2

AFIT/GSO/ENG/93D-02

1

# FILTERING, CODING, AND COMPRESSION WITH MALVAR WAVELETS

THESIS

Stephen Robert Hall
Captain, USAF

AFIT/GSO/ENG/93D-02

1508

AFIT/GSO/ENG/93D-02

# FILTERING, CODING, AND COMPRESSION WITH MALVAR WAVELETS

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Space Operations

Stephen Robert Hall, B.S.

Captain, USAF

December, 1993

## *Preface*

I would like to thank my thesis committee for their contributions to making this project possible. Special thanks to my advisor, Dr. Bruce Suter, whose enthusiasm for his work was truely contagious. His knowledge, background, and mathematical developments were critical to this thesis. I must also personally thank Dr. Matthew Kabrisky for answering my many silly questions and providing valuable insight; Dr. Tim Anderson for providing an "engineers" perspective on things and telling me how things work in the real world; and Janet Slifka for all the computer wizardry.

I would like to thank my wife, Lisa, who was able to adjust to the long hours and lack of attention that AFIT creates. Without the love and support of Lisa and my daughter, Katie, I would not have been able to complete this program. AFIT has renewed my belief in the importance of family. I also need to thank the Lord for making everything in my life possible. Finally, this is for you Pop-Pop.

Stephen Robert Hall

## *Table of Contents*

Page

## *List of Figures*

Figure Page

# *List of Tables*

AFIT/GSO/ENG/93D-02

## *Abstract*

This thesis develops and evaluates a number of new concepts and tools for the analysis of signals using Malvar wavelets (lapped orthogonal transforms). Windowing, often employed as a spectral estimation technique, can result in irreparable distortions in the transformed signal. By utilizing the Malvar wavelet, any signal distortion resulting from the transformation can be eliminated or cancelled during reconstruction. This is accomplished by placing conditions on the window and the basis function and then incorporating the window into the orthonormal representation.

Paradigms for both a complex-valued and a real-valued Malvar wavelet are summarized. The algorithms for the wavelets were implemented in the DOD standard language, Ada. The code was verified to ensure perfect reconstruction could be obtained and experiments were performed using the wavelet programs. Various compression techniques were investigated and evaluated using the Malvar wavelet in both homomorphic and non-homomorphic systems. The Malvar wavelet has the unique ability to overlap adjacent windows without increasing the number of transform coefficients. Various amounts of window overlap were investigated to determine if there is an optimal amount which should be used. In addition, the real-valued basis function was used to attempt real-valued deconvolution. It was found that the real-valued Malvar wavelet, with just one point of overlap for each window, provided better or equally as good reconstruction of signals as most of the more complex systems. This same real-valued basis function could be used to perform deconvolution, if the original signal has certain symmetries.

# FILTERING, CODING, AND COMPRESSION WITH MALVAR WAVELETS

# *I. Introduction*

## *Background*

The United States has a great need for improved digital signal processing techniques in order to reduce the burden on overloaded communications satellites and to reap more information from satellite imagery.

*Data Compression:* One of the largest problems during Operation Desert Storm was the lack of communication channels available to field and command units. Virtually 100% of the military ultra-high frequency and super-high frequency satellite communication channels were consumed by logistics, administrative, and intelligence data during the conflict (5:82). Over 90% of the communications into and out of the theater went over communication satellites, however over 20% of the information had to be sent via commercial satellites (36:4).

Compressing speech requires a three way trade-off among the goals of preserving intelligibility and quality, limiting the bit rate, and minimizing computational complexity (23:225). All three of these objectives must be considered when chocsing the best compression algorithm. Transform coding is one of the major low bit rate speech coding techniques being investigated by the military (38).

*Imagery:* Space imagery often requires adaptive restoration to deblur out-of-focus imagery because it is generally impossible to merely retake a given photograph under better conditions (12). "Digital image restoration is useful whenever needed information in an image is hidden by some type of degradation. The classic statement of the image restoration problem is: given a noisy and blurred image, find an estimate of the ideal image using *a priori* information about the blur, noise, and the ideal image" (12). The research for this thesis deals only with one dimensional signals, however many of the techniques can easily be adapted to two dimensional images. The Malvar wavelet is becoming increasingly important to all fields of digital signal processing and has been successfully used with an imaging technique called motion estimation (41).

Whenever a signal is transmitted through any medium it becomes convolved with a function related to that medium. Speech is a signal that is a convolution of the speaker's voice pitch and the sound of the word made by the vocal tract. A satellite image can be represented by the original image convolved with its optical transfer function (or blurring function, which is a mathematical representation of the atmosphere). A seismic recording can be described as the original signal convolved with impulses, which appear as echoes in the seismic event. The term deconvolution indicates the technique used to remove the transfer function from a given signal. This includes the process of debluring images (12) or restoring old acoustic recordings (27).

*Processing Technique:* The Lapped Orthogonal Transform (LOT) is an important signal processing tool that has been used to filter and code both speech (26) and image (17) (41) signals. The LOT was first introduced by H.S. Malvar and D.H. Staelin, in 1988, in order to reduce errors in image reconstruction (17). The complete derivation and implementation of the LOT is outlined in Chapter 2. The LOT has also been termed the *Malvar wavelet*, by Yves Meyer (19), because of the algorithm's importance to wavelet analysis; complimenting wavelet packet analysis. Ville proposed two types of signal analysis in 1947. He explained that "we can either: first cut a signal into slices (in time) with a switch; then pass these different slices through a system of filters to analyze them. Or we can: first filter different frequency bands; then cut these bands into slices (in time) to study their energy variations (19)." Yves Meyer has explained that the first approach leads us to "Malvar wavelets" and the second approach leads to "wavelet packets" (19). These two algorithms are the underlying foundation for wavelet analysis (19) (29). The terms "Lapped Orthogonal Transform" and "Malvar wavelet" will be used interchangeably throughout this paper.

## *Problem Statement*

Using the Malvar wavelet, investigate new ways in which this signal processing technique may be used to filter, code, and compress digital signals.

## *Research Goals*

The research for this thesis effort was essentially a set of experiments all related to the Malvar wavelet (LOT) analysis. Several goals were established which guided the research for this thesis. The research goals are as follows:

1. Investigate the use of a complex-valued basis with the Malvar wavelets. Can orthogonality be preserved and can the algorithm be efficiently coded? Can this wavelet be successfully used to perform data compression?
2. Investigate the use of the Lapped Orthogonal Transform in a homomorphic filter for speech processing. Determine if there is a best method for data compression.
3. Investigate the use of overlap with real-valued Malvar wavelets. Is there a correlation between the amount of overlap and the amount of compression?
4. Investigate the use of the sinusoidal basis function (as described in Chapter II and used with the Sinusoidal Malvar wavelet) in a homomorphic filter to perform signal deconvolution. Can real-valued basis functions be useful for signal deconvolution?

## *Outline*

Chapter II expands on the background and theory used for this research effort. It includes development of homomorphic filtering and the lapped transform. A complex-valued basis function has many inherent advantages and Chapter III reports the effort to use an exponential basis function with the Malvar wavelet. Chapter IV presents the results of the lapped transform evaluated with numerous compression techniques, including both homomorphic and non-homomorphic filters. Chapter V is concerned with finding the optimal amount of overlap within the real-valued Malvar wavelet. Homomorphic deconvolution was investigated and the results of this work constitute Chapter VI. Finally, a summary of key results, and a recommendation for follow-on research areas, are outlined in Chapter VII.

# *II. Background*

## *Introduction*

Two important signal processing techniques must be well-defined for this research effort:

1. The Homomorphic Filter

2. The Malvar Wavelet (Lapped Orthogonal Transform)

This chapter will review homomorphic systems, the cepstrum (a specific case of the homomorphic filter), transforms, their measurements of performance, and finally the lapped transform.

## *Homomorphic Filters*

Homomorphic filters are non-linear digital signal processing (DSP) systems that are very important to speech and image processing (22). Homomorphic systems were originally described by Dr. Alan Oppenheim (21). Oppenheim showed that, in its most general form, a homomorphic system can be represented by algebraically linear (homomorphic) mappings between the input and output signals (22:771). This system can be broken down into three separate homomorphic system (as shown in figure 2.1); a characteristic system (**D**), a linear system (**L**), and the inverse characteristic system (**D(inv)**) (22:772).

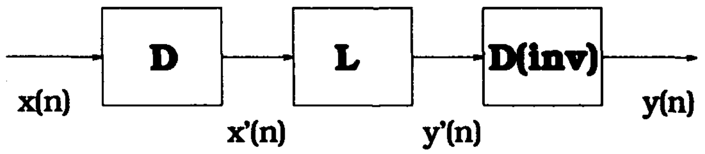


Figure 2.1. Canonic representation of homomorphic systems

The cepstrum is a popular dsp tool that is a special case of a homomorphic system. The cepstrum implements the Fourier transform as its characteristic system and the natural logarithm as its linear system (27).

## *Cepstrum*

Bogert, Healy, and Tukey first defined the cepstrum by paraphrasing the term "spectrum", because cepstrum techniques perform a further spectral analysis on a frequency spectrum (4). Using the same naming convention, terms such as "quefrency" and "liftering" correspond to frequency and filtering in the cepstral domain (6:1429). Two versions of the cepstrum are used in signal processing applications; the complex cepstrum and the power cepstrum.

The power cepstrum is defined as: "the power spectrum of the logarithm of the power spectrum" (27). This is denoted as:

$$c_l = |\mathcal{F}[\log(|X(f)|^2)]|^2 \tag{2.1}$$

where $|X(f)|^2$ is the power spectrum of the signal and $\mathcal{F}$ is the Fourier transform. Note that the final squaring operation is really unnecessary and is often not used (27). The independent variable of the cepstrum is termed "quefrency" (4), which is a time signal in terms of frequency (units of time which are treated like frequency). The Fourier transform of a spectrum yields time on the independent axis. A large time value (high quefrency) represents a rapid fluctuation in the spectrum and a low quefrency represents a slow fluctuation (27). Thus the cepstrum can space apart these fluctuations in the the frequency of the signal and ultimately separate two convolved signals in this quefrency domain.

Speech processing has made great use of the power cepstrum's ability to separate signals. Voiced speech is made up of two distinct signals; the voice pitch (p(x)) and the sound of the word from the vocal tract (v(x)). These two signals are convolved together to make coherent speech; $s(x) = p(x) * v(x)$ (23). Because of the convolution property of the Fourier transform, the power spectrum of the signal changes the convolution into multiplication (6):

$$|S(f)|^2 = |P(f)|^2 \cdot |V(f)|^2 \tag{2.2}$$

The logarithm now separates the signal into two added terms (6):

$$\log|S(f)|^2 = \log|P(f)|^2 + \log|V(f)|^2 \tag{2.3}$$

The power cepstrum of the input signal is now the sum of the power cepstrum of the two convolved signals. The two sums may be easily separated by filtering if they occupy different quefrency ranges (27). Pitch and vocal tract signals generally do have different ranges in quefrency (23). The pitch of the voice fluctuates very quickly in the spectral domain of speech - therefore it has a large spike at a high quefrency. The vocal tract is made up of the lips, mouth, and tongue. These can not change nearly as quickly as the vocal cords can, therefore the vocal tract fluctuates slowly in the frequency domain and has a spike in the low quefrency region. These spikes are called formant peaks and have a number of uses in speech processing (6). Speech identification systems do not use pitch information; male and female speech can be made to look the same by eliminating the formant corresponding to the pitch (2). The cepstrum is used *to* find the formants of the pitch so that this information can be removed from the speech signal. The biggest problem with the power cepstrum is that many signals can not be directly restored because the phase information has been lost.

The separated signals could be reconstructed if the signal's phase information were maintained. The complex cepstrum accomplishes this by using the complete spectrum of the signal instead of simply the power spectrum (10). The complex cepstrum is defined as the inverse Fourier transform of the logarithm of the spectrum of a signal (10). This can be shown in equation form as:

$$c_l = \mathcal{F}^{-1} log[S(f)] = \frac{1}{2\pi}\int_{-\pi}^{\pi} \log[S(f)] e^{j2\pi f} df \tag{2.4}$$

The complex cepstrum retains the phase information of the original signals by taking the complex logarithm of the spectrum. The natural logarithm of a complex value is equivalent to the natural logarithm of the magnitude plus the phase of the signal (22:769) as shown below:

$$\ln[X(f)] = \ln\left[|X(f)|\, e^{j\angle X(f)}\right] \tag{2.5}$$

$$= \ln|X(f)| + j\angle X(f) \tag{2.6}$$

The complex cepstrum is a complex function, as the name implies. However, all complex cepstral coefficients will be real, if the original signal is real (22). The real coefficients result due to the fact that a real signal's spectrum will have phase with odd symmetry and the resulting inverse Fourier transform will yield real coefficients. Because it keeps all phase and magnitude information, the complex cepstrum is reversible – it is possible to return to the original signal after performing filtering operations. Signals may not only be separated, like with the power cepstrum, but the signal may also be fully reconstructed. Parts of the signal may be completely removed with the complex cepstrum. This separation and removal is called deconvolution. A transmitted signal may be separated into its original signal plus effects of transmission. These effects may then be filtered and removed and the original signal restored (27).

The complex cepstrum is complicated by ambiguities in the phase angle, $\angle X(e^{jw})$ (22:775). At any given value of $w$, integer multiples of $2\pi + w$ will yield the same phase value. This leads to the complex logarithm being multi-valued. The phase of the signal will have discontinuities because each point can take on any value modulo $2\pi$ of its principle value. There have been many algorithms defined to solve this phase unwrapping problem (6):

1. A correction sequence method, where a correction is calculated and added to the modulo $2\pi$ phase.
2. Integrating the phase derivative .
3. An adaptive numerical integration procedure.
4. A recursive procedure to remove the linear phase.

Note that for minimum phase signals (no poles or zeros outside the unit circle), phase unwrapping is not necessary. In this case the complex cepstrum will be the same as the power cepstrum.

The problems encountered and the additional computational complexity are a significant drawback to the use of the complex cepstrum. Because of the laborious mathematics required for these phase unwrapping techniques, there have been a number of alternative forms for calculating the complex cepstrum without dealing with the phase information. The differential cepstrum as described by Polydoros and Fam (24) takes the Fourier transform, then finds a normalized derivative of the transform coefficients, takes the inverse Fourier, and finally the logarithm to get the differential cepstral coefficients. In another approach, Bednar and Watt estimate the derivative of the log-amplitude as well as the phase spectrum and then perform a frequency domain integration by dividing in time (3). This is similar to the integration method of Oppenheim and Schaefer (22), however Bednar and Watt avoid any numerical integration or phase unwrapping to obtain $\hat{x}(n)$ from $x(n)$. Cepstral coefficients have also been calculated entirely in the time domain in order to avoid the phase, windowing, and aliasing problems of the Fourier transform (30). This time domain cepstral transform is actually more computationally intensive than the Fourier based cepstrum, but it can yield better results (30). It has also been shown that homomorphic filtering can be accomplished without directly calculating cepstral coefficients (11). Khare and Yoshikawa demonstrated a relationship between the moments of a signal sequence and of its corresponding cepstral sequence.

## *Transforms*

Transforms are composed of matrices, $\mathbf{A}$, which calculate coefficients from an input signal. These coefficients can be easily processed and transmitted. At a receiver, the signal is reconstructed by performing the inverse transform, $\mathbf{A}^{-1}$, and putting the blocks back together (16). One of the most common transforms is the block transform. The block transform simply divides a signal into equal blocks and transforms each block into a group of coefficients. Each block of length N is changed by the transform matrix, **A** (N×N matrix) so that:

$$x(n) = [x(nN),\ x(nN-1) \ldots x(nN-N+1)]^T \tag{2.7}$$

This transformation can also be written as a general linear equation: $\mathbf{X} = \mathbf{A}^T\mathbf{x}$ (16). The matrix **X** is called the transform of the given signal **x**. One goal, when using transform coding, is to reduce the number of bits per second required to transmit a given signal. This is accomplished through careful selection of **A**. When the transform matrix is orthogonal, the transpose of **A** equals the inverse of **A**. We can then easily find the given signal by the relation: $\mathbf{x} = \mathbf{AX}$ (16). The columns of the transform matrix, **A**, are called the basis functions (or basis vectors) because they are used to transform one element of **x** to a corresponding element of **X**. These basis functions, $a_n$, are what differentiate block transforms from each other. Orthogonal transform matrices provide us with the ability to easily find the forward and inverse transform, because we do not need to actually calculate the inverse of the matrix (16).

The discrete Fourier transform (DFT) and the discrete cosine transform (DCT) are two of the best known and most widely used block transforms. The DFT has basis functions that are complex sinusoids (28):

$$a_{n,k} = \sqrt{\frac{1}{N}} exp\left(j\frac{2\pi kn}{N}\right) \tag{2.8}$$

where N is the block size. Like the Fourier series, the DFT represents the signal, x[n], as a combination of harmonically-related sinusoids (16). The discrete cosine transform (DCT) provides N different frequencies between 0 and $\pi$, where the DFT has only N/2 frequencies. The DCT basis is defined as (28):

$$a_{n,k} = c(k)\sqrt{\frac{2}{N}} cos\left[\left(n+\frac{1}{2}\right)\frac{k\pi}{N}\right] \tag{2.9}$$

where c(k) is $\frac{1}{\sqrt{2}}$ when k = 0 and is equal to one otherwise. The DCT is so common in signal processing that it has become the standard for the video-phone and other image coding applications (28).

*Performance*

Block transforms encounter a problem called the blocking effect, which occurs when signals are reconstructed. Blocking effects arise due to the lack of overlap between blocks of the signal. These effects are noticed as discontinuities along the boundaries of an image or as extraneous tones in a speech signal. These effects become even more pronounced at low data rates (37). The DFT and DCT lead to blocking effects because of the independent processing of each block. The final segments of a prior block will generally not match with the first samples of the next block, because the basis functions (as defined above) vary from one block to the next (16). Two methods have been proposed to reduce blocking effects: overlapping and filtering. Overlapping consists of blocks of samples which simply overlap some predetermined amount. In this way, the information around the boundaries is transmitted twice and the receiver averages the data in the overlapped areas. Unfortunately, overlapping increases the bit rate because extra samples must be sent. For filtering, a low pass filter is added at the receiver to filter the boundary pixels. Although this does not increase the bit rate, it tends to blur the signal across the boundaries (15).

Malvar explained that performance of a multirate filter bank (a transform coding application) is directly related to; "perfect reconstruction, high stopband attenuation, narrow transition width, high coding gain, absence of blocking effects, and fast algorithms" (15). A compromise must be reached between these conflicting goals. Two methods of coding are often used: transform and subband coding. Subband coding yields filters of high performance (ie high stopband attenuation and narrow transition widths) but they are also of high complexity (longer, slower algorithms). Transform coding yields a faster process but poorer filters, which lead to the blocking effects.

## *The Lapped Transform (Malvar Wavelet)*

The lapped transform (or Malvar wavelet) provides an excellent tradeoff between complexity and filter performance (15). The lapped transform has the same benefits as the overlapping method, but without the increase in the bit rate (18). The LT eliminates the blocking effects by overlapping adjacent blocks of the signal, hence the term "lapped" transform.

First let the incoming signal, x[n], be composed of MN samples, where M is the block size. Traditional overlapping would transform this signal into N blocks each of length M. The lapped transform will instead transform the signal into N blocks of length L, where L > M, so that adjoining blocks will overlap by L - M samples (note that traditionally L $\leq$ 2M). The basis functions have been made longer than the block length, which allows for a smoother transition at the ends of the blocks (see Figure 2.2) (16).

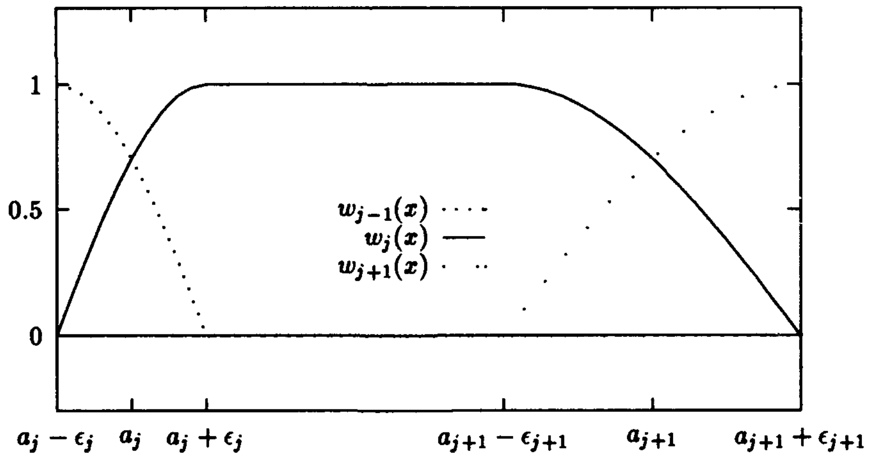


Figure 2.2. Overlapping of Basis Functions

This lapped transform method closely resembles the basic method of overlapping. The fundamental difference between the two is that the LT maps the L samples of each block back into M transform coefficients. Therefore, there is no increase in the data rate when using the LT, because the number of coefficients are now equal to the original block size again (18). The LT can also be designed as a perfect reconstruction process -- if all of the transform coefficients are used at the receiver, the reconstructed signal will be an exact copy of the original signal (with the possibility of a pure delay) (37). In order to achieve perfect reconstruction, the transform matrices must obey the following properties (37):

$$\mathbf{A}\mathbf{A}^T = I \; and \; \mathbf{A}^T\mathbf{A} = I \tag{2.10}$$

where I is the identity matrix. This is the same as saying that the basis functions in **A** must be orthogonal to each other. Because the basis functions overlap, they must also be orthogonal to the basis functions in the neighboring blocks. The LT is often called the lapped orthogonal transform (LOT), due to this orthogonality requirement (16).

The LOT is a better alternative than the conventional block transforms for signal coding applications because it eliminates the blocking effects without increasing the data rate. The price paid is the increase in computational complexity of the transform algorithm (1). Transform coding generally employs the discrete cosine transform (DCT) because of its close approximation to the statistically optimal Karhunen-Loeve transform (14). The LOT actually may require as much as 30 percent more computations (mainly additions) than the DCT. However, the LOT leads to slightly smaller reconstruction errors than does the DCT (18). There have been a number of variations and extensions to the theory of lapped transforms since the introduction of the first LOT by Cassereau. Some of the other types of LOTs which have been characterized include: (i) Modulated Lapped Transform (MLT), which is based on Quadrature Mirror Filter Banks (14); (ii) Extended Lapped Transform (ELT), which is an MLT utilizing a larger basis function (15); (iii) The LOT then can be generalized with any orthogonal bases desired (1). The LOT, although not a wavelet in the traditional sense, compliments wavelet packet analysis, as such, it has been termed the *Malvar wavelet* by Yves Meyer (19).

*The Malvar Wavelet as a Multirate Filterbank*

Transforms can also be represented as a series of filterbanks (37:113). "A digital filter bank is a collection of digital filters with a common input" (37:113). The Fourier transform can be shown as in figure 2.3. The analysis bank splits the signal, $x[n]$, into M subband signals, $x_k[n]$, where

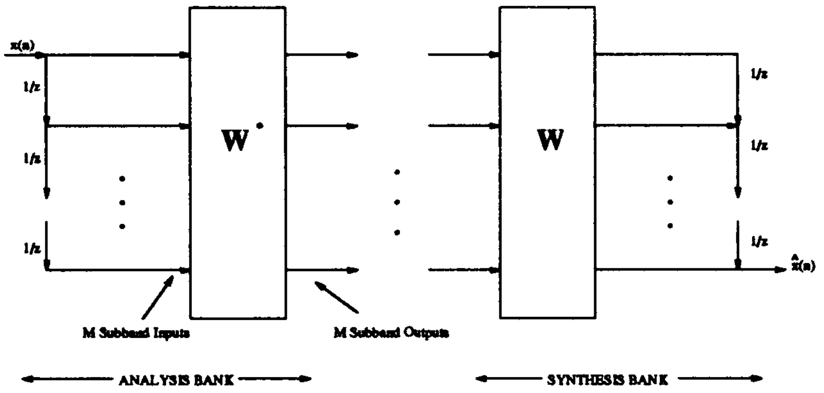


Figure 2.3. The DFT Represented in Multirate Form

$0 \leq k \leq M - 1$. The synthesis filters then recombine the M subband signals into a single signal, $\hat{x}[n]$. The Discrete Fourier Transform (DFT) is defined as (37:794):

$$X(k) = \sum_{n=0}^{M-1} x[n] W_M^{kn} \tag{2.11}$$

where $W_M = e^{-j2\pi/M}$. The analysis and synthesis banks of the DFT are therefore constructed of the matrix **W** which has M×M elements.

The Malvar wavelet may be represented as a multirate filterbank where the matrix **T** is now MxL (rather than MxM, where L > M) as shown in figure 2.4 (37:325). This means that there are more input subbands to the filter **T** than output subbands. The Malvar wavelet, as stated previously, will take the L samples of each block and map them onto M transform coefficients. The $\downarrow$M (in figure 2.4) represents a decimator. A decimator (or downsampler) retains only those samples of $x[n]$ which occur at time equal to multiples of M (37:100). For example, if M=2 then

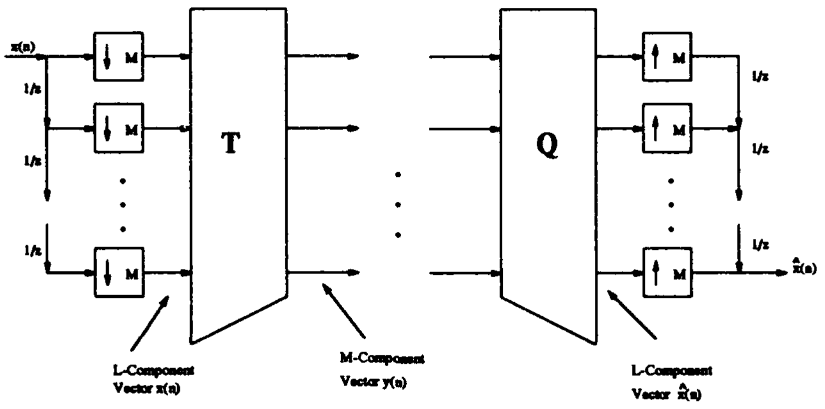


Figure 2.4. The Malvar Wavelet Represented in Polyphase Form

the signal would be broken into its even and odd components. The $\uparrow$M then puts the samples back into their proper place in the data stream when the signal is reconstructed at the receiver.

By downsampling prior to transmission, the computation rate through a communications satellite can be reduced. Therefore slower, cheaper, more efficient, and more reliable elements can be used on-board the satellite (32). This, in turn, would mean that the lose of an element would have less of an effect on the mission.

## *Generalized Malvar Wavelet*

Windowing, often employed in filtering and spectral estimation, can result in irreparable distortions in the transformed signal. By utilizing the Malvar wavelet, any signal distortion resulting from the transformation can be eliminated at reconstruction. Suter and Oxley have presented a Generalized Malvar wavelet (GMW) which places conditions on the windows and orthonormal basis and allows for perfect reconstruction (34). The basis is defined by performing an odd and even extension of the orthogonal basis function, $f_{j,k}(x)$, about the end points of the window such that:

$$\tilde{f}_{j,k}(x) = \begin{cases} 0 & , \quad -\infty < x \leq a_j - \epsilon_j \\ -f_{j,k}(2a_j - x) & , \quad a_j - \epsilon_j < x < a_j \\ f_{j,k}(x) & , \quad a_j \leq x \leq a_{j+1} \\ f_{j,k}(2a_{j+1} - x) & , \quad a_{j+1} < x < a_{j+1} + \epsilon_{j+1} \\ 0 & , \quad a_{j+1} + \epsilon_{j+1} \leq x < \infty. \end{cases} \tag{2.12}$$

where $a_j$ and $a_{j+1}$ are the end points of the frame and $\epsilon_j$ and $\epsilon_{j+1}$ are the overlaps about the two points. An associated weight function, $\tilde{p}_j(x)$, is then used to ensure perfect reconstruction is possible. All of the basis functions investigated in this thesis do not require a weight function; let $\tilde{p}_j(x) = 1$. The normalized window of the GMW is chosen such that (8):

$$\begin{aligned} w_j(x) = 1 \quad & for \quad a_j + \epsilon_j \leq x \leq a_{j+1} - \epsilon_{j+1} \\ w_j(x) = 0 \quad & for \quad a_j - \epsilon_j > x > a_{j+1} + \epsilon_{j+1} \\ w_j(a_j - \sigma) = w_{j-1}(a_j + \sigma) \quad & for \quad \sigma \in [-\epsilon_j, \epsilon_j] \\ w_j^2(x) + w_{j-1}^2(x) = 1 \quad & for \quad a_j - \epsilon_j \leq x \leq a_j + \epsilon_j \end{aligned} \tag{2.13}$$

The Generalized Malvar wavelet (generalized lapped orthogonal transform) is thus a lapped transform which allows for many variations within the same transform. Most transforms, like the DFT or DCT, maintain the same block length and basis function throughout the data set. Suter and Oxley derived the GMW based on being able to vary the:

1. Lengths of the Windows.
2. Orthonormal Basis Functions (**A**).
3. Overlap Between Adjacent Windows.
4. Window Amplitudes.

The goal of the generalized Malvar wavelet, like all transform coding, is to determine a set of coefficients which may then be used to reconstruct the original data. The benefit of being able to vary the parts of the transform is that the transform may be tailored to the data set. For example, the number of coefficients required to model a slowly varying signal can be reduced by increasing the length of the window (32).

Two Malvar wavelets were studied for this thesis effort. A Complex Valued Malvar wavelet, which is very similar to the short time Fourier transform because it utilizes an exponential basis function (32):

$$f(x) = e^{j2\pi m x} \tag{2.14}$$

With the weight function , $p(x)$, equal to one. The Complex Malvar wavelet also keeps a constant length window throughout the data set as well as constant amounts of overlap between the windows. The second transform is a Generalized Real Valued Malvar wavelet. This wavelet uses a sinusoidal basis function:

$$f_k(x) = \sqrt{\frac{2}{N}} \sin\left(\pi(k + 1/2)\left[\frac{x}{N}\right]\right) \tag{2.15}$$

with $p(x) = 1$. This transform is a generalized wavelet because it may use variable length windows, and variable amounts of overlap within any given data set (34).

# *III. Complex-Valued Malvar Wavelets*

## *Introduction*

The Balian-Low Theorem states that, if a complex-valued exponential basis is windowed, the resulting transform will not be localized in time or not be localized in frequency. As a result, orthogonality of the transform can not be maintained. This represents a significant constraint to signal processing because of the central role of the Short Time Fourier Transform. Although this represents a "no-go" condition, by changing the underlying assumptions a positive result is possible. Namely, if the functions have extensions outside the interval and windows are defined with constraints, an orthogonal windowed transform is possible.

Suter and Oxley have formulated an orthogonal Malvar wavelet that has a complex-valued exponential basis (33). Moreover, the resulting algorithm has the same "Big Oh" complexity as the conventional Fast Fourier Transform (FFT).

## *Complex-Valued Malvar Wavelet Algorithm*

Suter and Oxley (33) generate the complex-valued Malvar wavelet (CMW) with eight primary steps which are summarized below:

(1) Define a Basis: An exponential-like basis can be applied to the Malvar wavelet and complete orthogonality will be preserved, even after windowing. The basis function is first defined as the even and odd extensions of an exponential function:

$$\tilde{f}_{n,m}(x) = \begin{cases} 0 & , \quad -\infty < x \leq n - \epsilon \\ e^{-j2\pi mx} & , \quad n - \epsilon < x < n \\ e^{j2\pi mx} & , \quad n \leq x < (n+1) \\ -e^{-j2\pi mx} & , \quad (n+1) < x < (n+1) + \epsilon \\ 0 & , \quad (n+1) + \epsilon \leq x < \infty. \end{cases} \tag{3.1}$$

Where $[n, n+1]$ is the $n^{th}$ interval, and $\epsilon$ is the amount of overlap between windows. The basis function definition is equivalent to even and odd extending both the real and imaginary parts of the function beyond (n) and (n+1). The basis functions and their extensions are shown in figures 3.1a and b. Note that the values of n and n+1, in the figure and the calculations, have been normalized to zero and one.

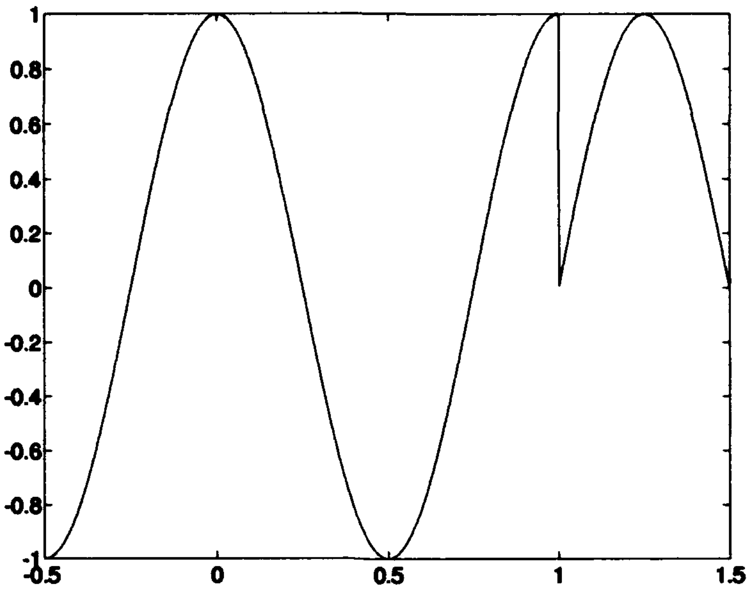
(2) Define a Window: The transform will have a single window function that will be used over each interval. The window, $g(x)$ (with $n = 0$), is defined as:

$$g(x) = \begin{cases} 0 & , \quad -\infty < x < 0 - \epsilon \\ \sin(\frac{\pi}{4\epsilon}\{x + \epsilon\}) & , \quad 0 - \epsilon \le x < 0 + \epsilon \\ 1 & , \quad 0 + \epsilon \le x \le (0+1) - \epsilon \\ \cos(\frac{\pi}{4\epsilon}\{x - (1) + \epsilon\}) & , \quad (0+1) - \epsilon < x < (0+1) + \epsilon \\ 0 & , \quad (0+1) + \epsilon \le x < \infty. \end{cases} \tag{3.2}$$

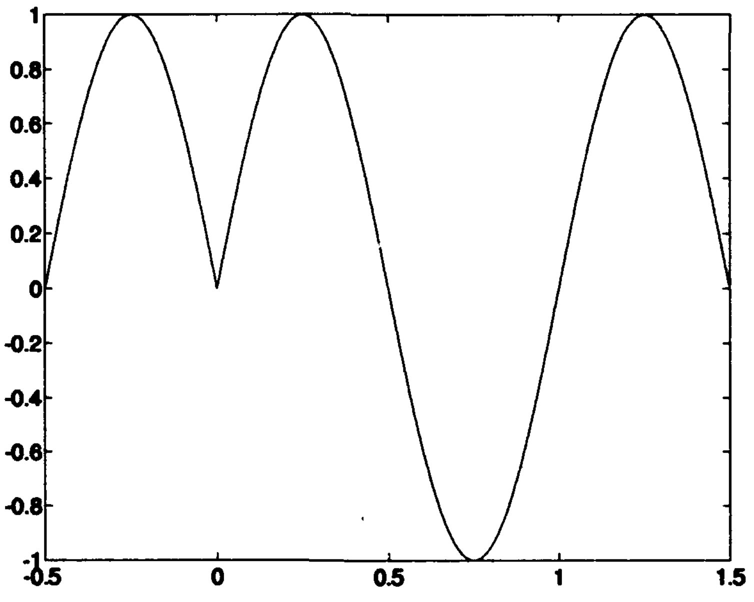
Now the basis is equal to $u_{n,m} = \tilde{f}_{n,m}(x)g(x-n)$. This is an orthonormal basis set for $L^2(\Re)$.

