Audio Denoising Using Wavelet Filter Banks Aimed at Real-Time Application

by

Peter W. Kassakian

B.S., Massachusetts Institute of Technology (1995)

Submitted to the Department of Mechanical Engineering
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

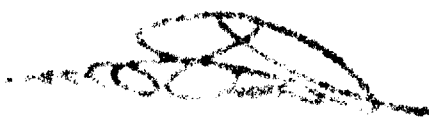
June 1999

Author _____
Department of Mechanical Engineering
May 20, 1999

Certified by _____
Bernard C. Lesieutre
Associate Professor of Electrical Engineering
Thesis Supervisor

Certified by _____
Derek Rowell
Professor of Mechanical Engineering
Thesis Supervisor

Accepted by _____
Ain A. Sonin
Chairman, Departmental Committee on Graduate Students



Audio Denoising Using Wavelet Filter Banks Aimed at Real-Time Application

by

Peter W. Kassakian

Submitted to the Department of Mechanical Engineering
on May 20, 1999, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

In this thesis we implement an audio denoising system targeted at real-time application. We investigate the delays associated with wavelet filter bank systems and propose methods to overcome them. We consider computational delays as well as structural delays and deal with each separately. We find that good denoising performance is achieved when filters are chosen so as to transform the input into an appropriately scaled time-frequency domain. Multiplications required for four different implementations of the system are compared and we conclude that a scheme that performs multiplications as far downstream as possible proves to be the best of the four options. Finally we propose several topics for further research including investigations into the human auditory system as well as studies of creative filter bank topologies.

Thesis Supervisor: Bernard C. Lesieutre
Title: Associate Professor of Electrical Engineering

Thesis Supervisor: Derek Rowell
Title: Professor of Mechanical Engineering

Dedication

---

to

my father

Acknowledgements

I would like to express my gratitude to Professor Lesieutre for advising my thesis and taking a great interest in a topic that's at best tangential to his current research. He made it a pleasure to learn about a subject that is multifaceted and complicated.

Also I want to thank Professor Rowell for reading my thesis on behalf of the department. I'm pleased that he took such an interest in the subject and value his advice on many issues. He also gave me the gift of a very enjoyable TA experience.

I wish to thank Leslie Regan and Joan Kravit for making me feel welcome in their office. Also thank you very very much, Vivian, Karin, and Sara.

I want to acknowledge Professor Pratt for allowing me to reconsider my involvement in his project, and for being a kind and caring individual who I will always remember. Professor Verghese deserves special mention too, for his guiding words and insights into these strange filter bank systems.

Finally I want to thank Mom, Meg, Ann, Andrew, and Dennis for their support over the last few years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis we look at a certain class of systems: wavelet filter bank systems. We are interested in using them in real-time to denoise audio.

The motivation for this work is derived from the desire to use these relatively efficient systems to operate in a setting different from internet related application, where wavelets have gained much attention. Internet compression applications are similar to our denoising system, but do not require a strictly causal system. Our system is geared towards the goal of denoising a piece of audio on the fly, so it could be used quickly in recording situations, or for live performance. This idea sets up an interesting challenge and creates a new angle for viewing these fascinating wavelet systems.

1.1 Organization of Thesis

The thesis is naturally broken into eight chapters, including this one. Chapters 2, 3, and 4 present background material necessary for the understanding of the more subtle discoveries explained in the later chapters. Wavelet systems are different from linear time-invariant systems because they involve a non-time-invariant operator, the downsampler. It's for this reason that these background chapters are included. The three chapters provide a self-contained foundation of material that can easily be referred to while reading the remainder of the thesis.

Chapter 2 describes the theoretical basis for the denoising algorithm as seen in from the point of view of stochastic theory. Several terms are defined which will be used frequently throughout the thesis. Also included in Chapter 2 are several interesting and key plots that were created using our system. They have been chosen to make certain theoretical points, however they also represent a graphical product of our work.

Chapter 3 approaches the system from a different angle; that of the joint time-frequency plane. We show that this is an intuitively satisfying way to view the process. Showing that filter banks can naturally arise out of the assumptions of this chapter, we set the stage for the following chapter which describes discrete wavelets, and wavelet transforms in general. In Chapter 3 we

also discuss the very important concept that a good denoising system must take into account the natural time-frequency characteristics of the human ear, as well as the mathematical structure of the input signal.

Discrete-time wavelets are introduced in Chapter 4. We derive the conditions for "perfect reconstruction", and introduce the "polyphase matrix". Also discussed are the delays associated with orthogonal transforms. This chapter concludes the necessary background for the rest of the thesis.

Chapter 5 is concerned with the measures of performance for our systems, and presents results about the time/frequency resolution associated with our system. The chapter introduces in more detail than previous chapters the problems faced in our actual implementation.

Chapter 6 contains the primary results of our research. We discuss four different methods of implementing a wavelet filter bank system, and compare the number of multiplies that are required for computation. The concern in this chapter is that of the *computational* delay associated with these types of transforms. Also discussed are general conclusions applicable to other problems in signal processing.

Chapter 7 presents our findings with respect to overcoming delays not associated with computation, but with the inherent structure of the system. We show that there exists a certain class of matrices called unimodular, that help to greatly reduce the unavoidable delay suffered by the heavily studied paraunitary matrices. We see that it is difficult to design such matrices to meet all of our constraints, and view this topic as an opportunity for future research.

Finally, in the Conclusion, we recount our findings and suggest possible other research areas that appear fruitful. The Appendix holds the C code used to perform our transformations. The code takes as arguments an input signal, output data file, threshold for coefficient squelching, number of frequency bands desired, and four filters for analysis/synthesis.

Chapter 2

Transformations and Basis Functions

2.1 Transformations

In order to discuss different classes of transforms, it's necessary to make clear the definition of a transform. This thesis uses the term *transform* liberally. When a sequence of numbers is altered in any way, it has been transformed. For example, passing a signal through a low-pass filter is a type of transform; the "convolution" transform. This concept of generalized transforms proves to be a valuable way of thinking about signal processing in general.

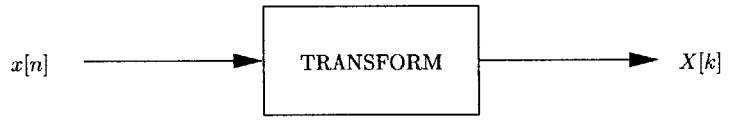


Figure 2.1: **A Transform**

2.1.1 Basis Functions

Another valuable conceptual tool is the idea of basis functions. A signal of length 5 has 5 values that indicate the weighting of 5 individual basis functions. In the time domain, the basis functions are {0 0 0 0 1}, {0 0 0 1 0}, {0 0 1 0 0}, etc. Each one of the time basis functions represents a point in time. Likewise, the discrete Fourier basis functions are the complex exponentials $\frac{1}{5}e^{j(2\pi/5)0n}$, $\frac{1}{5}e^{j(2\pi/5)1n}$, $\frac{1}{5}e^{j(2\pi/5)2n}$, $\frac{1}{5}e^{j(2\pi/5)3n}$, $\frac{1}{5}e^{j(2\pi/5)4n}$. These functions are naturally derived from the definition of the DFT shown below [13]. All basis functions are referenced to the time (n) domain.

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]e^{j(2\pi/N)kn}. \tag{2.1}$$

2.1.2 Orthogonality of Basis Functions

Two basis functions are considered **orthogonal** if they have zero correlation with one another. In precise terms, if $f_j[n]$ and $f_k[n]$ are two orthogonal basis functions, they must satisfy

$$\sum_{n=-\infty}^{\infty} f_j[n] f_k[n] = 0 \quad , \quad j \neq k. \tag{2.2}$$

A basis is an **orthogonal basis** if all the basis functions are each orthogonal with one another. The basis is **orthonormal** if the correlation of each of the basis functions with itself is 1:

$$\sum_{n=-\infty}^{\infty} f_k[n] f_k[n] = 1 \quad \text{for all} \quad k. \tag{2.3}$$

Note that (2.3) is actually a measure of energy since it is a sum of squares. An **orthonormal transform** is a transformation that can be written in the form (2.4). This merely states that the time series $x[n]$ can be written as a weighted sum of mutually orthonormal basis functions:

$$x[n] = \sum_{k=0}^{N-1} X[k] f_k[n], \quad f_0, f_1, ..., f_{N-1} \text{ orthonormal} . \tag{2.4}$$

It's worth pointing out a few important properties of orthonormal transforms. The first is that the transforms are energy preserving. This is seen neatly in the following small proof.

$$\sum_{n=0}^{N-1} x^2[n] = \sum_{n=0}^{N-1} (X[0]f_0[n] + X[1]f_1[n] + X[2]f_2[n] + ...)^2 \tag{2.5}$$

$$\sum_{n=0}^{N-1} x^2[n] = \sum_{n=0}^{N-1} \{(X^2[0]f_0^2[n] + X^2[1]f_1^2[n] + ...) + (X[0]X[1]f_0[n]f_1[n] + ...)\} \tag{2.6}$$

$$\sum_{n=0}^{N-1} x^2[n] = X^2[0] + X^2[1] + X^2[2] + ... \tag{2.7}$$

$$\sum_{n=0}^{N-1} x^2[n] = \sum_{k=0}^{N-1} X^2[k] \tag{2.8}$$

A second property of orthonormal transforms follows from the above proof. The components in the transform domain are energetically decoupled. So setting a transform coefficient $X[k_0]$ to zero removes exactly $X^2[k_0]$ of "energy" from the signal in both the time and transform domains.

2.1.3 Vector Spaces

Exploring further the concept of orthonormal transformations, we see that these operations can be thought of as coordinate system rotations. Take for example a deterministic signal that can be represented by two points in time, say {1 1}. This signal point can be plotted on the orthonormal "time" coordinate system shown in Figure 2.2 where the x-axis and y-axis represent the trivial basis functions {1 0} and {0 1} respectively. Transforming this signal to a different orthonormal basis corresponds to rotating the axes of the "time" coordinate system. This is proven simply because of the fact that an orthonormal transform must preserve the energy of the signal. The "energy" of the signal is calculated by squaring the signal's distance from the origin, in this case $1^2 + 1^2 = 2$.

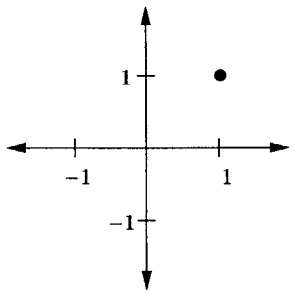


Figure 2.2: **Signal {1 1}** A deterministic signal of length 2 represented as a point on a plane

The coefficients of the transformed signal are constrained by this energy preserving property.

Thus the transform coefficients c_0, c_1 must be related by $c_0^2 + c_1^2 = 2$ This is the equation of a circle about the origin, so in this two dimensional example all that's needed to specify the orthonormal transform is a rotation angle.

Note that angle of rotation can be chosen such that all the energy is contained in one transform coefficient. Choosing the transform in this way implies that one of the basis functions is pointed exactly in the same direction as the signal of interest. In other words it is a scaled version of the signal of interest (in this case $\frac{1}{\sqrt{2}}\{1\ 1\}$). The other basis function[s] would naturally be orthogonal to the signal, (in this case $\frac{1}{\sqrt{2}}\{1\ -1\}$). Figure 2.3 shows the geometry of this example.

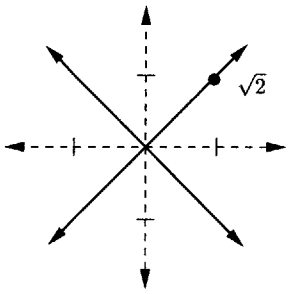


Figure 2.3: **Transformed Signal** $\{\sqrt{2}\ \ 0\}$ An orthonormal transformation corresponds to a rotation of axes. In the above case, the axes are rotated so as to align with the signal point.

As implied above, these concepts hold in many dimensions as well as just two. Of course it is difficult to visualize a signal of length 5 as being a point in a 5 dimensional space, save visualizing an infinite dimensional space as would be required in general for a continuous-time signal. The problem of aligning the axis in the same direction as the signal of interest can be thought of as an eigenvalue/vector problem.

2.2 Estimation of a Stochastic Signal

2.2.1 Karhunen-Loeve Transform

The example in Section 2.1 showed how a deterministic signal can be represented in two different orthonormal bases. In this thesis we are concerned with musical signals which are random in some senses and organized in others. The relation between randomness and organization can be quantified approximately by the autocorrelation function $\phi_{xx}[m]$, which explains how strongly correlated different sample points are with one another. Equation (2.9) defines the autocorrelation function where $\mathcal{E}\{\mathbf{x}\}$ is the expectation of the random variable $\mathbf{x}$. Note that it makes the assumption that this function is the same for all n, which is reasonable in most cases.

$$\phi_{xx}[m] = \mathcal{E}\{\mathbf{x}[n]\mathbf{x}[n+m]\} \tag{2.9}$$

The autocorrelation function allows us to speak in probabilistic terms about random sequences. Since we don't know a priori the energy distribution among the samples of a random process, we settle for what we do know which is the *expected* energy distribution among the samples. We will see later that an interesting transformation is one that squeezes the most expected energy into the least number of transform coefficients. Figures 2.2 and 2.3 show this type of transformation where the signal is not stochastic. The same example is depicted in Figure 2.4, where the signal is stochastic and can be described by the autocorrelation matrix,

$$\begin{bmatrix} \mathcal{E}\{\mathbf{x}[0]\mathbf{x}[0]\} & \mathcal{E}\{\mathbf{x}[0]\mathbf{x}[1]\} \\ \mathcal{E}\{\mathbf{x}[1]\mathbf{x}[0]\} & \mathcal{E}\{\mathbf{x}[1]\mathbf{x}[1]\} \end{bmatrix} = \begin{bmatrix} \phi_{xx}[0] & \phi_{xx}[1] \\ \phi_{xx}[-1] & \phi_{xx}[0] \end{bmatrix} = \begin{bmatrix} 1 & .9 \\ .9 & 1 \end{bmatrix}. \tag{2.10}$$

Notice that the autocorrelation matrix is related only to the second-order statistics of the random process. The complete probability density functions for each random variable in the random sequence are not known. For most circumstances, this is all right, and the autocorrelation matrix proves useful. Figure 2.4 assumes that the probability density functions of each of the two samples are both Gaussian with zero mean.

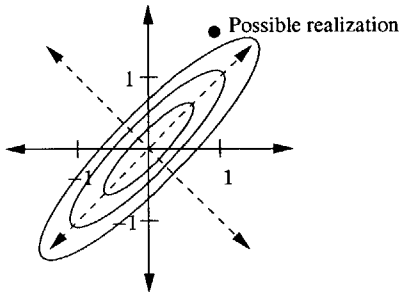


Figure 2.4: **Stochastic Signal of Length 2** A signal specified only by a mean and an autocorrelation matrix. The ellipses are lines of constant probability.

The same transform used to rotate the coordinate system in Figure 2.3 could be used in this stochastic example to maximize the expected energy difference between $\mathbf{x}[0]$ and $\mathbf{x}[1]$. This linear orthonormal transformation is called the **Karhunen-Loeve** transform and is optimal in the sense that it diagonalizes the autocorrelation matrix, and consequently places the expected energies of the transform coefficients at their maxima and minima (the eigenvalues of the autocorrelation matrix).

The basis functions are the eigenvectors of the autocorrelation matrix. So in the above example, we see that the basis functions of the **KLT** (Karhunen-Loeve Transform) are $\{\frac{\sqrt{2}}{2}\ \frac{\sqrt{2}}{2}\}$, and $\{\frac{\sqrt{2}}{2}\ -\frac{\sqrt{2}}{2}\}$. More importantly the expected energies of the transform coefficients (the eigenvalues) are 1.9 and .1. One is big and one is small.

