THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

Adjustable Subband Allocation Algorithm for Critically Sampled Subband Adaptive Filters

by

Adam Shabti Charles

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Engineering

May 6, 2009

Advisor

Dr. Fred L. Fontaine

THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

Dr. Eleanor Baum

Dean, School of Engineering

Dr. Fred L. Fontaine

Candidate's Thesis Advisor

Acknowledgments

I would like to thank my advisor, Dr. Fred Fontaine, for his guidance and patience throughout this process. Without his teachings I would not be where I am today. I would also like to thank the rest of the faculty, as well as my friends and peers at The Cooper Union Albert Nerken School of Engineering. A special thanks is due to David Nummey, Deian Stefan, Ashwin Kirpalani, Stefan Münzel and Matthew Epstein, all of whom gave their time to listen patiently to my ideas and help me improve this thesis into what it is today. I would also like to thank Dr. Jack Lowenthal for keeping me motivated with his interest in my studies and projects. Lastly I like to thank my family, especially my parents, Dr. Richard and Shulamit Charles, my uncle Louis Charles, and my sister Aya for their constant support throughout my life.

Abstract

Subband adaptive filters utilize subband decompositions to reduce the length of the adaptive filters, and thus reduce the number of computations needed to adapt for very large filters. Smaller bands have been shown to greatly reduce the computational complexity, but at the cost of performance. Both the convergence rate and the misadjustment of the adaptive structure suffer due to the decomposition.

Tridiagonal transfer functions as well as oversampling have been proposed to reduce these effects [6, 21]. More recently, non-uniform subband decompositions have been proposed in order to cover the cross-terms and reduce the convergence time [20]. The issue then arises that the optimal subband decomposition is often not known a-priori.

This paper proposes a method of adapting the subband decomposition for non-uniform adaptive filters in order to reduce the misadjustment and convergence time when modeling non-stationary processes. A QMF based tree structure, along with an adaption algorithm were designed and implemented in MATLAB. The algorithm was able to correctly adapt for the changes in the non-stationary unknown transfer function. Both the convergence rate as well as the misadjustment were improved with minimal excess computation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In many signal processing applications, it is necessary to use a filter which is optimal with respect to given criteria. Often, this optimal filter cannot be known prior to the filtering operation, and thus the coefficients cannot be hard-coded into the algorithm. This has led to a class of filters known as adaptive filters, which can adjust their coefficients to achieve the desired optimal behavior of the filter.

A prime example is the problem of echo cancellation in telephony [4, 6, 14]. In this scenario, illustrated in Figure 1.1, a user at terminal A sends a signal $a[n]$ to user B, that is simultaneously responding with a signal $b[n]$, which for simple analysis is uncorrelated to $a[n]$. If the transmitter at B is in close proximity to the receiver, the transformed signal from A may mix with the signal $b[n]$ prior to transmission at B.Thus the signal received at A, $y[n]$, has the z-transform:

$$Y(z) = G_{AB}(z)(B(z) + G_{BA}(z)A(z)) \tag{1.1}$$

where $G_{AB}(z)$ is the transfer function for the path from B to A, $G_{BA}(z)$ is the transfer function from A to B , and $B(z)$ and $A(z)$ are the z-transforms of $b[n]$ and $a[n]$, respectively. In order to clean the received signal at A, the part of the signal that depends on the echo of $a[n]$ must be estimated and eliminated. If the behavior of the transfer function applied to $a[n]$ is known, then a filter $H(z) = G_{AB}(z)G_{BA}(z)$, as in Figure 1.1, can be placed to accomplish this. This filter will remove the undesired part of the received signal, and all that will be left is $E(z) = G_{AB}(z)B(z)$.

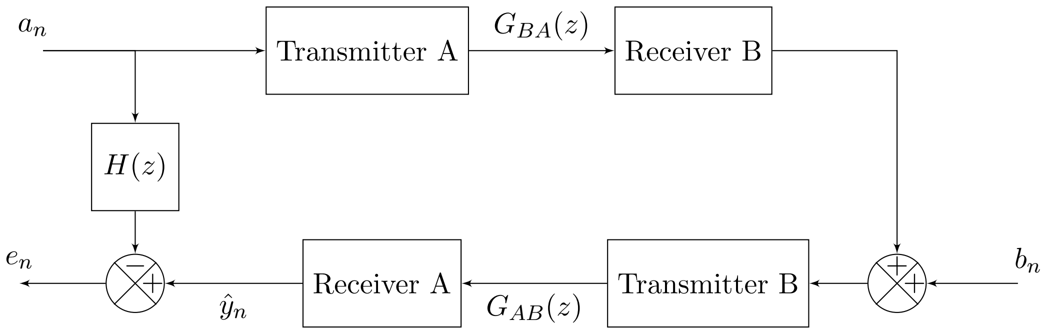

Figure 1.1: Echo Cancellation Problem

Usually, though, the transfer function $G_{BA}(z)G_{AB}(z)$ is unknown and difficult to estimate. Instead, the echo cancellation filter $H(z)$ is initialized at some value, and then adapted in time to accomplish its task. The adaption algorithm aims to change the tap weights of $H(z)$ in order to best cancel out the part of $y[n]$ that is dependent on $a[n]$.

For this particular task, linear prediction and the orthogonality principle are implemented. Instead of attempting to explicitly define dependence of $y[n]$ on each of the inputs, the problem is reformulated into attempting to find the best estimate of $y[n]$ given only past samples of $a[n]$.At each time step, $y[n]$ is a linear sum of past $a[n], b[n]$ and is consequently in $span\{a[i], b[i]\}, i \leq n$. The predictor, on the other hand, is a linear combination of past $a[n]$ and therefore can only calculate what is in the $span\{a[i]\}, i \leq n$. Thus, by the orthogonality principal, the error exists in the space orthogonal to the prediction space, in this case $span\{b[i]\}, i \leq n$.

Generally this method takes into account all past $a[i]$ and $b[i]$, indicating an infinite impulse response (IIR). Although the transfer function from $y[n]$ to $a[n]$ may have an IIR filter response, all the poles are assumed to be inside the unit circle. Thus all impulse responses are of the form $|p|^n \exp(jn\theta)$, with $|p| \leq 1$ and $\theta = \tan^{-1}(\text{Im}(p)/\text{Re}(p))$ is the angle of p. Therefore, it is not unreasonable to say that $y[n]$ can be approximated with a finite number of past $a[n]$ and $b[n]$, putting it in the $span\{a[n], a[n-1], ...a[n-M_1], b[n], b[n-1], ...b[n-M_2]\} \cup V$. Here, V represents the noise space, which is orthogonal to the signal spaces spanned by the M_1 past $a[n]$ values and the M_2 past $b[n]$ values. In terms of the impulse response, given an arbitrarily small number ϵ, there is a finite time where the impulse response associated with every pole will drop below ϵ. In fact this time period can be calculated to be $n_\epsilon = \ln(\epsilon/g_H) / \ln(\max_i |p_i|)$, where g_H is the gain for the mode corresponding to the pole with the maximum magnitude. Then, given noise in the system with variance ϵ, after time n_ϵ the level of the signal is below the noise floor, resulting in a negative signal to noise ratio (SNR). Thus the projection of $y[n]$ onto the signal space of $\{a[n], a[n-1], ..., a[n-M_1]\}$, estimated by the adaptive filter $H(z)$ can be realized as a finite impulse response (FIR) filter rather than an IIR filter.

Modeling $H(z)$ as a FIR filter is advantageous for adaptive filtering because of the relative ease of retaining stability [7]. In adaptive filters the filter coefficients can take on any values and as such the zeros (and possibly poles) of the filter are free to move about the z-plane. In algorithms that only adapt the zeros of a filter, the necessary condition to force the magnitude of the poles to continuously satisfy $|p| < 1$ is no longer required. Instability, however, may still occur in an all-zeros model due to the adaptation feedback algorithm, especially in the case where significant quantization is present. The FIR case provides a very simple model for the case where only the zeros are adapted. This is the reason that the two most widely used adaptive filter algorithms, the least mean squares (LMS) and recursive least squares (RLS) both use an FIR structure [7].

The LMS algorithm is designed to recursively calculate the Wiener filter (explained in section 2.2), the corresponding Wiener-Hopf equations, and the concept of steepest descent. Essentially, the error signal, along with a prescribed step size, determine the amount by which the tap weights change in the direction specified by the data vector $\vec{u}[n]$. This brings the tap weights closer to the optimal Wiener filter coefficients. The actual tap weights might never reach the optimal weights, however, due to the resolution determined by the step size. Instead the tap weight values will oscillate about the Wiener filter coefficients. The RLS algorithm is instead based on the least squares curve fitting problem, utilizing the matrix inversion lemma to recursively compute the solution.

Many variations of these two base adaptive algorithms have been designed [7]. These variations resolve some of the inadequacies of the standard algorithms under certain conditions. For example, the adaptive step size least mean squares (ASS-LMS) attempts to eliminate long term large oscillations about a local minima, which occur when the step size is too large. A similar variation is the adaptive forgetting factor recursive lease squares (AFF-RLS) algorithm. A more specific example is the recursive least squares adaptive threshold nonlinear algorithm (RLS-ATNA) which smooths out impulsive noise by introducing a nonlinear denominator dependent on the error [10, 11].

One of the larger classes of variations of these adaptive algorithms are subband methods [21, 22]. In the subband approach, an M-channel analysis filter bank is employed to break up the desired signal into M subband signals. The adaptive algorithm is then applied to each subband independently, with the final output retrieved by filtering the M subband outputs through a synthesis bank. Using this approach, very large adaptive filters with tap weights numbering in the hundreds can be adapted with only a fraction of the computational complexity. For example the effective number of computations of an LMS filter can be reduced from $2M$ to $2L + M$ using a two band filter structure. Here L is the length of the analysis and synthesis bank filters. This type of structure was derived for both the LMS and the RLS adaptive algorithms [6, 23].

This method, though, does not account for either the cross-terms between bands or the slower convergence due to running the subsystems at a slower rates. It has been shown by Petralgia et. al. [2, 21] that the overlap of the analysis filter banks causes energy in the overlap regions to not be fully represented in the subbands. Thus the signal space that is spanned by the input after the analysis bank is a proper subset of the space spanned by the input prior to the analysis bank. The estimation of the output may then be lacking the part of the projection that would have landed in that space, causing the error of the adaptive filter to rise. In addition the slower rates which the subsystems are running at cause a decrease in convergence approximately proportional to the decimation factor. These phenomena have both been experimentally verified [2, 6, 21–23].

Several methods have been proposed to deal with these issues, including oversampling, tridiagonal transfer functions, and non-uniform subband decompositions, [19, 20]. Oversampling all of the subbands in the analysis bank causes certain regions of the spectrum to be seen by multiple subbands, eliminating the need for extra filters. This solution expands each of the subbands, increasing their update rates and increasing the convergence rates. This method entails unnecessary sampling, and therefore unnecessary computations spent on subsequent calculations.

Tridiagonal and non-uniform subband methods have been proposed to utilize critical sampling. The tridiagonal approach involves artificially including cross-terms, which depend on the subband filter bank coefficients and the coefficients of the two neighboring analysis bank filters. For instance, the cross-term in the k^{th} band from the transition between the k^{th} and $(k+1)^{th}$ subband filters would be $H_k(z)H_{k+1}(z)G_{k+1}(z)$. Here $H_i(z)$ is the i^{th} analysis bank filter, and $G_i(z)$ is the i^{th} subband filter. A symmetric calculation would account for the lower cross-term from the transition between the $(k-1)^{th}$ and k^{th} subband: $H_{k-1}(z)H_k(z)G_{k-1}(z)$. This process adds extra bands, and although they are not updated separately since they depend on the neighboring filters, they require more computation to filter and decimate separately. In addition, they do not have the freedom to adapt based on their own errors since these cross-terms are directly related to the main band filters.

The non-uniform subband decomposition accounts for this by simply expanding bands of interest, automatically including all the cross-terms within that band. The trade-off is that the larger bands have more tap weights to adjust than the smaller bands, increasing the computations required to update them. Also, the bandwidths have to be decided ahead of time, requiring a-priori knowledge of the desired input-output characteristics. In many cases this is not feasible (e.g. if the system is time-varying). In the case of time-varying systems, a method would be needed to change the subband decomposition depending on the power in each band at the input and output.

1.2 Problem Statement

This paper addresses the task of reallocating the bandwidth of non-uniform subband structures on the fly in order to follow the input-output characteristics of the unknown system. With a-priori knowledge of the system, the problem becomes trivial, since the bandwidths can be allocated optimally beforehand. In the case of time-varying or completely unknown systems, the bandwidths cannot be pre-allocated and therefore need to be adjusted as information is obtained about the system.

This problem can be broken up into two main parts. The first is to design an appropriate subband filter structure. This structure must be able to change subband decompositions quickly and efficiently while maintaining the perfect reconstruction condition. Initialization of the structure may or may not be important depending on the adaptability of the algorithm itself. There has been much work in this area, and optimal structures, as well as structures that increase the ease of implementation have been proposed [17, 28]. While the optimal structures are very generalized, the structures that ease implementation are usually based on tree structured filter banks.

The second task is to devise an algorithm by which the bandwidths are reallocated while the adaptive filter is running. Some algorithms to make decisions on the reallocation have been proposed, but these algorithms deal with the oversampled case [15–17]. An algorithm than can retain the critically sampled property of the subband decomposition as it adjusts the subband widths would be beneficial in such adaptive systems.

1.3 Previous Work

There have been some proposals for such a subband reallocation algorithm by McCloud and Etter in [15,16]. The algorithm outlined in [15] deals with oversampled systems, and focuses on using the subband widths to isolate transition regions. This has been proven by Griesbach in [8] to decrease the minimum attainable steady state error, $J(\infty)$. Using smaller subbands around transition regions has been shown to lead to better results when compared to uniform subband decompositions [8]. Their design, however, still tends to larger subband widths in regions where there is no signal relative to the minimum allowable band width, when using smaller widths would save more computations [8].

The algorithm closest to the one proposed here is given by McCloud and Etter [16], but this algorithm depends on the error power to change the subband decomposition. This dependence can cause the algorithm to suffer from similar unnecessary adaptations as the adaptive algorithms themselves. For instance, under burst noise conditions, not only are the adaptive filters changing due to this extraneous error, but now the entire structure is being changed because of this error. In addition, this algorithm does not address the initialization of the new subband filters. Thus, after every shift in subband structure, the filter has to reconverge to the optimal tap weights for a longer period of time.

1.4 Proposed Solution

This paper proposes an algorithm for allocating subbands for use in critically sampled systems and with a focus on signal strength rather than transition bands, as used by the McCloud and Etter, [16]. Based on the estimates of the input and output power spectra, the widths of the subbands are changed in order to better suit the unknown desired transfer function. Consequently, in bands with low signal power, the bands are kept small in order to continue saving computations.

Tree structured analysis and synthesis banks are used in order to ease the transitioning between subband widths. In addition, to avoid unnecessary setbacks in the convergence, the new subband filters are initialized with an effective transfer function that approximates the total response of the old branch.

This paper is organized as follows: Chapter 2 deals with the underlying theory of adaptive filters. Chapter 3 reviews multirate filter bank theory and the perfect reconstruction criterion. Chapter 4 applies the subband technique to adaptive filters as demonstrated in [20] and Chapter 5 proposes an algorithm to adjust the subband widths and reinitialize the newly formed subband adaptive structure. Chapter 6 shows the results of testing the algorithm under various conditions against the non-adjustable subband filter, and finally Chapter 7 states the conclusions of the experimentation and details additional methods to expand and optimize the proposed algorithm.

Chapter 2

Adaptive Filtering

2.1 Adaptive Filter Theory

Adaptive filtering is a powerful tool with many applications, from beam-forming to system identification [7]. The basic concept of adaptive filtering is to have a filter in which the coefficients are not constant in time, and instead vary with respect to a feedback quantity. The feedback algorithm serves to change the weights with respect to this quantity in order to minimize a cost function. As an example, one of the most common applications for adaptive filters is system identification, as shown in Figure 2.1. In system identification, the noisy output of an unknown filter, $d[n] + v[n]$ is compared to the output of the adaptive filter. The weights of the adaptive filter are then changed with respect to the difference, $e[n]$, between the two outputs in such a way as to reduce that error in the next iteration. In this case, the cost function is usually a function of the error, such as the mean square error (MSE) or the sum of the squares.

In order to simplify the adaptive filtering problem, certain assumptions are made. The most basic assumptions are that the filter is linear, discrete time, and has an

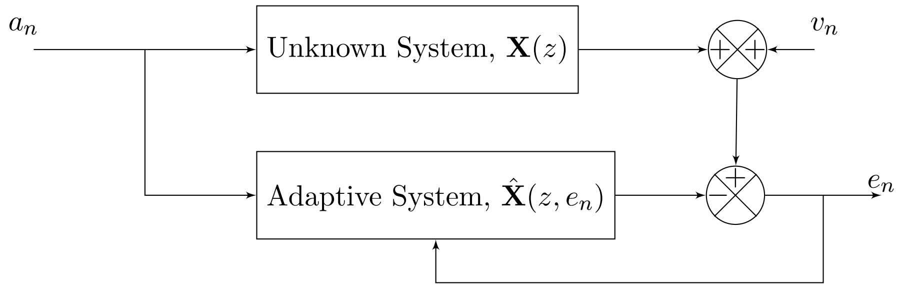
a_n
Unknown System, $\mathbf{X}(z)$
v_n
$+$
$+$
Adaptive System, $\hat{\mathbf{X}}(z, e_n)$
$+$
$-$
e_n

Figure 2.1: General Adaptive Filter as a System Identifier

FIR structure. The reason that the filter is chosen to have a finite impulse response is because FIR filters with constant coefficients have no poles and are therefore automatically stable. When connecting the feedback algorithm to change the filter coefficients, however, even the FIR filter may become unstable. The discrete time assumption allows for a greater flexibility provided for by the programmable nature of the algorithms.

Under these assumptions, there exist a variety of adaptive algorithms to adapt the filter coefficients in order to converge on some optimal set of tap weights. Two of the most widely used and studied are the LMS and RLS algorithms.

2.2 Wiener Filters and LMS Filtering

Given the assumptions of linearity, discrete time and FIR structure, there exists an optimal set of filter tap weights in the least mean squared sense for a stationary input [7]. The least mean squares sense means that the cost function is given by:

$$J[n] = \mathrm{E}\left[|e[n]|^2\right] \tag{2.1}$$

As the name suggests, this is the expected value of the magnitude squared of the error signal. Such a filter is called the Wiener filter. The Wiener-Hopf equation:

$$w_{opt} = \mathbf{R}^{-1}\vec{P} \tag{2.2}$$

states that the optimal set of tap weights for a filter of length M, w_{opt}, is equal to the inverse of the autocorrelation matrix of the past M inputs, $\vec{u}[n]$:

$$\mathbf{R} = \mathrm{E}\left[\vec{u}[n]\vec{u}^{\mathrm{H}}[n]\right] \tag{2.3}$$

multiplied by the cross-correlation vector of $\vec{u}[n]$ with the output, $d[n]$:

$$\vec{P} = \mathrm{E}\left[\vec{u}[n]d^{\mathrm{H}}[n]\right] \tag{2.4}$$

We can observe that the solution depends on a-priori knowledge of the input and output signal statistics. Since this often not the case, adaptive methods that approximate the Wiener filter based on sampled data have been developed. The most common and simplest one is the LMS algorithm.

The LMS algorithm takes the sampled data taken up until time n, and makes the approximations $R \approx \vec{u}[n]\vec{u}^{H}[n]$ and $\vec{P} \approx \vec{u}[n]d^{*}[n]$. Using these approximations in conjunction with the steepest descent method (which states that the optimal change in $\vec{w}$ is in the direction of $-\nabla J$) of estimating the next step result in:

$$\begin{aligned} e(n) &= d[n] - \vec{w}^{H}[n]\vec{u}(n) && (2.5)\\ \vec{w}[n+1] &= \vec{w}[n] + \mu\vec{u}[n]e^{*}[n] && (2.6) \end{aligned}$$

If $w[n]$ has length K, these update equations require an effective total of $2K$ multiplications to evaluate the new tap weights: K multiplications for the $\vec{w}^H[n]\vec{u}[n]$ operation and $K+1$ more for the $\mu\vec{u}[n]e^*[n]$ operation. In this paper, single multiplications such as $\mu e^*[n]$ will be ignored as they are dominated by the terms that depend on the filter length ($O(K)$ or higher).

2.3 Kalman Filters and RLS Filtering

The RLS filter is based on the least squares (LS) curve fitting problem [7]. In the LS problem, a curve is fit to a given set of data points by minimizing the sum of the squares of the errors from all points to the curve. The RLS cost function to minimize is:

$$J(e[n]) = \sum_{i=0}^{n} \beta[i]e^2[i] \tag{2.7}$$

where $\beta[i]$ is a weight factor that tells the importance of each point in the least squares algorithm and $e[i] = d[i] - \vec{w}^H[i]\vec{u}[i]$ is the error of the i^{th} iteration. In the RLS algorithm $\beta[i] = \lambda^{n-i}$ for $0 \leq \lambda \leq 1$. The parameter λ is called the forgetting factor and gives exponentially less credence to past error values. Larger values of λ make the algorithm less adaptable to quick changes since past errors are considered important for longer periods of time. The matrix inversion lemma is then used to calculate the solution to this minimization recursively. An alternate way to view the RLS filtering problem is as a special case of the Kalman Filter. The Kalman filter deals with a dynamical system consisting of a non-observable internal state vector, $\vec{x}[n]$, and an observable noisy measurement, $\vec{y}[n]$. The Kalman filter characterizes the system by finding the estimate of the hidden state vector with the minimum mean squared error (MMSE). The formulation of the system is:

$$\vec{x}[n+1] = \mathbf{F}[n+1,n]\vec{x}[n] + \vec{\nu}_1[n] \tag{2.8}$$

$$\vec{y}[n] = \mathbf{C}[n]\vec{x}[n] + \vec{\nu}_2[n] \tag{2.9}$$

Equation (2.8) is referred to as the process equation and updates the unknown state $\vec{x}[n]$. The measurement equation (2.9) produces the observed quantity $\vec{y}[n]$, known as the measurement. Here $\mathbf{F}[n+1,n]$ is referred to as the state transition matrix from n to $n+1$, $\mathbf{C}[n]$ is the measurement matrix at time n, and $\vec{\nu}_1[n], \vec{\nu}_2[n]$ are independent additive white stochastic processes with covariance matrices $\mathbf{Q}_1[n], \mathbf{Q}_2[n]$ respectively. The equations that define the Kalman filter are:

$$\pi[n] = \mathbf{K}[n,n-1]\mathbf{C}^H[n] \tag{2.10}$$

$$\mathbf{G}_f[n] = \mathbf{F}[n+1,n]\pi[n]\left(\mathbf{C}[n]\pi[n] + \mathbf{Q}_2[n]\right)^{-1} \tag{2.11}$$

$$\vec{\alpha}[n] = \vec{y}[n] - \mathbf{C}[n]\hat{x}[n|n-1] \tag{2.12}$$

$$\hat{x}[n+1|n] = \mathbf{F}[n+1,n]\hat{x}[n|n-1] + \mathbf{G}_f[n]\vec{\alpha}[n] \tag{2.13}$$

$$\mathbf{K}[n] = \left[\mathbf{I} - \mathbf{F}[n,n+1]\mathbf{G}_f[n]\mathbf{C}[n]\right]\mathbf{K}[n,n-1] \tag{2.14}$$

$$\mathbf{K}[n+1,n] = \mathbf{F}[n+1,n]\mathbf{K}[\mathbf{n}]\mathbf{F}^H[n+1,n] + \mathbf{Q}_1[n] \tag{2.15}$$

The basic idea of this algorithm is to whiten the measurement $\vec{y}[n]$ to form $\vec{\alpha}[n]$ (equation 2.12). Then, independent (white) segments of $\vec{y}[n]$ are then pieced together to form the state estimate $\hat{x}[n+1|n]$ (equation (2.13)) based on the Kalman gain, $\mathbf{G}_f[n]$ (equation (2.11)). The Kalman gain calculated in equation (2.11) can be thought of as a ratio of covariances and is used to decide how much credence will be given to the new independent piece of information (contained in the latest measurement). Equations

(2.14) and (2.15) are used to update the estimate of the covariance matrix of $\hat{x}[n+1|n]$. These updates are called the Riccati Equations.

The RLS algorithm can then be formulated as a special case of the Kalman filter where:

$$\begin{aligned}\mathbf{F}[n] &= \lambda^{-1/2} && (2.16)\\ \mathbf{C}[n] &= \vec{u}^H[n] && (2.17)\\ \vec{v}_1[n] &= 0 && (2.18)\\ \vec{v}_2[n] &= \nu[n] && (2.19)\end{aligned}$$

Here λ is the forgetting factor and $\nu[n]$ is white Gaussian noise. The process and measurement equations then become:

$$\begin{aligned}\vec{x}[n+1] &= \lambda^{-\frac{1}{2}}\vec{x}[n] && (2.20)\\ \vec{y}[n] &= \vec{u}^H[n]\vec{x}[n] + \nu[n] && (2.21)\end{aligned}$$

The transformation of the Kalman filtering equations is then as follows:

$$\begin{aligned}\vec{x}[n] &\rightarrow \lambda^{-\frac{n}{2}}\vec{w}[0]\\ \vec{y}[n] &\rightarrow \lambda^{-\frac{n}{2}}d[n]\\ \vec{\nu}_2[n] &\rightarrow \lambda^{-\frac{n}{2}}e_0^*[n]\\ \hat{x}[n+1|n] &\rightarrow \lambda^{-\frac{n+1}{2}}\vec{w}[n]\\ \mathbf{K}[n] &\rightarrow \lambda^{-1}\mathbf{P}[n]\\ \mathbf{G}_f[n] &\rightarrow \lambda^{-1/2}\vec{G}_f[n]\\ \vec{\alpha}[n] &\rightarrow \lambda^{-n/2}\eta[n]\end{aligned}$$

The quantity $\mathbf{P}[n]$ above is the inverse of the input correlation matrix:

$$\mathbf{R}[n] = \sum_{i=0}^{n} \lambda^{n-i} \vec{u}[i]\vec{u}^H[i] \tag{2.22}$$

The Kalman filtering equations then yield the RLS algorithm:

$$\begin{aligned}
\vec{g}_f[n] &= \frac{\mathbf{P}[n-1]\vec{u}[n]}{\lambda + \vec{u}^H[n]\mathbf{P}[n-1]\vec{u}[n]} && (2.23)\\
\eta[n] &= d[n] - \vec{w}^H[n-1]\vec{u}[n] && (2.24)\\
\vec{w}[n] &= \vec{w}[n-1] + \vec{g}_f[n]\eta^*[n] && (2.25)\\
\mathbf{P}[n] &= \lambda^{-1}\left[\mathbf{I} - \vec{g}_f[n]\vec{u}^H[n]\right]\mathbf{P}[n-1] && (2.26)
\end{aligned}$$

We note that the total computational complexity of the RLS update equations is $3K^2 + 2K$ multiplications per iteration. This comes from the three matrix-vector multiplications (one in equation (2.23) and two in equation (2.26)) and two dot products (one in each of equations (2.23) and (2.24)). Although the RLS algorithm has a large increase in the computational complexity over the LMS algorithm per iteration, the convergence time is substantially reduced.

2.4 Evaluation of Adaptive Filter Performance

With so many adaptive filter algorithms, it is necessary to have methods to compare the performance of the filters. The major attributes of an adaptive filter that can be compared are the misadjustment and the convergence rate.