(3) Multiply By the Window and Define the Coefficients, $\alpha$: The complex Malvar wavelet coefficients are calculated by:

$$\begin{aligned} \alpha_{n,m} &= \langle s, u_{n,m} \rangle \\ &= \int_{n-\epsilon}^{n+1+\epsilon} s(x)g(x-n)\tilde{f}_{n,m}(x)dx \\ &= \int_{n}^{n+1} h_n(x)\, e^{j2\pi mx} dx \end{aligned} \tag{3.3}$$



(a)



(b)

Figure 3.1. (a) Real Part and (b) Imaginary Part of the Complex-Valued Basis Function Plotted Over The Extended Interval. Let 0 = (n) and 1 = (n+1).

where $s(x)$ is the input signal and $g(x)$ and $\tilde{f}(x)$ are as defined above and $h_n$ is defined as:

$$h_n(x) = \begin{cases} s(x)g(x-n) + s(2n-x)g(n-x) & , \quad n \le x \le n+\epsilon \\ s(x) & , \quad n+\epsilon < x < (n+1)-\epsilon \\ s(x)g(x-n) - s(2(n+1)-x)g(n+2-x) & , \quad (n+1)-\epsilon \le x \le (n+1) \end{cases} \tag{3.4}$$

(4) Define a New Array: Let $H_{n,l} = h_n(n+\delta l)$. Where, $H_{n,0} = h_n(n)$ denotes the Folded Data at the start of the interval, and $H_{n,N} = h_n(n+1)$ denotes the Folded Data at the end of the interval. The trapezoidal rule can then be used to approximate the integration:

$$\alpha_{n,m} \approx \alpha^T_{n,m} = \frac{1}{2}H_{n,0} + \sum_{l=1}^{N-1} H_{n,l}e^{j2\pi ml/N} + \frac{1}{2}H_{n,N} \tag{3.5}$$

From the $h_n$ equations, it is clear that $H_{n,N}$ will equal zero. Now let:

$$\beta_{n,l} = \begin{cases} H_{n,0} & , \quad l = 0 \\ 2H_{n,l} & , \quad 1 \le l \le N-1 \end{cases} \tag{3.6}$$

(5) Perform an Inverse FFT: The summation from step 4 thus becomes an N point inverse Fourier transform of $\beta$. The spectral coefficients, $\alpha$, are calculated with a Fourier transform of the windowed and scaled input signal:

$$\begin{aligned} \alpha_{n,m} &= \frac{1}{2N}\sum_{l=0}^{N-1} \beta_{n,l}e^{j2\pi ml/N} \\ &= \frac{1}{2}\mathcal{F}^{-1}[\beta] \end{aligned} \tag{3.7}$$

The Reconstruction of the signal is accomplished by reversing the above steps:

(1) Perform a Forward FFT: The signal may now be reconstructed, given the coefficients $(\alpha_{n,m})$.

$$\beta_{n,l} = 2 \cdot \mathcal{F}[\alpha_{n,m}] \tag{3.8}$$

(2) Scale the Values: Scaled the $\beta$ values back to $H_{n,L}$:

$$H_{n,l} = \begin{cases} \beta_{n,0} & , \quad l = 0 \\ (\frac{1}{2})\beta_{n,l} & , \quad 1 \leq l \leq N-1 \end{cases} \tag{3.9}$$

(3) Divide By the Window: Finally, the reconstructed signal, $S_{n,l}$, is calculated by dividing back out the original window, $g(x)$:

$$S_{n,l} = \begin{cases} \frac{H_{n,0}}{2g(0)} & , \quad x = n \\ g(x-n)H_{n,l} - g(n-x)H_{n-1,N-l} & , \quad n < x < n+\epsilon \\ H_{n,l} & , \quad n+\epsilon \leq x \leq (n+1)-\epsilon \\ g(x-n)H_{n,l} + g((n+2)-x)H_{n+1,N-l} & , \quad (n+1)-\epsilon < x < n+1 \end{cases} \tag{3.10}$$

*Software Implementation*

It is important that follow-on students be able to understand, utilize, and modify the Malvar wavelet programs. The Ada programming language is well known for being easily modified and maintained. Therefore, the complex Malvar wavelet was implemented using Ada. The complex transform program will correspond to the sinusoidal Malvar wavelet Ada program written by Raduenz (26). Thus all of the Malvar code will be in the same language and be easily compared and contrasted. This section will step through each package of the wavelet program to outline what it is doing and how the package relates to the rest of the code. A complete listing of the source code is listed in Appendix B.

*Get_File_Data:* The *Get_File_Data* procedure simply reads in the required input files and then prepares the data vectors to be used by the rest of the program. The user must give the name of an input file to be used by this procedure. This input file must contain the following:

1. The filename containing the data set to be processed.
2. The filename for the spectral coefficient data.
3. The filename for the reconstructed data set.
4. The number of data points within the data set (or number of points wished to be processed).
5. The window size to be used (this must be an integer multiple of the size of the data set).
6. The amount of overlap to be used between intervals.

An example input file is shown in figure 3.2. It reports a data file called "input.dat" and requests the spectral coefficients to be placed in "alpha.dat" and the reconstructed data set into the file called "output.dat". The file is 1000 points long (or just want to process 1000 points of a larger data set). The window size is 200 points. Therefore there will be 5 windows of data: {(0,200), (200,400), $\cdots$, (900,1000)}. Now the first point (0) and the last point (1000) are actually the same partition point. There is no data which resides in point zero; it is used only as a place holder for the overlap region. The first window will overlap with the last window and the last will overlap with the first. This overlap has been accomplished without copying any data. This initial overlap could be modified so that the first window would overlap with itself, thus allowing for a continuous stream of data. The program calls in the correct data positions to accomplish the folding and unfolding of all of the windows, including the first and the last. Finally, the last value tells us that there are 10 points of overlap between intervals. The partitions are established after all of the data is read into the program. The program determines where all of the intervals are going to start and stop. These correspond to the positions; {N(n), N(n+1), N(n+2), $\cdots$ }. The last thing *Get_File_Data* does is to read in the actual data set into a vector called "Data(j)".

*Multiply_By_Windows:* This procedure performs the "folding" of the data by multiplying the data by the window as defined by equation 3.4. The window and data values are passed to this

```
input.dat
alpha.dat
output.dat
1000
200
10
```

Figure 3.2. Example Input File

procedure from the master package, *Do_Work*. The procedure steps through equation 3.4 starting with the left edge of the interval, then the center, and finally the right edge. There are unique right edge equations for the first window. This is due to the fact that data must be pulled from the other end of the data file (and not just the adjoining interval). For the same reasons, there are special equations for the left edge when using the last interval. The folded data is then sent back to the master package, *Do_Work*.

*Compute_Coefficients:* This procedure receives the folded data in segments (from N(n) to N(n+1)-1, or 0 to N-1). It first scales the coefficients as in equation 3.6 and converts the data into complex notation. It then sends the complex data to the FFT package to perform an N point (size of the interval) inverse fast Fourier transform. The output of the FFT is then scaled by $\frac{1}{2}$ per the algorithm. This complex data (form 0 to N-1) is then sent back to *Do_Work*.

*Reconstruction_FFT:* The reconstruction FFT package simply takes in the N points passed from the *Compute_Coefficients* and sends it to the FFT package. Within the FFT package, an N point forward FFT is performed. The very first coefficient is then doubled because of the way $\beta$ was defined (see equation 3.6) and the entire segment is sent back to *Do_Work*.

*Divide_By_Windows:* *Divide_By_Windows* calculates the reconstructed signal as defined in equations 3.10. As shown in the algorithm, this package calculates the segment from $(n - \epsilon)$ to $((n + 1) + \epsilon)$. If the regular interval, from n to (n+1), were calculated then coefficients from the previous interval would be altered prior to being used to calculate the values for the next interval. As in *Multiply_By_Windows*, there are special equations for the first window. The first window will be pulling data from the end of the data file. The last window does not need any special equations because we only calculate up to the end of file minus the overlap. These "missing" points (from the end of file - overlap to the end of file) have been calculated with the first window and are located from point (0) to $(0 - \epsilon)$.

*Error_Out:* This procedure calculates the reconstruction errors as defined later in this chapter. It receives the reconstructed and original data sets and calculates the three numerical error criteria:

1. Relative $L_2$ Error based on equation 3.11.
2. $L_\infty$ Error based on equation 3.12.
3. Root Mean Square Error based on equation 3.13.
4. Normalized Rényi Information Value based on equation 3.15.

*Do_Work:* This package controls the entire wavelet procedure. This procedure first calculates the window based upon the set of equations 3.2. This window is normalized from zero to one and is used for all folding and unfolding of the data set. Each data segment is processed in the main loop of this procedure. The main loop is defined from (0) to (Partitions'last-1) this means that the program will start with the first window (point 0 to the first partition) and end with the last window (the last partition - 1 to the last partition, which is the end of the file). In this way, the finite length signal can be calculated without copying data points or processing segments multiple times.

*Do_Work* calls each procedure as it is needed by the algorithm. This main procedure sends all of the necessary data points and receives the transformed data back from the procedures. The main loop is started over prior to the *Divide_By_Windows* procedure, because the reconstruction of the first window of the data set requires the transformed data from the end of the file. After all steps of the wavelet are complete, *Do_Work* shifts the first $\epsilon$ values of the reconstructed data to the end of the reconstructed data file. This is done because *Divide_By_Windows* calculates the data based on shifted windows ($n - \epsilon < x \leq (n+1) - \epsilon$ instead of $n < x \leq (n+1)$). *Do_Work* then prints all of the information out to the requested data files.

*Called Packages* The complex Malvar wavelet program references multiple outside called packages, including:

1. Text_Io
2. Complex_Pkg
3. Math_Lib
4. Vector_Package
5. Type_Package
6. Print_Package
7. FFT_Package

These packages are discussed and documented by Raduenz (26:4-10) and are quite integral to the workings of all of the wavelet programs.

*Error Criteria*

When testing and evaluating speech coding systems, one is usually interested in two measures - quality and intelligibility. These two measures are closely related but not identical. If the quality of a signal is high then the intelligibility will also be high. However, the opposite is not true, it is possible to have very intelligible speech that is not of high quality. Speech synthesis systems produce speech that is readily understandable, but they have a definite synthetic sound which degrades their quality. Both quality and intelligibility are difficult to quantify. Coders are not normally tested using numerical methods but with a group of listeners. The listeners indicate which rhyming word from a given set was perceived. The number of words correctly perceived provides a score that indicates the intelligibility of the system.

Numerical error criteria were utilized to determine the basic capabilities of each system. These error criteria were very useful when simple sine waves and other known signals were input into the systems. Six different criteria were used; four of these are numerical and two are more subjective in nature.

The relative $L_2$ measure is the error of the $L_2$ norm of a signal (approximated by n coefficients) relative to the $L_2$ norm of the original signal (using all coefficients) (32). The $L_\infty$ criteria is a measure of the point-wise peak error; the point-wise maximum difference between two signals (32). Coifman used the relative $L_2$ error criteria when he researched the link between compression of a signal and the basis-set of a given window (7). The $L_2$ and $L_\infty$ are defined as:

$$L_2 = \sqrt{\frac{|\sum_x actual^2(x) - \sum_x recon^2(x)|}{\sum_x actual^2(x)}} \tag{3.11}$$

$$L_\infty = Max\left[|actual(x) - recon(x)|\right] \tag{3.12}$$

The normalized root mean square (RMS) criteria is the basic error criteria that compares the point-wise difference between two signals. It relates the total power difference between the two signals. The normalized RMS is defined as:

$$RMS = \frac{1}{N}\sqrt{\sum_x (actual(x) - recon(x))^2} \tag{3.13}$$

The final numerical error criteria used is the Rényi Information Value (RIV). Information measures are normally used to describe the distributions of random variables (40:146). However this measure has the potential to give some real insight as to how similar two signals are. The Rényi value relates the total amount of "information" residing in the signal. When a signal is transformed, compressed, and reconstructed, it should loose some of the original signals information. The general definition of the $\alpha^{th}$ order Rényi information value for a discrete function, p, is (40:147):

$$R^{\alpha}(p) = \frac{1}{1-\alpha}\log_2 \sum_x p^{\alpha}(x) \tag{3.14}$$

where $0 < \alpha < 1$ or $1 < \alpha < \infty$. In order to obtain a relation between the original signals information and the reconstructed signals information, the second order Rényi value of the reconstructed signal was divided by the second order Rényi value of the original signal (32). Using this method, a value of 1.0 would represent perfect reconstruction (with respect to the amount of information present) and 0.0 would represent a total loss of the signal (100% compression). The Rényi measure will take the form:

$$\hat{R}^2(actual, recon) = \frac{\left[\log_2 \sum_x recon^2(x)\right]}{\left[\log_2 \sum_y actual^2(y)\right]} \tag{3.15}$$

The final two error tests are a more subjective criteria, but may have been the most important tests of all. First, the reconstructed signals were plotted and visually compared to the original signal. The plots gave a great amount of insight into the amount of noise in the reconstructed signal and how much information was lost. Then, in the case of speech signals, the reconstructed signal was played back via Entropics Waves+. This is the ultimate test for a digitized speech signal. A signal can have the lowest reconstruction error possible, but if it does not sound correct, then the reconstruction has been poor.

## *Validation*

The program was debugged with two simple input files; a unit step and a sine wave. Once these were reconstructed correctly, the size of the data file, size of the windows, and the amount of overlap were varied to ensure that all of these combinations yielded perfect reconstruction. More complex signals, like speech, were used as the final test of the program's reconstruction abilities.

Table 3.1 presents the reconstruction errors associated with the three signals; unit step, sinusoid, and speech. These signals were broken-down and reconstructed using a sample size of 2000, with 250 point windows and 10 points of overlap between windows. The last entry on the table is for an entire sentence of speech (28250 points at 8KHz), with 250 point windows and 10 point overlap. The table shows that each of the four signals were reconstructed to within the machine round-off of the original data set. Note that the RIV error is plotted as 1.0 plus the given value in the table. This was done because all of the RIV outputs were 0.999· · ·999, which does not help to distinguish the errors. The error rises slight as the signals get more complicated and as the amplitudes increase. The worst RMS error was the utterance of the word "dark." This is a relatively short, high amplitude, and quickly varying signal that was still reconstructed perfectly (RMS error = $4.216x10^{-10}$). The errors introduced when reconstructing an entire sentence were lower because the numerical error was spread out and averaged across a much larger number of samples.

| *Input Data* | *Relative* $L_2$ | $L_\infty$ | *RMS* | *RIV (1.0 + )* |
|---|---|---|---|---|
| Unit Step | $2.231x10^{-8}$ | $1.877x10^{-10}$ | $5.060x10^{-13}$ | $-0.066x10^{-12}$ |
| Sinusoid | $2.237x10^{-6}$ | $1.876x10^{-10}$ | $3.640x10^{-13}$ | $-0.072x10^{-12}$ |
| Utterance of "DARK" | $2.861x10^{-6}$ | $4.798x10^{-7}$ | $4.216x10^{-10}$ | $-0.392x10^{-12}$ |
| Speech Sentence | $1.051x10^{-6}$ | $4.489x10^{-7}$ | $5.251x10^{-11}$ | $-0.050x10^{-12}$ |

Table 3.1. Output Errors of the CMW Program with Different Input Signals

Table 3.2 shows how the error changes with increasing overlap. The signal used was the 2000 point unit step with 250 point windows. Early runs showed that the majority of the error is found in the overlap regions. This overlap error is provided in table 3.3, which shows the given reconstruction around an end point (N) of a unit step function. Thus, for signals that are not compressed, as the amount of overlap increases so will the calculation error.

The Malvar wavelet program not only provide very low numerical errors but also the graphic displays and the audio reconstructions were perfect. The reconstruction of the unit step and the

| *Amount Overlap* | *Relative $L_2$* | $L_\infty$ | *RMS* | *RIV (1.0 + )* |
|---|---|---|---|---|
| 1 point | $2.384x10^{-8}$ | $2.442x10^{-14}$ | $1.807x10^{-16}$ | $0.000x10^{-12}$ |
| 5 % | $2.482x10^{-6}$ | $1.908x10^{-10}$ | $5.628x10^{-13}$ | $-0.081x10^{-12}$ |
| 10% | $3.728x10^{-6}$ | $1.984x10^{-10}$ | $8.446x10^{-13}$ | $-0.183x10^{-12}$ |
| 20% | $5.367x10^{-6}$ | $2.017x10^{-10}$ | $1.215x10^{-12}$ | $-0.379x10^{-12}$ |
| 30% | $6.611x10^{-6}$ | $2.028x10^{-10}$ | $1.497x10^{-12}$ | $-0.575x10^{-12}$ |
| 40% | $7.656x10^{-6}$ | $2.034x10^{-10}$ | $1.734x10^{-12}$ | $-0.771x10^{-12}$ |
| 50% | $8.575x10^{-6}$ | $2.037x10^{-10}$ | $1.942x10^{-12}$ | $-0.968x10^{-12}$ |

Table 3.2. Output Errors of the CMW Program with the Unit Step and Varying Amount of Overlap

| *Position* | *Reconstructed Value* |
|---|---|
| N - 6 | 9.99999999999999E-01 |
| N - 5 | 1.00000000000002E+00 |
| N - 4 | 9.99999999997335E-01 |
| N - 3 | 9.99999999995931E-01 |
| N - 2 | 9.99999999995932E-01 |
| N - 1 | 9.99999999997335E-01 |
| N | 1.00000000000002E+00 |
| N + 1 | 1.00000000001683E+00 |
| N + 2 | 1.00000000001252E+00 |
| N + 3 | 1.00000000000798E+00 |
| N + 4 | 1.00000000000367E+00 |
| N + 5 | 1.00000000000001E+00 |
| N + 6 | 1.00000000000000E+00 |

Table 3.3. Reconstruction of the Unit Step Function

utterance of the word "dark" are shown in figures 3.3 and 3.4. There were no distortions of the signal, even when the original and reproduced signal were compared in very small segments. The ASCII data files may be transformed into speech files (.sd) to be played on Entropics Waves+ using the following commands:

1. `addfea -f frequency_plot -t FLOAT -c comment filename.asc filename.out`

2. `tofeasd -f frequency_plot -R 8000 filename.out filename.sd`

When played through the Ariel audio interface, the original and recreated words and sentences could not be differentiated from each other.

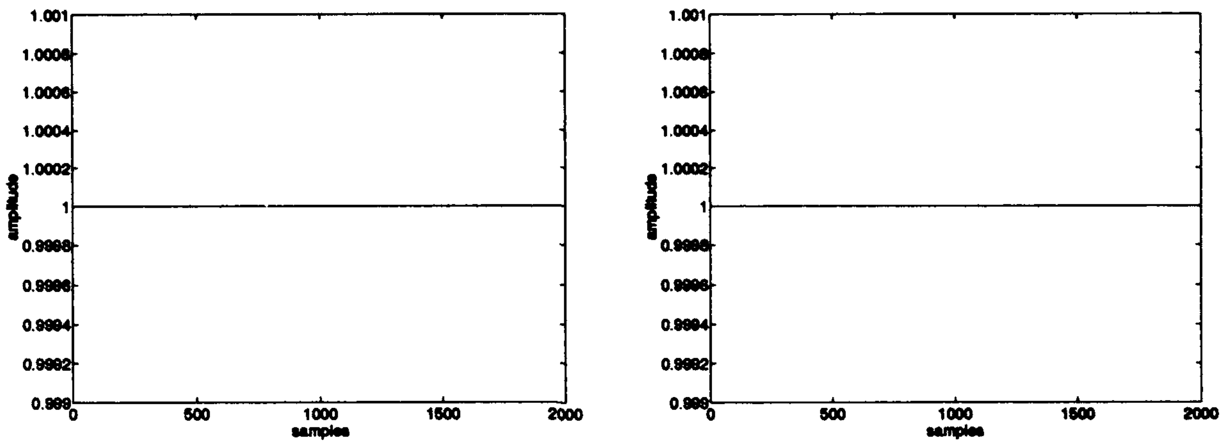


Figure 3.3. (a) Unit Step Function and (b) The Complex Malvar Wavelet Reconstruction of Unit Step

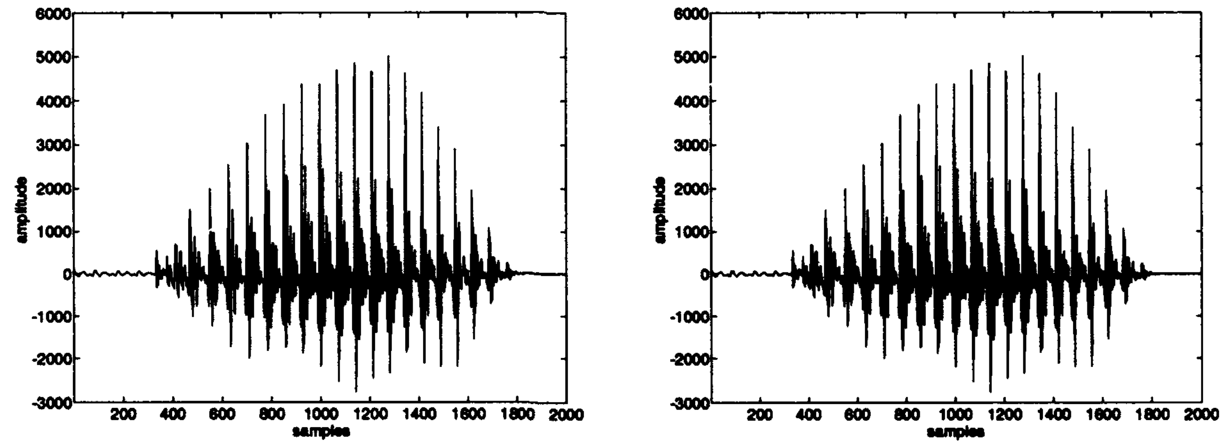


Figure 3.4. (a) Utterance of the Word "Dark" and (b) The Complex Malvar Wavelet Reconstruction

## *Compression Methods*

There are many different methods which can be used to compress the number of coefficients used for the transmission of a signal. Switzer studied eight different ways to select the most critical values of a transformed signal (35:31-39). The first scheme was a simple max-picking method, where the M largest values were used. The second scheme was a max-picking method with the frequencies above 600 Hz being boosted. The third selection scheme used a peak-picking method, where the max values are selected and the 12 nearest neighbors were then eliminated (to allow for more local maxima to ᵇ e selected). The fourth used a max-peak selection with the four nearest neighbors eliminated. The other methods used these plus a method for neighbor estimation. He found that the basic max-picking technique performed better than most of the other methods and worse than just two of the more complicated methods (35:70). Raduenz also performed similar experiments with the LOT (Malvar wavelet) and came to the same conclusion that a basic max-picking method performs just as well as most more elaborate methods (26:4-15).

*Absolute Thresholding:* Selecting the maximum values, as the max-picking methods of Switzer and Raduenz do, is the same as eliminating all of the transform coefficients under a given threshold. The Lapped Transform program was originally written so that a threshold could be arbitrarily selected. Any coefficients under this threshold would be set to zero and not used in the reconstruction of the signal. For the experiments being performed, a given PERCENT compression had to be obtained, so the program was modified to allow the user to select a given percent compression. If one requires 85% compression (85 out of every 100 coefficients are set to zero), the code will recursively and iteratively determine the amount of thresholding required to achieve 85% compression. The source code listed in Appendix E was added to the *Do Work* routine of the Malvar wavelet program.

The thresholding package first determines the number of zeros required to obtain the required percent compression. It then selects an initial threshold equal to the percent compression times the maximum coefficient value (so that the initial threshold increases as the compression ratio increases). The procedure then thresholds the coefficients by setting a temporary value to zero and compares the number of zeroed temporary values to the number of zeros required. If there are too many zeros, the threshold is decreased and if there are too few zeros, the threshold is increased. Once the correct number of zeros are achieved, the program drops out of the infinite loop and the coefficients are multiplied by the temporary values (1.0 if value is retained or 0.0 if value is thresholded).

### *Data Rates*

The overall goal for coding schemes is to reduce the data rates necessary to transmit essentially the same information. How is the compression ratio related to the actual data rate? There are two areas that need to be addressed in order to calculate the data rate. The first is how many bits will be required to represent the spectral coefficients that will be transmitted. The second factor is determining the overhead required to relate the coefficient to its position in the window (for example, was the sample the first, last or some other harmonic within a given frame). The typical sampling rate for communications equipment is at 8 kHz (2), therefore, all experiments were performed at the 8 kHz rate. The DARPA TIMIT Phonetic Speech database used for this research is sampled at 16 kHz so it was down sampled to 8 kHz prior to processing.

The coefficients of the Malvar wavelet have a symmetry such that only half of the coefficients need to be used. For a window of size N, the complex-valued Malvar wavelet, like the FFT, will yield N/2 unique magnitude and N/2 unique phase coefficients. The amplitude and position of these coefficients must now be transmitted using the least number of bits possible.

*Amplitude Information:* Initial experiments were performed with full 32 bit accuracy of the coefficients. Compression of 90% to 95% of the coefficients (magnitude and phase) resulted in speech files that had a mechanical tone of voice with ringing bell-like artifacts. The speech may have warbled and had a "spacey" sound to it, but it was still intelligible. The higher compression was slightly lower quality with more sudden starts and stops and more evident clicking, but the overall intelligibility was the same. The number of bits to code the magnitude and phase amplitudes were then reduced and the effects on the numerical and graphical errors and on the speech quality/intelligibility were recorded. Lloyd's algorithm was used to determine the quantization levels of the magnitude and phase (13).

Lloyd's algorithm starts by computing the mean of all the elements in the data to form a codebook of length one. Larger codebooks are then generated by splitting the previous codebook (replacing each codeword by two codewords, separated by a small number), and going through a process of improving the codebook until convergence is reached. This process works as follows: The squared distance is computed between every element of data and every codeword. Each codeword is then replaced by the mean of all the elements of data for which that codeword is the nearest neighbor. The mean square error (MSE) that results when each element of data is replaced by the nearest element in the codebook is computed. The improvement loop stops when the difference between the old distortion (MSE) and the new distortion, divided by the old distortion, becomes less than the convergence threshold.

Optimally quantizing a set of coefficients and then using the same data to transform the speech file would not be a fair test because only one set of quantization levels is used in a voice coder for ALL speakers. To avoid the use of an optimal quantization, Lloyd's algorithm was used to quantize one speech data file and those levels were used to transform a second speech file. In the first example, sixteen levels (4 bits) were used for both the magnitude and the phase information. For 4 bit quantization of magnitude and phase information, the cutoffs are more dramatic at the start and stops than the 32 bit quantization, however the overall quality is almost as good as the 32 bit reconstruction. Experiments involving several other combinations of magnitude and phase bits (denoted magnitude/phase bits) were performed. The errors associated with a few representative combinations of magnitude and phase bits are summarized in table 3.4.

| *Mag/Phase Bits* | *Relative $L_2$ @90%* | *RMS @90%* | *Relative $L_2$ @95%* | *RMS @95%* |
|---|---|---|---|---|
| 32/32 | 0.21783 | 0.47239 | 0.33298 | 0.71801 |
| 4/4 | 0.12256 | 0.79731 | 0.24137 | 0.88967 |
| 5/3 | 0.26711 | 0.62891 | 0.35930 | 0.79482 |
| 4/3 | 0.10249 | 0.81426 | 0.28627 | 0.90497 |
| 5/2 | 0.38574 | 0.79464 | 0.44870 | 0.92421 |
| 4/2 | 0.30004 | 0.94257 | 0.39389 | 1.01829 |
| 3/2 | 0.23641 | 1.33182 | 0.08885 | 1.36057 |

Table 3.4. Reconstruction Error of the CMW Using 90% and 95% Compression and Varying the Number of Bits to Represent the Magnitude and Phase

The number of phase bits were decreased for some experiments because speech signals are often not as sensitive to the phase content of the transform coefficients. This approach helped to improve the signal reconstruction without increasing the bit rate. For example, using 5 bits for magnitude and 3 bits for phase (5/3) produced a better sounding reconstruction (and lower numerical errors) than using 4 bits for both the magnitude and phase. By comparing the reconstructed signal to the required bit rate, the different coding schemes could be compared. The techniques were compared back to the 4 bits magnitude/4 bits phase (4/4) case at 95% compression; signal reconstructions that were clearly worse and techniques requiring a higher data rate were thrown out. Two of the quantization schemes appear to be superior to the other methods tested; the 5/3 scheme at 95% compression and the 3/2 technique at 90% compression have better sounding reconstructed signals than the baseline 4/4 technique and also require a lower data rate to achieve the reconstruction.

Note that the numerical error values do not always point toward the best audio reconstruction (25). The $L_2$ values are not consistent with the RMS values and neither criteria always points to the higher quality speech signal. The 3/2 (90%) case has a much higher RMS error, but a lower $L_2$ error than the 5/3 (95%) case. The reconstructed quality and intelligibility of the 3/2

(90%) signal is slightly better than the 5/3 (95%) reconstruction. However, the 5/3 scheme requires slightly fewer bits per second than the 3/2 case. The original speech signal and the 5/3 and 3/2 reconstructions are plotted in figures 3.5 and 3.6.

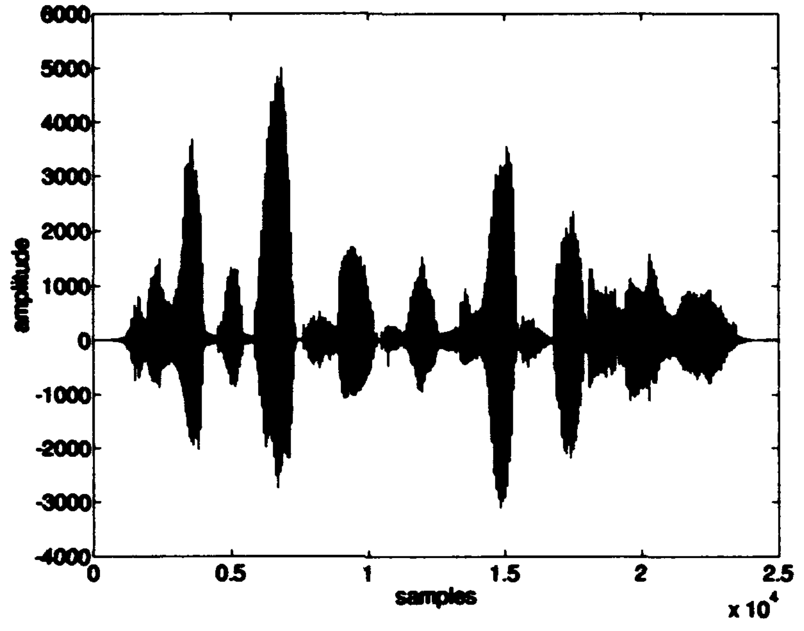
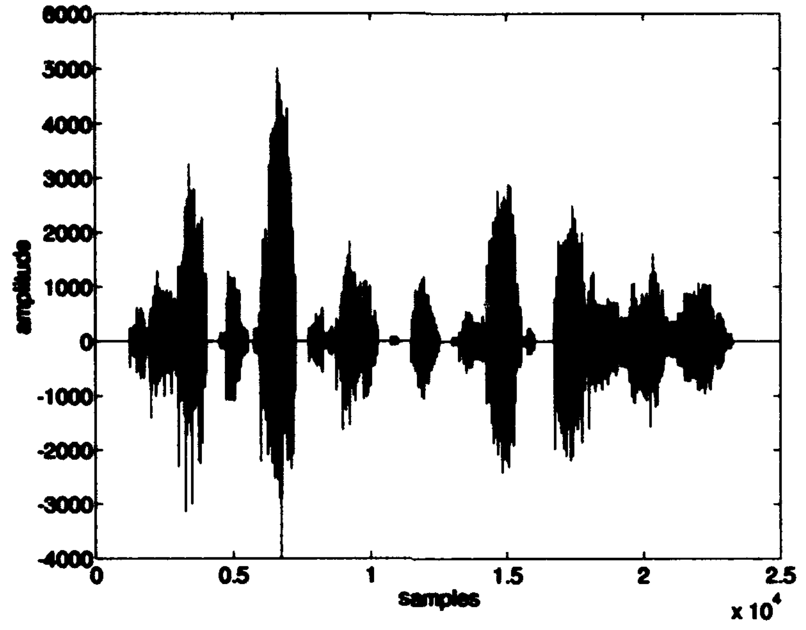


Figure 3.5. Utterance of the Sentence "She had your dark suit in greasy wash water all year"

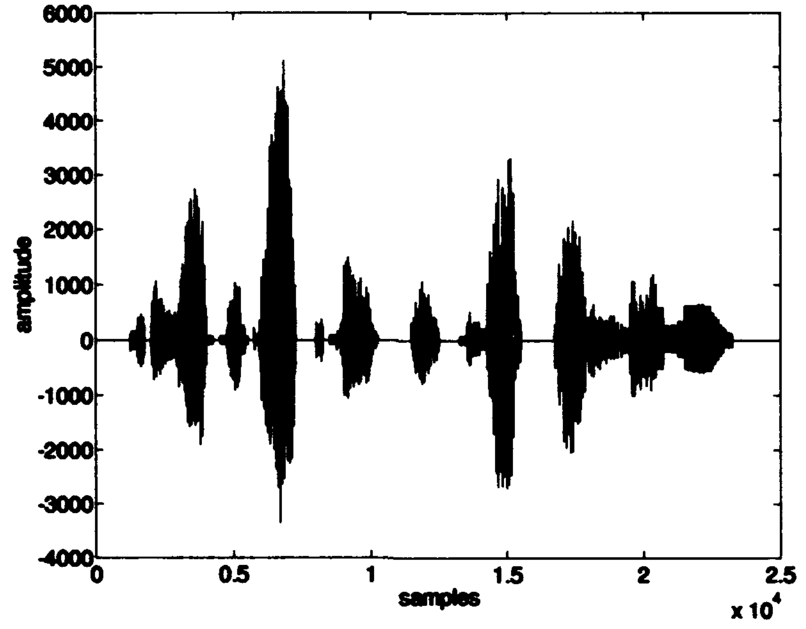
*Position Information:* The spectral position information can be coded and transmitted in many different ways. Two methods were evaluated:

1. Sending the position information with each individual non-zeroed coefficient.
2. Transmitting a large positional word prior to each window that tells where the non-zeroed coefficients belong.

The first method requires enough bits for each coefficient to account for every possible position. A 128 point window requires a seven bit word for each coefficient ($2^7 = 128$ possible positions) (2). The second method transmits the same information, but sends it prior to the window instead of



(a)



(b)

**Figure 3.6. Reconstruction of Sentence Using (a) 3 Mag Bits/2 Phase Bits at 90% Compression and (b) 5 Mag Bits/3 Phase Bits at 95% Compression**

with each coefficient. A 128 point window would have a 128 bit word that would contain a 1 at the positions where coefficients were being transmitted. At compression ratios of 90% or more, the first method generally requires the fewest number of bits because of the low number of coefficients which are being transmitted. The relationship between the length of the window and the amount of compression demonstrates which method to use. Method two always sends the number of positional bits equal to the length of the window, while the first method sends $(1-c)L\log_2 L$ bits (where L is the window length and c is the percent compression). Therefore, for a given length of window, if:

$$c > 1 - \frac{1}{\log_2 L} \tag{3.16}$$

then it is most economical to transmit the explicit position with each term (method one). If the compression is less than the value given in equation 3.16, then the positional vector should be transmitted. For windows of length 128, the positional vector would not be more advantageous until a compression ratio of 85% or lower was used.

The first method still requires almost half of the system's bit rate to be used just for positional information. The window can be broken down into smaller frames to reduce the amount of overhead. For large compression ratios, there will be many zeros within this information word. Instead of transmitting 7 bits to give the exact position of the coefficient, the position difference between two coefficients could be transmitted. This difference would be limited to a portion of the larger window. For example, the 128 point window can be broken into eight 16 point sub-windows. If there are no coefficients within 16 samples, then a one bit "dummy" coefficient (equal to zero) is inserted to hold the 16th samples place. Thus the total bits sent prior to the window can be reduced with a "divide and conquer" approach (32) (35:4-4). The maximum number of "dummy" bits to break a 128 sample window into 16 sample sub-windows is seven (need a "dummy" at only one end point). The symmetry of the complex-valued Malvar wavelet coefficients can be used to reduce the amount of positional information required. The position can be conveyed with a six bit word (64 sample window with symmetry). This can be transmitted with eight eight-point sub-windows and only four "dummy" bits will be required for the 64 sample window.

*Examples:* The bit rate for the transmission of the spectral coefficients can be computed by following the above steps. This example will assume a sampling rate of 8 KHz, 128 point windows, 3 bits/magnitude coefficient, 2 bits/phase coefficient, and 90% compression.

1. 8000 samples/sec / 128 samples/window = 62.5 windows/second.
2. 90% compression of a 128 point window $\Rightarrow$ 12 samples/window.
3. Symmetry of CMW $\Rightarrow$ 6 samples/window
4. 5 bits/sample (magnitude and phase of coefficient)
   + 3 bits/sample (coefficient position within 8 point sub-window)

   8 bits/sample
5. 4 bits/window ("dummy" coefficients)
6. (8 bits/sample)·(6 samples/window) + (4 bits/window) = 52 bits/window
7. (62.5 windows/second)·(52 bits/window) = 3.25 Kbits/second

The next example utilizes 5 bits/magnitude coefficient, 3 bits/phase coefficient, and 95% compression:

1. 8000 samples/sec / 128 samples/window = 62.5 windows/second.
2. 95% compression of a 128 point window $\Rightarrow$ 6 samples/window.
3. Symmetry of CMW $\Rightarrow$ 3 samples/window
4. 8 bits/sample (magnitude and phase of coefficient)
   + 3 bits/sample (coefficient position within 8 point sub-window)

   11 bits/sample

5. 4 bits/window ("dummy" coefficients)

6. (11 bits/sample)·(3 samples/window) + (4 bits/window) = 37 bits/window

7. (62.5 windows/second)·(37 bits/window) = 2.31 Kbits/second

Even with the higher number of bits/coefficient, the higher compression leads to a lower data rate. The 95% compressed version did not have the same quality output that the 90% version had, but the bit rate is significantly reduced. The CMW competes very well with current speech encoders. Typical low bit rate speech encoders used today such as the LPC-10 coder (used in the STU-III secure voice system) utilizes 2400 bps (2).