The KLT transforms a stochastic signal into a signal with very large and very small coefficients. If it were necessary to approximate the signal with only a few coefficients, this property would be ideal; we needn't keep the tiny coefficients - only the big ones. We might be able to store 99% of the signal's energy in half the number of coefficients. This forms the foundation of most compression schemes.

2.2.2 Stochastic Signal with White Noise

Consider a stochastic signal comprised of nothing but **white noise**. Process $\nu[n]$ is white if there is no correlation between any two sample points, i.e., the autocorrelation function is given by Equation 2.11.

$$\phi_{\nu\nu}[m] = \sigma_\nu \delta[n] \tag{2.11}$$

This corresponds to an autocorrelation matrix that is given by Equation 2.12.

$$\begin{bmatrix} \mathcal{E}\{\nu[0]\nu[0]\} & \mathcal{E}\{\nu[0]\nu[1]\} & \cdots \\ \mathcal{E}\{\nu[1]\nu[0]\} & \mathcal{E}\{\nu[1]\nu[1]\} & \cdots \\ \vdots & & \ddots \end{bmatrix} = \begin{bmatrix} \phi_{\nu\nu}[0] & \phi_{\nu\nu}[1] & \cdots \\ \phi_{\nu\nu}[-1] & \phi_{\nu\nu}[0] & \cdots \\ \vdots & & \ddots \end{bmatrix} = \begin{bmatrix} \sigma_\nu & 0 & \cdots \\ 0 & \sigma_\nu & \cdots \\ \vdots & & \ddots \end{bmatrix} \tag{2.12}$$

The autocorrelation matrix as shown in Equation 2.12 is a multiple of the identity matrix, and therefore has eigenvalues that are always σ_ν regardless of the orthonormal transform. Geometrically, lines of equal probability form circles - not ellipses, so every orthonormal transform is the KLT for white noise [14].

2.2.3 Denoising A Stochastic Signal

This leads to the main method of denoising used in this thesis. Given a stochastic signal (music) corrupted with white noise, we seek a method of extracting the signal from the noise. The method used is similar to that used in compression. An orthonormal basis is found (preferably related to the KL basis), and the signal is transformed to that vector space. Small coefficients are discarded, and the signal is transformed back to the time domain.

Because the transform is presumed linear, and the signal of interest is a linear sum of music $\mathbf{x}[n]$ and noise $\nu[n]$, we expect the energy of the musical part to aggregate itself in a few coefficients (as promised by the KLT) and the distribution of energy of the noise to be unaffected by the transformation. Therefore removing small coefficients removes more noise than music (especially if the noise was small to begin with). For example if the musical part of a signal of length 1000 happened to aggregate almost all of its energy into 50 coefficients, removing the other (small) coefficients would result in removing 95% of the noise (along with a small amount of signal). The Figures below make this point graphically.

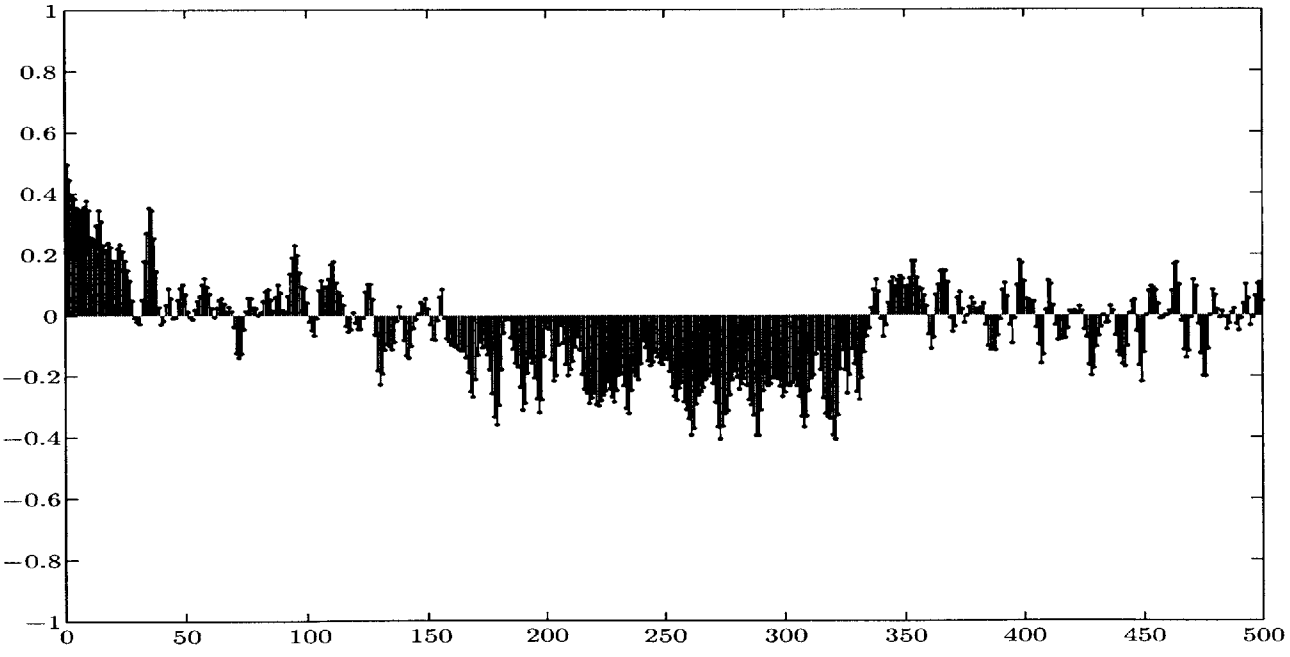


Figure 2.5: **Noise Corrupted Musical Signal Represented in Time Domain**

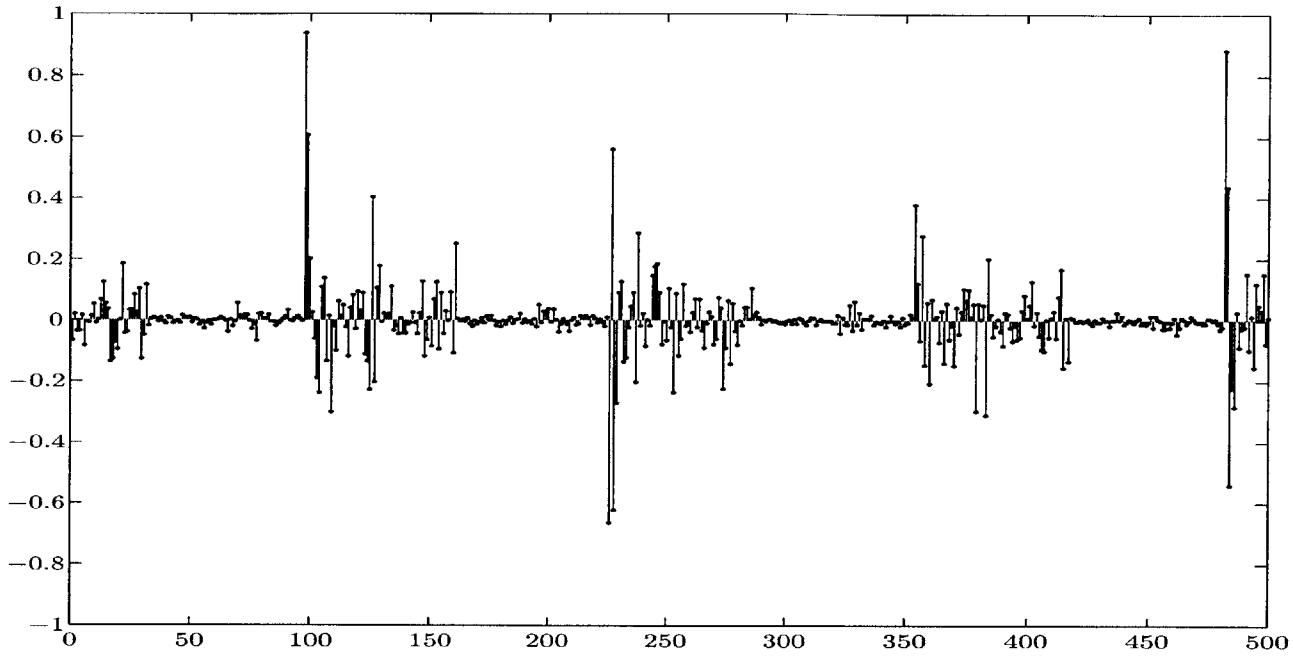


Figure 2.6: **Noise Corrupted Musical Signal Represented in Transform Domain**

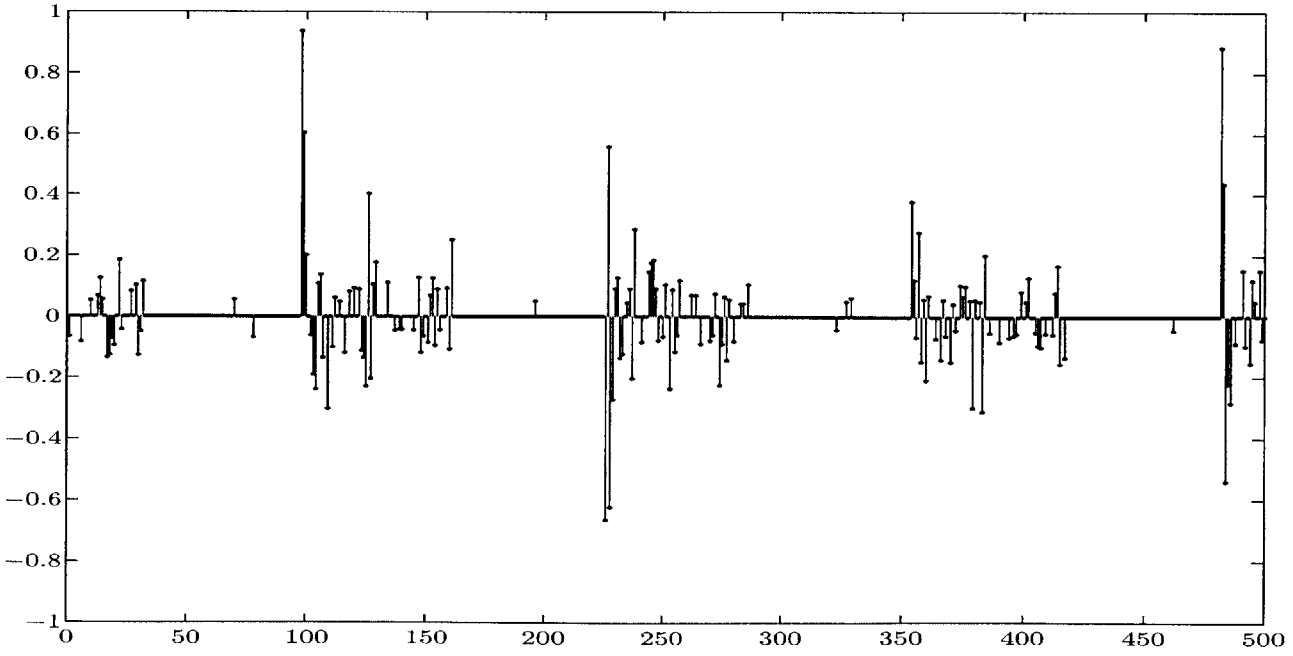


Figure 2.7: **Denoised Musical Signal Represented in Transform Domain (Small Coefficients Discarded)** Here we set a threshold value and remove the smallest coefficients thereby gaining a higher SNR

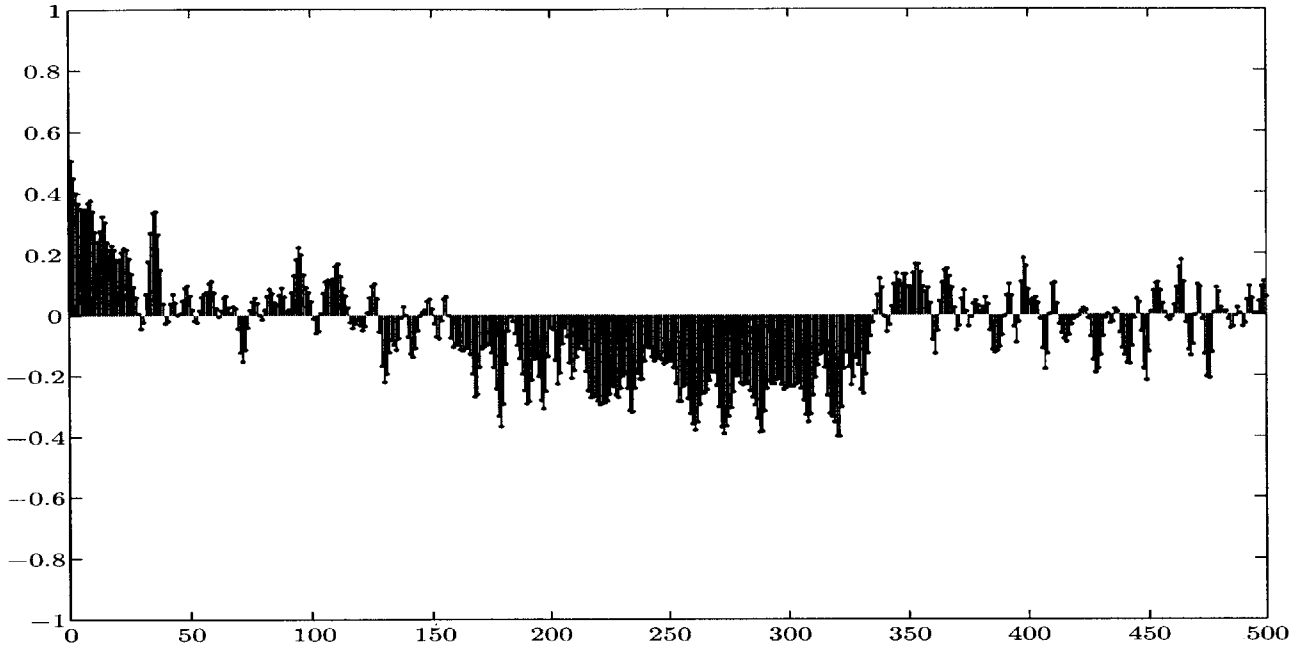


Figure 2.8: **Denoised Musical Signal Represented in Time Domain** The signal is reconstructed from the thresholded coefficients of Figure 2.7.

Chapter 3

Joint Time-Frequency Analysis

There are multiple ways of understanding the denoising scheme of Chapter 2. Karhunen-Loeve methods treat a musical signal like an arbitrary stochastic signal. The KL methods make the assumption that nothing except the autocorrelation matrix and the mean are known about the signal. Because of this, the KLT bears a high computational cost both in calculating the basis functions, and in actually transforming the signal [7]. Other methods are better for real-time processing.

3.1 Smooth Signals and the Time-Frequency Plane

The signals we are interested in denoising in this thesis are smooth, meaning that they are comprised of bursts of sine waves. That is the nature of music and many other natural processes. Music is made up of sequences of notes, each with harmonic content and each lasting for finite durations. Because of this information we can readily establish a basis that approximates the KL basis without enduring an undue amount of computation. The approximations are more intuitive and can prove to be more appropriate also.

The goal of the KL basis is to lump large numbers of highly correlated samples into only a few transform coefficients. Because the signals of interest are made up of small bursts of harmonically related sine waves, a good basis would be one in which one transform coefficient represented one sine wave burst.

Good bases like these are important for just the reasons outlined above. One such basis is the **Short-Time Fourier Transform (STFT)** [13], defined in Equation 3.1.

$$X[n,k] = \sum_{m=0}^{L-1} x[n+m]w[m]e^{j(2\pi/N)km} \tag{3.1}$$

The STFT is not orthonormal in general, but possesses the desirable property that the basis functions are time limited sine waves (similar to musical notes). Applying the STFT is identical to taking a spectrogram of the signal. It shows how the frequency content of the signal changes through time. The signal is transformed from the time domain to the **time-frequency domain**.

