8-2002

Decomposition of High-Order FIR Filters and Minimum-Phase Filter Design

Wei Su

University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by Wei Su entitled "Decomposition of High-Order FIR Filters and Minimum-Phase Filter Design." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

L. Montgomery Smith, Major Professor

We have read this thesis and recommend its acceptance:

Bruce Bomar, Bruce Whitehead

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Wei Su entitled “Decomposition of High-Order FIR Filters and Minimum-Phase Filter Design.” I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

L. Montgomery Smith
Major Professor

We have read this thesis and
recommend its acceptance:

Bruce Bomar

Bruce Whitehead

Accepted for the Council:

Anne Mayhew
Vice Provost and Dean of Graduate Studies

(Original signatures are on file with official student records.)

DECOMPOSITION OF HIGH-ORDER FIR FILTERS AND MINIMUM-PHASE FILTER DESIGN

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Wei Su

August 2002

DEDICATION

To my family and friends

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to Dr. Monty Smith, Dr. Bruce Bomar, and Dr. Bruce Whitehead for their assistance in the preparation of this manuscript. Their patience and guidance played an important role towards the completion of this manuscript. Additionally, other faculty members and staff of the Electrical Engineering Department of The University of Tennessee Space Institute also enriched my knowledge in electrical engineering.

ABSTRACT

In this study, the implementation of high-order FIR filter decomposition and minimum-phase filter design is investigated. One method is presented for decomposing arbitrary linear-phase FIR filters with distinct roots into the cascade of first-order, second-order and fourth-order subfilters. The other method is described for transforming nonrecursive filters with even-order, equal-ripple attenuation in the pass-band, stop-band and linear-phase into those with minimum-phase and half the degree, and again with equal-ripple attenuation in the pass-band and stop-band. The technique consists of quick and accurate polynomial root finding of the z-domain filter transfer function by searching a finite region in the complex z-plane, and separating the zeros in the complex z-domain. In FIR filter decomposition, the search of roots to determine the subfilter impulse response coefficients is restricted to distinct roots in four regions in the complex z-plane: on the real axis, on the unit circle, inside the unit circle and at (1, 0) or (-1, 0). In minimum-phase filter design, the search of roots is restricted in two categories: on the unit circle and inside the unit circle. In both methods, we used Lang's root finding program to get the zeros of the FIR filter.

Arbitrary FIR filters were designed and decomposed for all possible orders of subfilters. FIR filters with even-order, zero-phase and equal-ripple were designed and generated the half degree minimum-phase filters. Both methods have been tested on FIR filters with orders ranging to over 500 and have proven effective in decomposing filters to the cascade realization and designing minimum-phase filters.

TABLE OF CONTENTS

CHAPTER **PAGE**

LIST OF FIGURES

FIGURE **PAGE**

LIST OF TABLES

TABLE **PAGE**

CHAPTER 1

INTRODUCTION

Finite-duration Impulse Response (FIR) linear-phase digital filters have several important properties which make them attractive for digital signal processing applications. Among these features are exact linear phase and the absence of any stability problems as are encountered in infinite impulse response filters. The design of linear phase frequency-selective digital filters has a well-established solution based on the work of Parks and McClellan in 1972 [1]. Since finite impulse response (FIR) digital filters guarantee stability and exhibit linear phase, they are commonly used.

Due to recent advances in digital electronics in the form of programmable logic devices (PLD) and field programmable gate arrays (FPGA), digital filters can be realized in a low-cost, high-speed implementation. However, for high-speed applications using these devices, filters are restricted to low order. To realize high-order FIR filters at high-speed using devices such as PLDs and FPGAs, the filter must be realized by the cascade connection of low-order subfilters which can operate in parallel.

The advantage of linear-phase filters is that the group delay is constant for all frequencies. In other words, there is no delay distortion for linear-phase filters. The problem with linear-phase filters, however, is that this constant delay is always equal to $(N-1)/2$, where N is the filter length. When a large attenuation is required in the stop

band and a sharp cutoff is desired, N must be quite large. Thus, linear-phase filters with large stop-band attenuation and a sharp cutoff must have a large, but constant, delay. This large delay could be a major drawback of a filter, for it could cause instability if the filter was inside a feedback loop in a digital control system. Thus, in these cases, the delay of these filters is intolerably high. When a filter with less delay is desired, the minimum-phase filter is a good choice. Hence the use of a minimum-phase filter is potentially advantageous in several applications, and their design has been treated extensively in the literature [2-5].

In this paper we address the following two problems: (a) Decomposition of high-order FIR filters. (b) Given a linear phase FIR digital filter transfer function, determine a FIR digital filter which has the square root of its magnitude response but is of minimum phase.

In [6], Lian and Lim addressed finding initial estimates for the zeros of a symmetric linear-phase FIR filter. Their method involved evaluating the frequency response of a given filter on a dense grid and approximating root values based on the angular locations of stop-band zeros and pass-band minima. This method was restricted to filters with well-defined pass-bands and stop-bands, and its effectiveness in completely decomposing FIR filters was not discussed. In [7], L. M. Smith addressed a method to decompose an FIR filters for implementation by the cascade connection of subfilters. This method reduced the problem to searching for roots on the real axis, on the unit circle and inside the unit circle in the complex z-plane. This method was restricted to even-order and zero-phase FIR filters and it was capable of decomposing FIR filters with orders ranging to 120.

In the somewhat related problem of minimum-phase filter design, O. Herrmann and H. W. Schuessler introduced a method to transform nonrecursive filters with equal-ripple attenuation in the pass-band, stop-band and linear-phase into those with minimum-phase and half the degree, but again with equal-ripple attenuation in the pass-band and stop-band [2]. This method modified the linear-phase, equal-ripple FIR filter to get double zeros on the unit circle in the complex z-plane. Then the roots inside the unit circle and simple roots on the unit circle were chosen as the roots of the minimum-phase filter. Also, this method was restricted to low-order FIR filters. In high-order FIR filters, the zeros on the unit circle and inside the unit circle are very close to the unit circle. This method didn't solve how to separate these zeros.

Root finding is very important both in decomposition of FIR filters and in minimum-phase filter design. The transfer function of an FIR digital filter, in the z-domain, is simply a polynomial of certain order [8]. The realization of FIR digital filters by the cascade connection of subfilters and the implementation of minimum-phase filter design both require the successful decomposition of the transfer function polynomial into its zeros or roots. Although root finding methods and commercial tools have long been readily available, they are limited in their ability to identify roots of high-order polynomials [9]. Lian and Lim have characterized the zeros of high-order FIR digital filters and outlined a method by which zeros can be evaluated [6]. More recently, a new method developed by Markus Lang was shown to find roots for polynomials rapidly and accurately [10]. This method did not construct a new algorithm with nice theoretical properties. Instead, it basically consisted of a combination of two well-known methods: Muller's method and Newton's Method. It used Muller's method to get the initial

estimate and used this result as the initial value of Newton's method. His program could find the roots of high-order FIR filters successfully for filter orders up to and beyond 400.

In this study, a method is presented for decomposing any arbitrary FIR filter with distinct roots into the cascade connection of first, second and fourth order subfilters. The arbitrary FIR filter includes even-order, zero-phase FIR filters, odd-order, zero-phase FIR filters, even-order, anti-symmetric FIR filters and odd-order, anti-symmetric FIR filters. The values of the roots are found using Lang's method. The search of roots to determine the subfilter impulse response coefficients is restricted to distinct roots in four regions in the complex z-plane:

1. the real axis except for (1, 0) and (-1, 0).
2. the perimeter of the unit circle except for (1, 0) and (-1, 0).
3. the interior of the unit circle.
4. the position at (1, 0) and (-1, 0) .

We use the radii and the angles of the zeros in z-plane to separate all the zeros.

A related method is presented for minimum-phase filter design. From the impulse response coefficients of an even-order, zero-phase prototype filter, we can draw the curve of the frequency response magnitude $A_e\left(e^{j\omega}\right)$ and get the stop-band tolerance δ_2. We modify the prototype filter by $\frac{A_e\left(e^{j\omega}\right)+\delta_2}{1+\delta_2}$ and get the frequency response coefficients and the frequency response magnitude of the modified filter. On the curve of the frequency response magnitude, we choose the minima whose magnitudes are very close to zero. There are zeros on the unit circle with related frequency response $\omega[n]$ in

complex z-plane, these zeros are chosen as the zeros on the unit circle of the minimum-phase filter. We use Lang's program to find all the zeros of the modified filter. From [2], we can find the double zeros on the unit circle whose frequency responses are very close to $\omega[n]$. From all the zeros, we choose the zeros whose radii are less than one, then eliminate double zeros on the unit circle. The results are the zeros inside the unit circle of the minimum-phase filter. After all zeros of minimum-phase filter are found, the impulse response coefficients of the minimum-phase filter are computed and stored.

Two computer programs were written to implement decomposition of high-order FIR filter and minimum-phase filter design. In the FIR filter decomposition program, we input the coefficients of the FIR filter and output the zeros of the filter and the coefficients of the subfilters. In the minimum-phase filter design program, we input the coefficients of the even-order, zero-phase FIR filter and get three outputs: the zeros of the minimum-phase filter, the impulse response coefficients of the minimum-phase filter and the subfilter coefficients of the minimum-phase filter. Both programs have been tested on many practical FIR filters and have been found to work well for input filter orders ranging up to 500. In some cases, they work for orders up to and above 1000 depending on the type of the FIR filter and the specifications.

The organization of the thesis is as follows: In Chapter 2, background methods and relationships in filter decomposition and Lang's root finder are described. Specific relationships and equations are developed. In Chapter 3, the filter decomposition and minimum-phase filter design methods are addressed. Specific study results associated

with FIR digital filters of low-pass, high-pass, band-pass, and band-stop classes are presented in Chapter 4. Finally, Chapter 5 summarizes the results.

