *input, unsigned long *long_counts);
void scale_counts(unsigned long *long_counts, NODE *nodes);
int build_tree(NODE *nodes);
void convert_tree_to_code(NODE *nodes,
                    CODE *codes,
                    unsigned int code_so_far,
                    int bits,
                    int node);
void output_counts(BIT_FILE *output, NODE *nodes);
void input_counts(BIT_FILE *input, NODE *nodes);
void print_model(NODE *nodes, CODE *codes);
void compress_data(FILE *input, BIT_FILE *output, CODE *codes);
void expand_data(BIT_FILE *input, FILE *output, NODE *nodes,
          int root_node);
void print_char(int c);

// The HuffC compression routine requests that the user specify the
// input and ouput filenames for the compression process, as well as,
// whether or not he desires to see the tree-model of the
// encoding/decoding  processes.  With this information the routine
// works in a fairly straightforward manner.  First, it has to
// allocate storage for three different arrays of data. Next, it
// counts all the bytes in the input file.  The counts are all stored
// in long int, so the next step is scale them down to single byte
// counts in the NODE array.  After the counts are scaled, the
// Huffman decoding tree is built on top of the NODE array.  Another
// routine walks through the tree to build a table of codes, one per
// symbol.  Finally, when the codes are all ready, compressing the
// file is a simple matter.  After the file is compressed,
// the storage is freed up, and the routine returns.

void HuffC(void)
{
        char inFileName[80];
        char outFileName[80];
        FILE *input;
        BIT_FILE *output;
        unsigned long *counts;
        NODE *nodes;
        CODE *codes;
        int root_node;
```

```
char model[10];
char *CompressionName = "static order 0 model with Huffman coding";
time_t first, second;
double time_diff;

// Obtain and locate the input and output files for compression

cout << "Enter the input filename.\n";
cin >> inFileName;
input = fopen(inFileName, "rb");
if (input == NULL)
        fatal_error("Error opening %s for input\n", inFileName);
cout << "Enter the output filename.\n";
cin >> outFileName;
output = OpenOutputBitFile(outFileName);
if (output == NULL)
        fatal_error("Error opening %s for output\n", outFileName);

// Ascertain whether or no you desire to see the tree model.

cout << "If you desire to see the tree model enter 'model'.\n";
cin >> model;
printf("\nCompressing %s to %s\n", inFileName, outFileName);
printf("Using %s.\n", CompressionName);

first = time(NULL);
counts = (unsigned long *) calloc(256, sizeof(unsigned long));
if (counts == NULL)
        fatal_error("Error allocating counts array\n");
if ((nodes = (NODE *) calloc(514, sizeof(NODE))) == NULL)
        fatal_error("Error allocating nodes array\n");
if ((codes = (CODE *) calloc(257, sizeof(CODE))) == NULL)
        fatal_error("Error allocating codes array\n");
count_bytes(input, counts);
scale_counts(counts, nodes);
output_counts(output, nodes);
root_node = build_tree(nodes);
convert_tree_to_code(nodes, codes, 0, 0, root_node);
if (strcmp( model, "model") == 0)
        print_model(nodes, codes);
compress_data(input, output, codes);
free((char *) counts);
free((char *) nodes);
free((char *) codes);
```

```
        // Calculate time for completion of compression algorithm.

        second = time(NULL);
        time_diff = difftime(second, first);
        cout << "\nTime for algorithm completion = " << time_diff << " seconds.\n";

        // Close Input and Output files.  Display Compression Information

        CloseOutputBitFile(output);
        fclose(input);
        print_ratios(inFileName, outFileName);
}

// PipeHuffC() is a compression function similar to HuffC(), but specially
// adpated for use as a back-end lossless compression technique that is to
// be preceded by a lossy front-end compression algorithm.

void PipeHuffC(char inFileName[], char outFileName[])
{
        FILE *input;
        BIT_FILE *output;
        unsigned long *counts;
        NODE *nodes;
        CODE *codes;
        int root_node;
        char model[10];

        // Open the input and output files for compression.

        input = fopen(inFileName, "rb");
        if (input == NULL)
                fatal_error("Error opening %s for input\n", inFileName);
        output = OpenOutputBitFile(outFileName);
        if (output == NULL)
                fatal_error("Error opening %s for output\n", outFileName);

        // Ascertain whether or no you desire to see the tree model

        cout << "If you desire to see the tree model enter 'model'.\n";
        cin >> model;

        counts = (unsigned long *) calloc(256, sizeof(unsigned long));
        if (counts == NULL)
                fatal_error("Error allocating counts array\n");
        if ((nodes = (NODE *) calloc( 514, sizeof(NODE))) == NULL)
```

```
                fatal_error("Error allocating nodes array\n");
        if ((codes = (CODE *) calloc( 257, sizeof(CODE))) == NULL)
                fatal_error("Error allocating codes array\n");
        count_bytes(input, counts);
        scale_counts(counts, nodes);
        output_counts(output, nodes);
        root_node = build_tree(nodes);
        convert_tree_to_code(nodes, codes, 0, 0, root_node);
        if (strcmp( model, "model") == 0)
                print_model(nodes, codes);
        compress_data(input, output, codes);
        free((char *) counts);
        free((char *) nodes);
        free((char *) codes);

        // Close Input and Output files.  Display Compression Information

        CloseOutputBitFile(output);
        fclose(input);
        print_ratios(inFileName, outFileName);
}


// HuffE() is the routine called to expand a file that has been
// compressed with order 0 Huffman coding, that is, HuffC().  All the
// routine has to do is read in the counts that have been stored in
// the compressed file, then build the Huffman tree.  The data can
// then be expanded by reading in a bit at a time from the compressed
// file.  Finally, the node array is freed and the routine returns.

void HuffE(void)
{
        char inFileName[80];
        char outFileName[80];
        BIT_FILE *input;
        FILE *output;
        NODE *nodes;
        int root_node;
        char model[10];
        char *CompressionName = "static order 0 model with Huffman coding";
        time_t first, second;
        double time_diff;

        // Obtain and locate the input and output files for expansion
```

```
	cout << "Enter the input filename.\n";
	cin >> inFileName;
	input = OpenInputBitFile(inFileName);
	if (input == NULL)
		fatal_error("Error opening %s for input\n", inFileName);
	cout << "Enter the output filename.\n";
	cin >> outFileName;
	output = fopen(outFileName, "wb");
	if (output == NULL)
		fatal_error("Error opening %s for output\n", outFileName);

	// Ascertain whether or no you desire to see the tree model

	cout << "If you desire to see the tree model enter 'model'.\n";
	cin >> model;
	printf("\nExpanding %s to %s\n", inFileName, outFileName);
	printf("Using %s.\n", CompressionName);

	first = time(NULL);
	if ((nodes = (NODE *) calloc(514, sizeof(NODE))) == NULL)
		fatal_error("Error allocating nodes array\n");
	input_counts(input, nodes);
	root_node = build_tree(nodes);
	if (strcmp(model, "model") == 0)
		print_model(nodes, 0);
	expand_data(input, output, nodes, root_node);
	free((char *) nodes);

	// Calculate time for completion of expansion algorithm

	second = time(NULL);
	time_diff = difftime(second, first);
	cout << "\nTime for algorithm completion = " << time_diff << " seconds.\n";

	// Close Input and Output files.

	CloseInputBitFile(input);
	fclose(output);
	putc('\n', stdout);
}

// PipeHuffE() is an expansion function similar to HuffE(), but specially
// adpated for use as a front-end lossless compression technique that is to
// precede a lossy back-end compression algorithm.
```

```
void PipeHuffE(char inFileName[], char outFileName[])
{
        BIT_FILE *input;
        FILE *output;
        NODE *nodes;
        int root_node;
        char model[10];


        // Open the input and output files for expansion

        input = OpenInputBitFile(inFileName);
        if (input == NULL)
                fatal_error("Error opening %s for input\n", inFileName);
        output = fopen(outFileName, "wb");
        if (output == NULL)
                fatal_error("Error opening %s for output\n", outFileName);

        // Ascertain whether or no you desire to see the tree model

        cout << "If you desire to see the tree model enter 'model'.\n";
        cin >> model;

        if ((nodes = (NODE *) calloc(514, sizeof(NODE))) == NULL)
                fatal_error("Error allocating nodes array\n");
        input_counts(input, nodes);
        root_node = build_tree(nodes);
        if (strcmp(model, "model") == 0)
        print_model(nodes, 0);
        expand_data(input, output, nodes, root_node);
        free((char *) nodes);

        // Close Input and Output files.