As an adaptive filter converges to the optimal filter, it will eventually oscillate about a steady state value, $\vec{w}(\infty)$. The reason the oscillation occurs is due to the resolution of the step size: the filter cannot converge to the exact optimal filter.

Instead, the path about the error performance surface attempts to reach that value as in the steepest descent, but keeps overshooting. The misadjustment is a measure of how close this steady state value is to the theoretical optimal value $\vec{w}_{opt}$ through the corresponding minimum cost function value J_{min}. Assuming $J[\infty] = \lim_{n\to\infty} J[n]$ exists, the misadjustment $\mathscr{M}$ is defined as:

$$\mathscr{M} = \frac{J[\infty]}{J_{min}} \tag{2.27}$$

This equation is the ratio of the cost function evaluated at $\vec{w}(\infty)$ and $\vec{w}_{opt}$. As J_{min} is the absolute minimum value that can be attained, the misadjustment it always greater then or equal to one. The smaller $\mathscr{M}$ is, the better the steady state performance of the filter.

An alternate way to understand the steady state error is to look at the mean squared error (MSE). The MSE is calculated by running the algorithm a number of times and averaging the results. The result is an approximation of $J[e[n]]$. Since for a given system and cost function, J_{min} is constant and the misadjustment of various algorithms can be compared by looking at the MSE after more iterations than the convergence time.

Although the steady state performance of an adaptive filter is important, the filter may take a long time to converge to steady state.The convergence rate gives a measure of how fast the adaptive filter approaches the steady state conditions in the mean. This quantity is usually measured as the average change in the MSE versus time. This quantity becomes important when dealing with real time applications. In such cases, it may be advantageous to use more complex algorithms that have faster convergence, such as the RLS-based algorithms.

Often, these two parameters are difficult to quantify, leading to visual methods of comparing these characteristics for competing algorithms. Error curves are such a visual aid that allow both the convergence rate and the misadjustment to be viewed simultaneously. The error curve plots either the magnitude of the error, $d[n]-\vec{w}^H[n]\vec{u}[n]$, the magnitude of the error squared, or the cost function over time either linearly or in decibels (dB). The choice of which scale to view the error curves with is dependent on the ease of which the data it represents can be viewed. The convergence rate can be found by observing the slope near the beginning of the curve, and the misadjustment can be calculated by observing the height of the curve at large n. To observe the misadjustments, decibel scale plots are usually optimal, since small changes in the misadjustment are easier to see. For the convergence, large convergence rates are easier to distinguish on a decibel scale, while slow convergence rates are easier to see on a linear scale.

The error curves can be viewed as the output of a single trial, or as an average of a series of runs. Single runs provide a more accurate representation of the steady state oscillations, while the averaging method eliminates these oscillations to better display the convergence rate and the steady state error.

An example plot is shown in Figure 2.2. This figure shows the plots of the error magnitude, averaged over 100 trials, of four filter types: LMS, RLS, Normalized LMS (NLMS) and RLS Adaptive Threshold Nonlinear Algorithm (RLS-ATNA). The plot shows that the RLS based algorithms have a much higher convergence rate and misadjustment than the LMS based algorithms. This increase in performance comes at the cost of a much higher computational complexity.

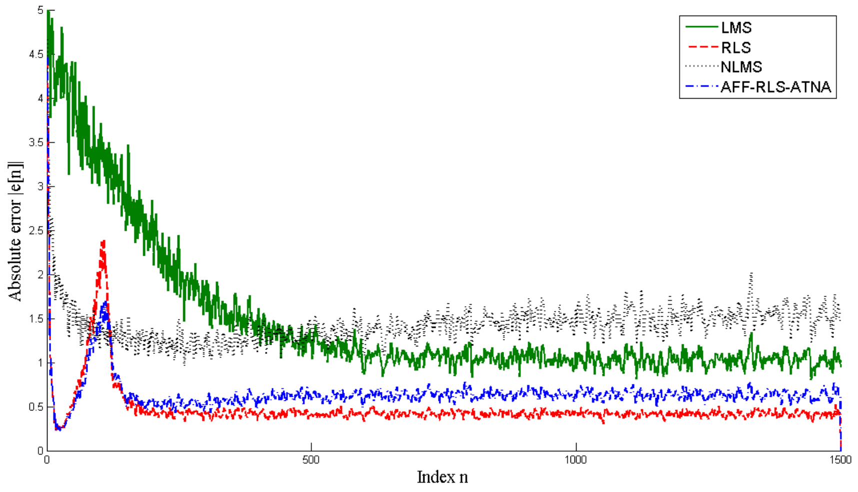

Figure 2.2: Error Curve Plot Comparison of Adaptive Algorithms

2.5 Variations of LMS and RLS Filters

Many derivatives of LMS and RLS, the two main adaptive algorithms, have been established. Each of these have been tailored to special situations in order to increase performance. Some of the algorithms are meant to deal with special noise conditions or other environmental conditions (e.g. non-stationarity), while others are designed to reduce the number of computations needed to run the algorithm.

Usually the variations used to increase performance are not used independently, but in conjunction with one another. This allows for an increase in performance in multiple areas. The issue then is that most algorithms that increase performance in one aspect or another require significantly more computations per iteration. Thus, although most variations can be merged together, the detriment in computational complexity may outweigh the gain in performance. The variations used to save computations, however, may lead to poorer performance than even the standard algorithms. Thus there is a trade-off between the computational complexity of an algorithm and its performance under various conditions.

One of the most basic changes to the LMS and RLS algorithms is to make the parameters, such as the step-size, time-varying [3,24]. In the case of the LMS algorithm, this leads to the Adaptive Step Size LMS (ASS-LMS) algorithm, and in the RLS case, this translates into the Adaptive Forgetting Factor RLS (AFF-RLS) algorithm [11,13].

The ASS-LMS algorithm deals with changing the step size μ into a function of time, $\mu[n]$. A very basic example is when $\mu[n] = \mu^n$ for $|\mu| < 1$. In this case, the step size dies out exponentially with time. The idea here as that as the adaption algorithm converges on the optimal tap weights, the step size decreases in order to lower the misadjustment. The AFF-RLS algorithm considers a similar adaption in the forgetting factor as $\lambda = \lambda[n]$.

Another common variation, the Normalized LMS (NLMS) algorithm, normalizes the $\mu\vec{u}[n]e^*[n]$ term by the total energy in the input vector. This effectively deals with cases where the input signals are large. When the vector $\vec{u}[n]$ has large components, the tap weights undergo large changes proportional to $\vec{u}[n]$. In order to prevent this, the input vector is instead treated as a unit vector by dividing by the norm squared $|\vec{u}[n]|^2 = \vec{u}^H[n]\vec{u}[n]$. This makes the step size and the error signal more dominant.

An issue here, however, arises when the the norm is very small. In this case the tap weight change is again very large again, because of the division by a small number. To account for this, a second parameter, δ, is introduced. The normalization factor is then modified to $[\delta + |\vec{u}[n]|^2]^{-1}$, where δ is a small number, but large enough to not cause the tap weights to increase too dramatically.

In RLS and LMS filtering, the adaptation of the tap weight filters is proportional to $e[n]$. Thus large errors correspond to large changes in the adaptive filter; this problem is similar to that addressed in the NLMS algorithm with the vector $\vec{u}[n]$.

For constant background noise, this does not cause too much of a problem, since the variance of this additive noise is usually small compared to the signal. For shot noise, however, the variance is relatively large, resulting in a very large error regardless of the difference between the adaptive tap weights and the optimal tap weights. Shot noise is defined as a Poisson process, where the events are characterized as additive independent random numbers following a Gaussian distribution with a much larger variance than the background noise ($\sigma^2_{shot} >> \sigma^2_{\nu}$). Thus for regular RLS and LMS filters, this causes the tap weights to change dramatically and the filter needs to re-converge on the optimal filter weights.

For the RLS filter type, a derivative has been proposed to combat this effect. The RLS-ATNA filter makes use of a nonlinear function of the error, $f(e[n])$ to limit the amount by which the tap weights can change at any given iteration. Again the comparison is made with the NLMS algorithm, where a similar term is introduced. The difference is that the RLS-ATNA has higher adaptability within the nonlinear section of the algorithm. For example, a closely related algorithm proposed by Koike in [11] uses $f(e[n]) = (x + y|e[n]|^k)^{-1}$, where both x and y can also be functions of n. Figure 2.3 shows the superior performance of the NLMS and RLS-ATNA algorithms to the regular LMS and RLS algorithms under shot noise.

Since most of the applications of adaptive filtering are real-time, the time it takes to compute each iteration of the algorithm is important. This is directly related to the computational complexity, or the number of addition and multiplication operations needed for every iteration. Since it is generally accepted that multiplication operations in very large integrated (VLSI) digital signal processing (DSP) systems take an order of magnitude grater time to compute than addition operations, it is the number of multiplications that usually dominate the processing time. This is because in the

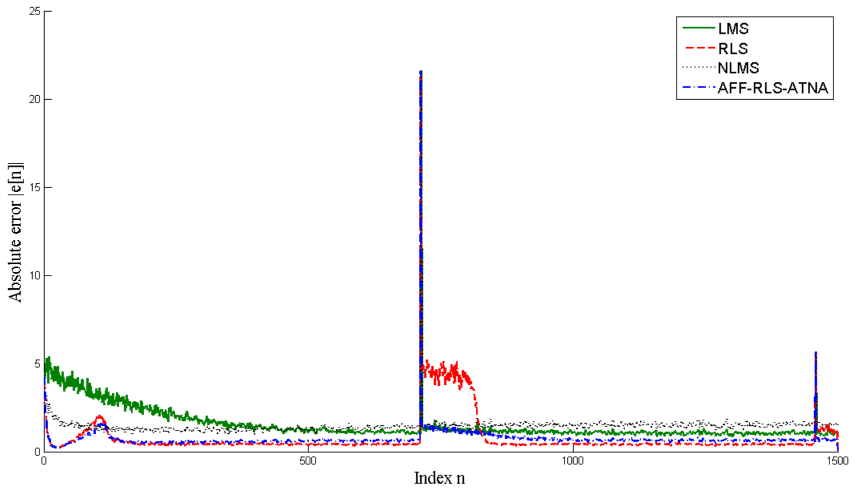

Figure 2.3: Error Curve Plot Comparison of Adaptive Algorithms with Shot Noise

simplest sense, a multiplication unit is a set of adders [9].

Based on this, methods have been devised in order to reduce the complexity of adaptive filtering [7]. One of the main ways of accomplishing this is through batch processing. In batch processing, each iteration is not executed as soon as the data is collected, but instead the data is buffered. When the desired amount of data is obtained, fast matrix operations are used instead of vector operations, allowing for faster calculations. For example fast convolution techniques can be used over a range of inputs instead of filtering directly [18]. Batch processing is one of the few derivatives that focus on saving computation time rather than increasing in performance.

These batch processing techniques are intrinsically tied into frequency domain processing. In the fast convolution case, this is manifested in performing Fast Fourier Transforms (FFT) in order to reduce an N^2 process to an $N\log(N)$ process. This connection to frequency domain calculations leads to formation of subband adaptive filtering techniques that will be discussed in detail in Chapter 4.

Chapter 3

Multirate Filter Banks

3.1 Basics of Multirate Filter Banks

Multirate filter bank theory deals with methods of designing and implementing multiple input-multiple output (MIMO) systems, with subsystems possibly running at different rates. One specific application uses filter bank theory results to calculate the outputs of a long FIR filter using M filters in parallel [25]. This application makes use of the decimation and interpolation operations to change the data rates in such a way that the filtering operation is performed quickly while maintaining the desired output.

The interpolation and decimation operations increase and decrease, respectively, the rate of the incoming signal. The decimation operation takes every M^{th} sample of the incoming signal, and drops the rest, effectively resampling the incoming signal at $(1/M)^{th}$ the initial rate. Thus if the initial signal had a time series $h[n] = \{h[0], h[1], h[2], \ldots\}$, the decimated signal would be $h[Mn] = \{h[0], h[M], h[2M], \ldots\}$.

The z-transform expression is given by:

$$\mathscr{Z}\left[h[Mk]\right] = \frac{1}{M} \sum_{m=0}^{M-1} H\left(z^{\frac{1}{M}} e^{-\frac{2jm\pi}{M}}\right) \tag{3.1}$$

where the $2m\pi/M$ terms for $m \neq 0$ come from the M^{th} roots of unity of the complex number z. The frequency domain expression can then be found by using $z = e^{j\omega}$, yielding:

$$(\downarrow M)H(\omega) = \frac{1}{M} \sum_{m=0}^{M-1} H\left(\frac{\omega}{M} - \frac{2m\pi}{M}\right) \tag{3.2}$$

In equations (3.1) and (3.2) the $m \neq 0$ terms are referred to as the aliasing terms. Interpolation, on the other hand, inserts $M - 1$ zeros between any two consecutive samples of the incoming signal, simulating a sampling at M times the initial sampling rate. The time domain signal is then $h_{up}[n] = h[k]$ if $n = kM$ and zero otherwise. The z-transform is then:

$$\mathscr{Z}\left[h_{up}[n]\right] = H(z^M) \tag{3.3}$$

In the frequency domain this becomes:

$$(\uparrow M)H(\omega) = H(M\omega) \tag{3.4}$$

Equation (3.4) indicates M replications of the spectrum centered at $2\pi m/M$. The spectra at the $m \neq 0$ locations are called imaging terms. It is important to note that the periodic nature of the frequency response of the discrete time Fourier transform (DTFT) needs to be taken into account. This implies that when resampling the signals, either aliasing or imaging can occur. For decimation, the effect is an aliasing of other copies into the $(-\pi, \pi)$ range. For interpolation, the contraction brings in duplicates of the frequency response to populate the rest of the $(-\pi, \pi)$ range. Figure 3.1 displays this concept graphically. Thus to achieve adequate separation of specific frequency bands, both anti-aliasing and anti-imaging filters need to be implemented. The cutoff frequencies for these bands depend on the decimation factor being used, since that decimation factor will dictate which spectral bands will be aliased or imaged.

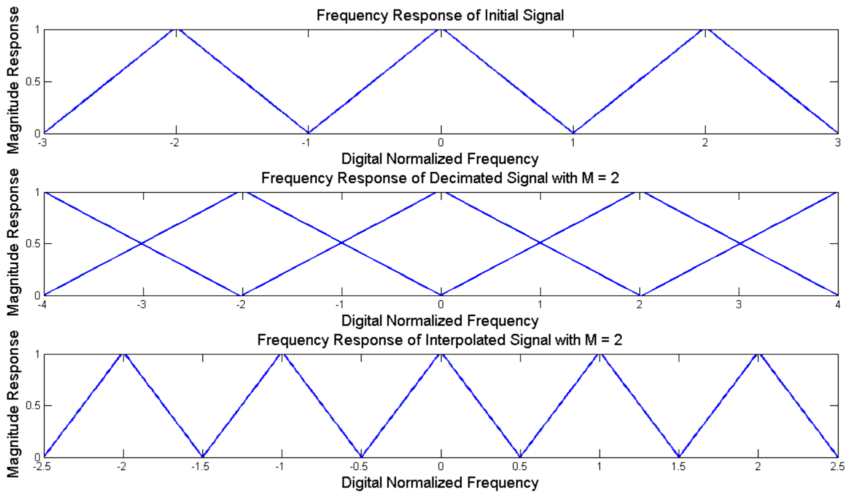

Figure 3.1: Decimation and Interpolation Frequency Transformations

Figure 3.2 shows the full structure of a filter bank complete with anti-aliasing and anti-imaging filters. In Figure 3.2, $H_k(z)$ represents the anti-aliasing filter for the k^{th} band, $F_k(z)$ represents the anti-imaging filter, $G_k(z)$ is the k^{th} subband filter, and M is the resampling factor. The H_k filters comprise what is called the analysis filter bank, while the F_k filters comprise the synthesis filter bank.

In general, not every band has to have the same decimation factor. Non-uniform subband filters utilize different decimation factors to isolate bands of different widths. This leads to the concept of undersampling, oversampling and critical sampling. Over-

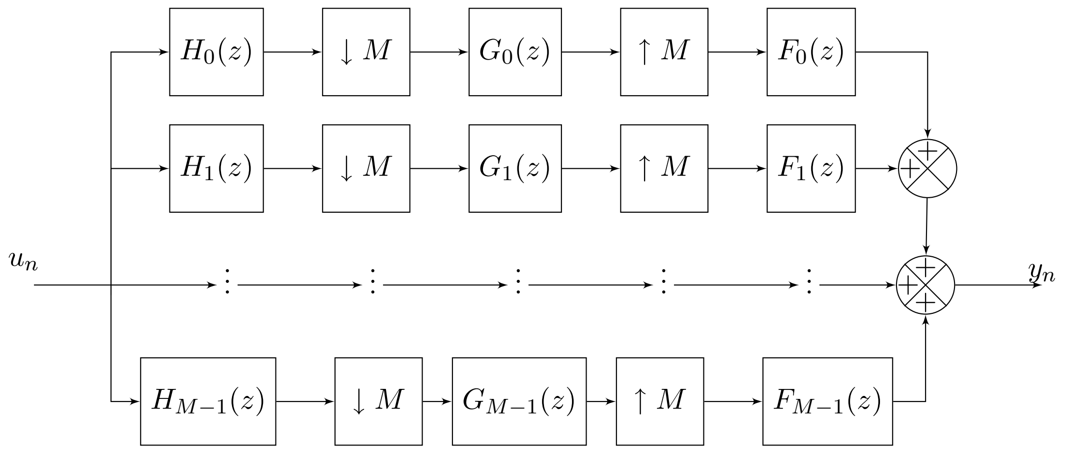

Figure 3.2: M Channel Uniform Filter Bank

sampling is when the number of samples exiting the analysis bank is greater than the number of samples entering. This corresponds to the sum of the inverse decimation factors over all the bands being greater then one. Undersampling is when fewer samples leave the analysis bank than enter, (the sum of the inverse decimation factors is less than one) and critical sampling is when exactly the same number of samples leave as enter (the sum equals one). In terms of information retention, critical sampling is the ideal case, since the information entering the system can always be perfectly represented by the exiting samples with no redundancy.

Here we consider the insertion of adaptive subband filters, $G_i(z)$, between the analysis and synthesis banks, as shown in Figure 3.2. The analysis bank, $H_i(z)$, and synthesis bank, $F_i(z)$, are then adjusted with the critical sampling constraint to achieve a dynamical subband decomposition. The adaptive subband filters $G_i(z)$ are adapted by algorithms similar to the LMS and RLS algorithms in order to achieve the underlying goals of the system.

The aliasing and imaging can also be used to the advantage of a system designer through what are called the Noble Identities. It can be shown that feeding the output of a filter $H(z^M)$ into a decimation operation ($\downarrow M$) is identical to first decimating by M, and then filtering by $H(z)$. Similarly, zero-interpolating by M, and then filtering by $H(z^M)$ is equivalent to first filtering by $H(z)$ and then interpolating by M.

This leads to one of the simplest and most effective ways to use such a filter bank in the FIR case: polyphase decomposition. Polyphase decomposition breaks an FIR filter into the M^{th} polyphase components, or components which only contain samples of the form $Mn + k$ for fixed k ($0 \leq k \leq M - 1$). In terms of the z-transform, this is represented as:

$$H(z) = \sum_{i=0}^{N} z^{-i} E_i(z^M) \tag{3.5}$$

where:

$$E_i(z) = \sum_{l=0}^{N} a_{lM+i} z^{-l} \tag{3.6}$$

Using this decomposition, any FIR filter can be formed made to fit in the structure of Figure 3.2. In this specific structure (called a polyphase filter bank), the analysis and synthesis banks simply consist of delays, while the Noble identities are used to push the $E_i(z^M)$ terms through the decimation operation. Thus instead of one NM length filter with NM multiplications per time-step, M filters of length N are used. This requires only N computations per time-step due to the resampling. This concept will be important when the filter bank is adaptive, and adapting the center filters separately will further reduce the number of necessary computations.

A special case of filter banks is the Quadrature Mirror Filter (QMF), or a filter bank with only two channels where $H_0(z) = H_1(-z) = F_0(z)$ and $F_1(z) = -H_1(z)$. In this case the only filter required to be designed is a low-pass filter, $H_0(z)$. The

QMF structure is popular because of the ease of satisfying certain conditions, as will be discussed in section 3.2. More general subband decompositions can also be constructed using embedded QMF filters. This will be discussed further in section 3.3.

3.2 The Perfect Reconstruction (PR) Condition

When a signal is broken up into its subband components, it is often desired to have the analysis and synthesis filter banks have a minimal effect on the signal. This leads to the idea of the perfect reconstruction (PR) condition. In essence, the PR condition states that: a) the aliased terms usually found in the output due to the multi-rate operations are all fully canceled out and b) the remaining total transfer function acting on the signal is reduced to a constant gain and a delay.

The analysis bank transfer function $\mathbf{H}(z)$ is defined by $\vec{Y}(z) = \mathbf{H}(z)X(z)$, where $X(z)$ is the z-transform of the input signal and $\vec{Y}(z)$ is the Mx1 vector consisting of the outputs which feed into the decimation blocks. The synthesis filter bank matrix $\mathbf{F}(z)$ is defined by $Y(z) = \mathbf{F}(z)\vec{X}(z)$, where $\vec{X}(z)$ is the Mx1 vector consisting of the outputs of the interpolation blocks and $Y(z)$ is the output of the filter bank. By defining $E_{i,k}$ to be the k^{th} polyphase component of the i^{th} analysis filter, and $R_{i,k}$ to be the i^{th} polyphase component of the k^{th} synthesis filter, the analysis filter matrix

$\mathbf{H}(z)$, and the synthesis filter matrix $\mathbf{F}(z)$ can be reformulated as follows:

$$\begin{aligned}
\mathbf{H}(z) &= \begin{bmatrix} H_0(z) & H_1(z) & \dots & H_{M-1}(z) \end{bmatrix}^{\mathrm{T}} \\
&= \begin{bmatrix} E_{0,0}(z^M) & E_{0,1}(z^M) & \dots & E_{0,M-1}(z^M) \\ E_{1,0}(z^M) & E_{1,1}(z^M) & \dots & E_{1,M-1}(z^M) \\ \vdots & \vdots & \ddots & \vdots \\ E_{M-1,0}(z^M) & E_{M-1,1}(z^M) & \dots & E_{M-1,M-1}(z^M) \end{bmatrix} \begin{bmatrix} z^0 \\ z^{-1} \\ \vdots \\ z^{1-M}(z) \end{bmatrix} \\
&= \mathbf{E}(z)\left(\begin{bmatrix} z^0 & z^{-1} & \dots & z^{1-M}(z) \end{bmatrix}^{\mathrm{T}}\right) \qquad (3.7)
\end{aligned}$$

$$\begin{aligned}
\mathbf{F}(z) &= \begin{bmatrix} F_0(z) & F_1(z) & \dots & F_{M-1}(z) \end{bmatrix} \\
&= \begin{bmatrix} z^{1-M} & z^{2-M} & \dots & z^0 \end{bmatrix} \begin{bmatrix} R_{0,0}(z^M) & R_{0,1}(z^M) & \dots & R_{0,M-1}(z^M) \\ R_{1,0}(z^M) & R_{1,1}(z^M) & \dots & R_{1,M-1}(z^M) \\ \vdots & \vdots & \ddots & \vdots \\ R_{M-1,0}(z^M) & R_{M-1,1}(z^M) & \dots & R_{M-1,M-1}(z^M) \end{bmatrix} \\
&= \begin{bmatrix} z^{1-M} & z^{2-M} & \dots & z^0 \end{bmatrix} \mathbf{R}(z) \qquad (3.8)
\end{aligned}$$

where $\mathbf{E}(z)$ is the analysis polyphase matrix and $\mathbf{R}(z)$ is the synthesis polyphase matrix. The PR condition can then be formulated as:

$$\mathbf{E}(z)\mathbf{R}(z) = cz^{-\Delta}\mathbf{I} \qquad (3.9)$$

where c and Δ are constants [25]. For the special case of two band filter banks, as shown in Figure 3.3, the PR condition on the analysis and synthesis filter banks is given by

$$H_0(z)F_0(z) - H_1(z)F_1(z) = 0 \tag{3.10}$$

$$H_0(z)F_0(z) + H_1(z)F_1(z) = C \tag{3.11}$$

for some constant C. In the QMF case, these constraints reduce to constraints on $H_0(z)$, the low-pass filter, only:

$$H_0(z)\tilde{H}_0(z) - H_0(-z)\tilde{H}_0(-z) = 0 \tag{3.12}$$

$$H_0(z)\tilde{H}_0(z) + H_0(-z)\tilde{H}_0(-z) = C \tag{3.13}$$

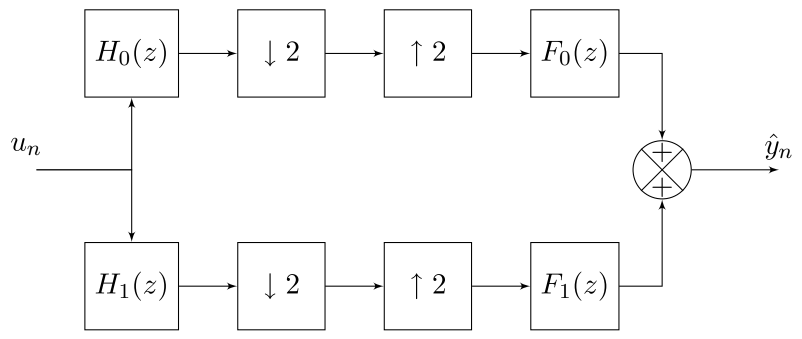

Figure 3.3: Two Channel Filter Bank

In general, the PR condition for FIR analysis and synthesis filter banks is difficult to design for. An alternative to achieve approximate PR is the near perfect reconstruction (NPR) condition. The NPR condition is a weakened version of the PR condition, and states that the constraints are not met precisely, but instead to within some tolerance. Thus, although a small aliased term exists, its magnitude is below some acceptable threshold. This allows iterative algorithms to use established FIR filter design methods to minimize the aliased terms to within an acceptable range [1].

3.3 Tree Structured Filter Banks

The tree structured filter bank is a structure that allows filter banks with large numbers of subbands to be constructed from smaller filter banks with smaller numbers of subbands. These structures can result in either uniform or non-uniform subband decompositions. For example, by implementing a structure such as shown in Figures 3.4 and 3.5, a non-uniform subband decomposition of three bands can be obtained. This is because the upper subband, obtained by high-pass filtering and then decimating by two, is then split again by a second embedded QMF filter bank. The result is one half-band subband and two quarter-band subbands.

This process of embedding filter banks within filter banks can be used to obtain a wide variety of decompositions. For example, to obtain a six channel, uniform filter bank, two three channel filter banks can be embedded in each subband of a QMF filter bank. The effective decimation factor for each resulting channel is then the product of all decimation operations leading up to that channel.