CHAPTER 2

BACKGROUND

In this study, the decomposition of high-order FIR digital filters and the design of minimum-phase filters will be presented. In this section, the appropriate relationships of all parameters observed in computable cases will be described. Parameters studied include individual subfilter frequency response, the implementation of minimum-phase filter, Newton's method and Muller's method used in root finding, and appropriate fundamental background is presented.

2.1 Decomposition of FIR Filters

Figure 2.1 illustrates the realization of an N^{th}-order filter characterized by overall impulse response coefficients $h(n)$, via the cascade connection of subfilters, where M is the number of subfilters, $h_m(n)$ are distinct impulse response coefficients of the m^{th} subfilter $(m = 1,2,....M)$, $x(n)$ is an input sequence and $y(n)$ is the resultant output sequence.

In this study, arbitrary FIR digital filters are considered with impulse response coefficients $h(n)$. Usually, there are four kinds of filters in all arbitrary FIR digital filters: Even-order, symmetric FIR filters, odd-order, symmetric FIR filters, even-order, anti-

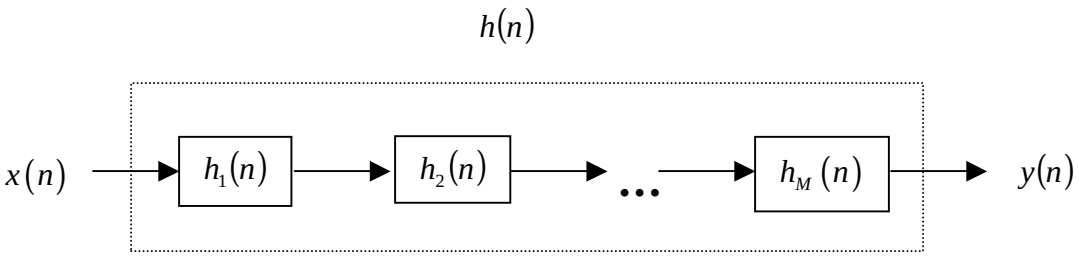

Figure 2.1. Realization of a Filter by Cascade Connection of Subfilters

symmetric FIR filters and odd-order, anti-symmetric FIR filters.

When the filter is an even-order, symmetric FIR digital filter with impulse response coefficients $h(n)$, where the linear phase is constrained through the impulse response symmetry that $h(n) = h(N-n)$, the z-domain transfer function is given by

$$H(z) = \sum_{n=0}^{N} h(n) z^{-n} \tag{2.1}$$

where N denotes the order of the filter and it is also the number of roots of the polynomial. This impulse response symmetry results in symmetry of zeroes in the transfer function polynomial. This transfer function can also be written as

$$H(z) = G \prod_{m=1}^{M} H_m(z) \tag{2.2}$$

where G is an overall gain factor. M is the number of subfilters with real-valued impulse response coefficients.

Zeros on the real axis occur in reciprocal pairs (i.e., real roots r_m and r_m^{-1}, where $|r_m|$ is the distance of the root from the origin of the z-plane). The symmetry of the

impulse response indicates that for any real root r_m, there is always a real root r_m^{-1}, so the subfilter transfer function for zeros on the real axis is given by

$$H_m(z) = 1 - \left(r_m + r_m^{-1}\right) z^{-1} + z^{-2} \tag{2.3}$$

and is of order two.

Zeros on the unit circle occur in complex conjugate pairs (i.e., complex roots $e^{j\omega_m}$ and $e^{-j\omega_m}$, where ω_m is a frequency representing the angle of the root from the positive real z-axis). Thus the subfilter transfer function for zeros on the unit circle is given by the second order relation

$$H_m(z) = 1 - 2\cos\omega_m z^{-1} + z^{-2} \tag{2.4}$$

All other zeros are inside the unit circle or outside the unit circle, and they occur in reciprocal and complex conjugate pairs (i.e., $r_m e^{j\omega_m}$, $r_m e^{-j\omega_m}$, $r_m^{-1} e^{j\omega_m}$ and $r_m^{-1} e^{-j\omega_m}$) and result in the fourth order transfer function

$$H_m(z) = 1 - 2\left(r_m + r_m^{-1}\right)\cos\omega_m z^{-1} + \left(r_m^{2} + 4\cos^2\omega_m + r_m^{-2}\right) z^{-2}$$

$$-2\left(r_m + r_m^{-1}\right)\cos\omega_m z^{-3} + z^{-4} \tag{2.5}$$

If the order of the filter is odd or the filter is anti-symmetric, at least one subfilter is first order. In this case, the zeros will be located at (1, 0) or (-1, 0) in z-plane. The general transfer function for a first order FIR digital filter is

$$H_m(z) = 1 - z_m z^{-1}, \qquad z_m = \pm 1 \tag{2.6}$$

Thus, in order to decompose the arbitrary FIR digital filter to cascade connection of subfilters, our task is to separate all the zeros to four categories: zeros located at (1, 0) and (-1,0), zeros located on the unit circle, on the real axis and inside the unit circle.

After finding all the zeros for the subfilter and the subfilter impulse response coefficients, the overall gain factor can be computed by

$$G = \frac{\int_0^{\pi} \left[\sum_{n=0}^{N-1} h(n) e^{-j\omega n} \right] \left[\prod_{m=1}^{M} H_m \left(e^{j\omega} \right) \right] d\omega}{\int_0^{\pi} \left[\prod_{m=1}^{M} H_m \left(e^{j\omega} \right) \right]^2 d\omega} \tag{2.7}$$

where G is an overall gain factor, M is the number of subfilters with real-valued impulse response coefficients.

2.2 Minimum-Phase Filter Fundamental Theory

In the minimum-phase filter design, we follow Herrmann and Schuessler's method [2]. Zero-phase, even-order, equiripple FIR filters are considered with impulse coefficients denoted $h_e[n]$, where the zero-phase is constrained through the impulse response symmetry that $h_e[n] = -h_e[n]$. Such filters can be designed through a variety of methods, such as the window method. The most common method is probably the Remez exchange algorithm utilized in the Parks-McClellan program [1]. We use the zero-phase filter because the customary case of linear-phase, causal filters can be achieved by introducing an integer delay to the impulse response coefficients of the zero-phase filter.

2.2.1 Zeros and Coefficients of the Minimum-Phase Filter

Under the constraints of even-order, zero-phase and equiripple, the frequency-domain transfer function of the FIR filter is given by

$$A_e\left(e^{j\omega}\right) = \sum_{n=-N/2}^{N/2} h_e[n]\ e^{-j\omega n}$$
$$= h_e[0] + 2\sum_{n=1}^{N/2} h_e[n]\ \cos(\omega n) \qquad (2.8)$$

where N denotes the order of the filter.

We explain the design procedure with an example of a low-pass filter. The frequency response magnitude of this low-pass filter is illustrated in Figure 2.2 with tolerated deviations δ_1 in the pass-band and δ_2 in the stop-band.

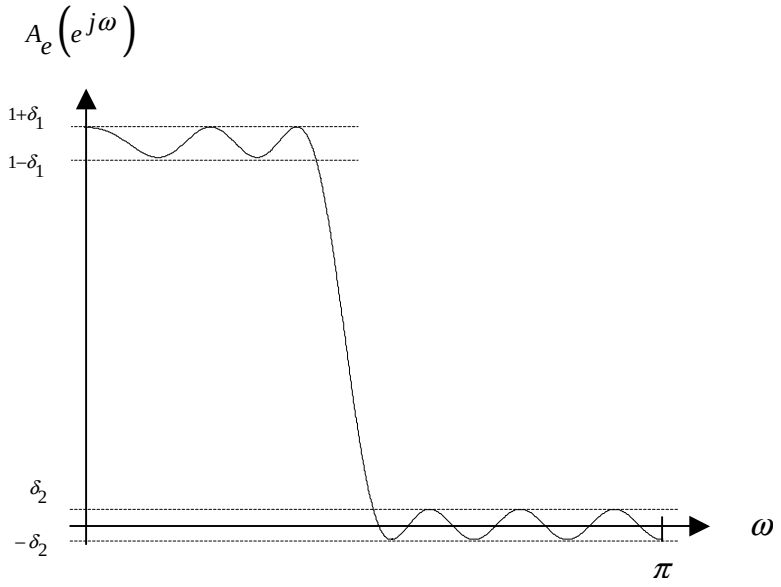

Figure 2.2. Frequency Response Magnitude of the Prototype Filter

Now we define a transfer function $H_1(e^{j\omega})$ by

$$H_1(e^{j\omega}) = \frac{A_e(e^{j\omega}) + \delta_2}{1 + \delta_2} \tag{2.9}$$

which has impulse response coefficients given by

$$h_1[n] = \begin{cases} \dfrac{h_e[0] + \delta_2}{1 + \delta_2}, & n = 0 \\ \dfrac{h_e[n]}{1 + \delta_2}, & \textit{otherwise} \end{cases} \tag{2.10}$$

Figure 2.3 shows all the zeros of $H_1(z)$ located in the z-plane. We can see there are double zeros on the unit circle. Zeros on the real axis are in reciprocal pairs and zeros inside the unit circle and the zeros outside the unit circle are in reciprocal and conjugate sets of four. We can get a transfer function $H_2(z)$ satisfying