The lapped transform allows us to overlap the data and smooth the boarder regions without increasing the number of windows transmitted per second. This is due to the fact that the Malvar wavelet "folds" the overlapped data back into the window before it calculates the coefficients used in the transmission. The number of bits/second without compression (but accounting for symmetry) would have been:

(8 bits/sample)(64 samples/window)(62.5 windows/second) = 32 Kbits/sec

The percent compression of the overall bit rate is thus 72% versus the original 95% compression ratio of the transform coefficients. This appears to be a dramatic decrease in compression, however the 95% ratio was simply the number of coefficients compressed. It did not take into account the positional information or the number of bits/coefficient. The 72% compression achieved still yielded a data rate less than 2400 bps.

Further reductions in the bit rate may be obtained by using more elaborate coding and compression techniques. Huffman developed a method to minimize the expected number of transmitted bits (9). He optimized the bit pattern by linking the frequency of occurrence of a given value to the number of bits to code that value such that;

Expected Number of Bits = (Frequency of Occurrence)·(Bits for the Values)

The Malvar wavelet has also been used to find voicing probabilities in speech patterns (39). The Malvar wavelet provides an automatic segmentation into phonetic units, such that the frequencies center of mass associated with each phonetic unit can be used to get a voiced/unvoiced segmentation algorithm. Voiced and unvoiced speech can then be coded and compressed differently to produce a higher quality reconstruction at a lower bit rate.

*Additional Promising Applications*

A complex-valued lapped orthogonal transform would be a powerful tool for image processing. A non-orthogonal complex-valued lapped transform (CLT) has been defined and utilized by Young and Kingsbury to perform frequency-domain motion estimation (41). Young and Kingsbury extended the lapped transform to form a complex transform because data shifts in the spatial domain have more meaningful interpretations in the complex domain (primarily as phase shifts of the complex coefficients) (41). The CLT gives smoother motion fields than the traditional block matching, due to the use of overlapping windows with no block edge discontinuities. The complex-valued lapped orthogonal transform proposed by Suter and Oxley may yield even clearer motion fields because of its perfect reconstructive properties.

There are many image processing applications which could be improved through the use of a complex lapped orthogonal transform, the CMW. The MITRE Corporation has proposed an objective image quality measure that is highly dependent upon the reconstructive capabilities of the short time Fourier transform (20). This quality measure takes the two dimensional Fourier transform of an image, bandpass filters it, and reconstructs the image with another Fourier transform. Potential applications in the image processing field are important for the complex Malvar wavelet.

RADAR signal analysis is another field that could take advantage of an orthogonal complex-valued transform. Phase information is very important to the study of RADAR signals and the

Fourier transform is the basic tool for analysis of the signals. Many spectral estimation techniques are used to improve the resolution and accuracy of the FFT. The complex-valued Malvar wavelet could assist these techniques.

### *Summary*

A complex Malvar wavelet, which maintains orthogonality, was introduced. This algorithm was programmed in the DOD standard language of Ada. The code was tested and evaluated to demonstrate that the transform produces perfectly reconstructed signals. The CMW, with high compression ratios, was shown to be able to produce intelligible speech and to compete with current low bit-rate vocoders. Imaging and RADAR processing techniques for the CMW were introduced as possible applications.

# *IV. Real-Valued Malvar Wavelets*

## *Introduction*

The sinusoidal Malvar wavelet (or generalized lapped orthogonal transform) is developed in much the same way as the complex-valued Malvar wavelet, however the algorithm for the sinusoidal wavelet is more complicated because it is a real basis function and it allows for variable size windows and overlaps. The complete algorithm is presented in (34) and (26). The benefit of using this real-valued basis is that the coefficients will be, by definition, real. Therefore if the same amount of information can be sent with one real coefficient vice two coefficients (magnitude and phase), there is a potential for a decrease in the bit rate. The question that must be answered is whether the sinusoidal MW, at a given percent compression, can produce the same quality signal as the complex MW. The following steps summarize the derivation of the sinusoidal Malvar wavelet as proposed by Suter and Oxley (34):

1. Define a Basis. The basis function is defined as the even and odd extensions of a sinusoidal basis:

$$\tilde{f}_{j,k}(x) = \begin{cases} 0 & , \quad -\infty < x \leq a_j - \epsilon_j \\ -\sqrt{\frac{2}{a_{j+1}-a_j}} \sin\left(\pi(k+\frac{1}{2})\left[\frac{a_j - x}{a_{j+1}-a_j}\right]\right) & , \quad a_j - \epsilon_j < x < a_j \\ \sqrt{\frac{2}{a_{j+1}-a_j}} \sin\left(\pi(k+\frac{1}{2})\left[\frac{x - a_j}{a_{j+1}-a_j}\right]\right) & , \quad a_j \leq x \leq a_{j+1} \\ \sqrt{\frac{2}{a_{j+1}-a_j}} \sin\left(\pi(k+\frac{1}{2})\left[\frac{a_j - x}{a_{j+1}-a_j}\right]\right) & , \quad a_{j+1} < x < a_{j+1} + \epsilon_{j+1} \\ 0 & , \quad a_{j+1} + \epsilon_{j+1} \leq x < \infty. \end{cases} \tag{4.1}$$

   Note that the odd/even extensions have been reversed from the complex-valued case. It does not matter what order the extensions are taken, as long as the equations remain consistent throughout the algorithm.

2. Obtain data for some interval; $a_j - \epsilon_j \leq x \leq a_{j+1} + \epsilon_{j+1}$ as shown if figure 4.1.

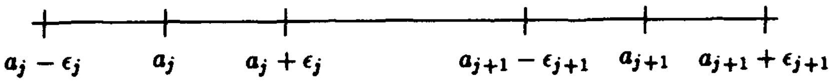


Figure 4.1. Partitioned Axis

3. Multiply the data by the window ($w_j(x)$).

$$w_j(x) = \begin{cases} 0 & , \ -\infty < x < a_j - \epsilon_j \\ \sin(\frac{\pi}{4\epsilon_j}\{x - [a_j - \epsilon_j]\}) & , \ a_j - \epsilon_j \le x \le a_j + \epsilon_j \\ 1 & , \ a_j + \epsilon_j < x < a_{j+1} - \epsilon j + 1 \\ \cos(\frac{\pi}{4\epsilon_{j+1}}\{x - [a_{j+1} - \epsilon_{j+1}]\}) & , \ a_{j+1} - \epsilon_{j+1} \le x \le a_{j+1} + \epsilon_{j+1} \\ 0 & , \ a_{j+1} + \epsilon_{j+1} \le x \le \infty. \end{cases} \tag{4.2}$$

4. Fold the sequence; fold in edges upon itself.

$$H_{j,l}(x) = \begin{cases} s(x)w_j(x) - s(2a_j - x)w_j(2a_j - x) & , \ a_j \le x \le a_j + \epsilon_j \\ s(x) & , \ a_j + \epsilon_j < x < a_{j+1} - \epsilon_{j+1} \\ s(x)w_j(x) + s(2a_{j+1} - x)w_j(2a_{j+1} - x) & , \ a_{j+1} - \epsilon_{j+1} \le x \le a_{j+1} \end{cases} \tag{4.3}$$

5. Define a new array.

$$\beta_{j,l} = H_{j,l} \sin\left(\frac{\pi l}{2N_j}\right) \quad for \quad l = 0, l, \cdots, N_j \tag{4.4}$$

6. Define the even extension of $\beta_{j,l}$.

$$\tilde{\beta}_{j,l} = \begin{cases} \beta_{j,l} & , \ 0 \le l \le N_j - 1 \\ \beta_{j,2N_j-1} & , \ N_j \le l \le 2N_j - 1 \end{cases} \tag{4.5}$$

7. Perform an FFT of length $2N_j$ on this even extended sequence.

$$D_{j,k} = Re\left(\frac{1}{2N_j}\sum_{l=o}^{2N_j-1}\tilde{\beta}_{j,l}e^{i2\pi kl/2N_j}\right) \tag{4.6}$$

8. Interpret the results of the FFT.

$$\alpha_{j,0} = \sqrt{2N_j}D_{j,0} \tag{4.7}$$

$$\alpha_{j,k} = \alpha_{j,k-1} + 2\sqrt{2N_j}D_{j,k} \tag{4.8}$$

9. Define the odd extension of the spectral coefficients.

$$\tilde{\alpha}_{j,k} = \begin{cases} \alpha_{j,k} & , \quad 0 \le k \le N_j - 1 \\ -\alpha_{j,2N_{j,}-k-1} & , \quad N_j \le k \le 2N_j - 1 \end{cases} \tag{4.9}$$

10. Perform an Inverse FFT of length $2N_j$ on $\tilde{\alpha}_{j,k}$ and define.

$$H_{j,l} = \sqrt{2N_j}Im\left(e^{\frac{i\pi l}{2N_j}}\left[\frac{1}{2N_j}\sum_{k=0}^{2N_j-1}\tilde{\alpha}_{j,k}e^{\frac{i2\pi kl}{2N_j}}\right]\right) \tag{4.10}$$

11. Reconstruct the signal by unfolding the data and dividing by the window to fold the edges back out.

$$S_{j,l} = \begin{cases} \frac{H_{j-1,l}}{2w_{j-1}(x_{j,l})} & , \quad x_{j,l} = a_j \\ w_j(2a_j - x_{j,l})H_{j-1,N_{j-1}-l} + w_j(x_{j,l}H_{j,l}) & , \quad a_j < x_{j,l} < a_j + \epsilon_j \\ H_{j,l} & , \quad a_j + \epsilon_j \le x_{j,l} \le a_{j+1} - \epsilon_{j+1} \\ w_j(x_{j,l})H_{j,l} - w_j(2a_{j+1} - x_{j,l})H_{j+1,N_j-l} & , \quad a_{j+1} - \epsilon_{j+1} < x_{j,l} < a_{j+1} \end{cases} \tag{4.11}$$

## *Validation*

The first step in validating the proposed system is to ensure that the basic Malvar wavelet code is correct. The program written by Raduenz was not perfect because he was concerned with only integer values of signals and the prime emphasis of his work was not on comparing the results of different systems. The Ada code was significantly modified so that it would give accurate results for the comparison of multiple systems. The complete program is provided in Appendix C. The important modifications include:

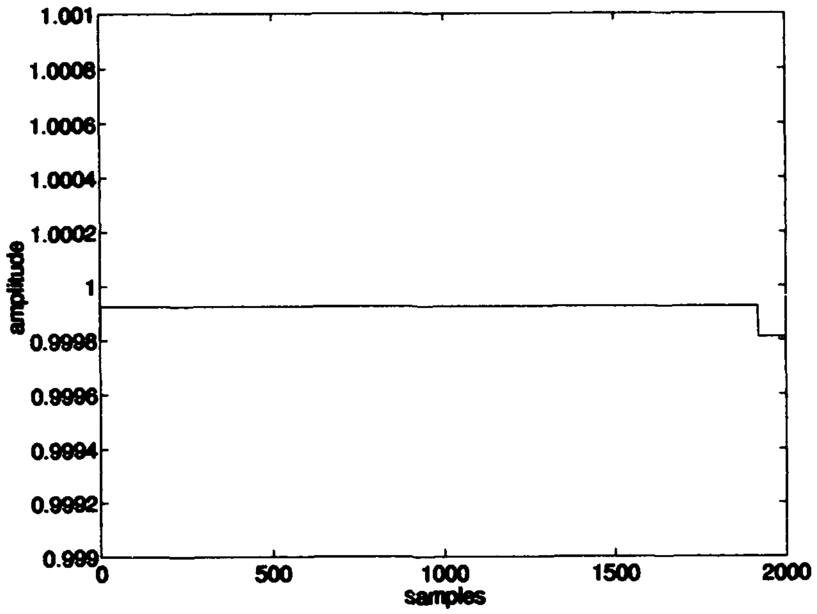
(1) Allow the system to read in and write out floating point values (with accuracy to the 12th decimal place).

(2) Enable the system to perform finite length processing without having to duplicate data or process intervals multiple times. This was achieved by "wrapping" the end points of the signal, so that the first value was considered as the point immediately *after* the last value of the data set.

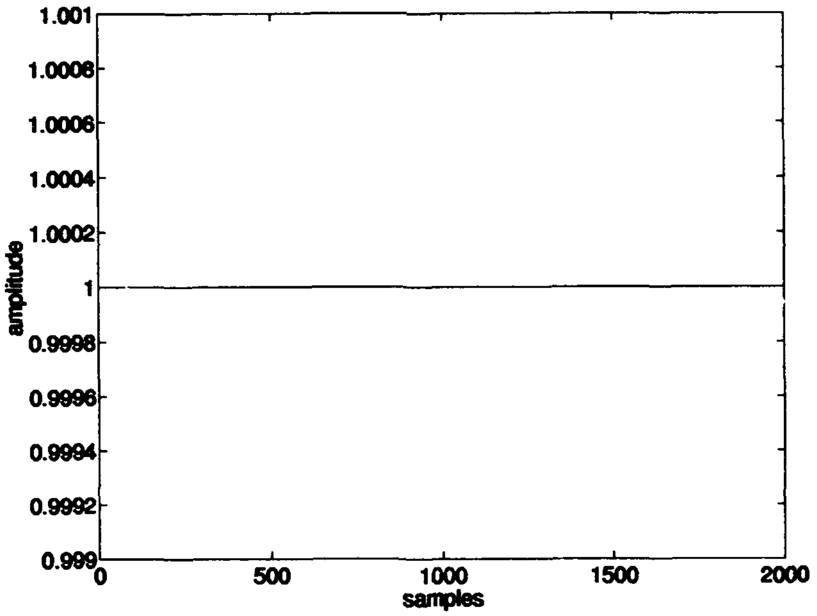
(3) The starting and stopping points were adjusted to correctly match the algorithm for equations 4.10 and 4.11.

(4) The percent compression and error calculations code (discussed in Chapter III and listed in Appendix E) were added to the baseline program.

The original code produced normalized RMS errors of about $10^{-5}$ when reproducing a simple unit step or sinusoid. The reproduction of a unit step is shown in figure 4.2a. This shows that the overall reproduction is very close, but not perfect and that the final window is being distorted by the algorithm. The modified program's output is plotted in figure 4.2b. This is almost exactly equal to 1.00..0 and has no distortions in the overlaps. The normalized RMS error for the reconstruction is just over $10^{-13}$. The reconstruction errors for three different signals are given in table



(a)



(b)

Figure 4.2. (a) Original Malvar wavelet and (b) the Modified MW Programs' Reconstruction of the Unit Step Function

| *Input Data* | *Relative $L_2$* | $L_\infty$ | *RMS* | *RIV (1.0 + )* |
|---|---|---|---|---|
| Unit Step | $1.835x10^{-8}$ | $1.877x10^{-10}$ | $2.079x10^{-13}$ | $-0.051x10^{-12}$ |
| Sinusoid | $1.849x10^{-8}$ | $1.876x10^{-10}$ | $1.879x10^{-13}$ | $-0.074x10^{-12}$ |
| Utterance of "DARK" | $1.615x10^{-5}$ | $3.018x10^{-10}$ | $2.886x10^{-12}$ | $-0.037x10^{-12}$ |

Table 4.1. Reconstruction Errors Associated with the Real-Valued Malvar Wavelet

### *Compression*

Raduenz's work suggested that the homomorphic filtering process may be superior to the use of the sinusoidal Malvar wavelet (LOT) alone (26:4-30). The homomorphic filtering allowed Raduenz to obtain better reconstruction of speech signals at comparable compression ratios. The goal of this research is to verify those results, investigate other homomorphic filters to determine if there is a best method for a given signal, and then explain why a particular method(s) worked best.

This chapter outlines how the problem was attacked. It outlines the different types of homomorphic filters evaluated, the technique used to threshold (compress) the resulting coefficients, the error criteria used to evaluate the results, and finally the results of the experiments.

### *Homomorphic Filters*

In its most general form, a homomorphic system can be represented by algebraically linear (homomorphic) mappings between the input and output signals (22:771). This system can then be decomposed into three separate homomorphic systems; a characteristic system, a linear system, and the inverse characteristic system.

There were three distinct systems used in this research. Only one of them was technically a homomorphic system. System 1 is shown in figure 4.3. The generalized Malvar wavelet (GMW) acts as the characteristic system and five different functions were used as the linear system. The five linear filters (and their inverses) that were utilized include:

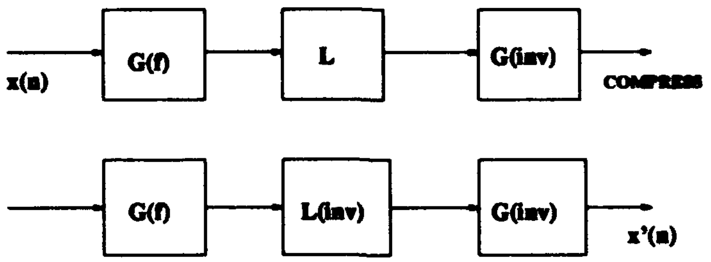


Figure 4.3. System 1: Homomorphic System Utilizing the Malvar wavelet

1. No Linear Transform: $1x$
2. Square Root: (Square)
3. Natural Logarithm: (Exponential)
4. $10x + 1$: $((x - 1)/10)$
5. Arc Hyperbolic Sine: (Sinh)

where, hyperbolic sine and arc hyperbolic sine are defined as (31):

$$\sinh(x) = 0.5 * (e^{x} - e^{-x}) \tag{4.12}$$

$$\sinh^{-1}(x) = \ln(x + \sqrt{x^2 + 1}) \tag{4.13}$$

The three systems consist of an analysis bank, which transforms the coefficients prior to transmission, and a synthesis bank, where the transmitted coefficients are reconstructed back into the original signal. For System 1, the analysis bank is composed of the forward GMW followed by one of the given linear filters and then an inverse GMW. The coefficients from this bank are then to be compressed (thresholded at a certain level), transmitted, and sent to the synthesis bank for signal reconstruction. The synthesis bank is made up of a forward GMW, the inverse of the linear filter used in the analysis bank, and finally an inverse GMW. System 2, figure 4.4, represents the system used by Raduenz. This system is comprised of the GMW, a linear filter, and another forward GMW. The synthesis filter is constructed from the inverse GMW, an inverse linear filter,

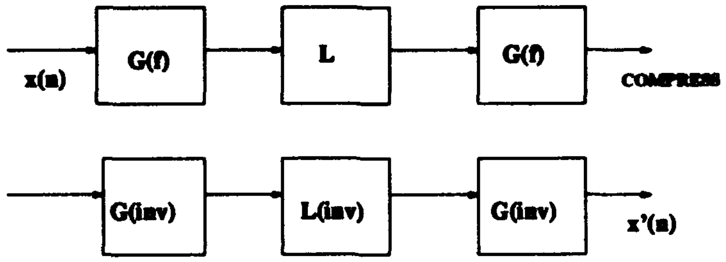


Figure 4.4. System 2: Homomorphic-Like System Utilizing the GMW

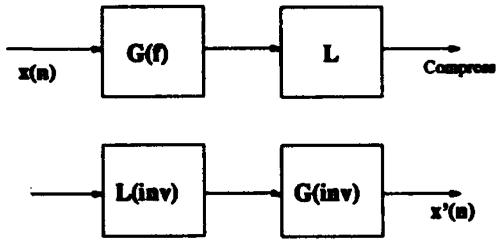


Figure 4.5. System 3: Malvar Wavelet with Linear Filter

and finally another inverse GMW. The final system (system 3) used in these experiments consists only of the GMW followed by a linear filter as shown in figure 4.5. There are 15 different systems to be analyzed; five different linear filters in each system. Raduenz performed the homomorphic filtering by completing the first forward GMW, taking the natural logarithm of all the GMW coefficients, and then performing another forward GMW. In order to be able to perform the linear filtering in real-time, a procedure was added to the GMW program to implement the linear filter on each window of data after the characteristic system (GMW) had transformed that given window. The linear filter will then pass the set of data on to the next function (Inv GMW for System 1). To accomplish this, a step was added to the procedure *Do Work* of the Lapped Transform program that calls the new procedure *Linear Transform*. The steps which comprise the *Linear Transform* procedure in the Lapped Transform program are found in Appendix D.

The first time the *Linear_Transform* procedure is called, the value of *Inverse* is set to zero so that the forward transform is performed. Subsequent times the procedure is called, *Inverse* is set to one, so that the inverse transform is used. The type of linear transform (log, $\sqrt{x}$, etc) is selected by the user, prior to the start of any processing. Note that the phase of the GMW coefficients do not have to be tracked because the coefficients of the GMW (as defined at the beginning of this chapter) are always real-valued. However the positive and negative values could be expressed as a $+/-$ or $0^o/180^o$ phase. The negative values pose a problem for the logarithm and the square root functions. This could be corrected for by expressing the functions with a phase term and utilizing the complex function, $log[X(f)] = ln|X(f)| + j\angle X(f)$. In order to avoid any kind of phase term, the coefficients can be offset so that the minimum coefficient value is 1.0 and the minimum log amplitude is therefore 0.0. The offset can then be removed after the inverse natural log ($e^x$). For example, if there exists a set of coefficients $p(x)$, an offset of $y =$ minimum$[p(x)] + 1$ is used with the logarithm:

1. Coefficient value $p(x)$
2. Add an offset of y: $(p(x) + y)$
3. Take the natural log: $ln(p(x) + y)$
4. Perform additional linear transforms: (Lapped Trans or Inv LOT)
5. Take inverse natural log: $e^{(p(x)+y)} = p(x) + y$
6. Subtract the offset: $p(x) + y - y = p(x) + (min + 1) - (min + 1) = p(x)$

This same procedure can be applied to the square root, with the offset equal to the minimum coefficient value, because the square root of zero is defined. This method works well when utilizing all of the coefficients. However, compressing the coefficients (setting coefficients to zero) introduces a non-linearity to the system. This non-linearity means that the minimum valued being carried through the system will distort the reconstructed signal. A third method can be used to track the sign of the original coefficients. The sign (+/-) of the original coefficient is stored in a vector and then the absolute value of the coefficient is used with the function. The sign is then put back on the signal after reconstruction ($e^x$ or $x^2$). The benefit of this method is that the sign vector will not distort the resulting signal when coefficients are eliminated.

*System Validation*

Each complete system must now be validated to ensure packages are working together properly. To validate each program and linear transform system, known signals were passed through the systems with 0% compression. Perfect reconstruction is expected when there is no lose of information (no coefficients thresholded). Allowing for some round-off error, each filter and linear system produced perfectly reconstructed signals. The speech signals sounded identical and reconstruction errors were very low. The numerical error criteria discussed in Chapter III were used to compare the results.

*Numerical Error Tests:* The reconstruction errors ($L_2$, $L_\infty$, RMS, and RIV) were collected and compared. System 1 (GMW, Linear Function, Inv GMW: see figure 4.3) and system 2 (GMW, linear Function, GMW: see figure 4.4) produced nearly identical output errors. There are many different ways in which both systems may be coded as part of the lapped transform routine, however they always produced very similar levels of reconstruction error. System 3 (GMW, Linear Function: see figure 4.5) also provided near perfect reconstruction at 0% compression (using 125 point windows with 50 point overlap). The numerical errors have been tabulated below for a 2000 point sinusoidal input.

| *Linear Function* | *Relative $L_2$* | $L_\infty$ | *RMS* | *RIV (1.0 + )* |
|---|---|---|---|---|
| 1 | $2.284x10^{-5}$ | $6.037x10^{-10}$ | $5.772x10^{-12}$ | $-7.555x10^{-11}$ |
| $\sqrt{x}$ | $3.058x10^{-5}$ | $7.981x10^{-10}$ | $8.626x10^{-12}$ | $-1.354x10^{-10}$ |
| $\log(x)$ | $2.327x10^{-5}$ | $5.052x10^{-10}$ | $5.283x10^{-12}$ | $-7.840x10^{-11}$ |
| $10x + 1$ | $2.284x10^{-5}$ | $6.090x10^{-10}$ | $5.818x10^{-12}$ | $-7.555x10^{-11}$ |
| $\sinh(x)$ | $2.915x10^{-5}$ | $7.261x10^{-10}$ | $7.736x10^{-12}$ | $-1.230x10^{-11}$ |

Table 4.2. Reconstruction Errors Associated with System 1

All four functions ($1x$, $\sqrt{x}$, $\log(x)$, $10x+1$, and $\sinh(x)$) produced identical outputs for system 3. These reconstructed signals had lost very little information and were perfect reconstructions of the originals. The output signals produced by system 1 and 2 were not as well defined. The linear functions ($1x$, and $10x + 1$) produced nearly identical output signals. Note that these functions

| *Linear Function* | *Relative $L_2$* | $L_\infty$ | *RMS* | *RIV (1.0 + )* |
|---|---|---|---|---|
| 1 | $2.464x10^{-5}$ | $6.525x10^{-10}$ | $5.876x10^{-12}$ | $-8.795x10^{-11}$ |
| $\sqrt{x}$ | $3.081x10^{-5}$ | $6.472x10^{-10}$ | $7.778x10^{-12}$ | $-1.374x10^{-10}$ |
| $\log(x)$ | $5.932x10^{-5}$ | $1.954x10^{-9}$ | $2.800x10^{-11}$ | $-5.094x10^{-10}$ |
| $10x+1$ | $2.464x10^{-5}$ | $6.550x10^{-10}$ | $5.880x10^{-12}$ | $-8.795x10^{-11}$ |
| $\sinh(x)$ | $3.396x10^{-5}$ | $7.532x10^{-10}$ | $9.419x10^{-12}$ | $-1.670x10^{-10}$ |

Table 4.3. Reconstruction Errors Associated with System 2

| *Linear Function* | *Relative $L_2$* | $L_\infty$ | *RMS* | *RIV (1.0 + )* |
|---|---|---|---|---|
| 1 | $1.615x10^{-5}$ | $3.018x10^{-10}$ | $2.886x10^{-12}$ | $-3.777x10^{-11}$ |
| $\sqrt{x}$ | $1.615x10^{-5}$ | $3.018x10^{-10}$ | $2.886x10^{-12}$ | $-3.777x10^{-11}$ |
| $\log(x)$ | $1.615x10^{-5}$ | $3.018x10^{-10}$ | $2.886x10^{-12}$ | $-3.777x10^{-11}$ |
| $10x+1$ | $1.615x10^{-5}$ | $3.018x10^{-10}$ | $2.886x10^{-12}$ | $-3.777x10^{-11}$ |
| $\sinh(x)$ | $1.615x10^{-5}$ | $3.018x10^{-10}$ | $2.886x10^{-12}$ | $-3.777x10^{-11}$ |

Table 4.4. Reconstruction Errors Associated with System 3

also had the lowest reconstruction errors for system 1 and 2. The other functions produced slightly higher errors than the $1x$ and $10x + 1$ functions. The recorded errors are so small that they are very sensitive to the number of flops (multiplies and adds) performed by the system. System 3, the most basic procedure, uses the least number of flops and thus has the lowest reconstruction errors. System 2 has the greatest number of flops because of the way the system had to be constructed. The original lapped transform program was not written to have two forward transforms in succession, so it has more coefficient calculations than do the other two systems. It follows then that system 2 has slightly higher reconstruction errors.

All of the reconstruction errors are extremely small (the largest normalized RMS error = $2.800x10^{-11}$) and easily demonstrate the perfect reconstruction properties of the systems.

*Graphical and Audio Tests:* Graphical and audio tests were also made to measure the signal reconstructions. These again showed how well all of the systems were restored when there is no thresholding of the coefficients. Figure 4.6 shows the sinusoid that was used for the initial testing of the systems. Figure 4.7 (a) is the output of system 3 using no linear function (GMW, Inverse GMW). The output of the systems using other functions (log, sinh, etc) were identical to figure 4.7. Figure 4.7 (b) shows the output from system 1 using no linear function (GMW, Inverse GMW, GMW, Inverse GMW). This restoration is also perfect. The outputs of the other functions performed virtually the same as the $1x$ case.

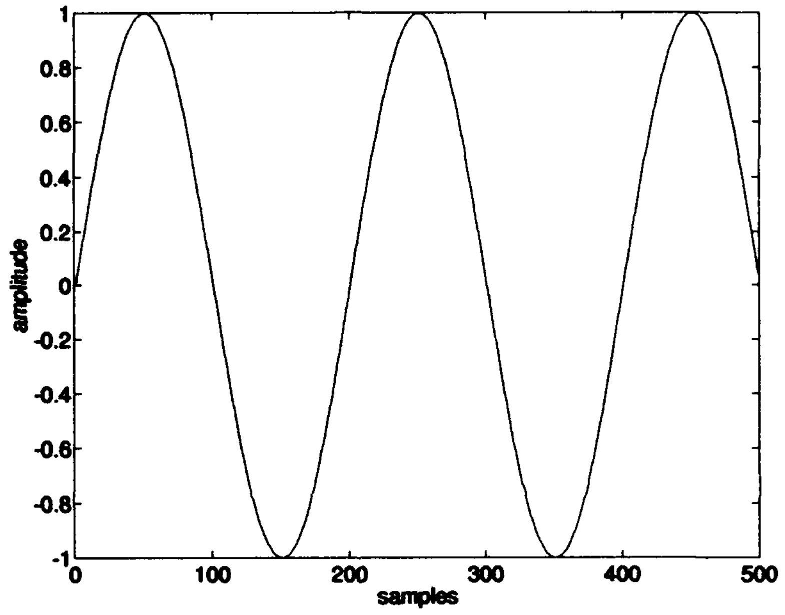
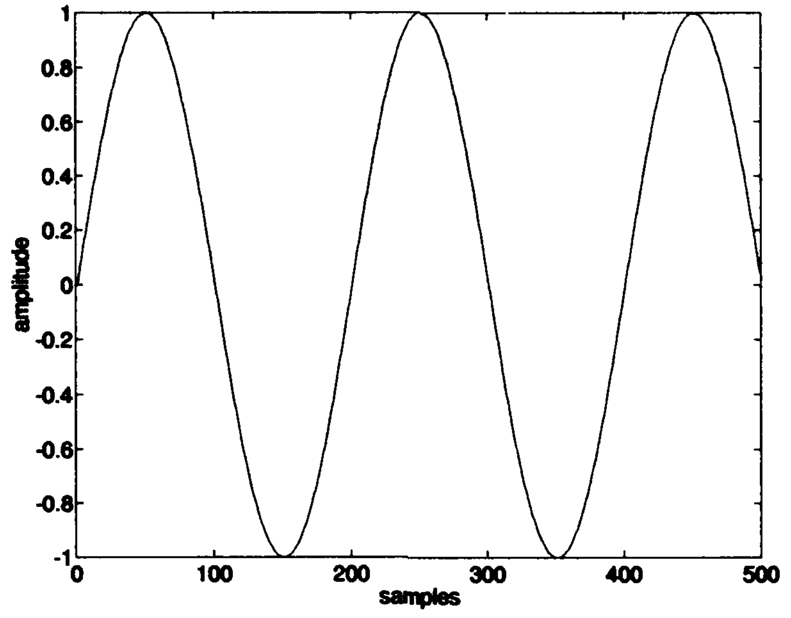


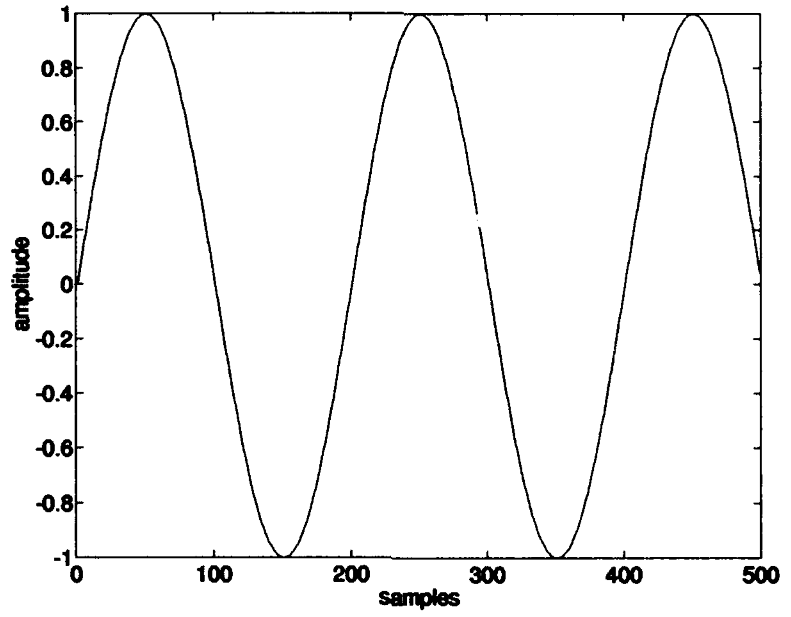
Figure 4.6. Original Sinusoid Signal

After testing the system with a step function and a sinusoid, speech files from the TIMIT database were used. Words and parts of words that were 2000 points long were tested with systems 1, 2, and 3. This allowed for more complicated signals to be transformed by the programs and to actually listen to the results instead of simply looking at numbers and plots. Figure 4.8 shows the graphical display of the utterance for the word "dark." The reconstructions utilizing system 3, and system 1 are plotted in figures 4.9(a) and (b).

The three plots of the word "dark" look very much the same, but that does not guarantee that they will sound identical. When played through the Ariel audio interface, the words could



(a)



(b)

Figure 4.7. (a) GMW (System 3) and (b) Homomorphic (System 1) Reconstruction of the Sinusoid

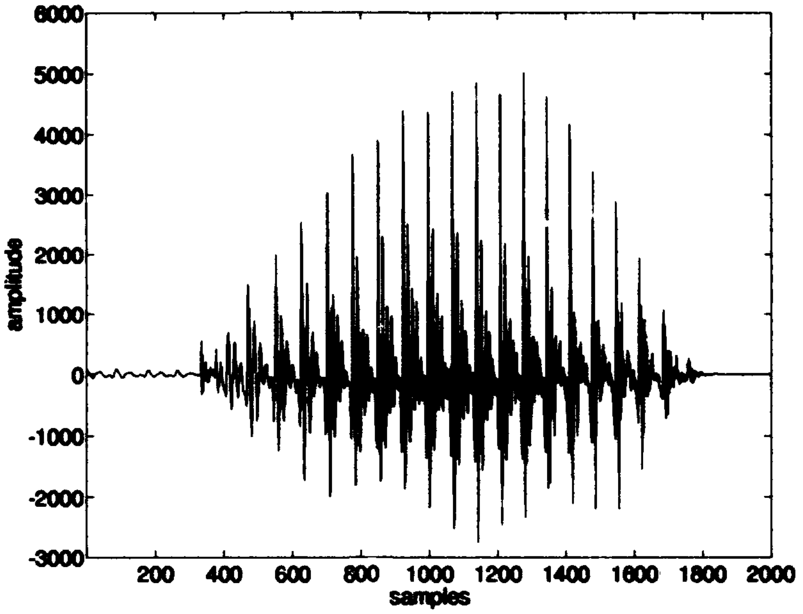
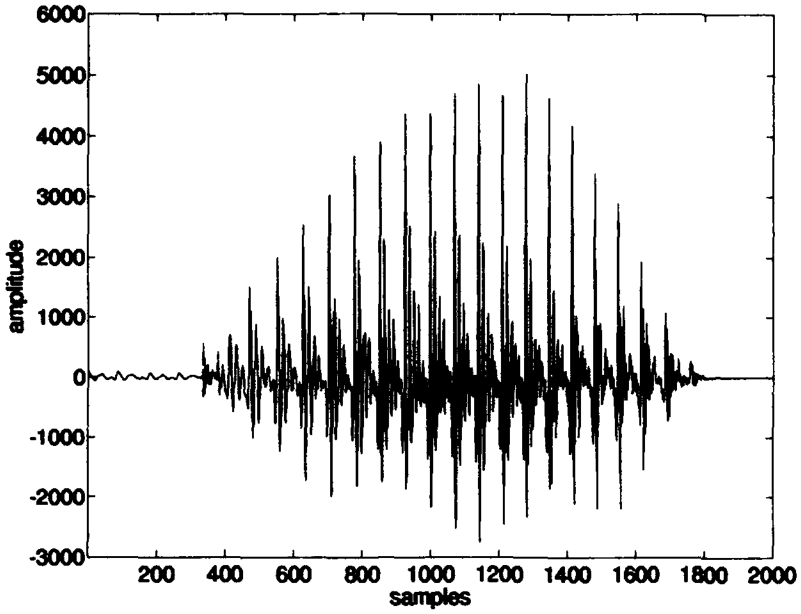


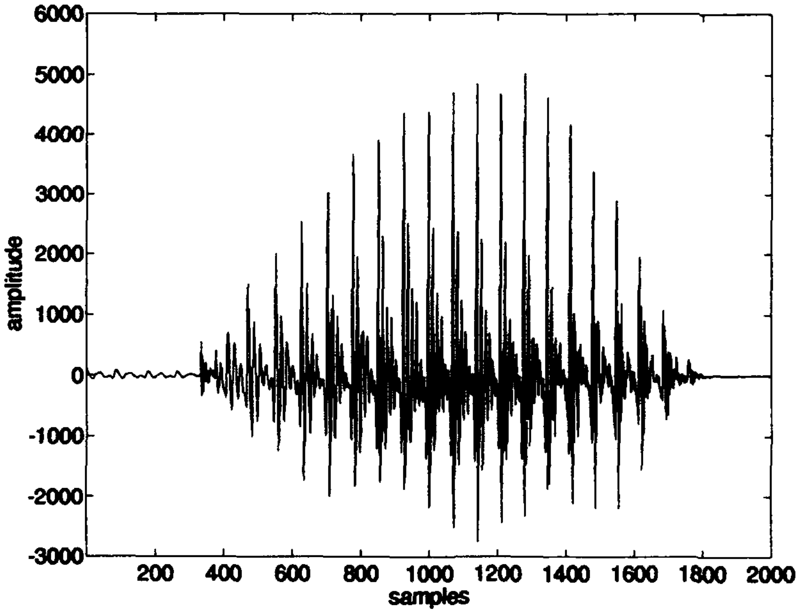
Figure 4.8. Utterance of the Word "Dark"

not be differentiated. However, short one syllable words often sound alike because there is so little time for the ear to differentiate them. The next step was to utilize entire sentences of speech. The results from transforming entire speech files was very similar to the previous results. All of the systems provided perfect reconstruction using all criteria; numerical values, plotting, and listening. Listening provided the least differentiation between the systems. The utterance, "She had your dark suit in greasy wash water all year," was reconstructed using the three systems. All provided perfect reconstruction of the speech signal (the worst normalized RMS error was $10^{-9}$). All of the signals sounded identical when played via xwaves. There were no added noise, pops, or spikes in the recreated signals.