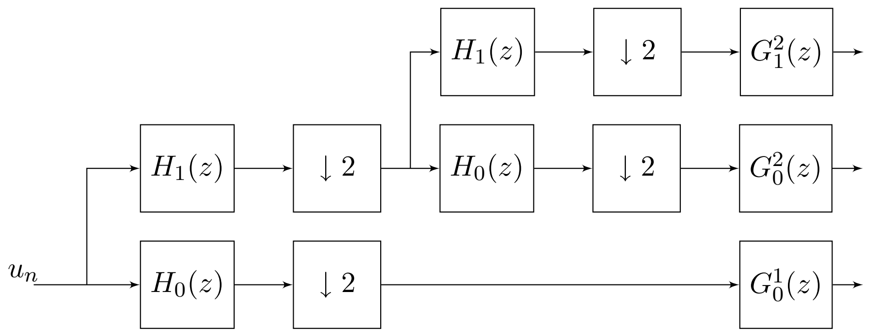

Figure 3.4: Two Tier Tree Structure Analysis Bank With Subband Filters

One of the main reasons to use tree structured filter banks is the ease of designing for the PR condition. It can be shown that as long as each filter bank satisfies the PR condition, the overall resulting structure also satisfies the PR condition [25]. In

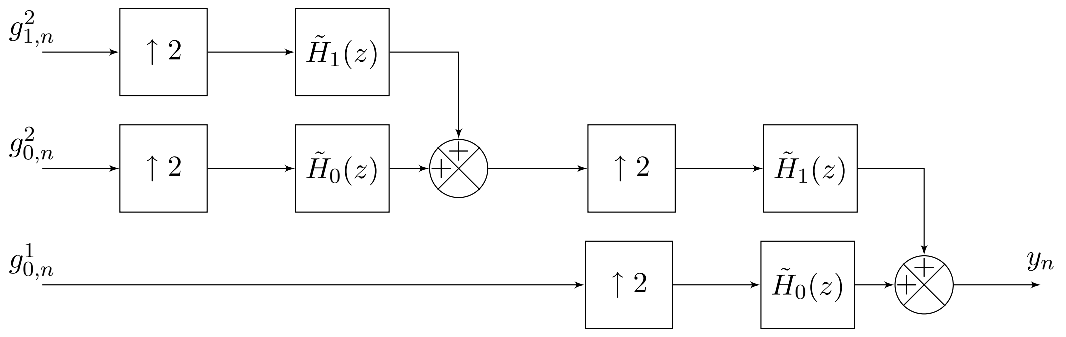

Figure 3.5: Two Tier Tree Structure Synthesis Bank

addition, non-uniform subband decompositions satisfying the PR condition can be realized with less effort; usually methods for creating arbitrary M channel PR filter banks rely on cosine modulation, and result in uniform decompositions [12]. Being able to transform small uniform decomposition filter banks into large non-uniform decompositions greatly relieves the computational burden of designing and realizing the desired system.

Chapter 4

Subband Adaptive Filtering

4.1 Background

Subband adaptive filtering is a well-developed concept that uses the resampling operations of a filter bank in order to lower the number of tap weights per adaptive filter. For example, in echo cancellation problems, the signal is restricted to either the bandwidth of human hearing, or the bandwidth of human speech. Thus the filter does not need to take into account the part of the spectrum outside of this area, since a pre-determined band-pass filter can easily cancel out any noise in that region. Thus appropriate filter banks can isolate this band, and adapt a filter $H(z)$ that operates solely on that band.

For illustration, consider a system with a sampling frequency of 80KHz where the signal of interest is in the audible range of 20-20000Hz. If an LMS-type algorithm of length M is used, the corresponding number of multiplications per iteration would be $2M$ [7]. Consider then, instead, if low- and high-pass filters of length L were used to decompose the signal in QMF form. Each of the resulting subband filters would be of length $M/2$. The total number of multiplications per iteration is then L per analysis and synthesis filter, a total of $4L$, and M per subband filter, a total of $2M$. This total is then divided by two since the system is now half-rate, for a total of $2L + M$ multiplications. Thus as long as $L < M/2$, computational savings can be realized. For the case of very large FIR adaptive filters, these savings are quite substantial. In this example, additional savings can be achieved if the high-pass filter branch is completely ignored. This is usually desired since all of the desired information is in the lower half of the spectrum.

4.2 Uniform Subband Adaptive Filtering

In uniform adaptive filtering, the subband decomposition is done in such a way that all the resulting subbands are of equal width. Therefore for an M band filter bank, each of the subband filters have length $\lceil N/M \rceil$, where N is the length of an equivalent fullband filter. These subband filters are updated with respect to the subband error signals $e_i[n] = d_i[n] - \hat{d}_i[n]$. Here $d_i[n]$ is the desired output of the i^{th} subband and $\hat{d}_i[n]$ is the output of the adaptive filter at the same band.

In subband adaptive filtering, both the input and desired output are passed through an analysis filter in order to ensure that all related systems are operating at the same rate. In general, the analysis filter for the input does not have to be identical to that of the desired output, since the adaption occurs post-decimation. As long as the analysis filter for the desired output and the synthesis filter satisfy the perfect reconstruction property, all that has to match are the rates [28].

Methods have been proposed to use this freedom of design to optimize the analysis filter for the input with respect to the MSE criterion, but here only the case where the two analysis banks are equivalent is considered [28]. This case was chosen since the subband adjustment algorithm presented is dependent on the input-output characteristics of the subbands of the unknown filter. Thus the power in each spectral region of the output must be compared to the power of the same spectral region of the input.

A variety of uniform subband algorithms have been proposed, both based on the LMS and the RLS concepts [5, 19, 21, 23, 26]. Originally, oversampled systems have been proposed in order to compensate for any information loss due to slight deviations from the PR condition. Recently, though, critically sampled systems have been of more interest as they allow for the minimum number of samples to be processed without loss of information. For the critically sampled systems, uniform band filters have been proposed for both the LMS and RLS algorithms. However, only LMS algorithms have been applied to non-uniform subband structures [20].

In critically sampled systems, there are two main disadvantages to using subband adaptive filtering. The first is an inherent increase in convergence time due to the adaptation taking place at a slower rate. The second is an increase in the MSE due to the lack of accounting for the aliased terms in the bands. This effect can bee seen by relating the response of the unknown filter through the analysis bank ($\vec{D}_{ideal}(z)$) to the response of the analysis bank through the adaptive filters $\hat{D}(z)$. Here $\vec{D}_{ideal}(z)$ is an Mx1 vector whose i^{th} entry is the i^{th} subband component of the ideal response. $\hat{D}(z)$ is the Mx1 vector containing the estimate of $\vec{D}_{ideal}(z)$ (the values directly before the synthesis filter). Defining $\mathbf{H}_{i,k}(z) = H_i(ze^{-\frac{j(k-1)2\pi}{M}})$ as the MxM matrix where $H_i(z)$ is the i^th analysis band filter, $\mathbf{G}(z)$ as the MxM subband transfer function matrix, $\mathbf{X}(z)$ as the MxM diagonal matrix with entries $\mathbf{X}_{k,k}(z) = X(ze^{-\frac{j(k-1)2\pi}{M}})$ where $X(z)$ is the unknown filter to be estimated and $\vec{U}_i(z) = U(ze^{-\frac{j(k-1)2\pi}{M}})$ as the Mx1 vector where $U(z)$ is the z-transform of the input signal, the condition that these responses should be equal is given by [6] as:

$$\mathbf{H}(z^{\frac{1}{M}})\mathbf{X}(z^{\frac{1}{M}})\vec{U}(z^{\frac{1}{M}}) = \mathbf{G}(z)\mathbf{H}(z^{\frac{1}{M}})\vec{U}(z^{\frac{1}{M}}) \tag{4.1}$$

In equation (4.1), $\mathbf{H}(z^{\frac{1}{M}})\mathbf{X}(z^{\frac{1}{M}})\vec{U}(z^{\frac{1}{M}})$ is the response ideal response $\vec{D}_{ideal}(z)$ and takes into account the aliasing present due to the decimation operation. Similarly, $\mathbf{G}(z)\mathbf{H}(z^{\frac{1}{M}})\vec{U}(z^{\frac{1}{M}})$ is the estimate of $\vec{D}_{ideal}(z)$, $\hat{D}(z)$, also including aliasing effects. By applying the transformation $z \to z^M$, the relationship $\mathbf{H}(z)\mathbf{X}(z) = \mathbf{G}(z^M)\mathbf{H}(z)$ follows directly. In using the PR condition, $\mathbf{F}(z) = z^L\mathbf{H}^{-1}(z)$ can be used to invert $\mathbf{H}(z)$ to obtain equation (4.2). $\mathbf{G}(z)$ can be expressed by [6]:

$$\mathbf{G}(z^M) = \mathbf{H}(z)\mathbf{X}(z)\mathbf{F}(z) \tag{4.2}$$

which is equivalent, element-wise, to:

$$G_{i,k}(z) = \sum_{l=1}^{M} H_i\left(ze^{\frac{2\pi j(l-1)}{N}}\right) X\left(ze^{\frac{2\pi j(l-1)}{N}}\right) F_l\left(ze^{\frac{2\pi j(k-1)}{N}}\right) \tag{4.3}$$

This shows that, in general, cross-terms between all subbands are required in order to perfectly model the unknown filter $X(z)$. More specifically, these terms depend on the products $H_i\left(ze^{\frac{2\pi j\omega l}{N}}\right) F_l\left(ze^{\frac{2\pi j\omega k}{N}}\right)$. In some special cases, such as ideal rectangular filters with no overlaps, these cross-terms are zero and cancel out. As an example, consider the two band case with $F_i(z) = H_i(z)$. Equation (4.2) can then be expressed as: [6]

$$\mathbf{G}(z^2) = \begin{bmatrix} H_0^2(z)X(z) + H_1^2(z)X(-z) & H_0(z)H_1(z)\left(X(z) + X(-z)\right) \\ H_0(z)H_1(z)\left(X(z) + X(-z)\right) & H_1^2(z)X(z) + H_0^2(z)X(-z) \end{bmatrix} \tag{4.4}$$

Here the off diagonal elements illustrate the dependency on the product of the filters $H_0(z)H_1(z)$.

For the LMS subband adaptive structure, the cost function is modified from that previously presented in section 2.2 to be the mean of the sum of the squares of error of all subbands:

$$J(e[n]) = E\left[\sum_{i=1}^{M} |e_i[M]|^2\right] \tag{4.5}$$

The error in each subband is defined as the output of the subband filter in that channel subtracted from the corresponding desired signal.

In the diagonal case, the subband filter structure is such that only one filter, $G_{i,i}(z)$, connects the output of the i^{th} analysis bank channel to the i^{th} synthesis bank channel, as shown in Figure 4.1. The transfer function matrix then has the form:

$$\mathbf{G}(z) = \begin{bmatrix} G_0(z) & 0 & 0 & \dots & 0 & 0 \\ 0 & G_1(z) & 0 & \dots & 0 & 0 \\ 0 & 0 & G_2(z) & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & G_{M-2}(z) & 0 \\ 0 & 0 & 0 & \dots & 0 & G_{M-1}(z) \end{bmatrix} \tag{4.6}$$

In this case the error $e_i[n]$ is only dependent on the corresponding filter $G_{i,i}(z)$. Thus, by taking the gradient with respect to those filter weights, all the other error terms are eliminated. The resulting update equations are then the same as the equations in section 2.2 for each channel, independently. The number of multiplications needed per iteration is calculated to be $3L + 2K/M$. This is derived from the

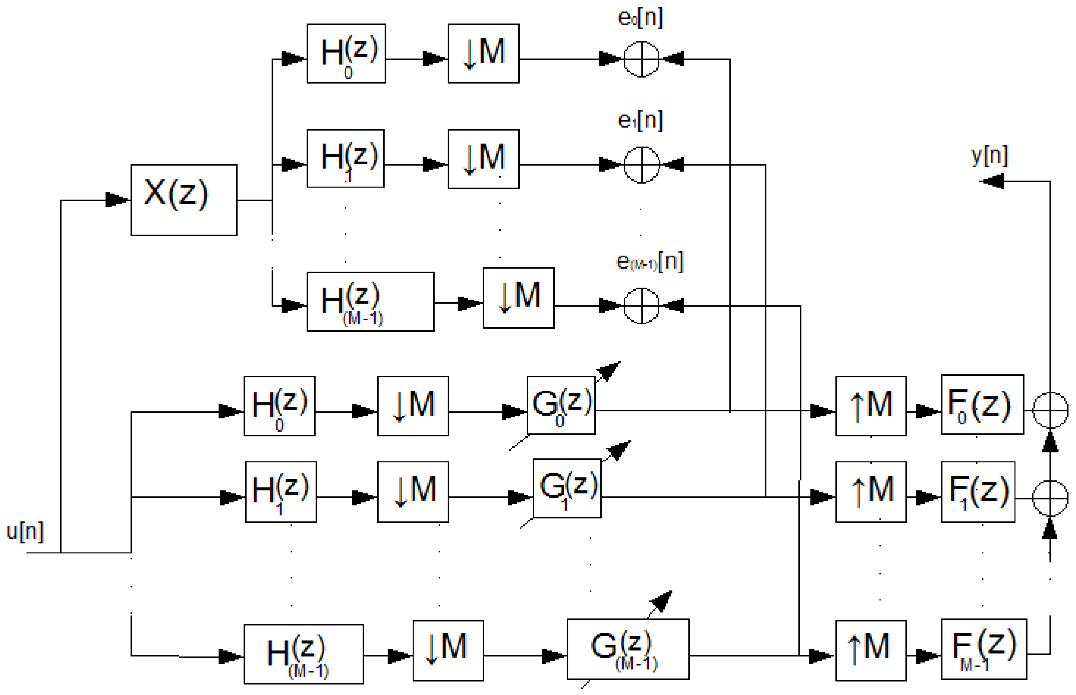

Figure 4.1: Diagonal Subband Filter Structure

$3LM$ multiplications needed to calculate the output of the synthesis and two analysis banks, and the $2KM/M$ multiplications it takes to update M LMS filters each of length K/M. When the factor of $1/M$ is accounted for (since the filter is run at $(1/M)^{th}$ the total rate), the total savings in computations is $2K(M-1)/M - 3L$ multiplies.

In the diagonal structure there is a lack of the cross-terms shown in equations (4.2) and (4.3) that allow for exact modeling of the system under general conditions. Therefore an alternate filter structure has been proposed where the transfer function matrix is not a diagonal matrix, but a tridiagonal matrix [6], [19]. As was shown in equation (4.2), the cross-terms that are required to adapt to an arbitrary unknown filter $X(\omega)$ are dependent on the products $H_i(z)F_j(z)$. The tridiagonal structure is motivated by the assumption that such products are zero for $|i-j| \geq 2$.Thus all terms aside from $G_{l,l}(z), G_{l,l-1}(z)$ and $G_{l,l+1}(z)$ in equation (4.3) become zero. The cross-terms are then introduced as shown in Figure 4.2; the corresponding transfer function matrix is:

$$\mathbf{G}(z) = \begin{bmatrix} G_0(z) & G_1(z) & 0 & 0 & \dots & 0 & 0 \\ G_0(z) & G_1(z) & G_2(z) & 0 & \dots & 0 & 0 \\ 0 & G_1(z) & G_2(z) & G_3(z) & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & G_{M-2}(z) & G_{M-1}(z) \\ 0 & 0 & 0 & 0 & \dots & G_{M-2}(z) & G_{M-1}(z) \end{bmatrix} \tag{4.7}$$

This structure uses a larger analysis bank, with M channels filtered by $H_i^2(z)$ and $M-1$ channels representing the transition regions, filtered by $H_i(z)H_j(z)$. The adaptive filters are then defined by $G_{i,j}(z) = G_{2i-j}(z)$ along the three main diagonals. The cross-terms having the analysis filter banks $H_i(z)H_j(z)$ are motivated by equation (4.3) in the special case where $F_i(z) = H_i(z)$. The resulting number of computations are higher because, in addition to the filtering operations performed in the diagonal design, each analysis filter for the input is twice as long and there are extra filtering operation to filter the off diagonal terms.

When minimizing the cost function in the tridiagonal structure, not all the error terms cancel out. This leads to a very large increase in the number of computations for the update of the subband filter tap weights. The update equations:

$$E_k[n] = d_k[n] - \vec{X}^H_{k,k}[n]\vec{G}_k[n] - \vec{X}^H_{k-1,k}\vec{G}_{k-1}[n] - \vec{X}^H_{k,k+1}\vec{G}_{k+1}[n] \quad (4.8)$$

$$\vec{G}_k[n+1] = \vec{G}_k[n] + \mu_k(\vec{X}_{k,k}E^*_k[n] + \vec{X}_{k-1,k}E^*_{k-1}[n] + \vec{X}_{k,+1k}E^*_{k+1}[n]) \quad (4.9)$$

reflect this fact, as they are no longer simply a function of the data vector associated with that particular subband [19]. In equations (4.8) and (4.9), $\vec{X}^H_{k,l}$ is the filtered input $\vec{u}_i$ filtered by the cascaded filter $H_k(z)H_l(z)$. Thus $k = l$ imply the main subbands, while $k \neq l$ are the cross-terms introduced into the system. A full derivation of the tridiagonal update equations for the case of real inputs and filter coefficients can be found in [19].

In terms of the computational complexity, the filtering in the analysis and synthesis banks increases to $LM + 2L(M-1)$ computations per M iterations. The LM term comes from the extra filtering in the $H^2_i(z)$ in the analysis bank for the input, and the $2L(M-1)/M$ comes from the $M-1$ cross-term filters of length $2L$. The biggest increase, though, comes from the tripling of the computations needed to update the LMS filters. In the diagonal LMS subband structure, each branch entailed $2K$ computations. In the diagonal case, accounting for the cross-terms adds two more terms to each equation, tripling the computations to $6K$. The total computational complexity then is $(3LM + 6K)/M$.

The same two topologies that were applied to the LMS subband filtering have been applied to the RLS algorithm. In the case of the subband RLS algorithm, the cost function to minimize is the sum of the least squares errors:

$$J(e_i[n]) = \sum_{i=0}^{M-1} \sum_{l=0}^{n} \lambda_i^{n-l} |e_i[n]|^2 \quad (4.10)$$

This is similar to solving M simultaneous least squares problems.

For the diagonal case, as with the LMS algorithm, the error in the i^{th} band is dependent only on $G_i(z)$. Therefore the differentiation results in solely that term, and the algorithm is identical to that in section 2.3 for each subband independently.

In the tridiagonal case the error is again dependent on both the cross-terms from the neighboring bands. The full derivation of this algorithm by Alves and Petralgia can be found in [23], and results in the following update equations:

$$\vec{k}[n] = \mathbf{P}[n-1]\chi[n]\left[\lambda\mathbf{I} + \chi^H[n]\mathbf{P}[n-1]\chi[n]\right]^{-1} \tag{4.11}$$

$$\mathbf{P}[n] = \lambda^{-1}\left[\mathbf{I} + \vec{k}[n]\chi[n]\right]\mathbf{P}[n-1] \tag{4.12}$$

$$\vec{\rho}[n] = \lambda\vec{\rho}[n-1] + \chi[n]\vec{d}[n] \tag{4.13}$$

$$\mathbf{G}[n] = \mathbf{P}[n]\vec{\rho}[n] \tag{4.14}$$

where $\chi[n]$ is a tridiagonal matrix consisting of the data. Specifically, $\chi_{i,k}[n] = u_{i,k}[n]$ for $|i-k| \leq 1$ and zero otherwise. The total computations required for the tridiagonal subband RLS filter is given by [23] and is shown in Table 4.2.

Algorithm	Fullband	Diagonal Uniform Subband
LMS	$2K$	$(3LM + 2K)/M$
RLS	$2K^2 + 5K$	$(3LM^2 + 3K^2 + 2KM)/M^2$

Table 4.1: Number of Computations for Various Adaptive Algorithms

Algorithm	Number of Computations
LMS	$(3LM + 6K)/M$
RLS	$K(3M + M^3 + 3M^2 + 2) + M^2 + 6LM - LM$

Table 4.2: Number of Computations for Tridiagonal Subband Adaptive Algorithms

4.3 Non-Uniform Subband Adaptive Filtering

In the non-uniform subband case, all of the decimation factors are unequal, as shown in Figure 4.3. This means that different adaptive filters will be run at different sampling rates, and over different spectral regions. Therefore the subband filters themselves are of different lengths. As in the uniform subband case, both diagonal and tridiagonal forms have been proposed for this subband decomposition. In the diagonal case, taking the gradient with respect to the i^{th} filter results in loss of all other terms except those involving $G_{i,i}(z)$.

In the tridiagonal case, however, the structure becomes more complicated than in the uniform case. This is because branches running at different rates are added together. Thus delays have to be placed within the cross-term connections. Further details can be found in [20].

The motivation for the non-uniform subband decomposition is the ability of a large subband to account for all internal cross-terms. Specifically, if one subband spans the spectral region which a uniform subband decomposition would require multiple subbands to span, the cross-terms between all those subbands would be accounted for. The cross-terms between any two non-uniform subbands still need to be accounted for, hence the derivation of the tridiagonal non-uniform subband filters [20]. If, however, the filtered signal $U(z)H_i(z)$ contains no power, no aliased terms can exist and the cross-terms disappear. Thus cross-terms are not necessary where there is no signal power, an idea that will be made use of when designing the subband allocation algorithm in section 5.3.

With respect to the number of computations per iteration, the difference in the rates at which all the filters are running causes two different increases. Firstly, the branches with the smaller decimation factors have larger adaptive filter lengths. This is because those branches cover larger spectral regions. Secondly, for every M iterations, the largest adaptive filters (those in the branches with the smallest decimation factors) are updated most often. The resulting computational complexities for such algorithms are detailed in Table 4.3.

Algorithm	Number of Computations
LMS	$\sum_{i=1}^{N}\left(\frac{3L_i}{M_i}+\frac{2K_i}{M_i^2}\right)$
RLS	$\sum_{i=1}^{N}\left(\frac{3L_i}{M_i}+\frac{3K_i^2}{M_i^3}+\frac{2K_i}{M_i^2}\right)$

Table 4.3: Number of Computations for Non-Uniform Subband Adaptive Algorithms

4.4 Properties of Subband Adaptive Filters

Regarding performance, subband adaptive filters suffer from both a longer convergence time and a higher misadjustment. The reason for the slower convergence rate is the slower rates at which the subsystems are running. If for a given signal to noise ratio and unknown system the fullband LMS algorithm takes, say, a hundred iterations to converge, running the system at half rate will mean that the entire system converges in approximately two hundred iterations. This has been demonstrated for both the RLS and LMS cases [21], [23]. The lower misadjustment has been attributed to the lack of cross-terms between bands [22], [17]. Although the cross-terms introduced in the tridiagonal filter bank structure seek to mitigate this, the decrease in the steady-state error is not justified by the amount of extra computations needed [22].

The non-uniform subband decomposition is able to achieve better performance than the uniform decomposition if the correct decomposition is chosen. In order to compensate for the cross-terms where necessary, larger subbands can be used over a spectral region. These larger subbands have the added benefit of increasing the convergence rate, since those bands are run at higher rates. In addition, small subbands can be chosen over areas where there is no signal, thereby decreasing the overall computational complexity. Taking full advantage of these benefits, though, requires knowledge of the signal/system pair since these subbands must be set up prior to any computations. In order to utilize the non-uniform subband decomposition more effectively for an arbitrary system, it would be necessary to be able to change the subband widths to suit the input-output characteristics of the unknown filter.

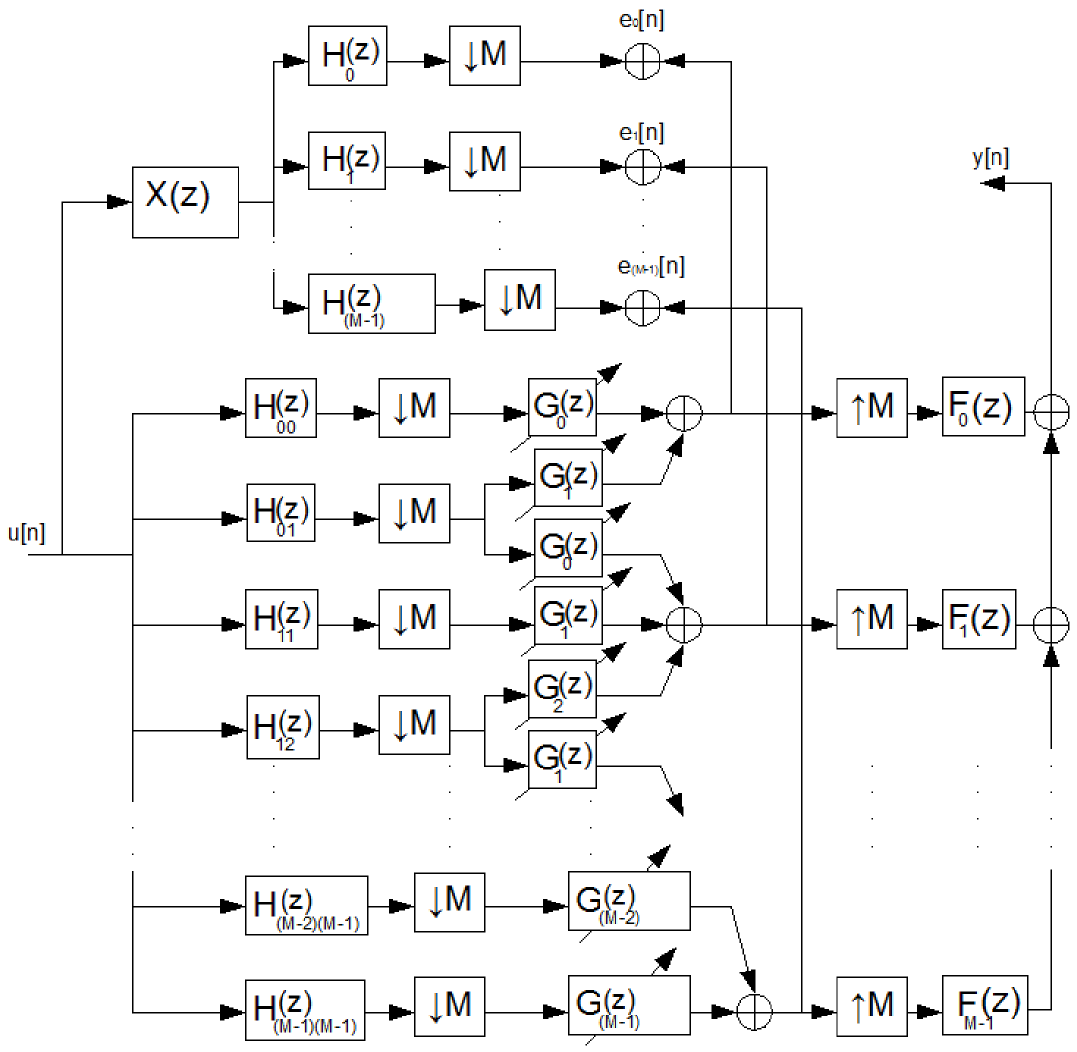

Figure 4.2: Tridiagonal Subband Filter Structure

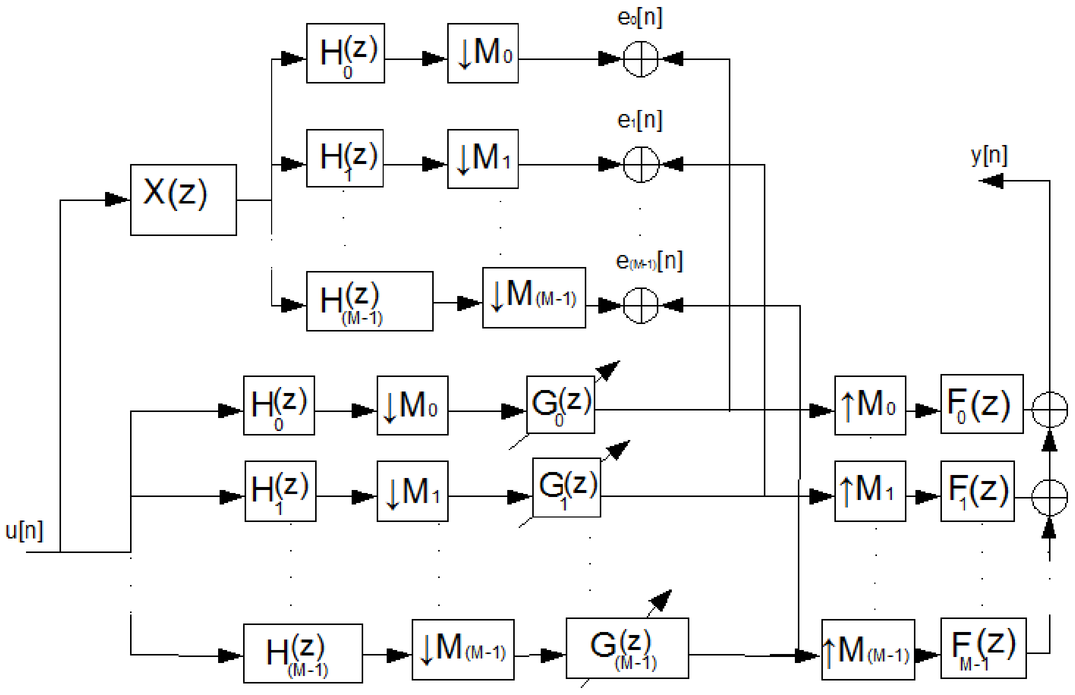

Figure 4.3: Non-Uniform Diagonal Subband Filter Structure

Chapter 5

Adjustable Subband Adaptive Filtering

5.1 Motivation for Adjustable Subband Filter Banks

Both uniform and non-uniform subband adaptive filtering reduce the number of total computations needed to adapt the filter at each time increment. The issue with each is that they are highly dependent on the unknown systems transfer function properties. Non-uniform subband designs are more desirable because they take all the cross-terms in an entire band into account, while simultaneously reducing the number of computations in the spectral regions of less interest. The problem that arises, then, is that the input-output characteristics are not always known a-priori. Therefore, it would be advantageous to have an algorithm which can determine these characteristics, and adjust the filter bank bands to merge those two bands, thus accounting for cross-terms between those spectral bands.