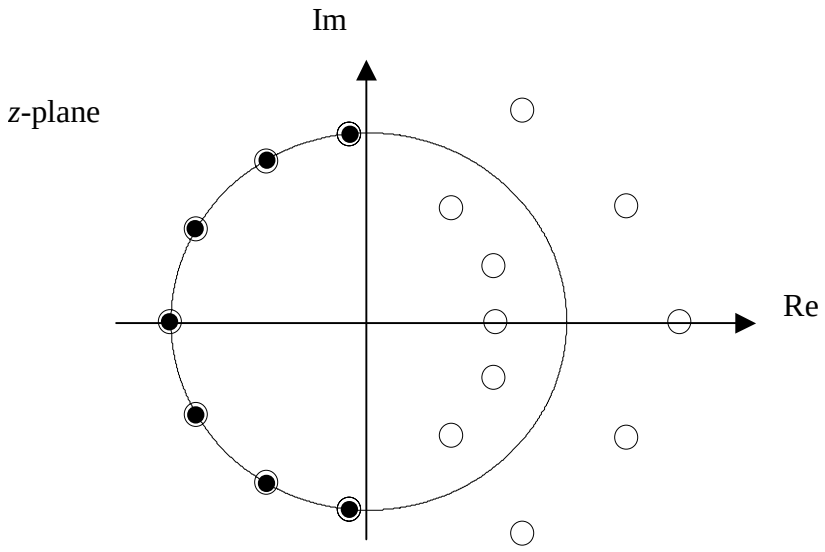

Figure 2.3. Zeros of the Modified Filter in z-Plane

$$H_1(z) = H_2(z)H_2\left(z^{-1}\right) \tag{2.11}$$

where

$$H_1\left(e^{j\omega}\right) = \left| \; H_2\left(e^{j\omega}\right) \; \right|^2 \tag{2.12}$$

and $H_2(z)$ is a $N/2$ order filter. This can also be written as

$$H_2(z) = G\prod_{n=1}^{N/2}\left(1 - z_n z^{-1}\right) \tag{2.13}$$

where *G* is an overall gain. From the definition of the minimum-phase filter [8], $H_2(z)$ will be minimum-phase if $|z_n| \le 1$ for all n. We form $H_2(z)$ by choosing the zeros of $H_1(z)$ inside the unit circle and one-half the zeros on the unit circle. The response of minimum-phase filter $H_2\left(e^{j\omega}\right)$ is then

$$\left| \; H_2\left(e^{j\omega}\right) \; \right| = \sqrt{\; H_1\left(e^{j\omega}\right)} \tag{2.14}$$

The frequency response magnitude of $H_2\left(e^{j\omega}\right)$ is shown in Figure 2.4.

The deviations of the minimum-phase filter are given by

$$\delta_1' = \frac{\delta_1}{2\left(1+\delta_2\right)} \tag{2.15}$$

$$\delta_2' = \sqrt{\frac{2\delta_2}{1+\delta_2}} \tag{2.16}$$

After all zeros are found, we should compute the gain of the new filter. According to (2.14), the magnitude of the minimum-phase is the square root of magnitude of the modified filter. While in theory it can be determined from the frequency response at any

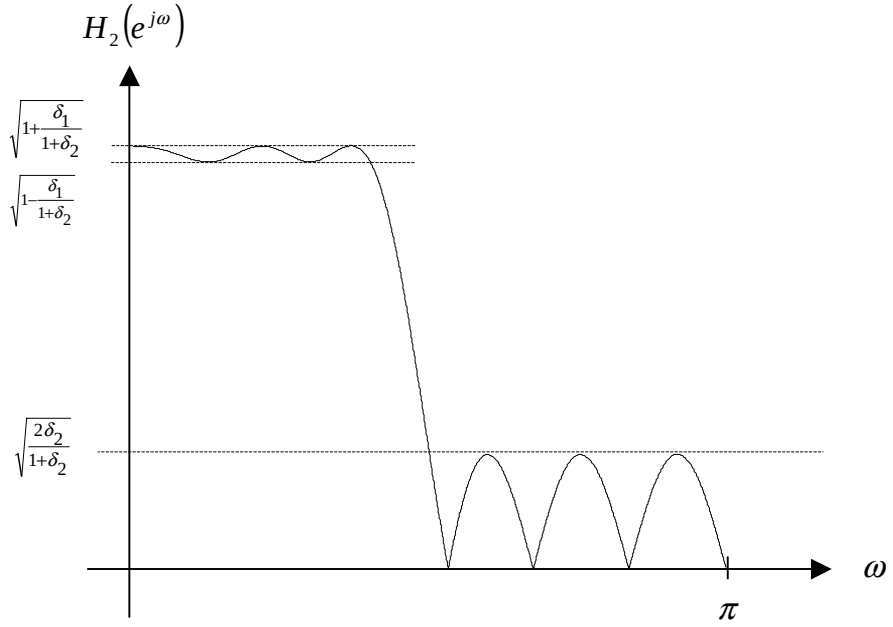

Figure 2.4. Frequency Response Magnitude of the Minimum-Phase Filter

arbitrarily selected frequency, it has been found that a formulation utilizing the full frequency response curve given by

$$G = \frac{\int_0^{\pi} \sqrt{H_1(e^{j\omega})}\, H_2(e^{j\omega})\, d\omega}{\int_0^{\pi} \left[H_2(e^{j\omega}) \right]^2 d\omega} \tag{2.17}$$

is a better method.

After we get all the zeros of the minimum-phase filter, the coefficients of the minimum-phase filter can be calculated.

2.2.2 Decomposition of the Minimum-Phase Filter

Because all the zeros of the minimum-phase filter are located inside and on the unit circle, the minimum-phase filter is decomposed to second-order subfilters. When there are zeros on the real axis, there are also first-order subfilters.

Zeros on the unit circle and inside the unit circle occur in complex conjugate pairs. The subfilter transfer function is given by the second order relation

$$H_m(z) = 1 - 2r_m \cos\omega_m z^{-1} + r_m{}^2 z^{-2} \tag{2.18}$$

The subfilter transfer function for the zeros on the real axis is given by

$$H_m(z) = 1 - r_m z^{-1} \tag{2.19}$$

In the last step, the gain is computed.

2.3 Polynomial Root Finding

Just as we mentioned above, in order to decompose an FIR filter to first, second and fourth order subfilters, and also, in order to design a minimum-phase filter, finding polynomial roots rapidly and accurately is very important. There exists a large number of different methods for finding all polynomial roots either iteratively or simulataneously. Most of them yield accurate results only for small degrees or can treat only special polynomials. Markus Lang introduced a new method to get the roots of high order filters [10]. It basically consists of a combination of two well-known iterative methods, i.e., Muller's and Newton's method.

We consider a complex polynomial

$$P(x)=\sum_{v=0}^{n} p_v x^v = p_n \prod_{v=1}^{n}(x-x_v), \quad p_n \neq 0 \tag{2.20}$$

of degree *n*. The problem is that of finding all *n* roots x_v of $P(x)$. This is the general purpose of root finding. For root finding used in complex z-domain of FIR filters, we use the substitution $x = z^{-1}$ and it can be written by

$$H(z)=\sum_{n=0}^{N} h(n) z^{-n} = G\prod_{n=0}^{M}\left(1-z_n z^{-1}\right) \tag{2.21}$$

where z_n is the root of $H(z)$ and it is also the zero in the z-plane. $h(n)$ is the coefficient of the FIR filter.

2.3.1 Muller's Method

Muller's method was developed in 1956, which is used to find zeros of arbitrary analytic functions. Lang has chosen Muller's method for computing an initial estimate of the root because it has the following two properties: One is a good convergence to a reasonable estimate of a root. The second is the possibility to get complex roots even when initialized with real values in opposition to other methods, e.g., Newton's method.

Muller's method extends the idea of the secant method which works with a linear polynomial to quadratical polynomial. Given three previous estimates $x^{(k-2)}$, $x^{(k-1)}$, and $x^{(k)}$ for an unknown root we compute a new value by determining one of the roots of a

parabola $Q(x)$ which interpolates $P(x)$ in these three "old" points. This is illustrated by Figure 2.5. The corresponding iteration formulae are [11].

$$h_k = x^{(k)} - x^{(k-1)} \tag{2.22}$$

$$r_k = h_k / h_{k-1} \tag{2.23}$$

$$A_k = r_k P\left(x^{(k)}\right) - r_k\left(1+r_k\right) P\left(x^{(k-1)}\right) + r_k^{\,2} P\left(x^{(k-2)}\right) \tag{2.24}$$

$$B_k = \left(2r_k+1\right) P\left(x^{(k)}\right) - \left(1+r_k\right)^2 P\left(x^{(k-1)}\right) + r_k^{\,2} P\left(x^{(k-2)}\right) \tag{2.25}$$