The time-frequency domain is a good place to operate from when denoising an audio signal. Unlike the pure frequency (Fourier) domain, there are many different transforms that could be classified as **time-frequency transforms**. This thesis explores the implementation of **wavelet transforms**, which are designed to efficiently transform from the time domain to the time-frequency domain. The wavelet transform's basis functions are little waves of finite duration, hence the name "wavelet".

Figure 3.1 is a plot of the transform coefficients of a wavelet expansion of a piece of music corrupted by noise. The music consists of a metronome clicking each beat and a guitar playing an ascending scale. Each dot represents one transform coefficient. Because the basis functions are localized in both time and frequency, each transform coefficient occupies a region in the time-frequency plane. In the plot, we can see information about the signal that we couldn't see in either the pure time or pure frequency domains (Figure 3.2). There are many beautiful elements of wavelet transforms, and one is that orthogonality is possible. The transform used to create Figure 3.1 is orthogonal.

Notice that if we just used a low-pass filter to attempt to remove the small coefficients from the Fourier domain, we run the risk of losing the sharpness (high frequency content) of the metronome clicks. The joint time-frequency domain allows for removing selective coefficicients without disturbing the natural time-frequency structure of the music. Most of the music's energy is in the low frequency bands, but the attack of the metronome clicks are important too. Human ears hear music in both time and frequency. So the basis functions we choose should reflect a balance between what's important to the ear, and what's implementable [9]. The Karhunen-Loeve transform represents an optimal compression algorithm because it creates the best approximation of the signal with the fewest coefficients, but it might not be what the ear is looking for. This is why (although related to the KLT sometimes), working in the joint time-frequency plane is better.

3.2 Time-Frequency Transforms by Filter Banks

A way to transform a signal from the time domain to the time-frequency domain is through the use of **filter banks**. The signal is put through parallel filters each of which corresponds to a different frequency band. The outputs of this type of system (shown in Figure 3.3) are multiple streams of

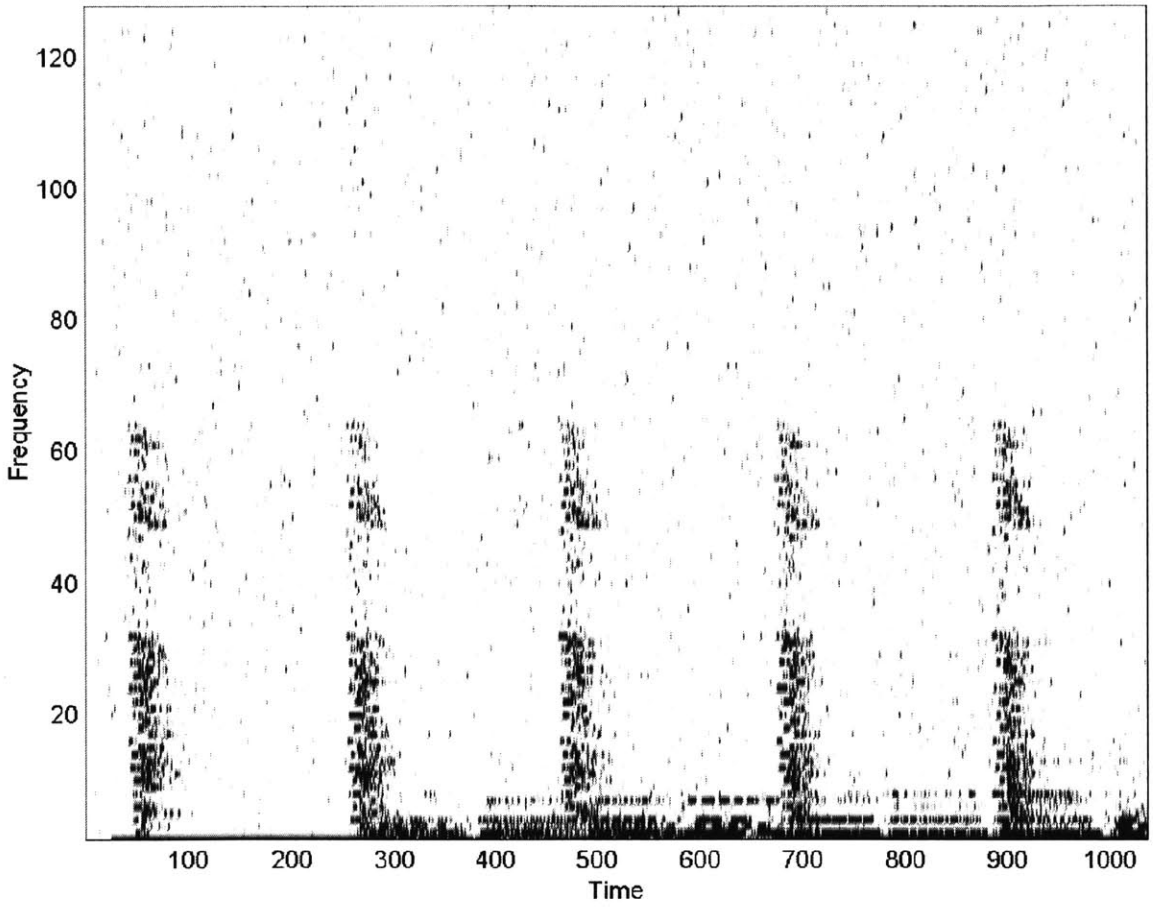


Figure 3.1: **Wavelet Transform Coefficients Plotted Against Time and Frequency** Each coefficient can be localized in time and frequency simultaneously. The noise is spread out evenly, and much of it can be removed by setting all the light coefficients to zero.

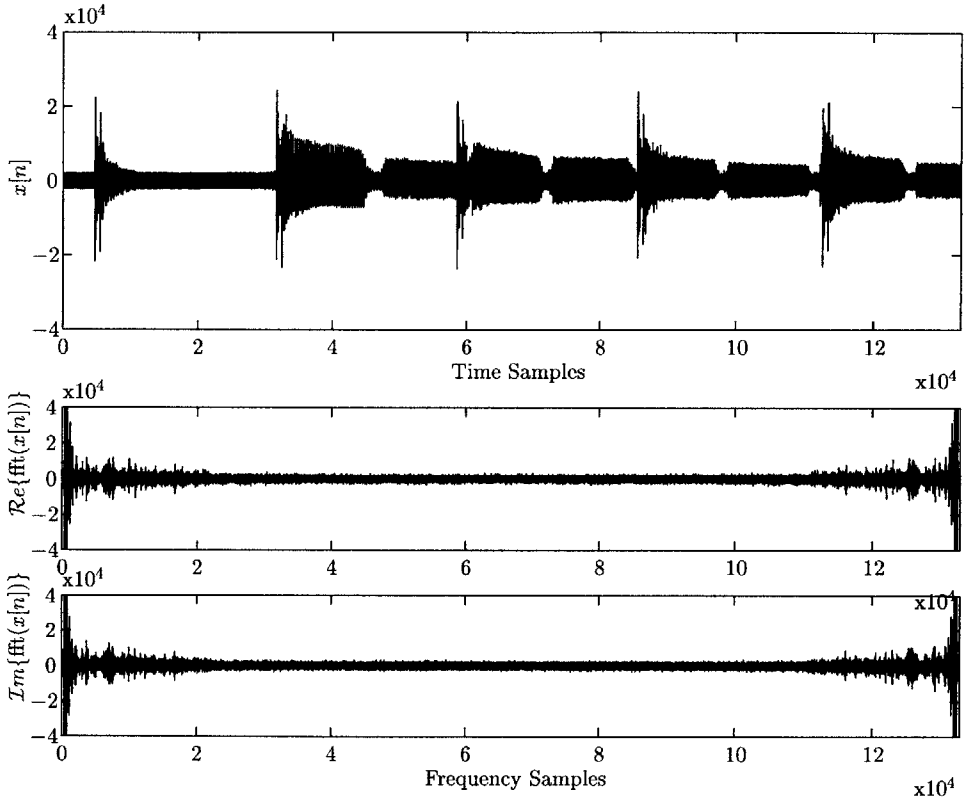


Figure 3.2: **Time Waveform and Fourier Transform of the Same Signal of Figure 3.1** Although the Fourier transform does give information about the signal of interest (that it has a lot of low-frequency energy), it doesn't show the structure seen in Figure 3.1.

coefficients. The STFT can be organized this way. Because the signal has been smeared through filters, each transform coefficient no longer corresponds exactly to a point in time. The coefficients correspond to *regions* of time, the size of which is dictated by the length of the filters in the bank.

A filter bank as shown in Figure 3.3 produces M times as much data as is contained by the signal. Most useful transforms maintain the same number of coefficients across domains, and this is guaranteed to be the case for orthogonal transforms. This is the motivation behind **downsampling**. In essence, the downsampling operator discards every other sample of a signal, and squeezes the remaining sample points into a sequence half its original length. Equation (3.2) defines this operator precisely.

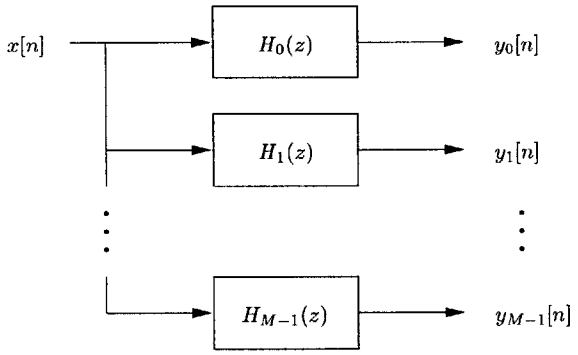


Figure 3.3: **General Filter Bank** A signal of length N can be filtered into M frequency bands, resulting in approximately $M \times N$ output samples

$$y[n] = (\downarrow 2)x[n] = x[2n] \tag{3.2}$$

A two-channel wavelet filter bank is shown in Figure 3.4. It is fine to discard half of the output samples, because the filters can be designed such that any information lost in one channel is kept in the other. So if $H_0(z)$ is low-pass, we might wish $H_1(z)$ to be high-pass. In this way each of the two transform samples occupies a different region in frequency. A transform like this cuts the time-frequency plane into 2 frequency parts, and $N/2$ time parts, where N is the length of the time signal.

It's extremely advantageous to maintain the same number of coefficients across the transform. This is because the real challenge of denoising (and taking transforms in general) lies not necessarily in the theory, but in the computation time. Faster is better. Unless needed for subtle reasons, redundant data should be discarded.

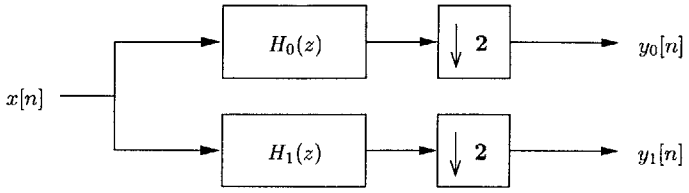


Figure 3.4: **Two Channel Filter Bank with Downsampling** The downsampling operator makes it possible to maintain the same number of samples in the transform domain as the time domain. No information is lost if the filters are chosen judiciously.

Using filter banks and downsampling, one can create systems of arbitrary complexity. For example, a bank like the one in Figure 3.3 can be designed with the addition of $\downarrow M$ operators to eliminate hidden redundancy. Alternatively, two more two-channel banks could be put after the two outputs y_0 and y_1 of Figure 3.4, resulting in a four-channel system. Carried further, this branching of banks results in large tree structures that turn out to have very nice computational properties. Figure 3.5 shows such a structure. Note that a structure like this also is guaranteed to be orthogonal if the intermediary two-channel banks were orthogonal.

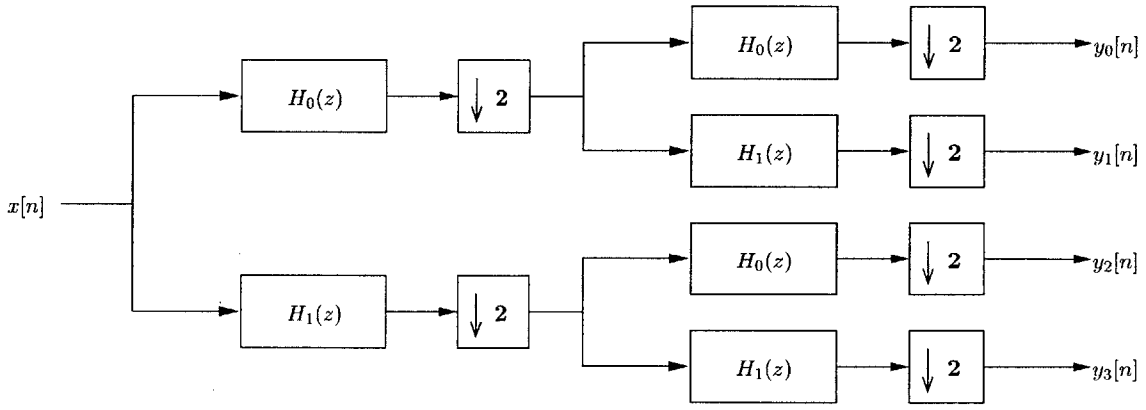


Figure 3.5: **Four Channel Filter Bank** Tree structures such as these prove to be computationally efficient due to the recursive nature of the tree.

The next chapter will look closely at filter banks such as the ones illustrated above. In particular we will find that systems like these can be designed to perfectly reconstruct the input given the transform. So investigating the *inverses* of these banks is the primary topic of Chapter 4. We will calculate the conditions for perfect reconstruction, and also see that we can create many different filter bank systems that are orthogonal or have other properties if needed.

Chapter 4

Discrete-Time Wavelets and Perfect Reconstruction

The topic of wavelets is multifaceted. Mathematicians have their reasons for investigating them, as do engineers. This thesis is concerned with the possibility of using **wavelet transforms** (transforms composed of filter banks as depicted in Figure 3.4) to remove noise from a signal in real-time.

The motivation stems from the fact that the transforms are fast (comparable to the FFT) [19], and they convert from the time domain to the joint time-frequency domain which is ideal for denoising music and most other natural processes. The *process* is non-linear since it involves squelching coefficients to zero, not simply filtering them. The *transform* is non time-invariant since it involves downsampling. It might be possible to achieve close to real-time denoising in new and improved ways with wavelets.

4.1 Conditions for Perfect Reconstruction

One of the advantages of working with wavelet transforms is that the analysis transform can be viewed as a filter bank. Similarly, the synthesis transform (inverse of the analysis transform) can also be represented as a filter bank. We will look first, though, at the system shown in Figure 4.1.

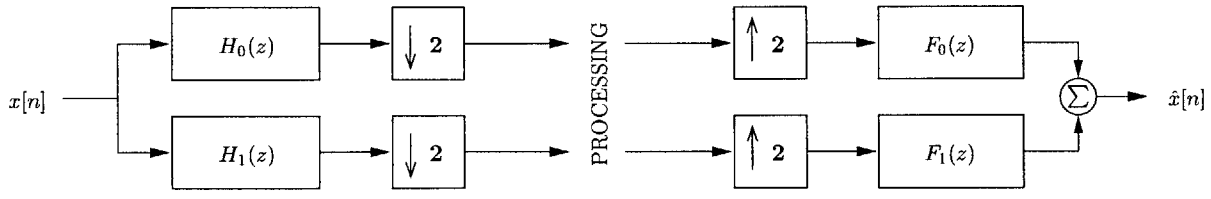


Figure 4.1: **Filter Bank Complete with Reconstruction** This is a two-channel filter bank. If no processing is performed, the filters $H_0(z)$, $H_1(z)$, $F_0(z)$, and $F_1(z)$ can be chosen to perfectly reconstruct $x[n]$, i.e., $\hat{x}[n] = x[n-l]$, where l is the delay in samples. Larger tree structures can be built from this basic system.