5.2 Previous Work

Previous work in the area of adjustable subbands for use in non-uniform subband adaptive filters has mostly focused on the use of oversampled systems. An algorithm proposed by Grieshbach and Etter sought to adjust the subband decomposition in order to minimize the mean squared error [8]; the focus of the algorithm was to isolate transition bands of the unknown filter to reduce the misadjustment.

Following this work, a structure to ease the implementation of such algorithms was proposed by Oakman and Naylor [17]. This structure utilizes a tree bank similar to the one proposed here, with a similar initialization procedure. The difference, however, is that the structure they propose keeps unused branches, simply moving out the filters past the QMF branches. This means that the initial analysis bank remains the same throughout the duration of computations, while the synthesis bank is the one that is adapted.

The method used is to combine the two bands that need to be merged by moving up the synthesis filters associated with those bands. Due to the PR condition imposed on each QMF branch, the reduced branch effectively becomes a delay. The filter placed at the output of the combined branch is then initialized as:

$$G_{01}(z) = F_0(z)C_0(z^2) + F_1(z)C_1(z^2) \tag{5.1}$$

The result is a framework to adjust subbands which was efficient in the sense that minimal operations had to be performed to change the subband widths [17]. The disadvantage to this structure is that the data is still run through the analysis and synthesis filters of the merged bands, resulting in unnecessary computations. It would be advantageous to be able to change the structure as proposed, but with greater care taken to avoid extraneous computations.

5.3 Proposed Algorithm

The algorithm proposed here utilizes a structure similar to that proposed by Oakman and Naylor. QMF tree structured filter banks allow for easier design of PR filter banks as well as ease of merging in the adaptive analysis and synthesis banks. Instead of merging bands by moving up the synthesis banks, when two bands are determined to need to be merged the entire branch is replaced by an effective filter. This requires a different initialization process than that proposed in [17].

In addition a decision algorithm is proposed to determine when bands should be merged or split. This decision algorithm is based on the ratio of the power spectrum densities of the input and the desired response. Thus any dependence on the error signals, and thus the adaptive filter tap weights, is removed. The power spectra are estimated using the Welch method [27], in which windowed FFT averages over time are used to approximate the power spectra.

Subband Decomposition

In choosing the structure for the analysis and synthesis banks, a QMF based tree topology was chosen. The tree topology allows for both the ease of the adjustment of the subband bandwidths as well as the determination of a two bank PR filter pair. For this system the only perfect reconstruction filters that need to be designed are one pair of high-pass and low-pass filters. These filters would be used in every embedded QMF bank, producing a maximum of 2^Y subbands, where Y is the number of levels in the analysis or synthesis banks.

The structure is set up by having the embedded QMF banks set up to be able to calculate the maximal number of coefficients. The adaptive filters then connect the nodes of the analysis filter with the corresponding node in the synthesis bank. When merging two bands, the strategy is to disconnect the entire embedded QMF filter bank and to connect in its place an adaptive filter, as shown in Figure 5.1. In this way any all-pass substructures are not performing any multiplications, thereby saving computations.

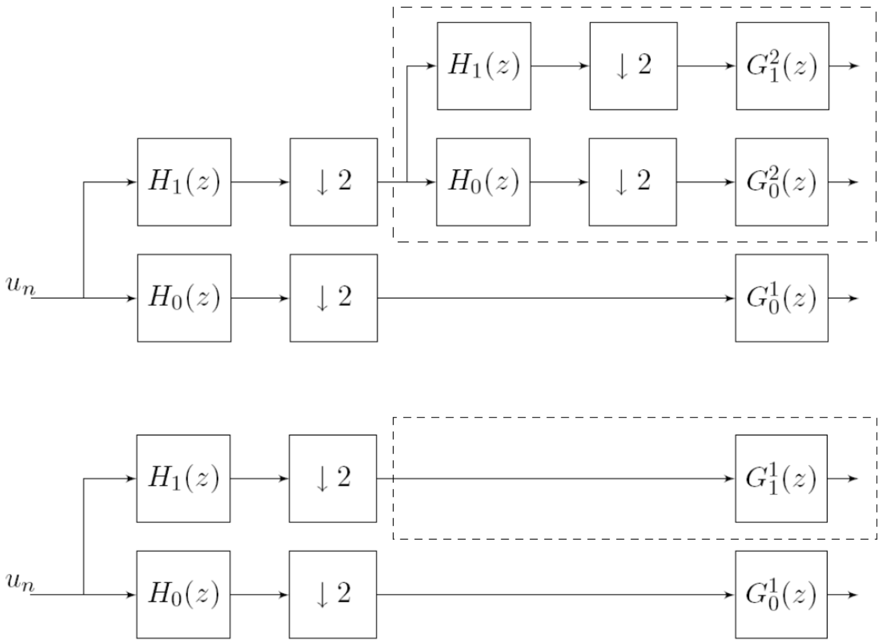

Figure 5.1: Subband Merging by Replacement

The strategy of splitting one subband into two smaller ones of half the original spectral length is to disconnect that adaptive filter, and to connect the next level embedded QMF filter bank with the associated half length adaptive filters. This will split the subband in two, reducing the computations needed to update that spectral band.

Initialization of Adjusted Filters

When merging a QMF branch, the resulting transfer function needs to be determined from the input-output characteristics of the branch. The output of a QMF branch, including aliasing, is given by:

$$\begin{aligned} Y(\omega) = \; & \tfrac{1}{2}(F_0(\omega)[G_0^M(2\omega) + G_0^M(2\omega - pi)][H_0(\omega)U(\omega) + H_0(\omega-\pi)U(\omega-\pi)] + \\ & F_1(\omega)[G_1^M(2\omega) + G_1^M(2\omega-\pi)][H_1(\omega)U(\omega) + H_1(\omega-\pi)U(\omega-\pi)]) \end{aligned} \tag{5.2}$$

Since F_0, F_1, H_0 and H_1 abide by the PR condition as previously mentioned in section 3.2, the aliased terms drop out, thus allowing for an equivalent transfer function $G_0^{2M}(\omega) = Y(\omega)/U(\omega)$ given by:

$$\begin{aligned} G_0^{2M}(\omega) = \; & \tfrac{1}{2}(F_0(\omega)[G_0^M(2\omega) + G_0^M(2\omega-\pi)]H_0(\omega) \\ & + F_1(\omega)[G_1^M(2\omega) + G_1^M(2\omega-\pi)]H_1(\omega)) \end{aligned} \tag{5.3}$$

In order to approximate the resulting equivalent transfer function, the impulse responses of G_0^M, G_1^M, F_0, F_1, H_0 and H_1 are calculated. This brings into question the relative sizes of the filters. G_0^{2M} is twice the length of G_0^M and G_1^M, each of which have length L. F_0, F_1, H_0 and H_1 have length K, and are not necessarily the same as length as G_0^M and G_1^M, or a multiple thereof. This discrepancy is compensated for by using $F_{0,\text{eff}}$, $F_{1,\text{eff}}$, $H_{0,\text{eff}}$ and $H_{1,\text{eff}}$, as calculated by the series of interpolations and decimations:

$$H_{0,\text{eff}} = \left(\downarrow K'\right)(Z(\omega))\left(\uparrow L'\right)(H_0) \tag{5.4}$$

The interpolation by L' and decimation by K'serve to change the length of the FFTs calculated for the $H(z)$'s and $F(z)$'s to the length of the FFT of $G_0^{2M}(z)$. The values of L' and K' are respectively:

$$L' = \frac{2L}{\text{GCF}(L,K)} \tag{5.5}$$

$$K' = \frac{K}{\text{GCF}(L,K)} \tag{5.6}$$

where $\text{GCF}(L,K)$ is the greatest common factor of L and K. The low-pass filter $Z(\omega)$, serves to smooth the zero filled interpolation.

Since the F_0, F_1, H_0, H_1, K, and L are all known prior to the start of the algorithm, all the needed approximations to the analysis and syntheses filters of length 2^iK for $1 \leq i \leq \log_2(M)$ can be pre-computed. Thus no time needs to be spent calculating this information while the algorithm is running.

When decomposing a channel into two smaller subbands, the tap weights are initialized to zeros and no extraneous calculations need to be performed. This is because the proposed algorithm splits a subband when there is minimal power in that spectrum. Thus the total contribution of that band to the total output is minimal.

The only other data that needs to be initialized is the data stored in the delays of the subband filters. This can be taken into account by storing the past 2^iK inputs for the filter in the i^{th} level filters. Saving data in this fashion does not take any multiplications, and the data can then be recalled easily into the filter, preventing the need to re-converge after high errors due to starting with zeros or other data.

Power Spectral Density Estimation

In order to adjust the widths of the subbands, the spectral locations of the signal power needs to be determined. This requires knowledge of the power spectrum density, defined as the Fourier transform of the autocorrelation function of a wide sense stationary (WSS) process. By definition, the autocorrelation of a WSS process only depends on the distance in time between any two samples.

For an arbitrary process, wide sense stationarity can be assumed over short periods of time, thus allowing for estimates of the PSD. Methods to estimate the PSD over a time span usually center around the use of periodograms, or the squared magnitude of the discrete fourier transform (DFT). One such method, using the magnitude of the FFT values squared over a moving window, was proposed by Welch in [27]. In Welch's PSD estimation, $\hat{P}[k]$, the estimate is given by:

$$\hat{P}[k] = \frac{L}{UK} \sum_{j=1}^{K} |A_j[k]|^2 \tag{5.7}$$

where $A_j[k]$ is the FFT of the k^{th} windowed time frame:

$$A_j[k] = \sum_{n=0}^{L-1} x_j[n] w[n] e^{-2j\pi k \frac{n}{L}} \tag{5.8}$$

and U is given by:

$$U = \frac{1}{L} \sum_{n=0}^{L-1} w^2[n] \tag{5.9}$$

In equation (5.7) there are two degrees of freedom: the windowing function and the number of FFT samples to average. In the PSD estimate used in the subband adjustment algorithm, the number of terms to average is left up to the user as an input. Thus for more stable conditions longer averages can be taken, while for less stable conditions the number of terms to be averaged may be decreased to allow for quicker responses to changing conditions. The windowing function was chosen to be a simple rectangular window to avoid both extraneous computations in calculating the dot product between the input and the window as well as spreading the spectrum due to convolution with the window's frequency response.

For the FFT operations, the number of points used was determined to be twice the maximum number subbands allowed. This quantity is set at the start of the algorithm. This number was chosen because the desired resolution was exactly one point per minimum size subband.

Decision Algorithm

The proposed decision algorithm was designed to change the subband decomposition in order to create large subbands in spectral regions where the signal is present, and small subbands in the spectral regions where the signal is minimal. The large subbands in the signal region minimize the convergence time as well as account for cross-terms. Smaller subbands in the low power region allows savings in the number of computations as described in section 4.3.

The algorithm makes the decisions on which subbands to merge and which subbands to decompose based on the ratio of the power spectra of the input and desired output. In this way the subband decomposition is dependent only on the input-output characteristics of the potentially changing unknown filter, and not on the error signals. This removes any dependence of the algorithm on the current tap weights of the adaptive filters. The decision itself is based on comparing this ratio against set threshold values γ_d and γ_u.

In this implementation, no windowing function was used so that there would be no associated computations. The PSD estimates can then be calculated with a total number of $4M^2 \log(2M)$ computations by taking the FFT of the $2M$ blocks of input and desired output data, and squaring the results. Blocks of length $2M$ were used in order to have the correct spectral resolution of bands of π/M. The result is then averaged with P past calculations, which are stored in memory, to produce the PSD estimate. The algorithm then calculated the ratio of the power spectra estimates. This is done in order to place more weight on the bands where the output power is much greater than the input power. In order to avoid artifacts of having small values of the PSD estimates, regularization terms δ_{in} and δ_{out} are introduced in the PSD ration formula:

$$PSD_r[k] = \frac{PSD_{out}[k] + \delta_{out}}{PSD_{in}[k] + \delta_{in}} \tag{5.10}$$

In particular, δ_{in} prevents the ratio from becoming excessively large for small input power. Although δ_{out} is not necessary for this purpose, it serves to guide the choice of the lower threshold γ_d. Once the algorithm calculates the ratio, the thresholding is applied as described in the pseudocode below.

```
for index goes from 1 to current number of subbands
    current band power <-- Average of all PSD ratios in that band
    if branch test = 0 & Not at last level
        band test <-- 0 if both bands are in the same branch
        if current subband = next subband & band test = 0
            Change branch test to 1
        else
            if Band power < Lower threshold & Not at last level
                Decompose band into 2 bands
            else
                Leave band the same size
            end
        end
    else if branch test = 0 & At last band
        if Band power < Lower threshold & Not at last level
            Decompose into 2 bands
        else
            Leave band the same size
        end
    else if branch test = 1
        if Average of both Branch's power > Upper threshold & Not at first level
            Consolidate branch and initialize filter
        else if Both band power ratios < Lower threshold & Not at last level
            Decompose both branches
        else if Only previous band power ratio < lower threshold & Not at last level
            Decompose last branch
        else if Only current band power ratio < lower threshold & Not at last level
            Decompose current
        else
            Leave both bands Alone
        end
        branch test <-- 0
    end
end
```

First the algorithm checks to see if the two channels that are being tested are part of the same embedded QMF bank. If it is, then the power ratio is compared to the preset thresholding values γ_u and γ_d as shown in Figure 5.2. If the average of the powers in both bands is greater then the upper thresholding value γ_u, then the two bands are combined. This accounts for the cases that one band has a very large power ratio while the other might have only a small ratio. In this case, it is possible that cross-terms between these bands need to be accounted for so when the bands are combined, the new effective filter is calculated and used in stead of that embedded QMF bank.

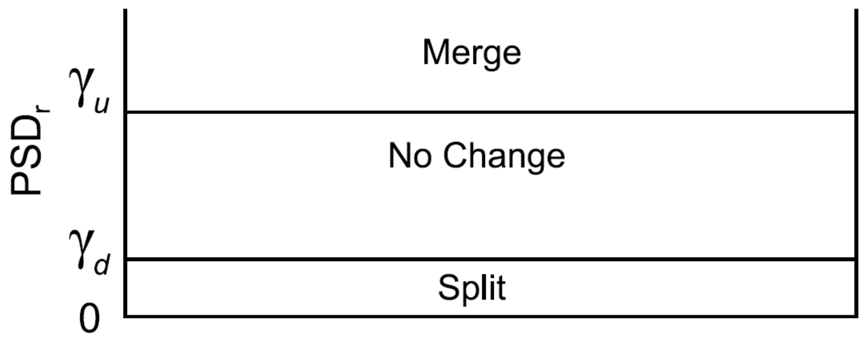

Figure 5.2: Thresholding Scheme

If the average power does not meet the previous condition, then the power ratios are compared to the lower thresholding value γ_d. If either of the power ratios is below this threshold, the band is split into two smaller bands by inserting another embedded QMF bank. Otherwise that subband is left alone. If a filter is not part of a pair that are in the same QMF bank, then the power ratio is only tested against the lower threshold. This is because the merging algorithm only replaces an entire embedded QMF bank. Thus if there are QMF banks embedded within the QMF bank in question, the merging cannot take place.

The threshold parameters γ_u and γ_d are chosen by the user in order to give more credence to either the computational complexity, or to the performance. Since γ_d determines when a band is split, a high value would cause the decision algorithm to favor smaller subbands. If δ_{out} is set to a number greater then zero, the PSD ratio floor is raised to δ_{out}/δ_{in}. Thus a choice of $\gamma_d \leq \delta_{out}/\delta_{in}$ would cause the decision algorithm to never split any subbands. γ_u, on the other hand, controls the bias the algorithm has towards merging bands. A low value would result in more merging, tending to higher rate systems and increasing performance at the cost of the computational complexity. A high value of γ_u would merge bands less often. Thus these choices are highly dependent on the desired performance and computational complexity in any implementation.

This decision algorithm is independent of the adaptive filtering itself, and can run every D time steps, as desired. Therefore the computational complexity of the PSD update is $4M^2 \log(2M)/D$ multiplications per iteration plus another M/D multiplications per iteration to calculate the ratio. The merging of the subbands is not guaranteed to occur at every iteration, but as a worst case scenario $3K$ computations will be needed to merge every pair of subbands during the same update. Therefore the rate at which this process updates the subbands mitigates any extra computations required by the update process. For rapidly changing systems, a value of $D \approx 2M$ may be more beneficial, while for slowly changing systems, the update rate for the subbands should be much higher.

Chapter 6

Experimental Results

6.1 Construction of a PR Filter Bank

In order to test the subband filter schemes, first a PR filter bank had to be constructed. Since the tree structure was used, relatively small FIR high pass and low pass filters on the order of approximately 50 tap weights were to be constructed. In order to minimize the energy in the stop-band, the FIR filters were designed using a Kaiser window. The design and analysis of the filters were implemented in MATLAB. Table 6.1 shows the parameters used to design the initial filters and Table 6.2 shows the coefficients of the analysis bank high pass and low pass filters. The frequency values in Table 6.2 are on a digital normalized frequency scale with 1 corresponding to the Nyquist bandwidth.

Parameter	Low Pass Filter H_0	High Pass Filter H_1
Pass Band Frequency	0.4500	0.4150
Stop Band Frequency	0.5850	0.5500
Pass Band Attenuation (dB)	0.1	0.1
Stop Band Attenuation (dB)	25	25

Table 6.1: QMF Perfect Reconstruction Filter Parameters

Filter:	Low Pass Filter H_0	High Pass Filter H_1
h_1	0.000425977502231	0.001307147423637
h_2	-0.002210129015354	0.001100291098292
h_3	-0.000565771466616	-0.002577044944605
h_4	0.004342836573103	-0.002476563925073
h_5	0.000385002046507	0.004216891571267
h_6	-0.007441304015882	0.004770373908285
h_7	0.000407183882700	-0.006173003560034
h_8	0.011729878415436	-0.008343959895693
h_9	-0.002224774388358	0.008346972383157
h_{10}	-0.017526499242410	0.013707933967288
h_{11}	0.005697574132323	-0.010602132788670
h_{12}	0.025414436775626	-0.021702329126364
h_{13}	-0.011939322732788	0.012775965919655
h_{14}	-0.036745298923579	0.034017831704410
h_{15}	0.023445924499667	-0.014697052292606
h_{16}	0.055479593540590	-0.054999847943743
h_{17}	-0.048143409201472	0.016204391880264
h_{18}	-0.098810775045496	0.100694598915520
h_{19}	0.135948617810551	-0.017166433321365
h_{20}	0.462330258853221	-0.316425588177542
h_{21}	0.462330258853221	0.517414076509360
h_{22}	0.135948617810551	-0.316425588177542
h_{23}	-0.098810775045496	-0.017166433321365
h_{24}	-0.048143409201472	0.100694598915520
h_{25}	0.055479593540590	0.016204391880264
h_{26}	0.023445924499667	-0.054999847943743
h_{27}	-0.036745298923579	-0.014697052292606
h_{28}	-0.011939322732788	0.034017831704410
h_{29}	0.025414436775626	0.012775965919655
h_{30}	0.005697574132323	-0.021702329126364
h_{31}	-0.017526499242410	-0.010602132788670
h_{32}	-0.002224774388358	0.013707933967288
h_{33}	0.011729878415436	0.008346972383157
h_{34}	0.000407183882700	-0.008343959895693
h_{35}	-0.007441304015882	-0.006173003560034
h_{36}	0.000385002046507	0.004770373908285
h_{37}	0.004342836573103	0.004216891571267
h_{38}	-0.000565771466616	-0.002476563925073
h_{39}	-0.002210129015354	-0.002577044944605
h_{40}	0.000425977502231	0.001100291098292
h_{41}	0	0.001307147423637

Table 6.2: QMF Perfect Reconstruction Filter Coefficients

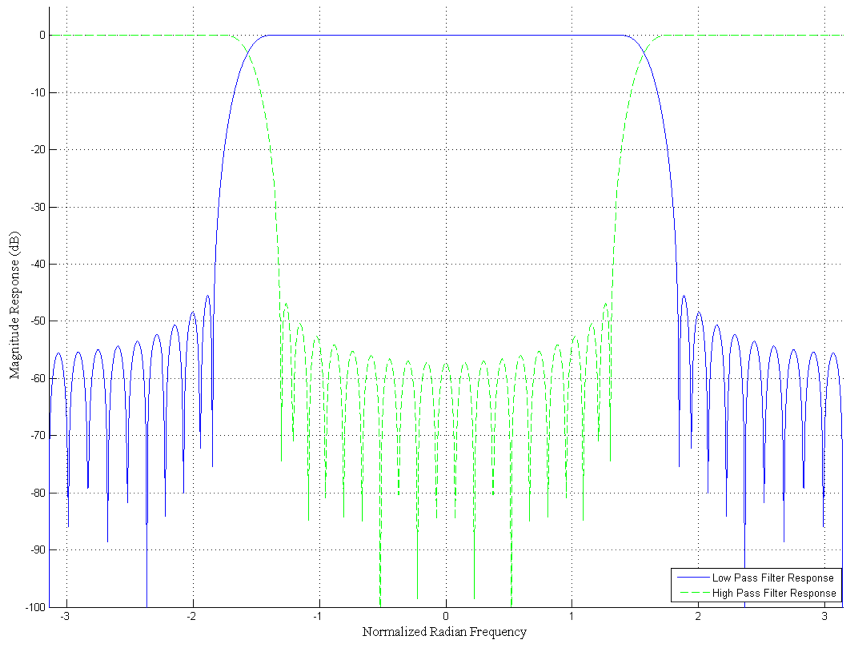

Figure 6.1: Magnitude Response of High Pass and Low Pass Filters

6.2 Consolidation of Tree Branches

Following the details of section 5.3, code was constructed to implement equations (5.3) and (5.4). Figure 6.2 shows a trial run where white noise was passed through both the QMF branch as well as the calculated effective filter. The plot shows that the spectral characteristics of the outputs of both filters are essentially the same. Features such as the peaks, bandwidth, and gain are comparable. The code may be found in Appendix A.

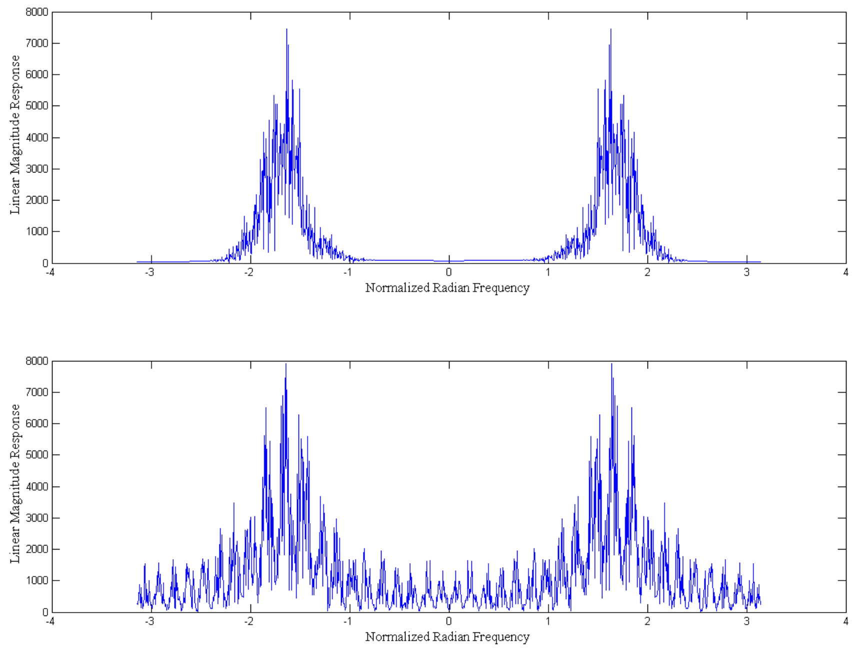

Figure 6.2: Effective Filter Comparison. Top: Output spectrum from QMF filter. Bottom: Output spectrum from effective filter.

6.3 Performance of Uniform Subband Adaptive Filters

Both the RLS and LMS versions of the uniform subband filter structure were implemented for comparison. The adaptive algorithms were tested against a series of filters. The filters and their properties are shown in Table 6.3.

Figure 6.3 shows the comparison between the performance of the fullband LMS and the subband LMS algorithm. White noise with variance $\sigma_u^2 = 0.36$ was used as an input to the low-pass filter of length 236 for 4000 iterations. At iteration 4000, the output was changed to that of the high-pass filter with length 239. The subband decomposition of the LMS algorithm was four uniform subbands, and the step size for both fullband and subband algorithms was chosen to be $\mu = 0.001$. The total number of tap weights to be distributed amongst the subbands was chosen to be $M\text{ceil}(239/M) = 240$ This overestimated the number of tap weights needed so that any changes in the MSE were from the properties of the subband structure and not of lack of tap weights in the adaptive filters. Figure 6.3 demonstrates both the increase in the MSE, a difference of approximately 25dB, as well as a faster convergence rate. It is important to note that the subband filter has an added delay from the analysis and filter banks, as can be observed in Figure 6.3.