$$C_k = \left(1+r_k\right) P\left(x^{(k)}\right) \tag{2.26}$$

$$q_k = \frac{-2C_k}{B_k \pm \sqrt{B_k^{\,2} - 4A_k C_k}} \tag{2.27}$$

$$x^{(k+1)} = x^{(k)} + h_k q_k \tag{2.28}$$

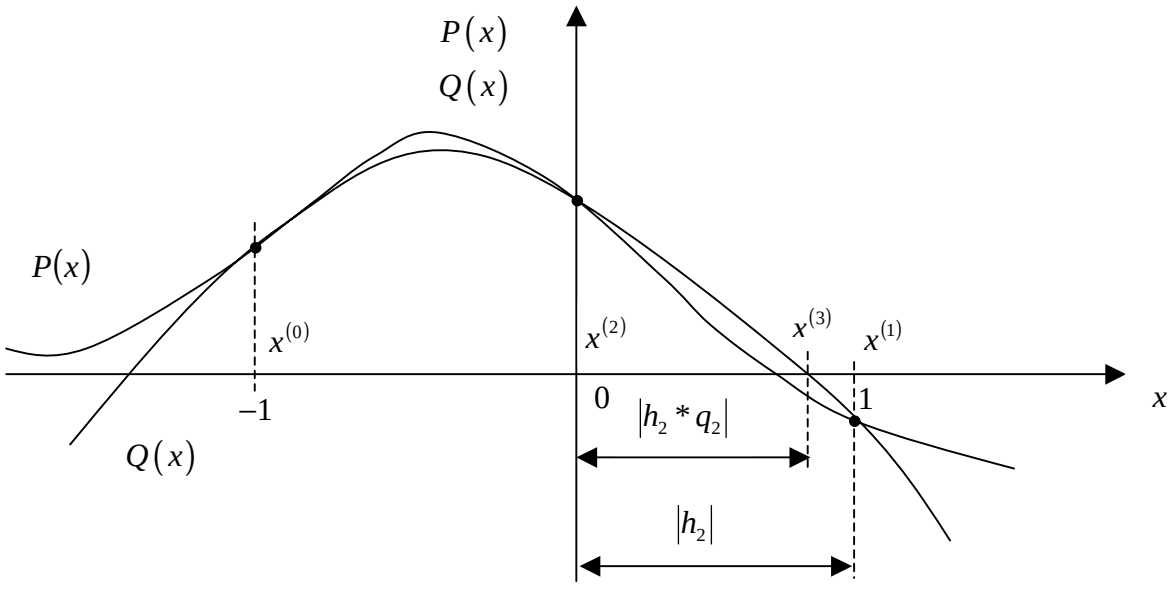

Figure 2.5. Iteration Step of Muller's Method

The new estimate $x^{(k+1)}$ is determined such that it is the one root of $Q(x)$ closer to $x^{(k)}$. An example is given in Figure 2.5 for $k=2$. In the case the denominator vanishes q_k is chosen as $|q_k|=1$ with arbitrary phase. The algorithm is initialized with $x^{(0)}=1$, $x^{(1)}=-1$ and $x^{(2)}=0$.

2.3.2 Newton's Method

Newton's method is well-known and works with the following simple iteration formula

$$x^{(k+1)} = x^{(k)} - \Delta x^{(k)}, \quad \text{where } \Delta x^{(k)} = \frac{P\left(x^{(k)}\right)}{P'\left(x^{(k)}\right)} \tag{2.29}$$

which is illustrated by Figure 2.6.

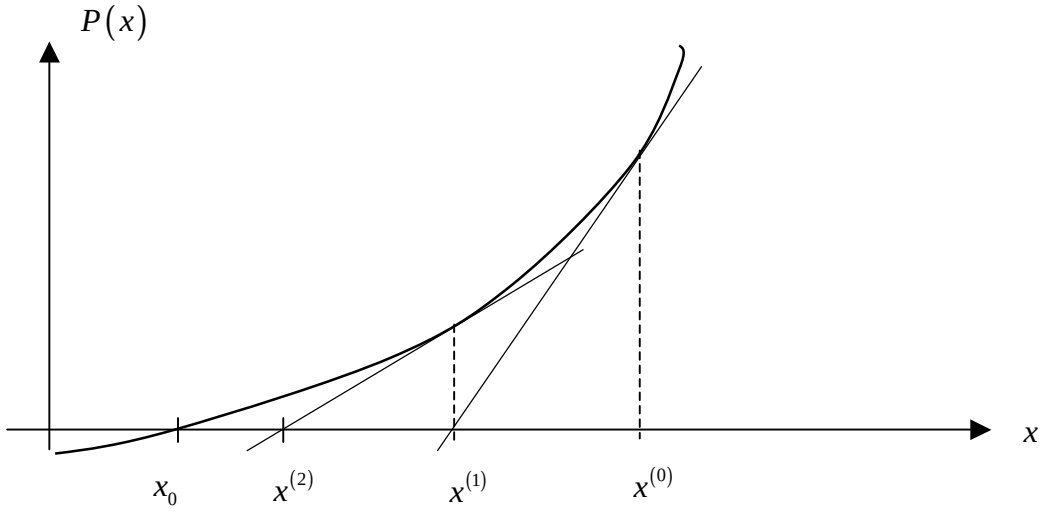

Figure 2.6. Iteration Step of Newton's Method

Newton's method works well when we are working in a small neighborhood of a point on a curve but it can break down when we are looking at a bigger portion of the curve. For this reason we will see that Newton's method can break down unless we begin with a good estimate for a solution of our equation, so we use the result of Muller's method as an estimate.

In figure 2.6, it yields the root of the tangent through the point $\left(x^{(k)}, P\left(x^{(k)}\right)\right)$ of the previous iteration step and is initialized with the result of Muller's method.

In Lang's method, he combined these two algorithms: Muller's method and Newton's method. In the first step, the coefficients are checked. If it is a first-order or second-order polynomial, the well-known explicit formulae are used. If the degree is more than 2, it executes the iterative procedure. Muller's method is numerically robust and yields an estimate for the root working with the actual deflated polynomial. In the second step, the result of Muller's method is used as the initial value of Newton's method. Then this value is refined using the quadratically converging Newton's method for the original polynomial. This procedure is repeated until the resulting polynomial is of degree two or one.

CHAPTER 3

METHODS AND PROCEDURES

We identified the necessary relationships and background of the study parameters in the last chapter. In this chapter, the methods and procedures by which they were applied are addressed. The techniques for polynomial root finding, decomposition of FIR filter and minimum-phase filter design are presented.

In high-order FIR filters, most of the zeros are very close to the unit circle. How to separate them and label them as “on the unit circle”, “inside the unit circle” and “outside the unit circle” is the problem we concentrate on. We used two different methods to separate the zeros in FIR filter decomposition and minimum-phase filter design. Before separating the zeros, we use Lang’s program to get all the zeros of the FIR filter.

3.1 Procedure Description for Polynomial Root Finding

In [10], Lang provided a C version of a program to implement polynomial root finding. Lang used a combination of Muller’s method and Newton’s method since both of them can be used to find complex roots. We modified this program to object-oriented programming style with some changes. In the following we give a pseudo code of our program:

1. Check polynomial and return if the input has errors.
2. If polynomial degree is 1 or 2 $\rightarrow$ compute roots; return.
3. Call function *monic*().
4. While degree of deflated polynomial >2.
   - Call function *muller*().
   - Call function *newton*().
   - Call function *poldef*1().
5. Compute root(s) of deflated polynomial.
6. Call function *newton*().

In the first step, the coefficients are formally checked, e.g., possible roots at zero are determined and deflated, leading zeros are cancelled, etc. If the polynomial is first-order or second-order, the well-known explicit formulae are used. For degree $n > 2$, we use our iterative procedure. At first the routine *monic*() yields a polynomial with the first coefficient equal to one. Then the routine *muller*() computes an estimate for a root of the actual, deflated polynomial $P_{defl}(x)$ which contains all roots of $P(x)$ except the roots found up to the actual iteration step (k). In Lang's program, the accuracy of the coefficients was only 10^{-8}; that is, if the value of the coefficient is less than 10^{-8}, it was treated as zero. But in practical applications, some users want smaller coefficients, so the accuracy of the coefficient was set to 10^{-16}. Lang's program has some problems in the memory allocation of deflated polynomial $P_{defl}(x)$ and $P(x)$. The pointer to $P_{defl}(x)$ is

changed in the program, but sometimes we still use the values of original $P_{defl}(x)$. So we use another pointer *psave* to point to $P_{defl}(x)$.

In the second step, the estimate resulting from *muller*() is used as the initial value of Newton's method. It is simple and known to have at least quadratical convergence near the solution (for single roots). We used the original polynomial to avoid errors introduced by the deflation process. Once Newton's method has converged the resulting root is deflated (and possibly its complex conjugate for real valued polynomials). This deflation procedure is repeated until the resulting polynomial is of degree two or one. An estimate for its roots can be obtained by using the well-known explicit formula.

3.1.1 Muller's method

For the practical convergence of this method we have to include additional measures which are summarized in the following pseudo code of our program implementing Muller's method:

1. Call *initialize*().
2. Repeat twice.

   (a) While iteration counter < ITERMAX *and* noise counter < NOISEMAX *and* root not found.

   - Call *root _ of _ parabola*().
   - Call *iteration _ equation*().

- Call *compute_function*()
- Call *check_x_value*()

(b) Call *root_check*()

(c) If root good enough, return.

After initializing, the main part is repeated twice with different starting values if the first result is not good enough. The main loop stops whenever one of the following criteria is fulfilled: First we give a maximum number of iterations considering the case of very slow convergence. Second we stop when the iteration is dominated by noise. This means that we get only minor improvements in the range of the computer accuracy during a fixed number of successive iterations. Of course the program stops when

$$\left| \frac{x^{(k+1)} - x^{(k)}}{x^{(k+1)}} \right| < \varepsilon_m \tag{3.1}$$

holds (*root_check*()), where ε_m is some small number depending on the computer accuracy. In our program, we used ε_m = 2.2204460492503131E-14 .

The first step of the main loop is computing the roots of the parabola equations (2.22) – (2.27). This is followed by the iteration equation (2.28). We have observed that values q_k computed according to equation (2.27) may yield too large changes to $x^{(k)}$ which possibly leads to another root and causes slow convergence. This can be circumvented (and is actually implemented in our program) by allowing a fixed maximum relative increase of $|q_k|$ from one iteration step to the next.

Before the new function value is evaluated we estimate $\left|P\left(x^{(k+1)}\right)\right|$ to avoid overflow. This is done by checking $n \cdot \log_{10}\left|x^{(k+1)}\right|$. If an estimate indicates a value greater than the maximum possible computer number we choose

$$x^{(k+1)} = x^{(k)} + h_k q_k / 2 \tag{3.2}$$

instead of equation (2.28) and repeat this until no overflow occurs. This means we go back closer and closer to the old value $x^{(k)}$.