Referring to Figure 4.1, we wish to find filters $H_0(z)$, $H_1(z)$, $F_0(z)$, and $F_1(z)$ such that

$\hat{x}[n] = x[n-l]$, where l is an overall delay. Further, it's desirable to work with *causal* FIR filters. Since this thesis is concerned with near real-time processing, l should be made as small as possible. Depending on what properties of the transform are desired, a small l may or may not be possible.

The system is rewritten in what's known as the **polyphase** form shown in Figure 4.2. Even-indexed samples are put through the first channel, and odd-indexed samples through the second. Then they meet the analysis polyphase matrix, $\mathbb{H}_p(z)$, which filters the samples appropriately. Likewise on the synthesis side, the samples are put through the corresponding synthesis polyphase matrix, $\mathbb{G}_p(z)$, upsampled (zeros inserted between the samples), and multiplexed. This is a useful form to work with because the conditions for perfect reconstruction (when no processing occurs) are reduced to the constraint that the product of the two polyphase matrices equal the identity. Also the polyphase form is more efficient from a computational point of view, since the downsampling is done before the filtering, saving computation.

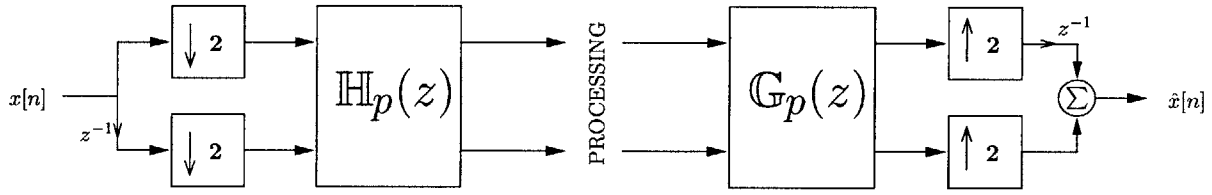


Figure 4.2: **Equivalent Polyphase Form of System in Figure 4.1** The polyphase form is easier to analyze and also faster to implement since the downsampling occurs before the filtering.

Equation (4.1) shows the requirement for perfect reconstruction in terms of the polyphase matrices. It's important to recognize that an identity matrix scaled by delay z^{-l} in between the downsampling and upsampling will result in perfect reconstruction. If the polyphase matrices themselves were identities, the system reduces to a demultiplexer/multiplexer. It is more interesting when the two polyphase matrices are meaningful with respect to the time-frequency plane, i.e., they perform frequency filtering functions.

$$\mathbb{G}_p(z)\mathbb{H}_p(z) = z^{-l}\mathbb{I} \tag{4.1}$$

Equation 4.1 can be written in expanded form,

$$\begin{bmatrix} G_{00}(z) & G_{01}(z) \\ G_{10}(z) & G_{11}(z) \end{bmatrix} \begin{bmatrix} H_{00}(z) & H_{01}(z) \\ H_{10}(z) & H_{11}(z) \end{bmatrix} = \begin{bmatrix} z^{-l} & 0 \\ 0 & z^{-l} \end{bmatrix} \tag{4.2}.$$

Note that if this were a *single* channel system, i.e., a filter and an inverse filter, it would

be impossible to achieve perfect reconstruction unless one of the two filters were IIR. This is not the case with the two channels, and it is another reason a system like this is special. Many sets of filters satisfy the perfect reconstruction constraint. It is not the goal of this thesis to analyze the pros and cons of each set. It serves as good background to mention a few things about a couple of them, however.

4.1.1 Haar and Conversion from Polyphase Form

Perhaps the simplest set of filters that satisfies **PR** (perfect reconstruction) is the **Haar** filter set. The filters in the Haar Polyphase matrix are scalars - they are zero for all $n \neq 0$ (4.3). They also create an orthogonal transform. All orthogonal transforms have the property that the inverse of the synthesis matrix is the conjugate transpose of the analysis matrix ($\mathbb{G}_p(z) = \mathbb{H}_p^{\star T}(z)$). This is trivial for the Haar example.

$$\underbrace{\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}}_{\mathbb{G}_p(z)} \underbrace{\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}}_{\mathbb{H}_p(z)} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.3}$$

It is useful to convert the polyphase form into the more standard filter bank form shown in Figure 4.1 because it lends insight into which frequency bands the transform coefficients represent. Figure 4.3 shows the analysis filter in polyphase form. This notation is equivalent to the diagram in Figure 4.4. To understand the step from Figure 4.4 to Figure 4.6, one must use the "First Noble Identity" depicted in Figure 4.5 [19]. So Figure 4.7 finally shows the analysis bank in standard form, in terms of the polyphase filters. This relation can be summarized in (4.4). Similarly, the relation for the synthesis bank can be written as in (4.5).

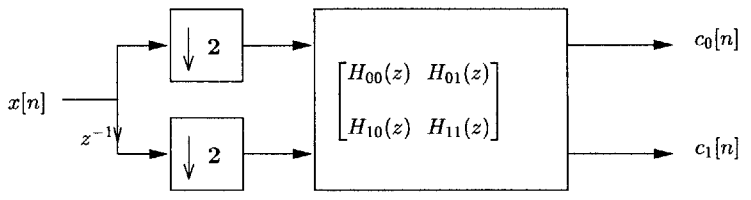


Figure 4.3: **Polyphase Analysis Bank**

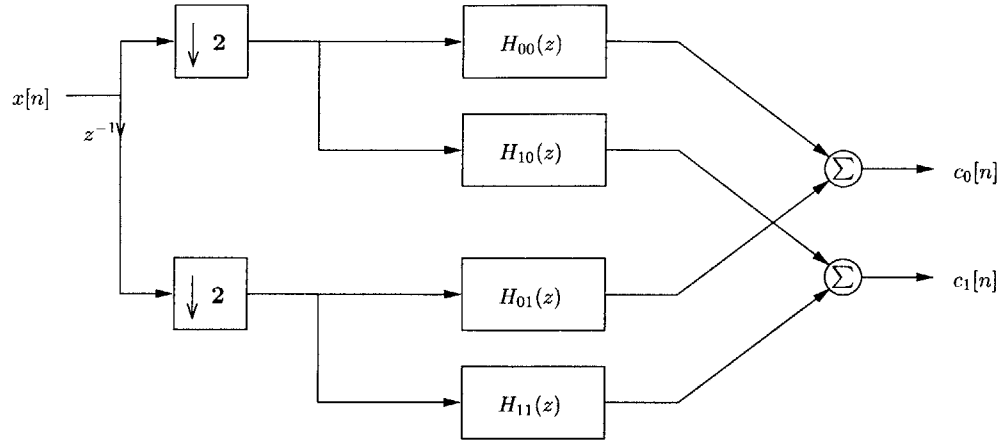


Figure 4.4: **Polyphase Analysis Bank (Expanded)**

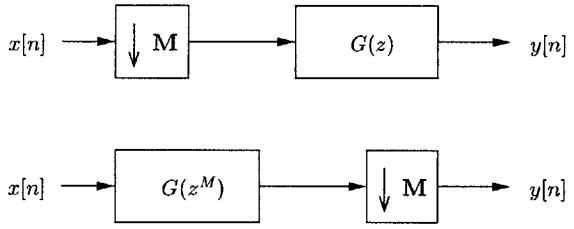


Figure 4.5: **First Noble Identity in Block Diagram Form**

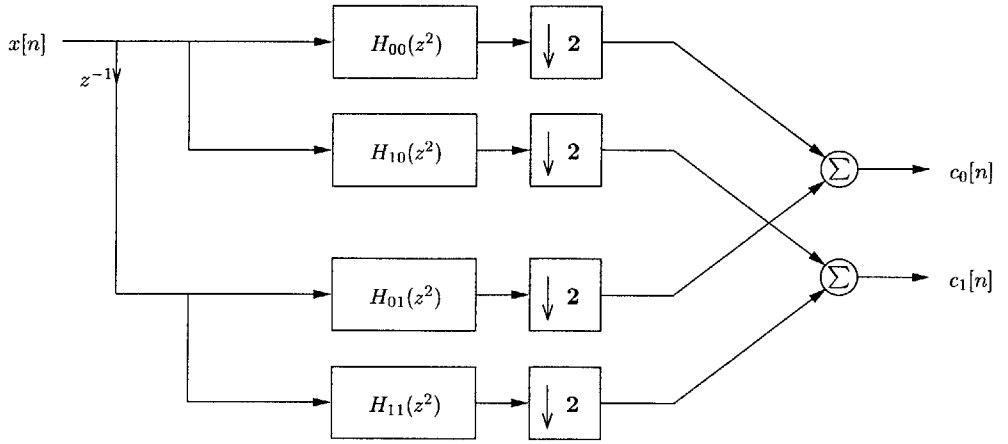


Figure 4.6: **Analysis Bank (Intermediary Form)**

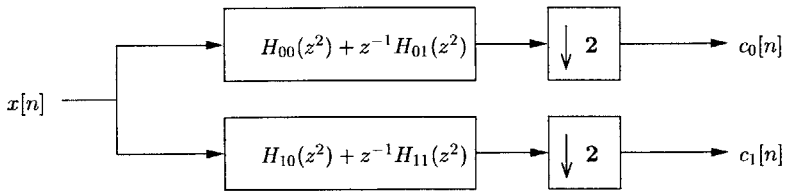


Figure 4.7: **Analysis Bank (Standard Form)**

$$\begin{bmatrix} H_0(z) \\ H_1(z) \end{bmatrix} = \begin{bmatrix} H_{00}(z^2) & H_{01}(z^2) \\ H_{10}(z^2) & H_{11}(z^2) \end{bmatrix} \begin{bmatrix} 1 \\ z^{-1} \end{bmatrix} \tag{4.4}$$

$$\begin{bmatrix} F_0(z) \\ F_1(z) \end{bmatrix} = \begin{bmatrix} F_{00}(z^2) & F_{10}(z^2) \\ F_{01}(z^2) & F_{11}(z^2) \end{bmatrix} \begin{bmatrix} z^{-1} \\ 1 \end{bmatrix} \tag{4.5}$$

With the above relations ((4.4) and (4.5)), we can create a standard filter bank given a transfer function (polyphase) matrix and its inverse. Only some analysis/inverse pairs are useful, however. The ones that are useful are usually those that produce low and high pass filters, since low and high pass filters can serve to transform into the joint time-frequency domain.

In the case of the Haar polyphase matrix, we can use relations (4.4) and (4.5) to find that the corresponding standard filters are $H_0(z) = \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}z^{-1}$ and $H_1(z) = \frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}}z^{-1}$. So we see that the Haar bank does consist of a low and a high pass filter.

4.1.2 Determining Basis Functions

Just as it is meaningful to study the filters used to convert into the transform domain, it is important to know the functions that the transform coefficients represent, i.e., the basis functions. These can be calculated by eigen methods, or alternatively by simply taking impulse responses of the synthesis system. All the transforms that we have considered have been linear (perhaps not time-invariant), so the concept of linear basis functions applies.

In the case of the Haar bank in the above example, all that's needed to determine a basis function is to reconstruct a signal $\hat{x}[n]$ from only one transform coefficient channel ($c_0[n] = \delta[n]$ for example). In the case of a two channel bank such as Haar, the two basis functions (one for each channel) are found by running $\delta[n]$ through an upsampler and then through either $F_0(z)$ or $F_1(z)$.

The remaining $N-2$ basis functions are found similarly using $\delta[n-1]$ in place of $\delta[n]$. In a two channel filter bank shown in Figure 4.1, a shift of 1 in the transform domain corresponds to a shift of 2 in the time domain. So the basis functions are composed of only two distinct shapes, repeated and shifted by two from one another.

This concept of synthesizing impulses to create basis functions proves valuable. The Haar basis function shapes are $V_0(z) = \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}z^{-1}$ and $V_1(z) = -\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}z^{-1}$. Figure 4.8 shows the construction of basis functions by this method for a four channel system. It's interesting that every function can be represented as a sum of these four functions and their shifts by four. By this same construction the functions of Figure 4.9 are found. It is these basis functions that are referred to as the "wavelets".

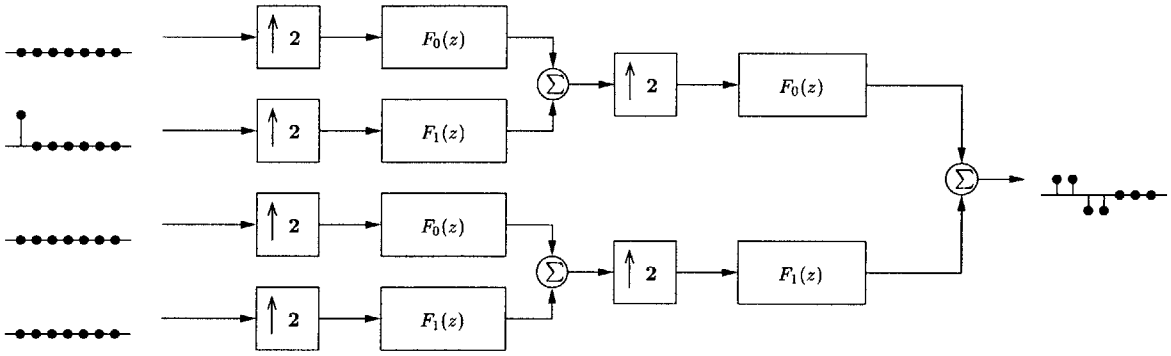


Figure 4.8: **Construction of a Single Basis Function** Here we construct a basis function by passing one impulse through a reconstruction bank. Four different *shapes* will be produced, along with their respective shifted versions as seen in Figure 4.9

4.2 Conjugate Transpose

This section will examine the condition for orthonormality among the basis functions. Orthonormality imposes conditions on the polyphase matrix: it's inverse must be its conjugate transpose. Another way to arrive at the basis functions is by synthesizing impulse responses as above but in the *polyphase* form. This takes the form of a product of the polyphase matrix and a vector of the form {1 0 0 ...}, {0 1 0 ...}, etc. These multiplications have the effect of isolating columns of the polyphase matrix. The requirements for orthonormality, given in Chapter 2 are that the basis functions must have no correlation with one another, and they must have unity correlation with themselves.

It is just these requirements that are met by matrices that have their conjugate transpose as their inverse. This can be seen readily by multiplying such matrices by hand. Since they are

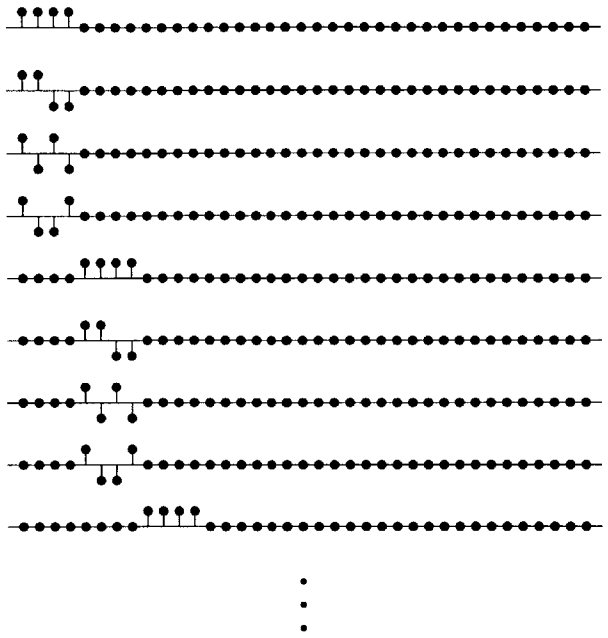


Figure 4.9: **Basis Functions of System in Figure 4.8** Notice that there are only four distinct basis function shapes. The total number of basis functions will be equal to the length of the decomposed signal.

inverses of one another, their product is the identity,

$$\begin{bmatrix} H_{00}(-z) & H_{10}(-z) \\ H_{01}(-z) & H_{11}(-z) \end{bmatrix} \begin{bmatrix} H_{00}(z) & H_{01}(z) \\ H_{10}(z) & H_{11}(z) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{4.6}$$

The off-diagonal zeros are formed by products like $H_{00}(-z)H_{01}(z) + H_{10}(-z)H_{11}(z) = 0$. Expanding this expression in the n domain reveals that it is the correlation function we need to to satisfy the *orthogonal* part of "orthonormal." The diagonal 1 terms are the *normal* part of "orthonormal." This can easily be verified by hand calculation.