*Observations:* The previous section demonstrates that perfect reconstruction was possible using homomorphic filtering techniques with Malvar wavelets. The results also demonstrates that the most basic system, consisting of only the GMW and the Inverse GMW, yields the best results

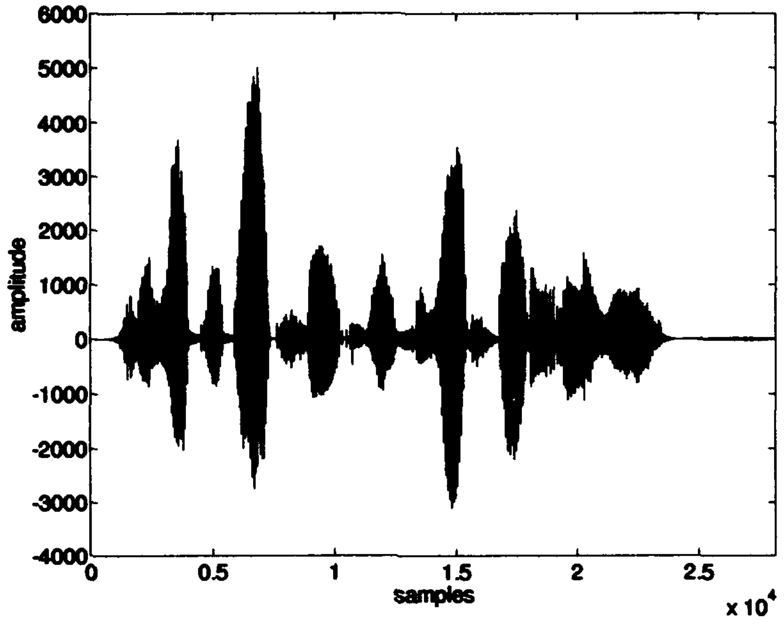


(a)

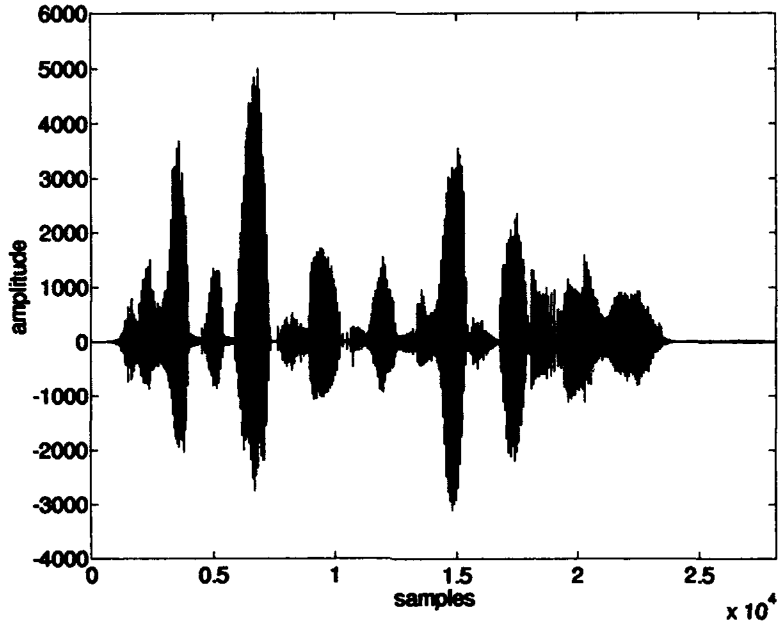


(b)

Figure 4.9. (a) System 3 and (b) System 1 Reconstruction of the Word "Dark"

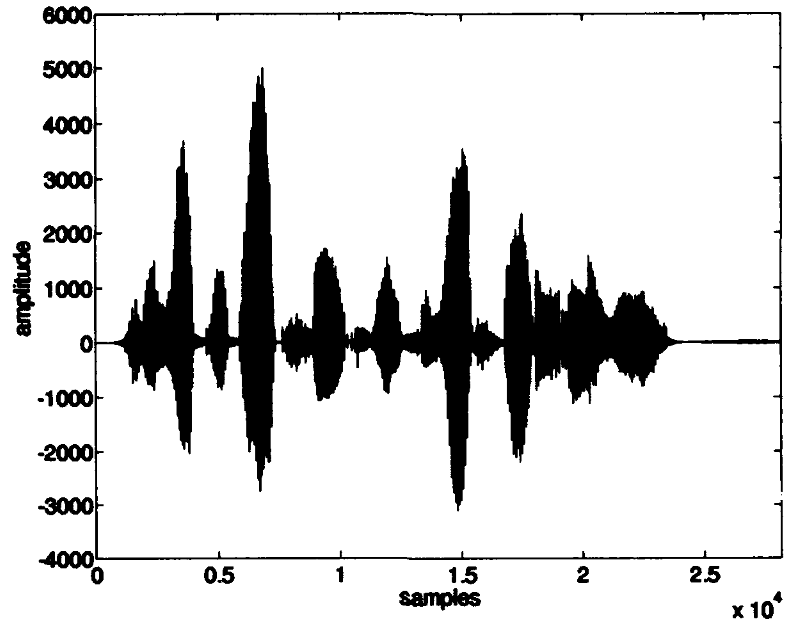


(a)

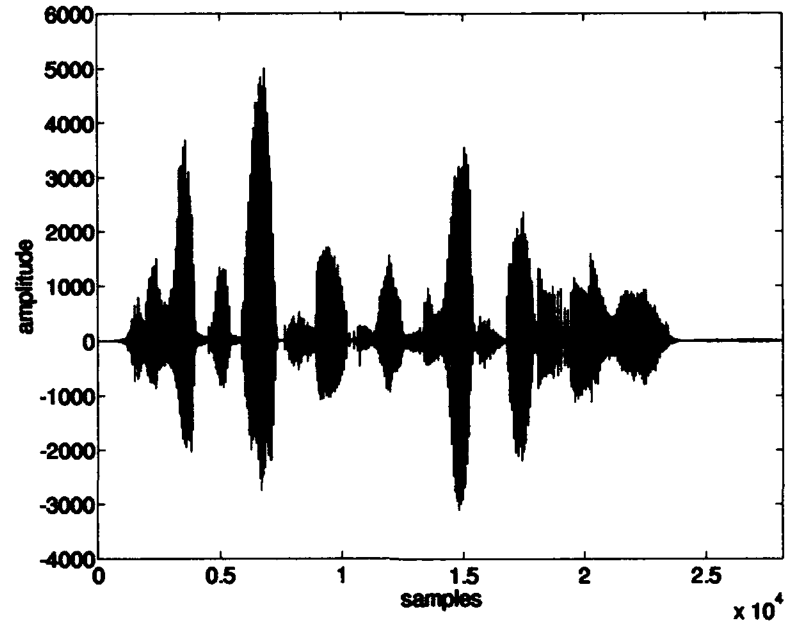


(b)

Figure 4.10. (a) Utterance of the Sentence "She had your dark suit in greasy wash water all year" and (b) Reconstruction of Sentence Using System 1 (1x)



(a)



(b)

Figure 4.11. (a) System 2 (log) (b) System 3 (sinh) Reconstruction of the Sentence "She had your dark suit in greasy wash water all year" at zero compression. This is due to the fact that the basic system had the fewest multiplies and additions. Another system may show much greater promise when there is a high degree of compression of the coefficients.

To observe the effects of compression, error criteria was collected using 10 different data sets. The data consisted of five different sinusoid signals and five different speech signals. The sinusoids varied in frequency and complexity, while the speech segments were short words and sounds consisting of both voiced and unvoiced speech: "dark," "she," "wash," "s," and "ear." All ten signals are plotted in Appendix A. The amount of compression was also varied for each signal: 0%, 80%, and 90% compression were used for each data set. This was done to ensure that a particular system does well at both high compression rates as well as at zero threshold. A total of 30 data files were used for each transform system (each of the ten data sets compressed to three different levels). A number of variables were held constant throughout the experiments, including: the data length (1920 points), window size (128 points), the amount of overlap (1 point), and the basis-set (sinusoidal basis described in Chapter II).

Tables 4.5, 4.6, and 4.7 list the average reconstruction errors for systems 1, 2, and 3 respectively. The variance of each of the error criteria are recorded in tables 4.8, 4.9, and 4.10.

| *Linear Function* | *Relative* $L_2$ | $L_\infty$ | *RMS* | *RIV* |
|---|---|---|---|---|
| 1 | 0.375381 | 161.326 | 1.27688 | 0.942922 |
| $10x+1$ | 0.398733 | 182.457 | 1.10785 | 0.939677 |
| $\sqrt{x}$ | 0.413016 | 192.152 | 1.03613 | 0.933322 |
| $\log(x)$ | 0.374349 | 428.568 | 1.92029 | 0.964840 |
| $\sinh(x)$ | 3.22121 | 7757.98 | 31.1507 | 0.993657 |

Table 4.5. Average Errors Associated with System 1

The results of the 450 experiments show that the system which consistently performed the best was system 3 (with the non-logarithmic functions; 1, $10x+1$, and $\sqrt{x}$). At every level of compression, with every type of signal, the non-logarithmic system 3 reconstructed the signals with the lowest numerical errors. It is expected that these functions would be identical for system 3,

| *Linear Function* | *Relative $L_2$* | $L_\infty$ | *RMS* | *RIV* |
|---|---|---|---|---|
| 1 | 0.363028 | 213.032 | 1.24357 | 0.950170 |
| $10x + 1$ | 0.398701 | 201.787 | 1.09956 | 0.948737 |
| $\sqrt{x}$ | 0.406773 | 178.925 | 1.01884 | 0.940515 |
| $\log(x)$ | 0.400874 | 370.767 | 1.74659 | 0.956094 |
| $\sinh(x)$ | 2.97311 | 6699.29 | 27.8690 | 0.996466 |

Table 4.6. Average Errors Associated with System 2

| *Linear Function* | *Relative $L_2$* | $L_\infty$ | *RMS* | *RIV* |
|---|---|---|---|---|
| 1 | 0.104472 | 213.757 | 0.685003 | 0.998475 |
| $10x + 1$ | 0.104472 | 213.757 | 0.685003 | 0.998475 |
| $\sqrt{x}$ | 0.104472 | 213.757 | 0.685003 | 0.998475 |
| $\log(x)$ | 1.03833 | 215.032 | 0.687536 | 1.14262 |
| $\sinh(x)$ | 1.03833 | 215.032 | 0.687536 | 1.14262 |

Table 4.7. Average Errors Associated with System 3

because the system (GMW, L $\rightarrow$ Inv L, Inv GMW) performs the inverse functions immediately after the forward function (with only compression in between). The logarithms ($\ln(x)$ and $\sinh(x)$) have a higher distortion across the compressions. The reason for this is that the natural logarithms, due to the nature of the function, are thresholding some higher coefficients while keeping smaller ones (for example $|\log(99)| < |\log(.01)|$). This distortion could be avoided by adding a term to the coefficients to ensure a minimum value of 1.0 but, as stated previously, this introduces another distortion when the coefficients are compressed. The sinh function works relatively well (same as the ln function) with system 1, but it distorts the signal quite badly when there is a second transform following the sinh (systems 1 and 2).

| *Linear Function* | *Relative $L_2$* | $L_\infty$ | *RMS* | *RIV* |
|---|---|---|---|---|
| 1 | 0.123228 | 81316.5 | 4.56385 | $8.22438x10^{-3}$ |
| $10x + 1$ | 0.125366 | 98737.7 | 4.36783 | $5.35678x10^{-3}$ |
| $\sqrt{x}$ | 0.128805 | 114023. | 3.16373 | $1.15164x10^{-2}$ |
| $\log(x)$ | 0.104304 | 584534. | 11.4624 | $6.07249x10^{-3}$ |
| $\sinh(x)$ | 21.1635 | $2.37950x10^{+8}$ | 4027.31 | $4.16942x10^{-2}$ |

Table 4.8. Variance of Errors Associated with System 1

| *Linear Function* | *Relative $L_2$* | $L_\infty$ | *RMS* | *RIV* |
|---|---|---|---|---|
| 1 | 0.114239 | 132930. | 4.28904 | $6.35135x10^{-3}$ |
| $10x + 1$ | 0.116201 | 122780. | 4.01349 | $7.76930x10^{-3}$ |
| $\sqrt{x}$ | 0.122680 | 102405. | 3.11571 | $8.78877x10^{-3}$ |
| $\log(x)$ | 0.124123 | 438373. | 9.30463 | $1.21154x10^{-2}$ |
| $\sinh(x)$ | 18.6011 | $1.94071x10^{+8}$ | 3556.47 | $3.73096x10^{-2}$ |

Table 4.9. Variance of Errors Associated with System 2

| *Linear Function* | *Relative $L_2$* | $L_\infty$ | *RMS* | *RIV* |
|---|---|---|---|---|
| 1 | 0.0125197 | 151753. | 1.44600 | $3.89581x10^{-6}$ |
| $10x + 1$ | 0.0125197 | 151753. | 1.44600 | $3.89581x10^{-6}$ |
| $\sqrt{x}$ | 0.0125197 | 151753. | 1.44600 | $3.89581x10^{-6}$ |
| $\log(x)$ | 5.73403 | 148896. | 1.42399 | 0.162145 |
| $\sinh(x)$ | 5.73403 | 148896. | 1.42399 | 0.162145 |

Table 4.10. Variance of Errors Associated with System 3

System 3 outperforms the other systems in the $L_2$, RMS, and RIV criteria. The $L_\infty$ value (the maximum point-wise difference between the original and reconstructed signal), alone, is not a good indicator of reconstruction quality. The $L_\infty$ value does indicate whether the output signal has many large spikes in the reconstruction. The RIV criteria produces the most accurate reproduction when its value is equal to one (information in reproduction = information in original). The RIV value should generally not be greater then one, because this would indicate information being added to the signal, however many of the systems distorted the signal badly enough to cause the RIV criteria to be greater than one. Therefore, the average RIV can not be used to accurately determine the quality of the reconstructed signal.

The variance (see table 4.10) of system 3's error values are also the lowest for the $L_2$, RMS, and RIV criteria. The $L_\infty$ case is the one criteria where system 1 has consistently lower values. System 3 (logarithm) produces incongruities and spikes at the window boundaries at high compression ratios. System 1 smoothes out these spikes and thus reduces the maximum point wise error. Figure 4.12(a) shows a portion of the reconstructed "s" using system 3, logarithm, and 80% compression. The same data is reconstructed with system 1 in figure 4.12(b). The large spikes in the first figure have been greatly reduced by the additional Malvar wavelet manipulations of system 1.

Not only are the numerical errors lower for the system 3 case but the graphical and audio reconstructions are also superior to the other methods. The $L_\infty$ measure does not have much bearing on the total sound or graphical quality of the reconstruction. Some of the large spikes manifest themselves as pops in the reproduction, but the overall quality of the signal is superior with system 3.

As the amount of compression increases, the reconstruction errors at the window boundaries lead to higher total reconstruction errors. This can be seen by taking a closer look at a given boundary point for a signal with increasing levels of data compression.

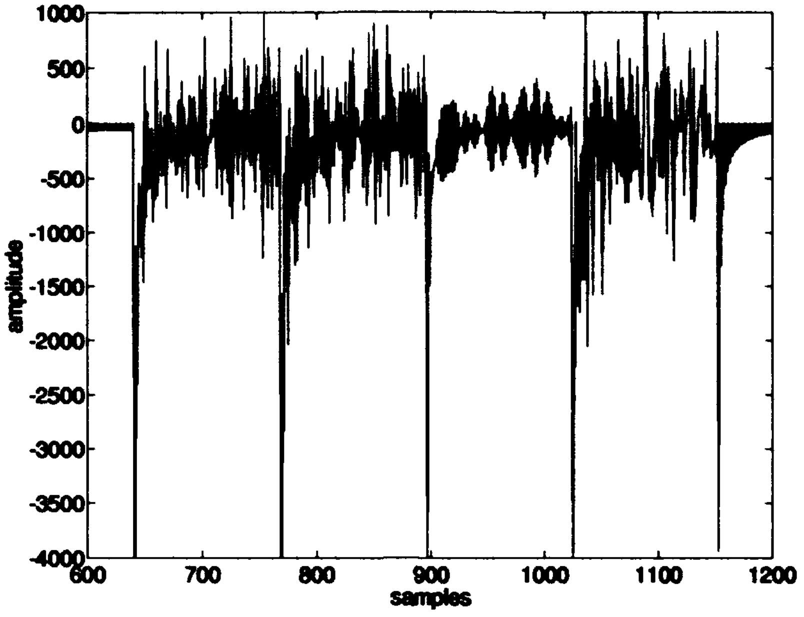
Figures 4.13 and 4.14 show the reconstruction of a sinusoid with compression levels of 0%, 80%, and 98% respectively. Each window is 128 points long, so the first window edge falls at point 128. The degradation of the quality of reconstruction is easily seen as the amount of compression increases. This decay in the quality of reconstruction is possibly due to the minimal amount of overlap (1 point) being used for these experiments. One of the primary benefits of the Lapped Transform is being able to increase the amount of overlap without increasing the amount of data that must be transmitted. This problem raises two new questions.

1. Will an increase in the amount of overlap produce a more accurate recreation of the original signal?

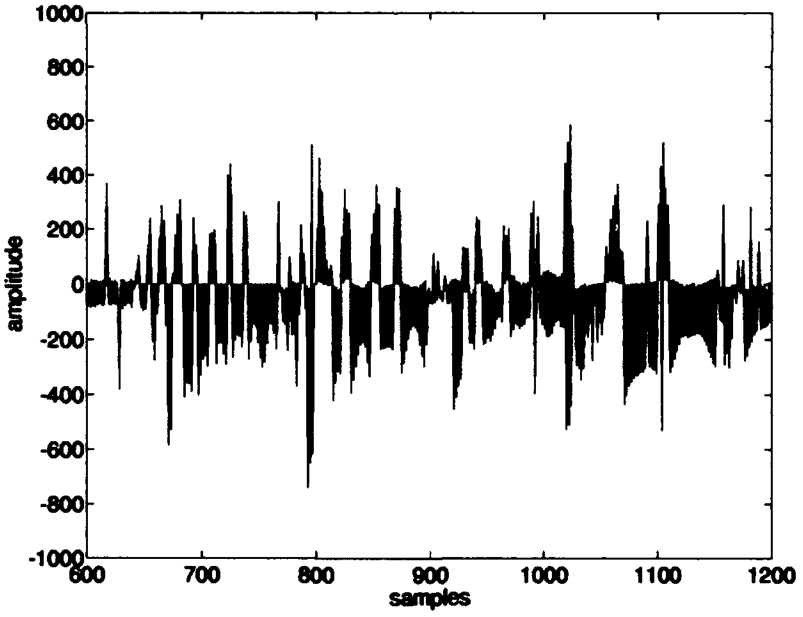
2. Is there an optimal amount of overlap which should be utilized?

These questions will be addressed and tested in Chapter V.

These systems were used to transform entire speech files from the TIMIT database. Plotting and listening to the reconstructed speech led to the same conclusion as above. System 3 (with a non-logarithmic function) produces superior reconstructed signals when compared to the other



(a)



(b)

Figure 4.12. (a) System 3 and (b) System 1 Reconstruction of the "S" Sound with 80% Compression.

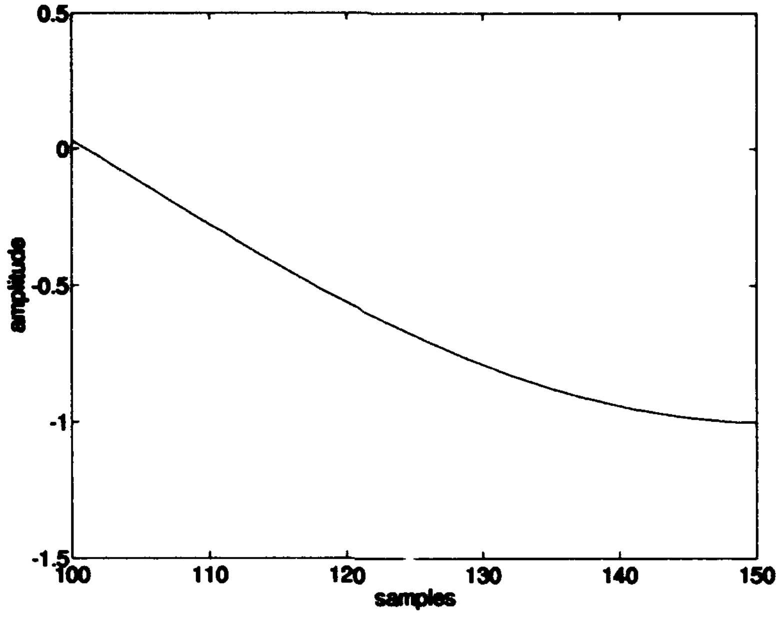
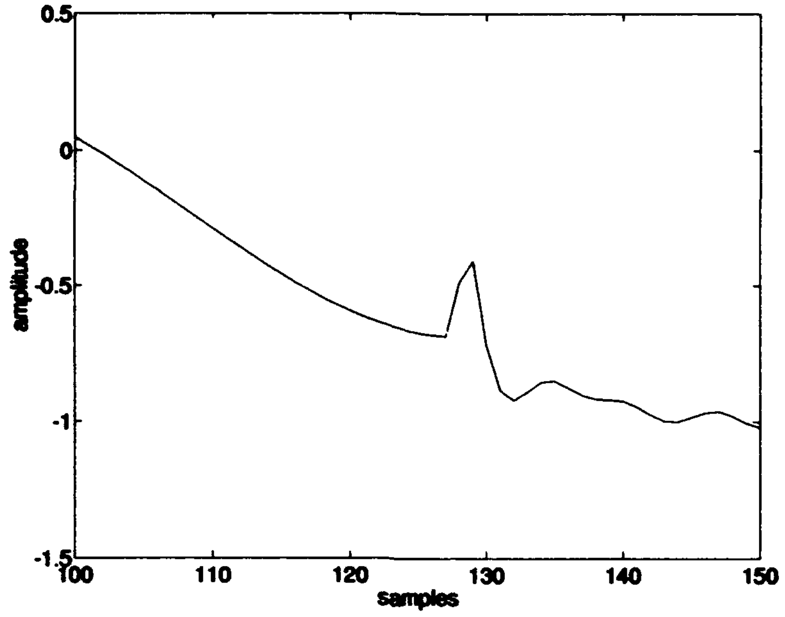


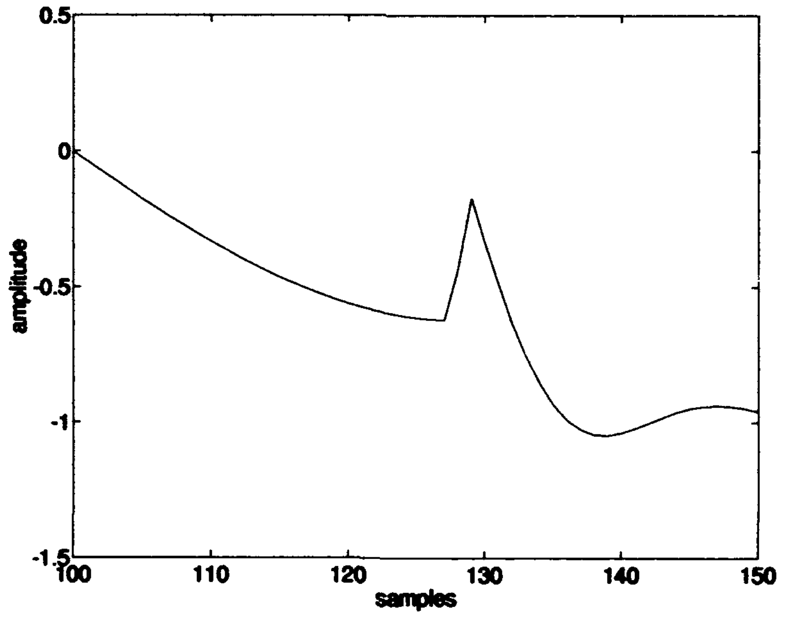
Figure 4.13. Reconstruction of a Sinusoid, 0% Compression, Window Edge = Point 128

systems in the test. The transformed speech from system 3 was consistently more intelligible and had higher reconstruction quality.

It is difficult to determine why Raduenz found that a transform like system 2 led to greater compression capabilities. He was not using a program that provided perfect reconstruction, so the more complicated system may have made up for the imperfections of the basic program. He also may have changed to 16 KHz signals, which do give higher fidelity reconstructions. Raduenz had assumed that changing from 16 KHz to 8 KHz would not effect his results. This is essentially true when there is no compression involved. However, at high compression ratios, the change from 16 KHz to 8 KHz does make a difference. For the experiments from this thesis, all of the signals (including the original speech files) were analyzed at 8 KHz.



(a)



(b)

Figure 4.14. (a) 80% Compression and (b) 98% Compression Reconstruction of Sine, Window Edge = Point 128

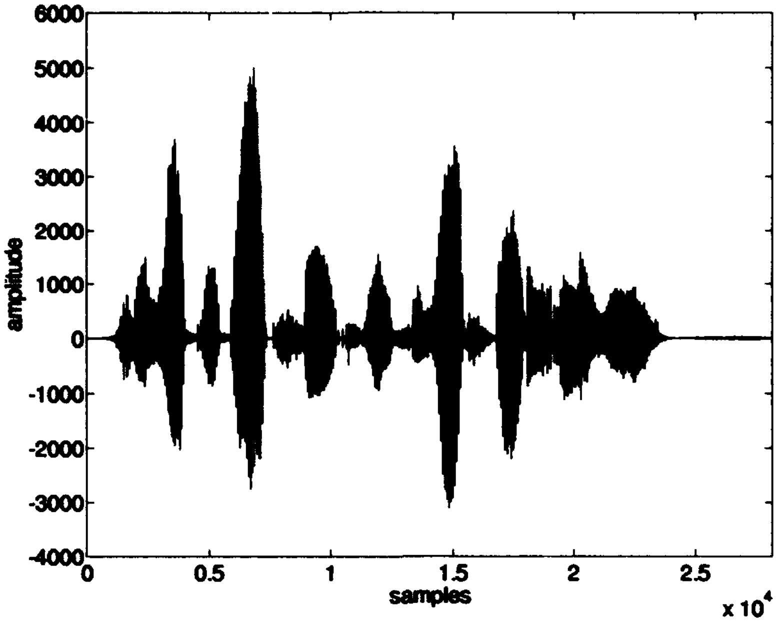
*Complex and Real-Valued Malvar Wavelets*

The Complex and the Sinusoidal Malvar wavelets (CMW and SMW) can now be directly compared, since it has been determined that the basic Malvar wavelet produces reconstructed signals as good as or better than any other system that was considered. It has been shown previously that both of the wavelet types produce very accurate signals even at quite high compression ratios. How do the two systems compare when considering reconstruction quality of speech signals at high compression ratios (90% - 99%)?

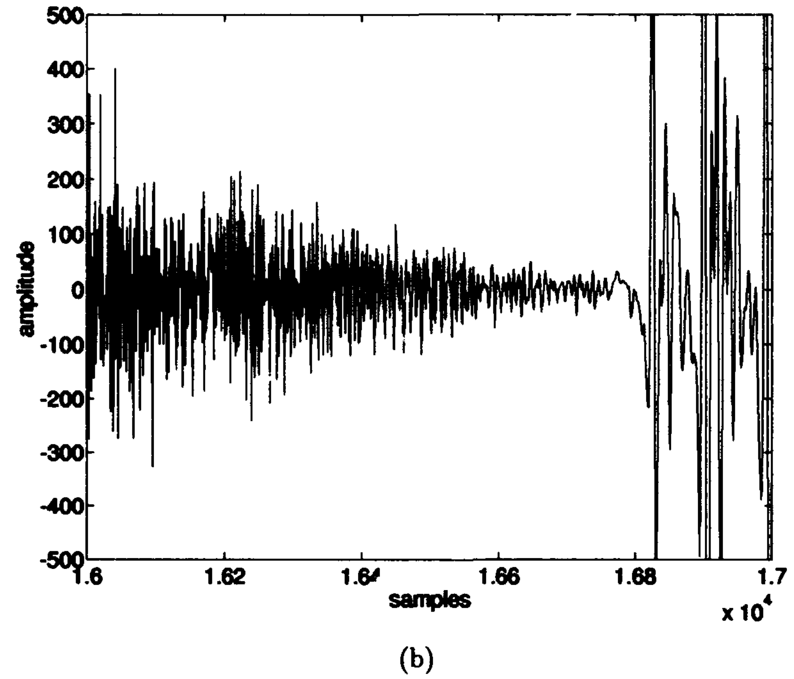
The two Malvar wavelets were used to reproduce a speech file at compression ratios of 90%, 95%, and 99%. The original signal is shown in figure 4.15a. Figures 4.16a and 4.17a show how the two transforms reconstructed the signal at 90% compression. There does not appear to be too much difference except that the Complex MW plot reaches fewer of the peaks of the signal. This can be shown more easily by plotting a smaller segment of the file. Figure 4.15b shows the original speech signal from sample 16000 to sample 17000. The transform reconstructions of this segment are shown in figures 4.16b and 4.17b. It is clear that both transforms are loosing information, but the complex wavelet has lost much more information at the same compression ratio. The difference in reconstruction is also shown by the numerical reconstruction errors. Table 4.11 lists the $L_2$ and RMS errors for the two transforms at 90%, 95%, and 99% compression. The sound of the SMW speech, at 90% compression, was very intelligible. Much of the quality had been lost. There was a mechanical tone of voice with ringing bell-like artifacts. The speech may have warbled and had a "spacey" sound to it, but it was quit understandable. The CMW signal was also intelligible, however it had added a constant low-level noise. There was a constant low popping all the way through the file. At 95% compression (plotted in figure 4.18), the complex transform had many sudden starts and stops at the beginning and ending of words. The low-level popping or clicking was also more evident at 95%. The SMW reconstruction contained fewer sudden starts/stops and did not have the same noise as the CMW version. More intelligibility had been lost, at 95%, but most of the words can still be picked out. The spectral coefficients of the wavelets (real coefficients only) are shown in figure 4.19. The spectral coefficients follow the same general pattern, however the SMW coefficients are much higher magnitude and have a few more peaks within the groups of coefficients. At 99% compression both systems have lost virtually all intelligibility. Speech waveforms are definitely still present (see figure 4.20 a and b), however they have been cut down and smoothed over so much that real vocalization is not present.

|  | *SMW, 90%* | *CMW, 90%* | *SMW, 95%* | *CMW, 95%* | *SMW, 99%* | *CMW, 99%* |
|---|---|---|---|---|---|---|
| $L_2$ | 0.16433 | 0.33098 | 0.26795 | 0.43108 | 0.53188 | 0.67070 |
| RMS | 0.35393 | 0.71379 | 0.57709 | 0.93077 | 1.14551 | 1.44693 |

Table 4.11. $L_2$ and RMS Errors for the Sinusoidal and Complex Malvar Wavelets at 90%, 95%, and 99% Compression.

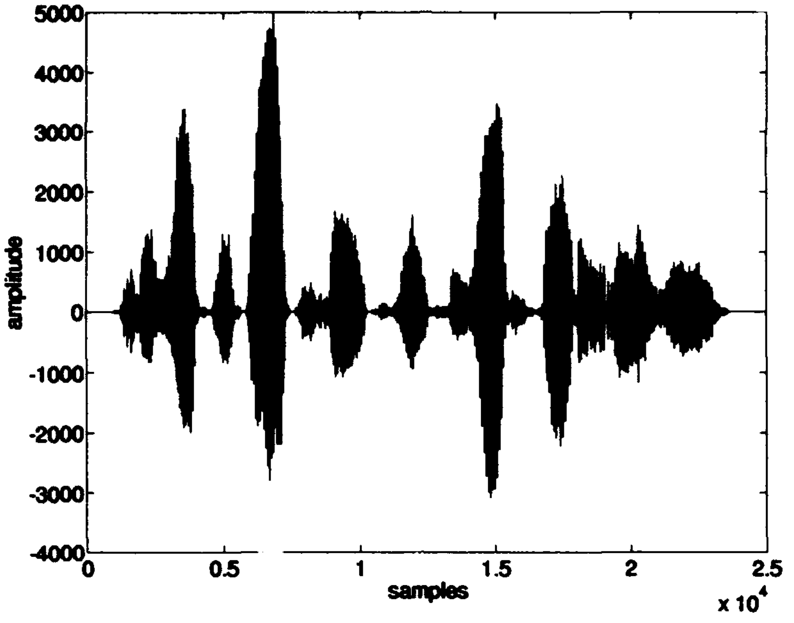


(a)

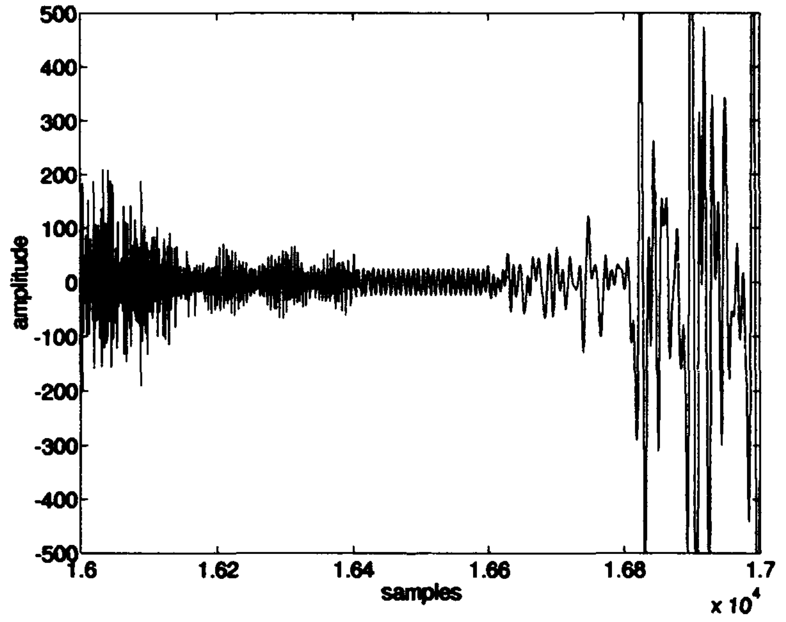


(b)

Figure 4.15. (a) Original Speech Signal and (b) Speech Signal Between Samples 16000 and 17000.