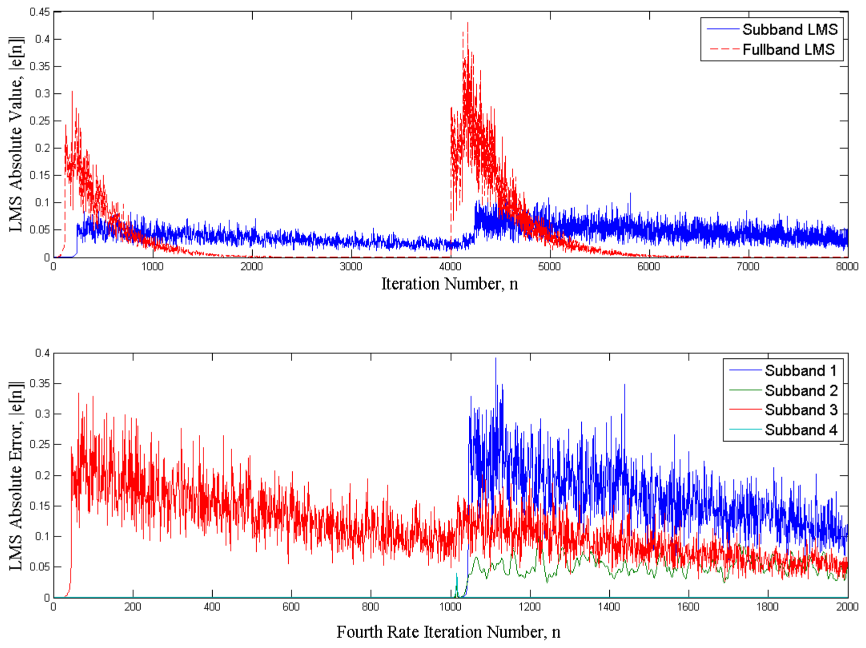

Figure 6.3: Comparison of Fullband LMS Filter to 4 Band Subband LMS

Figure 6.4 shows the relative performance of the 16 band RLS subband structure to the fullband RLS structure. The two filters were run with a white test signal having a variance $\sigma_u^2 = 0.36$ through the high-pass filter of length 239 for 16000 iterations, followed by the low-pass filter of length 236 for another 16000 iterations.

The results of 10 trials were averaged to obtain the error performance curves shown. Both RLS algorithms were initialized with an inverse covariance matrix $\mathbf{P}_0 = (0.9)^{-1}\mathbf{I}$ and forgetting factor $\lambda = 0.99$. No additive white noise was inserted into the desired output during this run in order to better show the response of the filters. Again the number of tap weights to be distributed was chosen by $M\text{ceil}(239/M) = 240$. The error performance curve in Figure 6.4 shows that the fullband RLS structure converges much faster, within 1000 iterations, than the subband structure, which converges in about 3000 iterations. In addition, the steady state MSE for the fullband RLS filter is approximately 10dB less then the subband structure. These results agree with previously published findings [23].

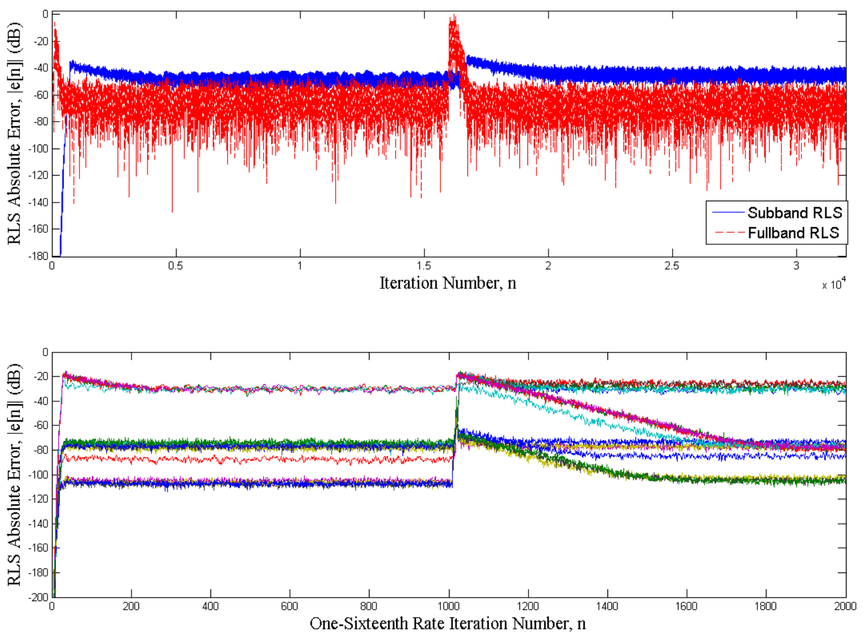

Figure 6.4: Comparison of Fullband RLS Filter to 16 Band Subband RLS. Top: Total Error in Each Filter. Bottom: Error in the RLS Subband Structure.

6.4 Performance of Non-Uniform Subband Adaptive Filters

To test the effects of using various non-uniform subband decompositions, the adaptive filter was tested against the low-pass filter. The maximum decimation factor used here was 8. $[2, 8, 8, 8, 8]$ was chosen as the optimal choice of decimation factors, and $[8, 8, 8, 8, 2]$ was chosen as the suboptimal decimation factors. The algorithm was run with a step size of $\mu = 0.01$ for 11000 iterations and averaged over 10 trials. The signal and noise variances were again $\sigma_u^2 = 0.36$ and $\sigma_\nu^2 = 0$ to better show the characteristics of the algorithm. The results in Figure 6.5 show that using the optimal decimation factors both lowered the misadjustment and increased the convergence rate.

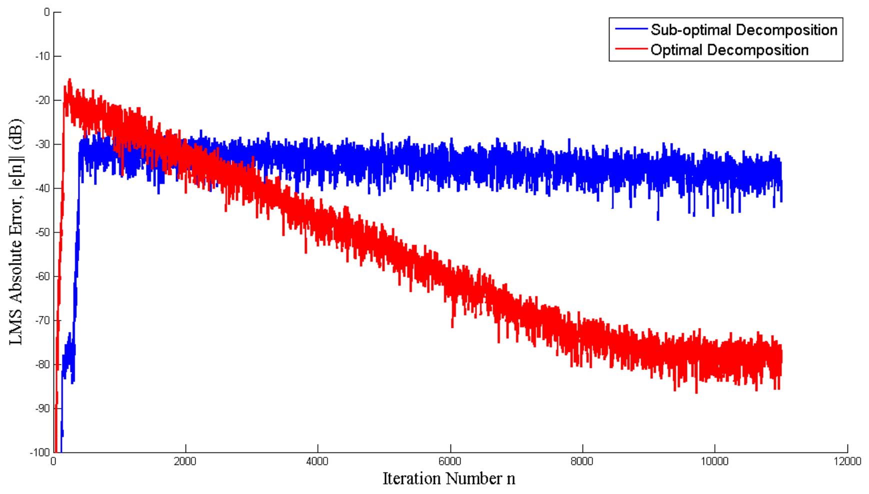

Figure 6.5: Convergence of Non-Uniform Subband Filters

The steady state error was decreased by approximately 40 dB, and the convergence rate was increased to approximately -0.0075dB per iteration. Figure 6.5 illustrates the importance of choosing the correct subband decomposition when using non-uniform filter banks.

6.5 Performance of the Adjustable Subband Algorithm

The proposed algorithm from section 5.3 was implemented in MATLAB and tested against other subband algorithms. Unless otherwise stated the subband decomposition was initialized to the outer QMF filter bank with subband decomposition [22]. First, the algorithm was tested on its own to determine its functionality. Then the algorithm was tested against the band-stop filter. The maximum allowed decimation factor for this trial was set to 16, with the threshold and regulation values $\gamma_d = 0.025$, $\gamma_u = 0.125$, $\delta_{out} = 0.001$ and $\delta_{in} = 0.0001$. The input variance, noise variance and step size were matched to the previous tests for comparison purposes. The PSD estimates obtained for every iteration are shown in Figure 6.6. These estimates were taken over 50 updates, and updated every $2M = 32$ time steps. The result accurately matches the desired input output characteristics, and the floor caused by the ratio of the regulation factors is seen. The error signals of the adaptive filters is shown in Figure 6.7. These plots were obtained by averaging 100 trials to observe the average behavior of the algorithm. The final subband decomposition was $[2, 16, 16, 16, 16, 4]$.

Next the algorithm was tested against a system where the desired response changes dramatically. The same values as the previous test were tested against a band-pass filter of length 197 for 8000 iterations, where the desired filter was changed to a high-pass filter of length 239. The PSD estimates of 50 past FFTs is shown in Figure 6.8. The plot shows the eventual shift in PSD responses over time as the incoming FFTs are changed from the band-pass filter to the low-pass filter. Figure 6.9 shows the convergence properties and the changing subbands. Approximately 80 samples after the switch to the low pass filter, the lower half-rate filter is fully merged from

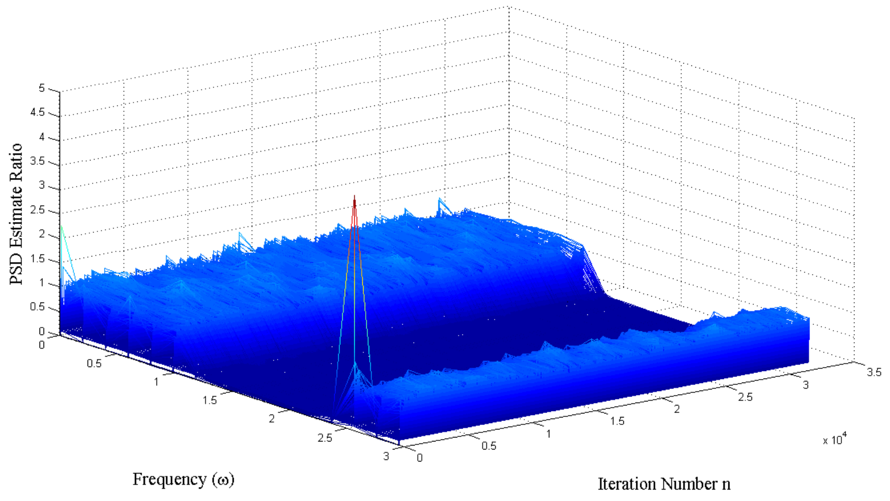

Figure 6.6: PSD Estimates for Stop-Band Test

the eighth-rate filters, speeding up the convergence. At approximately 600 samples after the switch, the upper half-band filter is split up in order to save computations. Another important result displayed in Figure 6.9 is the inability of the proposed structure to account for cross-terms in the center of the frequency range. Although a decomposition of $[4, 2, 4]$ or $[3, 3, 3]$ would be optimal for this band-pass filter, these are not possible due to the tree structure consisting of QMF subsystems: the closest two allowable decompositions are $[2, 2]$ and $[4, 4, 4, 4]$.

In order to check the allocation algorithm, the algorithm was run against both a stop-band filter and a low-pass filter. The same values were used as for the bandpass test: $\gamma_d = 0.025$, $\gamma_u = 0.125$, $\delta_{out} = 0.001$ and $\delta_{in} = 0.0001$. The PSD averaging was again done over 50 past estimates and updated every $2M = 16$ iterations. The maximum number of subbands was reduced to eight, and the resulting decompositions are shown in Figures 6.10 and 6.11. These plots demonstrate that the algorithm can correctly detect the extra signal component at the high frequencies and merge the

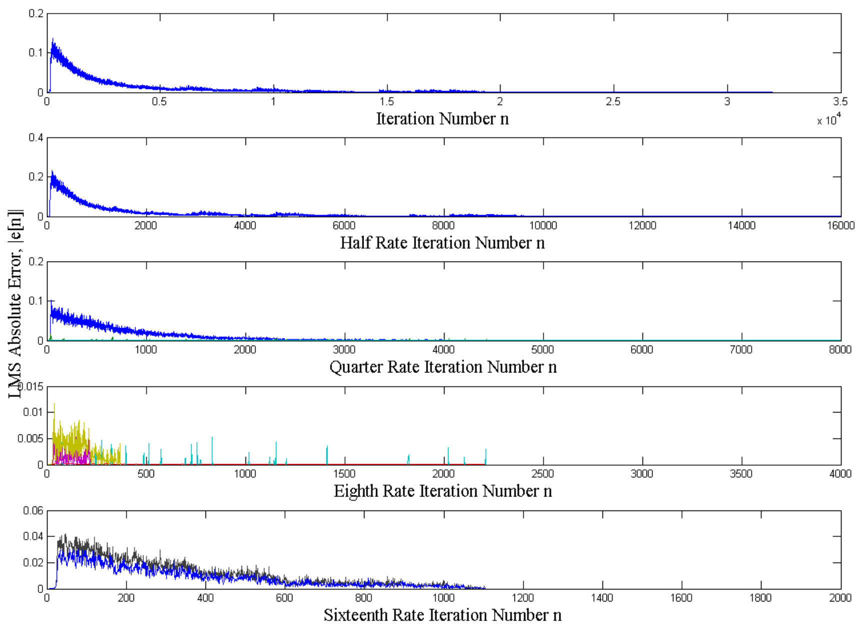

Figure 6.7: Error Signals for Stop-Band Test. From top to bottom: Full Band Errors, Half Band Errors, Quarter Rate Errors, Eighth Rate Errors, Sixteenth Rate Errors

bands to improve performance while retaining the smaller subbands in the stop band region.

The adjustable subband algorithm was then tested against the uniform subband filters. Figures 6.13 and 6.14 show examples of the relative performance. For comparative purposes, the values for signal power and the adaptive parameters were kept the same as the tests shown previously for these algorithms. Figure 6.13 shows the relative performance of uniform LMS and RLS structures with 4 bands against the adjustable subband algorithm using a maximum of 4 bands. The desired response was the high-pass filter.

In order to test the retention of performance for noisy systems, the three algo-

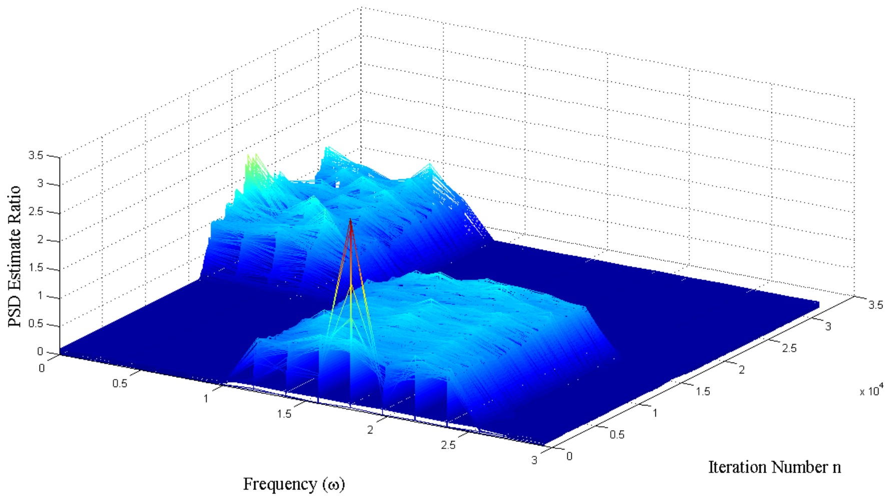

Figure 6.8: PSD Estimates for Band-Pass to Low-Pass Test

rithms were run for 11000 iterations and averaged over 3 trials for a number of noise variances. The uniform subband LMS and RLS algorithms were run using 4 bands. The adjustable subband algorithm was run with a maximum allowed decomposition of 4 subbands and the same parameters as the previous tests. Tables 6.4 and 6.5 show that the larger variances in the additive noise raise the MSE of the results, but have little effect on the convergence rate. The adjustable subband algorithm managed to correctly obtain the subband decomposition that allowed it to outperform the uniform subband LMS while maintaining a low computational complexity, as shown in Tables 6.4 and 6.5.

As a final test, the adjustable filter algorithm was tested against a time invariant non-uniform subband LMS filter. Noiseless conditions were used, with the signal variance again at $\sigma_u^2 = 0.36$ and a step size of $\mu = 0.01$. Both subband decompositions were initialized to $[4, 4, 16, 16, 16, 16, 16, 16, 16, 16]$, and the adaption parameters for the adjustable subband LMS were again set to $\gamma_d = 0.025$, $\gamma_u = 0.125$, $\delta_{out} = 0.001$

Filter Response	Low Pass Filter	High Pass Filter	Band Pass Filter	Band Stop Filter
Filter Length	236	239	197	197
Pass Band Frequency 1	0.2500	0.8125	0.3704	0.3333
Pass Band Frequency 2	N/A	N/A	0.6422	0.9292
Stop Band Frequency 1	0.2813	0.7813	0.3333	0.3704
Stop Band Frequency 1	N/A	N/A	0.9292	0.6422
Pass Band Attenuation 1 (dB)	0.1	0.1	0.1	0.1
Pass Band Attenuation 2 (dB)	N/A	N/A	N/A	0.2
Stop Band Attenuation 1 (dB)	60	0.1	60	60
Stop Band Attenuation 2 (dB)	N/A	N/A	70	N/A

Table 6.3: Test Filters for Adaptive Algorithms

Noise Variance σ_ν^2	0.00009	0.00025	0.001	0.009
Uniform LMS	-49dB	-49dB	-45dB	-43dB
Uniform RLS	-63dB	-63dB	-52dB	-44dB
Adjustable Subband LMS	-60dB	-58dB	-52dB	-43dB

Table 6.4: Noise Effect on Subband Filtering MSE

Noise Variance σ_ν^2	0.00009	0.00025	0.001	0.009
Uniform LMS	-0.001dB/n	-0.001dB/n	-0.001dB/n	-0.001dB/n
Uniform RLS	-0.016dB/n	-0.016dB/n	-0.011dB/n	-0.015dB/n
Adjustable Subband LMS	-0.009dB/n	-0.0055dB/n	-0.008dB/n	-0.008dB/n

Table 6.5: Noise Effect on Subband Filtering Convergence Rate (dB/n indicates change in dB error per iteration n

	Number of Computations Per Iteration
Uniform LMS	253
Uniform RLS	11043
Adjustable Subband LMS	327.5

Table 6.6: Comparison of Computation Complexities for Noise Tests

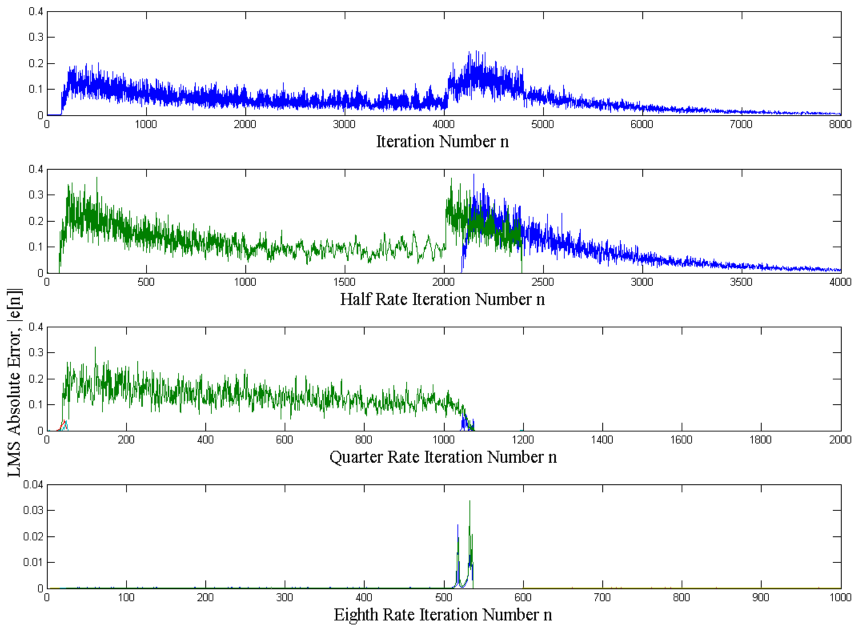

Figure 6.9: Error Signals for Band-Pass to Low-Pass Test. From top to bottom: Full Band Errors, Half Band Errors, Quarter Rate Errors, Eighth Rate Errors, Sixteenth Rate Errors

and $\delta_{in} = 0.0001$. First the desired response was set to the low-pass filter for 10000 iterations, at which the response was changed to the high-pass filter for another 10000 iterations. 10 trials were averaged to obtain an estimated response. As seen in Figure 6.15, both filters had identical convergences over the first 10000 iterations, where the time-invariant decomposition was well suited for the input-output characteristics of the desired response. However when the response changed to one which the time-invariant decomposition was ill-suited for, the adaptable algorithm was able to change its decomposition to $[16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 4]$, giving a much better convergence rate, as seen in Figure 6.15.

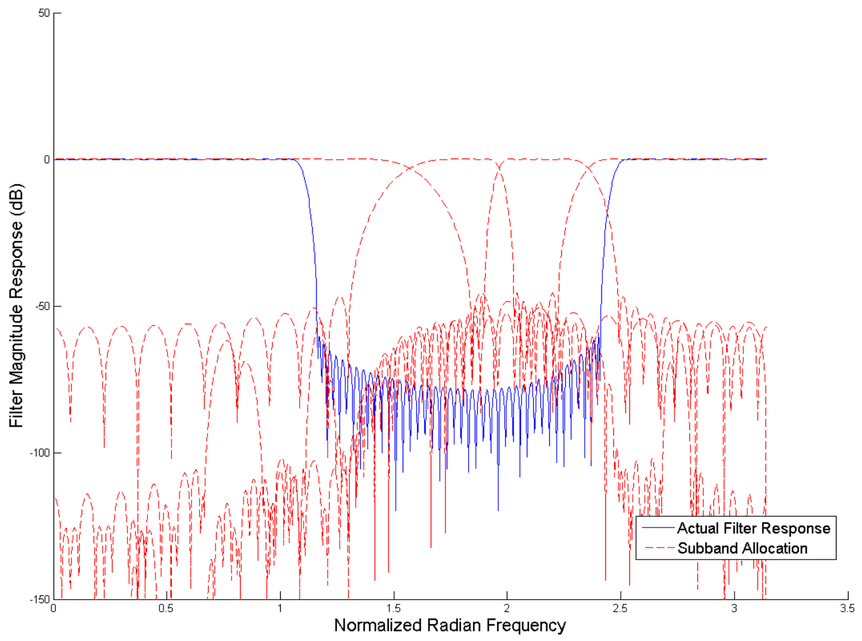

Figure 6.10: Steady State Subband Allocation for a Stop-Band Filter

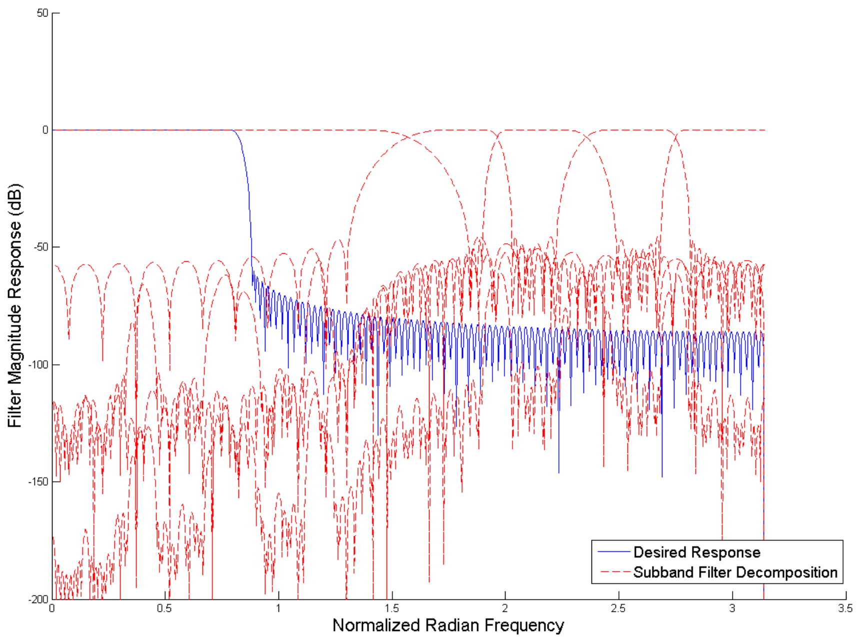

Figure 6.11: Steady State Subband Allocation for a Low-Pass Filter

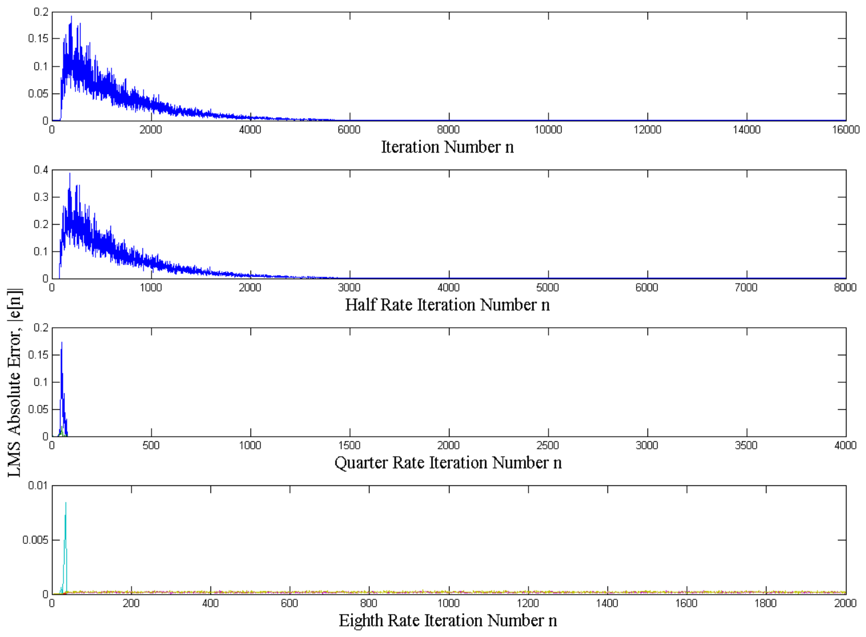

Figure 6.12: Error Signals for Low-Pass Filter Test. From top to bottom: Full Band Errors, Half Band Errors, Quarter Rate Errors, Eighth Rate Errors.