Orthogonality implies that inverse of the polyphase matrix must be the conjugate transpose. This means that if one matrix is causal, the other must necessarily be anti-causal. In order to circumvent this problem, we are forced to delay one of the matrices and suffer an overall delay in the system equal to l, the length of the longest filter. This satisfies the definition of "perfect reconstruction" as stated in (4.1), but is not desirable. Later chapters will address possible solutions to this unfortunate reality, by employing non-orthogonal transforms.

Chapter 5

Near Real-Time Processing

This chapter along with the next chapters describe the original work done for this thesis. We place our attention on minimizing the delay associated with the types of transforms spoken of in earlier chapters. We seek to understand what methods of implementation will result in small delays, and what limitations different types of transforms possess.

5.1 Motivation for Real-Time

In the signal processing world, faster is always better. "Real-time" is a special case of faster because it implies that the process not only is computationally efficient, but results in an output signal that has minimal shift delay (preferably no shift delay whatsoever). This is quite an order for most systems, especially when a complex operation is desired such as denoising audio signals. In this thesis there is a tradeoff between system performance (perceived quality of processed audio) and delay.

In the audio world it is advantageous to have systems that are capable of processing in real-time. Recording engineers require the flexibility to listen to a recording with and without a given process on the fly. Moreover, an artist who wishes to perform live is only able to use equipment that can itself perform live in real-time. Noise is a central problem encountered in both recording and performing. Conventional yet sophisticated denoising apparatae that are widely used in live audio are really operating only from the time domain. They are comprised of a gate and a filter that are controlled (sometimes in clever ways) by the signal level. These systems are indeed effective, but we are interested in the possibility of increased effectiveness through transforming quickly to the time-frequency domain, and doing the gating there. Perfect reconstruction wavelet filter banks can be designed to be computationally efficient, and nearly causal which is ideal for this goal.

5.2 Measures of Performance

As to be consistent with the audio engineering literature, the primary performance measure of denoising must be perceptual [8]. Other more quantitative measures such as signal to noise ratio (SNR) are not fully representative of what sounds good to a human. When a recording has noise on it, the SNR can generally be increased by lowpass filtering the signal since most of the signal is in the lower part of the spectrum. But many have experienced the frustration of turning the treble down on a stereo system in an effort to make noisy music sound clearer. The ear is not fooled by such a trick. Certain high frequency attacks of notes (impulses) become smeared when lowpassed, and the ear realizes this and adjusts its perception. In the mind, there is still the same amount of noise present, even though the SNR has been increased.

This is the reason for transforming into the time-frequency domain. The impulses in the music can be preserved. Perhaps there is an optimal time-frequency domain with respect to which the ear perceieves the best [8]. In any case, the optimal domain to operate from should be called the "musical" domain. The musical signals themselves have been created by humans for humans. The domain that we work in should be a balance between what is found mathematically and what is known about humans and what they're expecting to hear in a clean piece of music.

Because we haven't spent a great deal of time invested in designing optimal filter banks with respect to perception (we've chosen to focus mainly on implementation issues common to all such filter bank systems), we certainly have not performed any A-B testing with human subjects. We have discovered that terrific performance is achieved when the basis functions have lengths on the order of 7000 samples in a 44.1 kHz sampled signal. This corresponds to about 0.16 seconds in time.

This figure represents a time/frequency balance where the minimal time resolution is 0.16 seconds and the maximal frequency resolution is about $\frac{1}{7000T} = 6.3$ Hz. Basis functions of this length can be achieved in two ways, assuming a system of the form shown in Figure 3.5. Long filters can be placed as $H_0(z)$, $H_1(z)$, $F_0(z)$, and $F_1(z)$, or more stages can be added to the tree. Both methods yield good results, although different basis functions are realized. The most important requirement for satisfactory denoising performance is that the basis functions occupy space in an appropriate time-frequency plane specified only by the length of the basis function. This length determines the approximate time/frequency balance which by far outweighs any other parameter (like basis function shape) in importance.

This is not to say that the basis function shapes are not at all important. It is implied if the basis functions have been created by a lowpass/highpass filter bank structure such as depicted

in Figure 3.5 that the functions will be localized in *both* time and frequency. This means that the functions actually will represent a time-frequency domain. One could think of many functions that are localized in time but not in frequency such as a very high frequency oscillation added to a very slow one. A signal like this might not occupy a single localized region in frequency, but two split regions, one high and one low. So we are implicitly prioritizing that the shape of the basis functions be appropriate. This naturally happens in lowpass/highpass filter banks.

5.3 Computational Delays

As this thesis is targeted at real-time implementation, we are naturally concerned about computational delays associated with the process. The group delay can be found theoretically. It is independent of system hardware, and is the delay that would occur if the hardware were infinitely fast. We have discovered that although there are limitations on this delay (particularly if we choose to work only with orthogonal filter banks), the computational delay proves to be an even harder challenge. The hardware we used progressed from a Matlab simulation to a Matlab/C hybrid language, to purely C compiled on a stand-alone Linux system.

Implementing in C is much faster than working in Matlab. But still it is not fast enough to keep up with the quickly sampled music. It is good in retrospect that this is the case, because it forced us to examine our computation methods very closely. Even though computers are becoming ever increasingly fast, the advances spur on more drive for even more sophisticated signal processing. It is naive to think that more computational power will solve signal processing problems.

Even the simplest of processes take on different forms when one tries to fully implement them. Take for instance convolution, which is represented by the $*$ symbol. There are multitudes of methods of implementing this[1] [5], even though it is only a single concept in the abstract world of signals and systems. Each method has advantages and liabilities. Such is the case with our large tree structure. The next chapter, which is at the heart of the thesis, will explain our findings with respect to this implementation, and will generalize to a variety of other systems.

[1] Overlap/Add, Overlap/Save, etc., etc.

Chapter 6

Implementation of Wavelet Filter Bank Tree

This chapter examines the heart of the thesis: how best to implement wavelet filter bank systems with the intent of using them in near real-time. The goal of the thesis is to find a system that can be implementable in real-time, however this chapter deals primarily with finding the most efficient method of computation. This was found to be an equally, if not more important concern in the design of a system for operation in real-time.

Four different methods are considered and their associated computational costs are compared, using the required number of multiplies as a measure of computational efficiency. None of the methods implemented on our 200 MHz PC could keep up with the sampling rate of the music. It is natural to attempt to overcome this problem before diving heavily into designing a system with zero group delay, although the next chapter recounts our efforts to do just that in spite of significant computational delay.

The following are methods of implementing transforms like the one represented in Figure 3.5, and its associated synthesis transform.

6.1 Minimal Delay Expected

A system as in Figure 3.5 has with it a certain minimal number of time steps that are necessary before one output sample is generated. This is what we will refer to as the minimal delay. Whereas the output of a linear filter depends on the current sample and previous samples, the transform coefficients in filter bank systems depend on the current *block* of M = (number of channels) samples, and past blocks. This is caused by the inclusion of the downsamplers. So for any given tree system, the minimal delay is equal to the number of channels. This delay is unavoidable, but small in comparison to other delays.

6.2 Non-Causal Approach

The most direct method is implemented by taking the whole signal and running it through each filter separately, downsampling, running the entire result through the next stage of filters, and so on. This is clearly not an option for us since it requires the entire signal in advance. It is non-causal, but might work theoretically for signals that are to be post-processed, such as previously recorded music. Even if we had access to the whole signal before processing, this idea has the same major pitfalls that convolving enormous signals has. Unless the signal of interest is short, it is unwieldy to work with in this way, due to memory constraints.

It is important to study the number of multiplications that are associated with this direct method, however, because it lends insight into the cost or benefit of computing the same result in another fashion. We did not actually implement this system for the reasons stated above (memory), but the number of multiplications can nevertheless be computed.

The specific transform that was studied in this thesis is a 128 channel filter tree with length 30 filters. Assuming an input signal of $10,000$ samples, we can calculate the number multiplications this direct method would require.

A convolution operation between two signals, of lengths m and n, requires $m \times n$ multiplications. Convolution followed by downsampling can be produced in $\frac{m \times n}{2}$ multiplications, since it is not necessary to calculate the samples that will be discarded. The length of a convolved signal becomes $m + n - 1$. A signal of length n becomes a signal of length $\frac{n}{2}$ or if odd $\frac{n+1}{2}$ after downsampling. Also, upsampling a signal of length n results in a signal of length $2n - 1$. Upsampling followed by filtering requires $\frac{(2n-1)m}{2}$ multiplications. From these facts, we can derive the number of multiplications required for all of our implementations.

The number of multiplications that the direct method requires can be found by constructing Table 6.1. Similarly, on the synthesis side, we can calculate the number of multiplies that would be required by constructing Table 6.2. The total number of multiplies associated with this process is the sum of the two totals: $421,438,980$. As a rule of thumb, the number of multiplies for a system like this, assuming a long input signal relative to the length of the filters l, and number of stages r, will be on the order of $2Nlr$, where N is the length of the input signal.

Stage	Channels	Length of Signal After Filtering/Downsampling	Multiplies/ Channel	Multiplies	Total
0a	1	1000000	0	0	0
1a	2	500015	15000000	30000000	30000000
2a	4	250022	7500225	30000900	60000900
3a	8	125026	3750330	30002640	90003540
4a	16	62528	1875390	30006240	120009780
5a	32	31279	937920	30013440	150023220
6a	64	15654	469185	30027840	180051060
7a	128	7842	234810	30055680	210106740

Table 6.1: **Multiplications for the Analysis Tree** This filter bank has filters of length 30. Note that at each stage, there are more channels, but the signal lengths become shorter, resulting in an almost linear relationship between number of stages and number of multiplies.

Stage	Channels	Length of Signal After Upsampling/Filtering	Multiplies/ Channel	Multiplies	Total
0s	128	7842	0	0	0
1s	64	15712	235260	30113280	30113280
2s	32	31452	471360	30167040	60280320
3s	16	62932	943560	30193920	90474240
4s	8	125892	1887960	30207360	120681600
5s	4	251812	3776760	30214080	150895680
6s	2	503652	7554360	30217440	181113120
7s	1	1007332	15109560	30219120	211332240

Table 6.2: **Multiplications for the Synthesis Tree** Notice that there are slightly more multiplies associated with reconstructing the signal than with analyzing it. Also notice that the output signal has been elongated by the approximate length of the basis functions ($\sim$ 7332). To conserve perfectly the length of the signal throughout the transform, a circular transform should be taken.

6.3 Non-Time Invariant Impulse Method

The following method yields a simple way of organizing the somewhat complicated looking tree structure in Figure 3.5. By realizing that each input sample is not treated identically by the system (because the downsamplers treat even and odd samples differently), the analysis bank cannot be characterized by a single impulse response as in a typical linear time-invariant system. The system is linear *block* invariant. Within each block of 128 samples, 128×128 separate impulse responses from input to transform domain are needed to completely define the analysis system. Similarly, the 128 basis function shapes represent the impulse responses of the synthesis bank.

We can rewrite the analysis system as a matrix multiplication, where there are 128 input

sequences (one for each phase), and 128 output sequences corresponding to the transform coefficients:

$$\begin{bmatrix} C_0(z) \\ C_1(z) \\ C_2(z) \\ . \\ . \\ . \\ C_{127}(z) \end{bmatrix} = \begin{bmatrix} A_{0,0}(z) & A_{0,1}(z) & A_{0,2}(z) & . & . & . & A_{0,127}(z) \\ A_{1,0}(z) & A_{1,1}(z) & A_{1,2}(z) & . & . & . & A_{1,127}(z) \\ A_{2,0}(z) & A_{2,1}(z) & A_{2,2}(z) & . & . & . & A_{2,127}(z) \\ . & . & . & . & & & . \\ . & . & . & & . & & . \\ . & . & . & & & . & . \\ A_{127,0}(z) & A_{127,1}(z) & A_{127,2}(z) & . & . & . & A_{127,127}(z) \end{bmatrix} \begin{bmatrix} X_0(z) \\ X_1(z) \\ X_2(z) \\ . \\ . \\ . \\ X_{127}(z) \end{bmatrix}. \tag{6.1}$$

Each of the entries in the A matrix in (6.1) is an impulse response from one of the input phases to one of the transform sequences. This representation is conceptually easy to implement in C, and helped us to see the system more intuitively. Although in the end, it is not as computationally efficient as using a recursive tree, it provided us with ideas about how to possibly overcome the delay associated with orthogonal filter banks (see Section 4.2).

For our filter length 30, 128 channel system, the impulse responses in (6.1) are each of length 29. The length of these responses are always on the order of the length of the filters $H_0(z)$, $H_1(z)$, $F_0(z)$, and $F_1(z)$. This can be shown easily using the convolution/downsampling rules outlined above. The impulse responses of the synthesis bank are of course equal to the length of the basis functions, which in this case is ~ 7400.

To calculate the number of multiplications required to process an input signal of length 1000000, we simply compute:

$$128 \times 128 \times 29 \times \frac{1000000}{128} + 7400 \times \frac{1000000}{128} = 3769812500 \tag{6.2}$$

This is significantly more multiplies than implementing using the tree structure. This is one of the major reasons that wavelet *trees* have gained so much attention - because they are efficient. We see a gain of about 9 in number of multiplies for this example. So we must search for an implementation that possesses the same computational benefits of the tree, but does not require the entire input signal in advance. The next section explains our findings.

6.4 Large Buffers at Each Stage

Here we present a possible solution to the dilemma discovered above. This section and the next involve subtle, yet important differences in implementation of the tree structure. Not only is the system of (6.1) relatively inefficient in terms of multiplies, but it requires precalculation of 128×128 impulse responses to be stored somewhere within the system's memory. This proved to be very difficult, even on a PC. The tree structure is quite elegant and only after working to implement the system in a number of ways did we fully appreciate its compactness and efficiency.

In terms of programming in C, implementing a near real-time tree is not transparent. The method described in this section makes use of large buffers at each splitting branch of the tree. Since 128 samples are needed before one transform coefficient is computed, it seemed to make sense to receive blocks of 128 samples at a time and at that time compute the next 128 transform coefficients (dependent on the current block and previous blocks). So each buffer holds information left over from previous blocks of data, to be added appropriately to the current data. In the end, 128 new coefficients are computed (along with unfinished data to be added in the future), and the next block of input samples can begin processing.

In the implementation of Section 6.3 (using a large matrix of impulse responses), there was a fair amount of precalculation, but the advantages of the tree structure were lost. Using large buffers at each stage is the worst of all, because it is essentially equivalent to the impulse method described in Section 6.3, but without precalculation. This is why it bears such a high computational cost. The next section will describe the preferred method, which preserves the benefit of the tree structure seen witnessed with the system described above in Section 6.2, and also operates block causally.

The required multiplications for this naive block causal tree implementation can be computed with the aid of Table 6.3:

The synthesis bank yields similarly inefficient values. We turn now to the best of both worlds, which is a block causal system that also maintains the efficiency promised by the recursive tree structure.