        CloseInputBitFile(input);
        fclose(output);
        putc('\n', stdout);
}

// In order for the compressor to build the same model, the symbol counts
// have to be stored in the compressed file so the expander can read them
// in.  In order to save space, all 256 symbols are not saved
// unconditionally.  The format used to store counts looks like this:
//
// start, stop, counts, start, stop, counts, ... 0
```

```
//
// This means that runs of counts are stored, until all the non-zero
// counts have been stored.  At this time the list is terminated by
// storing a start value of 0.  Note that at least 1 run of counts has
// to be stored, so even if the first start value is 0, it is read in.
// It also means that even in an empty file that has no counts, at least
// one count has to be passed.
//
// In order to efficiently use this format, the runs of non-zero counts
// have to be identified.  It is not desirable to stop a run because of
// just one or two zeros in the count stream.  So the program sits in a
// loop looking for strings of three or more zero values in a row.

void output_counts(BIT_FILE *output, NODE *nodes)
{
  int first;
  int last;
  int next;
  int i;

  first = 0;
  while (first < 255 && nodes[first].count == 0)
        first++;

      // At the start of each loop, it is assumed that first is the
      // number for a run of non-zero values.  The rest of the loop is
      // concerned with finding the value for last, which is the end of the
      // run, and the value of next, which is the start of the next run.
      // At the end of the loop, next is  assigned to first, so it starts
      // there on the next run.

  for (; first < 256 ; first = next)
      {
            last = first + 1;
            for (;;)
            {
                  for (; last < 256 ; last++)
                        if (nodes[last].count == 0)
                              break;
                  last--;
                  for (next = last + 1; next < 256 ; next++)
                        if (nodes[next].count != 0)
                        break;
                  if (next > 255)
                        break;
```

```
                    if ((next - last) > 3)
                            break;
                    last = next;
            }

            // Here is where first, last, and all the counts in between are output

            if (putc(first, output->file) != first)
                    fatal_error("Error writing byte counts\n");
            if (putc(last, output->file) != last)
                    fatal_error("Error writing byte counts\n");
            for (i = first ; i <= last ; i++)
            {
                    if (putc(nodes[i].count, output->file) !=
         (int) nodes[i].count)
                            fatal_error("Error writing byte counts\n" );
            }
   }
   if (putc( 0, output->file ) != 0)
           fatal_error("Error writing byte counts\n");
}

// When expanding, the same set of counts have to be read in.  Initially first,
// is read in, then the function checks to see if its done, and if not, read in
// last and a string of counts.

void input_counts(BIT_FILE *input, NODE *nodes)
{
        int first;
        int last;
        int i;
        int c;

        for (i = 0 ; i < 256 ; i++)
                nodes[i].count = 0;
        if ((first = getc(input->file)) == EOF)
                fatal_error("Error reading byte counts\n");
        if ((last = getc( input->file)) == EOF)
                fatal_error("Error reading byte counts\n");
        for (;;)
        {
                for (i = first; i <= last; i++)
                        if ((c = getc(input->file)) == EOF)
                                fatal_error( "Error reading byte counts\n" );
                        else
                        nodes[i].count = (unsigned int) c;
                if ((first = getc(input->file)) == EOF)
                        fatal_error("Error reading byte counts\n");
                if (first == 0)
                        break;
                if ((last = getc(input->file)) == EOF)
                        fatal_error("Error reading byte counts\n");
        }
        nodes[END_OF_STREAM].count = 1;
}

// This routine counts the frequency of occurence of every byte in
// the input file.  It marks the place in the input stream where it
// started, counts up all the bytes, then returns to the place where
// it started.  In most C implementations, the length of a file
// cannot exceed an unsigned long, so this routine should always
// work.

#ifndef SEEK_SET
#define SEEK_SET 0
#endif

void count_bytes(FILE *input, unsigned long *counts)
{
        long input_marker;
        int c;

        input_marker = ftell(input);
        while ((c = getc(input)) != EOF )
        counts[c]++;
        fseek(input, input_marker, SEEK_SET);
}


// In order to limit the size of the Huffman codes to 16 bits, the counts
// are scaled down so they fit in an unsigned char, and then they are all
// stored as initial weights in the NODE array.  The only thing to be
// careful of is to make sure that a node with a non-zero count doesn't
// get scaled down to 0.  Nodes with values of 0 don't get codes.

void scale_counts(unsigned long *counts, NODE *nodes)
{
        unsigned long max_count;
        int i;
        max_count = 0;
        for (i = 0 ; i < 256 ; i++)
        if (counts[ i ] > max_count)
                max_count = counts[i];
        if (max_count == 0)
        {
                counts[0] = 1;
                max_count = 1;
        }
        max_count = max_count / 255;
        max_count = max_count + 1;
        for (i = 0; i < 256; i++)
        {
                nodes[i].count = (unsigned int) (counts[i] / max_count);
                if (nodes[i].count == 0 && counts[i] != 0)
                        nodes[ i ].count = 1;
        }
        nodes[ END_OF_STREAM ].count = 1;
}

// Building the Huffman tree is fairly simple.  All of the active nodes
// are scanned in order to locate the two nodes with the minimum
// weights.  These two weights are added together and assigned to a new
// node.  The new node makes the two minimum nodes into its 0 child
// and 1 child.  The two minimum nodes are then marked as inactive.
// This process repeats until their is only one node left, which is the
// root node.  The tree is done, and the root node is passed back
// to the calling routine.
//
// Node 513 is used here to arbitratily provide a node with a guaranteed
// maximum value.  It starts off being min_1 and min_2.  After all active
// nodes have been scanned, the function can tell if there is only one
// active node left by checking to see if min_1 is still 513.

int build_tree(NODE *nodes)
{
        int next_free;
        int i;
        int min_1;
        int min_2;

        nodes[513].count = 0xffff;
        for (next_free = END_OF_STREAM + 1; ; next_free++)
        {
                min_1 = 513;
                min_2 = 513;
                for (i = 0; i < next_free; i++)
                        if (nodes[i].count != 0)
                        {
                                if (nodes[i].count < nodes[min_1].count)
                                {
                                        min_2 = min_1;
                                        min_1 = i;
                                }
                                else
                                        if (nodes[i].count < nodes[min_2].count)
                                                min_2 = i;
                        }
                if (min_2 == 513)
                        break;
                nodes[next_free].count = nodes[min_1].count
                          + nodes[min_2].count;
                 nodes[min_1].saved_count = nodes[min_1].count;
                nodes[min_1].count = 0;
                nodes[min_2].saved_count =  nodes[ min_2 ].count;
                nodes[min_2].count = 0;
                nodes[next_free].child_0 = min_1;
                nodes[next_free].child_1 = min_2;
        }
        next_free--;
        nodes[next_free].saved_count = nodes[next_free].count;
        return(next_free);
}


// Since the Huffman tree is built as a decoding tree, there is
// no simple way to get the encoding values for each symbol out of
// it.  This routine recursively walks through the tree, adding the
// child bits to each code until it gets to a leaf.  When it gets
// to a leaf, it stores the code value in the CODE element, and
// returns.

void convert_tree_to_code(NODE *nodes, CODE *codes, unsigned int code_so_far,
int bits, int node)
{
        if (node <= END_OF_STREAM)
        {
                codes[node].code = code_so_far;
                codes[node].code_bits = bits;
                return;
        }
        code_so_far <<= 1;
        bits++;
        convert_tree_to_code(nodes, codes, code_so_far, bits,
                                                    nodes[node].child_0);
        convert_tree_to_code(nodes, codes, code_so_far | 1,
                bits, nodes[node].child_1);
}

// If a tree model is to be displayed, this routine is called to print
// out some of the model information after the tree is built.  Note that
// this is the only place that the saved_count NODE element is used for
// anything at all, and in this case it is just for diagnostic
// information.  By the time this function is called the tree has already
// been built and every active element will have 0 in its count.

void print_model(NODE *nodes, CODE *codes)
{
        int i;

        for (i = 0 ; i < 513 ; i++)
        {
                if (nodes[i].saved_count != 0)
                {
                        printf("node=");
                        print_char(i);
                        printf("  count=%3d", nodes[i].saved_count);
                        printf("  child_0=");
                        print_char( nodes[i].child_0);
                        printf("  child_1=");
                        print_char(nodes[i].child_1);
                        if (codes && i <= END_OF_STREAM)
                        {
                                printf("  Huffman code=");
                                FilePrintBinary(stdout, codes[i].code, codes[i].code_bits);
                        }
                        printf( "\n" );
                }
        }
}

// The print_model routine uses this function to print out node numbers.
// The catch is, if it is a printable character, it gets printed out
// as a character.  This makes the debug output a little easier to read.
```

```
void print_char(int c)
{
        if ( c >= 0x20 && c < 127 )
                printf( "'%c'", c );
        else
                printf( "%3d", c );
}

// Once the tree gets built, and the CODE table is built, compressing
// the data is a breeze.  Each byte is read in, and its corresponding
// Huffman code is sent out.