In the last step, the actual value $\left|P\left(x^{(k+1)}\right)\right|$ is compared to the best value until the current iteration step and it substitutes the latter if it is smaller.

In function $root\_check()$, in order to calculate $f2 = P(x_0)$ and $df = P'(x_0)$, Lang used the deflated polynomial $P_{defl}(x)$, but the pointer to $P_{defl}(x)$ has been changed. Since what we want is the original deflated polynomial, *psave* was used instead.

The stopping criteria $\varepsilon, ITERMAX, NOISEMAX$ were determined on an experimental basis to get a program which is reliable and fast. We used $\varepsilon_m = 2.2204460492503131\text{E-}14$, $ITERMAX = 150$, $NOISEMAX = 5$.

3.1.2 Newton's Method

Similar to Muller's method, we also use additional measures to improve the performance of the program. This can be seen by the following pseudo code of our implementation:

1. While iteration counter $<$ *ITERMAX* .

   - Call *fdvalue*() .
   - If $\left|P\left(x^{(k+1)}\right)\right| < \left|P\left(x_{\min}\right)\right|$, then $x_{\min} = x^{k+1}$
   - If $\left|P'\left(x^{(k+1)}\right)\right| \neq 0$ and $\left|P\left(x^{(k+1)}\right)\right| \Big/ \left|P'\left(x^{(k+1)}\right)\right| < \left|\Delta x^{(k-1)}\right|$

     then $\Delta x^{(k)} = P\left(x^{(k+1)}\right) \Big/ P'\left(x^{(k+1)}\right)$;

     else $\Delta x^{(k)} = \Delta x^{(k-1)}$.
   - If $\Delta x^{(k)} \Big/ x^{(k)} < \varepsilon$ or noise *counter* $>$ *NOISEMAX* .
     - If in the case of a real valued polynomial $\mathrm{Im}\left(x\right) < \delta \rightarrow \mathrm{Im}\left(x_{\min}\right) = 0;$ return.
   - $x^{(k+1)} = x^{(k)} + \Delta x^{(k)}$.

2. If in the case of real valued polynomial $\mathrm{Im}\left(x\right) < \delta,$ then $\mathrm{Im}\left(x_{\min}\right) = 0$.

As in Muller's method we give a maximum number of iterations. The first step of the main loop is computing the function value and its derivative at the actual point $x^{(k+1)}$. This substitutes the minimum value up to the actual iteration step $x_{\min}$ if it yields a smaller function value. We do not permit too large changes by using the new improvement $\Delta x^{(k)}$ according to equation (2.29) only if it is smaller than the old one from the previous iteration step. Otherwise the latter one is used to avoid the algorithm switching to another root.

The algorithm is stopped when it is dominated by noise (see Muller's method) or when

$$e = \left| \frac{\Delta x^{(k)}}{x_{\min}} \right| < \varepsilon_n \tag{3.3}$$

where $x_{\min}$ is that $x^{(k)}$ leading to the minimum $\left|P(x)\right|$ up to the actual iteration step and ε_n again depends on the computer accuracy; we used $\varepsilon_n = 2.2204460492503131\text{E-}16$. In our application, all the values of the polynomial coefficients are real. In this case, for every complex root its complex conjugate is also a root which can be deflated together. Consequently we have to decide whether a root with a very small imaginary part is to be seen as a real or a complex root. We assume we have a real root if the imaginary part is less than δ which we have chosen as the square root of the computer accuracy. δ was set to 1.490116119384766E-8 in our program. To avoid this decision which may lead to errors, one could simply multiply the real valued polynomial by the imaginary unit. In this case the program would interpret the polynomial as a complex valued polynomial and it consequently deflates only one root at each iteration step.

3.2 Procedure for FIR Filter Decomposition

In the decomposition of FIR filter, L. M. Smith showed a method to search for roots in the complex z-plane [7]. Since the roots of the z-domain filter polynomial fall into the four categories discussed in the last chapter, the search algorithm is naturally divided into four corresponding parts. We used Lang's root finding program to find all

the roots of the filters and store them in a file. We assume the real part of the root is *Root.Real* and the imaginary part of the root is *Root.Imag* .

1. Search over the two points $(1,0)$ and $(-1,0)$ in the complex z -plane. In Lang's root finding program, if $|Root.Imag| < \delta$, *Root.Imag* is considered to be zero. We search all the roots, if *Root.Imag* equals to zero and $1-\Delta r < |Root.Real| < 1+\Delta r$, this root will be considered to be on point $(1,0)$ or $(-1,0)$. If $Root.Real > 0$, this root is located at $(1,0)$, if $Root.Real < 0$, this root is located at $(-1,0)$. The criteria Δr was determined on an experimental basis and the computer accuracy. We set Δr =1.0000E-8 in our program. If the FIR filter is even-order and symmetric, the search over the two point $(1,0)$ and $(-1,0)$ will have no results.
2. Search over the real axis. Among all the roots, if *Root.Imag* equals to zero and $|Root.Real| < 1-\Delta r$, this root is considered to be on the real axis and inside the unit circle. The zeros on the real axis occur in reciprocal pairs, so we can get the root on the real axis and outside the unit circle at $\left(\frac{1}{Root.Real}, 0\right)$.
3. Search over the unit circle. Assume the radial variable is r_m , if $r_m < 1+\Delta r$ and $r_m > 1-\Delta r$ and $Root.Imag > 0$, the root is considered to be on the unit circle and above the real axis. Zeros on the unit circle occur in complex conjugate pairs, so there also exists a root at $(Root.Real, -Root.Imag)$.

4. Search over the interior of the unit circle. If $r_m < 1 - \Delta r$ and $Root.Imag > 0$, this root is inside the unit circle and located above the real axis. Because all zeros inside the unit circle and outside the unit circle occur in reciprocal and complex conjugate pairs, we can find the other root located inside the unit circle at $(Root.Real, -Root.Imag)$.

Once all roots are found, the subfilters response coefficients can be determined through the equations (2.3) - (2.6). In the last step, the gain is calculated by (2.7).

3.3 Procedure for Minimum-Phase Filter Design

In minimum-phase filter design, as we discussed in last chapter, we keep half of the roots on the unit circle and the roots inside the unit circle of the modified filter. So the zeros of the z-domain only fall into the two categories: on the unit circle and inside the unit circle. The search algorithm is naturally divided into two corresponding parts. We explain the design procedure with an example of a low-pass FIR filter. After we modified the prototype filter using (2.9), the frequency response magnitude of the modified filter is shown in Figure 3.1, where δ_1 and δ_2 are the pass-band and stop-band tolerances of the prototype filter.

The frequency response is calculated from 0 to π. In calculating the curve of the frequency response magnitude, it is necessary to set a number of grid points N_{grid}. Experience has shown that $N_{grid} = 2048$ is usually sufficient to ensure to get a satisfactory curve when the FIR filter order is less than 500. We put all the points of the

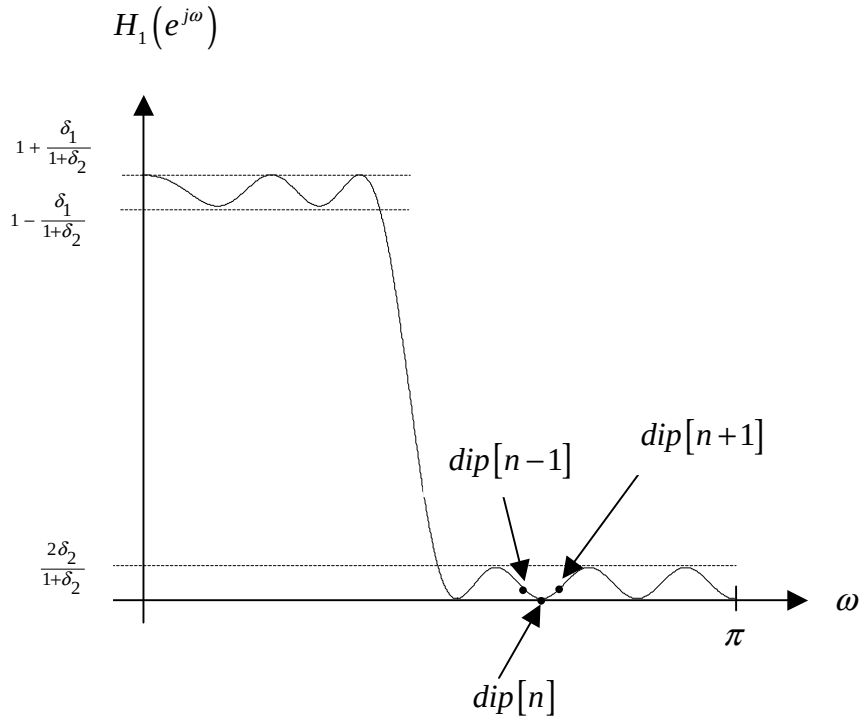

Figure 3.1. Frequency Response Magnitude of the Modified Filter

frequency response magnitude figure in an array $dip[n]$ and the related frequency in an array $\omega[n]$. Assume $dip[n-1]$ is the point on the left and next to $dip[n]$, $dip[n+1]$ is the point on the right and next to $dip[n]$. If $\left| dip[n] \right| < \delta_m$ and $dip[n] < dip[n-1]$ and $dip[n] < dip[n+1]$, we consider $dip[n]$ to be a double zero on the curve, where δ_m depends on our experience accuracy. We set $\delta_m = \left| A_{\max}(e^{j\omega}) \right| \times 10^{-5}$, where $\left| A_{\max}(e^{j\omega}) \right|$ was the maximum magnitude of the prototype filter. The related frequency is $\omega[n]$. In the complex z-plane, there are double zeros on the unit circle and the angles of these

double zeros are close to $\omega[n]$, as illustrated in Figure 3.2. There are two special cases here and they are the first point and the last point of the curve. Because the frequency response is symmetric in $(0,\pi)$ and $(0,-\pi)$, for the first point $dip[0]$ and the last point $dip[N]$, if $dip[0] < dip[1]$ and $dip[0] < \delta_m$, $dip[0]$ is a double zero on the curve. Also, if $dip[N] < dip[N-1]$ and $dip[N] < \delta_m$, $dip[N]$ is a double zero on the curve too. The angles for these two points are 0 and π respectively. After we got all the double zeros of the frequency response and the related angles, we get the zeros on the unit circle for the minimum-phase filter as $\left(\cos(\omega[n]),\ \pm\sin(\omega[n]) \right)$.