6.5 Just In Time Multiplication

Although there could very well be other better ways to implement such systems, we found what we believe to be the optimal method of implementation of systems like the one in Figure 3.5. The

Stage	Channels	Length of Signal After Filtering/Downsampling	Multiplies/Channel	Multiplies	Total
0a	1	1	0	0	0
1a	2	15	15	30	30
2a	4	22	225	900	930
3a	8	26	300	2640	3570
4a	16	28	390	6240	9810
5a	32	29	420	13440	23250
6a	64	29	435	27840	51090
7a	128	29	435	55680	106770
$\times 1e6$					$1.0677e11$

Table 6.3: **Multiplications for the Naive Block Causal Analysis Tree Implementation** The number of multiplies associated with this transform is extremely high because nothing is precalculated, and multiplications occur as far upstream as possible.

rule of thumb in this approach is to only calculate the samples that are *needed* for the next stage, and leave remaining samples in buffers to be processed for the next input block. This has the effect of producing only 128 samples of output at a time (without leftover samples to be added in the future). So in order to produce one block of transform coefficients, 128 input samples are needed at the first stage. Similarly, if we have full buffers at the second stage, we only need 64 samples from each channel to eventually produce the one block of transform coefficients. This continues, until of course the seventh stage, when all that is required from the previous stage is 2 samples per channel.

This produces an algorithm in which multiplications occur only between fully added sample points. In other words, multiplication is not done on sample groups that will eventually be added together in the end. In a sense the algorithm combines all like terms before multiplying them, instead of multiplying a bunch of individual like terms. It is a subtle realization that makes a big difference.

The computational cost of this method is identical to that of the non-causal approach. The only disadvantage of this method is that it requires careful and complex programming. But this is worth it in the end. A copy of the C code is included in the Appendix for reference.

The program works nicely, although it is still much too slow to keep up with the music. Figure 3.2 was made using data created by this program. This is the wisdom that came from this programming experience, although there are other programming issues that are interesting, but less relevant to the thesis. The concept of always multiplying at the last minute can be generalized to many programming and digital signal processing situations.

Chapter 7

Search for Zero Group Delay

This chapter outlines our efforts to overcome the other significant delay in the system unavoidably produced by orthogonality. Recalling from Chapter 4 that orthogonal transforms possess an unavoidable group delay equal to the length of the longest basis function , we seek an alternative that would produce a delay equal only to the minimal delay imposed by the block convolution of the transform. In this chapter we are concerned with the theoretical delay that would occur assuming infinite computing power. None of the results of the previous chapter have any bearing on the delay spoken of in this chapter.

We have only begun the search, and have discovered that while many researchers are concerned with the elegance of orthonormal transforms, other transforms can be valuable as well. We will introduce the concept of **unimodular** matrices.

7.1 Unimodularity

An orthogonal transform possesses a certain type of polyphase matrix whose inverse (used in synthesis) is its conjugate transpose. This fact makes a causal analysis matrix necessarily possess an anti-causal inverse. To be used in a completely causal system, this in turn necessitates a delay equal to the length of the filters included in the polyphase matrices. Such matrices associated with orthogonal transforms are called **paraunitary**.

Unimodular matrices have different properties at the expense of losing orthogonality. All causal FIR unimodular matrices have causal FIR inverses. This is exactly the property we desire for a system with a small delay. The challenge is to design such matrices.

7.2 Problems in Design

The important property that these matrices must be designed to have is that they must transform to a useful domain. The domain of interest to us is the joint time-frequency domain. This implies

that the basis functions must be localized in both time and frequency. It is not at all obvious how to design to achieve this goal. Assuming it were possible, the system would be able to transform to this domain using only past sample values, and also able to reconstruct causally after deciding which samples to keep and which to discard. This would be a very valuable system.

There are simple rules for constructing unimodular matrices, but there are no guarantees that the matrices created will be useful. The Haar transform is in fact both unimodular and paraunitary, but it is a special case because it is a scalar matrix (as opposed to a transfer function matrix). A system that incorporates unimodular polyphase matrices would have a total shift delay equal to (not greater than) the number of channels (frequency bands) desired.

Take for example the unimodular polyphase pair shown below:

$$\begin{bmatrix} 1 & -R_1(z) \\ -R_0(z) & R_1(z)R_0(z)+1 \end{bmatrix} \begin{bmatrix} R_1(z)R_0(z)+1 & R_1(z) \\ R_0(z) & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{7.1}$$

If $R_0(z)$ and $R_1(z)$ are FIR and causal, then these matrices could be used in a larger tree system with no neccessity for delay. Figure 7.1 shows the output of two systems, one unimodular and the other paraunitary. The shift delay in the unimodular system is not dependent on basis function length as it is in the paraunitary case. Both of these plots were generated with five stages. We see that the unimodular system only suffers a delay of $2^5 = 32$ due to the block invariance of the tree. This delay is independent of the choice of filters $R_0(z)$ and $R_1(z)$. The output of the paraunitary system is unfortunately delayed by the length of its basis functions.

Appendix B contains the filters used to create these plots.

Our first insight into the possibility of unimodular matrices came as we worked with the impulse response method of the previous chapter. Looking at the system as represented in (6.1), we see that it is nothing more than a large matrix multiplication. If the A matrix of (6.1) were unimodular, we would have the desired system [20].

This is an excellent topic for further research. Some research has been done on the topic already, but for other reasons. If a unimodular matrix has transfer functions with only integers, its inverse will also only possess integers [2]. This is because the thing that makes the unimodular matrix special is that the determinant (seen in the denominator of the inverse) is a constant, and for matrices that only involve integers, the determinant is an integer. There are many mysterious challenges awaiting. There are numerous other tree topologies, including those involving $\downarrow 3$ operators,

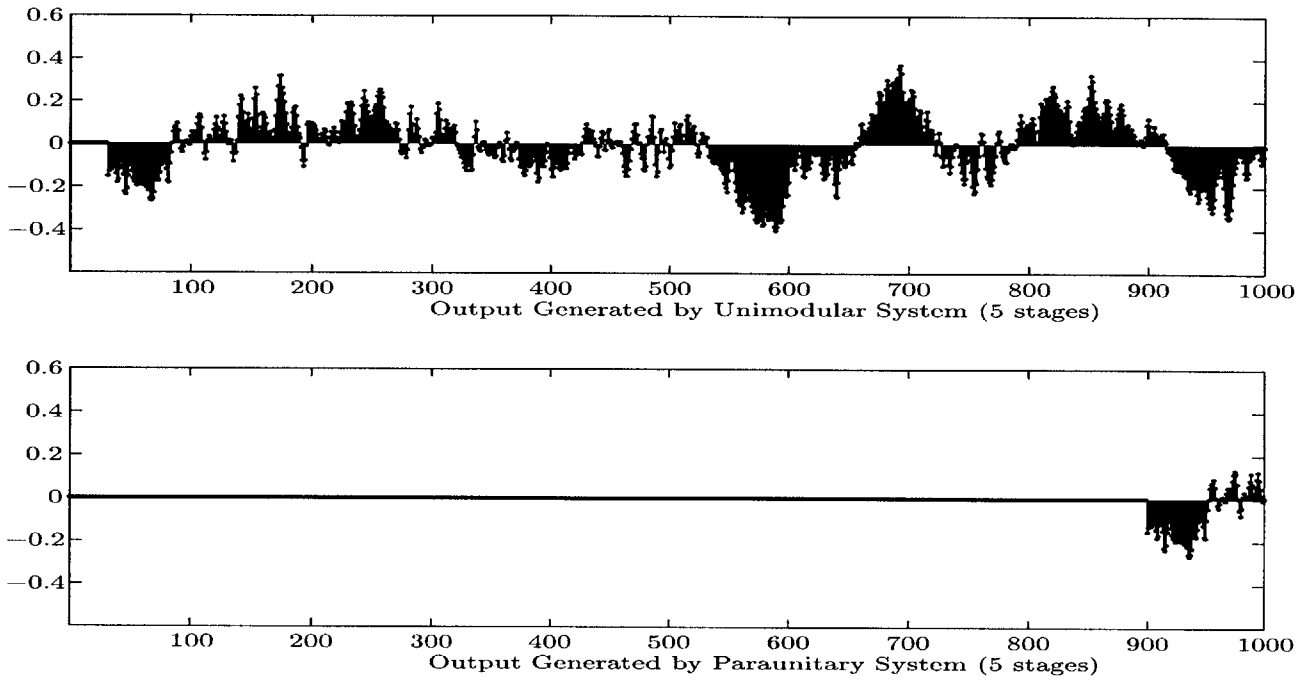


Figure 7.1: **Output Sequences for Paraunitary and Unimodular Systems** We see that the unimodular system has the striking advantage of possessing very little shift delay. The challenge, however, is to design the filters to be *useful*

$\downarrow 4$ operators, or block downsampling [12], etc., all with the possibility of efficient implementation, near zero group delay, and/or other interesting properties.

Chapter 8

Concluding Remarks

This thesis provides a great foundation for further research. The topic of wavelets is subtle, and there is a lot of room in the field for creativity in the design of systems like the one we studied. In this thesis we implemented a large tree structure in the fastest way we could find. This involved looking at different types of delay, and working with each type separately.

8.1 Block Delay

The block delay is the delay associated with the fact that a wavelet tree system can be at best block causal. That is to say, there is no avoiding a delay equal to the number of channels. This can be seen readily in the representation in (6.1). In order to produce one transform coefficient (actually 128 at a time), since this is a matrix multiplication, we need to know at least the first 128 input samples (one for each phase). Although (6.1) yields an inefficient implementation, it lends insight into the system and is an extremely valuable way of viewing the system. We did implement a small version of this type of system before discovering the final method of Chapter 6.

8.2 Computational Delay

Much progress was made in this area. We started with little insight into the system other than the block diagram representation which does not reveal how best to implement itself. After discovering other representations such as the "impulse" method in (6.1), we were able to overcome the non-causal implementation directly implied by the block diagram as seen in Figure 3.5. Knowing that there existed block causal implementations, we sought to find the one that matched the direct non-causal method in terms of computational complexity as quantified by the number of multiplies required for processing.

We conclude that a method which performs multiplications as far downstream as possible produces an efficient implementation, since it naturally combines like terms before requiring a costly

multiply. This result can certainly be applied to other signal processing and programming systems.

8.3 Delay Associated with Orthogonal Systems

Since orthogonal systems are composed of paraunitary matrices (the polyphase matrices are paraunitary as well as the A matrix in (6.1)), they suffer from the unfortunate fact that the inverse matrices are necessarily anti-causal. This results in a delay equal to the length of the filters $H_0(z)$, $H_1(z)$, $F_0(z)$, and $F_1(z)$, in order to force the inverse to be causal. This constraint can only be bypassed by sacrificing the nice properties of orthogonal transforms (energy preserving, etc.). We found that if unimodular matrices are used in place of these paraunitary matrices, we could achieve zero delay where there once was a delay equal to the length of the filters.

The design of such systems is not straightforward, because we wish the unimodular matrices to still possess the property that their eigenvectors (the basis functions) be localized in time and frequency. This would produce a transform that has the potential of being used to transform into the time-frequency domain and used for the analysis of music and other natural phenomena. Although it is easy to find unimodular matrices, we could not find one that has the above property with the exception of the Haar matrix, which is a degenerate case.

8.4 Future Work

Because of the advances in computing power and interest in signal processing for all sorts of applications from the internet to music processing to control, there is much research being done in this area. There are a lot of areas to explore. We dealt with only one type of system - a wavelet tree with $\downarrow 2$ operators. There are many other types of systems that could be devised that generalize this basic form. For example $\downarrow 3$ operators could be used, or block downsamplers could be used [12].

Another fascinating area which is tied very closely to the work in this thesis is the study of how the ear hears. The ear hears in a complex, non-linear way and its characteristics influence the performance of any audio denoising system. There has been much work involved in audio coding for compression, like MP3, which compresses raw audio by a factor of ten without any perceptual loss in audio quality. Schemes like MP3 take into account which elements the ear is sensitive to, and which it isn't. It is quite an achievement.

Appendix A

Program Code (C)

A.1 Zipper.h

```
int getsound(float *x, char *insoundfile);
int playsnd(unsigned char *y, long int q);
int filesnd(char *argv[], float *h, int q);
int sndconvert(unsigned char *ychar, float *y, int vectorsize);
int getoutimpulse(float *w, char *p);
int getinimpulse(float *w, char *p);
int GlobalAssign(int numbands, int *inlengthmax, int *outlengthmax);
int Threshold(float *f, float thr, int numbands);
int OutProcess(float receive[], float send[], int numbands, int outlength);
int Process(float *h, float *y, const float *x, float thr, long int q, int numbands, int inlength, int outlength);
int main(int argc, char *argv[]);
int rndsnd(float *r, long int q);
float ran1(long *idum);
void reverse(char s[]);
int trim(char s[]);
void itoa(int n, char s[]);
void PrintTime();
int convolvedown2(float *out, float *s1, float *s2, float ls1, float ls2);
int convolveup2(float *out, float *s, float *feven, float *fodd, float ls, float lfeven, float lfodd);
float *convolve(float *ss1, float *ss2, float ls1, float ls2);
int TreeFiltAssign(float *filt, char *filename);
int TreeProcess(float *h, float *y, float *x, float thr, long int q, int levels, int flth, float *h0, int h0q, float *h1, int h1q, flc
float ***MakeInputDataBuffers(int *datachunk, int flth, int levels);
float ***MakeOutputDataBuffers(int *sdatachunk, int flth, int levels);
int shift(float *buf, int points, int extent);
int add(float *buf1, float *buf2, int extent);
int AssignToH(float ***stg, float *h, int levels);
int GetEvenOdd(float *feven, float *fodd, float *f, int fq);
int FindHalfLength(int *fevenq, int *foddq, int fq);
int ZeroOut(float *buf, int extent);
float Energy(float *sig, int q);
int PrintVector(char *message, float *vec, int vecq);
int MakeOnes(float *x, int q);
```

A.2 Muixtree.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <assert.h>
#include "zipper.h"

int main(int argc, char *argv[])
{
  float thr;
  float *x, *y, *h, *r, *h0, *h1, *f0, *f1;
  unsigned char *ychar;
  int i, q, h0q, h1q, f0q, f1q, flth, levels;

  if(argc != 9){
    printf("usage:  muixtree soundfilename outpufilename threshold levels H0filename H1filename F0filename F1fi
  return(1);
  }
  levels = atoi(argv[4]);
  /* ---------------------------------------- */
  PrintTime();
  printf("Allocating x...\n");
  x = calloc(5000000, sizeof(float));
  assert(x != NULL);
  /* ---------------------------------------- */

  printf("Loading sound into variable x...\n");
  q = getsound(x, argv[1]);
  realloc(x, q*sizeof(float));
  thr = atof(argv[3]);
  /* ---------------------------------------- */

  printf("Loading filters...\n");
  h0 = calloc(100, sizeof(float));
  h0q = TreeFiltAssign(h0, argv[5]);
```

```
h0 = realloc(h0, h0q*sizeof(float));
h1 = calloc(100, sizeof(float));
h1q = TreeFiltAssign(h1, argv[6]);
h1 = realloc(h1, h1q*sizeof(float));
f0 = calloc(100, sizeof(float));
f0q = TreeFiltAssign(f0, argv[7]);
f0 = realloc(f0, f0q*sizeof(float));
f1 = calloc(100, sizeof(float));
f1q = TreeFiltAssign(f1, argv[8]);
f1 = realloc(f1, f1q*sizeof(float));
flth = h0q;
printf("levels:  %d\n", levels);
assert(h0q == f1q);
/* ------------------------------------------- */

printf("Allocating variables...\n");
y = calloc(q, sizeof(float));
h = calloc(q, sizeof(float));
ychar = calloc(q, sizeof(unsigned char));
r = calloc(q, sizeof(float));
/* ------------------------------------------- */

printf("Adding noise...\n");
rndsnd(r, q);
for(i=0; i<q; ++i){
  x[i] = x[i] +r[i]/8;
  }
/* ------------------------------------------- */
PrintTime();
printf("TreeProcess started\n");
TreeProcess(h, y, x, thr, q, levels, flth, h0, h0q, h1, h1q, f0, f0q, f1, f1q);
printf("TreeProcess ended\n");
/* ------------------------------------------- */
PrintTime();
printf("Converting data to unsigned char...\n");
sndconvert(ychar, y, q);
/* ------------------------------------------- */
PrintTime();
printf("Sending data to file...\n");
filesnd(argv, x, q);
/* ------------------------------------------- */
PrintTime();
printf("Calculating energies...\n");
printf("Energy in x = %f\n", Energy(x, q));
printf("Energy in h = %f\n", Energy(h, q));
```