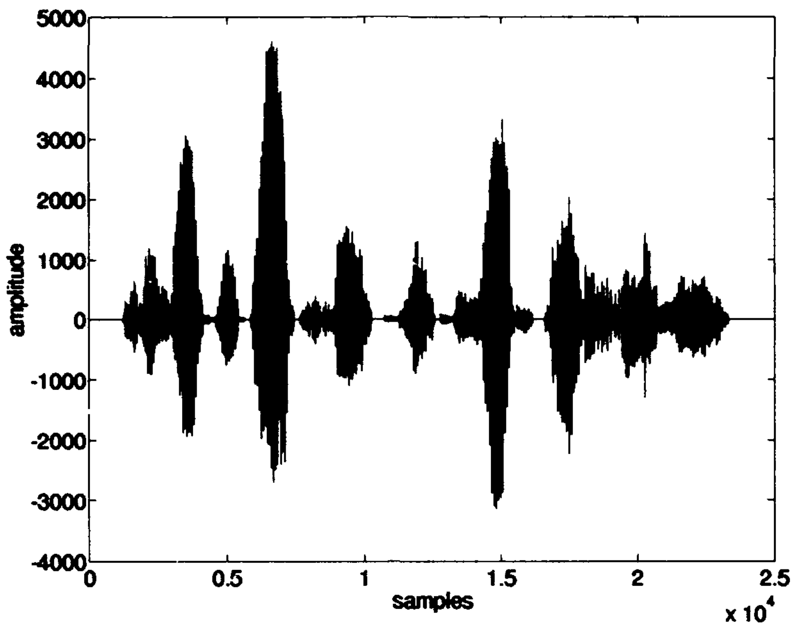


(a)

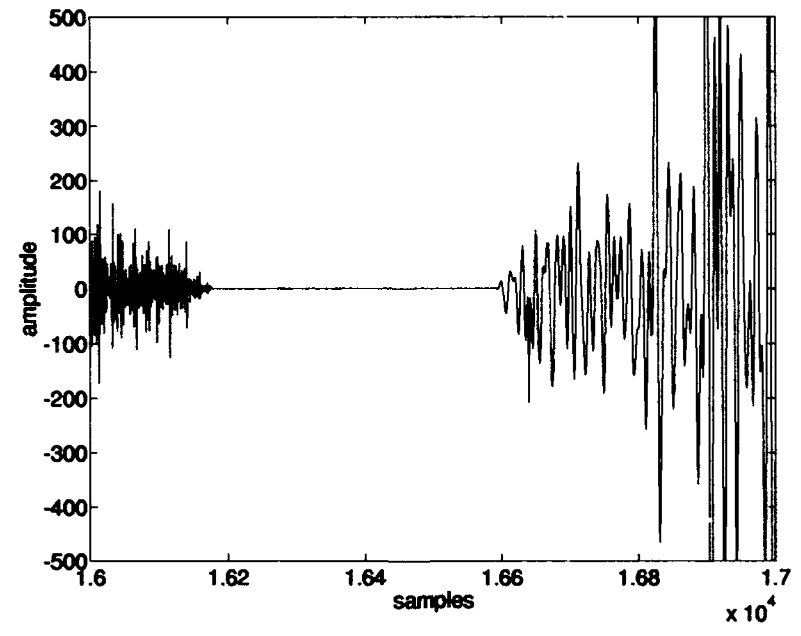


(b)

Figure 4.16. Sinusoidal MW Reconstruction at 90% Compression (a) Complete Signal and (b) Samples 16000 to 17000.

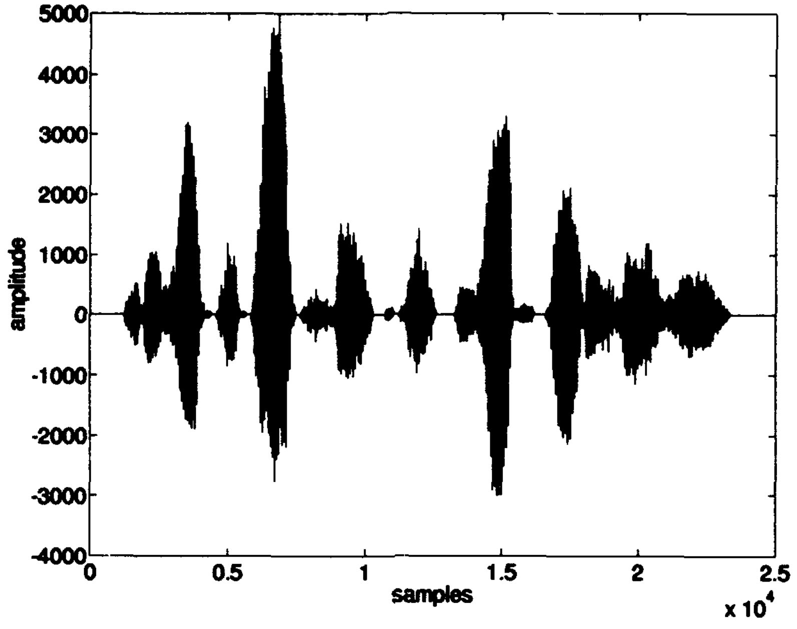


(a)

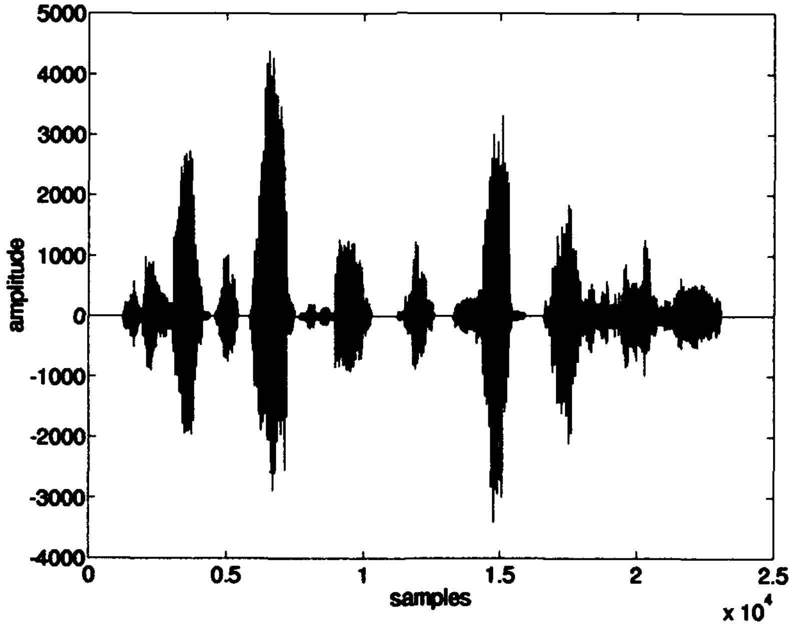


(b)

Figure 4.17. Complex MW Reconstruction at 90% Compression (a) Complete Signal and (b) Between Samples 16000 and 17000.

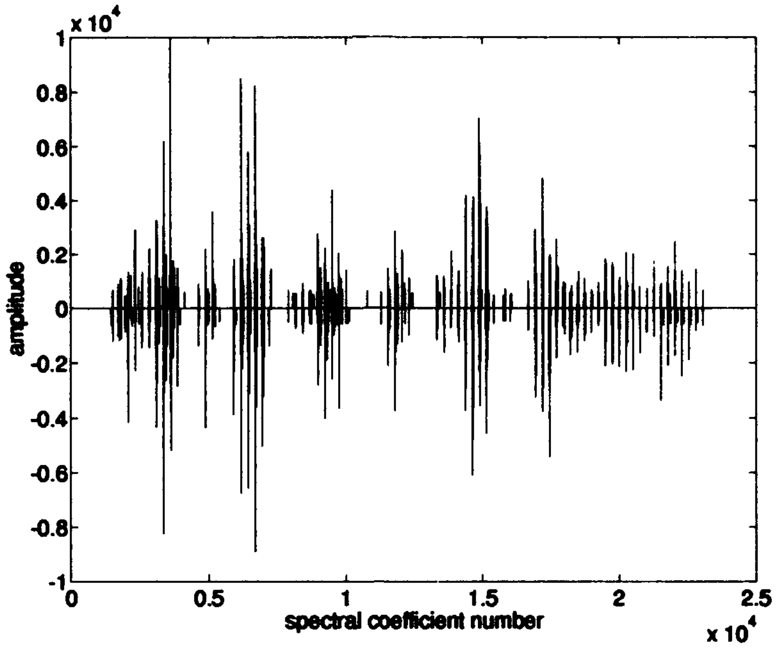


(a)

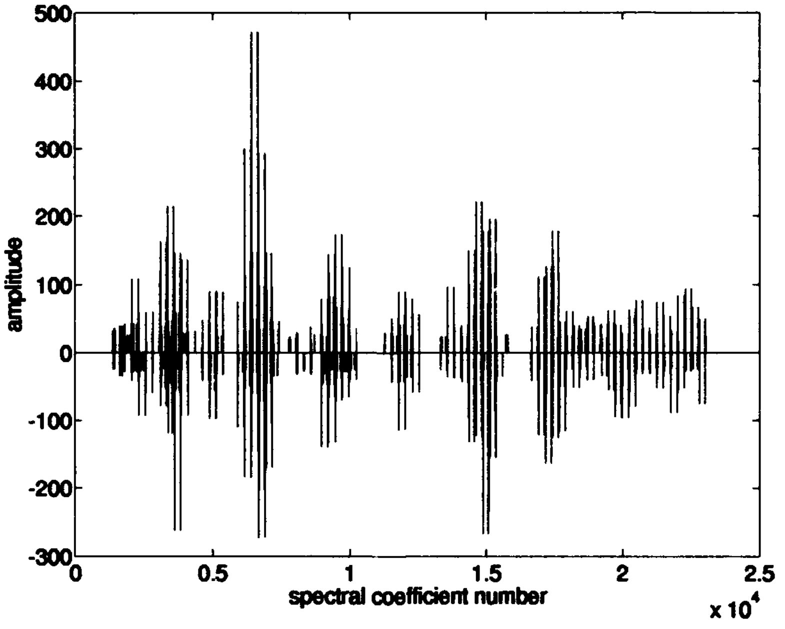


(b)

Figure 4.18. (a) Sinusoidal and (b) Complex MW Reconstruction at 95% Compression.

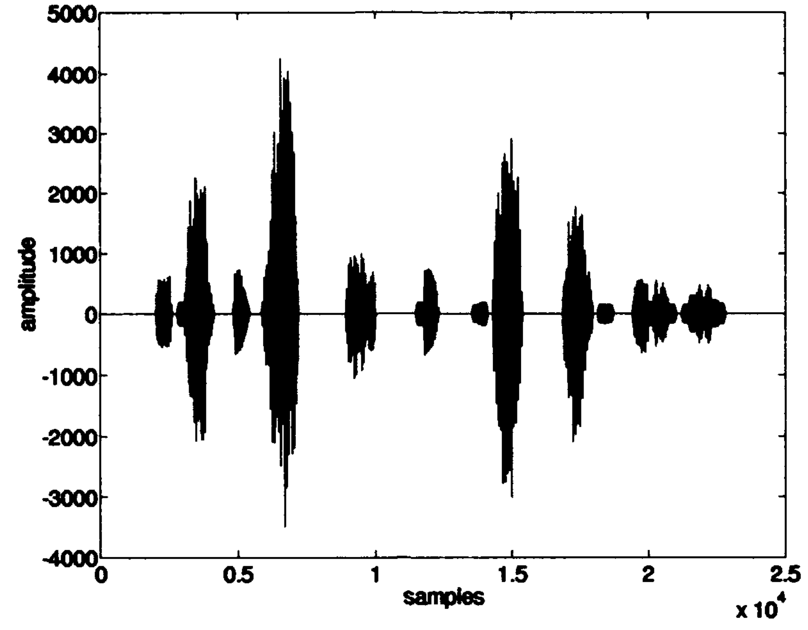


(a)

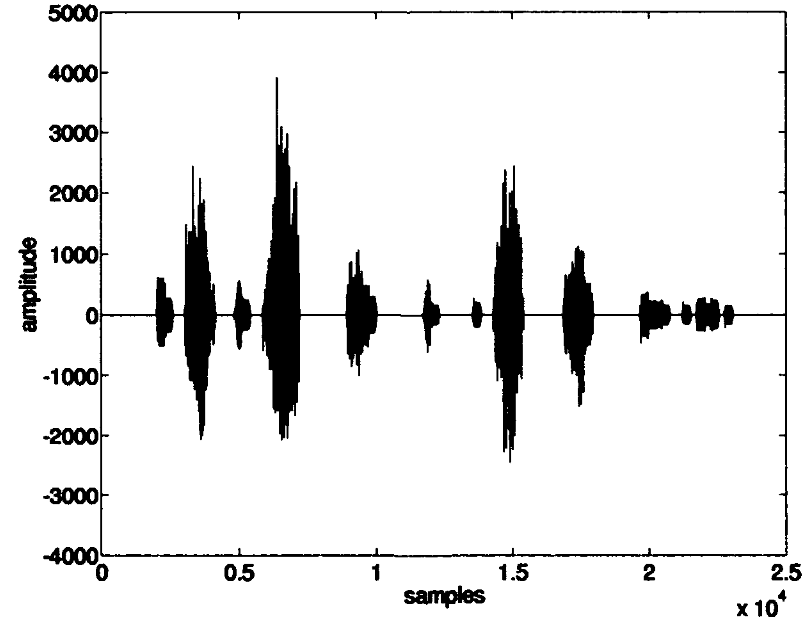


(b)

Figure 4.19. (a) Spectral Coefficients from the SMW and (b) Real Spectral Coefficients from the CMW.



(a)



(b)

Figure 4.20. (a) Sinusoidal and (b) Complex MW Reconstruction at 99% Compression.

*Compression/Data Rates*

The data rate calculations from Chapter III can easily be repeated for the sinusoidal Malvar wavelet. The data rate calculations will use a window size of 128 samples, sample rate of 8 KHz, and 95% compression. The real-valued wavelet produces 128 real coefficients for each transformed window. This transform, like the Discrete Cosine Transform, has no coefficient symmetry. The amplitude of these coefficients can be coded using 4 bits/coefficient. The bits/sec for the spectral values are computed by:

1. 8000 (samples/sec) / 128 (samples/window) = 62.5 windows/second.
2. 95% compression of a 128 point window $\Rightarrow$ 6 samples/window.
3. 4 bits/sample (amplitude for the real coefficient)
   + 7 bits/sample (coefficient position)

   11 bits/sample
4. (11 bits/sample)$\cdot$(6 samples/window) = 66 bits/window
5. (62.5 windows/second)$\cdot$(66 bits/window) = 4.125 Kbits/second

However, over 60% of the system's 4125 bits are used just for positional information. The position of the coefficients is being kept by one 128 point word. This one large word can be broken down into smaller words. For large compression ratios, there will be many zeros within this information word. Instead of transmitting 7 bits ($2^7 = 128$ possible positions) to give the exact position of the coefficient, the position difference between two coefficients could be transmitted. This difference could be limited to a portion of the larger window. For example, the 128 point window can be broken into eight 16 point sub-windows. If there are no coefficients within 16 samples, then a one bit "dummy" coefficient (equal to zero) is inserted to hold the 16th samples place. Thus the total bits sent prior to the window can be reduced with a "divide and conquer" approach (32) (35:4-4). The maximum number of "dummy" bits to break a 128 sample window into 16 sample sub-windows is seven (need a "dummy" at only one end point). The bit rate is then recalculated as:

1. 62.5 windows/second
2. 6 samples/window
3.   4 bits/sample (amplitude for the real coefficient)
   + 4 bits/sample (coefficient position within sub-window)

   8 bits/sample
4. 7 bits/window ("dummy" coefficients)
5. (8 bits/sample)·(6 samples/window) + (7 bits/window) = 55 bits/window
6. (62.5 windows/second)·(55 bits/window) = 3.437 Kbps

The CMW maintains a lower data rate for the same compression ratio, however the intelligibility of the reconstructed signal is slightly worse. Note that the CMW reconstructions for this section were done using the real and imaginary coefficients. It was shown previously that the magnitude and phase gave a better quality reconstruction. Therefore the CMW and the SMW do yield about the same quality reconstructions at comparable compression ratios and the two transforms are virtually identical when both intelligibility and data rate are taken into account.

## *Summary*

The experiments performed and the data collected demonstrate that the most accurate method of compressing and reconstructing digitized data files is also the most basic method; the Sinusoid Malvar wavelet. This conclusion makes intuitive sense; by performing just the forward GMW and then compressing the data, there are fewer calculations to be made and therefore a higher degree of accuracy may be maintained. Experiments also demonstrated that intelligible speech signals can be obtained for 3 Kbps or less using both the real-valued and complex-valued Malvar wavelet.

# *V. Overlap Optimization For Real-Valued Malvar Wavelets*

## *Introduction*

It was shown in the previous chapter that at high compression rates it may be necessary to increase the amount of overlap between adjacent intervals in order to smooth-out the transition. There is no benefit to increasing the amount of overlap for small amounts of compression, because the reconstruction is nearly flawless as it is. However, when the signal is being compressed 80% to 99% , increasing the overlap can improve the reconstruction greatly. The advantage of increasing the amount of overlap is shown in figures 5.1(a) and (b).

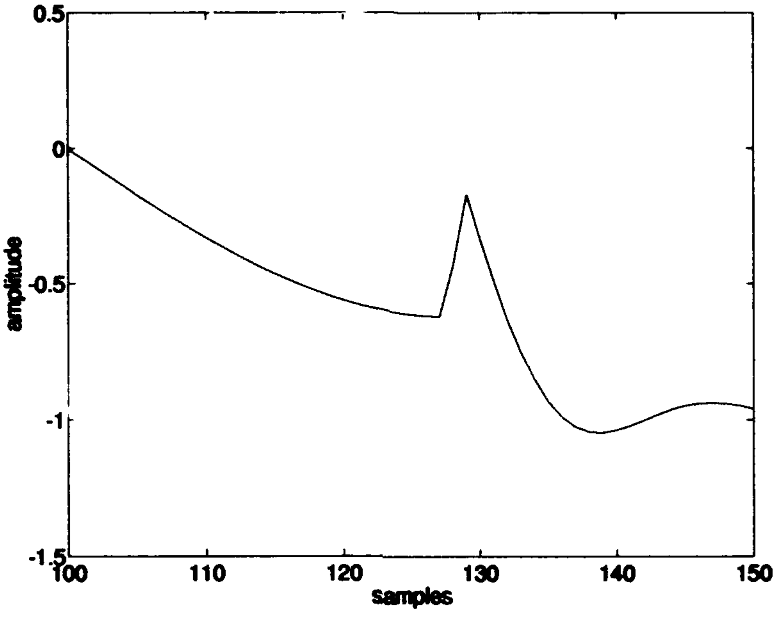
Figure 5.1(a) is the reconstruction of a sine wave after 98% compression of the transform coefficients. The boundary between blocks has been severely distorted. The next plot shows the same signal with the same amount of compression, but with 50% (64 point) overlap. The distortion has been eliminated. The question that needs to be answered is whether it is necessary to always maximize the amount of overlap or is there some threshold where continuing to increase the overlap yields limited returns.

## *Selecting An Error Criteria*

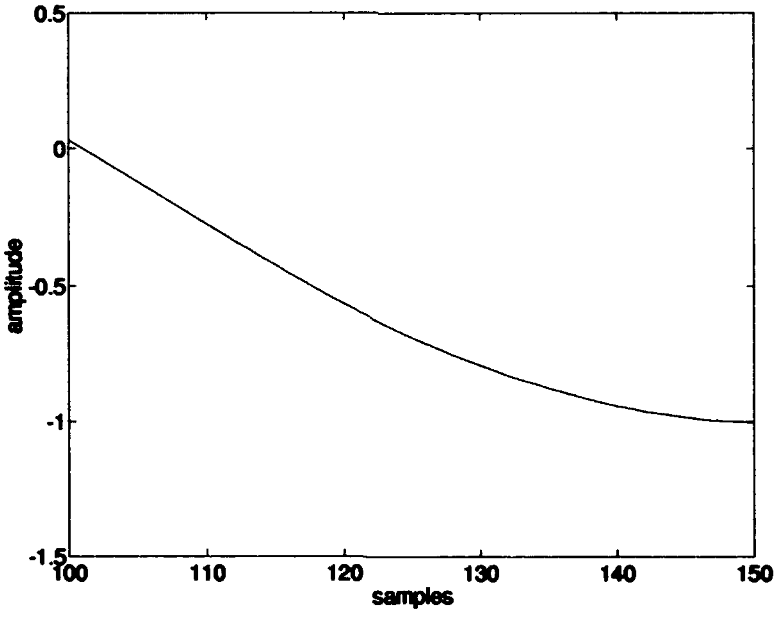
Some sort of numerical error criteria must be used in order to be able to have a statistically meaningful confidence interval for the amount of overlap. Chapters III and IV showed that, in general, the lower the error value the better the reconstruction. Table 5.1 shows how the error criteria of Chapters III and IV vary with increasing compression for a given signal.

| *Percent Compression* | *RIV* | *Relative $L_2$* | *RMS* | *Squared* |
|---|---|---|---|---|
| 0% | 0.99997 | 0.01263 | $6.592x10^{-7}$ | 0.99984 |
| 99% | 0.94209 | 0.52631 | $4.168x10^{-3}$ | 0.72582 |
| 100% | $-\infty$ | 1.00000 | $7.954x10^{-3}$ | 0.00000 |

Table 5.1. Error Criteria Comparison



(a)



(b)

Figure 5.1. (a) 1 Point Overlap and (b) 50% Overlap Reconstructions of a Sinusoid, 98% Coefficient Compression

Results from the previous chapters demonstrate that the RIV and the $L_\infty$ criteria are not very good reconstruction predictors. The Rényi Information Value can be skewed because of the non-linearity of the compression. The RIV is also always very close to 1.0 and therefore does not differentiate between the reconstructions very well. The $L_\infty$ value does give information on the size and magnitude of extraneous spikes in the reproduced signal. These spikes are not always to worst part of the reproduction. Some signals were reproduced with large spikes, but were otherwise very close to the original signal; while other reconstructions were very smooth signals but did not match the original very well.

The two error criteria that do follow the reconstruction fairly well are the relative $L_2$ and the RMS errors. The $L_2$ was selected because it is normalized between zero and one (see table 5.1). This normalization will make comparisons between signals and percent overlap more meaningful.

## *Comparison Study*

A comparison study was performed to analyze the effects of overlap with the amount of reconstruction error. The ten signals that were utilized in the experiments of Chapter IV were used in this study (the plots of the original data are shown in Appendix A). Eight values of overlap were selected: 1 point, 5%, 10%, 15%, 20%, 30%, 40%, and 50%. The most meaningful representation of the results is a graph of the error ($L_2$) versus the percent overlap. The following plots are the results of implementing the Generalized Malvar wavelet and the ten different signals:

These figures demonstrate a definite correlation between the percentage of overlap and the level of reconstruction error. More importantly, the plots show that there is generally a point where the slope of the error curve decreases. The reconstructed signals have a sharp decrease in the amount of error through 5 to 15% overlap and then the slope begins to level out.

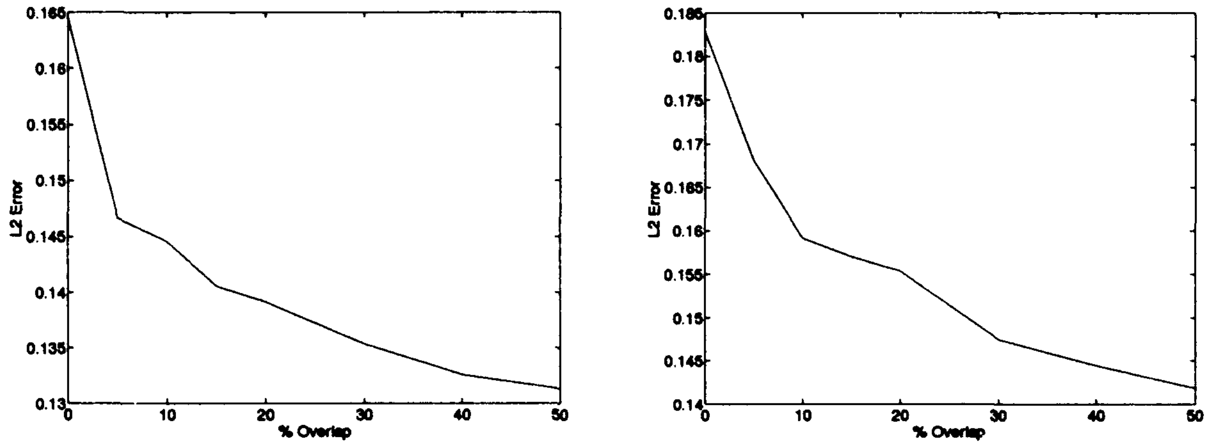


Figure 5.2. $L_2$ Error Versus Percent Overlap (a. "Wash", b. "Dark")

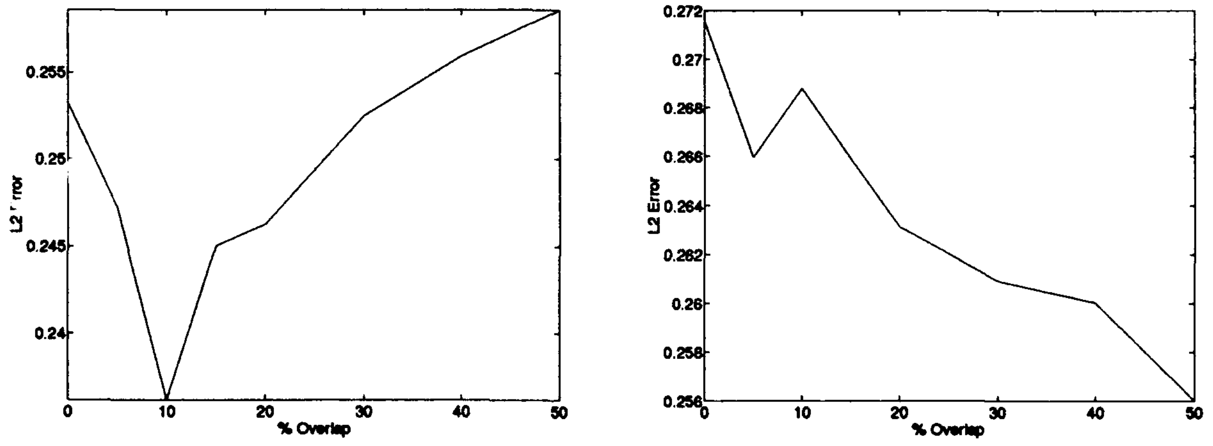


Figure 5.3. $L_2$ Error Versus Percent Overlap (a. "S", b. "She")

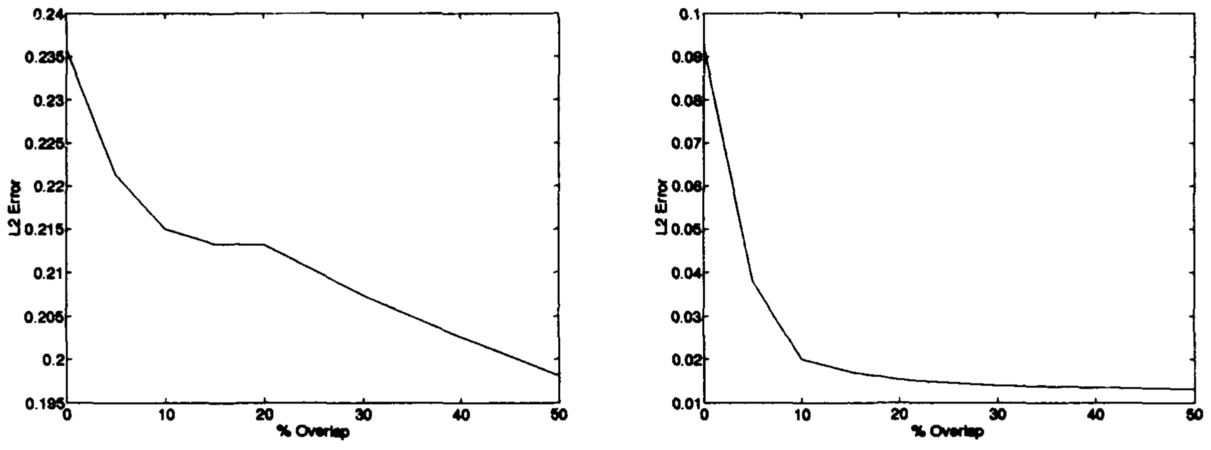


Figure 5.4. $L_2$ Error Versus Percent Overlap (a. "Ear", b. Sine)

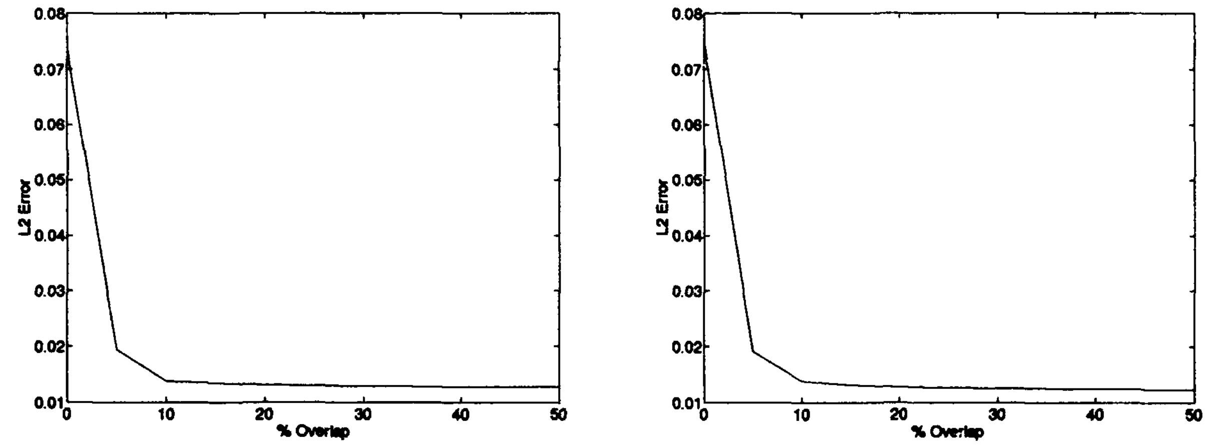
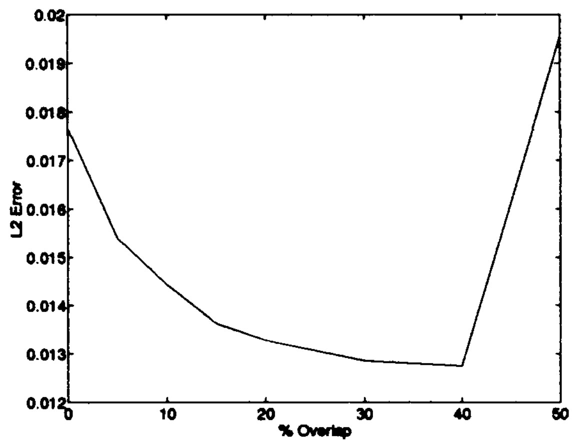


Figure 5.5. $L_2$ Error Versus Percent Overlap (Two Low Freq Sines)



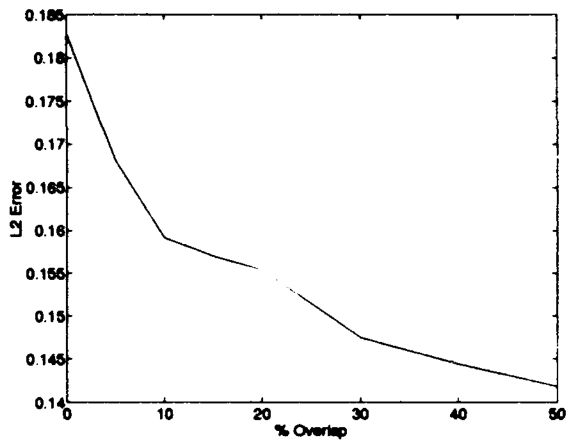


Figure 5.6. $L_2$ Error Versus Percent Overlap (a. Convolution of Two Sines, b. Damped Sine)

*Speech Signals*

Three different sentences from the TIMIT database where transformed using three amounts of overlap and three amounts of compression. The speech signals were overlapped by 1 point, 10% (26 points), and 50% (128 points) and compressed by 80%, 90%, and 95%. There was a definite decrease in the numerical error as the amount of overlap was increased. The $L_2$ and RMS errors are shown in table 5.2. The RMS error, at 80% compression, improved 8.3% from one point overlap to 26 point overlap and the error improved 16.3% when the 128 point overlap is compared to the one point. All of the sentences showed the same level of improvement; 5-10% at 26 point and 15-20% at 128 point overlap.

The improvement in numerical error can also be found in the graphical display of the signals. Figure 5.7a and b plots the reconstruction of an utterance with one point and 128 point overlaps. Differences are noted in the low amplitude sections of the reconstructions. One area of the signal is expanded in figure 5.8 to show the differences in more detail. The 128 point overlap demonstrates a better ability to reconstruct the ends of the signal and wherever the signal has an amplitude less

| *Overlap(Compression)* | $L_2$ | RMS |
|---|---|---|
| Sentence 1 | | |
| *1 pt (80%)* | $9.7447x10^{-2}$ | $3.2882x10^{-1}$ |
| *26 pt (80%)* | $8.9384x10^{-2}$ | $3.0161x10^{-1}$ |
| *128 pt (80%)* | $8.1582x10^{-2}$ | $2.7529x10^{-1}$ |
| Sentence 2 | | |
| *1 pt (90%)* | $1.5125x10^{-1}$ | $5.1367x10^{-1}$ |
| *26 pt (90%)* | $1.3476x10^{-1}$ | $4.5767x10^{-1}$ |
| *128 pt (90%)* | $1.2046x10^{-1}$ | $4.0912x10^{-1}$ |
| Sentence 3 | | |
| *1 pt (95%)* | $2.9503x10^{-1}$ | $6.3541x10^{-1}$ |
| *26 pt (95%)* | $2.7649x10^{-1}$ | $5.9547x10^{-1}$ |
| *128 pt (95%)* | $2.5293x10^{-1}$ | $5.4474x10^{-1}$ |

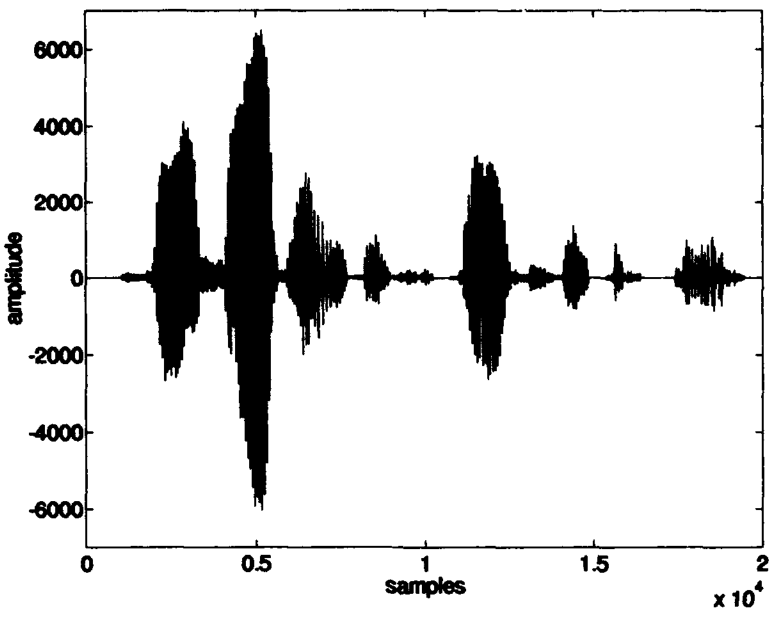
Table 5.2. Reconstruction Errors Versus Amount of Overlap and Percent Compression

than a few hundred. The higher overlaps gained more detail for the signal, but did not improve the peaks or overall shape of the reconstruction.