The second step is to find all the zeros of the modified filter on the unit circle. We use Lang's root finding program to find all the zeros of the modified filter. There are double zeros whose angles are close to $\omega[n]$. We use following steps to find all of these double zeros:

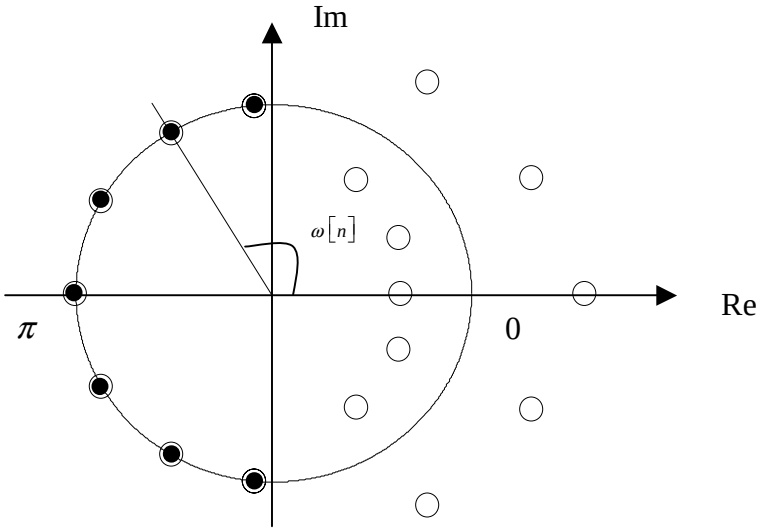

Figure 3.2. Double Zeros on the Unit Circle and the Angles

1. We look for one root among all the roots, which angle is closest to $\omega[n]$ and the radius is between $1-\Delta r$ and $1+\Delta r$. This root is considered to be on the unit circle. Δr is the same as in the FIR filter decomposition program.
2. Eliminate the root we got in step 1 and search again among the remaining roots, and find another root whose angle is closet to $\omega[n]$ and radius is between $1-\Delta r$ and $1+\Delta r$, this root is also considered to be on the unit circle.
3. Iterate step 1 and step 2, we can get all the double zeros on the unit circle of the modified filter.

The radius r_m of the zeros inside the unit circle is always less than one. The radius r_m of the zeros on the unit circle is satisfied: $1-\Delta r < r_m < 1+\Delta r$. If we choose the radii of all the zeros by $r_m <= 1$, we can get all the zeros inside the unit circle and some zeros on the unit circle. We compare these zeros with the double zeros on the unit circle we got in the last step and eliminate those zeros on the unit circle giving the zeros inside the unit circle. These zeros are also chosen as the zeros inside the unit circle of the minimum-phase filter.

Now, having determined all the zeros inside the unit circle and on the unit circle, we have the zeros of the minimum-phase filter. The next necessary step is computing the polynomial coefficients from the roots. We can use the usual recurrence method to get the coefficients from the roots with the following pseudo code, where *root*[*i*] is the roots

we got, N_roots is the number of the roots and $coeff[j]$ is the coefficients we want to get:

$$Initialize \quad coeff[j]$$

$$For \;\; i = 0 \;\; to \;\; N\_roots$$

$$For \;\; j = i + 1 \;\; to \;\; 0$$

$$coeff[j] \; -= root[i] * coeff[j-1]$$

In [13], Nachtigal presented a new method to get the coefficients. They reindexed the order of the arbitrary roots and then used the usual method above. Using this method, the results are more accurate. We wrote a program function $RootCoeff(\;)$ following Nachtigal's method. We can get the coefficients of the minimum-phase filter by using this function. The gain is computed in the last step.

After we have all the zeros, the minimum-phase filter can also be implemented via first-order and second-order subfilters by (2.18) and (2.19).

CHAPTER 4

RESULTS

In this chapter, we describe the results of FIR filter decomposition and minimum-phase filter design methods. Two computer programs were written to implement these methods. Initially, FIR filters of different order representative of each basic filter class, such as lowpass, highpass, bandpass, bandstop, were designed using the window method or Parks-McClellan program [1] [12]. Even-order, zero-phase FIR filters were designed using window method or Parks-McClellan program for minimum-phase filter design. Both of our programs accept input data files of impulse response coefficients in the format supplied by the Parks-McClellan program [12] or window method.

There are two outputs from the FIR filter decomposition program:

1. The roots of the filter with the gain factor.
2. The subfilter impulse response coefficients with the gain factor.

There are three outputs from the minimum-phase filter design program:

1. The roots of the minimum-phase filter with the gain factor.
2. The subfilter impulse response coefficients of the minimum-phase filter with the gain factor.
3. The impulse response coefficients of the minimum-phase filter.

These two programs have been tested on many practical filters and have been found to be effective in determining the roots of the filters, forming the cascade

realization via subfilters and designing the minimum-phase filters. Filters of order up to 500 have been decomposed via this method. Also, prototype filters of order up to 500 have been used to design the minimum-phase filter via the minimum-phase filters design method, so the order of the minimum-phase filter is up to 250. For filters order above 500, some numerical instabilities have been observed, so not all the roots of the transfer function could be found. In specific cases, these two programs can decompose the FIR filter and design the minimum-phase filter with filter orders in excess of 1000, but it depends on the filter type and specifications.

We use two examples with different kinds of filters, low-pass filter and band-pass filter, to describe our results.

4.1 Decomposition of FIR Filter

First we consider the decomposition of a 495^{th}-order low-pass FIR filter designed using the Parks-McClellan program. This is an odd-order, linear-phase FIR filter. The input specifications used to obtain this filter design are listed in Table 4.1.

Table 4.1. Specifications for 495^{th}-Order Low-Pass Filter

Band	Lower Band Edge	Upper Band Edge	Gain	Weight
1	0.00	0.25	1.0	1.0
2	0.26	0.50	0.0	1.0

The frequency response magnitude of this filter is illustrated in Figure 4.1. The roots of the transfer function are given in Table 4.2. The impulse response coefficient file was used as input to the decomposing filter program. The subfilter coefficients were calculated and are listed in Table 4.3.

Two roots were found on the real axis, 121 on the upper half unit circle, 62 inside the upper half unit circle, and because it is an odd-order FIR filter, there is also one root located at (-1, 0), so that there are total of 186 subfilters required to form the cascade realization. To check the results, the frequency response curve of the cascade structure was calculated using these parameters and compared with that of the direct realization. The results were identical to that computed via the direct method.