```
  printf("Energy in y = %f\n", Energy(y, q));
  /* ---------------------------------------- */
  PrintTime();
  printf("Playing sound...\n");
  playsnd(ychar, q);
  /* ---------------------------------------- */
  PrintTime();
  free(x);
  free(y);                                                    90
  free(h);
  free(ychar);
  return(0);
}

void PrintTime()
{
  char s[20];
  time_t tm;
                                                              100
  tm = time(NULL);
  strftime(s, 20, "%M:%S", localtime(&tm));
  printf("%s\n",s);
}

int Threshold(float *f, float thr, int levels)
{
  int i;
  for(i=0; i<pow(2, levels); i++) {
    if( fabs( f[i]) < thr){                                   110
      f[i]=0;
    }
  }
  return(0);
}

int TreeFiltAssign(float *filt, char *filename)
{
  FILE *fp;
  int i;                                                      120

  fp = fopen(filename, "r");
  assert(fp != NULL);
  fseek(fp, 0, SEEK_SET);
  for(i=0; feof(fp) < 1; ++i){
    fscanf(fp, "%f", &filt[i]);
```

```
 }
 fclose(fp);
 return(i);
}

int TreeProcess(float *h, float *y, float *x, float thr, long int q, int levels, int filh, float *h0, int h0q, float *h1, int h1q, fl
{
 float ***astg, ***sstg;
 float *temp, *stemp0, *stemp1, *f0even, *f0odd, *f1even, *f1odd;
 int i, j, n, tempsize;
 int *adatachunk, *sdatachunk, *f0evenq, *f0oddq, *f1evenq, *f1oddq;

 tempsize = 200;
 adatachunk = calloc(levels+1, sizeof(int));
 sdatachunk = calloc(levels+1, sizeof(int));
 temp = calloc(tempsize, sizeof(float));
 stemp0 = calloc(tempsize, sizeof(float));
 stemp1 = calloc(tempsize, sizeof(float));
 f0evenq = calloc(1, sizeof(float));
 f0oddq = calloc(1, sizeof(float));
 f1evenq = calloc(1, sizeof(float));
 f1oddq = calloc(1, sizeof(float));
 FindHalfLength(f0evenq, f0oddq, f0q);
 FindHalfLength(f1evenq, f1oddq, f1q);
 f0even = calloc(f0evenq[0], sizeof(float));
 f0odd = calloc(f0oddq[0], sizeof(float));
 f1even = calloc(f1evenq[0], sizeof(float));
 f1odd = calloc(f1oddq[0], sizeof(float));
 astg = MakeInputDataBuffers(adatachunk, filh, levels);
 sstg = MakeOutputDataBuffers(sdatachunk, filh, levels);
 GetEvenOdd(f0even, f0odd, f0, f0q);
 GetEvenOdd(f1even, f1odd, f1, f1q);
 for(n=0; n<q; n=n+pow(2,levels)){
  astg[0][0] = x+n;
  for(i=1; i<levels+1; ++i){
   for(j=0; j<pow(2, i); j=j+2){
       ZeroOut(temp, tempsize);
       convolvedown2(temp, astg[i-1][j/2], h0, (float)(pow(2, levels-i+1)), (float)h0q);
       shift(astg[i][j], pow(2, levels-i), adatachunk[i]);
       add(astg[i][j], temp, adatachunk[i]);
   }
   for(j=1; j<pow(2, i); j=j+2){
       ZeroOut(temp, tempsize);
       convolvedown2(temp, astg[i-1][(int)(floor((float)j/2))], h1, (float)(pow(2, levels-i+1)), (float)h1q);
       shift(astg[i][j], pow(2, levels-i), adatachunk[i]);
```

```
        add(astg[i][j], temp, adatachunk[i]);
    }
  }
  AssignToH(astg, h+n, levels);
  Threshold(h+n, thr, levels);
  for(i=0; i<pow(2, levels); ++i){
    sstg[0][i] = h+n+i;
  }
  for(i=1; i<levels+1; ++i){                                                    180
    for(j=0; j<pow(2, levels-i); j=j+1){
        ZeroOut(stemp0, tempsize);
        ZeroOut(stemp1, tempsize);
        convolveup2(stemp0, sstg[i-1][2*j], f0even, f0odd, pow(2, i-1), (float)f0evenq[0], (float)f0oddq[0]);
        convolveup2(stemp1, sstg[i-1][2*j+1], f1even, f1odd, pow(2, i-1), (float)f1evenq[0], (float)f1oddq[0]);
        add(stemp0, stemp1, sdatachunk[i]);
        shift(sstg[i][j], pow(2, i), sdatachunk[i]);
        add(sstg[i][j], stemp0, sdatachunk[i]);
    }
  }                                                                             190
  AssignToY(sstg, y+n, levels);
 }
 free(astg);
 free(sstg);
 free(temp);
 free(stemp0);
 free(stemp1);
 free(f0even);
 free(f0odd);
 free(f1even);                                                                  200
 free(f1odd);
 free(adatachunk);
 free(sdatachunk);
 free(f0evenq);
 free(f0oddq);
 free(f1evenq);
 free(f1oddq);
}

                                                                               210
float ***MakeInputDataBuffers(int *adatachunk, int flth, int levels)
{
 int i, j, totalbufsize;
 float *master, *temp;
 float ***astg;
```

```
  totalbufsize = 0;
  adatachunk[0] = pow(2, levels);
  for(i=1; i<levels+1; ++i){
    adatachunk[i] = (int)ceil(.5*(float)(pow(2, levels-i+1) + flth - 1));
  }
  for(i=0; i<levels+1; ++i){
    totalbufsize = totalbufsize + pow(2, i)*adatachunk[i];
  }
  master = calloc(totalbufsize, sizeof(float));
  astg = calloc(levels+1, sizeof(float **));
  for(i=0; i<levels+1; ++i){
    astg[i] = calloc(pow(2, i), sizeof(float *));
    assert(astg[i] != NULL);
  }
  astg[0][0] = &master[0];
  temp = &astg[0][0][0];

  j = 1;
  for(i=1; i<levels+1; ++i){
    temp = temp + j*adatachunk[i-1];
    for(j=0; j<pow(2, i); ++j){
      astg[i][j] = temp + j*adatachunk[i];
    }
  }
  return(astg);
}

float ***MakeOutputDataBuffers(int *sdatachunk, int flth, int levels)
{
  int i, j, totalbufsize;
  float *smaster, *temp;
  float ***sstg;

  totalbufsize = 0;
  sdatachunk[0] = 1;
  for(i=1; i<levels+1; ++i){
    sdatachunk[i] = pow(2, i) - 2 + flth;
  }
  for(i=0; i<levels+1; ++i){
    totalbufsize = totalbufsize + pow(2, levels-i)*sdatachunk[i];
  }
  printf("totalbufsize = %d\n", totalbufsize);
  smaster = calloc(totalbufsize, sizeof(float));
  sstg = calloc(levels+1, sizeof(float **));
  for(i=0; i<levels+1; ++i){
```

```
    sstg[i] = calloc(pow(2, levels-i), sizeof(float *));
    assert(sstg[i] != NULL);
  }
  sstg[0][0] = &smaster[0];
  temp = &sstg[0][0][0];

  for(i=0; i<levels+1; ++i){
    for(j=0; j<pow(2, levels-i); ++j){
      sstg[i][j] = temp +j*sdatachunk[i];                                    270
    }
    temp = sstg[i][j-1] + sdatachunk[i];
  }
  return(sstg);
}

int shift(float *buf, int points, int extent)
{
  int i;
                                                                             280
  for(i=0; i< (extent-points) ; ++i){
    buf[i] = buf[i+points];
  }
  for(i=(extent-points); i<extent; ++i){
    buf[i] = 0;
  }
}

int add(float *buf1, float *buf2, int extent)
{                                                                            290
  int i;

  for(i=0; i<extent; ++i){
    buf1[i] = buf1[i] + buf2[i];
    buf2[i] = 0;
  }
}

int AssignToH(float ***stg, float *h, int levels)
{                                                                            300
  int i;

  for(i=0; i<pow(2, levels); ++i){
    h[i] = stg[levels][i][0];
  }
}
```

```
int AssignToY(float ***stg, float *y, int levels)
{
  int i;                                                          310

  for(i=0; i<pow(2, levels); ++i){
    y[i] = stg[levels][0][i];
  }
}

int GetEvenOdd(float *feven, float *fodd, float *f, int fq)
{
  int i;

                                                                  320
  for(i=0; i<fq-1; i=i+2){
    feven[i/2] = f[i];
    fodd[i/2] = f[i+1];
  }
}

int FindHalfLength(int *fevenq, int *foddq, int fq)
{

  if(((float)fq)/2 == ceil(((float)fq)/2)){                       330
    fevenq[0] = fq/2;
    foddq[0] = fq/2;
  }
  else{
    fevenq[0] = (fq+1)/2;
    foddq[0] = (fq-1)/2;
  }
  return 0;
}

                                                                  340
int ZeroOut(float *buf, int extent)
{
  int i;

  for(i=0; i<extent; ++i){
    buf[i] = 0;
  }
}

float Energy(float *sig, int q)                                   350
{
```

```
 int i;
 float e;

 printf("q = %d\n", q);
 e = 0;
 for(i=0; i<q; ++i){
  e = e + pow(sig[i], 2);
 }
 return e;
}

int PrintVector(char *message, float *vec, int vecq)
{
 int i;

 printf("%s\n", message);
 if(vecq > 100){
  vecq = 100;
 }
 for(i=0; i<vecq; ++i){
  printf("%20.16f\n", vec[i]);
 }
 printf("--------------------\n");
}

int MakeOnes(float *x, int q)
{
 int i;

 for(i=0; i<q; ++i){
  x[i] = 1;
 }
}
```

A.3 Treefunctions.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "convtest.h"

int convolvedown2(float *out, float *ss1, float *ss2, float ls1, float ls2)
```

```
{
 int i, j, k, outlength, fringelength1, fringelength2, middlelength, flagA, flagB, flagC;
 float l1, l2;
 float *s1, *s2;

 if(ls1 > ls2){
  l1 = ls1;
  l2 = ls2;
  s1 = ss1;
  s2 = ss2;
 }
 else{
  l1 = ls2;
  l2 = ls1;
  s1 = ss2;
  s2 = ss1;
 }

 if(ceil(l1/2) == l1/2 && ceil(l2/2) == l2/2){
  flagA = 1;
  flagB = 1;
  flagC = 0;
 }
 if(ceil(l1/2) != l1/2 && ceil(l2/2) == l2/2){
  flagA = 1;
  flagB = 2;
  flagC = 1;
 }
 if(ceil(l1/2) == l1/2 && ceil(l2/2) != l2/2){
  flagA = 0;
  flagB = 1;
  flagC = 0;
 }
 if(ceil(l1/2) != l1/2 && ceil(l2/2) != l2/2){
  flagA = 0;
  flagB = 2;
  flagC = 0;
 }
 outlength = (int)ceil((l1+l2-1)/2);
 fringelength1 = ceil(l2/2);
 fringelength2 = fringelength1-flagC;
 for(i=0, j=1; i < fringelength1; ++i, j=j+2){
  for(k=0; k < j; ++k){
   out[i] = out[i] + s1[k]*s2[j-k-1];
  }
```

```
  }
  middlelength = outlength − fringelength1 − fringelength2;
  for(i=fringelength1, j=flagA; i < fringelength1 + middlelength; ++i, j=j+2){
    for(k=j; k < (int)l2 + j; ++k){
      out[i] = out[i] + s1[k]*s2[(int)l2+j−k−1];
    }
  }                                                                          60
  for(i=fringelength1 + middlelength, j=(int)l1−(int)l2+flagB; i < outlength; ++i, j=j+2){
    for(k=j; k< (int)l1; ++k){
      out[i] = out[i] + s1[k]*s2[(int)l2+j−k−1];
    }
  }
}

int convolveup2(float *out, float *s, float *feven, float *fodd, float ls, float lfeven, float lfodd)
{
  int i, outlength;                                                          70
  float *evenreturn, *oddreturn;

  outlength = (int)(ls*2 − 1 + lfeven + lfodd − 1);
  evenreturn = convolve(feven, s, lfeven, ls);
  oddreturn = convolve(fodd, s, lfodd, ls);
  for(i=0; i<outlength; i=i+2){
    out[i] = evenreturn[i/2];
    out[i+1] = oddreturn[i/2];
  }
}                                                                            80

float *convolve(float *ss1, float *ss2, float ls1, float ls2)
{
  int i, j, l1, l2, outlength;
  float *s1, *s2, *out;

  outlength = ls1+ls2−1;
  out = calloc(outlength, sizeof(float));
  if(ls1 > ls2){
    l1 = ls1;                                                                90
    l2 = ls2;
    s1 = ss1;
    s2 = ss2;
  }
  else{
    l1 = ls2;
    l2 = ls1;
    s1 = ss2;
```

```
  s2 = ss1;
 }
 for(i=0; i<l2-1; ++i){
  for(j=0; j<i+1; ++j){
   out[i] = out[i] + s1[j]*s2[i-j];
  }
 }
 for(i=l2-1; i<outlength-l2+1; ++i){
  for(j=0; j<l2; ++j){
   out[i] = out[i] + s1[i-l2+1+j]*s2[l2-j-1];
  }
 }
 for(i=outlength-l2+1; i<outlength; ++i){
  for(j=0; j<outlength-i; ++j){
   out[i] = out[i] + s1[i-l2+1+j]*s2[l2-j-1];
  }
 }
 return out;
}
```

A.4 Getsnd.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <linux/soundcard.h>
#include <math.h>
#include "zipper.h"

#define RATE 44100
#define CHANNELS 2
#define SIZE 16


int getsound(float *x, char *insoundfile)
{
 FILE *ifp;
 size_t nmemb;
```

```
 signed short *buf;
 unsigned short *bufunsigned;
 long int q, i;

 nmemb = 10000000;
 buf = calloc(2, sizeof(signed short));
 bufunsigned = calloc(nmemb, sizeof(unsigned short));

 ifp = fopen(insoundfile, "r");
                                                                    30
 i = 0;
 fread(buf, sizeof(signed char), 1, ifp);
 while(feof(ifp) < 1){
    fread(buf, sizeof(signed short), 2, ifp);
    bufunsigned[i] = (unsigned short)(buf[0] + pow(2, 15));
    x[i] = (float)(bufunsigned[i] - pow(2, 15));
    ++i;
  }
 q = i;
 fclose(ifp);                                                       40
 free(buf);
 free(bufunsigned);
 return(q);
}

int sndconvert(unsigned char *ychar, float *y, int vectorsize)
{
 int i;

 for(i=0; i < vectorsize; ++i){                                     50
   ychar[i] = (unsigned char)((y[i] + pow(2, 15))/pow(2, 8));
 }
 return 0;
}


int playsnd(unsigned char *ychar, long int q)
{
 int status, fd, arg, i, fmt, formats;
 unsigned char *temp;                                               60

 temp = calloc(2, sizeof(unsigned char));

 status = fd = open("/dev/dsp", O_RDWR);
 if (status == -1) {
```

```
  perror("error opening /dev/dsp");
  exit(1);
}

arg = SIZE;                                                          70
status = ioctl(fd, SOUND_PCM_WRITE_BITS, &arg);
if (status == -1){
  perror("SOUND_PCM_WRITE_BITS ioctl failed");
}
if (arg !=SIZE){
  perror("unable to set sample size");
}

arg = RATE;
status = ioctl(fd, SOUND_PCM_WRITE_RATE, &arg);                      80
if (status == -1) {
  perror("error from SOUND_PCM_WRITE_RATE ioctl");
  exit(1);
}

arg = CHANNELS;
status = ioctl(fd, SOUND_PCM_WRITE_CHANNELS, &arg);
if (status == -1) {
  perror("error from SOUND_PCM_WRITE_CHANNELS ioctl");
  exit(1);                                                           90
}
if (arg != CHANNELS){
  perror("unable to set number of channels");
}

fmt = AFMT_U8;
status = ioctl(fd, SOUND_PCM_SETFMT, &fmt);
if(status == -1){
  perror("SOUND_PCM_SETFMT ioctl failed");
}                                                                    100