void compress_data(FILE *input, BIT_FILE *output, CODE *codes)
{
        int c;

        while ((c = getc(input)) != EOF)
                OutputBits(output, (unsigned long) codes[c].code,
                        codes[c].code_bits);
        OutputBits(output, (unsigned long) codes[END_OF_STREAM].code,
                codes[END_OF_STREAM].code_bits);
}

// Expanding compressed data is a little harder than the compression
// phase.  As each new symbol is decoded, the tree is traversed,
// starting at the root node, reading a bit in, and taking either the
// child_0 or child_1 path.  Eventually, the tree winds down to a
// leaf node, and the corresponding symbol is output.  If the symbol
// is the END_OF_STREAM symbol, it doesn't get written out, and
// instead the whole process terminates.

void expand_data(BIT_FILE *input, FILE *output, NODE *nodes, int root_node)
{
        int node;

        for (;;)
        {
                node = root_node;
                do
                {
                        if (InputBit(input))
                                node = nodes[node].child_1;
                        else
                                node = nodes[node].child_0;
                }while (node > END_OF_STREAM);
                if (node == END_OF_STREAM)
                        break;
                if ((putc(node, output)) != node)
                        fatal_error("Error trying to write expanded byte to output");
        }
}

// Program:             Arithmetic Compression/Expansion Functions
// Filename:            arith.cpp
// Description:         This file provides compression/expansion functions that
//                      utilize the Arithmetic entropy coding algorithm as the means
//                      of compression.
// Author:              Craig Chin
// Date :               02/15/01
// Notes:               The code is a modification of the source code ARITH.C obtained
//                      from "The Data Compression Book", by M. Nelson and J.L.
//                      Gailly.  This file must be linked to the files errhand.cpp, bitio.cpp,
//                      and utility_func.cpp for proper functionality.

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "errhand.h"
#include "bitio.h"
#include "utility_func.h"

// The SYMBOL structure is what is used to define a symbol in
// arithmetic coding terms.  A symbol is defined as a range between
// 0 and 1.  Since we are using integer math, instead of using 0 and 1
// as our end points, we have an integer scale.  The low_count and
// high_count define where the symbol falls in the range.


typedef struct
{
        unsigned short int low_count;
        unsigned short int high_count;
        unsigned short int scale;
} SYMBOL;

// Internal function prototypes.

void build_model(FILE *input, FILE *output);
void scale_counts(unsigned long counts[], unsigned char scaled_counts[]);
void build_totals(unsigned char scaled_counts[]);
void arith_count_bytes(FILE *input, unsigned long counts[]);
void output_counts(FILE *output, unsigned char scaled_counts[]);
void input_counts(FILE *stream);
void convert_int_to_symbol(int symbol, SYMBOL *s);
void get_symbol_scale(SYMBOL *s);
int convert_symbol_to_int(int count, SYMBOL *s);
void initialize_arithmetic_encoder(void);
void encode_symbol(BIT_FILE *stream, SYMBOL *s);
void flush_arithmetic_encoder(BIT_FILE *stream);
short int get_current_count(SYMBOL *s);
void initialize_arithmetic_decoder(BIT_FILE *stream);
void remove_symbol_from_stream(BIT_FILE *stream, SYMBOL *s);

#define END_OF_STREAM 256
short int totals[258];          // The cumulative totals

// This ArithC() routine first gathers statistics, and initializes the
// arithmetic encoder.  It then encodes all the characters in the file,
// followed by the EOF character.  The output stream is then flushed,
// and then the function is exited.  Note that an extra two bytes are
// output.  When decoding an arithmetic stream, we have to read in extra
// bits.  The decoding process takes place in the msb of the low and
// high range ints, so when decoding the last bit there still has to be
// at least 15 junk bits loaded into the registers.  The extra two bytes
// account for that.

void ArithC(void)
{
        char inFileName[80];
        char outFileName[80];
        FILE *input;
        BIT_FILE *output;
        char *CompressionName = "Fixed order 0 model with arithmetic coding";
        int c;
        SYMBOL s;
        time_t first, second;
        double time_diff;

        // Obtain and locate the input and output files for compression

        cout << "Enter the input filename.\n";
        cin >> inFileName;
        input = fopen(inFileName, "rb");
        if (input == NULL)
```

```
                fatal_error("Error opening %s for input\n", inFileName);
        cout << "Enter the output filename.\n";
        cin >> outFileName;
        output = OpenOutputBitFile(outFileName);
        if (output == NULL)
                fatal_error("Error opening %s for output\n", outFileName);
        printf("\nCompressing %s to %s\n", inFileName, outFileName);
        printf("Using %s.\n", CompressionName);

        first = time(NULL);
        build_model(input, output->file);
        initialize_arithmetic_encoder();
        while ((c = getc(input)) != EOF)
        {
                convert_int_to_symbol(c, &s);
                encode_symbol(output, &s);
        }
        convert_int_to_symbol(END_OF_STREAM, &s);
        encode_symbol(output, &s);
        flush_arithmetic_encoder(output);
        OutputBits(output, 0L, 16);

        // Calculate time for completion of compression algorithm.

        second = time(NULL);
        time_diff = difftime(second, first);
        cout << "\nTime for algorithm completion = " << time_diff << " seconds.\n";

        // Close Input and Output files.  Display Compression Information

        CloseOutputBitFile(output);
        fclose(input);
        print_ratios(inFileName, outFileName);
}

// The ArithE() routine reads in the model, initializes the decoder,
// then sits in a loop reading in characters.  When an END_OF_STREAM
// is decoded, it signals the function that files can be closed and
// the function exits.  Note decoding a single character is a three
// step process:  first it is determined what the scale is for the
// current symbol by looking at the difference between the high and
// low values. Next it is seen where the current input values fall in
// that range.  Finally, the totals array is searched to find out what
// symbol is a match.  After that is done, the symbol is removed from
// the decoder.
```

```
void ArithE(void)
{
        char inFileName[80];
        char outFileName[80];
        BIT_FILE *input;
        FILE *output;
        char *CompressionName = "Fixed order 0 model with arithmetic coding";
        SYMBOL s;
        int c;
        int count;
        time_t first, second;
        double time_diff;

        // Obtain and locate the input and output files for expansion

        cout << "Enter the input filename.\n";
        cin >> inFileName;
        input = OpenInputBitFile(inFileName);
        if (input == NULL)
                fatal_error("Error opening %s for input\n", inFileName);
        cout << "Enter the output filename.\n";
        cin >> outFileName;
        output = fopen( outFileName, "wb" );
        if ( output == NULL )
                fatal_error( "Error opening %s for output\n", outFileName);
        printf("\nExpanding %s to %s\n", inFileName, outFileName);
        printf("Using %s.\n", CompressionName);

        first = time(NULL);
        input_counts(input->file);
        initialize_arithmetic_decoder(input);
        for (;;)
        {
                get_symbol_scale(&s);
                count = get_current_count(&s);
                c = convert_symbol_to_int(count, &s);
                if (c == END_OF_STREAM)
                break;
                remove_symbol_from_stream(input, &s);
                putc((char) c, output);
        }

        // Calculate time for completion of expansion algorithm.
        second = time(NULL);
        time_diff = difftime(second, first);
        cout << "\nTime for algorithm completion = " << time_diff << " seconds.\n";

        // Close Input and Output files.

        CloseInputBitFile(input);
        fclose(output);
        putc('\n', stdout);
}

// This is the routine that is called to scan the input file, scale
// the counts, build the totals array, then output the scaled counts
// to the output file.

void build_model(FILE *input, FILE *output)
{
        unsigned long counts[256];
        unsigned char scaled_counts[256];
        arith_count_bytes(input, counts);
        scale_counts(counts, scaled_counts);
        output_counts(output, scaled_counts);
        build_totals(scaled_counts);
}

// This routine runs through the file and counts the appearances of each
// character.