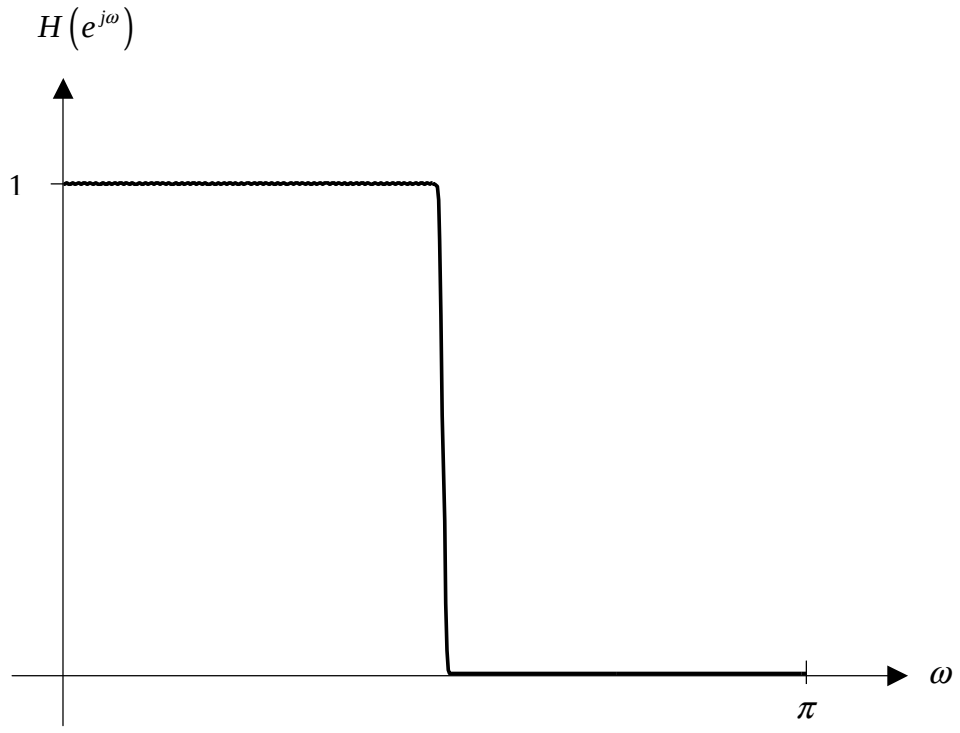

Figure 4.1. Frequency Response Magnitude of 495th-Order Low-Pass Filter

Table 4.2. Roots of the 495th-Order Low-Pass Filter Transfer Function

Real Part	Imaginary part
2.80271450947597600e+000	0.00000000000000000e+000
-2.35910720308782410e-001	-9.71774733178111030e-001
-2.35910720308782410e-001	9.71774733178111030e-001
9.67514211020097830e-001	-3.90592297391165410e-001
9.67514211020097830e-001	3.90592297391165410e-001
-9.32843940750387350e-001	-3.60280699185077380e-001
-9.32843940750387350e-001	3.60280699185077380e-001
3.28664803127067420e-001	-9.89880375137925150e-001
3.28664803127067420e-001	9.89880375137925150e-001
6.10952612114171620e-001	-8.45606620294252930e-001
6.10952612114171620e-001	8.45606620294252930e-001
-8.33936560435654230e-001	5.51860320342706380e-001
-8.33936560435654230e-001	-5.51860320342706380e-001
-4.48910219005640650e-001	8.93576865900358340e-001
-4.48910219005640650e-001	-8.93576865900358340e-001
7.23470507970056940e-002	-1.03833852913114090e+000
7.23470507970056940e-002	1.03833852913114090e+000
1.04206101590181380e+000	-5.33783092178389820e-002
1.04206101590181380e+000	5.33783092178389820e-002
9.09831970152925160e-001	-5.10674235686716550e-001
9.09831970152925160e-001	5.10674235686716550e-001
-9.99273242784267320e-001	-3.81180568157872410e-002
-9.99273242784267320e-001	3.81180568157872410e-002
1.49503757151741880e-001	-1.03157830093039120e+000
1.49503757151741880e-001	1.03157830093039120e+000
-8.97150379583852910e-001	-4.41725249915087050e-001
-8.97150379583852910e-001	4.41725249915087050e-001
-5.68859801032260530e-001	8.22434512146429150e-001
-5.68859801032260530e-001	-8.22434512146429150e-001
1.00944321316501280e+000	-2.64051473967562180e-001
1.00944321316501280e+000	2.64051473967562180e-001
8.37376195694698900e-001	-6.22344894544405890e-001
8.37376195694698900e-001	6.22344894544405890e-001
-5.04642140511208040e-001	8.63328622264008080e-001
-5.04642140511208040e-001	-8.63328622264008080e-001
-9.81882812670750170e-001	1.89489161119520520e-001
-9.81882812670750170e-001	-1.89489161119520520e-001
4.27357834391473400e-001	9.51557936799828590e-001
4.27357834391473400e-001	-9.51557936799828590e-001

Table 4.2. Continued.

1.043427699580757900e+000	0.000000000000000000e+000
-3.324802647021339900e-001	-9.431102128508648300e-001
-3.324802647021339900e-001	9.431102128508648300e-001
3.034624778728708400e-001	-9.978544002697934400e-001
3.034624778728708400e-001	9.978544002697934400e-001
-9.919333228058887500e-001	1.267607317242361900e-001
-9.919333228058887500e-001	-1.267607317242361900e-001
7.695861547264732900e-001	-7.044065689397324100e-001
7.695861547264732900e-001	7.044065689397324100e-001
5.892159360695449200e-001	8.608819829511612500e-001
5.892159360695449200e-001	-8.608819829511612500e-001
-7.413536284971786200e-002	-9.972481877522468800e-001
-7.413536284971786200e-002	9.972481877522468800e-001
-8.196400202684465100e-001	5.728789027136035300e-001
-8.196400202684465100e-001	-5.728789027136035300e-001
9.862221536185510300e-001	3.406364680352327300e-001
9.862221536185510300e-001	-3.406364680352327300e-001
-3.797496970975857700e-001	-9.250892754509111300e-001
-3.797496970975857700e-001	9.250892754509111300e-001
-3.578181134448460800e-003	-1.030773408976602500e+000
-3.578181134448460800e-003	1.030773408976602500e+000
6.938420867996779600e-001	-7.790824951586246000e-001
6.938420867996779600e-001	7.790824951586246000e-001
9.462831006887951800e-001	-4.395126041113606200e-001
9.462831006887951800e-001	4.395126041113606200e-001
-5.792462686905903700e-001	8.151525993383265100e-001
-5.792462686905903700e-001	-8.151525993383265100e-001
-8.543610313586345000e-001	-5.196799285096604900e-001
-8.543610313586345000e-001	5.196799285096604900e-001
1.031148543591865500e+000	-1.595645302435371700e-001
1.031148543591865500e+000	1.595645302435371700e-001
-9.499502670993839000e-001	-3.124011684322127200e-001
-9.499502670993839000e-001	3.124011684322127200e-001
-3.205316947985330500e-001	-9.472377909635887300e-001
-3.205316947985330500e-001	9.472377909635887300e-001
-7.572768031938080800e-001	6.530940539804088700e-001
-7.572768031938080800e-001	-6.530940539804088700e-001
1.753489427976303600e-001	1.027700822656471000e+000
1.753489427976303600e-001	-1.027700822656471000e+000
6.322977043652519000e-001	-8.297785928128727000e-001
6.322977043652519000e-001	8.297785928128727000e-001
1.034895282473055300e+000	1.331335074210670000e-001
1.034895282473055300e+000	-1.331335074210670000e-001

Table 4.2. Continued.

-8.796593195985333800e-001	-4.756043328686628800e-001
-8.796593195985333800e-001	4.756043328686628800e-001
4.751794506163185500e-001	-9.286396055390194700e-001
4.751794506163185500e-001	9.286396055390194700e-001
2.195951088259737500e-002	-1.036583238550925300e+000
2.195951088259737500e-002	1.036583238550925300e+000
-9.709865013660284900e-001	-2.391343015231406400e-001
-9.709865013660284900e-001	2.391343015231406400e-001
8.045228430292691300e-001	6.642482697331035800e-001
8.045228430292691300e-001	-6.642482697331035800e-001
1.015870232612843900e+000	-2.381499717730466100e-001
1.015870232612843900e+000	2.381499717730466100e-001
-1.749728501634345300e-001	-9.845732586789495100e-001
-1.749728501634345300e-001	9.845732586789495100e-001
-5.895497656243724500e-001	8.077320557290298400e-001
-5.895497656243724500e-001	-8.077320557290298400e-001
4.986493331165255400e-001	9.162671671811732900e-001
4.986493331165255400e-001	-9.162671671811732900e-001
-9.709433818365434000e-001	-7.617664184052250200e-002
-9.709433818365434000e-001	7.617664184052250200e-002
2.011646578191498300e-001	-1.023102248244367200e+000
2.011646578191498300e-001	1.023102248244367200e+000
7.873093221374150000e-001	-6.845520219282215100e-001
7.873093221374150000e-001	6.845520219282215100e-001
-9.233866699161071800e-001	-3.838711474196054300e-001
-9.233866699161071800e-001	3.838711474196054300e-001
8.964868472505895800e-001	-5.337417127313799400e-001
8.964868472505895800e-001	5.337417127313799400e-001
-9.645935390668829400e-001	-2.637409797328190100e-001
-9.645935390668829400e-001	2.637409797328190100e-001
-2.602274433270441200e-001	-9.655473461925444500e-001
-2.602274433270441200e-001	9.655473461925444500e-001
8.680587105957501300e-001	-5.788055819583876000e-001
8.680587105957501300e-001	5.788055819583876000e-001
-5.997627510060031800e-001	8.001778817898586000e-001
-5.997627510060031800e-001	-8.001778817898586000e-001
3.784722656640890900e-001	9.719904714976569400e-001
3.784722656640890900e-001	-9.719904714976569400e-001
2.780881445662325400e-001	-1.005171849916195700e+000
2.780881445662325400e-001	1.005171849916195700e+000
5.446263504085708800e-001	8.897363046268680400e-001
5.446263504085708800e-001	-8.897363046268680400e-001
-7.894790715380787200e-001	6.137774805280589200e-001

Table 4.2. Continued.

-7.894790715380787200e-001	-6.137774805280589200e-001
-1.388187982274594900e-001	9.903177981126477500e-001
-1.388187982274594900e-001	-9.903177981126477500e-001
9.347388061566923600e-001	-4.635444623487551300e-001
9.347388061566923600e-001	4.635444623487551300e-001
-7.816116809385997800e-001	6.237653246376653200e-001
-7.816116809385997800e-001	-6.237653246376653200e-001
1.043085985026796200e+000	-2.669812218271746000e-002
1.043085985026796200e+000	2.669812218271746000e-002
-4.714278698335722100e-001	-8.819046227025808500e-001
-4.714278698335722100e-001	8.819046227025808500e-001
7.134810772986732700e-001	-7.611492618826993800e-001
7.134810772986732700e-001	7.611492618826993800e-001
7.326597773848005900e-001	-7.427177565970740600e-001
7.326597773848005900e-001	7.427177565970740600e-001
-4.146171637375133900e-001	-9.099959381965758700e-001
-4.146171637375133900e-001	9.099959381965758700e-001
-9.863812063403348800e-001	-1.644752740644689300e-001
-9.863812063403348800e-001	1.644752740644689300e-001
9.572106793600178000e-001	-4.151899655153603000e-001
9.572106793600178000e-001	4.151899655153603000e-001
-6.298011201015855100e-001	7.767564284373776900e-001
-6.298011201015855100e-001	-7.767564284373776900e-001
-9.999999999999991100e-001	0.000000000000000000e+000
9.793039140445594600e-002	1.036996643535055100e+000
9.793039140445594600e-002	-1.036996643535055100e+000
4.701959989858625600e-002	1.038435761297679300e+000
4.701959989858625600e-002	-1.038435761297679300e+000
-8.123005000412317800e-001	5.832391427474323800e-001
-8.123005000412317800e-001	-5.832391427474323800e-001
5.218035760448139800e-001	-9.032965738702676300e-001
5.218035760448139800e-001	9.032965738702676300e-001
-9.080817855688841900e-001	4.187928732894409300e-001
-9.080817855688841900e-001	-4.187928732894409300e-001
1.026726664199254700e+000	-1.858892299721426300e-001
1.026726664199254700e+000	1.858892299721426300e-001
2.525638145723596600e-001	-1.011825111810502400e+000
2.525638145723596600e-001	1.011825111810502400e+000
-1.040022995149882000e-001	9.945770566907309300e-001
-1.040022995149882000e-001	-9.945770566907309300e-001
1.037964278034845500e+000	1.066137063013788100e-001
1.037964278034845500e+000	-1.066137063013788100e-001
4.030393858906000700e-001	-9.620885418489817400e-001

Table 4.2. Continued.