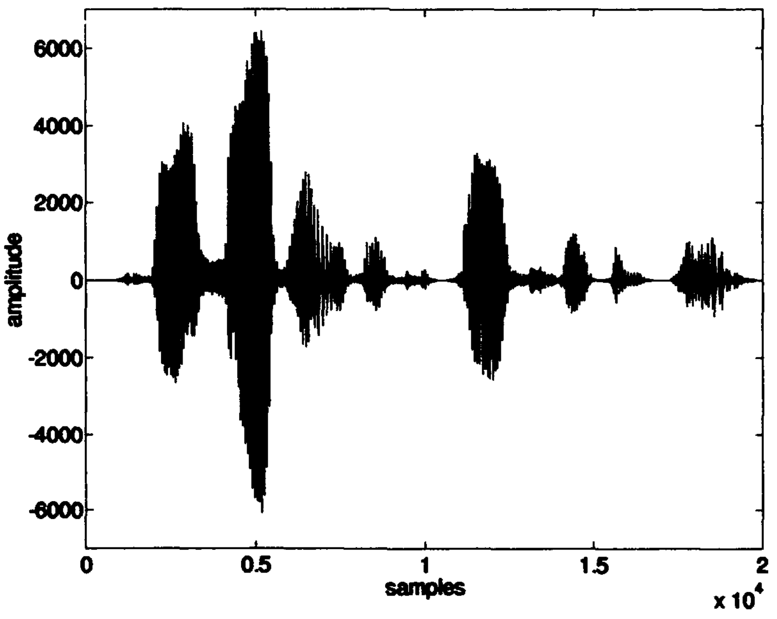
The improvements found in the numerical error and in the plots were not as evident in the audio reconstruction. The increased overlap did reduce some of the background noise, but the increase did little to actually improve the quality or intelligibility of the speech signal. The intelligibility of the sentence with one point overlap was not very different from that of the signal with 128 point overlap because these low level details are not as important to the human ear as the high level peaks and the overall shape of the speech signal.

## *Summary*

The exact amount of overlap which should be used has not been analytically determined. However, for speech signals it was determined that a one point overlap performs as well as the maximum amount of overlap. The percent overlap used for other types of signals may be more important and will depend upon the level of reconstruction desired and the limits on the amount of pre/post processing to be done. Figure 5.1 demonstrates that there is a real improvement to be made by increasing the overlap. For Malvar wavelets, the amount of overlap does not effect the

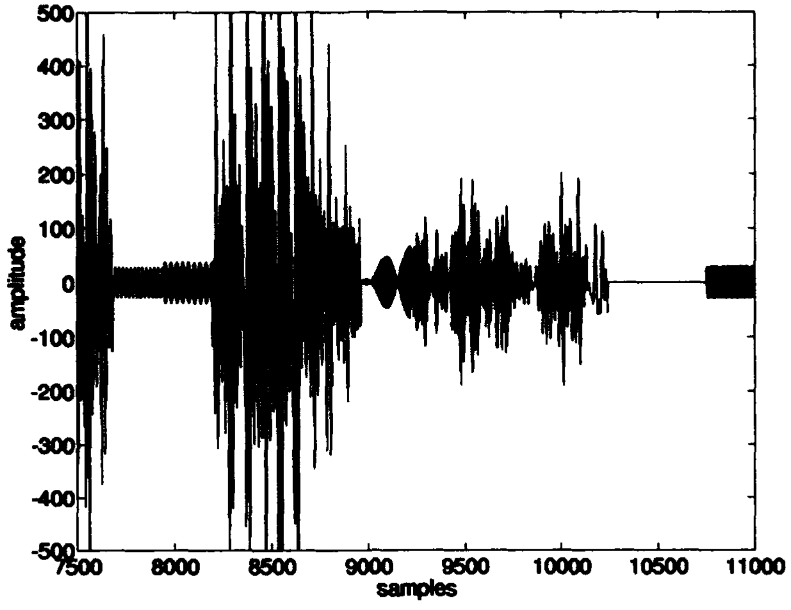


(a)

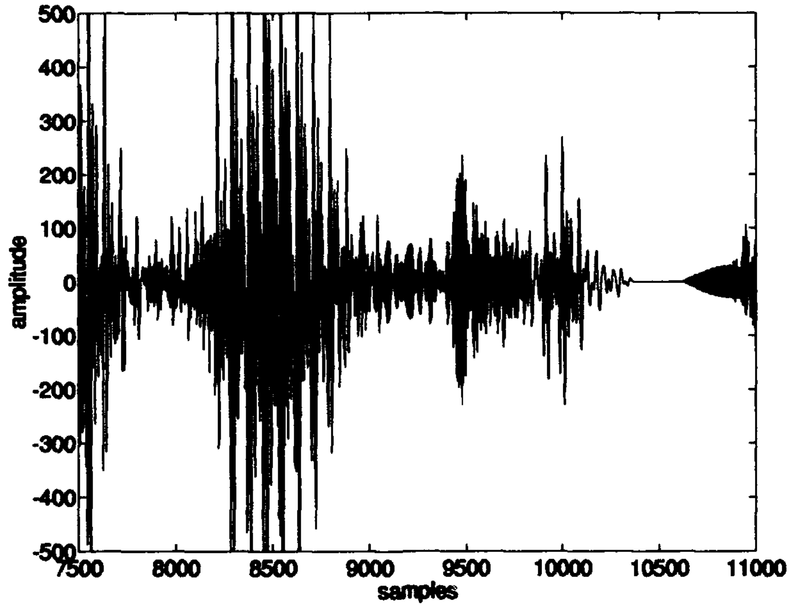


(b)

Figure 5.7. Reconstruction of the Sentence, "A Boring Novel Is A Superb Sleeping Pill", Using (a) 1 Point Overlap and (b) 50% (128 Point) Overlap



(a)



(b)

Figure 5.8. Reconstruction of the Same Sentence (From Sample 7500 to 11000) Using (a) 1 Point Overlap and (b) 50% (128 Point) Overlap

bit rate. The lapped transform, for uncompressed data, sends the same "N" number of samples regardless of the overlap amount. In other words, the frame size, $[a_j, a_{j+1}]$, remains the same, but the amount of overlap with the adjacent frames, $[\epsilon_j, \epsilon_{j+1}]$, changes. There are many instances where the processing available, prior to transmission or after reception, is quit acceptable to allow for greater overlap. Increasing the amount of overlap will increase the number of computations required prior to transmission and after reception. The area that gives the best return for the least number of computations is the 5 to 10% overlap region. Most signals have improved significantly by the 10% mark. When the slope of the curve levels out, the user is spending more on computations and getting less of a return (in the form of a high quality reconstructed signal).

# *VI. Real-Valued Homomorphic Filtering*

## *Introduction*

The primary use of Homomorphic filtering in the literature is to perform deconvolution. In this way, a signal may be separated into its components; whether they be a transmitted signal convolved with the transmission path system function or the vocal tract system function convolved with the vocal cords system function. The most common form of homomorphic filtering is the complex cepstrum. But, as noted in Chapter II, the complex cepstrum is computationally expensive to utilize because of the phase discontinuities.

The discrete cosine transform (DCT), discrete sine transform (DST), and the lapped orthogonal transform (with a DST-like basis, the Generalized Sinusoidal Malvar Wavelet (GMW)) (34), (18) are, by definition, real functions (28). Therefore, if the DCT, DST or GMW is used as part of a homomorphic filter, we may reap the benefits of deconvolution without the problems of phase discontinuities. The basis function used with the Sinusoidal Malvar wavelet in Suter and Oxley's paper is a real-valued function (34):

$$f_k(x) = \sqrt{\frac{2}{N}} \sin\left[\pi\left(k+\frac{1}{2}\right)\frac{x}{N}\right] \tag{6.1}$$

where k is the basis number, N is the window length, and x is the data position within the given interval. The homomorphic filter would be implemented as in figure 6.1; where **G(f)** represents the forward Generalized Malvar wavelet (GMW) and **G(i)** represents the inverse GMW, **x(n)** is the original signal and **x'(n)** is the reconstructed signal, and **L** shall be called the homomorphic coefficients. Because the resulting coefficients are always real-valued, there is no phase information to track. Is it therefore less computationally expensive to use a real-valued basis function to perform homomorphic deconvolution? The GMW, by definition, uses more than one window to calculate its coefficients. However, all of the signal must lie within a single window when performing

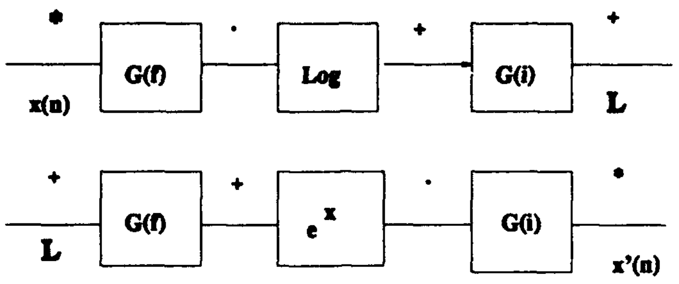


Figure 6.1. A Generalized Malvar Wavelet Homomorphic System

deconvolution. Therefore it is not really the lapped transform that is being utilized, but its basis function and a single window.

*Development*

When using a real-valued function, the analysis of the deconvolution can become very complex and there are certain limitations on what signals can actually be separated.

For example, a signal consisting of an original signal and a simple echo (as in figure 6.2) can be represented as:

$$s_{tot}(t) = s(t) + as(t - \tau) \tag{6.2}$$

where a is the echo amplitude ($0 < a < 1$) and $\tau$ is the time difference between the original signal and the echo ($\tau > 0$). Assume that the signal, $s(t)$, damps out prior to the arrival of the echo, so that the two signals do not overlap. The Fourier transform of this signal can be represented simply as:

$$S_{tot}(w) = S(w) + a\mathcal{F}[s(t - \tau)] \tag{6.3}$$

Now the Fourier transform's shift property ($\mathcal{F}[s(t - \tau)] = S(w)e^{-jwt}$) is utilized to yield:

$$S_{tot}(w) = S(w)[1 + ae^{-jwt}] \tag{6.4}$$

However, for a homomorphic transform (with a different basis than the Fourier transform) to be applied, we must first know the time shift property of the transform.

$$G(f(t)) = \sqrt{2/M} \int f(t) \sin\left[\pi(k+\frac{1}{2})\frac{t}{m}\right] dt \tag{6.5}$$

Thus for $f(t+\tau)$ we have:

$$G(f(t+\tau)) = \sqrt{2/M} \int f(t+\tau) \sin\left[\pi(k+\frac{1}{2})\frac{t}{M}\right] dt \tag{6.6}$$

Now let $p = t+\tau$ so that dp = dt and $t = p-\tau$; also using the trig identity that $\sin(a+b) = \sin(a)\cos(b) - \cos(a)\sin(b)$ yields:

$$\begin{aligned} G(f(t+\tau)) &= \sqrt{2/M} \int f(p) \sin\left[\pi(k+\frac{1}{2})\frac{p}{M}\right] \cos\left[\pi(k+\frac{1}{2})\frac{\tau}{M}\right] dp && (6.7) \\ &\quad -\sqrt{2/M} \int f(p) \cos\left[\pi(k+\frac{1}{2})\frac{p}{M}\right] \sin\left[\pi(k+\frac{1}{2})\frac{\tau}{M}\right] dp && (6.8) \end{aligned}$$

Now we will let $G^c(f(t)) = \sqrt{2/M} \int f(t) \cos[\pi(k+\frac{1}{2})\frac{t}{M}]dt$ so that:

$$G(f(t+\tau)) = \cos\left[\pi\left(k+\frac{1}{2}\right)\frac{\tau}{M}\right] G(f(t)) - \sin\left[\pi\left(k+\frac{1}{2}\right)\frac{\tau}{M}\right] G^c(f(t)) \tag{6.9}$$

The time shift property for a purely sinusoidal basis is therefore a combination of the original basis and a cosine basis.

Another difficulty in using a sinusoidal basis is that the convolution property of the Fourier transform does not hold. The convolution property for our transform is given by:

$$G[x(t) * y(t)] = \sqrt{2/M}\ [G(x(t)) \cdot G(y(t)) - G^c(x(t)) \cdot G^c(y(t))] \tag{6.10}$$

This means that the signal must be able to be directly expressed as two or more terms like our example signal ($s_{tot}(t) = s(t) + as(t-\tau)$) or the signal must have either even or odd symmetry, so that one part of the convolution property (equation 6.10) will be equal to zero. If the signal has even symmetry then the $G^c(x) \cdot G^c(y)$ product will equal zero and, likewise, if the input signal is odd symmetric then the product of $G(x) \cdot G(y)$ will be zero.

The use of a real-valued basis function has some definite merit because there is no need to perform phase unwrapping. However, it is limited by the type of signal that can actually be deconvolved.

*Implementation*

The homomorphic filtering system was implemented with the modified Malvar wavelet program used to perform the homomorphic filtering (System 1 using the natural logarithm/exponential) in Chapter IV. This program accomplishes all the steps of the Sinusoidal Malvar wavelet as outlined in Chapter IV and as proposed by Suter and Oxley (34). Input data ($x[n]$)was created with MATLAB data files and the DARPA TIMIT Phonetic speech database.

| *Function* | Input | Output |
|---|---|---|
| Forward Transform | $x[n]$ | Transform Coeffs ($\alpha$) |
| Natural Log | $\alpha$'s | $\ln \alpha$'s |
| Inverse Transform | $\ln \alpha$'s | Homomorphic Coeffs (**L**) |
| | | |
| Forward Transform | **L**'s | Transform Coeffs ($\mathcal{L}$) |
| Exponential | $\mathcal{L}$'s | $e^{\mathcal{L}}$ |
| Inverse Transform | $e^{\mathcal{L}}$ | Output Signal (x'[n]) |

Table 6.1. Homomorphic Filtering System with Malvar Wavelet

*Deconvolution:* Deconvolution is achieved by filtering the homomorphic coefficients (**L**). Does the homomorphic GMW system deconvolve a process into its basic signals like the complex cepstrum? A signal such as a damped sinusoid with an echo (as in figure 6.2) is often used to demonstrate the deconvolution property of the complex cepstrum.

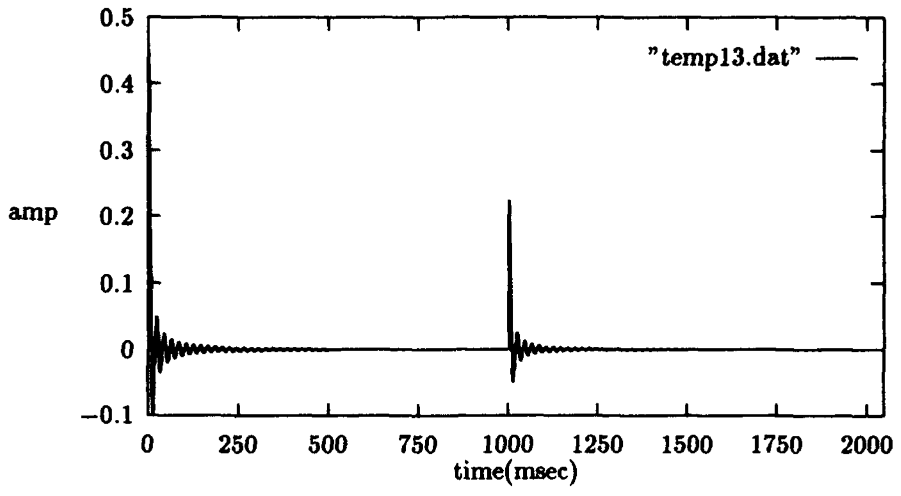


Figure 6.2. Damped Sinusoid with an Echo

The signal is first transformed by the GMW. These wavelet coefficients are plotted in figure 6.3. The next step is to take the natural logarithm of the coefficients ($\alpha$). When using the power or complex cepstrum, the problem of an undefined logarithm will only occur when a coefficient is exactly 0.0. The same procedure as outlined in Chapter IV was used to avoid any null coefficient when using the power or complex cepstrum.

The inverse lapped transform is taken on the coefficients following the natural logarithm. This set of values (plotted in figure 6.4) are the homomorphic coefficients. The homomorphic coefficients are log amplitude values plotted against time (known as quefrency for cepstral users). These homomorphic coefficients can now be filtered (liftered) to deconvolve the signal. The first filter implemented is an all-pass filter to demonstrate that the process is completely reversible.

*Reconstruction:* The homomorphic coefficients are transformed by the following functions:

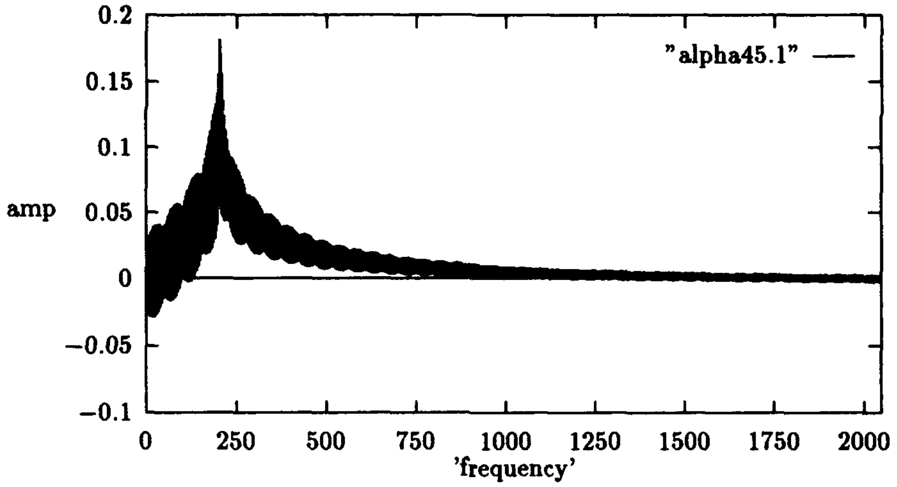


Figure 6.3. Malvar Wavelet Coefficients

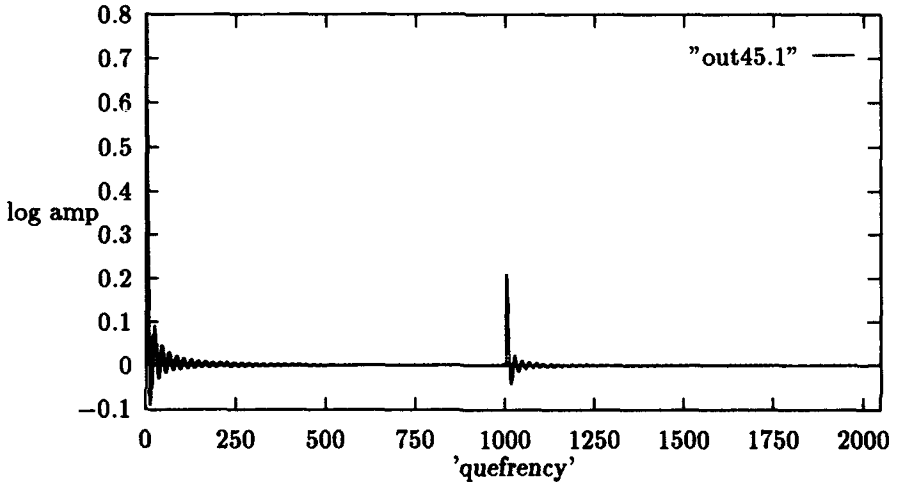


Figure 6.4. Homomorphic Domain

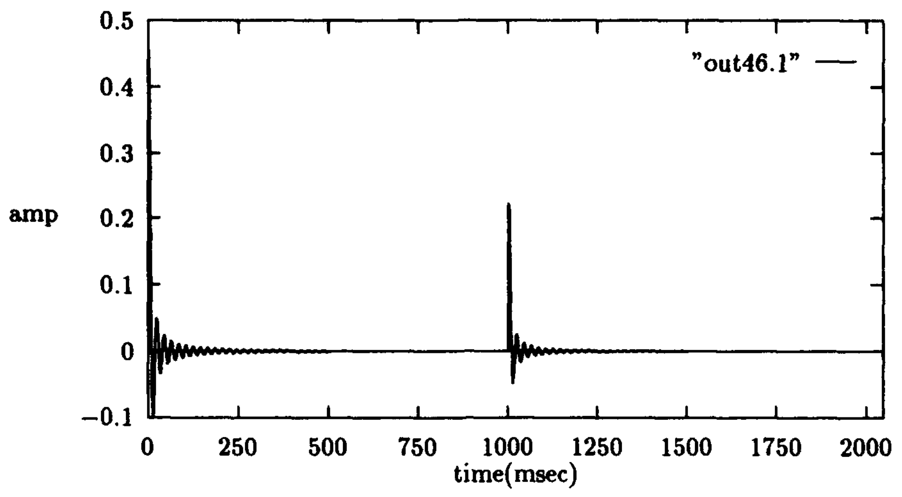


Figure 6.5. Perfectly Reconstructed Damped Sinusoid with Echo

1. Forward Sinusoidal Malvar Wavelet.
2. Exponential ($e^{\mathcal{L}}$).
3. Inverse Sinusoidal Malvar Wavelet.

The resulting signal is a perfect reconstruction (with a normalized RMS error = $6.3587x10^{-10}$) of the original damped sinusoid and echo (see figure 6.5).

In order to remove the echo, the high quefrency information must be removed. The desired signal (original damped sine) is a high frequency signal that is transformed to the low quefrency during the homomorphic filtering process. A low-pass filter is used on the quefrency domain to eliminate the high quefrency (low frequency) components of the signal. The high frequency (low quefrency) information is maintained and the original 50 Hz signal is reproduced as shown in figure 6.6.

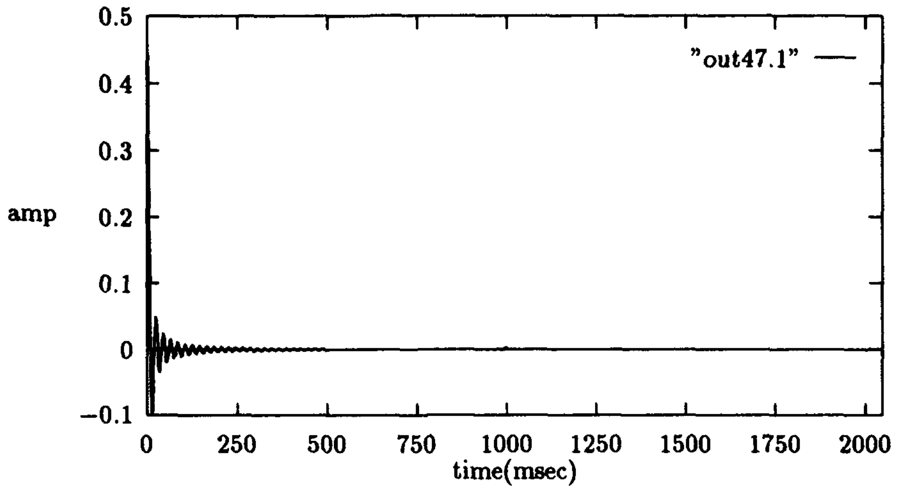


Figure 6.6. Reconstructed Damped Sinusoid with No Echo

*Summary*

This chapter demonstrated that it is possible to utilize standard deconvolution procedures with a real-valued homomorphic system. The primary difficulty with this procedure is that only certain types of signals are candidates for deconvolution. Signals that can be deconvolved with a real-valued system are limited to even/odd symmetric signals or a signal that is able to be directly expressed as two or more terms ($s_{tot}(t) = s(t) + as(t - \tau)$, where $s(t)$ is sufficiently damped).

# *VII. Conclusions*

## *Summary and Conclusions*

The coding of the algorithms and the experiments run with these programs encompass the most important contributions of this thesis effort. During the coding effort of the Complex Malvar wavelet algorithm, the theory behind the transform was validated. This untested algorithm was coded and bugs were worked out of both the program and the theory. The validation effort was used to ensure that the program (listed in Appendix B) would lead to perfect reconstruction of a signal. Perfect reconstruction and orthonormality was accomplished and compression experiments were run with the CMW. With the exception of rectangular windowed short time Fourier transforms (STFT), this is the first completely orthogonal STFT to be implemented and tested for digital signal processing applications. The CMW provides perfect reconstruction of signals and can reconstruct signals fairly well at high compression ratios, it was found that the CMW can reconstruct intelligible speech signals at data rates below 2400 bps. These rates were reached by compression of the transform coefficients, reducing the number of bits/coefficient, and keeping positional information to a minimum. The CMW bay be even more appropriate for image and RADAR processing. Unfortunately, because the CMW was defined so late in the thesis cycle, the transform was not extended to two dimensions or tested with image processing applications.

The Real-Valued Malvar wavelet software was modified and improved to provide more accurate operation and to incorporate more features. The original code was not thoroughly validated by Raduenz and had a few problems that did not allow for perfect reconstruction. The program and theory were tested and scrutinized to identify and eliminate the bugs. The testing ensured that the Ada software would provide an identical reconstruction of any type of signal with varying size windows and overlaps. Previous research in the area of lapped transforms has demonstrated perfect reconstruction for data using a 50% overlap, and mathematically derived conditions for a more general size overlap. This thesis implemented a lapped transform algorithm, based on the work of Suter and Oxley, that achieved perfect reconstruction of a signal with a one point overlap between intervals. This result is significant because a transform based on a one point overlap requires half the processing of a transform based on 50% overlap.

Three different improvements were made once the basic program was providing error free operations. The basic program performed finite length window processing by copying the first window of data to the end of the file and then transforming an extra window of data. The new program is able to obtain perfect reconstruction of the first and last interval without copying data or performing multiple calculations on one window. The program "wraps" the data around so that the first and last data point become neighbors and overlapping of the end windows is done directly between the two windows. This is the first technique that allows for a finite length signal to be perfectly reconstructed using the lapped transform without artificially expanding the signal. The Ada software for the non-expansive lapped transform is listed in Appendix C. A package was written to allow the Malvar wavelet programs to recursively find the level of thresholding necessary to provide a given percent compression. This enables the researcher to qualitatively compare the results of different systems at a given level of compression. The second Ada package written for the wavelet programs provides the error criteria for a reconstruction. This also gives the researcher a better tool for studying reconstruction techniques.

Compression capabilities of different lapped transform systems were evaluated and compared. Raduenz had found that a homomorphic system (using the lapped transform, the natural logarithm, and a second lapped transform) had provided superior reconstruction of a speech signal. The three systems (as shown in figures 4.3, 4.4, and 4.5) were validated to ensure they were working properly at zero percent compression. The systems were then evaluated for their reconstructive abilities by comparing numerical errors, graphical displays, and audio reconstructions of many different types of signals at various amounts of compression. In all cases the most basic system (figure 4.5) provided the best reconstruction of the signals. This is important because it allows for much less pre/post processing to gain the best reproduction possible. Data rates close to 3000 bps were found to provide reasonable quality and intelligible speech reconstruction for the real-valued Malvar wavelet.

The lapped transform theory allows for any amount of overlap from one point to 50% of the size of the window, but the theory does not suggest what amount of overlap would be the best. This thesis reports the results of multiple experiments to compare and contrast the amount of overlap to the level of reconstruction. The results show that the numerical and graphical error tends to continue to decrease as the amount of overlap increases, although this is not the case for all signals. The rate at which the error decreases typically leveled out between 5 and 15% overlap. This would suggest that about 10% overlap would be an optimal amount for most signals. Speech signals showed this same pattern of improvement, however the decrease in error could not be heard in the reconstructed speech. The speech signal is improved in low amplitude sections that are not easily differentiated by the human ear. This again suggests that the most basic method, the lapped transform with one point overlap, be used for speech compression purposes. This will save half of the processing required for a 50% overlap. Deconvolution was also studied using the basis from the real-valued Malvar wavelet. Deconvolution was shown to be possible using a sinusoidal function, however there are certain limitations on the signal. The signal must have even or odd symmetry or be able to be directly expressed as two individual terms (such as a damped signal with an echo).

### *Follow-On Research*

The extension of the Complex Malvar wavelet to two dimensions is an intriguing possibility for follow-on work. The orthogonality of the CMW could provide excellent results with image processing applications. The CMW could be substituted into almost any algorithm or application which utilizes the complex spectrum of a signal. A two dimensional Ada program would not be trivial because the FFT (see Raduenz (26)) routine would also have to be extended to two dimensions. The CMW also has some important one dimensional applications that should be researched. Further speech compression experiments could be performed using more elaborate compression and coding techniques. Data rates should be calculated, taking into account the quality of the reconstruction, and the results compared back to a baseline system like LPC-10. The CMW can also be utilized in RADAR signal analysis and to perform complex cepstral processing. Deconvolution or other complex cepstral applications may have higher performance with the CMW. Besides the code presented in this thesis, a phase unwrapping package would have to be produced in order to use the CMW in a homomorphic filter. Similar overlap experiments could be performed with the CMW. It was found that the overlap has little effect on speech signals, but that may not hold true for images.

A hybrid wavelet approach looks promising for future research. Wavelet packets and the Malvar wavelet could be used together for digital signal processing. Wavelet packets could be used to segment a data stream while the Malvar wavelet would then analyze the data within each segment.

The basis functions, window size, and amount of overlap are all variable entities within the Generalized Malvar wavelet theory. All work to date has been accomplished using a set value for these entities throughout a given signal. It may be possible to vary the basis function, size of the window, and the amount of overlap to the changes within the signal. As the statistics of the signal are changing, different basis functions may yield better reconstruction than others. Further compression may also be obtained by scaling the size of the window to the type of signal. A signal that is not changing very rapidly could utilize a large window, while a signal that changes very quickly would probably be best transformed with a smaller window.

A comparison of the lapped transform with a speech coder like LPC-10 would provide greater insight into the capabilities of the algorithm. There are many advanced coding techniques that were not utilized with the lapped transform in this thesis. If these advanced coding techniques could be incorporated with the wavelet compression, then a better comparison could be made. The wavelet technique may be able to provide superior quality and intelligibility for less than 2400 bps.

# Appendix A. *Input Data for Experiments*

The following signals were used as input for the system and overlap experiments of Chapters III, IV, and V.

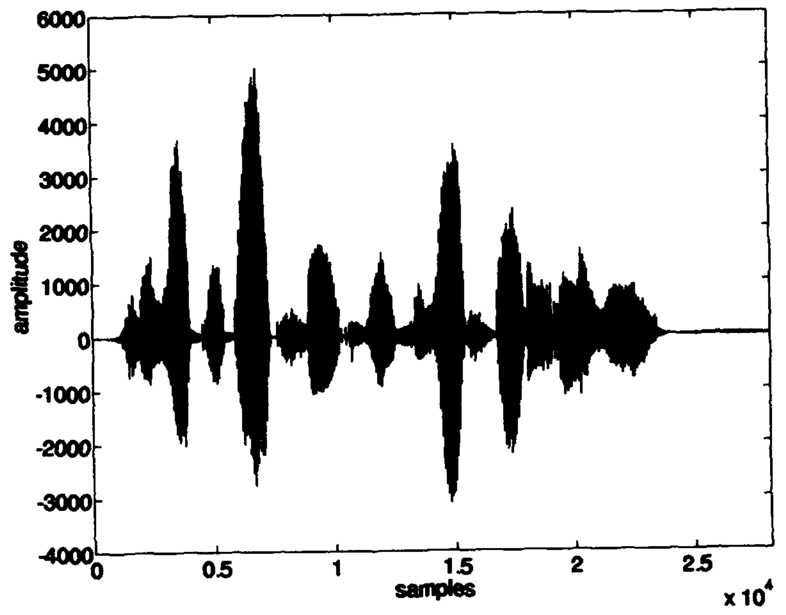
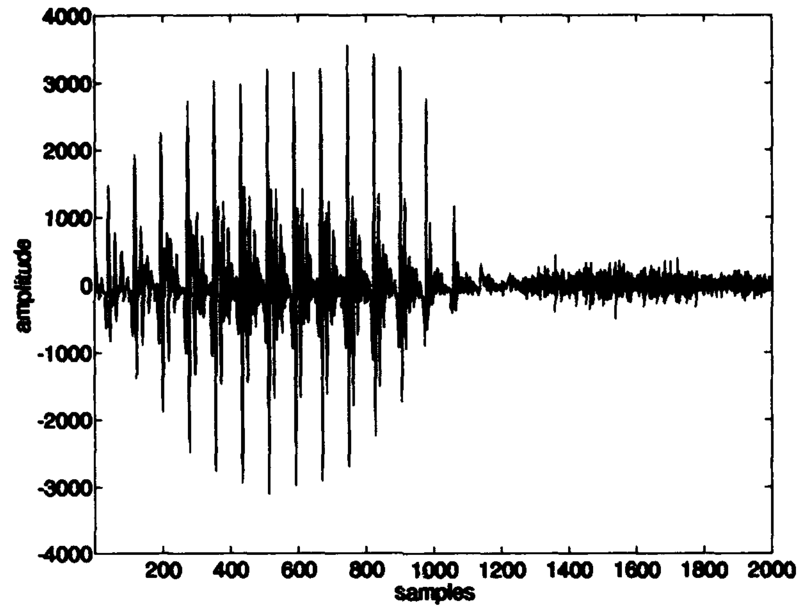
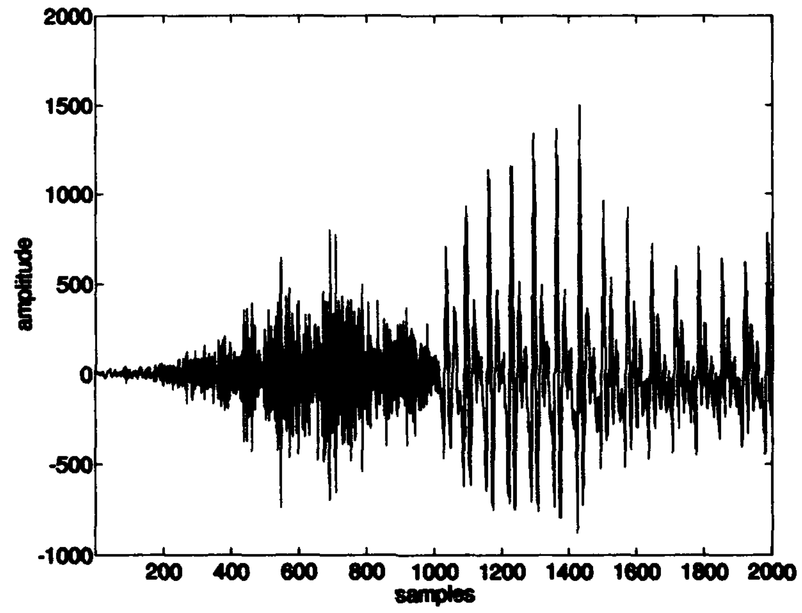


Figure A.1. Utterance of the Sentence "She had your dark suit in greasy wash water all year"

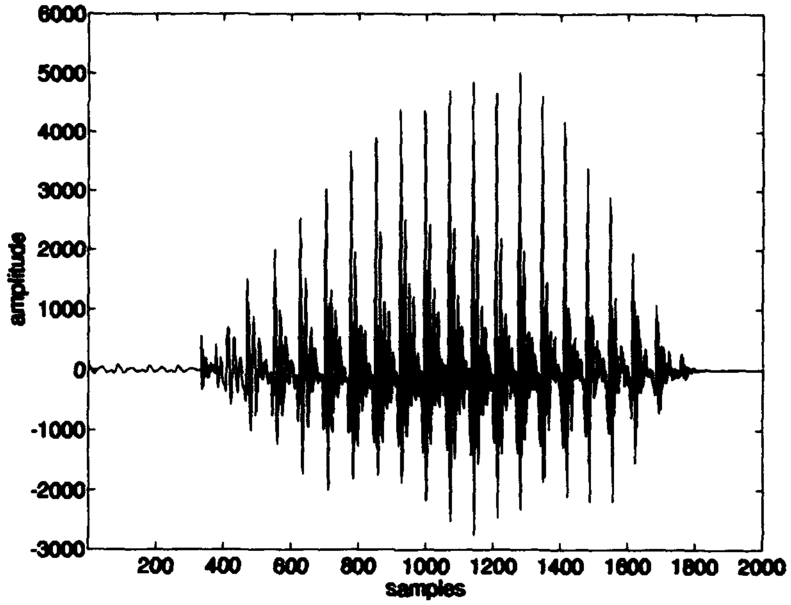


(a)

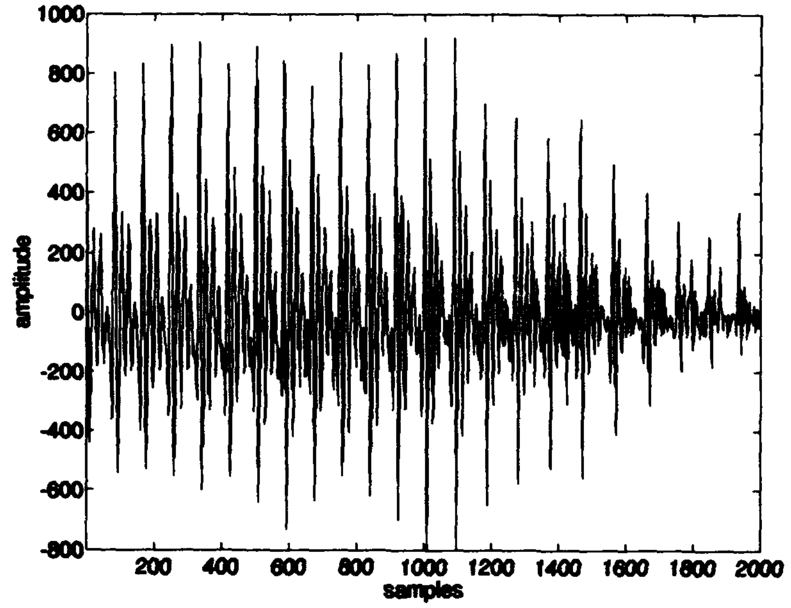


(b)