#ifndef SEEK_SET
#define SEEK_SET 0
#endif

void arith_count_bytes(FILE *input, unsigned long counts[])
{
        long input_marker;
        int i;
        int c;

        for (i = 0; i < 256; i++)
                counts[i] = 0;
        input_marker = ftell(input);
        while ((c = getc(input)) != EOF)
                counts[c]++;
        fseek(input, input_marker, SEEK_SET);
}
```

```
// This routine is called to scale the counts down.  There are two types
// of scaling that must be done.  First, the counts need to be scaled
// down so that the individual counts fit into a single unsigned char.
// Then, the counts need to be rescaled so that the total of all counts
// is less than 16384.

void scale_counts(unsigned long counts[], unsigned char scaled_counts[])
{
        int i;
        unsigned long max_count;
        unsigned int total;
        unsigned long scale;

        // The first section of code makes sure each count fits into a single byte.

        max_count = 0;
        for (i = 0; i < 256; i++)
                if (counts[i] > max_count)
                        max_count = counts[i];
        scale = max_count / 256;
        scale = scale + 1;
        for (i = 0; i < 256; i++)
        {
                scaled_counts[i] = (unsigned char) (counts[i] / scale);
                if (scaled_counts[i] == 0 && counts[i] != 0)
                        scaled_counts[i] = 1;
        }

        // This next section makes sure the total is less than 16384.  The total is
        // initialized to 1 instead of 0 because there will be an additional 1 added
        // in for the END_OF_STREAM symbol;

        total = 1;
        for (i = 0; i < 256; i++)
                total += scaled_counts[i];
        if (total > (32767 - 256))
                scale = 4;
        else
                if (total > 16383)
                        scale = 2;
                else
                        return;
        for (i = 0; i < 256; i++)
                scaled_counts[i] /= scale;
```

```
}

// This routine is used by both the encoder and decoder to build the
// table of cumulative totals.  The counts for the characters in the
// file are in the counts array, and we know that there will be a single
// instance of the EOF symbol.

void build_totals(unsigned char scaled_counts[])
{
        int i;

        totals[0] = 0;
        for (i = 0; i < END_OF_STREAM; i++)
                totals[i + 1] = totals[i] + scaled_counts[i];
        totals[END_OF_STREAM + 1] = totals[END_OF_STREAM] + 1;
}

// In order for the compressor to build the same model, the symbol counts
// have to be stored in the compressed file so the expander can read them
// in.  In order to save space, all 256 symbols are not saved
// unconditionally.  The format used to store counts looks like this:
//
// start, stop, counts, start, stop, counts, ... 0
//
// This means that runs of counts are stored, until all the non-zero
// counts have been stored.  At this time the list is terminated by
// storing a start value of 0.  Note that at least 1 run of counts has
// to be stored, so even if the first start value is 0, it is read in.
// It also means that even in an empty file that has no counts, at least
// one count has to be passed.
//
// In order to efficiently use this format, the runs of non-zero counts
// have to be identified.  It is not desirable to stop a run because of
// just one or two zeros in the count stream.  So the program sits in a
// loop looking for strings of three or more zero values in a row.

void output_counts(FILE *output, unsigned char scaled_counts[])
{
        int first;
        int last;
        int next;
        int i;

        first = 0;
        while ( first < 255 && scaled_counts[ first ] == 0 )
```

```
            first++;

        // At the start of each loop, it is assumed that first is the
        // number for a run of non-zero values.  The rest of the loop is
        // concerned with finding the value for last, which is the end of the
        // run, and the value of next, which is the start of the next run.
        // At the end of the loop, next is  assigned to first, so it starts
        // there on the next run.

        for ( ; first < 256; first = next)
        {
            last = first + 1;
            for (;;)
            {
                for (; last < 256; last++)
            if (scaled_counts[last] == 0)
                    break;
            last--;
            for (next = last + 1; next < 256; next++)
            if (scaled_counts[next] != 0)
                    break;
            if (next > 255)
                    break;
            if ((next - last) > 3)
                    break;
            last = next;
            }

            // Here is where first, last, and all the counts in between are output.

            if (putc(first, output) != first)
                fatal_error("Error writing byte counts\n");
            if (putc(last, output) != last)
                fatal_error("Error writing byte counts\n");
            for (i = first; i <= last; i++)
            {
                if (putc(scaled_counts[i], output) != (int) scaled_counts[i])
                    fatal_error("Error writing byte counts\n");
            }
        }
        if (putc(0, output) != 0)
            fatal_error("Error writing byte counts\n");
}

// When expanding, the same set of counts have to be read in.  Initially first,
// is read in, then the function checks to see if its done, and if not, read in
// last and a string of counts.

void input_counts(FILE *input)
{
        int first;
        int last;
        int i;
        int c;
        unsigned char scaled_counts[256];

        for (i = 0; i < 256; i++)
                scaled_counts[i] = 0;
        if ((first = getc(input)) == EOF)
                fatal_error("Error reading byte counts\n");
        if ((last = getc(input)) == EOF)
                fatal_error("Error reading byte counts\n");
        for (;;)
        {
                for (i = first; i <= last; i++)
                if ((c = getc(input)) == EOF)
                                fatal_error("Error reading byte counts\n");
                else
                        scaled_counts[i] = (unsigned char) c;
                if ((first = getc(input)) == EOF)
                        fatal_error("Error reading byte counts\n");
                if (first == 0)
                        break;
                if ((last = getc(input)) == EOF)
                        fatal_error("Error reading byte counts\n");
        }
        build_totals(scaled_counts);
}

// Everything from here down defines the arithmetic coder section
// of the program.

// These four variables define the current state of the arithmetic
// coder/decoder.  They are assumed to be 16 bits long.  Note that
// by declaring them as short ints, they will actually be 16 bits
// on most 80X86 and 680X0 machines, as well as VAXen.

static unsigned short int code;  // The present input code value
static unsigned short int low;   // Start of the current code range
static unsigned short int high;  // End of the current code range
```

```
long underflow_bits;          // Number of underflow bits pending

// This routine must be called to initialize the encoding process.
// The high register is initialized to all 1s, and it is assumed that
// it has an infinite string of 1s to be shifted into the lower bit
// positions when needed.

void initialize_arithmetic_encoder()
{
        low = 0;
        high = 0xffff;
        underflow_bits = 0;
}

// At the end of the encoding process, there are still significant
// bits left in the high and low registers.  Two bits are output,
// plus as many underflow bits as are necessary.

void flush_arithmetic_encoder(BIT_FILE *stream)
{
        OutputBit(stream, low & 0x4000);
        underflow_bits++;
        while (underflow_bits-- > 0)
                OutputBit(stream, ~low & 0x4000);
}

// Finding the low count, high count, and scale for a symbol
// is really easy, because of the way the totals are stored.
// This is the one redeeming feature of the data structure used
// in this implementation.

void convert_int_to_symbol(int c, SYMBOL *s)
{
        s->scale = totals[END_OF_STREAM + 1];
        s->low_count = totals[c];
        s->high_count = totals[c + 1];
}

// Getting the scale for the current context is easy.

void get_symbol_scale(SYMBOL *s)
{
        s->scale = totals[END_OF_STREAM + 1];
}
```

```
// During decompression, the table has to be searched until
// the symbol that straddles the "count" parameteris found.  When
// it is found, it is returned. The reason for also setting the
// high count and low count is so that symbol can be properly removed
// from the arithmetic coded input.

int convert_symbol_to_int(int count, SYMBOL *s)
{
        int c;

        for (c = END_OF_STREAM; count < totals[c]; c--)
                ;
        s->high_count = totals[c + 1];
        s->low_count = totals[c];
        return(c);
}

// This routine is called to encode a symbol.  The symbol is passed
// in the SYMBOL structure as a low count, a high count, and a range,
// instead of the more conventional probability ranges.  The encoding
// process takes two steps.  First, the values of high and low are
// updated to take into account the range restriction created by the
// new symbol.  Then, as many bits as possible are shifted out to
// the output stream.  Finally, high and low are stable again and
// the routine returns.

void encode_symbol(BIT_FILE *stream, SYMBOL *s)
{
        long range;

        // These three lines rescale high and low for the new symbol.

        range = (long) (high-low) + 1;
        high = low + (unsigned short int) ((range * s->high_count) / s->scale - 1);
        low = low + (unsigned short int) ((range * s->low_count) / s->scale);

        // This loop turns out new bits until high and low are far enough
        // apart to have stabilized.

        for (;;)
        {

        // If this test passes, it means that the MSDigits match, and can
        // be sent to the output stream.
```

```
                if ((high & 0x8000) == (low & 0x8000))
                {
                        OutputBit(stream, high & 0x8000);
                        while (underflow_bits > 0)
                        {
                                OutputBit(stream, ~high & 0x8000);
                                underflow_bits--;
                        }
                }

        // If this test passes, the numbers are in danger of underflow, because
        // the MSDigits don't match, and the 2nd digits are just one apart.

                else
                        if ((low & 0x4000) && !(high & 0x4000))
                        {
                                underflow_bits += 1;
                                low &= 0x3fff;
                                high |= 0x4000;
                        }
                        else
                                return;
                low <<= 1;
                high <<= 1;
                high |= 1;
        }
}

// When decoding, this routine is called to figure out which symbol
// is presently waiting to be decoded. This routine expects to get
// the current model scale in the s->scale parameter, and it returns
// a count that corresponds to the present floating point code:
//
// code = count / s->scale

short int get_current_count(SYMBOL *s)
{
        long range;
        short int count;

        range = (long) (high - low) + 1;
        count = (short int) ((((long) (code - low) + 1) * s->scale-1) / range);
        return(count);
}
```