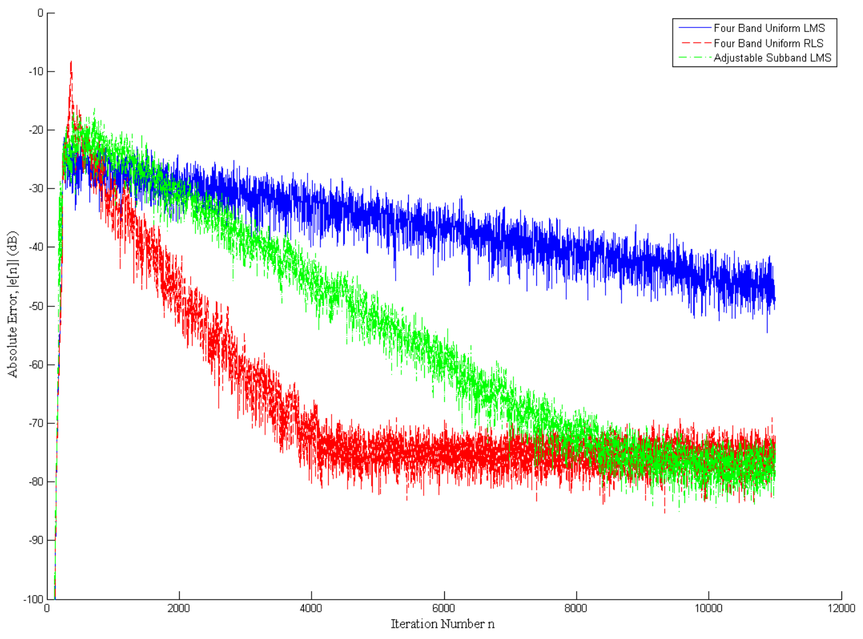

Figure 6.13: Comparison of Subband Filters from a HPF

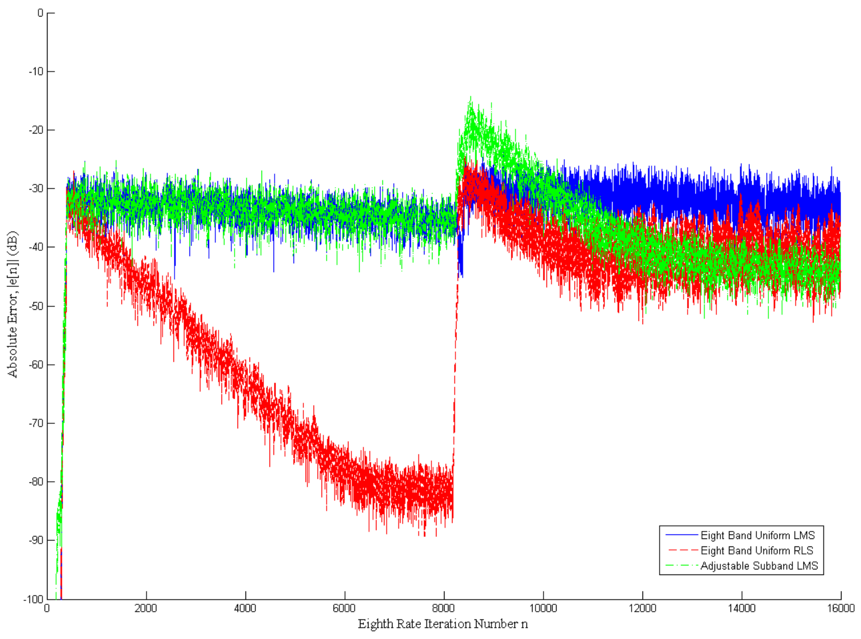

Figure 6.14: Comparison of Subband Filters from a HPF to a LPF

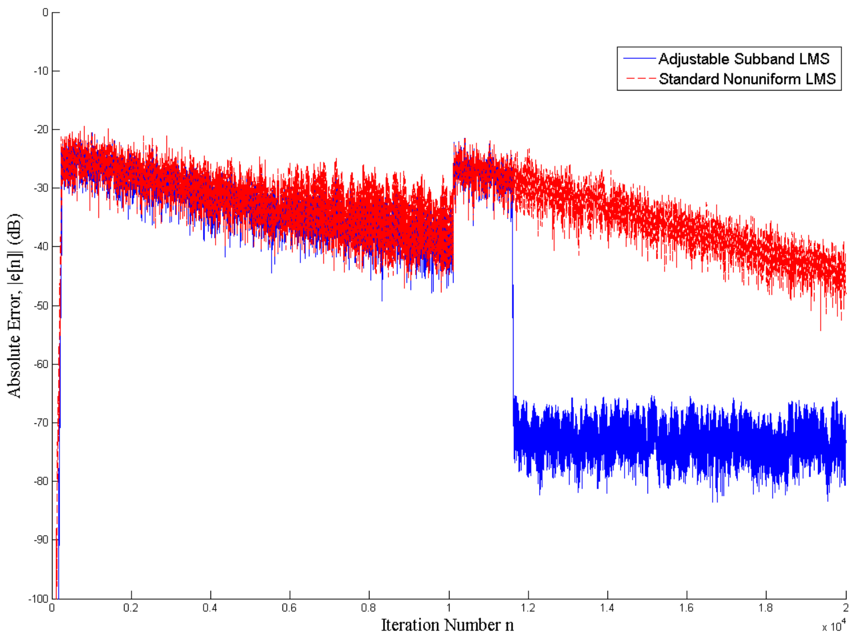

Figure 6.15: Comparison of Adjustable Non-Uniform Subband LMS to the Time-Invariant Non Uniform Subband LMS

Chapter 7

Conclusions and Further Work

This thesis describes a new algorithm to adjust the non-uniform subband decomposition of FIR filter banks for use in adaptive filter schemes. Both a structure capable of varying the subband widths, as well as an algorithm to determine the widths to implement were designed. A tree structure with interchangeable subsystems was proposed as a time-varying subband structure, allowing the subband widths to be easily adjusted. The bandwidth adjustment algorithm was based off of comparing the PSD estimates of the input and output of the unknown desired filter. The algorithm presented was implemented in MATLAB and compared to other subband adaptive structures utilizing either the LMS or RLS adaptive algorithms.

The subband adjustment algorithm performed successfully, matching the subband decompositions for a variety of test filters. The only unknown filter frequency responses the algorithm was unable to adapt well for were responses in which the passband crossed the center frequency, $f_s/4$. In these cases, the inherent inability for the the tree structure to merge bands across this frequency did not allow for those cross-terms to be taken into account.

The overall adaptive structure converged faster than similar uniform subband LMS structures, but slower than the uniform subband RLS structure. The computational complexity was also intermediary - greater than the uniform LMS algorithm but less than the uniform RLS algorithm. The algorithm also showed improvement in tracking non-stationary unknown filters over the stationary non-uniform subband structure. When the unknown filter changed its frequency response, the subband widths of the structure using the proposed algorithm were adjusted in order to speed up the convergence while the stationary subband structure converged slowly due to the now sub-optimal subband decomposition.

Further work in adjustable subband adaptive filtering should begin by focusing on alternate time-varying filter bank structures. Although tree structures built from QMF filter banks obtained good results in some cases, frequency responses that cross one fourth the sampling frequency have cross-terms that cannot be accounted for. The use of tree structures utilizing embedded filter banks with odd numbers of subbands is one possible approach to address this, but would suffer from a similar limitation at $f_s/6$ and $f_s/3$. Ideally, an easily implementable method of combining any two neighboring filters in a uniform M-channel filter bank should be determined, thereby eliminating the need for tree structures.

In addition, other methods of deciding on the optimal decomposition should be tested. For instance, if there is a large amount of memory available, a look-up table of all the possible decompositions can be implemented. In this way, the optimal decomposition for the calculated PSD estimates can be realized directly, without the need to split or combine filters multiple times.

Appendix A

MATLAB CODE

A.1 Table of Functions

Table A.1: List of MATLAB Functions

Function Name	Functionality	Function Calls
`standardLMS.m`	Performs the standard LMS algorithm.	
`standardRLS.m`	Performs the standard RLS algorithm.	
`NLMS_alg.m`	Performs the standard NLMS algorithm.	
`AFF_RLS_ATNA.m`	Performs the AFF-RLS-ATNA algorithm.	
`AFinput_noise.m`	Calculates a vector containing background noise and shot noise from user specified parameters.	
`non_stat_filt.m`	Calculates the output of a non-stationary filter.	

Continued on next page

Table A.1 – continued from previous page

Function Name	Functionality	Function Calls
`AF_testing.m`	Test file to compare the full-band adaptive filters.	`standardLMS.m`, `standardRLS.m`, `NLMS_alg.m`, `AFF_RLS_ATNA.m`, `AFinput_noise.m`, `non_stat_filt.m`
`interp_zeros.m`	Performs zero-fill interpolation.	
`smartinterp.m`	Re-samples a vector from N samples to M samples.	
`eff_filt.m`	Calculates the effective filter for a QMF PR filter bank.	`smartinterp.m`.
`PRfiltbankFIR.m`	Generates a two band PR FIR filter pair.	
`filt_bank_gen.m`	Generates an 2^k channel filter bank using a tree structure based on QMF bands.	`PRfiltbankFIR.m` `interp_zeros.m`.
`filt_test_create.m`	Generates the test filters	
`subbandLMSsimple2.m`	Implementation of a Uniform Subband LMS Algorithm	`filt_bank_gen.m`, `interp_zeros.m`
`subbandRLSsimple2.m`	Implementation of a Uniform Subband RLS Algorithm	`filt_bank_gen.m`, `interp_zeros.m`
`subbandNULMS_adapt.m`	Adjustable non-uniform subband LMS algorithm.	`filt_bank_gen.m`, `subband_update.m`, `interp_zeros.m`
`subband_update.m`	Updates Nonuniform Filter Bank Decimation Factors	`eff_filt.m`
`subband_filt_test.m`	Testing code for the subband adaptive filters	`subbandNULMS_adapt.m`, `subbandRLSsimple2.m`, `subbandLMSsimple2.m`, `AFinput_noise.m`, `non_stat_filt.m`, `standardLMS.m`, `standardRLS.m`, `filt_test_create.m`

A.2 Standard LMS Algorithm

```
function [d_hat, LMS_error, w_mat] = standardLMS(u_in, d_out, w_start,
    step_size, figno, plot_opt)

% Adam Charles
% Standard LMS algorithm
%
% Instintanuoeus approximation of R and P.

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Error Checking Inputs

if size(u_in, 1) ≠ 1 && size(u_in, 2) ≠ 1
    error('u_in must be a vector!');
elseif size(u_in, 1) == 1 && size(u_in, 2) ≠ 1
    u_in = u_in.';
end

if size(d_out, 1) ≠ 1 && size(d_out, 2) ≠ 1
    error('d_out must be a vector!');
elseif size(d_out, 1) == 1 && size(d_out, 2) ≠ 1
    d_out = d_out.';
end

if size(w_start, 1) ≠ 1 && size(w_start, 2) ≠ 1
    error('w_start must be a vector!');
elseif size(d_out, 1) ≠ 1 && size(d_out, 2) == 1
    w_start = w_start.';
end

if size(step_size, 1) ≠ 1 || size(step_size, 2) ≠ 1
    error('step_size must be a scalar!');
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Initializations and Preliminary Calculations

% Take care of possible length discrepancies.
num_iterations = min(length(u_in), length(d_out));
u_in = [zeros(length(w_start)-1, 1); u_in];

% Initializations
w_mat = zeros(num_iterations+1, length(w_start));
w_mat(1, :) = w_start;
```

```
d_hat = zeros(num_iterations, 1);
LMS_error = zeros(num_iterations, 1);


%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Run the Algorithm:

for index = 1:num_iterations
    d_hat(index) = (w_mat(index, :))*(u_in(index:index+length(w_start)
        -1));
    LMS_error(index) = d_out(index) - d_hat(index);
    w_mat(index+1, :) = w_mat(index, :) + (step_size)*...
        (u_in(index:index+length(w_start)-1).')*conj(LMS_error(index));
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Optional Plots

if plot_opt == 1
    figure(figno);
    plot(1:num_iterations, [d_out, d_hat, LMS_error]);
    title('System Identification of an FIR Filter using Standard LMS')
    xlabel('Index');
    legend('Ideal', 'Predicted', 'Error')
end
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.3 Standard RLS Algorithm

```
function [d_hat, Xi, w] = standardRLS(u_vector, d_ideal, w_start,
    lambda, △, figno, plot_opt)

% Adam Charles
% standard RLS algorithm
% Follows the following equations:
%
% Xi(n) = d_ideal(n)-w(n-1)x(n)                        A Priori Error
% Pi(n) = P(n-1)x(n)                                   A Priori Whitening
% k(n) = Pi(n)/(lambda+x_hermetian(n)*Pi(n))           Gain Vector
% w(n) = w(n-1) + k(n)Xi_conj(n)                       Weight Adaption
% P(n) = lambda^(-1)[I-k(n)x_hermitian(n)]P(n-1)     Riccati Update
%
%
```

```
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Error Checking of Inputs

if size(u_vector, 1) ≠ 1 && size(u_vector, 2) ≠ 1
    error('u_vector must be a vector!');
elseif size(u_vector, 1) == 1 && size(u_vector, 2) ≠ 1
    u_vector = u_vector.';
end

if size(d_ideal, 1) ≠ 1 && size(d_ideal, 2) ≠ 1
    error('d_ideal must be a vector!');
elseif size(d_ideal, 1) == 1 && size(d_ideal, 2) ≠ 1
    d_ideal = d_ideal.';
end

if size(w_start, 1) ≠ 1 && size(w_start, 2) ≠ 1
    error('w_start must be a vector!');
elseif size(w_start, 1) ≠ 1 && size(w_start, 2) == 1
    w_start = w_start.';
end

if size(lambda, 1) ≠ 1 || size(lambda, 2) ≠ 1
    error('Forgetting Factor (lambda) must be a scalar!');
end

if size(△, 1) ≠ 1 || size(△, 2) ≠ 1
    error('Initial Variance (△) must be a scalar!');
end

if size(figno, 1) ≠ 1 || size(figno, 2) ≠ 1
    error('Figure Number (figno) must be a scalar!');
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Initialization and Preliminary Calculations

last_time = min([length(u_vector), length(d_ideal)]);

% Initialize vectors for efficient for-loops
u_vector = [zeros(length(w_start)-1, 1); u_vector];
Xi = zeros(last_time, 1);
k = zeros(last_time+1, length(w_start));
w = zeros(last_time+1, length(w_start));
d_hat = zeros(last_time, 1);
P = eye(length(w_start))/△;
```

```
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Run algorithm

for index = 2:last_time
    d_hat(index-1) = w(index-1, :)*u_vector(index:index+length(w_start)
        -1);
    Xi(index-1) = d_ideal(index)-w(index-1, :)*u_vector(index:index+
        length(w_start)-1);
    Pi = P*u_vector(index:index+length(w_start)-1);
    k(index, :) = Pi./(lambda+u_vector(index:index+length(w_start)-1)'*
        Pi);
    w(index, :) = w(index-1, :) + k(index, :)*conj(Xi(index-1));
    P = lambda^(-1)*[eye(length(w_start))-(k(index, :).')*(u_vector(
        index:index+length(w_start)-1)')]*P;
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Optional Plotting

if plot_opt == 1
    figure(figno);
    plot(1:length(d_ideal), [d_ideal,d_hat,Xi]);
    title('System Identification of an FIR Filter');
    legend('Desired','Output','Error');
    xlabel('Time Index'); ylabel('Signal Value');
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.4 Normalized LMS Algorithm

```
function [d_hat, NLMS_error, w_mat] = NLMS_alg(u_in, d_out, w_start,
    step_size, reg_fact, figno, plot_opt)

% Adam Charles
% Normalized LMS algorithm
%

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Error Checking Inputs
```

```

if size(u_in, 1) ≠ 1 && size(u_in, 2) ≠ 1
    error('u_in must be a vector!');
elseif size(u_in, 1) == 1 && size(u_in, 2) ≠ 1
    u_in = u_in.';
end

if size(d_out, 1) ≠ 1 && size(d_out, 2) ≠ 1
    error('d_out must be a vector!');
elseif size(d_out, 1) == 1 && size(d_out, 2) ≠ 1
    d_out = d_out.';
end

if size(w_start, 1) ≠ 1 && size(w_start, 2) ≠ 1
    error('w_start must be a vector!');
elseif size(d_out, 1) ≠ 1 && size(d_out, 2) == 1
    w_start = w_start.';
end

if size(step_size, 1) ≠ 1 || size(step_size, 2) ≠ 1
    error('step_size must be a scalar!');
end

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Initializations and Preliminary Calculations

% Take care of possible length discrepancies.
num_iterations = min(length(u_in), length(d_out));
u_in = [zeros(length(w_start)-1, 1); u_in];

% Initializations
w_mat = zeros(num_iterations+1, length(w_start));
w_mat(1, :) = w_start;
d_hat = zeros(num_iterations, 1);
NLMS_error = zeros(num_iterations, 1);


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Run the Algorithm:

for index = 1:num_iterations
    d_hat(index) = (w_mat(index, :))*(u_in(index:index+length(w_start)-1));
    NLMS_error(index) = d_out(index) - d_hat(index);
    w_mat(index+1, :) = w_mat(index, :) + (step_size)*...
        (u_in(index:index+length(w_start)-1).')*conj(NLMS_error(index))
```

```
            ./(reg_fact + (u_in(index:index+length(w_start)-1).')*conj(u_in(index:index+length(w_start)-1)));
%     (u_in(index:index+length(w_start)-1).')*conj(u_in(index:index+length(w_start)-1))
end

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Optional Plots

if plot_opt == 1
    figure(figno);
    plot(1:num_iterations, [d_out, d_hat, NLMS_error]);
    title('System Identification of an FIR Filter using Standard LMS')
    xlabel('Index');
    legend('Ideal', 'Predicted', 'Error')
end
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.5 AFF-RLS-ATNA Algorithm

```
function [d_hat, Xi, w] = AFF_RLS_ATNA(u_vector, d_ideal, w_start, lambda, ...
    lambda_adapt, △, non_lin_factor, leak_factor, ATNA_multiplier, figno, plot_opt)

% Adam Charles
% Adaptive Forgetting Factor (AFF) RLS algorithm
% RLS algorithm with added adaptive forgetting factor (lambda)
%
% [d_hat, Xi, w] = AFF_RLS(u_vector, d_ideal, w_start, lambda, ...
% lambda_adapt, △, non_lin_factor, leak_factor, ATNA_multiplier, figno, plot_opt)
%     Inputs:
%           u_vector = input vector
%           d_ideal = output of filter to be approximated
%           w_start = initial tap weight vector
%           lambda = initial forgetting factor (between 0 and 1)
%           lambda_adapt = constant in forgetting factor update
%           △ = initial "covariance" matrix
%           non_lin_factor
%           leak_factor
%           ATNA_multiplier
%           figno = figure number to plot to
%           plot_opt = 1 if plot outputs
%     Outputs:
```

```
%
%
%
% Follows the following equations:
%
% AFF adition
% FF(n) = 1 - FFc(n)
% FFc(n+1) = FFc(n) + rho_g*f[e(n+1);A(n+1),m]*f[e(n);A(n),m]
%
% c(n+1) = c(n) + a_c*f[e(n); A(n),m]*g(n)
% g(n) = P(n)*a(n)/[lambda + a.'(n)*P(n)*a(n)]
% P(n+1) = (1/lambda)[P(n) -g(n)*a.'(n)*P(n)]
% f[e(n); A(n),m] = e/[1+(|e|/A)^m]
% A(n+1) = (1-p_a)A(n) + p_a*M_a*|e(n)|
% p_a -> leakage, M_a -> multiplier
% w is the tap weight vector; a is the input; A is the threshold
    parameter;
% lambda is the forgetting factor; P is the estimated inverse covar
    matrix;
% k is the kalman gain;

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Error Checking

if size(u_vector, 1) == 1;
    u_vector = u_vector.';
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Initialization and Preliminary Calculations

last_time = min([length(u_vector), length(d_ideal)]);

% Initialize vectors for efficient for-loops
u_vector = [zeros(length(w_start)-1, 1); u_vector];
Xi = zeros(last_time, 1);
k = zeros(last_time+1, length(w_start));
w = zeros(last_time+1, length(w_start));
d_hat = zeros(last_time, 1);
P = eye(length(w_start))/Δ;
f_ATNA = zeros(last_time+1, length(w_start));
A_ATNA = 100;
lambda_c = 1-lambda;

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%% Run algorithm

for index = 2:last_time
    lambda = 1-lambda_c;
    d_hat(index-1) = w(index-1, :)*u_vector(index:index+length(w_start)
        -1);
    Xi(index-1) = d_ideal(index)-w(index-1, :)*u_vector(index:index+
        length(w_start)-1);
    Pi = P*u_vector(index:index+length(w_start)-1);
    k(index, :) = Pi./(lambda+u_vector(index:index+length(w_start)-1)'*
        Pi);
    % Difference is here: instead of Kalman*error: Kalman*f(error, A, m
        )
    f_ATNA(index-1) = Xi(index-1)/(1+(abs(Xi(index-1))/A_ATNA)^
        non_lin_factor);
    w(index, :) = w(index-1, :) + k(index, :)*conj(f_ATNA(index-1));
    A_ATNA = (1-leak_factor)*A_ATNA + leak_factor*ATNA_multiplier*Xi(
        index-1);

    P = lambda^(-1)*[eye(length(w_start))-(k(index, :).')*(u_vector(
        index:index+length(w_start)-1)')]*P;
    if index ≥ 3
        lambda_c = lambda_c + (lambda_adapt*f_ATNA(index-1)*f_ATNA(
            index-2)*(u_vector(index:index+length(w_start)-1)')*(k(index
            , :).'))/lambda_c;
    elseif index == 1 || index == 2
        lambda_c = lambda_c;
    else
        error('Index ≤ 0');
    end
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Optional Plotting

if plot_opt == 1
    figure(figno);
    plot(1:length(d_ideal), [d_ideal,d_hat,Xi]);
    title('System Identification of an FIR Filter');
    legend('Desired','Output','Error');
    xlabel('Time Index'); ylabel('Signal Value');
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.6 Noise Generating Function

```
function [total_noise] = AFinput_noise(noise_len, gauss_stats,
    pois_lambda, impulse_stats, opts)

% Adam Charles
% 11/3/2009
% Creates Noise for Adaptive filtering
%
% Background noise + Impulse noise.
% Impulse noise is poisson process with gaussian magnitudes
%

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Decifer Inputs:

if opts(1) == 1;
    gauss_mean = gauss_stats(1);
    gauss_var = gauss_stats(1);
else
    gauss_mean = 0;
    gauss_var = 0;
end

if opts(2) == 1;
    impulse_mean = impulse_stats(1);
    impulse_var = impulse_stats(2);
else
    impulse_mean = 0;
    impulse_var = 0;
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Generate Noise:

% Background Noise
background_noise = gauss_var*randn(noise_len, 1) + gauss_mean;

%Impulse Noise
impulse_times = cumsum(poissrnd(pois_lambda, noise_len, 1));
impulse_times2 = impulse_times(find(impulse_times ≤ noise_len));
impulse_vals = impulse_var*randn(length(impulse_times2), 1) +
    impulse_mean;
impulse_noise = zeros(noise_len, 1);
impulse_noise(impulse_times2) = impulse_vals;
```

```

total_noise = background_noise + impulse_noise;

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.6.1 Non-Stationary Filter Evaluation

```
function [output] = non_stat_filt(b_in, a_in, data)

% non_stat_filt filters the data through a non-stationary filter
%
% b_in has a row of neumerator coefficients for each time-step
% a_in has a row of denomenator coefficients for each time-step
% data is the data to be processed

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Error Checking

[a_i, a_j] = size(a_in);
[b_i, b_j] = size(b_in);
[data_i, data_j] = size(data);

if a_i == 1
    a_in = repmat(a_in, data_i, 1);
elseif a_i ≠ data_i && a_i ≠ 1
    error('Number of rows of neumerator coefficients must equat number
        of rows of denomenator coefficients');
end

if b_i == 1
    b_in = repmat(b_in, data_i, 1);
elseif b_i ≠ data_i && b_i ≠ 1
    error('Number of rows of neumerator coefficients must equat number
        of rows of input data');
end

if data_j ≠ 1
    error('Data input must be a column vector');
end

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Run Algorithm

```

```
% buffer inputs and outputs
d_eff = [zeros(b_j-1, 1);data];
out_eff = zeros(a_j + data_i, 1);

for index = 1:data_i
    A0 = fliplr(a_in(index, 2:end))*out_eff((index+1):(index+a_j-1), 1)
        ;
    B0 = fliplr(b_in(index, :))*d_eff(index:(index+b_j-1), 1);
    out_eff(a_j + index) = (B0-A0)./(a_in(index, 1));
end

output = out_eff(a_j+1: end);

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.7 Fullband Testing Code

```
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% RLS testing code:

% Clean Slate:
close all
clear
clc

% Options:
run_tests = [1 1 1 1]; % Standard LMS, Standard RLS, NLMS, AFF-RLS-ATNA
allow_noise = 1;
figno = 1;
stationary_filt = 0;
t_last = 1500;
num_trials = 100;

err_AFF_ATNA = 0;
LMS_error = 0;
NLMS_error = 0;
e_i = 0;

[pulse_noise] = AFinput_noise(t_last, [0, 0.001], 700, [1, 25], [0,1]);

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Test Inputs/Outputs
for index = 1:num_trials
```

```
% Filter Selection
A_test = [linspace(7,7.1,t_last); linspace(5,5.1,t_last); linspace(2, 1.9, t_last); linspace(1,0.9,t_last)].';
B_test = [linspace(9,11,t_last); linspace(2,-2,t_last); linspace(1, 5, t_last); linspace(3,-2,t_last)].';

% A_test = ([linspace(7,7.1,t_last); linspace(5,5.1,t_last); linspace(2, 1.9, t_last); linspace(1,0.9,t_last)] +...
%     0*[zeros(1, floor(t_last/2)), 8*ones(1, ceil(t_last/2)); zeros(1, floor(t_last/2)),...
%     8*ones(1, ceil(t_last/2)); zeros(1, floor(t_last/2)), 8*ones(1,
%     ceil(t_last/2)); zeros(1, floor(t_last/2)),...

8*ones(1, ceil(t_last/2))]).';
% B_test = [linspace(9,11,t_last); linspace(2,-2,t_last); linspace(1, 5, t_last); linspace(3,-2,t_last)].';
% B_test = [10 -6 -5 9 -8 2 2 1 -3 1 0 -7 4 -2 3 4 2 9 1 2 -6 5 3 -5 3 2 -6 -8 0 9 -2 4];

u_in = 4*randn(t_last, 1);
% u_in = sin(2*pi*[1:1000]);

d_out = non_stat_filt(B_test, A_test, u_in);
% fdtool(B_test, A_test)

FIR_filt_size = 100;

% Freq response of ideal filter
W_vec = linspace(0,pi, 10000);
% if stationary_filt == 1
    Htest = freqz(B_test(end, :), A_test(end, :), W_vec);
% end

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Noise in the System

d_var = 0.01;
d_mean = 0;

[d_noise] = AFinput_noise(t_last, [0, 0.001], 700, [1, 30], [1,0]);

% colored noise
if allow_noise == 1
    d_out = d_out + d_noise + pulse_noise;
end

%
```

```
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Standard LMS

if run_tests(1) == 1
    LMS_step_size = 0.0003;                          % Step size
    w_start_LMS = zeros(1, FIR_filt_size);           % Start Vector
    figno = figno+1;                                 % Incriment figure
        number
    plot_opt_LMS = 0;                                % Plot

    [d_hat, LMS_error_temp, w_mat_LMS] = standardLMS(u_in, d_out,
        w_start_LMS, LMS_step_size,...
        figno, plot_opt_LMS);
    LMS_error = LMS_error + abs(LMS_error_temp)/num_trials;
    clear LMS_error_temp
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Normal RLS

if run_tests(2) == 1
    w_start_RLS = zeros(1, FIR_filt_size);           % start vector
    figno = figno + 1;                               % Next figure
    lambda = 0.9;                                    % Lambda
    △ = 0.1;

    [y_i, e_i_temp, w_i] = standardRLS(u_in, d_out, w_start_RLS, lambda
        , △, figno, 0);
    e_i = e_i + abs(e_i_temp)/num_trials;
    clear e_i_temp

end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% NLMS

if run_tests(3) == 1
    NLMS_step_size = 0.3;                            % Step size
    w_start_NLMS = zeros(1, FIR_filt_size);          % Start Vector
    figno = figno+1;                                 % Incriment figure
        number
    plot_opt_NLMS = 0;                               % Plot
    reg_fact = 0.01;                                 % Regulation Factor

    [d_hat_NLMS, NLMS_error_temp, w_mat_NLMS] = NLMS_alg(u_in, d_out
        ,...
        w_start_NLMS, NLMS_step_size, reg_fact, figno, plot_opt_NLMS);
    NLMS_error = NLMS_error + abs(NLMS_error_temp)/num_trials;
    clear NLMS_error_temp
end

%
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Koike ATNA w/ AFF

if run_tests(4) == 1

    % start at w_init = zeros
    w_start_koike = zeros(1, FIR_filt_size);
    % reasonable forgetting factor
    forget_fact = 0.99;
    % Start Var
    △ = 0.1;
    % leakage, multiplier, nonlinear order
    A_leak_AFFATNA = 2e-8;
    A_mult_AFFATNA = 2.5e-10;
    nonlin_order_AFFATNA = 32;
    adapt_coeff = 1e-10;

    [d_AFF_ATNA, err_AFF_ATNA_temp, w_AFF_ATNA] = AFF_RLS_ATNA(u_in,
       d_out,...
        w_start_koike, forget_fact, adapt_coeff, △,
           nonlin_order_AFFATNA,...
        A_leak_AFFATNA, A_mult_AFFATNA, figno, 0);
    err_AFF_ATNA = err_AFF_ATNA + abs(err_AFF_ATNA_temp)/num_trials;
    clear err_AFF_ATNA_temp

end

end
%
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Error Plots

err_fig = 6;
figure(err_fig);
if run_tests(1) == 1
    figure(err_fig), hold on;
    plot(LMS_error, 'b');
end
if run_tests(2) == 1
    figure(err_fig), hold on;
    plot(e_i, '--r');
```

```
end
if run_tests(3) == 1
    figure(err_fig), hold on;
    plot(NLMS_error, ':k');
end
if run_tests(4) == 1
    figure(err_fig), hold on;
    plot(err_AFF_ATNA, '-.b');
    legend('LMS', 'RLS', 'NLMS', 'AFF-RLS-ATNA')
    xlabel('Index n', 'FontSize', 12, 'FontName', 'Times')
    ylabel('Absolute error |e[n]|', 'FontSize', 12, 'FontName', 'Times'
        )
end

figure(err_fig), hold off;

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.8 Zero-Fill Interpolation Code