/*-----------------------------------------------
printf("Supported Formats:\n");
status = ioctl(fd, SOUND_PCM_GETFMTS, &formats);
if(formats & AFMT_U16_BE){
  printf("  AFMT_U16_BE is an option\n");
}
if(formats & AFMT_U8){
  printf("  AFMT_U8 is an option\n");
}                                                                    110
```

```
 if(formats & AFMT_S16_BE){
  printf("  AFMT_S16_BE is an option\n");
 }
 if(formats & AFMT_U16_BE){
  printf("  AFMT_U16_BE is an option\n");
 }
 if(formats & AFMT_S16_LE){
  printf("  AFMT_S16_LE is an option\n");
 }
 if(formats & AFMT_U16_LE){                                          120
  printf("  AFMT_U16_LE is an option\n");
 }*/
 /*-------------------------------------------------*/


 for(i = 0; i < q; ++i){
  temp[0] = ychar[i]*1.2;
  status = write(fd, temp, sizeof(unsigned char));
  if (status == -1) {
   perror("error writing to /dev/dsp");                              130
   exit(1);
  }
 }

 status = close(fd);
 if (status == -1) {
  perror("error cloing /dev/dsp");
  exit(1);
 }
 free(temp);                                                         140
 return 0;
}

int filesnd(char *argv[], float *h, int q)
{
 FILE *fp;
 int i;

 fp = fopen(argv[2], "w");
 for(i=0; i<q ; ++i){                                                150
  fprintf(fp, "%f\n", h[i]);
  }
 fclose(fp);
 return 0;
}
```

A.5 Randsnd.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "zipper.h"
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define MNMX (1.0-EPS)

int rndsnd(float *r, long int q)
{
  int i;
  long *idum;

  idum = calloc(1, sizeof(long));
  *idum = -1223;
  for(i=0; i<q; ++i){
    r[i] = (float)ran1(idum)*pow(2, 15);
  }
  free(idum);
}

float ran1(long *idum)
{
  int j;
  long k;
  static long iy=0;
  static long iv[NTAB];
  float temp;

  if (*idum <= 0 || !iy){
    if (-(*idum) < 1) *idum=1;
    else *idum = -(*idum);
```

```
    for (j=NTAB+7; j>=0; j--){
      k=(*idum)/IQ;
      *idum=IA*(*idum-k*IQ)-IR*k;
      if (*idum <0) *idum+=IM;
      if (j< NTAB) iv[j] = *idum;
    }
    iy=iv[0];
  }
  k = (*idum)/IQ;
  *idum = IA*(*idum-k*IQ)-IR*k;
  if (*idum <0) *idum += IM;
  j = iy/NDIV;
  iy = iv[j];
  iv[j] = *idum;
  if ((temp = AM*iy) > MNMX) return MNMX;
  else return temp;
}
```

A.6 Str.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "zipper.h"

void itoa(int n, char s[])
{
  int i, sign;

  if (( sign=n ) < 0)
    n = -n;
  i = 0;
  do {
    s[i++] = n % 10 + '0';
  } while ((n /= 10) > 0);
  if (sign < 0 )
    s[i++] = '-';
```

```
 s[i] = '\0';
 reverse(s);
}

int trim(char *s)
{
 int n;

 for (n = strlen(s)-1; n >= 0; n--)
  if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
   break;
 s[n+1] = '\0';
 return n;
}

void reverse(char *s)
{
 int c, i, j;

 for (i = 0, j= strlen(s)-1; i < j; ++i, --j) {
  c = s[i];
  s[i] = s[j];
  s[j] = c;
 }
}
```

A.7 Makefile

```
muixblock : muixblock.o dataread.o getsnd.o randsnd.o str.o
        cc -o muixblock muixblock.o dataread.o getsnd.o randsnd.o str.o

muixtree: muixtree.o getsnd.o randsnd.o treefunctions.o
        cc -lm -o muixtree muixtree.o getsnd.o randsnd.o treefunctions.o

muixblock.o : muixblock.c
        cc -c -g  muixblock.c

muixtree.o : muixtree.c
        cc -c -g muixtree.c

dataread.o : dataread.c
        cc -c -g dataread.c
```

```
getsnd.o : getsnd.c
        cc -c -g getsnd.c

randsnd.o : randsnd.c
        cc -c -g randsnd.c

str.o : str.c
        cc -c -g str.c

treefunctions.o : treefunctions.c
        cc -c -g treefunctions.c
```

Appendix B

Paraunitary and Unimodular Filters

B.1 Paraunitary Filters

What follows are the four filters used the most often in this thesis. They are the Daubechies length 30 filters:

$$H_0(z) = \begin{bmatrix} 6.13335991330609e-08 \\ -6.31688232588197e-07 \\ 1.81127040794067e-06 \\ 3.36298718173796e-06 \\ -2.81332962660485e-05 \\ 2.57926991553219e-05 \\ 0.00015589648992062 \\ -0.000359565244362443 \\ -0.000373482354137603 \\ 0.00194332398038203 \\ -0.000241756490763196 \\ -0.00648773456032142 \\ 0.00510100036039435 \\ 0.0150839180278113 \\ -0.0208100501697309 \\ -0.0257670073284824 \\ 0.054780550584479 \\ 0.0338771439235032 \\ -0.111120936037216 \\ -0.039666176555759 \\ 0.190146714007163 \\ 0.0652829528487984 \\ -0.288882596566965 \\ -0.193204139609152 \\ 0.339002535454739 \\ 0.645813140357444 \\ 0.492631771708155 \\ 0.206023863987003 \\ 0.0467433948927679 \\ 0.00453853736157906 \end{bmatrix}$$

$$H_1(z) = \begin{bmatrix} -0.00453853736157906 \\ 0.0467433948927679 \\ -0.206023863987003 \\ 0.492631771708155 \\ -0.645813140357444 \\ 0.339002535454739 \\ 0.193204139609152 \\ -0.288882596566965 \\ -0.0652829528487984 \\ 0.190146714007163 \\ 0.039666176555759 \\ -0.111120936037216 \\ -0.0338771439235032 \\ 0.054780550584479 \\ 0.0257670073284824 \\ -0.0208100501697309 \\ -0.0150839180278113 \\ 0.00510100036039435 \\ 0.00648773456032142 \\ -0.000241756490763196 \\ -0.00194332398038203 \\ -0.000373482354137603 \\ 0.000359565244362443 \\ 0.00015589648992062 \\ -2.57926991553219e-05 \\ -2.81332962660485e-05 \\ -3.36298718173796e-06 \\ 1.81127040794067e-06 \\ 6.31688232588197e-07 \\ 6.13335991330609e-08 \end{bmatrix}$$

$$F_0(z) = \begin{bmatrix} 0.00453853736157906 \\ 0.0467433948927679 \\ 0.206023863987003 \\ 0.492631771708155 \\ 0.645813140357444 \\ 0.339002535454739 \\ -0.193204139609152 \\ -0.288882596566965 \\ 0.0652829528487984 \\ 0.190146714007163 \\ -0.039666176555759 \\ -0.111120936037216 \\ 0.0338771439235032 \\ 0.054780550584479 \\ -0.0257670073284824 \\ -0.0208100501697309 \\ 0.0150839180278113 \\ 0.00510100036039435 \\ -0.00648773456032142 \\ -0.000241756490763196 \\ 0.00194332398038203 \\ -0.000373482354137603 \\ -0.000359565244362443 \\ 0.00015589648992062 \\ 2.57926991553219e-05 \\ -2.81332962660485e-05 \\ 3.36298718173796e-06 \\ 1.81127040794067e-06 \\ -6.31688232588197e-07 \\ 6.13335991330609e-08 \end{bmatrix} \qquad F_1(z) = \begin{bmatrix} 6.13335991330609e-08 \\ 6.31688232588197e-07 \\ 1.81127040794067e-06 \\ -3.36298718173796e-06 \\ -2.81332962660485e-05 \\ -2.57926991553219e-05 \\ 0.00015589648992062 \\ 0.000359565244362443 \\ -0.000373482354137603 \\ -0.00194332398038203 \\ -0.000241756490763196 \\ 0.00648773456032142 \\ 0.00510100036039435 \\ -0.0150839180278113 \\ -0.0208100501697309 \\ 0.0257670073284824 \\ 0.054780550584479 \\ -0.0338771439235032 \\ -0.111120936037216 \\ 0.039666176555759 \\ 0.190146714007163 \\ -0.0652829528487984 \\ -0.288882596566965 \\ 0.193204139609152 \\ 0.339002535454739 \\ -0.645813140357444 \\ 0.492631771708155 \\ -0.206023863987003 \\ 0.0467433948927679 \\ -0.00453853736157906 \end{bmatrix}$$

B.2 Unimodular Filters

The following are the filters used in the unimoular example of Chapter 7. These are the direct form filters that are needed for the computer program of Appendix A.

$$H_0(z) = \begin{bmatrix} 0.997558593750017 \\ -0.0105974017849973 \\ -0.0151510326456218 \\ -0.0328830116669829 \\ -0.0230868349889613 \\ 0.030841381835987 \\ 0.0446868225686526 \\ 0.187034811718881 \\ 0.149613862445247 \\ -0.0279837694169838 \\ -0.0423879964631563 \\ -0.63088076792959 \\ -0.316317568640353 \\ 0.714846570552542 \\ 0.0257044130802508 \\ -0.230377813308855 \\ 0.316317568640353 \\ 0 \\ -0.0423879964631563 \\ 0 \\ -0.149613862445247 \\ 0 \\ 0.0446868225686526 \\ 0 \\ 0.0230868349889613 \\ 0 \\ -0.0151510326456218 \\ 0 \\ 0.00244140624998303 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad H_1(z) = \begin{bmatrix} 0.230377813308855 \\ 1 \\ 0.714846570552542 \\ 0 \\ 0.63088076792959 \\ 0 \\ -0.0279837694169838 \\ 0 \\ -0.187034811718881 \\ 0 \\ 0.030841381835987 \\ 0 \\ 0.0328830116669829 \\ 0 \\ -0.0105974017849973 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$F_0(z) = \begin{bmatrix} -0.230377813308855 \\ 1 \\ -0.714846570552542 \\ 0 \\ -0.63088076792959 \\ 0 \\ 0.0279837694169838 \\ 0 \\ 0.187034811718881 \\ 0 \\ -0.030841381835987 \\ 0 \\ -0.0328830116669829 \\ 0 \\ 0.0105974017849973 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad F_1(z) = \begin{bmatrix} 0.997558593750017 \\ 0.0105974017849973 \\ -0.0151510326456218 \\ 0.0328830116669829 \\ -0.0230868349889613 \\ -0.030841381835987 \\ 0.0446868225686526 \\ -0.187034811718881 \\ 0.149613862445247 \\ 0.0279837694169838 \\ -0.0423879964631563 \\ 0.63088076792959 \\ -0.316317568640353 \\ -0.714846570552542 \\ 0.0257044130802508 \\ 0.230377813308855 \\ 0.316317568640353 \\ 0 \\ -0.0423879964631563 \\ 0 \\ -0.149613862445247 \\ 0 \\ 0.0446868225686526 \\ 0 \\ 0.0230868349889613 \\ 0 \\ -0.0151510326456218 \\ 0 \\ 0.00244140624998303 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Bibliography

[1] K. G. Beauchamp. *Applications of Walsh and Related Functions with an Introduction to Sequence Theory.* Microelectronics and Signal Processing. Acedemic Press, Inc., 1984.

[2] A. R. Calderbank, Ingrid Daubechies, Wim Sweldens, and Boon-Lock Yeo. Lossless image compression using integer to integer wavelet transforms. In *Proceedings of the 1997 International Conference on Image Processing.*, volume 1, pages 596–599, Santa Barbara, CA, October 1997. IEEE Comp. Soc.

[3] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comput.*, 19(2):297–301, April 1965.

[4] James W. Cooley. How the fft gained acceptance. *IEEE SP Magazine*, pages 10–13, January 1992.

[5] William G. Gardner. Efficient convolution without input-output delay. *AES, Journal of the Audio Engineering Society*, 43(3):127–136, March 1995.

[6] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. Gauss and the history of the fast fourier transform. *IEEE ASSP Magazine*, pages 14–21, October 1984.

[7] B. R. Hunt. *Applications of Digital Signal Processing*, chapter 4, page 193. Prentice-Hall Signal Processing Series. Prentice-Hall Inc., 1978.

[8] Toshio Irinoa and Hideki Kawahara. Signal reconstruction from modified auditory wavelet transform. *IEEE Transactions on Signal Processing*, 41(12):3549–3554, December 1993.

[9] Hyuk Jeong and Jeong-Guon Ih. Implementation of a new algorithm using the stft with variable frequency resolution for the time-frequency auditory model. *AES, Journal of the Audio Engineering Society*, 47(4):240–251, April 1999.

[10] Al Kelley and Ira Pohl. *A Book on C.* Addison-Wesley, 1998.

[11] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall Software Series. Prentice Hall PTR, 1998.

[12] Masoud R. K. Khansari and Alberto Leon-Garcia. Subband decomposition of signals with generalized sampling. *IEEE Transactions on Signal Processing*, 41(12):3365–3376, December 1993.

[13] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing.* Prentice-Hall Signal Processing Series. Prentice-Hall Inc., 1989.

[14] Athanasios Papoulis. *Probability, Random Variables, and Stochastic Processes.* McGraw-Hill Series in Electrical Engineering; Communications and Signal Processing. WCB/McGraw-Hill, 1991.

[15] Ken C. Pohlmann. *Principles of Digital Audio.* McGraw-Hill, Inc., third edition, 1995.

[16] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C - The Art of Scientific Computing.* Cambridge University Press, second edition edition, 1997.

[17] Shie Qian and Dapang Chen. *Joint Time-Frequency Analysis.* Prentice Hall PTR, 1996.

[18] Pavan K. Ramarapu and Robert C. Maher. Methods for reducing audible artifacts in a wavelet-based broad-band denoising system. *AES, Journal of the Audio Engineering Society*, 46(3):178–190, March 1998.

[19] Gilbert Strang and Truong Nguyen. *Wavelets and Filter Banks.* Wellesley-Cambridge Press, 1997.

[20] P. P. Vaidyanathan. *Multirate Systems and Filter Banks.* Prentice-Hall Signal Processing Series. Prentice-Hall Inc., 1993.

[21] M. V. Wickerhauser and R. R. Coifamn. Entropy based algorithms for best basis selection. *IEEE Transactions on Information Theory*, 32:712–718, March 1992.