```
// This routine is called to initialize the state of the arithmetic
// decoder. This involves initializing the high and low registers
// to their conventional starting values, plus reading the first
// 16 bits from the input stream into the code value.

void initialize_arithmetic_decoder(BIT_FILE *stream)
{
        int i;

        code = 0;
        for (i = 0; i < 16; i++)
        {
                code <<= 1;
                code += InputBit(stream);
        }
        low = 0;
        high = 0xffff;
}

// Just figuring out what the present symbol is doesn't remove
// it from the input bit stream. After the character has been
// decoded, this routine has to be called to remove it from the
// input stream.

void remove_symbol_from_stream(BIT_FILE *stream, SYMBOL *s)
{
        long range;

        // First, the range is expanded to account for the symbol removal.

        range = (long)(high - low) + 1;
        high = low + (unsigned short int) ((range * s->high_count) / s->scale - 1);
        low = low + (unsigned short int) ((range * s->low_count) / s->scale);

        // Next, any possible bits are shipped out.

        for (;;)
        {

        // If the MSDigits match, the bits will be shifted out.

                if ((high & 0x8000) == (low & 0x8000))
                {
                }
        // Else, if underflow is threatening, shift out the 2nd MSDigit.

                else
                        if ((low & 0x4000) == 0x4000  && (high & 0x4000) == 0)
                        {
                                code ^= 0x4000;
                                low   &= 0x3fff;
                                high  |= 0x4000;
                        }
                        else
                        // Otherwise, nothing can be shifted out, so I return.
                                return;
                low <<= 1;
                high <<= 1;
                high |= 1;
                code <<= 1;
                code += InputBit( stream );
        }
}
```

Appendix 2 – Performance Evaluation Software

```
// Program:          Compression Performance Evaluation Program
// Filename:         eval.cpp
// Description:      This program allows you to evaluate the performance of
//                   compression programs.  It allows you to measure Compression
//                   Ratio, Rate, Entropy, Mean-Squared Error (MSE), and Signal-to-
//                   Noise Ratio (SNR).
// Author:           Craig Chin
// Date :            02/13/01
// Notes:            To obtain useful results, the input filename must be of the input to
//                   the compression program, and the output filename must be of the
//                   compression output or the expansion output.  This progam must
//                   include the header file perfmeas.h, and it must be linked to
//                   perfmeas.cpp for proper functionality.

#include <iostream.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "eval.h"

// This main program provides the user with a menu interface that allows the user to
// choose the compression performance measure he wishes to utilize.

int main()
{
  int PerformanceMeasureChoice;

  do
  {
            cout << "This program will allow you to evaluate the performance of\n";
            cout << "your compression algorithms. This will be done with the use
                  of\n";
            cout << "certain performance measures.  These measures will include:\n";
            cout << "Compression ratio, Rate, Entropy, Mean-Squared
                  Error(MSE),\n";
            cout << "and Signal-to-Noise Ratio (SNR).\n";
            cout << "To choose the algorithm that will allow you to obtain one of\n";
            cout << "these performance measures, enter the number that
                  corresponds\n";
            cout << "to the menu items listed below.\n\n";
            cout << "1) Compression Ratio\n";
            cout << "2) Rate (bits/sample)\n";
            cout << "3) Entropy (bits/sample)\n";
            cout << "4) Mean-Squared Error\n";
            cout << "5) Signal-to-Noise Ratio\n";
            cout << "6) Exit\n\n";
            cin >> PerformanceMeasureChoice;

            switch(PerformanceMeasureChoice)
            {
                case 1: CompressionRatio();
                        break;
                case 2: Rate();
                        break;
                case 3: Entropy();
                        break;
                case 4: MSE();
                        break;
                case 5: SNR();
                        break;
                case 6: break;
                default: cout << "You have not made a valid entry. Try again.\n";
                        break;
            }
    }while(PerformanceMeasureChoice != 6);

    return 0;
}

// Program:          Compression Performance Evaluation Functions
// Filename:         perfmeas.cpp
// Description:      This file contains the function definitions that will allow you
//                   to evaluate the performance of compression programs.
// Author:           Craig Chin
// Date :            02/13/01
// Notes:            The associated function declarations are found in perfmeas.h. In
//                   order to use the file_size() function this program must be linked
//                   to utility_func.cpp.

#include <iostream.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "perfmeas.h"
#include "utility_func.h"

// This function obtains the compression ratio between compression input and output files
```

```
void CompressionRatio(void)
{
        double input_size;
        double output_size;
        double ratio;
        char inFileName[80];
        char outFileName[80];

        // Obtain the input and output filenames

        cout << "Enter the input filename.\n";
        cin >> inFileName;
        cout << "Enter the output filename.\n";
        cin >> outFileName;

        // Obtain the size of input an output files and calculate compression ratio

        input_size = (double) file_size(inFileName);
        if (input_size == 0)
                input_size = 1;
        output_size = (double) file_size(outFileName);
        if (output_size == 0)
                output_size = 1;
        ratio = input_size / output_size;

        cout << "Size of Input File = " << input_size << " bytes.\n";
        cout << "Size of Output File = " << output_size << " bytes.\n";
        cout << "Compression Ratio = " << ratio << ":1\n\n";
}

void Rate(void)
{
        double input_size;
        double output_size;
        double rate;
        double BytesPerSample;
        char inFileName[80];
        char outFileName[80];

        // Obtain the input and output filenames and the number of bytes per sample

        cout << "Enter the input filename.\n";
        cin >> inFileName;
        cout << "Enter the output filename.\n";
        cin >> outFileName;
        cout << "Enter the number of bytes per sample.\n";
        cin >> BytesPerSample;

        // Obtain the size of input an output files and calculate rate

        input_size = (double) file_size(inFileName);
        if (input_size == 0)
                input_size = 1;
        output_size = (double) file_size(outFileName);
        if (output_size == 0)
                output_size = 1;
        rate = (output_size * 8) / (input_size/BytesPerSample);

        cout << "Size of Input File = " << input_size << " bytes.\n";
        cout << "Size of Output File = " << output_size << " bytes.\n";
        cout << "Rate = " << rate << " bits/sample.\n\n";
}

void Entropy(void)
{
        char inFileName[80];
        FILE *input;
        long TotalSamples;
        int symbol;
        long SampleCountArray[256];
        double probArray[256];
        double entropy;
        int i;

        // Obtain input filename, input file size, and open input file

        cout << "Enter the input filename.\n";
        cin >> inFileName;
        TotalSamples = file_size(inFileName);
        input = fopen(inFileName, "rb");
        if (input == NULL)
        {
                cout << "Cannot open " << inFileName << " for input.\n";
                exit(0);
        }

        // Initialize SampleCountArray

        for(i = 0; i < 256; i++)
```

```
			SampleCountArray[i] = 0;

		// Fill SampleCountArray

		while ((symbol = getc(input)) != EOF)
			SampleCountArray[symbol]++;

		// Determine input file entropy

		entropy = 0.0;
		for (i = 0; i < 256; i++)
		{
			probArray[i] = (double)SampleCountArray[i] / (double)TotalSamples;
			if (probArray[i] > 0.0)
			entropy -= probArray[i] * log(probArray[i]);
		}

		entropy = entropy / log((double)2.0);

		cout << "Total Samples = " << TotalSamples << "\n";
		cout << "Entropy = " << entropy << " bits/sample.\n\n";
		fclose(input);
}

void MSE(void)
{
		char inFileName[80];
		char outFileName[80];
		long input_size;
		FILE *input;
		FILE *output;
		unsigned long square_error;
		int input_value, output_value;
		double MSE;

		// Obtain the input and output filenames

		cout << "Enter the input filename.\n";
		cin >> inFileName;
		cout << "Enter the output filename.\n";
		cin >> outFileName;

		// Obtain input file size

		input_size = file_size(inFileName);
```

```
        // Obtain Square-Error for input and output files

        input = fopen(inFileName, "rb");
        output = fopen(outFileName, "rb");
        square_error = 0;
        while ((input_value = getc(input)) != EOF)
        {
                output_value = getc(output);
                square_error += ((unsigned long)input_value - (unsigned
                                long)output_value)* ((unsigned long)input_value –
                               (unsigned long)output_value);
        }

        // compute mean-square error

        MSE = (double)square_error / (double)input_size;

        cout << "Total Samples = " << input_size << "\n";
        cout << "Mean-Squared Error = " << MSE << "\n\n";

        fclose(input);
        fclose(output);
}

void SNR(void)
{
        char inFileName[80];
        char outFileName[80];
        long input_size;
        FILE *input;
        FILE *output;
        unsigned long signal_squared, square_error;
        int input_value, output_value;
        double MSE, avg_sq_signal, SNR;