```
function output = interp_zeros(input, ammt, dim)

% Adam Charles
% 3/15/2009
%
% Inpterpolates with zero fills along the dimention 'dim'
% number of zeros given by (ammt-1).
% This function was written since MATLAB's interp() function does not
% zero-fill, but instead low pass filters.
%

if ammt == 0
    output = input;
else

    [M, N] = size(input);

    if dim == 1
        output = zeros(ammt*M - ammt + 1, N);
        output(1:ammt:end ,:) = input;
    elseif dim == 2
        output = zeros(M, ammt*N - ammt + 1);
        output(:, 1:ammt:end) = input;
    else
        error('Not a valid dim!');
    end
end
```

A.9 Length Adjusting Code

```
function [h_eff] = smartinterp(h0, K)

% Adam Charles
% 3/8/2009
%
% [h_eff] = smartinterm(h0, K) calculates h_eff in the following manner
    :
%
% h_eff = (decimate by M/gcd(M,K)) (LPF) (interpolate by K/gcd(M,K)) (
    h0)
%
% Where M = length(h0). h_eff is essentially h0 estimated with K points
    ,
% rather then M points.
%

gcd_num = gcd(K, length(h0));
dec_factor = length(h0)/gcd_num;
interp_factor = K/gcd_num;
h_temp1 = interp(h0, interp_factor);
h_eff = h_temp1(1:dec_factor:end);

% Plots were for testing purposes.
% plot(h0, 'r')
% figure
% plot(h_eff, 'b')
```

A.10 Effective Filter Evaluation Code

```
function [g0_eff] = eff_filt(g0, g1, h0, h1, f0, f1)

% Adam Charles
% 3/8/2009
% Effective Filter Calculations

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Error Checking

```

```matlab
if length(g0) ≠ length(g1)
    error('g0 and g1 must have equal lengths!')
end

if (length(h0) ≠ length(h1)) || (length(h0) ≠ length(f0)) ||...
        (length(h0) ≠ length(f1)) || (length(h1) ≠ length(f0)) ||...
        (length(h1) ≠ length(f1)) || (length(f0) ≠ length(f1))
    error('h0, h1, f0, and f1 must have equal lengths!')
end


%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Calculations

% Get special lengths:
K = length(g0);
% Make all filters the desired 2*K length:
h0_eff = smartinterp(fftshift(abs(fft(h0))), 2*K);
h1_eff = smartinterp(fftshift(abs(fft(h1))), 2*K);
f0_eff = smartinterp(fftshift(abs(fft(f0))), 2*K);
f1_eff = smartinterp(fftshift(abs(fft(f1))), 2*K);
% Take into account imaged parts:
g0f = fftshift(abs(fft(g0)));
g1f = fftshift(abs(fft(g1)));
g0_mid = [g0f((floor(K/2)+1):end) , g0f, g0f(1:floor(K/2))];
g1_mid = [g1f((floor(K/2)+1):end) , g1f, g1f(1:floor(K/2))];
% Get aliased parts taken into account:
h0_mid = h0_eff + [h0_eff(K:end), h0_eff(1:K-1)];
h1_mid = h1_eff + [h1_eff(K:end), h1_eff(1:K-1)];
% Put tugether final effective filter:
g0_eff = 0.5*real(ifft(fftshift(f0_eff.*g0_mid.*h0_mid + f1_eff.*g1_mid
    .*h1_mid)));

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.11 Two Band PR Filter Gererating Code

```matlab
function [flp, fhp] = PRfiltbankFIR(opt)

% PR Filter Generators
%
% equivals: [3/100, 0.1, 20]
% kaiservals: [0.45, 0.585, 0.415, 0.55, 0.1, 25]


% Basic LP/HP filters
flpspec.fp= 1/2 - 5/100;          % pass-band frequency as calculated
    above
flpspec.fst= 1/2 + 8.5/100;       % stop-band frequency is 20Hz above
    pass-band
flpspec.Ap= 0.1;                  % worst pass-band attenuation is 1db
flpspec.Ast= 25;                  % best stop-band attenuation is 20db
% generate actual filter from specs
flpspecs= fdesign.lowpass('fp,fst,Ap,Ast', ...
    flpspec.fp, flpspec.fst, flpspec.Ap, flpspec.Ast);
flp= design(flpspecs,'kaiserwin');

fhpspec.fp= 1/2 - 8.5/100;            % pass-band frequency as
    calculated above
fhpspec.fst= 1/2 + 5/100;             % stop-band frequency is 20Hz
    above pass-band

fhpspecs= fdesign.highpass('fst,fp,Ast,Ap', ...
    fhpspec.fp, fhpspec.fst, flpspec.Ast, flpspec.Ap);
fhp= design(fhpspecs,'kaiserwin');

if strcmp(opt, 'o')
    w_1 = linspace(0, pi, 1000);
    H1 = freqz(flp, w_1);
    H2 = freqz(fhp, w_1);
    plot(w_1, abs(H1).^2 + abs(H2).^2);
    fvtool(flp, fhp)
end
```

A.12 Uniform Subband Filter Generating Code

```
function [filtbank] = filt_bank_gen(M, band_type, dir, plot_opt)

% Tree Struchtured Fitler Bank.
%
%
%

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Error Checking

if strcmp(band_type, 'uniform')
    % Uniform Filter Bank can only have M = 2^L outputs
    if 2^floor(log2(M)) ≠ M
        error('M must be a power of 2 for a uniform tree filter bank!')
            ;
    end
```

```
end

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Base Filters

[flp, fhp] = PRfiltbankFIR('x');    % Get predesigned filters
h_lp = [flp.Numerator, 0];               % Extract LP filter coefficients
h_hp = fhp.Numerator;               % Extract HP filter coefficients

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Decimation Filter Bank:

if strcmp(band_type, 'uniform')
    % Uniform Bands
    M_2 = log2(M);
    filt_val = 0;
    for index = 1:M
%         disp(sprintf('Filter H%d:', index-1))
        clear h_temp;
        h_temp = 1;
        filt_path = fliplr(dec2binvec(filt_val, M_2));
        for index2 = 1:M_2
            if filt_path(index2) == 0
%                 disp(sprintf('Layer %d is LPF with decimation %d', index2, index2-1))
                h_temp = conv(h_temp, interp_zeros(h_lp, 2^(index2-1), 2));
            elseif filt_path(index2) == 1
%                 disp(sprintf('Layer %d is HPF with decimation %d', index2, index2-1))
                h_temp = conv(h_temp, interp_zeros(h_hp, 2^(index2-1), 2));
            end
        end
        filt_val = filt_val + 1;
        eval(sprintf('filtbank.H%d = h_temp;', index-1));
    end
elseif strcmp(band_type, 'log')
    % Logarithmic sized bands
    for index = 1:M
        clear h_temp;
        if index == 1
            h_temp = h_hp;
        elseif index == 2
            h_temp = conv(h_lp, interp_zeros(h_hp, index, 2));
        elseif index > 2 && index < M
            h_temp = h_lp;
            for index2 = 2:index-1
                h_temp = conv(h_temp, interp_zeros(h_lp, 2^(index2-1),
                    2));
            end
            h_temp = conv(h_temp, interp_zeros(h_hp, 2^(index-1), 2));
        elseif index == M
            h_temp = 1;
            for index2 = 1:index-1
                h_temp = conv(h_temp, interp_zeros(h_lp, 2^(index2-1),
                    2));
            end
        end
        eval(sprintf('filtbank.H%d = h_temp;', index-1));
    end
end

if strcmp(dir, 's')
    % Leave Filter Bank Alone
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Synthesis Filter

if strcmp(dir, 's')
    % F(z) = z^(-N)H_para(z)
    for index = 1:M
        eval(sprintf('filtbank.H%d = fliplr(conj(filtbank.H%d));',
            index-1, index-1));
    end
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Plotting Options

if plot_opt
    plot_string = 'fvtool(filtbank.H0, 1';
    w_1 = linspace(0, pi, 1000);
    H_sum = abs(freqz(filtbank.H0, 1, w_1)).^2;
    for index = 2:M
        plot_string = sprintf('%s, filtbank.H%d, 1', plot_string, index
            -1);
        eval(sprintf('H_sum = H_sum + abs(freqz(filtbank.H%d, 1, w_1))
            .^2;', index-1));
    end
    plot(w_1, H_sum);
    plot_string = sprintf('%s);', plot_string);
    disp(plot_string)
    eval(plot_string);
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.13 Uniform Subband LMS Algorithm

```
function [output, output_error] = subbandLMSsimple2(u_in, d_ideal,...
    step_size, w_start, num_bands, band_type)

% Adam Charles
% 2/26/2009
%
% Subband LMS Filtering.
%
% Filters through Filterbank with M = num_bands, then performs LMS
% filtering on each subband seperately. Those results are then put
    through
% the synthesis filter to get the final outpt y(n).
%
% For the matricies used, each row indicates the subband-1 (row 1 is
% subband 0), and the column represents the iteration. The updates are
% essentially round robin, as they start at H_0, work up to H_{M-1},
    then
% go back to H_0.
%


%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Error Checking

if strcmp(band_type, 'uniform')
    % Uniform Filter Bank can only have M = 2^L outputs
    if 2^floor(log2(num_bands)) ≠ num_bands
        error('M must be a power of 2 for a uniform tree filter bank!')
            ;
    end
end

% Make sure u_in is a vector
if size(u_in, 1) ≠ 1 && size(u_in, 2) ≠ 1
    error('u_in must be a vector!');
```

```
elseif size(u_in, 1) == 1 && size(u_in, 2) ≠ 1
    u_in = u_in.';
end

% Make sure d_ideal is a vector
if size(d_ideal, 1) ≠ 1 && size(d_ideal, 2) ≠ 1
    error('d_ideal must be a vector!');
elseif size(d_ideal, 1) == 1 && size(d_ideal, 2) ≠ 1
    d_ideal = d_ideal.';
end

% Check size of w_start
if size(w_start, 1) ≠ num_bands && size(w_start, 2) ≠ num_bands
    error('w_start must be a vector!');
elseif size(w_start, 1) ≠ num_bands && size(w_start, 2) == num_bands
    w_start = w_start.'; % w's as row vectors
elseif isscalar(w_start)
    error('One tap is useless!')
end

if ¬isscalar(step_size)
    error('step_size must be a scalar!');
end


%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Prior Calculations and Initializations

% Initialize Tap Weight Matrix
w_mat = w_start;

% Calculate numbwe of 'time steps' or # times to updaye all filter
    banks
num_update_iters = floor(min(length(u_in), length(d_ideal))/num_bands);
% Pad input vector
u_in = [zeros(num_bands*size(w_start, 2)-1, 1); u_in];
d_hat = zeros(num_update_iters, num_bands);
LMS_error = zeros(num_bands, num_update_iters);

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Run Inputs and Outputs through Filter Banks:

[filtbank_dec] = filt_bank_gen(num_bands, 'uniform', 'f', 0);
[filtbank_syn] = filt_bank_gen(num_bands, 'uniform', 's', 0);


```

```
% fvtool(filtbank_dec.H0, 1, filtbank_dec.H1, 1)
% fvtool(filtbank_syn.H0, 1, filtbank_syn.H1, 1)


for index = 1:num_bands
    % Filter through the kth filter bank filter
    eval(sprintf('u_filt1.H%d = filter(filtbank_dec.H%d, 1, u_in);',
        index-1, index-1));
    eval(sprintf('d_filt1.H%d = filter(filtbank_dec.H%d, 1, d_ideal);',
         index-1, index-1));
    % Decimate by M
    eval(sprintf('u_dec.H%d = u_filt1.H%d(1:num_bands:end);', index-1,
        index-1));
    eval(sprintf('d_dec.H%d = d_filt1.H%d(1:num_bands:end);', index-1,
        index-1));
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% LMS Filtering

for index =  1:num_update_iters
    for index2 = 1:num_bands
        eval(sprintf('d_hat(index2, index) = (w_mat(index2, :))*(u_dec.
            H%d(index:index+size(w_mat, 2)-1));', index2-1));
        eval(sprintf('LMS_error(index2, index) = d_dec.H%d(index) -
            d_hat(index2, index);', index2-1));
        eval(sprintf('w_mat(index2, :) = w_mat(index2, :) + (step_size)
            *(u_dec.H%d(index:index+size(w_mat, 2)-1).'')*conj(LMS_error
            (index2, index));', index2-1));
    end
end


%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Filter/Interpolate Outputs:

d_out_sum = 0;
d_ideal_sum = 0;
for index = 1:num_bands
    % Interpolate by M: use function
    eval(sprintf('d_syn.H%d = interp_zeros(d_hat(%d, :), num_bands, 2);
        ', index-1, index));
    eval(sprintf('d_ideal_syn.H%d = interp_zeros(d_dec.H%d, num_bands,
        1);', index-1, index-1));
    % Filter through the kth synthesis bank filter
    eval(sprintf('d_filt2.H%d = filter(filtbank_syn.H%d, 1, d_syn.H%d);
        ', index-1, index-1, index-1));
    eval(sprintf('d_ideal_filt1.H%d = filter(filtbank_syn.H%d, 1, d_ideal_syn.H%d);', index-1, index-1, index-1));
    eval(sprintf('d_out_sum = d_out_sum + d_filt2.H%d;', index-1));
    eval(sprintf('d_ideal_sum = d_ideal_sum + d_ideal_filt1.H%d;', index-1));
end

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Specify Outputs:

output.out = d_out_sum;
output.ideal = d_ideal_sum;
% size(d_ideal_sum) , (1:length(d_out_sum))
% size(d_out_sum)
output_error.full = d_ideal_sum.' - d_out_sum;
output_error.subbands = LMS_error;

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.14 Uniform Subband RLS Algorithm

```
function [output, output_error] = subbandRLSsimple2(u_in, d_ideal,...
    forget_factor, △, w_start, num_bands)

% Adam Charles
% 2/26/2009
%
% Subband RLS Filtering.
%
% Filters through Filterbank with M = num_bands, then performs RLS
% filtering on each subband seperately. Those results are then put through
% the synthesis filter to get the final outpt y(n).
%
% For the matricies used, each row indicates the subband-1 (row 1 is
% subband 0), and the column represents the iteration. The updates are
% essentially round robin, as they start at H_0, work up to H_{M-1}, then
% go back to H_0.
%


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%% Error Checking


% Uniform Filter Bank can only have M = 2^L outputs
if 2^floor(log2(num_bands)) ≠ num_bands
    error('M must be a power of 2 for a uniform tree filter bank!');
end


% Make sure u_in is a vector
if size(u_in, 1) ≠ 1 && size(u_in, 2) ≠ 1
    error('u_in must be a vector!');
elseif size(u_in, 1) == 1 && size(u_in, 2) ≠ 1
    u_in = u_in.';
end

% Make sure d_ideal is a vector
if size(d_ideal, 1) ≠ 1 && size(d_ideal, 2) ≠ 1
    error('d_ideal must be a vector!');
elseif size(d_ideal, 1) == 1 && size(d_ideal, 2) ≠ 1
    d_ideal = d_ideal.';
end

% Check size of w_start
if size(w_start, 1) ≠ num_bands && size(w_start, 2) ≠ num_bands
    error('w_start must be a vector!');
elseif size(w_start, 1) ≠ num_bands && size(w_start, 2) == num_bands
    w_start = w_start.'; % w's as row vectors
elseif isscalar(w_start)
    error('One tap is useless!')
end

if ¬isscalar(forget_factor)
    error('forget_factor must be a scalar!');
end

if ¬isscalar(△)
    error('△ must be a scalar!');
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Prior Calculations and Initializations

% Initialize Tap Weight Matrix
w_mat = w_start;

% Calculate numbwe of 'time steps' or # times to updaye all filter
```

```
    banks
num_update_iters = floor(min(length(u_in), length(d_ideal))/num_bands);
% Pad input vector
u_in = [zeros(num_bands*size(w_start, 2)-1, 1); u_in];
d_hat = zeros(num_update_iters, num_bands);
RLS_error = zeros(num_bands, num_update_iters);
P0 = eye(size(w_mat, 2))/△;
kal_gain = 0;
pi_vec = 0;

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Run Inputs and Outputs through Filter Banks:

[filtbank_dec] = filt_bank_gen(num_bands, 'uniform', 'f', 0);
[filtbank_syn] = filt_bank_gen(num_bands, 'uniform', 's', 0);

for index = 1:num_bands
    % Filter through the kth filter bank filter
    eval(sprintf('u_filt1.H%d = filter(filtbank_dec.H%d, 1, u_in);',
        index-1, index-1));
    eval(sprintf('d_filt1.H%d = filter(filtbank_dec.H%d, 1, d_ideal);',
         index-1, index-1));
    % Decimate by M
    eval(sprintf('u_dec.H%d = u_filt1.H%d(1:num_bands:end);', index-1,
        index-1));
    eval(sprintf('d_dec.H%d = d_filt1.H%d(1:num_bands:end);', index-1,
        index-1));
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% RLS Filtering

for index =  1:num_update_iters
    for index2 = 1:num_bands
        eval(sprintf('kal_gain = (P0*(u_dec.H%d(index:index+size(w_mat,
             2)-1)))/(forget_factor + (u_dec.H%d(index:index+size(w_mat,
             2)-1))''*P0*(u_dec.H%d(index:index+size(w_mat, 2)-1)));',
            index2-1, index2-1, index2-1));
        eval(sprintf('d_hat(index2, index) = (w_mat(index2, :))*(u_dec.
            H%d(index:index+size(w_mat, 2)-1));', index2-1));
        eval(sprintf('RLS_error(index2, index) = d_dec.H%d(index) -
            d_hat(index2, index);', index2-1));
        eval(sprintf('w_mat(index2, :) = w_mat(index2, :) + kal_gain.''
            *conj(RLS_error(index2, index));'));
        eval(sprintf('P0 = (eye(size(w_mat, 2)) - kal_gain*(u_dec.H%d(
            index:index+size(w_mat, 2)-1))'')*P0/forget_factor;', index2
            -1));
    end
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Filter/Interpolate Outputs:

d_out_sum = 0;
d_ideal_sum = 0;
for index = 1:num_bands
    % Interpolate by M: use function
    eval(sprintf('d_syn.H%d = interp_zeros(d_hat(%d, :), num_bands, 2);
        ', index-1, index));
    eval(sprintf('d_ideal_syn.H%d = interp_zeros(d_dec.H%d, num_bands,
        1);', index-1, index-1));
    % Filter through the kth synthesis bank filter
    eval(sprintf('d_filt2.H%d = filter(filtbank_syn.H%d, 1, d_syn.H%d);
        ', index-1, index-1, index-1));
    eval(sprintf('d_ideal_filt1.H%d = filter(filtbank_syn.H%d, 1,
        d_ideal_syn.H%d);', index-1, index-1, index-1));
    eval(sprintf('d_out_sum = d_out_sum + d_filt2.H%d;', index-1));
    eval(sprintf('d_ideal_sum = d_ideal_sum + d_ideal_filt1.H%d;',
        index-1));
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Specify Outputs:

output.out = d_out_sum;
output.ideal = d_ideal_sum;
output_error.full = d_ideal_sum(1:length(d_out_sum)).' - d_out_sum;
output_error.subbands = RLS_error;

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.15 Adjustable Non-Uniform Subband LMS Algorithm

```
function [output, output_error, fin_dec_vals] = subbandNULMS_adapt(u_in
    , d_ideal,...
    step_size, w0_len, num_bands_max, start_dec_factrs, PSD_mem,
        thresh_vals)

% Adam Charles
% 3/21/2009
%
% Adjustable Nonuniform Subband LMS Filtering.
%
% Filters through Filterbank with M = num_bands, then performs LMS
% filtering on each subband seperately. Those results are then put through
% the synthesis filter to get the final outpt y(n).
%
% For the matricies used, each row indicates the subband-1 (row 1 is
% subband 0), and the column represents the iteration. The updates are
% essentially round robin, as they start at H_0, work up to H_{M-1}, then
% go back to H_0.
%


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Error Checking


% Uniform Filter Bank can only have M = 2^L outputs
if 2^floor(log2(num_bands_max)) ≠ num_bands_max
    error('M must be a power of 2 for a uniform tree filter bank!');
end

for index_check = 1:length(start_dec_factrs)
    if 2^floor(log2(start_dec_factrs(index_check))) ≠ start_dec_factrs(index_check)
         error('Decimation Factors must be a power of 2 for a uniform tree filter bank!');
    end
end

% Make sure u_in is a vector
if size(u_in, 1) ≠ 1 && size(u_in, 2) ≠ 1
    error('u_in must be a vector!');
elseif size(u_in, 1) == 1 && size(u_in, 2) ≠ 1
    u_in = u_in.';
end

% Make sure d_ideal is a vector
if size(d_ideal, 1) ≠ 1 && size(d_ideal, 2) ≠ 1
    error('d_ideal must be a vector!');
elseif size(d_ideal, 1) == 1 && size(d_ideal, 2) ≠ 1
    d_ideal = d_ideal.';
end

```

```

if ¬isscalar(step_size)
    error('step_size must be a scalar!');
end

if sum(1./start_dec_factrs) ≠ 1
    error('Not a valid starting point!')
end

if floor(w0_len/num_bands_max) ≠ w0_len/num_bands_max
    error('w0_len must be divisable by the max number of bands!')
end

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Prior Calculations and Initializations

% Calculate number of 'time steps' or # times to updaye all filter
    banks
% LMS related cariables
num_update_iters = floor(min(length(u_in), length(d_ideal))/
    num_bands_max);
d_hat = zeros(num_update_iters, num_bands_max);
LMS_error = zeros(num_bands_max, num_update_iters);
dec_factors = start_dec_factrs;
n_per_band = num_bands_max*(dec_factors);
PSDin_est = zeros(1, 2*num_bands_max);
PSDout_est = zeros(1, 2*num_bands_max);
u_max_eff = num_bands_max*floor(length(u_in)/num_bands_max);

% Feedback related variables
PSD_mem_in = zeros(2*num_bands_max, PSD_mem);
PSD_mem_out = zeros(2*num_bands_max, PSD_mem);
PSD_array_in =zeros(num_bands_max, floor(num_update_iters/2));
PSD_array_out = zeros(num_bands_max, floor(num_update_iters/2));
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Set up Filter Banks and Initialize Memory:

[filtbank_bin_dec] = filt_bank_gen(2, 'uniform', 'f', 0);
[filtbank_bin_syn] = filt_bank_gen(2, 'uniform', 's', 0);

% Each synthesis/analysis filter needs length(filtbank_bin_dec) ammount
     of memory
% the i^th level has 2^i filters in it (2 for 1st level, 4 for second
    ...)
% First column cooresponds to the 'bottom' branch, or lowest spectral
    band.
```

```
for index = 1:log2(num_bands_max)
    eval(sprintf('analysis_data.level%d = zeros(floor(u_max_eff/(2^
        index))+w0_len/(2^index), 2^index);', index));
    eval(sprintf('ideal_analysis_data.level%d = zeros(floor(length(
        d_ideal)/(2^index)), 2^index);', index));
    eval(sprintf('ideal_synthesis_data.level%d = zeros(floor(u_max_eff
        /(2^index)), 2^index);', index));
    eval(sprintf('synthesis_data.level%d = zeros(floor(u_max_eff/(2^
        index)), 2^index);', index));
    eval(sprintf('LMS_error.level%d = zeros(floor(u_max_eff/(2^index)),
         2^index);', index));
    eval(sprintf('w_array.level%d = zeros(w0_len/(2^index), 2^index);',
         index));
end

eval(sprintf('analysis_data.level0 = [zeros(w0_len-1, 1); u_in];'));
eval(sprintf('ideal_analysis_data.level0 = d_ideal;'));
eval(sprintf('synthesis_data.level0 = zeros(u_max_eff, 1);'));
eval(sprintf('ideal_synthesis_data.level0 = zeros(u_max_eff, 1);'));
eval(sprintf('LMS_error.level0 = zeros(u_max_eff, 1);'));
eval(sprintf('w_array.level0 = zeros(w0_len, 2^index);'));


%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Get Signal at Each Node in the Analysis Tree

disp('Analysis Filter Start')
for index = 1:log2(num_bands_max)
    for index2 = 1:index
        % Input Analysis Processing
        eval(sprintf('temp_LP = filter(filtbank_bin_dec.H0, 1,
            analysis_data.level%d(:, index2));', index-1));
        eval(sprintf('temp_HP = filter(filtbank_bin_dec.H1, 1,
            analysis_data.level%d(:, index2));', index-1));
        eval(sprintf('analysis_data.level%d(:, (2*index2-1)) = temp_LP
            (1:2:end);', index));
        eval(sprintf('analysis_data.level%d(:, 2*index2) = temp_HP(1:2:
            end);', index));
        clear temp_HP temp_LP
        % Ideal Output Analysis Processing
        eval(sprintf('temp_DLP = filter(filtbank_bin_dec.H0, 1,
            ideal_analysis_data.level%d(:, index2));', index-1));
        eval(sprintf('temp_DHP = filter(filtbank_bin_dec.H1, 1,
            ideal_analysis_data.level%d(:, index2));', index-1));
        eval(sprintf('ideal_analysis_data.level%d(:, (2*index2-1)) =
            temp_DLP(1:2:end);', index));
        eval(sprintf('ideal_analysis_data.level%d(:, 2*index2) =
            temp_DHP(1:2:end);', index));
```

```matlab
        clear temp_DHP temp_DLP
    end
end
disp('Analysis Filter End')