Figure A.2. Utterance of the Word (a) "Wash" and (b) "She"

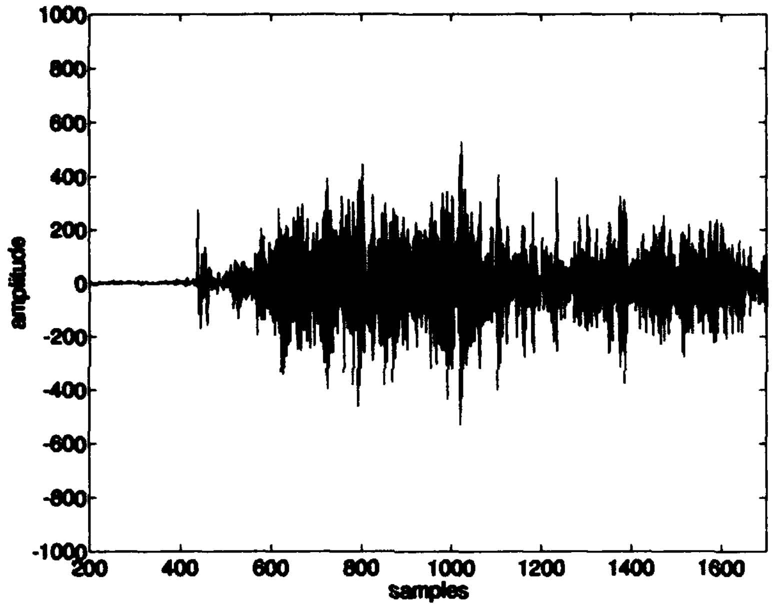


(a)

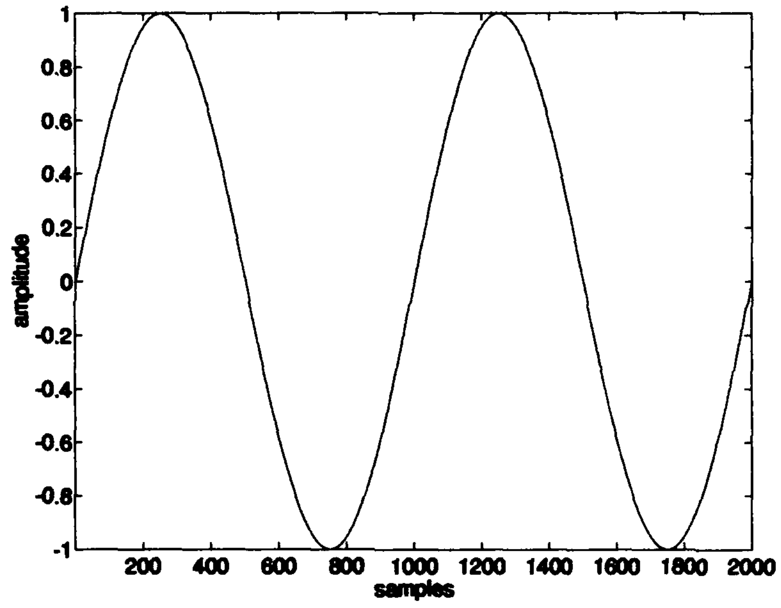


(b)

Figure A.3. Utterance of the Word (a) "Dark" and (b) "Ear"

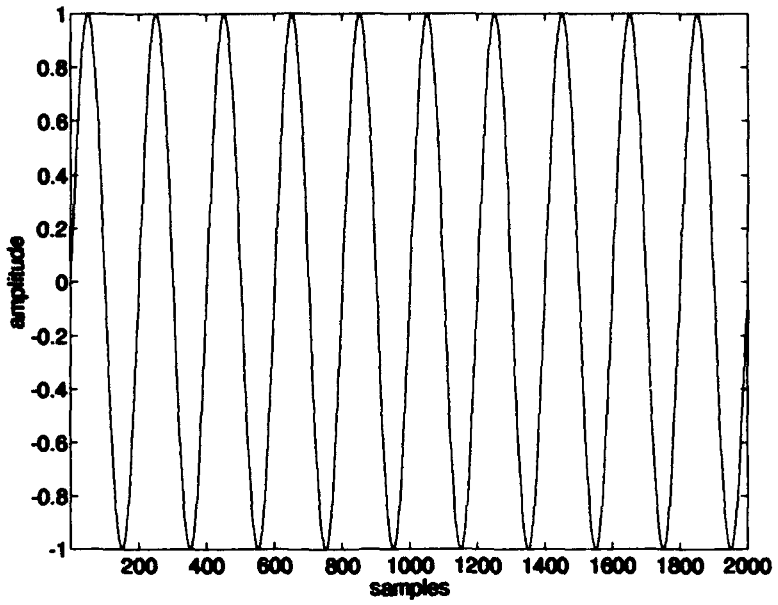


(a)

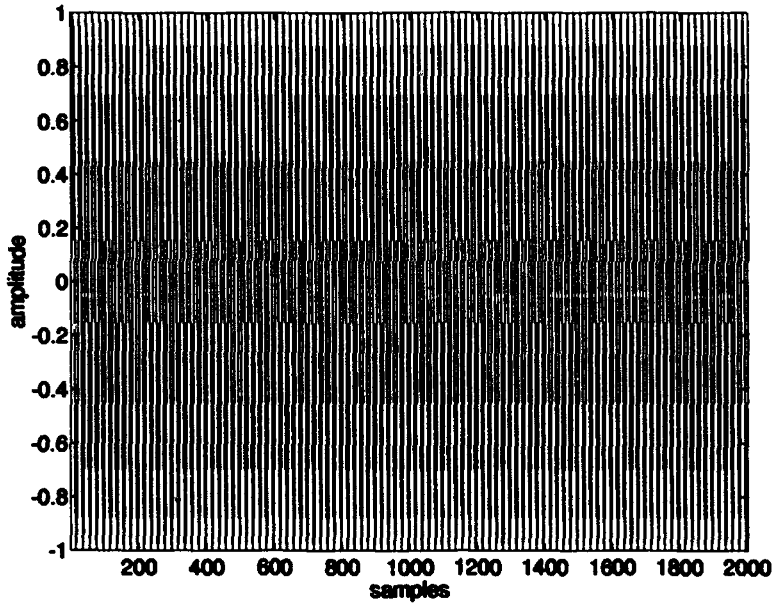


(b)

Figure A.4. (a) The Sound "S" and (b) Low Frequency Sinusoid

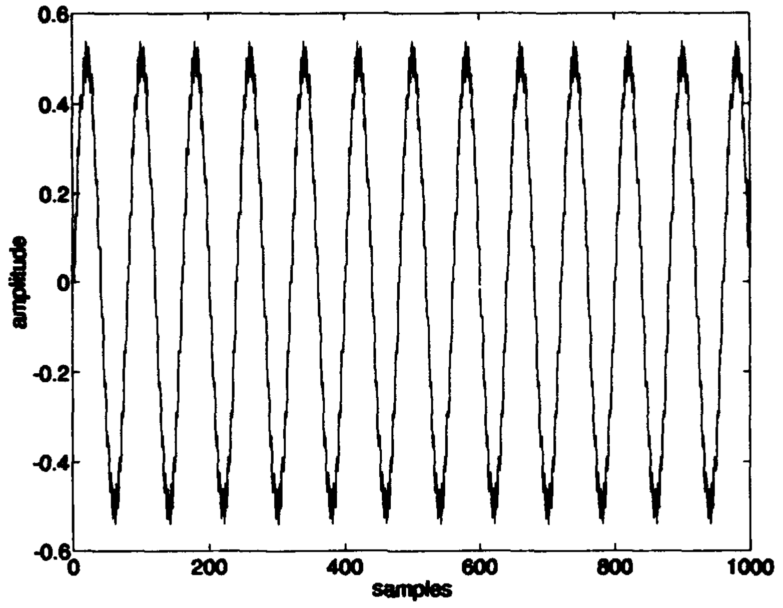


(a)

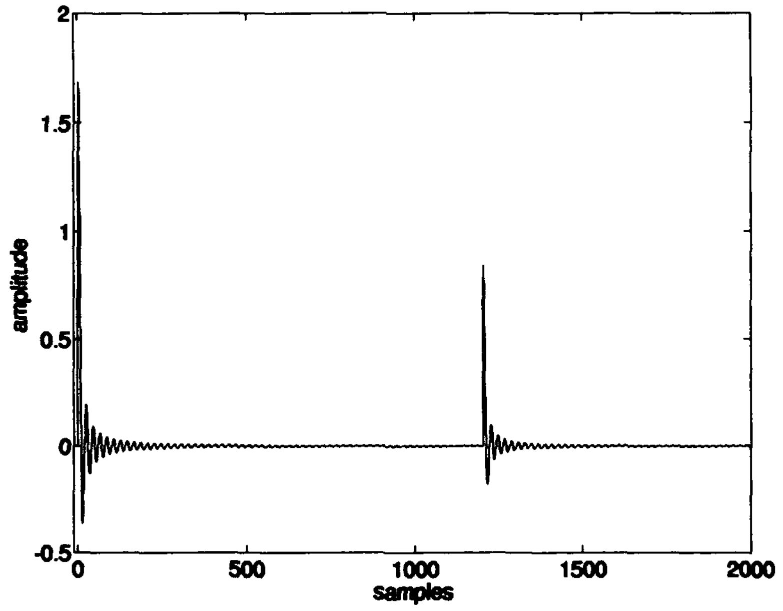


(b)

Figure A.5. (a) Low and (b) High Frequency Sinusoids



(a)



(b)

Figure A.6. (a) Convolution of Two Different Frequency Sinusoids and (b) Damped Sinusoid with an Echo

# Appendix B. *Ada Source Code For The Complex Valued Malvar Wavelet*

This appendix provides the source code used during the development and testing of the Complex Valued Malvar wavelet.

```
---------10---------20---------30---------40---------50---------60---------70------78
with Text_Io;       --         -         -         -         -         -      78*
with Complex_Pkg;
use Complex_Pkg;
with Math_Lib;
use Math_Lib;
with Vector_Package;
use Vector_Package;
with Type_Package;
use Type_Package;
with Print_Package;
use Print_Package;
with FFT_Pack;
use FFT_Pack;


   procedure cmw is

-- Performs a forward complex Malvar Wavelet
-- and an Inverse complex Malvar wavelet.
-- Inputs must be real, transform coefficients are real and imaginary.
-- Output values are real.

      package Integer_Io is new Text_Io.Integer_Io(integer);
      package Float_Io is new Text_Io.Float_Io(float);

      Max_Data_Length        : constant := 100_000;
      Max_Partitions         : constant := 500;

      Big_Daddy_Data_Vector       : Real_Vector       (1..Max_Data_Length);
      Big_Daddy_Partition_Vector  : Partition_Vector  (0..Max_Partitions);

      Actual_Data_Length          : integer := 0;
      Actual_Partitions           : integer := 0;
      Window_Size                 : integer := 0;
      Overlap_Amount              : integer := 0;

      Alpha_Name,
      Outfile_Name                : string (1..30);

      Alpha_Name_Length,
      Outfile_Name_Length         : integer range 1..30;
```

```
--------10--------20--------30--------40--------50--------60--------70------78
------------------------------------------------------------------------------
--   Get_File_Data prompts the user for the prepared info file name
--   and then reads in the data in the following order
--      (1) Input Data Filename
--      (2) Derivative Output Filename
--      (3) Alpha Output Filename
--      (4) Output Data Filename
--      (5) Input Output Difference Filename
--      (6) General Info Filename
--      (7) Number of Points in the Data Vector
--      (8) Number of required partitions
--      (9) - (?) loop for input given in (8) and get
--       Boundary point then Overlap for each partition
------------------------------------------------------------------------------
     procedure Get_File_Data (Alpha_Filename        : in out string;
                              Alpha_Length          : in out integer;
                              Output_Filename       : in out string;
                              Outname_Length        : in out integer;
                              Partitions            : in out Partition_Vector;
                              Window_Size           : in out integer;
                              Overlap_Amount        : in out integer;
                              Number_Of_Partitions  : in out integer;
                              Data                  : in out Real_Vector;
                              Points                : in out integer   ) is

           --   Filenames, lengths must be in out so they can
           --   be written to info file.
           --   Number_Of_Partitions and Points must be in out
           --   Parameters because they are used in a loop to
           --   Read in the appropriate amount of data

           Infile,
           Datafile               :  Text_Io.File_Type;
           Temp_Input             :  integer;
           Data_Filename          :  string(1..30);
           Data_Filename_Length   :  integer;
           In_Filename            :  string(1..30);
           In_Filename_Length     :  integer;
           Counter                :  integer := 0;

        begin

           Text_Io.New_Line;
           Text_Io.put("What is the name of the file containing ");
           Text_Io.put("the prepared information ? > ");
           Text_Io.Get_Line(In_Filename, In_Filename_Length);
           Text_Io.Open(Infile, Text_Io.In_File,
                                       In_Filename(1..In_Filename_Length));
```

```
          -- Now I call in the required user data
          Text_Io.Get_Line(Infile, Data_Filename, Data_Filename_Length);
          Text_Io.Get_Line(Infile, Alpha_Filename, Alpha_Length);
          Text_Io.Get_Line(Infile, Output_Filename, Outname_Length);
          Integer_Io.get(Infile,Points);
          Integer_Io.get(Infile,Window_Size);
          Integer_Io.get(Infile,Overlap_Amount);

-----------------------------------------------------------------
-----------------------------------------------------------------
          Number_Of_Partitions := Points/Window_Size;

          for j in 1..Number_Of_Partitions loop
             Counter := Counter + 1;
             Partitions(j).Boundary := Window_Size*Counter;
             Partitions(j).Overlap  := Overlap_Amount;
          end loop;
          Text_Io.Close(Infile);
          Partitions(0).Boundary := 1;
          Partitions(0).Overlap  := Overlap_Amount;


          -- now I call in the numbers for the vector Data
          Text_Io.Open(Datafile,Text_Io.In_File,
                                 Data_Filename(1..Data_Filename_Length));
          for j in 1..Points loop
             Float_Io.Get(Datafile, Data(j));  -- for floating point inputs
             --Integer_Io.Get(Datafile,Temp_Input);
             --Data(j) := float(Temp_Input);  -- for integer inputs
          end loop;
          Text_Io.Close(Datafile);

       end Get_File_Data;

--------10--------20--------30--------40--------50--------60--------70------78
-------------------------------------------------------------------------------
-- Multiply_By_Window
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
    procedure Multiply_By_Window (Data                : in Real_Vector;
                                  Folded_Data         : in out Real_Vector;
                                  Window_j            : in out Real_Vector;
                                  A_j                 : in integer;
                                  A_j_plus_1          : in integer;
                                  Epsilon             : in out integer;
                                  Window_size         : in integer;
                                  J                   : in integer) is


          Pi             : constant  :=  3.141592654;
```

```
        Window_Boundary  : constant float     := 1.0/sqrt(2.0);
        Start_File            : integer := Data'first;
        End_File              : integer := Data'last;
        Counter               : integer := 0;
        n_plus_1              : integer := window_size;


begin
  -------------*****Multiply By Window*****-------------

        -- Left edge
         if j = 0 then   --A_j = 0
           Folded_Data(A_j) := 2.0 * Data(End_File) * Window_j(0);
                  counter := 1;
           for x in (A_j+1)..(A_j + EPSILON) loop
           Folded_Data(x)  :=  Data(x) * Window_j(counter) +
               Data(End_file - x) * Window_j(-counter);
               counter := counter + 1;
           end loop;
         else
                  counter := 0;
           for x in (A_j)..(A_j + EPSILON) loop
           Folded_Data(x)  :=  Data(x) * Window_j(counter) +
                          Data(2*A_j - x) * Window_j(-counter);
                  counter := counter + 1;
           end loop;
         end if;


        -- Center
        for x in (A_j + EPSILON + 1)..(A_j_plus_1 - EPSILON - 1) loop
           Folded_Data(x) := Data(x);
        end loop;

        -- Right edge
         if A_j_plus_1 = Data'last then  --(j = partitions'last-1)
                  counter := EPSILON;
             for x in (A_j_plus_1 - EPSILON)..(A_j_plus_1 - 1) loop
                 Folded_Data(x) := Data(x) * Window_j(n_plus_1 - counter) -
                                Data(0 + counter) *
                                Window_j(n_plus_1 + counter);
                  counter := counter - 1;
             end loop;
             Folded_Data(A_j_plus_1) := 0.0;
         else
                  counter := EPSILON;
            for x in (A_j_plus_1 - EPSILON)..(A_j_plus_1) loop
                Folded_Data(x) := Data(x) * Window_j(n_plus_1 - counter) -
                      Data(2*A_j_plus_1 - x) * Window_j(n_plus_1 + counter);
                  counter := counter - 1;
            end loop;
         end if;

     end Multiply_By_Window;

--------10--------20--------30--------40--------50--------60--------70------78
------------------------------------------------------------------------------
--  Compute_Coefficients

--  This procedure takes in data from 0 - N-1  or A_j - A_j+1-1 (N points) and
--  returns coefficients stored from 0 - N-1 or A_j - (A_j+1 - 1) (N points)
------------------------------------------------------------------------------

    procedure Compute_Coefficients (Data_Segment : in Real_Vector;
                                    FFT_Out      : out Complex_Vector;
                                    N            : in integer) is

           Beta      : Real_Vector(Data_segment'range);
           Temp_FFT  : Complex_Vector(Data_Segment'range);
               -- Data_Segment'range = (Start_Data)..(End_Data - 1)
         begin

           -- (1) Scale Data_Segment Prior to taking FFT
               Beta(Beta'first)  := Data_Segment(Data_Segment'first);
           for x in 1..(N-1) loop
               Beta(Beta'first + x) :=
                              2.0*Data_Segment(Data_Segment'first + x);
           end loop;

           -- (2) Make Data_Segment Complex Valued
           Temp_FFT    :=   Complex_Of((Beta));

           -- (3) Perform Inverse FFT  (See FFT_Pack)
           FFT    (Temp_FFT, True);  -- includes 1/N scaling


           -- (4) Scale data by 1/2 (Inv FFT includes 1/N scaling, but
           --     we require 1/2N).
           for x in FFT_Out'range loop   --(Start_Data)..(End_Data-1)
               FFT_Out(x) := 0.5*Temp_FFT(x);
           end loop;


        end Compute_Coefficients;

--------10--------20--------30--------40--------50--------60--------70------78
------------------------------------------------------------------------------
 --  Reconstruction FFT
------------------------------------------------------------------------------
------------------------------------------------------------------------------

      procedure Reconstruction_FFT(FFT_In     :  in Complex_Vector;
                                  FFT_Out    :  out Complex_Vector) is

      Temp_FFT : Complex_Vector(FFT_In'range) := FFT_In;

   begin

      -- (1) Perform FFT  (See FFT_Pack)
      FFT(Temp_FFT, False);

      for x in FFT_Out'range loop
          FFT_Out(x) := Temp_FFT(x);
      end loop;
        -- Scale from Beta values to H_nl values.
          FFT_out(FFT_Out'first) := 2.0*Temp_FFT(Temp_FFT'first);

   end Reconstruction_FFT;


------------------------------------------------------------------------------
------------------------------------------------------------------------------
 --  Divide_By_Windows
------------------------------------------------------------------------------
------------------------------------------------------------------------------

   procedure Divide_By_Windows ( Temp_Data          : in Complex_Vector;
                                 Data_Segment       : out Complex_Vector;
                                 Window             : in  Real_Vector;
                                 A_j                : in integer;
                                 A_j_plus_1         : in integer;
                                 EPSILON            : in integer;
                                 Window_Size        : in integer;
                                 J                  : in integer) is

      Counter          : integer := 0;
      End_File         : integer := Temp_Data'last;
      n_plus_1         : integer := window_size;
      test             : float;

   begin


      -- Calculate Data_Segment(A_j - e_j + 1.. A_j - 1)
     if j = 0 then   --A_j = 0
          counter := EPSILON;
      for i in (A_j - EPSILON + 1)..(A_j - 1) loop
         Data_Segment(i) := Window(n_plus_1 - counter + 1) *
                            Temp_Data(End_File + i)
                          + Window(n_plus_1 + counter - 1) *
                            Temp_Data(2*A_j - i);
          counter := counter - 1;
      end loop;
```

```
         else
                counter := EPSILON;
          for i in (A_j - EPSILON + 1)..(A_j - 1) loop
             Data_Segment(i) := Window(n_plus_1 - counter + 1) *
                                Temp_Data(i) +
                                Window(n_plus_1 + counter - 1) *
                                Temp_Data(2*A_j - i);
                counter := counter - 1;
          end loop;
         end if;


          -- Calculate Data_Segment(A_j)
           test := 2.0 * window(0);
          Data_Segment(A_j) := Temp_Data(A_j)/test;

               -- Temp_Data was calculated from A_(j)-e to A_(j+1)-1, so there
               -- is no Temp_Data(End_File)

          -- Calculate Data_Segment(A_j + 1 .. A_j + e_j - 1)
          if j = 0 then  --A_j = 0
                 counter := 1;
           for i in (A_j + 1)..(A_j + EPSILON - 1) loop
               Data_Segment(i) := Window(counter) * Temp_Data(i) -
                              Window(-counter) * Temp_Data(End_File - i);
                 counter := counter + 1;
           end loop;
          else
                 counter := 1;
           for i in (A_j + 1)..(A_j + EPSILON - 1) loop
               Data_Segment(i) := Window(counter) * Temp_Data(i) -
                            Window(-counter) * Temp_Data(2*A_j - i);
                 counter := counter + 1;
           end loop;
          end if;


         -- Calculate Data_Segment(A_j + e_j .. A_j+1 - e_j)
          for i in (A_j + EPSILON)..(A_j_plus_1 - EPSILON) loop
             Data_Segment(i) := Temp_Data(i);
          end loop;

       end Divide_By_Windows;

-------------------------------------------------------------------------------

    procedure Error_Out (Recon_Data  : in Complex_Vector;
                         Actual_Data : in Real_Vector;
                         L2_Error    : in out float;
                         RMS_Error   : in out float;
                         L_INF_Error : in out float;
```

```
                                  Norm_RIV    : in out float) is

          error,
           sum1,
           sum2,
           sum3,
           count          : float := 0.0;
           Recon_RIV      : float := 0.0;
           Actual_RIV     : float := 0.0;

     begin

        for l in Actual_Data'range loop
           if abs(Actual_Data(l) - Recon_Data(l).real) > error then
              error := abs(Actual_Data(l) - Recon_Data(l).real);
           end if;
           sum1 := Actual_Data(l)*Actual_Data(l) + sum1;
           sum2 := Recon_Data(l).real*Recon_Data(l).real + sum2;
           sum3 := (Actual_Data(l)-Recon_Data(l).real)*
                     (Actual_Data(l)-Recon_Data(l).real) + sum3;
           count := count + 1.0;
           if l = 1 then
              sum1 := 0.0;
              sum2 := 0.0;
              sum3 := 0.0;
              error:= 0.0;
              count := 0.0;
           end if;
        end loop;

        RMS_Error    := sqrt(sum3)/count;
        L2_Error     := sqrt(abs(sum1-sum2)/sum1);
        L_INF_Error  := error;
        Recon_RIV    := ln(sum2)/ln(2.0);
        Actual_RIV   := ln(sum1)/ln(2.0);
        Norm_RIV     := Recon_RIV/Actual_RIV;

      end Error_Out;

--------10--------20--------30--------40--------50--------60--------70------78
------------------------------------------------------------------------------
 --  Do_Work
------------------------------------------------------------------------------
------------------------------------------------------------------------------
      procedure Do_Work   ( Data              :  in Real_Vector;
                            Partitions        :  in Partition_Vector;
                            Window_Size       :  in out integer;
                            Epsilon           :  in out integer;
                            Alphafile_String  :  in String;
                            Outfile_String    :  in String ) is
     Pi                     : constant  :=  3.141592654;
     Transformed_Data       :  Real_Vector((Data'first-1)..(Data'last));
     Temp_data              :  Complex_Vector
                                 ((Data'first-1)..Data'last);
     Window                 :  Real_Vector
                                 ((-EPSILON)..(Window_Size + EPSILON));
     FFT_Data,
     Output_Data            :  Complex_Vector ((Data'first-Partitions(0).Overlap-1)..(Data'l
     Multiply               : Real_Vector((Data'first-1)..(Data'Last-1));


     Start_Data, End_Data,         --  A_j  and  A_j+1
     Start_Window, End_Window,     --  A_j  - e_j    and A_j+1 + e_j+1
     Last_Output_Point,            --  A_j+1 - e_j+1
     Last_Window_Point,            --  A_j   + e_j
     Counter, J_Count, Temp        : integer  :=  0;

     Alphafile,
     Outfile                : Text_Io.File_Type;
     L2_Error               : float := 0.0;
     L_INF_Error            : float := 0.0;
     RMS_Error              : float := 0.0;
     Norm_RIV               : float := 0.0;
     Percent_Compression    : float    := 0.0;
     Threshold              : float    := 0.0;
     Percent_Compressed     : float    := 0.0;
     Compression_Count      : integer := 0;
     No_Zeros               : integer := 0;
     No_Zeros_R             : float    := 0.0;
     Max                    : float    := 0.0;
     Adjust                 : float    := 1.0;
     TooHigh                : integer := 0;
     TooLow                 : integer := 0;
     Start_File             : integer := Data'first;
     End_File               : integer := Data'last;
     n                      : constant integer := 0;
     n_plus_1               : constant integer := Window_Size;

begin
     Text_Io.Create (Alphafile, Text_Io.Out_File, Alphafile_String);
     Text_Io.Create (Outfile,   Text_Io.Out_File, Outfile_String  );

     Text_Io.New_Line;
     Text_Io.put("What is the desired percent compression ? ");
     Text_Io.New_line;
     Text_Io.put("(ie 0.85 for 85% compression)");
     Text_Io.New_line;
     Float_Io.get(Percent_Compression);
     Text_Io.New_Line;
```

```
--------*****CREATE WINDOW*****--------

   -- left rising edge
   if EPSILON = 0 then
      Window(n) := 1.0/(sqrt(2.0));
   else
      for x in (n - EPSILON)..(n + EPSILON - 1) loop
         Window(x) := sin((Pi/(4.0*float(EPSILON))) *
                                        (float(x - n + EPSILON)));
      end loop;
   end if;

   -- center
   for x in (n + EPSILON)..(n_plus_1 - EPSILON) loop
      Window(x) := 1.0;
   end loop;

   -- right falling edge
   if EPSILON = 0 then
      Window(n_plus_1) := 1.0/(sqrt(2.0));
   else
      for x in (n_plus_1 - EPSILON + 1)..(n_plus_1 + EPSILON - 1)
               loop
         Window(x) := cos((Pi/(4.0*float(EPSILON))) *
                              (float(x - n_plus_1 + EPSILON)));
      end loop;
      Window(n_plus_1 + EPSILON) := 0.0;
   end if;


      J_Count := -1;
for j in 0..(Partitions'last-1) loop
      J_Count := J_Count + 1;

   begin -- each segment is transformed within this loop
         if j_count = 0 then
            TEMP := 1;
         else
            TEMP := 0;
         end if;
-----------------------------Assign j Dependent Values------------------------

     Start_Data   := Partitions(j).Boundary - Temp;
     End_Data     := Partitions(j+1).Boundary;
     Start_Window := Start_Data -  EPSILON;
     End_Window   := End_data + EPSILON;
---------10--------20--------30--------40--------50--------60--------70------78

           Text_Io.put("Working.... Processing Data Segment");
           Integer_Io.put(j+1);
           Text_Io.New_Line;
```

```
          Multiply_By_Window(Data(Start_File..End_File),
                             Transformed_Data((Start_Data)..End_Data),
                             Window(window'range), Start_data,
                             End_Data, EPSILON, Window_Size, J_Count);


         Compute_Coefficients(Transformed_Data((Start_Data)..(End_Data-1)),
                              FFT_Data((Start_Data)..(End_Data-1)),
                              Window_Size);
                 -- This routine performs
                 -- (1) scale the data,
                 -- (2) an inverse FFT,
                 -- (3) assigns coefficients 0 - N-1 to Transformed_Data

---------------------Print Alpha Files-----------------------------------
          for x in (Data'first-1)..(Data'last-1) loop
             -- Print output values to the alpha file (coefficients).
               Float_Io.Put (Alphafile, FFT_Data(x).real);
               Text_Io.New_Line(Alphafile);
          end loop;

--------10--------20--------30--------40--------50--------60--------70------78
---------------------- RECONSTRUCTION -----------------------------------


          Reconstruction_FFT (FFT_Data(Start_Data..(End_Data-1)),
                              Temp_Data(Start_Data..(End_Data-1)));
                 -- This routine performs a forward FFT


     end; -- --  block
  end loop;  -- -- j loop


              J_Count := -1;
     for j in 0..(Partitions'last-1) loop
              J_Count := J_Count + 1;

         begin -- each segment is transformed within this loop
                if j_count = 0 then
                   TEMP := 1;
                else
                   TEMP := 0;
                end if;
-----------------------------Assign j Dependent Values-----------------------

        Start_Data   := Partitions(j).Boundary - Temp;
        End_Data     := Partitions(j+1).Boundary;
        Start_Window := Start_Data -  EPSILON;
        End_Window   := End_Data + EPSILON;
        Last_Output_Point  :=  End_Data - EPSILON;
        Last_Window_Point  :=  Start_Data + EPSILON;
--------10--------20--------30--------40--------50--------60--------70------78

          Divide_By_Windows(Temp_Data(Temp_Data'range),
                            Output_Data(Start_Window..Last_Output_Point),
                            Window(window'range),
                            Start_Data, End_Data, EPSILON,
                            Window_Size, J_Count);

                 if j_count = 0 then
                    for k in 0..(EPSILON - 1) loop
                      Output_Data(End_file - k) := Output_Data(-k);
                    end loop;
                 end if;


        end; -- --  block
    end loop;  -- -- j loop


------------------------CALCULATE OUTPUT ERRORS ------------------------

           Error_Out ( Output_Data(Data'range),
                       Data(Data'range),
                       L2_Error, RMS_Error, L_INF_Error, Norm_RIV);


------------------------ PRINT OUT TO FILES ----------------------------
           Text_Io.Close(Alphafile);

           Text_io.Put_Line("print output to file");
           Text_Io.New_Line;
           Text_Io.put("L2 Error =  ");
           Float_Io.put(L2_Error);
           Text_Io.New_Line;
           Text_Io.put("L(Infinity) Error =  ");
           Float_Io.put(L_INF_Error);
           Text_Io.New_line;
           Text_Io.put("RMS Error =  ");
           Float_Io.put(RMS_Error);
           Text_Io.New_Line;
           Text_Io.put("Normalized RIV =  ");
           Float_Io.put(Norm_RIV);
           Text_Io.New_Line;

           for x in Data'range loop
              Float_Io.put(Outfile,Output_Data(x).real);
             -- Real signals in --> Real signal out.
             ---- Float_Io.put(Outfile,Output_Data(x).imag);
              Text_Io.New_Line(Outfile);
          end loop;


          Text_Io.Close(Outfile);

       end Do_Work;
---------------------------------------------------------------------------
---------------------------------------------------------------------------

begin --main

       Get_File_Data   (Alpha_Name, Alpha_Name_Length,
                        Outfile_Name, Outfile_Name_Length,
                        Big_Daddy_Partition_Vector, Window_Size,
                        Overlap_Amount, Actual_Partitions,
                        Big_Daddy_Data_Vector, Actual_Data_Length);

       -- By sending the correct size arrays into Do_Work, the array
       -- attributes can be used to determine the size rather than
       -- passing another parameter
    Do_Work  (Big_Daddy_Data_Vector(1..Actual_Data_Length),
              Big_Daddy_Partition_Vector (0..Actual_Partitions),
              Window_Size, Overlap_Amount,
              Alpha_Name(1..Alpha_Name_Length),
              Outfile_Name(1..Outfile_Name_Length));
end; --main
```