        // Obtain the input and output filenames

        cout << "Enter the input filename.\n";
        cin >> inFileName;
        cout << "Enter the output filename.\n";
        cin >> outFileName;

        // Obtain input file size
        input_size = file_size(inFileName);

        // Obtain Square-Error and Signal_Squared from input and output files

        input = fopen(inFileName, "rb");
        output = fopen(outFileName, "rb");
        signal_squared = 0;
        square_error = 0;
        while ((input_value = getc(input)) != EOF)
        {
                output_value = getc(output);
                signal_squared += (unsigned long)input_value * (unsigned
                                        long)input_value;
                square_error += ((unsigned long)input_value - (unsigned
                                        long)output_value)*((unsigned long)input_value –
                                    (unsigned long)output_value);
        }

        // compute average square input value, mean-square error and SNR

        MSE = (double)square_error / (double)input_size;
        avg_sq_signal = (double)signal_squared / (double)input_size;
        cout << "Total Samples = " << input_size << "\n";
        if (MSE == 0.0)
                cout << "Signal-to-Noise Ratio =  infinity\n\n";
        else
        {
                SNR = avg_sq_signal / MSE;
                cout << "Total Signal-Squared = " << signal_squared <<"\n";
                cout << "Average Squared Signal = " << avg_sq_signal <<"\n";
                cout << "Mean-Squared Error = " << MSE << "\n";
                cout << "Signal-to-Noise Ratio = " << SNR << "\n\n";
        }

        fclose(input);
        fclose(output);
}

// Program:         Compression Performance Evaluation Function declarations
// Filename:        perfmeas.h
// Description:     This file contains the function declarations that will allow you
//                  to evaluate the performance of compression programs.
// Author:          Craig Chin
// Date :           02/13/01
// Notes:           The associated function definitions are found in perfmeas.cpp.
```

```
void CompressionRatio(void);
void Rate(void);
void Entropy(void);
void MSE(void);
void SNR(void);


// Program:           Utility Function Definitions for Compression/Expansion Programs
// Filename:          utility_func.cpp
// Description:       This program contains function definitions that are utilized in some
//                    of the compression programs.
// Author:            Craig Chin
// Date :             02/01/01
// Notes:             The code is a modification of source code obtained from "The Data
//                    Compression Book", by M. Nelson and J.L. Gailly.  The function
//                    declarations are found in utility_func.h

#include <stdio.h>
#include "utility_func.h"

// This routine prints out the compression ratios after the input
// and output files have been closed.

void print_ratios(char *input, char* output)
{
   long input_size;
   long output_size;
   double ratio;

   input_size = file_size(input);
   if (input_size == 0)
         input_size = 1;
   output_size = file_size(output);
   if (output_size == 0)
         output_size = 1;
   ratio =  (double)input_size / (double)output_size;
   printf("\nInput bytes:        %ld\n", input_size);
   printf("Output bytes:       %ld\n", output_size);
   printf("Compression ratio:  %f:1\n", ratio);
}

// This routine is used get the size of a file.  It does all the work, and returns a long.

#ifndef SEEK_END
```

```
#define SEEK_END 2
#endif

long file_size(char *name)
{
   long eof_ftell;
   FILE *file;

   file = fopen(name, "r");
   if (file == NULL)
      return(0L);
   fseek(file, 0L, SEEK_END);
   eof_ftell = ftell(file);
   fclose(file);
   return(eof_ftell);
}

// Program:          Utility Function Declarations
// Filename:         utility_func.h
// Description:      This program contains function declarations that are utilized in
//                   some of the compression programs.
// Author:           Craig Chin
// Date :            02/01/01
// Notes:            The function definitions are found in utility_func.cpp

void print_ratios(char *input, char* output);
long file_size(char *name);
```

Appendix 3 – VOIP Application Software

```
// Program:         Voice-over-IP half-duplex application
// Filename:        voipmenu.cpp
// Description:     This program allows you to send and receive voice messages over
//                  the internet.  A DCT compression/expansion algorithm is used to
//                  reduce the bandwidth utilized in the transmission of audio data
//                  across the internet.
// Author:          Craig Chin
// Date :           02/01/01
// Notes:           The code is a modification of the source code client.c and server.c
//                  obtained from "Computer Networks and Internets", by D. E.
//                  Comer.
//                  The following support files must be linked to this program in order
//                  to provide proper functionality: utility_func.cpp, onedddct.cpp,
//                  errhand.cpp, ahuff.cpp, bitio.cpp, and writeWaveData.cpp.

#ifndef unix
#define WIN32
#include <windows.h>
#include <winsock.h>
#else
#define closesocket close
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#endif

#include <stdio.h>
#include <string.h>
#include <io.h>
#include <stdlib.h>
#include <iostream.h>
#include "utility_func.h"

#define PROTOPORT     5193      // default protocol port number
#define QLEN          6         // size of request queue
extern  int        errno;

// function prototypes
void send_message(void);
void receive_message(void);
void writeWaveData(char *file_name);
void readWaveData(void);

// These functions are used to compress and expand files for a client-server
// application.  Refer to file onedddct.cpp for the function definitions.

void send_dctC(char inFileName[], char outFileName[]);
void recv_dctE(char inFileName[], char outFileName[]);

// The main() function is used allow the user to interact with the program via
// menu-type selections.

int main()
{
   int choice;   //Menu choice variable

   do
   {

         cout << "This program facilitates communication across the internet via voice
         messages.\n";
         cout << "Enter one of the following numbers to perform the task you desire\n\n";
         cout << "1) To send a voice message\n";
         cout << "2) To receive a voice message\n";
                  cout << "3) To exit the program\n";
         cin >> choice;
         switch(choice)
         {
                          case 1: send_message();
                                          break;
                          case 2: receive_message();
                                          break;
                          case 3: break;
                          default: cout << "You have made an invalid entry. Try again.\n";
                          break;
                  }
         }while(choice != 3);

         return( 0 );
}

// The send_message() function is used to compress and send a voice message to a
// remote computer.

void send_message(void)
{
```

```
struct  hostent  *ptrh = 0;              // pointer to a host table entry
struct  protoent *ptrp = 0;              // pointer to a protocol table entry
struct  sockaddr_in sad;                 // structure to hold an IP address
int     sd;                              // socket descriptor
const int port = PROTOPORT;              // protocol port number
char host[30];                           // buffer to hold host name
char recv_buf[20];                       // buffer for confirmation response
char inFileName[] = "recordData";        // Input filename buffer
char outFileName[] = "comp_out";         // Output filename buffer
FILE * output;                           // Compression output file pointer
char message_buf[640000];                // Output message buffer
long    size_of_file[1];                 // size of file to be sent
int n;                                   // number of characters read
int i;

#ifdef WIN32
        WSADATA wsaData;
        WSAStartup(0x0101, &wsaData);
#endif
memset((char *)&sad,0,sizeof(sad));      // clear sockaddr structure
sad.sin_family = AF_INET;                // set family to Internet

// Write protocol port number to address structure

sad.sin_port = htons((u_short)port);

// Request host name of computer from user and store it

cout << "Enter the host name of the computer you wish to contact.\n";
cout << "If you desire to use the local computer type 'localhost'.\n";
cin >> host;

// Convert host name to equivalent IP address and copy to sad.

ptrh = gethostbyname(host);
if (((char *)ptrh) == NULL)
{
        fprintf(stderr,"invalid host: %s\n", host);
        exit(1);
}
memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);

// Map TCP transport protocol name to protocol number.

if (((int)(ptrp = getprotobyname("tcp"))) == 0)
```

```
{
        fprintf(stderr, "cannot map \"tcp\" to protocol number");
        exit(1);
}

// Create a socket.

sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
if (sd < 0)
{
        fprintf(stderr, "socket creation failed\n");
        exit(1);
}

// Connect the socket to the specified server.

if (connect(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0)
{
        fprintf(stderr,"connect failed\n");
        exit(1);
}

// Record audio file with filename "recordData"

readWaveData();

// Compress input file to produce output file
send_dctC(inFileName, outFileName);

// Obtain size of Compression output
size_of_file[0] = file_size(outFileName);

// Send message to host

send(sd, (char*)size_of_file, sizeof(size_of_file), 0);

closesocket(sd);

// Create a socket.

sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
if (sd < 0)
{
        fprintf(stderr, "socket creation failed\n");
        exit(1);
```

```
}

// Connect the socket to the specified server.

if (connect(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0)
{
        fprintf(stderr,"connect failed\n");
        exit(1);
}

// Wait for confirmation response

n = recv(sd, recv_buf, sizeof(recv_buf), 0);
while (n > 0)
{
        n = recv(sd, recv_buf, sizeof(recv_buf), 0);
}

// Print confirmation response

cout << "File size acknowledge message has been ";
for (i = 0; i < sizeof(recv_buf); i++)
{
        if (recv_buf[i] == '\0')
                break;
        else
                cout << recv_buf[i];
}

cout << "\n";