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Run Algorithm

disp('LMS Filter Start')
for index =  1:num_update_iters
    % Number in dec_factors is decimation factor bands: also number of times
    % that band is updated per iteration. e.g. num_update_iters/[2, 4, 4] =
    % [2, 1, 1]...
    for index2 = 1:length(dec_factors)
        for index3 = 1:(num_bands_max/dec_factors(index2))
            %% LMS Algorithm
            % Data into the filter is from
            % analysis_data.level[log2(dec_factors(index2))]()
            % Ideal Output is from
            % ideal_analysis_data.level[log2(dec_factors(index2))]
            % Output goes into
            % synthesis_data.level[log2(dec_factors(index2))]
            % Filter is w_array.level[log2(dec_factors(index2))]
            % Branch number is 1 + sum(1./dec_factors(1:index2))*dec_factors(index2)
            dec_factors2 = [inf, dec_factors];
            branch_num = 1 + sum(1./dec_factors2(1:index2))*dec_factors(index2);
            u_pos_back =  (index-1)*num_bands_max/dec_factors(index2) + index3;
%             dec_factors;
            u_pos = u_pos_back + w0_len/(dec_factors(index2));
            % Evaluate output
            eval(sprintf('ideal_synthesis_data.level%d(u_pos_back, branch_num) = ideal_analysis_data.level%d(u_pos_back, branch_num);'...
                , log2(dec_factors(index2)), log2(dec_factors(index2))))
            eval(sprintf('synthesis_data.level%d(u_pos_back, branch_num) = w_array.level%d(:, branch_num)''*analysis_data.level%d((u_pos_back+1):u_pos, branch_num);'...
                , log2(dec_factors(index2)), log2(dec_factors(index2)), log2(dec_factors(index2)) ));
            % Get error
            eval(sprintf('LMS_error.level%d(u_pos_back, branch_num) = ideal_analysis_data.level%d(u_pos_back, branch_num) -
                synthesis_data.level%d(u_pos_back, branch_num);'...
                , log2(dec_factors(index2)), log2(dec_factors(index2)), log2(dec_factors(index2)) ));
            % Update tap weights:
            eval(sprintf('w_array.level%d(:, branch_num) = w_array.level%d(:, branch_num) + step_size*analysis_data.level%d((u_pos_back+1):u_pos, branch_num)*conj(LMS_error.level%d(u_pos_back, branch_num));'...
                , log2(dec_factors(index2)), log2(dec_factors(index2)), log2(dec_factors(index2)), log2(dec_factors(index2)) ));
        end
    end
    %% Place Feedback HERE
    %Update PSD arrays every other full iteration (sice we need
    %2*num_bands_max) new samples for new window.
    if index == 2*floor(index/2) && (index + 2)*num_bands_max - 1 ≤ length(u_in)
        % Calculate net FFT^2 for pwelch-type PSD estimate:
        temp_PSDin = fftshift(abs(fft(u_in(index*num_bands_max:(index+2)*num_bands_max-1))).^2);
        temp_PSDout = fftshift(abs(fft(d_ideal(index*num_bands_max:(index+2)*num_bands_max-1))).^2);
        % Update PSD estimate memory array
        PSD_mem_in = [temp_PSDin, PSD_mem_in(:, 1:(end-1))];
        PSD_mem_out = [temp_PSDout, PSD_mem_out(:, 1:(end-1))];
        % Take mean of past K FFT^2 to get PSD approximate at time N
        PSD_array_in(:, index) = mean(PSD_mem_in(num_bands_max+1:2*num_bands_max, :), 2);
        PSD_array_out(:, index) = mean(PSD_mem_out(num_bands_max+1:2*num_bands_max, :), 2);
%         PSD_array_out(:, index)./PSD_array_in(:, index)
        % And now the important part: make a decision for the new
        % dec_factors vector!! This is a function of the PSD_INest,
        % PSD_OUTest, current dec_factorsand possible some black magic.
        [dec_factors_new, w_new] = subband_update(dec_factors, w_array,...
            PSD_array_in(:, index), PSD_array_out(:, index), thresh_vals,...
            num_bands_max, filtbank_bin_syn.H0, filtbank_bin_syn.H1);
        dec_factors = dec_factors_new;
        w_array = w_new;
    end
%     if index == 100
%         dec_factors = [8 8 8 8 4 4];
%         w_array.level1(:, 2) = eff_filt(w_array.level2(:, 4).', w_array.level2(:, 3).'...
%             , filtbank_bin_dec.H0, filtbank_bin_dec.H1, filtbank_bin_syn.H0, filtbank_bin_syn.H1).';
%     end
```

```
end
disp('Analysis Filter End')

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Filter/Interpolate Outputs:

disp('Synthesis Filter Start')
% Rebuild signals: start from the outside in...
for index = log2(num_bands_max):-1:1
    for index2 = 1:index
        % Output Analysis Processing
        eval(sprintf('temp_LP = [interp_zeros(synthesis_data.level%d(:,
          (2*index2-1)), 2, 1); 0];', index));
        eval(sprintf('temp_HP = [interp_zeros(synthesis_data.level%d(:,
          2*index2), 2, 1); 0];', index));
        eval(sprintf('synthesis_data.level%d(:, index2) =
            synthesis_data.level%d(:, index2) + filter(filtbank_bin_dec.
            H0, 1, temp_LP) + filter(filtbank_bin_dec.H1, 1, temp_HP);'
            ...
            , index-1, index-1));
        clear temp_HP temp_LP
        % Ideal Output Analysis Processing
        eval(sprintf('temp_DLP = [interp_zeros(ideal_synthesis_data.
            level%d(:, (2*index2-1)), 2, 1); 0];', index));
        eval(sprintf('temp_DHP = [interp_zeros(ideal_synthesis_data.
            level%d(:, 2*index2), 2, 1); 0];', index));
        eval(sprintf('ideal_synthesis_data.level%d(:, index2) =
            ideal_synthesis_data.level%d(:, index2) + filter(
            filtbank_bin_dec.H0, 1, temp_DLP) + filter(filtbank_bin_dec.
            H1, 1, temp_DHP);'...
            , index-1, index-1));
        clear temp_DHP temp_DLP
    end
end
disp('Synthesis Filter End')

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Specify Outputs:

fin_dec_vals = dec_factors;
output.out = synthesis_data;
output.ideal = ideal_synthesis_data;
output.PSD_INest = PSD_array_in;
output.PSD_OUTest = PSD_array_out;
output_error.full = ideal_synthesis_data.level0(1:length(
    ideal_synthesis_data.level0)) - synthesis_data.level0;
```

```
output_error.subbands = LMS_error;

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.16 Adjustable Non-Uniform Subband Update Algorithm

```
function [dec_factors_new, w_new] = subband_update(dec_factors_old, w_old, PSD_array_in, PSD_array_out, thresh_vals, max_bands, H0, H1)

%
% Adam Charles
% 3/24/2009
%
% Decision algorithm for updating subband decimation factors
%

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%
w_new = w_old;
% dec_factors_new = 0;

% PSD_in0 = find(PSD_array_in == 0);
% PSD_array_in(PSD_in0) = 10^(-5);
PSDratio = (PSD_array_out + 0.0001)./(PSD_array_in + 0.001);

% index_low = find(PSDratio < thresh_vals(1));
% index_high = find(PSDratio > thresh_vals(2));
% power_weights = zeros(length(dec_factors_old), 1);
% power_weights(index_low) = -2;
% power_weights(index_high) = 1;

branch_test = 0;
df_new_index = 1;

for index = 1:length(dec_factors_old)
    index_start = max_bands*sum(1./dec_factors_old(1:index-1)) + 1;
    band_power(index) = mean(PSDratio(index_start:index_start-1+max_bands/(dec_factors_old(index))));
    if branch_test == 0 && index ≠ length(dec_factors_old)
        band_test = ceil(dec_factors_old(index)*sum(1./dec_factors_old
            (1:index))/2) - ceil(dec_factors_old(index+1)*sum(1./
            dec_factors_old(1:index+1))/2); % dec_factors_old(index+1)*
        if dec_factors_old(index) == dec_factors_old(index+1) &&
            band_test == 0
            branch_test = 1;
        else% if dec_factors_old(index) ≠ dec_factors_old(index+1) ||
            branch_test ≠ 0
            if band_power(index) < thresh_vals(1) && dec_factors_old(
                index) < max_bands
                % Decompose into 2 bands
                dec_factors_new(df_new_index:df_new_index+1) = 2*[1,1]*
                    dec_factors_old(index);
                df_new_index = df_new_index+2;
            else
                dec_factors_new(df_new_index) = dec_factors_old(index);
                df_new_index = df_new_index+1;
            end
        end
    elseif branch_test == 0 && index == length(dec_factors_old)
        if band_power(index) < thresh_vals(1) && dec_factors_old(index)
             < max_bands
            % Decompose into 2 bands
            dec_factors_new(df_new_index:df_new_index+1) = 2*[1,1]*
                dec_factors_old(index);
            df_new_index = df_new_index+2;
        else
            dec_factors_new(df_new_index) = dec_factors_old(index);
            df_new_index = df_new_index+1;
        end
    elseif branch_test == 1
        if (0.5*(band_power(index) + band_power(index-1))> thresh_vals
            (2)) && (dec_factors_old(index) > 2)
            % Consolodate Branch
%             index
%             dec_factors_old
            dec_factors_new(df_new_index) = 0.5*dec_factors_old(index);
            band_num = dec_factors_old(index)*sum(1./dec_factors_old(1:
                index));
            eval(sprintf('w_new.level%d(:, 0.5*(band_num)) = eff_filt(
                w_old.level%d(:, band_num-1).'', w_old.level%d(:,
                band_num).'', H0, H1, H0, H1).'';'...
                , log2(dec_factors_old(index))-1, log2(dec_factors_old(
                    index)), log2(dec_factors_old(index)) ))
            df_new_index = df_new_index + 1;
        elseif (band_power(index-1)< thresh_vals(1))&&(band_power(index
            )< thresh_vals(1))&&(dec_factors_old(index-1) < max_bands)
            % decompose both
            dec_factors_new(df_new_index:df_new_index+1) = 2*[1,1]*
                dec_factors_old(index-1);
            dec_factors_new(df_new_index+2:df_new_index+3) = 2*[1,1]*
```

```
                dec_factors_old(index);
            df_new_index = df_new_index+4;
        elseif (band_power(index-1)< thresh_vals(1))&&(dec_factors_old(
            index-1) < max_bands)
            % decompose index-1
            dec_factors_new(df_new_index:df_new_index+1) = 2*[1,1]*
                dec_factors_old(index-1);
            dec_factors_new(df_new_index+2) = dec_factors_old(index);
            df_new_index = df_new_index+3;
        elseif (band_power(index)< thresh_vals(1))&&(dec_factors_old(
            index) < max_bands)
            % decompose index
            dec_factors_new(df_new_index) = dec_factors_old(index-1);
            dec_factors_new(df_new_index+1:df_new_index+2) = 2*[1,1]*
                dec_factors_old(index);
            df_new_index = df_new_index+3;
        else
            % leave it alone...
            dec_factors_new(df_new_index) = dec_factors_old(index-1);
            dec_factors_new(df_new_index+1) = dec_factors_old(index);
            df_new_index = df_new_index+2;
        end
        branch_test = 0;
    end



end
```

A.17 Test Filter Construction Code

```
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Low-Pass Filter
ftest_lpspec.fp= 1/4;                   % pass-band frequency as
    calculated above
ftest_lpspec.fst= 1/4 + 1/32;           % stop-band frequency is 20Hz
    above pass-band
ftest_lpspec.Ap= 0.1;                   % worst pass-band attenuation is
    1db
ftest_lpspec.Ast= 60;                   % best stop-band attenuation is
    20db
% generate actual filter from specs
ftest_lpspecs= fdesign.lowpass('fp,fst,Ap,Ast', ...
    ftest_lpspec.fp, ftest_lpspec.fst, ftest_lpspec.Ap, ftest_lpspec.
        Ast);
ftest_lp= design(ftest_lpspecs,'kaiserwin');

```

```

% fvtool(ftest_lp)

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Band-Pass Filter
ftest_bpspec.fst1 = 1/3;
ftest_bpspec.fp1 = 1/3 + 1/27;
ftest_bpspec.fp2 = 1/3 + 1/17 + 1/4;
ftest_bpspec.fst2 = 1/3 + 1/17 + 1/2 + 1/27;
ftest_bpspec.ast1 = 60;
ftest_bpspec.ap = 0.1;
ftest_bpspec.ast2 = 70;
ftest_bpspecs= fdesign.bandpass('fst1,fp1,fp2,fst2,ast1,ap,ast2',...
    ftest_bpspec.fst1, ftest_bpspec.fp1, ftest_bpspec.fp2, ftest_bpspec.fst2,...
    ftest_bpspec.ast1, ftest_bpspec.ap, ftest_bpspec.ast2);
ftest_bp= design(ftest_bpspecs,'kaiserwin');


% fvtool(ftest_bp)

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% High-Pass Filter
ftest_hpspec.fst= 25/32;                 % pass-band frequency as calculated above
ftest_hpspec.fp= 13/16;         % stop-band frequency is 20Hz above pass-band
ftest_hpspec.Ast= 60;                 % worst pass-band attenuation is 1db
ftest_hpspec.Ap= 0.1;                 % best stop-band attenuation is 20db
% generate actual filter from specs
ftest_hpspecs= fdesign.highpass('fst,fp,ast,ap', ...
    ftest_hpspec.fst, ftest_hpspec.fp, ftest_hpspec.Ast, ftest_hpspec.Ap);
ftest_hp= design(ftest_hpspecs,'kaiserwin');

% fvtool(ftest_hp)

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Band-Stop Filter

ftest_bsspec.fp1 = 1/3;
ftest_bsspec.fst1 = 1/3 + 1/27;
ftest_bsspec.fst2 = 1/3 + 1/17 + 1/4;
ftest_bsspec.fp2 = 1/3 + 1/17 + 1/2 + 1/27;
ftest_bsspec.ap1 = 0.1;
ftest_bsspec.ast = 60;
ftest_bsspec.ap2 = 0.2;
ftest_bsspecs= fdesign.bandstop('fp1,fst1,fst2,fp2,ap1,ast,ap2',...
    ftest_bsspec.fp1, ftest_bsspec.fst1, ftest_bsspec.fst2,
        ftest_bsspec.fp2,...
    ftest_bsspec.ap1, ftest_bsspec.ast, ftest_bsspec.ap2);
ftest_bs= design(ftest_bsspecs,'kaiserwin');

% fvtool(ftest_bs)

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.18 Subband Algorithm Testing Code

```
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Subband Adaptive Filtering Testing Code
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear
clc
close all
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Choose which filters to try
lms_uniform_opt = 0;
lms_reg_opt = 0;
rls_uniform_opt = 0;
rls_reg_opt = 0;
lms_nonuniform_opt = 1;

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Make testing data
n_max = 32000;
% H = poly(rand(60, 1));
% H = H./norm(H);

% Test Filters
filt_test_create
```

```

Hlp = ftest_lp.Numerator;
Hhp = ftest_hp.Numerator;
Hbp = ftest_bp.Numerator;
Hbs = ftest_bs.Numerator;
% fvtool(ftest_lp)
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Make Options

step_size = 0.01;
num_bands = 16;
num_weights = ceil(length(Hbs)/num_bands)*num_bands;
num_trials = 1;
band_type = 'uniform';
w_start = zeros(num_bands, num_weights/num_bands);
w_start_LMS = zeros(1, num_weights);
forget_factor = 0.99;
△ = 0.9;
thresh_vals = [0.1, 0.5]/4; %10^(12/20)*10.^(-[50;15]/20);
for index2 = 1:log2(num_bands)
    eval(sprintf('output_error_NULMS.subbands.level%d = 0;', index2))
end
output_error_LMS.full = 0;
output_error_LMS.subbands = 0;
output_error_NULMS.full = 0;
LMS_error = 0;

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Run tests

for index = 1:num_trials
    disp(sprintf('Iteration %d', index))
    u_in = 0.6*randn(n_max, 1);
    [sys_noise] = AFinput_noise(length(u_in), [0, 0.01], 5000, [1, 30],
         [1,0]);
    d_ideal = [ filter(Hbp, 1, u_in(1:n_max/2)); filter(Hlp, 1, u_in(
        n_max/2+1:end))] + sys_noise;
%      d_ideal = filter(Hbs, 1, u_in) + sys_noise;

if lms_uniform_opt
    [output_LMS, output_error_LMS_temp] = subbandLMSsimple2(u_in,
        d_ideal,step_size, w_start, num_bands, band_type);
    output_error_LMS.full = output_error_LMS.full + abs(
        output_error_LMS_temp.full)/num_trials;
    output_error_LMS.subbands = output_error_LMS.subbands + abs(
        output_error_LMS_temp.subbands)/num_trials;
```

```
end
if lms_reg_opt
    [d_hat_LMS, LMS_error_temp, w_mat] = standardLMS(u_in, d_ideal,
        w_start_LMS, step_size, 7, 0);
    LMS_error = LMS_error + abs(LMS_error_temp)/num_trials;
    clear LMS_error_temp
end
if rls_uniform_opt
    [output, output_error] = subbandRLSsimple2(u_in, d_ideal,
        forget_factor, △, w_start, num_bands);
    output_error.full = output_error.full + abs(output_error.full)/
        num_trials;
    for index2 = 1:log2(num_bands)
        eval(sprintf('output_error.subbands.level%d = output_error.
            subbands.level%d + abs(output_error.subbands.level%d)/
            num_trials;'...
            , index2, index2, index2))
    end
end
if rls_reg_opt
    [d_hat_RLS, Xi, w] = standardRLS(u_in, d_ideal, w_start_LMS,
        forget_factor, △, 8, 0);
end
if lms_nonuniform_opt
    w0_len = num_weights;
    num_bands_max = num_bands;
    start_dec_factrs = [2 2];
    PSD_mem = 50;
    [output_NULMS, output_error_NULMS_temp, fin_dec_vals] =
        subbandNULMS_adapt(u_in, d_ideal,step_size,...
        w0_len, num_bands_max, start_dec_factrs, PSD_mem, thresh_vals);
    output_error_NULMS.full = output_error_NULMS.full + abs(
        output_error_NULMS_temp.full)/num_trials;
    for index2 = 1:log2(num_bands)
        eval(sprintf('output_error_NULMS.subbands.level%d =
            output_error_NULMS.subbands.level%d + abs(
            output_error_NULMS_temp.subbands.level%d)/num_trials;'...
            , index2, index2, index2))
    end
    clear output_error_NULMS_temp
end

end
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Plotting

if rls_uniform_opt && rls_reg_opt
    figure;
    subplot(3,1,1), plot(1:size(output_error.subbands, 2), abs(Xi(1:
        size(output_error.subbands, 2))))
    xlabel('Iteration Number, n', 'FontSize', 12, 'FontName', 'Times');
    ylabel('RLS Absolute Error, |e[n]|', 'FontSize', 12, 'FontName', '
        Times');
    subplot(3,1,2), plot(1:length(output_error.full), abs(output_error.
        full.'), 'b', 1:length(Xi), abs(Xi), '--r')
    xlabel('Iteration Number, n', 'FontSize', 12, 'FontName', 'Times');
    ylabel('RLS Absolute Error, |e[n]|', 'FontSize', 12, 'FontName', '
        Times');
    legend('Subband RLS', 'Fullband RLS')
    subplot(3,1,3), plot(abs(output_error.subbands.'))
    xlabel('Iteration Number, n', 'FontSize', 12, 'FontName', 'Times');
    ylabel('RLS Absolute Error, |e[n]|', 'FontSize', 12, 'FontName', '
        Times');
    legend('Subband 1', 'Subband 2', 'Subband 3', 'Subband 4', 'Subband
         5', 'Subband 6', 'Subband 7', 'Subband 8');
end

if lms_uniform_opt && lms_reg_opt
    figure;
    subplot(3,1,1), plot(1:length(output_error_LMS.full), abs(
        output_error_LMS.full.'), 'b', 1:length(LMS_error), abs(
        LMS_error), '--r')
    xlabel('Iteration Number, n', 'FontSize', 12, 'FontName', 'Times');
    ylabel('LMS Absolute Value, |e[n]|', 'FontSize', 12, 'FontName', '
        Times');
    legend('Subband LMS', 'Fullband LMS')
    subplot(3,1,2), plot(1:size(output_error_LMS.subbands, 2), abs(
        LMS_error(1:size(output_error_LMS.subbands, 2))))
    xlabel('Iteration Number, n', 'FontSize', 12, 'FontName', 'Times');
    ylabel('LMS Absolute Error, |e[n]|', 'FontSize', 12, 'FontName', '
        Times');
    subplot(3,1,3), plot(abs(output_error_LMS.subbands.'))
    xlabel('Iteration Number, n', 'FontSize', 12, 'FontName', 'Times');
    ylabel('LMS Absolute Error, |e[n]|', 'FontSize', 12, 'FontName', '
        Times');
    legend('Subband 1', 'Subband 2', 'Subband 3', 'Subband 4', 'Subband
         5', 'Subband 6', 'Subband 7', 'Subband 8');
end

if lms_nonuniform_opt
    figure;
    subplot(4,1,1), plot((abs(output_error_NULMS.full)))
    subplot(4,1,2), plot((abs(output_error_NULMS.subbands.level1)))
    subplot(4,1,3), plot((abs(output_error_NULMS.subbands.level2)))
    subplot(4,1,4), plot(abs(output_error_NULMS.subbands.level3))
end

if lms_nonuniform_opt %&& lms_reg_opt
```

```
    figure; hold on
    plot((abs(output_error_NULMS.full)))
    plot((abs(output_error_NULMS2.full)), '--r')
    xlabel('Iteration Number n', 'FontSize', 12, 'FontName', 'Times')
    ylabel('LMS Absolute Error, |e[n]|', 'FontSize', 12, 'FontName', '
        Times')
    legend('Sub-optimal Decomposition', 'Optimal Decomposition')
    hold off

    figure; hold on
    subplot(4, 1, 1), plot((abs(output_error_NULMS.full)))
    xlabel('Iteration Number n', 'FontSize', 12, 'FontName', 'Times')
    ylabel('LMS Absolute Error, |e[n]|', 'FontSize', 12, 'FontName', '
        Times')
    subplot(4, 1, 2), plot((abs(output_error_NULMS.subbands.level1)))
    xlabel('Half Rate Iteration Number n', 'FontSize', 12, 'FontName',
        'Times')
    ylabel('LMS Absolute Error, |e[n]|', 'FontSize', 12, 'FontName','
        Times')
    subplot(4, 1, 3), plot((abs(output_error_NULMS.subbands.level2)))
    xlabel('Quarter Rate Iteration Number n', 'FontSize', 12, 'FontName
        ', 'Times')
    ylabel('LMS Absolute Error, |e[n]|', 'FontSize', 12, 'FontName','
        Times')
    subplot(4, 1, 4), plot(20*log10(abs(output_error_NULMS.subbands.
        level2)))
    xlabel('Eigth Rate Iteration Number n', 'FontSize', 12, 'FontName',
         'Times')
    ylabel('LMS Absolute Error, |e[n]|', 'FontSize', 12, 'FontName','
        Times')

    figure; hold on
    plot(20*log10(abs(output_error_NULMS.full))) %plot(output_NULMS.out
        .level0)%
    plot(20*log10(abs(output_error_NULMS2.full)), '--r')
    xlabel('Iteration Number n', 'FontSize', 12, 'FontName', 'Times')
    ylabel('LMS Absolute Error, |e[n]|', 'FontSize', 12, 'FontName', '
        Times')
    legend('Sub-optimal Decomposition', 'Optimal Decomposition')
    hold off
end


figure;
xPSDvals = num_bands_max*(1:size(output_NULMS.PSD_OUTest, 2));
yPSDvals = pi*((1:size(output_NULMS.PSD_OUTest, 1))-1)/num_bands_max;
surf(xPSDvals, yPSDvals, (output_NULMS.PSD_OUTest+0.001)./(output_NULMS
    .PSD_INest + 0.01))
xlabel('Iteration Number n', 'FontSize', 12, 'FontName', 'Times')
ylabel('Frequency Band', 'FontSize', 12, 'FontName', 'Times')
```

```
zlabel('PSD Ratio Estimate', 'FontSize', 12, 'FontName', 'Times')
```

Bibliography

[1] G. K. Singh A. Kumar and R. S. Anand. Near perfect reconstruction quadrature mirror filter. *Proceedings of World Academy of Science, Engineering and Technology*, 27:204–207, February 2008.

[2] S. K. Mitra A. Mahalanobis, S. Song and M. R. Petraglia. Adaptive fir filters based on structural subband decomposition for system identification problems. *IEEE Transactions on Circuits and Systems*, 40(6):375–381, June 1993.

[3] M. A. Aldajani. Adaptive step size sign least mean squares. *IEEE ICASSP*, pages 996–672, 2004.

[4] M. R. Petralgia F. S. Hallack. Performance comparison of adaptive algorithms applied to acoustic echo cancelling. *IEEE*, pages 1147–1150, 2003.

[5] A. Gilloire and M. Vetterli. Adaptive filtering in sub-bands. *IEEE*, pages 1572–1575, 1988.

[6] A. Gilloire and M. Vetterli. Adaptive filtering in subbands with critical sampling: Analysis, experiments, and application to acoustic echo cancellation. *IEEE Transactions on Signal Processing*, 40(8):1862–1875, August 1992.

[7] S. Haykin. *Adaptive Filter Theory.* Prentice-Hall, fourth edition, 2002.

[8] T. Bose J. D. Grieshbach and D. M. Etter. Non-uniform filterbank bandwidth allocation for system modeling subband adaptive filters. *IEEE*, pages 1473–1476, 1999.

[9] D. E. Knuth. *The Art of Computer Programming Volume 2; Seminumerical Algorithms.* Addison-Wesley, third edition, 1998.

[10] S. Koike. Adaptive threshold nonlinear algorithm for adaptive filters with robustness against impulse noise. *IEEE Transactions on Signal Processing*, 45(9):2391–2395, September 1997.

[11] S. Koike. Adaptive forgetting factor recursive least squares adaptive threshold nonlinear algorithm (aff-rls-atna) for identification of nonstationary systems. *IEEE Transactions*, 3(3):617–620, 2003.

[12] R. D. Koipillai and P. P. Vaidyanathan. Cosine-modulated fir filter banks satisfying perfect reconstruction. *IEEE Transactions on Signal Processing*, 40(4):770–783, April 1992.

[13] V. Krishnamurthy and S. Singh. Adaptive forgetting factor recursive least squares for blind interference suppression in ds/cdma systems. *IEEE*, pages 2469–2472, 2000.

[14] R. T. B. Vasconcellos M. R. Petralgia and R. G. Alves. Performance comparison of adaptive subbands and structures applied to acoustic echo cancelling. *IEEE*, pages 1535–1539, 2001.

[15] M. L. McCloud and D. M. Etter. Subband adaptive filtering with time-varying nonuniform filter banks. *IEEE*, pages 1953–1956, 1997.

[16] M. L. McCloud and D. M. Etter. A technique for nonuniform subband adaptive filtering with varying bandwith filter banks. *IEEE*, pages 1315–1318, 1997.

[17] A. Oakman and P. Naylor. Dynamic structures for non-uniform subband adaptive filtering. *IEEE Transactions*, pages 3717–3720, 2001.

[18] K. K. Parhi. *VLSI Digital Signal Processing Systems, Design and Implementation.* John Wiley & Sons, Inc, 1999.

[19] M. R. Petraglia and P. B. Batalheiro. Filter bank design for a subband adaptive filtering structure with critical sampling. *IEEE Transactions on Circuits and Systems*, 51(6):1194–1202, June 2004.

[20] M. R. Petraglia and P. B. Batalheiro. Nonuniform subband adaptive filtering with critical sampling. *IEEE Transactions*, pages 565–575, 2008.

[21] M. R. Petraglia and S. K. Mitra. Adaptive fir filter structure based on the generalized subband decomposition of fir filters. *IEEE Transactions on Circuits and Systems*, 40(6):354–362, June 1993.

[22] M. R. Petraglia and S. K. Mitra. Performance analysis of adaptive filter structures based on subband decomposition. *IEEE Transactions*, 6:60–63, 1993.

[23] J. A. Apolinario Jr. R. G. Alves, M. R. Petraglia and P. S. R. Diniz. Rls algorithm for a new subband adaptive structure with critical sampling. *IEEE Transactions*, pages 442–447, 1998.

[24] A. Balasubramanian R. W. Wies and J. W. Pierre. Using adaptive step-size least mean squares (aslms) for estimating low-frequency electromechanical modes in power systems. *IEEE*, pages 1–8, 2006.

[25] P. P. Vaidyanathan. *Multirate Filters and Filter Banks.* Prentice-Hall, 1993.

[26] M. Vetterli. A theory of multirate filter banks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(3):1572–1575, March 1987.

[27] P. D. Welch. The use of fast fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. *IEEE Transactions on Audio and Electronics*, 14(2):70–73, June 1967.

[28] S. Kinjo Y. Higa, H. Ochi. A subband adaptive filter with the statistically optimum analysis filter bank. *IEEE Transactions on Circuits and Systems*, 45(8):1150–1154, August 1998.