# Appendix C. *Ada Source Code For The Real Valued Malvar Wavelet*

This appendix provides the source code used during the development and testing of the Real Valued Malvar wavelet.

```
--------10--------20--------30--------40--------50--------60--------70------78
------------------------------------------------------------------------------
--Notes : --------------------------------------------------------------------------------
with Text_Io;      --       -         -         -         -         -     78*
with Complex_Pkg;
use Complex_Pkg;
with Math_Lib;
use Math_Lib;
with Vector_Package;
use Vector_Package;
with Type_Package;
use Type_Package;
with Print_Package;
use Print_Package;
with FFT_Pack;
use FFT_Pack;


   procedure glot is
-- Performs the basic GLOT and Inverse GLOT
-- Performs the overlapping without copying the end point data.
-- The program "wraps" the data around, so that the first data
-- point connects with the last data point.

      package Integer_Io is new Text_Io.Integer_Io(integer);
      package Float_Io is new Text_Io.Float_Io(float);

      Max_Data_Length              : constant := 100_000;
      Max_Partitions               : constant := 500;

      Big_Daddy_Data_Vector        : Real_Vector     (1..Max_Data_Length);
      Big_Daddy_Partition_Vector   : Partition_Vector(0..Max_Partitions);

      Actual_Data_Length           : integer := 0;
      Actual_Partitions            : integer := 0;

      Alpha_Name,
      Outfile_Name                 : string (1..30);

      Alpha_Name_Length,
      Outfile_Name_Length          : integer range 1..30;
```

```
----------------------------------------------------------------------------
--------10--------20--------30--------40--------50--------60--------70------78
----------------------------------------------------------------------------
--   Get_File_Data prompts the user for the prepared info file name
--   and then reads in the data in the following order
--      (1) Input Data Filename
--      (2) Alpha Output Filename
--      (3) Output Data Filename
--      (4) Number of Points in the Data Vector
--      (5) Number of required partitions
--      (6) - (?) loop for input given in (5) and get
--      Boundary point then Overlap for each partition
----------------------------------------------------------------------------

      procedure Get_File_Data (Alpha_Filename       : in out string;
                               Alpha_Length         : in out integer;
                               Output_Filename      : in out string;
                               Outname_Length       : in out integer;
                               Partitions           : in out Partition_Vector;
                               Number_Of_Partitions : in out integer;
                               Data                 : in out Real_Vector;
                               Points               : in out integer   ) is

            --   Filename , lengths must be "in out" so they can
            --   be written to info file (if want to use an
            --   information file.
            --   Number_Of_Partitions and Points must be "in out"
            --   parameters because they are used in a loop to
            --   read in the appropriate amount of data.

            Infile,
            Datafile              :  Text_Io.File_Type;
            Temp_Input            :  integer;
            Data_Filename         :  string(1..30);
            Data_Filename_Length  :  integer;
            In_Filename           :  string(1..30);
            In_Filename_Length    :  integer;
            Counter               :  integer := 0;

         begin

            Text_Io.New_Line;
            Text_Io.put("What is the name of the file containing ");
            Text_Io.put("the prepared information ? > ");
            Text_Io.Get_Line(In_Filename, In_Filename_Length);
            Text_Io.Open(Infile, Text_Io.In_File,
                         In_Filename(1..In_Filename_Length));

            -- Now I call in the required user data
            Text_Io.Get_Line(Infile, Data_Filename, Data_Filename_Length);
            Text_Io.Get_Line(Infile, Alpha_Filename, Alpha_Length);
         Text_Io.Get_Line(Infile, Output_Filename, Outname_Length);
         Integer_Io.get(Infile,Points);
         Integer_Io.get(Infile,Number_Of_Partitions);

------------------ Read In Data From Data File --------------------
         -- Set up all of the Partitions and the amount of
         -- overlap for each window.
         for j in 1..Number_Of_Partitions loop
            Integer_Io.get(Infile,Partitions(j).Boundary);
            Integer_Io.get(Infile,Partitions(j).Overlap);
         end loop;
         Text_Io.Close(Infile);

         -- A(0) position initially set to 1, later set to 0
         -- so that there are 0 -> N data positions.
         Partitions(0).Boundary := 1;
         Partitions(0).Overlap  := Partitions(Number_of_Partitions).Overlap;

         -- Call in the data for the vector Data
         Text_Io.Open(Datafile,Text_Io.In_File,
                      Data_Filename(1..Data_Filename_Length));
         for j in 1..Points loop
            Float_Io.Get(Datafile, Data(j));
            --Integer_Io.Get(Datafile,Temp_Input);
            --Data(j) := float(Temp_Input);       -- for integer inputs
         end loop;
         Text_Io.Close(Datafile);

      end Get_File_Data;

--------10--------20--------30--------40--------50--------60--------70------78
------------------------------------------------------------------------------
-- Multiply_By_Window
------------------------------------------------------------------------------
------------------------------------------------------------------------------
     procedure Multiply_By_Window (Data              : in Real_Vector;
                                   Folded_Data       : in out Real_Vector;
                                   Window_j          : in out Real_Vector;
                                   A_j                  : in integer;
                                   A_j_plus_1           : in integer;
                                   Epsilon_j            : in out integer;
                                   Epsilon_j_plus_1     : in out integer;
                                   J                    : in integer) is


         Pi                    : constant         :=  3.141592654;
         Start_File            : integer          := Data'first;
         End_File              : integer          := Data'last;
         Counter               : integer          := 0;
         Window_Boundary_Value : constant float   := 1.0/sqrt(2.0);
```

```
        begin

----------------------*****CREATE WINDOW*****--------------------------------
           -- Window Defined Page 11


           -- left rising edge
           if Epsilon_j = 0 then
              Window_j(A_j) := Window_Boundary_Value;
           else
             for x in (A_j - EPSILON_j)..(A_j + EPSILON_j) loop
                 Window_j(x) := sin((Pi/(4.0*float(EPSILON_j))) *
                                         (float(x-(A_j-EPSILON_j))));
              end loop;
           end if;


           -- center
           for x in (A_j + EPSILON_j + 1)..(A_j_plus_1 - EPSILON_j_plus_1 - 1)
           loop
              Window_j(x) := 1.0;
           end loop;


           -- right falling edge
           if EPSILON_j_plus_1 = 0 then
              Window_j(A_j_plus_1) := Window_Boundary_Value;
           else
              for x in (A_j_plus_1 - EPSILON_j_plus_1)..
                                 (A_j_plus_1 + EPSILON_j_plus_1)  loop
                 Window_j(x) := cos((Pi/(4.0*float(EPSILON_j_plus_1))) *
                                  (float(x-(A_j_plus_1 -EPSILON_j_plus_1))));
              end loop;

           end if;
----------------------*****Multiply By Window*****--------------------------

           -- Left Side
           if j = 0 then
              Folded_Data(A_j) := 0.0;
           for x in (A_j + 1)..(A_j + EPSILON_j) loop
              Folded_Data(x)  :=  Data(x) * Window_j(x) -
                                  Data(End_File - x) * Window_j(2*A_j - x);
           end loop;
           else
           for x in (A_j)..(A_j + EPSILON_j) loop
              Folded_Data(x)  :=  Data(x) * Window_j(x) -
                                  Data(2*A_j - x) * Window_j(2*A_j - x);
           end loop;
           end if;
```

```
            -- Center
            for x in (A_j + EPSILON_j + 1)..(A_j_plus_1 - EPSILON_j_plus_1 - 1)
            loop
              Folded_Data(x) := Data(x) * Window_j(x);
            end loop;

            -- Right Edge
             if A_j_plus_1 = Data'last then  --(j = partitions'last-1)
                for x in (A_j_plus_1 - Epsilon_j_plus_1)..(A_j_plus_1 - 1) loop
                    Folded_Data(x) := Data(x) * Window_j(x) +
                                      Data(A_j_plus_1 - x) *
                                      Window_j(2*A_j_plus_1 - x);
                end loop;
                Folded_Data(A_j_plus_1) := 2.0*Data(A_j_plus_1) *
                                          Window_j(A_j_plus_1);
             else
             for x in (A_j_plus_1 - EPSILON_j_plus_1)..(A_j_plus_1) loop
                 Folded_Data(x) := Data(x) * Window_j(x) +
                           Data(2*A_j_plus_1 - x) * Window_j(2*A_j_plus_1 - x);
             end loop;
             end if;


    end Multiply_By_Window;

--------10--------20--------30--------40--------50--------60--------70------78
------------------------------------------------------------------------------
--  Compute_Coefficients

--  This procedure takes in data from 0 - N  or A_j - A_j+1 (N+1 points) and
--  returns coefficients stored from 0 - N-1 or A_j - (A_j+1 - 1) (N points)
--  Nth point is used in calculations but is not a valid out point
--  It will be used to store the zero'th coefficient in the next data segment
------------------------------------------------------------------------------
------------------------------------------------------------------------------

      procedure Compute_Coefficients (Data_Segment : in out Real_Vector) is

                   -- Data_Segment is from A_j .. A_j_plus_1
          Two_N        : integer := 2*(Data_Segment'last - Data_Segment'first);
          End_FFT_Data : integer := Data_Segment'first + Two_N - 1;
          FFT_Data     : Complex_Vector(Data_Segment'first..End_FFT_Data);


         begin

            -- (1) Even Extend Data_Segment(See Vector_Package-Complex_Package)
            FFT_Data    :=   Complex_Of(Even_Extend(Data_Segment));

            -- (2) Perform Inverse FFT  (See FFT_Pack)
      FFT    (FFT_Data, True);  -- includes 1/N scaling

      -- (3.a) Assign zero coefficient to first point in Data_Segment
      Data_Segment(Data_Segment'first) := FFT_Data(FFT_Data'first).Real
                                  *  sqrt(float(Two_N));

      -- (3.b) Assign coefficients 1 to N-1 back to Data_Segment
      for k in (Data_Segment'first + 1)..(Data_Segment'last-1) loop
        Data_Segment(k) := Data_Segment(k-1) +
                          (2.0 * sqrt(float(Two_N)) * FFT_Data(k).Real);
      end loop;


    end Compute_Coefficients;

--------10--------20--------30--------40--------50--------60--------70------78
------------------------------------------------------------------------------
 --  Reconstruction FFT
------------------------------------------------------------------------------
------------------------------------------------------------------------------

    procedure Reconstruction_FFT(Input_Data   :  in Real_Vector;
                                 Output_Data  :  out Real_Vector ) is

      Pi             : constant   :=  3.141592654;
      X              : float      := 0.0;  -- float counter
      Scale_Factor   : float;
      Complex_Factor : Complex;
      Two_N          : integer    := 2 * Input_Data'length;
      Two_N_Float    : float      := float(Two_N);
      End_FFT_Data   : integer    := Input_Data'first + Two_N - 1;
      FFT_Data       : Complex_Vector(Input_Data'first..End_FFT_Data);


    begin

      -- (1) Odd Extend Input_Data (See Vector_Package, Complex_Package)
      FFT_Data := Complex_Of(Odd_Extend(Input_Data));


      -- (2) Perform Inverse FFT  (See FFT_Pack)
      FFT(FFT_Data, True);

      X := 0.0;
      Scale_Factor := sqrt(Two_N_Float);

   -- Range for the scaling must be from 0 to N-1
     for l in FFT_Data'range loop
        Complex_Factor.real := Cos(Pi*X/Two_N_Float);
```

```
         Complex_Factor.imag := Sin(Pi*X/Two_N_Float);
         FFT_Data(l) := Complex_Factor * FFT_Data(l);
         X := X + 1.0;  -- float counter
      end loop;

      -- scale and assign imaginary part to Output_Data 1..N
      for l in Output_Data'range loop
         Output_Data(l) := Scale_Factor * FFT_Data(l).imag;
      end loop;

   end Reconstruction_FFT;


-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
 --  Divide_By_Windows
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------

   procedure Divide_By_Windows ( Temp_Data          : in Real_Vector;
                                 Data_Segment       : out Real_Vector;
                                 Window_Right       : in  Real_Vector;
                                 A_j                : in integer;
                                 A_j_plus_1         : in integer;
                                 Epsilon_j          : in integer;
                                 Epsilon_j_plus_1   : in integer;
                                 J                  : in integer) is

      Counter        : integer := 0;
      End_File       : integer := Temp_Data'last;
      Window_Left    : Real_Vector(Window_Right'range);

   begin

      Window_Left := Reverse_Assignment(Window_Right);
      -- Window_left is now the opposite slope of Window_Right
      -- and can be used for the j-1 windows and Window_Right
      -- will be used for the jth windows.


      -- Calculate Data_Segment (A_j - e_j + 1.. A_j - 1)
      if j = 0 then
      for i in (A_j - Epsilon_j + 1)..(A_j - 1) loop
         Data_Segment(i) := Window_Left(i) * Temp_Data(End_File + i) -
                            Window_Left(2*A_j - i) *
                            Temp_Data(2*A_j - i);
     end loop;
     else
      for i in (A_j - Epsilon_j + 1)..(A_j - 1) loop
         Data_Segment(i) := Window_Left(i) * Temp_Data(i) -
                            Window_Left(2*A_j - i) *
```

```
                                 Temp_Data(2*A_j - i);
      end loop;
      end if;


       -- Calculate Data_Segment (A_j)
       if j = 0 then
       Data_Segment(A_j) := Temp_Data(End_File) / (2.0 * Window_Left(A_j));
       else
       Data_Segment(A_j) := Temp_Data(A_j) / (2.0 * Window_Left(A_j));
       end if;



       -- Calculate Data_Segment (A_j + 1 .. A_j + e_j - 1)
       if j = 0 then
       for i in (A_j + 1)..(A_j + Epsilon_j - 1) loop
          Data_Segment(i) := Window_Right(2*A_j - i) *
                             Temp_Data(End_File - i) +
                             Window_Right(i) * Temp_Data(i);
       end loop;
       else
       for i in (A_j + 1)..(A_j + Epsilon_j - 1) loop
          Data_Segment(i) := Window_Right(2*A_j - i) *
                             Temp_Data(2*A_j - i) +
                             Window_Right(i) * Temp_Data(i);
       end loop;
       end if;


       -- Calculate Data_Segment(A_j + e_j .. A_j-1 - e_j-1)
         for i in (A_j + Epsilon_j)..(A_j_plus_1 - Epsilon_j_plus_1) loop
            Data_Segment(i) := Temp_Data(i);
         end loop;


     end Divide_By_Windows;
--------------------------------------------------------------------------------

  procedure Error_Out (Recon_Data  : in Real_Vector;
                       Actual_Data : in Real_Vector;
                       L2_Error    : in out float;
                       RMS_Error   : in out float;
                       L_INF_Error : in out float;
                       Norm_RIV    : in out float) is

     error,
      sum1,
      sum2,
      sum3,
      count         : float := 0.0;
```

```
        Recon_RIV      : float := 0.0;
        Actual_RIV     : float := 0.0;

  begin

     for l in Actual_Data'range loop
         if abs(Actual_Data(l) - Recon_Data(l)) > error then
            error := abs(Actual_Data(l) - Recon_Data(l));
         end if;
         sum1 := Actual_Data(l)*Actual_Data(l) + sum1;
         sum2 := Recon_Data(l)*Recon_Data(l) + sum2;
         sum3 := (Actual_Data(l)-Recon_Data(l))*
                   (Actual_Data(l)-Recon_Data(l)) + sum3;
         count := count + 1.0;
         if l = 1 then
            sum1 := 0.0;
            sum2 := 0.0;
            sum3 := 0.0;
            error:= 0.0;
            count := 0.0;
         end if;
     end loop;

     RMS_Error   := sqrt(sum3)/count;
     L2_Error    := sqrt(abs(sum1-sum2)/sum1);
     L_INF_Error := error;
     Recon_RIV   := ln(sum2)/ln(2.0);
     Actual_RIV  := ln(sum1)/ln(2.0);
     Norm_RIV    := Recon_RIV/Actual_RIV - 1.00000000;

   end Error_Out;


--------10--------20--------30--------40--------50--------60--------70------78
------------------------------------------------------------------------------
 --  Do_Work
------------------------------------------------------------------------------
-       procedure Do_Work   ( Data              :  in out Real_Vector;
                              Partitions        :  in out Partition_Vector;
                              Alphafile_String  :  in String;
                              Outfile_String    :  in String) is

    -- Define Data Vectors and Their Ranges.
       FFT_Data              :  Real_Vector(Data'range);
       Transformed_Data ,
       Temp_Data             :  Real_Vector((Data'first-1)..(Data'last));
       Output_Data           :  Real_Vector
       ((Data'first-Partitions(0).Overlap-1)..(Data'last+Partitions(0).Overlap));
       Window                :  Real_Vector
       ((0-Partitions(0).Overlap)..(Data'last + Partitions(0).Overlap));
```

```
          Start_Data, End_Data,         --  A_j  and  A_j+1
          Start_Window, End_Window,     --  A_j  - e_j    and  A_j+1 + e_j+1
          Last_Output_Point,            --  A_j+1 - e_j+1
          Last_Window_Point,            --  A_j   + e_j
          Counter                         : integer  :=  0;
          Two_N                           : float    := 0.0;
          Pi                              : constant  :=  3.141592654;
          Temp,
          J_Count                         : integer  := 0;
          End_File                        : integer := Data'last;
          Start_File                      : integer := Data'first;
          Alphafile,
          Outfile                         : Text_Io.File_Type;
          L2_Error                : float := 0.0;
          L_INF_Error             : float := 0.0;
          RMS_Error               : float := 0.0;
          Norm_RIV                : float := 0.0;

       begin
          Text_Io.Create (Alphafile, Text_Io.Out_File, Alphafile_String);
          Text_Io.Create (Outfile,   Text_Io.Out_File, Outfile_String  );


--------10--------20--------30--------40--------50--------60--------70------78
            J_Count := -1;
     for j in 0..(Partitions'last-1) loop
            J_Count := J_Count + 1;

           begin -- each segment is transformed within this loop

           if J_Count = 0 then
              Temp := 1; -- To set A(0) = 0.
           else
             Temp := 0;
           end if;

----------------------------Assign j Dependent Values----------------------
       Start_Data   := Partitions(j).Boundary - Temp;
       End_Data     := Partitions(j+1).Boundary;
       Start_Window := Start_Data - Partitions(j).Overlap;
       End_Window   := End_Data + Partitions(j+1).Overlap;
       Two_N        := float(2 * (End_Data-Start_Data));
---------------------------------------------------------------------------

              Text_Io.put("Working.... Processing Data Segment");
              Integer_Io.put(j+1);
              Text_Io.New_Line;

         Multiply_By_Window(Data(Data'range),
                          Transformed_Data(Start_Data..End_Data),
                          Window(Start_Window..End_Window), Start_data,
                             End_Data, Partitions(j).Overlap,
                             Partitions(j+1).Overlap,  J_Count);
                    -- Steps 2 and 3 of Coefficient Evaluation (page 14)
                    -- Window added as passed parameter only so
                    -- it does not have to be recalculated
                    -- in Divide_By_Windows

                    -- Step 4 of Coefficient Evaluation (page 15)
                 Counter := 0;
              for l in Start_Data..End_Data loop
                    Transformed_Data(l) := Transformed_Data(l) *
                                  sin((Pi * float(Counter)) / (Two_N));
                    Counter := Counter + 1;  -- Counter goes from 0 to N
              end loop;



          Compute_Coefficients (Transformed_Data(Start_Data..End_Data));
                 -- This routine performs
                 -- (1) an even extension of the data,
                 -- (2) an inverse FFT,
                 -- (3) assigns coefficients 0 - N-1 to Transformed_Data
                 -- Steps 5, 6, and 7 of Coefficient Evaluation (page 16)

----------------------Print Alphas for each segment to file-----------------
                 for x in Start_Data..(End_Data-1) loop
                    float_io.put(Alphafile,Transformed_Data(x));
                    text_io.new_line(Alphafile);
                 end loop;
---------------------------------------------------------------------------

        Reconstruction_FFT(Transformed_Data((Start_Data)..(End_Data-1)),
                           FFT_Data((Start_Data+1)..End_Data));
                 -- This routine performs
                 -- (1) an odd extension of the data,
                 -- (2) an inverse FFT,
                 -- (3) assigns the scaled imaginary result to
                 --     the Output_Data values 1 - N or A_j + 1 to A_j+1
                 --  This indicates that data values 0 - N
                 --  create alphas from 0 - N-1 which in turn are used
                 --  to reconstruct data values from 1 - N
                 --  Steps 1 and 2 of Reconstruction (page 15)


      end; -- --  block
    end loop;  -- -- j loop
--------10--------20--------30--------40--------50--------60--------70------78
              J_Count := -1;
      for j in 0..(Partitions'last-1) loop
              J_Count := J_Count + 1;
        begin -- each segment is transformed within this loop

           if J_Count = 0 then
              Temp := 1;
           else
              Temp := 0;
           end if;
---------------------------Assign j Dependent Values----------------------
      Start_Data    := Partitions(j).Boundary - Temp;
      End_Data      := Partitions(j+1).Boundary;
      Start_Window  := Start_Data -  Partitions(j).Overlap;
      End_Window    := End_Data + Partitions(j+1).Overlap;
      Two_N              := float(2 * (End_Data-Start_Data));
      Last_Output_Point  :=  End_Data - Partitions(j+1).Overlap;
      Last_Window_Point  :=  Start_Data + Partitions(j).Overlap;
--------------------------------------------------------------------------

      Divide_By_Windows(FFT_Data(FFT_Data'range),
                        Output_Data((Start_Window+1)..Last_Output_Point),
                        Window(Start_Window..Last_Window_Point),
                        Start_Data, End_Data,Partitions(j).Overlap,
                        Partitions(j+1).Overlap, J_Count);

           -- Move the (0..(-Epsilon(0)+1)) Data Back To The
           -- (End_File..(End_File-Epsilon(0)+1)) Range.
           if j_count = 0 then
              for k in 0..(Partitions(0).Overlap-1) loop
                  Output_Data(End_File - k) := Output_Data(-k);
              end loop;
           end if;


      end; -- --  block
   end loop;  -- -- j loop

--------------------------------------------------------------------------

----------------------CALCULATE OUTPUT ERRORS ----------------------

        Error_Out ( Output_Data(Data'range),
                    Data(Data'range),
                    L2_Error, RMS_Error, L_INF_Error, Norm_RIV);

---------------------- PRINT OUT TO FILES -------------------------

        Text_io.Put_Line("print output to file");
        Text_Io.New_Line;
        Text_Io.put("L2 Error =  ");
        Float_Io.put(L2_Error);
        Text_Io.New_Line;
```

```
          Text_Io.put("L(Infinity) Error =  ");
          Float_Io.put(L_INF_Error);
          Text_Io.New_line;
          Text_Io.put("RMS Error =  ");
          Float_Io.put(RMS_Error);
          Text_Io.New_Line;
          Text_Io.put("Normalized RIV =  ");
          Float_Io.put(Norm_RIV);
          Text_Io.New_Line;

          for x in Data'range loop
             Float_Io.put(Outfile,((Output_Data(x))));
             Text_Io.New_Line(Outfile);
              Text_Io.New_Line(Outfile);
          end loop;

          Text_Io.Close(Alphafile);
          Text_Io.Close(Outfile);


       end Do_Work;

-------------------------------------------------------------
-------------------------------------------------------------
-------------------------------------------------------------

   begin --main

       Get_File_Data   (Alpha_Name, Alpha_Name_Length,
                         Outfile_Name, Outfile_Name_Length,
                         Big_Daddy_Partition_Vector, Actual_Partitions,
                         Big_Daddy_Data_Vector, Actual_Data_Length);

       -- By sending the correct size arrays into Do_Work, the array
       -- attributes can be used to determine the size rather than
       -- passing another parameter

     Do_Work  (Big_Daddy_Data_Vector(1..Actual_Data_Length),
               Big_Daddy_Partition_Vector (0..Actual_Partitions),
               Alpha_Name(1..Alpha_Name_Length),
               Outfile_Name(1..Outfile_Name_Length));


   end; --main
```

# Appendix D. *Ada Source Code For Linear Transforms*

The following code was written to perform the linear transforms for the homomorphic and non-homomorphic systems studied in Chapter IV.

```
-------------------- Linear Transform -------------------------
       procedure Linear_Transform( Input_Data  : in Real_Vector;
                                   Output_Data : out Real_Vector;
                                   Inverse     : in out Integer;
                                   Transform   : in out Integer;
                                   Min         : in out Real_Vector;
                                   Sign        : in out Real_Vector;
                                   J           : in Integer ) is
                  x : integer;
               temp : float;
               mult : float   := 10.0000;
               add  : float   :=  1.0000;
    begin

      if Inverse = 0 then
             --Find minimum value for each window
            for l in Input_Data'range loop
              if Input_Data(l) < min(J) then
                 Min(j) := Input_Data(l);
              end if;
            end loop;

         if Transform = 1 then
            for l in Input_Data'range loop
              Output_Data(l) := Input_Data(l);
            end loop;
         end if;

       if Transform = 2 then
          for l in Input_Data'range loop
           if Input_Data(l) < 0.0 then
                sign(l) := -1.0;
              else
                sign(l) := 1.0;
             end if;
              Output_Data(l) := sqrt(abs(Input_Data(l)));
          end loop;
        end if;

        if Transform = 3 then
           for l in Input_data'range loop
              if Input_Data(l) < 0.0 then
          sign(l) := -1.0;
        else
          sign(l) := 1.0;
       end if;
         Output_Data(l) := ln(abs(Input_Data(l)));
     end loop;
  end if;

  if Transform = 4 then
     for l in Input_data'range loop
         Output_Data(l) := mult * Input_Data(l) + add;
     end loop;
   end if;

   if Transform = 5 then
      for l in Input_Data'range loop
      Output_Data(l) := ln ( Input_Data(l) +
                        sqrt(Input_Data(l)*Input_Data(l) + 1.0) );
        end loop;
   end if;

end if;   -- End Forward Transform Loop


if Inverse = 1 then

  if Transform = 1 then
     for l in Input_Data'range loop
         Output_Data(l) := Input_Data(l);
     end loop;
  end if;

  if Transform = 2 then
     for l in Input_Data'range loop
         temp := Input_Data(l)*Input_Data(l);
          if sign(l) = -1.0 then
             Output_Data(l) := -temp;
          else
             Output_Data(l) := temp;
          end if;
       end loop;
   end if;

   if Transform = 3 then
      for l in Input_Data'range loop
          temp := exp(Input_Data(l));
          if sign(l) = -1.0 then
             Output_Data(l) := -temp;
          else
             Output_Data(l) := temp;
          end if;
```

```
           end loop;
         end if;

        if Transform = 4 then
           for l in Input_Data'range loop
               Output_Data(l) := (Input_Data(l) - add)/mult;
           end loop;
         end if;

         if Transform = 5 then
            for l in Input_data'range loop
           Output_Data(l) := 0.50 * ( exp(Input_Data(l)) -
                                      exp(-Input_Data(l)) );
            end loop;
          end if;

     end if;    -- End Inverse Transform Loop


end Linear_Transform;
```

# Appendix E. *Ada Source Code For Compression and Error Calculations*

## *Percent Compression*

The following source code was used to perform the compression of coefficients in the experiments of Chapters III, IV, and V. It is in a generic form, because the ranges for the loops of the Compression package are determined by what precedes it (complex MW, sinusoidal MW, or one of the systems presented in Chapter IV). The data that is changed (in this version "Output_Data") also depends upon what precedes it (for example, the complex MW uses "FFT_Data" at this point of the program.

```
----------Compress the Homomorphic Coefficients ---------------------

    No_Zeros_R  := float(Percent_Compression) *
                   float(data'last);
    No_Zeros := integer(No_Zeros_R);
    Text_Io.New_line;
    Text_Io.put("No_Zeros = ");
    Integer_Io.put(No_Zeros);
    Text_Io.New_line;

    -- Find max output to establish initial threshold and to set the
    -- level for adjustment.
    for x in ( ) .. ( ) loop
          if Output_Data(x) > Max then
             Max := Output_Data(x);
          end if;
    end loop;

        Cep_Threshold := Percent_Compression * Max;
        Adjust        := 0.10 * Max;


    loop   -- Start Infinite Loop

       for x in (  ) .. (  ) loop
        -- Compression Output for each segment to file-
             if abs(Output_Data(x)) < Cep_Threshold then
                Temp(x) := 0.0;
                Compression_Count := Compression_Count + 1;
             else Temp(x) := 1.0;
             end if;
```

```
   end loop;  -- the for-loop inside of the infinite loop

        if (Compression_Count < No_zeros) then
            if TooLow > 0  then
               Cep_Threshold := Cep_Threshold + Adjust;
            end if;
            if TooHigh > 0 then
               Adjust := Adjust/2.0;
               Cep_Threshold := Cep_Threshold + Adjust;
            end if;
           TooLow  := 1;
           TooHigh := 0;
         end if;

         if (Compression_Count > No_Zeros) then
             if TooHigh > 0 then
                Cep_Threshold := Cep_Threshold - Adjust;
             end if;
             if TooLow > 0 then
                Adjust := Adjust/2.0;
                Cep_Threshold := Cep_Threshold - Adjust;
             end if;
            TooHigh := 1;
            TooLow  := 0;
           end if;

           if (Compression_Count = No_Zeros) then
              exit;
           end if;

        Compression_Count := 0;
        Cep_Threshold     := Cep_Threshold;
        Adjust            := Adjust;

end loop; --end of infinite loop

for x in ( ) .. ( ) loop
     -- Set compressed Output values to zero
     Output_Data(x) := Output_Data(x) * Temp(x);
     -- Print output values to the alpha file (coefficients).
     Float_Io.Put (Alphafile, Output_Data(x));
     Text_Io.New_Line(Alphafile);
end loop;
```

*Error Calculations Code*

The error calculation code below is for all real valued outputs and inputs. The complex MW produces complex outputs. The user must ensure that the correct type of data is being sent to this package. It was assumed that all inputs would be real and therefore all outputs from the CMW would also be real. In this case only the real part of "Recon_Data" is used. Also the RIV criteria that is calculated is actually (1 - RIV), because the values are often so close to one that it is difficult to distinguish them.

```
-------------------------ERROR_OUT------------------------------

   procedure Error_Out (Recon_Data  : in Real_Vector;
                        Actual_Data : in Real_Vector;
                        L2_Error    : in out float;
                        RMS_Error   : in out float;
                        L_INF_Error : in out float;
                        Norm_RIV    : in out float) is

       error,
        sum1,
        sum2,
        sum3,
        count          : float := 0.0;
        Recon_RIV      : float := 0.0;
        Actual_RIV     : float := 0.0;

  begin

     for l in Actual_Data'range loop
         if abs(Actual_Data(l) - Recon_Data(l)) > error then
            error := abs(Actual_Data(l) - Recon_Data(l));
         end if;
         sum1 := Actual_Data(l)*Actual_Data(l) + sum1;
         sum2 := Recon_Data(l)*Recon_Data(l) + sum2;
         sum3 := (Actual_Data(l)-Recon_Data(l))*
                   (Actual_Data(l)-Recon_Data(l)) + sum3;
         count := count + 1.0;
     end loop;

     RMS_Error   := sqrt(sum3)/count;
     L2_Error    := sqrt(abs(sum1-sum2)/sum1);
     L_INF_Error := error;
     Recon_RIV   := ln(sum2)/ln(2.0);
     Actual_RIV  := ln(sum1)/ln(2.0);
     Norm_RIV    := Recon_RIV/Actual_RIV - 1.00000000;

 end error_out
```

## *Bibliography*

1. Akansu, Ali N. and Frank E. Wadas. "On Lapped Orthogonal Transforms," *IEEE Transactions on Signal Processing*, *SP-40*(2):439–443 (February 1992).
2. Anderson, Timothy R. Thesis Discussions, June-November 1993.
3. Bednar, Bruce P. and T.L. Watt. "Calculating the Complex Cepstrum without Phase Unwrapping or Integration," *IEEE Transactions on Accoustics, Speech, and Signal Processing*, *ASSP-33*(4):1014–1017 (August 1985).
4. Bogert, Bruce P., et al. *Time Series Analysis: The Quefrency Alanysis of Time Series for Echoes; Cepstrum, Pseudo-autocovariance, Cross-cepstrum, and Saphe Cracking*. M. Rosenblatt, Editor. New York: Wiley, 1963.
5. Campen, Alan. "Gulf War's Silent Warriors Bind U.S. Units Via Space," *Signal*, *40*:81–84 (April 1991).
6. Childers, D.G., et al. "The Cepstrum: A Guide to Processing," *Proceedings of the IEEE*, *65*(10):1428–1443 (October 1977).
7. Coifman, Ronald, "Test of the Bell." Private Communication, April 1992.
8. Coifman, Ronald et Meyer, Yves. "Remarques sur l'analyse de Fourier à fenêtre," *C.R. Academie Sci Paris*, *t.312*(Serie I):259–261 (1991).
9. Huffman, D. A. "A Method of Construction of Minimum Redundancy Codes," *Proceedings of the IRE*, *40(10)* (September 1952).
10. Jin, D.J. and E. Eisner. "A Review of Homomorphic Deconvolution," *Reviews of Geophysics and Space Physics*, *22*(3):255–263 (November 1992).
11. Khare, Anil and Toshinori Yoshikawa. "Moment of Cepstrum and its Applications," *IEEE Transactions on Signal Processing*, *40*(11):2692–2702 (November 1992).
12. Lewis, T. R. and S. Mitra. "Application of Blind Deconvolution Restoration Technique to Space Imagery," *SPIE: Adaptive Signal Processing*, *1565*:221–226 (June 1991).
13. Lloyd, S. P. "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory*, *IT-28*:129–137 (March 1982).
14. Malvar, Henrique S. "Lapped Transforms for Efficient Transform/Subband Coding," *IEEE Transactions on Accoustics, Speech, and Signal Processing*, *38*(6):969–978 (June 1990).
15. Malvar, Henrique S. "Extended Lapped Transforms: Fast Algorithms and Applications," *Proceedings of International Conf. on Accoustics, Speech, and Signal Processing*, (D4b.4):1797–1800 (1991).
16. Malvar, Henrique S. *Signal Processing with Lapped Transforms*. Boston : Artech House, 1992.
17. Malvar, Henrique S. and David H. Staelin. "Reduction of Blocking Effects in Image Coding with a Lapped Orthogonal Transform," *Proceedings of International Conf. on Accoustics, Speech, and Signal Processing*, 781–784 (1988).
18. Malvar, Henrique S. and David H. Staelin. "The LOT: Transform Coding Without Blocking Effects," *IEEE Transactions on Accoustics, Speech, and Signal Processing*, *37*(4):553–559 (April 1989).
19. Meyer, Yves. *Wavelets, Algorithms and Applications*. Translated and Revised by Robert D. Ryan, Philadelphia: Society for Industrial and Applied Mathematics, 1993.

20. Nill, Norman B. and Brian H. Bouzas. "Objective Image Quality Measure Derived From Digital Image Power Spectra," *Optical Engineering*, *31*(4):813–825 (April 1992).

21. Oppenheim, Alan V. *Superposition a Class of Nonlinear Systems*. Technical Report Ph.D. Dissertation 432, Massachusetts Institute of Technology, March 1965.

22. Oppenheim, Alan V. and Ronald W. Schafer. *Discrete Time Signal Processing*. New Jersey: Prentice Hall, 1989.

23. Parsons, Thomas W. *Voice and Speech Processing*. McGraw Hill, 1987.

24. Polydoros, A. and A. T. Fam. "The Differential Cepstrum: Definition and Properties," *Proceedings IEEE Int Symp. Circuits Syst*, 77–80 (April 1981).

25. Quackenbush, Schuyler R. and Thomas P. Barnwell III. *Objective Measures for Speech Processing*. New Jersey: Prentice Hall, 1988.

26. Raduenz, Brian D. *Digital Signal Processing Using Lapped Transforms with Variable Parameter Windows and Orthonormal Bases*. MS thesis, AFIT/GE/ENG/92D-30, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

27. Randall, R. B. *Application of B & K Equipment to Frequency Analysis*. Naerum Denmark: Bruel & Kiser Instruments, 1977.

28. Rau, K. R. and P. Yip. *Discrete Cosine Transform: Algorithms, Advantages, Applications*. New York: Academic Press, 1990.

29. School of Engineering, Air Force Institute of Technology (AU). *Proceedings of the AFIT/AFOSR Workshop on the Role of Wavelets in Digital Signal Processing*, 12-13 March 1992.

30. Sokolov, R. T. and J. C. Rogers. "Time-Domain Cepstral Transformation," *IEEE Transactions on Signal Processing*, *41*(3):1161–1169 (March 1993).

31. Spanier, Jerome and Keith Oldham. *An Atlas of Functions*. New York: Hemisphere Publishing Corp., 1987.

32. Suter, Bruce W. Thesis Discussions, January-November 1993.

33. Suter, Bruce W. and Mark E. Oxley, "Getting Around The Balian-Low Theorem Using Generalized Malvar Wavelets." Technical Report; Air Force Institute of Technology, Wright-Patterson AFB, OH, October 1993.

34. Suter, Bruce W. and Mark E. Oxley. "On Variable Overlapped Windows and Weighted Orthonormal Bases," *IEEE Transactions on Signal Processing* (Submitted April 1992).

35. Switzer, Shane. *Frequency Domain Speech Coding*. MS thesis, AFIT/GE/ENG/91D, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

36. USSPACECOM. "United States Space Command Operations Desert Shield and Desert Storm Assessment,". Military Report, Peterson AFB, CO, January 1992.

37. Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. New Jersey: Prentice Hall, October 1992.

38. Weinstein, Clifford J. "Opportunities for Advanced Speech Processing in Military Computer-Based Systems," *Proceedings of the IEEE*, *79*(11):1627–1639 (November 1991).

39. Wesfried, Eva M. and Victor Wickenhauser, "Signal Processing via Fast Malvar Wavelet Transform Algorithm." Technical Report; Ceremade, University of Paris, Dauphine, 1993.

40. Williams, William J., et al. "Uncertainty, Information and Time-Frequency Distributions," *SPIE: Advanced Signal Processing Algorithms, Architectures, and Implementations II*, *1566*:144–156 (May 1991).

41. Young, Robert and Nick Kingsbury. "Frequency-Domain Motion Estimation Using a Complex Lapped Transform," *IEEE Transactions on Image Processing*, *2*(1):2–17 (Jan 1993).

## *Vita*

Captain Stephen R. Hall was born in Ridgewood, New Jersey on 8 July 1965. He graduated from Chautauqua Central School, Chautauqua New York in 1983. Captain Hall graduated magna cum laude from Grove City College with a Bachelor of Science degree in Electrical Engineering. He was an ROTC Distinguished Graduate and was commissioned into the US Air Force in 1987. Captain Hall was assigned to the 1st Space Operations Squadron, Falcon Air Force Base, Colorado. He served as a Satellite Operations Crew Commander and as the Assistant Chief of Current Operations for three satellite programs. Captain Hall entered the Air Force Institute of Technology in May 1992 to pursue a Masters in Space Operations. Upon graduation, Captain Hall will be assigned to the Air Force Technical Applications Center, Patrick Air Force Base, Florida. Steve is married to Lisa Todd Hall of Murrysville, Pennsylvania and has one daughter, Caitlyn.

Permanent address: 37 Evans Street
Mayville, New York 14757

**REPORT DOCUMENTATION PAGE**

*Form Approved*
*OMB No. 0704-0188*

| | | |
|---|---|---|
| 1. AGENCY USE ONLY (*Leave blank*) | 2. REPORT DATE<br>December 1993 | 3. REPORT TYPE AND DATES COVERED<br>Master's Thesis |

4. TITLE AND SUBTITLE
**FILTERING, CODING, AND COMPRESSION WITH MALVAR WAVELETS**

5. FUNDING NUMBERS

6. AUTHOR(S)
Stephen R. Hall, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GSO/ENG/93D-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Jon Sjogren, Ph.D.
AFOSR/NM
Bolling AFB, DC 20332-6448

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

This thesis develops and evaluates a number of new concepts and tools for the analysis of signals using Malvar wavelets (lapped orthogonal transforms). Windowing, often employed as a spectral estimation technique, can result in irreparable distortions in the transformed signal. By utilizing the Malvar wavelet, any signal distortion resulting from the transformation can be eliminated or cancelled during reconstruction. This is accomplished by placing conditions on the window and the basis function and then incorporating the window into the orthonormal representation. Paradigms for both a complex-valued and a real-valued Malvar wavelet are summarized. The algorithms for the wavelets were implemented in the DOD standard language, Ada. The code was verified to ensure perfect reconstruction could be obtained and experiments were performed using the wavelet programs. Various compression techniques were investigated and evaluated using the Malvar wavelet in both homomorphic and non-homomorphic systems. The Malvar wavelet has the unique ability to overlap adjacent windows without increasing the number of transform coefficients. Various amounts of window overlap were investigated to determine if there is an optimal amount which should be used. In addition, the real-valued basis function was used to attempt real-valued deconvolution.

14. SUBJECT TERMS
Wavelets, Malvar Wavelet, Lapped Transform, LOT, Homomorphic Filters

15. NUMBER OF PAGES
154

16. PRICE CODE

| | | | |
|---|---|---|---|
| 17. SECURITY CLASSIFICATION OF REPORT<br>Unclassified | 18. SECURITY CLASSIFICATION OF THIS PAGE<br>Unclassified | 19. SECURITY CLASSIFICATION OF ABSTRACT<br>Unclassified | 20. LIMITATION OF ABSTRACT<br>UL |