// Close the socket.

closesocket(sd);

// If confirmation response has been received sucessfully; send message.

if(recv_buf[0] == 'r')
{
        // Create a socket.

        sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
        if (sd < 0)
        {
                fprintf(stderr, "socket creation failed\n");
                exit(1);
            }

            // Connect the socket to the specified server.

            if (connect(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0)
            {
                fprintf(stderr,"connect failed\n");
                exit(1);
            }

            // Open compression output file

            output = fopen(outFileName, "rb");
            if (output == NULL)
            {
                cout << "Error opening " << outFileName << " for reading.\n";
                exit(1);
            }

            // Read output file into messsage buffer

            fread(message_buf, sizeof(char), size_of_file[0], output);

            // Send message to host

            send(sd, message_buf, sizeof(message_buf), 0);

            closesocket(sd);
        }
        else
            cout << "Error with receive message from server.\n";

}

// The receive_message() function is used to receive and expand a voice message
// from a remote computer.

void receive_message(void)
{
        struct hostent  *ptrh = 0;              // pointer to a host table entry
        struct protoent *ptrp = 0;              // pointer to a protocol table entry
        struct sockaddr_in sad;                 // structure to hold server's address
        struct sockaddr_in cad;                 // structure to hold client's address
```

```
int sd, sd2;                            // socket descriptors
const int port = PROTOPORT;             // protocol port number
int alen;                               // length of address
int n;                                  // number of characters read
long size_of_file[1];                   // size of file to be received
char send_buf[] = "received";           // file size confirmation message
char message_buf[640000];               // Input message buffer
FILE * input;                           // Expansion input file pointer
char inFileName[] = "exp_in";           // Expansion input filename buffer
char outFileName[] = "exp_out";         // Expansion output filename buffer

#ifdef WIN32
        WSADATA wsaData;
        WSAStartup(0x0101, &wsaData);
#endif
memset((char *)&sad,0,sizeof(sad));     // clear sockaddr structure
sad.sin_family = AF_INET;               // set family to Internet
sad.sin_addr.s_addr = INADDR_ANY;       // set the local IP address

// Write protocol port number to address structure

sad.sin_port = htons((u_short)port);

// Map TCP transport protocol name to protocol number

if (((int)(ptrp = getprotobyname("tcp"))) == 0)
{
        fprintf(stderr, "cannot map \"tcp\" to protocol number");
        exit(1);
}

// Create a socket

sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
if (sd < 0)
{
        fprintf(stderr, "socket creation failed\n");
        exit(1);
}

// Bind a local address to the socket

if (bind(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0)
{
        fprintf(stderr,"bind failed\n");
        exit(1);
}

// Specify size of request queue

if (listen(sd, QLEN) < 0)
{
        fprintf(stderr,"listen failed\n");
        exit(1);
}

// Accept connection request

alen = sizeof(cad);
if ((sd2=accept(sd, (struct sockaddr *)&cad, &alen)) < 0)
{
        fprintf(stderr, "accept failed\n");
        exit(1);
}

// Receive file size information

n = recv(sd2, (char*)size_of_file, sizeof(size_of_file), 0);
while (n > 0)
{
        n = recv(sd2, (char*)size_of_file, sizeof(size_of_file), 0);
}

cout << "Size of incoming file is " << size_of_file[0] << ".\n";

closesocket(sd2);

// Accept connection request

alen = sizeof(cad);
if ( (sd2=accept(sd, (struct sockaddr *)&cad, &alen)) < 0)
{
        fprintf(stderr, "accept failed\n");
        exit(1);
}


// Send file size confirmation

send(sd2, send_buf, sizeof(send_buf), 0);
```

```
        cout << "I have sent my reply.\n";

        closesocket(sd2);

        // Accept connection request

        alen = sizeof(cad);
        if ( (sd2=accept(sd, (struct sockaddr *)&cad, &alen)) < 0)
        {
                fprintf(stderr, "accept failed\n");
                exit(1);
        }

        // Open expansion input file
        input = fopen(inFileName, "wb");
        if (input == NULL)
        {
                cout << "Error opening " << inFileName << " for writing.\n";
                exit(1);
        }

        // Receive message

        n = recv(sd2, message_buf, sizeof(message_buf), 0);
        while (n > 0)
        {
                n = recv(sd2, message_buf, sizeof(message_buf), 0);
        }

        // Write message to expansion input file

        fwrite(message_buf, sizeof(char), size_of_file[0], input);
        fclose(input);
        closesocket(sd2);
        closesocket(sd);

        // Expand input file to produce output file
        recv_dctE(inFileName, outFileName);

        // Output audio file to speakers
        writeWaveData(outFileName);
}

// Program:            Record audio waveform data from input device
// Filename:        readWaveData.cpp
// Description:     This file contains a function that allows you to read a block of
//                  8-bit, PCM, 8 kHz audio data from a microphone.
// Author:          Craig Chin
// Date :           02/03/01
// Notes:           The code is a modification of the source code found on the MSDN
//                  website at the following URL
//                  http://msdn.microsoft.com/library/psdk/multimed/wave_8rj5.htm.
//                  The title of the webpage is "Example of Writing Waveform Data."
//                  The following resource files must be included with this file for
//                  proper  functionality: winmm.lib, kernel32.lib.


#define WIN32
#include <windows.h>
#include <mmsystem.h>
#include <mmreg.h>
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "utility_func.h"

//Function Prototypes

void readWaveData(void);

// This function performs the actual reading of audio data from the microphone

void readWaveData(void)
{
        HANDLE hDataIn  = NULL;           // handle of waveform data memory
        LPVOID lpDataIn = NULL;           // pointer to waveform data memory
        HWAVEIN hWaveIn;                  // handle to waveform-audio input device
        HGLOBAL hWaveHdr;                 // handle to WAVEHDR object
        LPWAVEHDR lpWaveHdr;              // pointer to WAVEHDR structure
        UINT wResult;
        HANDLE hFormat = 0;
        WAVEFORMATEX pcmWaveFormat;          // WAVEFORMATEX structure
        WAVEFORMATEX *pFormat = &pcmWaveFormat; // pointer to
                                             // WAVEFORMATEX structure
        DWORD dwDataSize = 640000L;       // # of bytes of memory to allocate
        HWND hwndApp = NULL;              // window handle
        unsigned long bytesRecorded = 0L; // # of bytes recorded
        char record_start = 0;            // variable to signal recording start
char record_stop = 0;              // variable to signal recording stop
char file_name[] = "recordData";   // filename for recorded audio file
FILE * record_file;                // FILE pointer to recorded audio file
time_t start, stop;                // time variables for record start and stop
                                   // times
double time_diff;                  // recording time in seconds

// Set up WAVEFORMATEX structure for 8 kHz, 8-bit mono.

pcmWaveFormat.wFormatTag = WAVE_FORMAT_PCM;
pcmWaveFormat.nChannels = 1;
pcmWaveFormat.nSamplesPerSec = 8000L;
pcmWaveFormat.nAvgBytesPerSec = 8000L;
pcmWaveFormat.nBlockAlign = 1;
pcmWaveFormat.wBitsPerSample = 8;

// Create an audio file called "recdata.raw"
record_file = fopen(file_name, "wb");

// Open a waveform device for output using window callback.

if (waveInOpen((LPHWAVEIN)&hWaveIn, WAVE_MAPPER,
      (LPWAVEFORMATEX)pFormat,
      (long)hwndApp, 0L, CALLBACK_WINDOW))
{
        cout << "Failed to open waveform input device.\n";
        LocalUnlock(hFormat);
        LocalFree(hFormat);
        fclose(record_file);
        return;
}

// Allocate and lock memory for the recording of waveform data.

hDataIn = GlobalAlloc(GMEM_MOVEABLE | GMEM_SHARE, dwDataSize);
if (!hDataIn)
{
        cout << "Out of memory.\n";
        fclose(record_file);
        return;
}
if ((lpDataIn = GlobalLock(hDataIn)) == NULL)
{
        cout << "Failed to lock memory for data chunk.\n";
        GlobalFree(hDataIn);
        fclose(record_file);
        return;
}

// Allocate and lock memory for the header.

hWaveHdr = GlobalAlloc(GMEM_MOVEABLE | GMEM_SHARE, (DWORD)
sizeof(WAVEHDR));
if (hWaveHdr == NULL)
{
        cout << "Not enough memory for header.\n";
        GlobalUnlock(hDataIn);
        GlobalFree(hDataIn);
        return;
}

lpWaveHdr = (LPWAVEHDR) GlobalLock(hWaveHdr);
if (lpWaveHdr == NULL)
{
        cout << "Failed to lock memory for header.\n";
        GlobalUnlock(hDataIn);
        GlobalFree(hDataIn);
        return;
}

// After allocation, set up and prepare header.

lpWaveHdr->lpData = (HPSTR) lpDataIn;
lpWaveHdr->dwBufferLength = dwDataSize;
lpWaveHdr->dwFlags = 0L;
lpWaveHdr->dwLoops = 0L;
waveInPrepareHeader(hWaveIn, lpWaveHdr, sizeof(WAVEHDR));

// The data buffer can now be sent to the waveform-audio input
// device for the recording of data.  The waveInAddBuffer()
// function is used for this purpose.

wResult = waveInAddBuffer(hWaveIn, lpWaveHdr, sizeof(WAVEHDR));
if (wResult != 0)
{
        cout << "Failed to write block to device.\n";
        waveInUnprepareHeader(hWaveIn, lpWaveHdr, sizeof(WAVEHDR));
        GlobalUnlock(hDataIn);
        GlobalFree(hDataIn);
        return;
```

```
}

cout << "To start recording an audio file press the 'r' key.\n";
cout << "To stop recording an audio file press the 'r' key again.\n";
cout << "You will be limited to a maximum of 10 seconds of audio.\n";

// Use 'r' key to start recording audio data

while(record_start != 'r')
{
        cin >> record_start;
        if(record_start == 'r')
        {
                wResult = waveInStart(hWaveIn);
                start = time(NULL);
                if (wResult != 0)
                {
                        cout << "Failed to write block to device.\n";
                        waveInUnprepareHeader(hWaveIn, lpWaveHdr,
                                                        sizeof(WAVEHDR));
                        GlobalUnlock(hDataIn);
                        GlobalFree(hDataIn);
                        return;
                }
        }
        else
                cout << "Invalid input! Please try again.\n";
}

cout << "Recording audio input.\n";

// Use 'r' key to stop recording audio data

while(record_stop != 'r')
{
        cin >> record_stop;
        if(record_stop == 'r')
        {
                wResult = waveInReset(hWaveIn);
                stop = time(NULL);
                if (wResult != 0)
                {
                        waveInUnprepareHeader(hWaveIn, lpWaveHdr,
                                                  sizeof(WAVEHDR));
                        GlobalUnlock(hDataIn);
```

```
                                        GlobalFree(hDataIn);
                                        cout << "Failed to write block to device.\n";
                                        return;
                                }

                        }
                        else
                                cout << "Invalid input! Please try again.\n";
        }

        cout << "Recording of audio has stopped.\n";

        // Calculate the # of bytes recorded

        time_diff = difftime(stop, start);
        bytesRecorded = 64000L * (long)time_diff;
        cout << "Bytes Recorded = " << bytesRecorded << ".\n";

        // Write the waveform data subchunk to file.


        if(fwrite(lpDataIn, sizeof(char), bytesRecorded, record_file) != bytesRecorded)
        {
                cout << "Failed to write the data chunk.\n";
                GlobalUnlock(hDataIn);
                GlobalFree(hDataIn);
                fclose(record_file);
                return;
        }

        // Cleanup procedures
        waveInUnprepareHeader(hWaveIn, lpWaveHdr, sizeof(WAVEHDR));
        GlobalUnlock(hDataIn);
        GlobalFree(hDataIn);
        waveInClose(hWaveIn);
        fclose(record_file);
        cout << "Audio file recorded successfully.\n";
        return;
}

// Program:     Write audio waveform data to output device
// Filename:    writeWaveData.cpp
// Description: This file contains a function that allows you to write a block of
//              8-bit, PCM, 8 kHz audio data to the speakers of your computer.
// Author:      Craig Chin
// Date :       02/01/01
// Notes:       The code is a modification of the source code found on the MSDN website
//              at the following URL
//              http://msdn.microsoft.com/library/psdk/multimed/wave_8rj5.htm.
//              The title of the webpage is "Example of Writing Waveform Data."  The
//              following resource files must be included with this file for proper
//              functionality: winmm.lib, and kernel32.lib.

#define WIN32
#include <windows.h>
#include <mmsystem.h>
#include <mmreg.h>
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include "utility_func.h"

// Function Prototypes

void writeWaveData(char *file_name);

// This function performs the actual writing of audio data out to the speakers

void writeWaveData(char *file_name)
{
        HANDLE hDataOut  = NULL;         // handle of waveform data memory
        LPVOID lpDataOut = NULL;         // pointer to waveform data memory
        HWAVEOUT   hWaveOut;             // handle to waveform-audio output device
        HGLOBAL    hWaveHdr;             // handle to WAVEHDR object
        LPWAVEHDR  lpWaveHdr;            // pointer to WAVEHDR structure
        HMMIO      hmmio;                // audio file handle
        UINT       wResult;
        HANDLE     hFormat = 0;
        WAVEFORMATEX pcmWaveFormat;          // WAVEFORMATEX structure
        WAVEFORMATEX  *pFormat = &pcmWaveFormat; // pointer to
                                             // WAVEFORMATEX structure
        DWORD      dwDataSize;           // # of bytes of memory to allocate
        HWND hwndApp = NULL;             // window handle

        // Obtain the size of the audio file

        dwDataSize = file_size(file_name);
        cout << "Size of expanded output file = " << dwDataSize << ".\n";

        // Set up WAVEFORMATEX for 8 kHz, 8-bit mono.
```

```
pcmWaveFormat.wFormatTag = WAVE_FORMAT_PCM;
pcmWaveFormat.nChannels = 1;
pcmWaveFormat.nSamplesPerSec = 8000L;
pcmWaveFormat.nAvgBytesPerSec = 8000L;
pcmWaveFormat.nBlockAlign = 1;
pcmWaveFormat.wBitsPerSample = 8;

// Open audio file for buffered I/O

hmmio = mmioOpen((LPSTR) file_name, NULL, MMIO_READ);
cout << "Audio file handle  = " << hmmio << "\n";

// Open a waveform device for output using window callback.

if (waveOutOpen((LPHWAVEOUT)&hWaveOut, WAVE_MAPPER,
      (LPWAVEFORMATEX)pFormat,
      (long)hwndApp, 0L, CALLBACK_WINDOW))
{
         cout << "Failed to open waveform output device.\n";
         LocalUnlock(hFormat);
         LocalFree(hFormat);
         mmioClose(hmmio, 0);
         return;
}

// Allocate and lock memory for the waveform data.

hDataOut = GlobalAlloc(GMEM_MOVEABLE | GMEM_SHARE, dwDataSize);
if (!hDataOut)
{
         cout << "Out of memory.\n";
         mmioClose(hmmio, 0);
         return;
}
if ((lpDataOut = GlobalLock(hDataOut)) == NULL)
{
         cout << "Failed to lock memory for data chunk.\n";
         GlobalFree(hDataOut);
         mmioClose(hmmio, 0);
         return;
}

// Read the waveform data into memory.
```

```
if(mmioRead(hmmio, (HPSTR) lpDataOut, dwDataSize) !=
(LRESULT)dwDataSize)
{
        cout << "Failed to read data chunk.\n";
        GlobalUnlock(hDataOut);
        GlobalFree(hDataOut);
        mmioClose(hmmio, 0);
        return;
}

// Allocate and lock memory for the header.

hWaveHdr = GlobalAlloc(GMEM_MOVEABLE | GMEM_SHARE,
                (DWORD) sizeof(WAVEHDR));
if (hWaveHdr == NULL)
{
        cout << "Not enough memory for header.\n";
        GlobalUnlock(hDataOut);
        GlobalFree(hDataOut);
        return;
}

lpWaveHdr = (LPWAVEHDR) GlobalLock(hWaveHdr);
if (lpWaveHdr == NULL)
{
        cout << "Failed to lock memory for header.\n";
        GlobalUnlock(hDataOut);
        GlobalFree(hDataOut);
        return;
}

// After allocation, set up and prepare WAVEHDR structure.

lpWaveHdr->lpData = (HPSTR) lpDataOut;
lpWaveHdr->dwBufferLength = dwDataSize;
lpWaveHdr->dwFlags = 0L;
lpWaveHdr->dwLoops = 0L;
waveOutPrepareHeader(hWaveOut, lpWaveHdr, sizeof(WAVEHDR));

// Now the data block can be sent to the output device. The
// waveOutWrite() function returns immediately and waveform
// data is sent to the output device in the background.

wResult = waveOutWrite(hWaveOut, lpWaveHdr, sizeof(WAVEHDR));
if (wResult != 0)
```

```
        {
                cout << "Failed to write block to device.\n";
                waveOutUnprepareHeader(hWaveOut, lpWaveHdr, sizeof(WAVEHDR));
                GlobalUnlock(hDataOut);
                GlobalFree(hDataOut);
                return;
        }
        cout << "Audio file output successfully!\n";
        return;
}
```