4.03039385890600700e-001	9.620885418489817400e-001
8.529942530276886100e-001	-6.007727835290750400e-001
8.529942530276886100e-001	6.007727835290750400e-001
-9.948355226260707800e-001	-1.015001621738237900e-001
-9.948355226260707800e-001	1.015001621738237900e-001
9.946139289930510100e-001	-3.153115252762900800e-001
9.946139289930510100e-001	3.153115252762900800e-001
-6.493305020220729900e-001	7.605063439207876400e-001
-6.493305020220729900e-001	-7.605063439207876400e-001
1.236765011605985000e-001	-1.034698997496122300e+000
1.236765011605985000e-001	1.034698997496122300e+000
-5.155498490934283800e-001	8.568595877387079600e-001
-5.155498490934283800e-001	-8.568595877387079600e-001
1.021632715547336600e+000	2.120901561219248900e-001
1.021632715547336600e+000	-2.120901561219248900e-001
-8.672952327410247100e-001	4.977941133286820100e-001
-8.672952327410247100e-001	-4.977941133286820100e-001
-6.778306739507523600e-001	7.352180475556021600e-001
-6.778306739507523600e-001	-7.352180475556021600e-001
-2.115297101202031900e-001	-9.773715678985437400e-001
-2.115297101202031900e-001	9.773715678985437400e-001
-9.934664684131907500e-001	-1.141243888860728500e-001
-9.934664684131907500e-001	1.141243888860728500e-001
6.532370092223359600e-001	-8.134082281765122600e-001
6.532370092223359600e-001	8.134082281765122600e-001
-9.458973329525648400e-001	-3.244660776001496200e-001
-9.458973329525648400e-001	3.244660776001496200e-001
-8.301999585825499400e-002	-9.965478815830682800e-001
-8.301999585825499400e-002	9.965478815830682800e-001
4.514101242671200700e-001	-9.404058376744934800e-001
4.514101242671200700e-001	9.404058376744934800e-001
-9.767524187930533000e-001	-2.143705025928787900e-001
-9.767524187930533000e-001	2.143705025928787900e-001
8.825594048270221100e-001	-5.564571338269919100e-001
8.825594048270221100e-001	5.564571338269919100e-001
5.671021639834777900e-001	-8.755950321842601600e-001
5.671021639834777900e-001	8.755950321842601600e-001
-3.914236482877270800e-001	9.202105887030046500e-001
-3.914236482877270800e-001	-9.202105887030046500e-001
9.771868441217912400e-001	-3.657356541763915300e-001
9.771868441217912400e-001	3.657356541763915300e-001
2.269137129362238700e-001	-1.017805378771894100e+000
2.269137129362238700e-001	1.017805378771894100e+000

Table 4.2. Continued.

9.22585625190288970e-001	-4.87270079068489010e-001
9.22585625190288970e-001	4.87270079068489010e-001
-7.48925355980061820e-001	6.62654367804315640e-001
-7.48925355980061820e-001	-6.62654367804315640e-001
1.00235612408226470e+000	2.89777616067713910e-001
1.00235612408226470e+000	-2.89777616067713910e-001
3.53674532320565170e-001	-9.81256761050079970e-001
3.53674532320565170e-001	9.81256761050079970e-001
-6.87107051140365850e-001	7.26556192096105000e-001
-6.87107051140365850e-001	-7.26556192096105000e-001
-5.37137852243354640e-001	8.43494474011180270e-001
-5.37137852243354640e-001	-8.43494474011180270e-001
-4.37548366079156930e-001	8.99194877288266410e-001
-4.37548366079156930e-001	-8.99194877288266410e-001
7.51365498585211730e-001	-7.23799425171789520e-001
7.51365498585211730e-001	7.23799425171789520e-001
-9.18435219675921010e-001	3.95571418657670040e-001
-9.18435219675921010e-001	-3.95571418657670040e-001
8.21215413991484900e-001	-6.43508607945057300e-001
8.21215413991484900e-001	6.43508607945057300e-001
-6.72314492691572780e-002	9.97737406449798690e-001
-6.72314492691572780e-002	-9.97737406449798690e-001
-1.50792388541296810e-001	-9.88565453350465310e-001
-1.50792388541296810e-001	9.88565453350465310e-001
1.04035340265685860e+000	-8.00227472724964330e-002
1.04035340265685860e+000	8.00227472724964330e-002
-7.31841598327557800e-001	6.81474779399330650e-001
-7.31841598327557800e-001	-6.81474779399330650e-001
6.73756303267837860e-001	-7.96505991496002100e-001
6.73756303267837860e-001	7.96505991496002100e-001
-9.99676731579486220e-001	2.54250336982865150e-002
-9.99676731579486220e-001	-2.54250336982865150e-002
-8.85636912911570650e-001	-4.64378357041384860e-001
-8.85636912911570650e-001	4.64378357041384860e-001
-8.47688824191639270e-001	-5.30493786335522350e-001
-8.47688824191639270e-001	5.30493786335522350e-001
-2.72359683681149710e-001	-9.62195511684143410e-001
-2.72359683681149710e-001	9.62195511684143410e-001
9.47112955392427120e-001	1.46560488063470720e-001
9.47112955392427120e-001	-1.46560488063470720e-001
3.47858385741665940e-001	-8.93368066952487830e-001
3.47858385741665940e-001	8.93368066952487830e-001
-9.31541628616455180e-002	-9.95651697101727520e-001

Table 4.2. Continued.

-9.315416286164551800e-002	9.956516971017275200e-001
8.692477281397200700e-001	-4.037325958103693500e-001
8.692477281397200700e-001	4.037325958103693500e-001
-9.575793947344142200e-001	-2.881695729602141700e-001
-9.575793947344142200e-001	2.881695729602141700e-001
9.058985400824789400e-001	-3.128930717686169100e-001
9.058985400824789400e-001	3.128930717686169100e-001
5.613754930061395400e-001	-7.769879757355578900e-001
5.613754930061395400e-001	7.769879757355578900e-001
-9.979812240517300600e-001	-6.350965627531740500e-002
-9.979812240517300600e-001	6.350965627531740500e-002
-5.263735559383366800e-001	-8.502534208157164200e-001
-5.263735559383366800e-001	8.502534208157164200e-001
-3.680175724350892600e-001	-9.298188352464080700e-001
-3.680175724350892600e-001	9.298188352464080700e-001
1.376005950609361100e-001	-9.944629328546193000e-001
1.376005950609361100e-001	9.944629328546193000e-001
7.974509470064037300e-001	-5.317256238907310800e-001
7.974509470064037300e-001	5.317256238907310800e-001
-9.611669544335305100e-001	-2.759675446587343500e-001
-9.611669544335305100e-001	2.759675446587343500e-001
-6.198870903857290900e-001	7.846910189196235600e-001
-6.198870903857290900e-001	-7.846910189196235600e-001
6.731406690835541900e-001	-6.823815678821002500e-001
6.731406690835541900e-001	6.823815678821002500e-001
-2.965233032649233800e-001	9.550256177825039400e-001
-2.965233032649233800e-001	-9.550256177825039400e-001
4.582345661465109200e-001	8.420051124974481400e-001
4.582345661465109200e-001	-8.420051124974481400e-001
2.322258171757742300e-001	9.303467079281203500e-001
2.322258171757742300e-001	-9.303467079281203500e-001
9.555582680115042300e-001	-7.350040629458107200e-002
9.555582680115042300e-001	7.350040629458107200e-002
-7.972128027651105300e-001	6.036983908438006100e-001
-7.972128027651105300e-001	-6.036983908438006100e-001
-8.048127144469357000e-001	-5.935288490583717800e-001
-8.048127144469357000e-001	5.935288490583717800e-001
-1.269901346295385400e-001	-9.919039800841459300e-001
-1.269901346295385400e-001	9.919039800841459300e-001
3.021139117506939100e-001	9.099137767500542500e-001
3.021139117506939100e-001	-9.099137767500542500e-001
-9.842132411662833900e-001	1.769867111196881900e-001
-9.842132411662833900e-001	-1.769867111196881900e-001

Table 4.2. Continued.

9.207002617070063600e-001	-2.661711945888007300e-001
9.207002617070063600e-001	2.661711945888007300e-001
-1.871402198174540200e-001	9.823332113527856200e-001
-1.871402198174540200e-001	-9.823332113527856200e-001
9.383834292438320000e-001	-1.948076691180805100e-001
9.383834292438320000e-001	1.948076691180805100e-001
-8.268513988987191700e-001	5.624204513877780800e-001
-8.268513988987191700e-001	-5.624204513877780800e-001
-9.373496285761312400e-001	-3.483901172653950400e-001
-9.373496285761312400e-001	3.483901172653950400e-001
8.887316261415970600e-001	-3.587872133194143200e-001
8.887316261415970600e-001	3.587872133194143200e-001
8.475001885051343100e-001	-4.476131782112694500e-001
8.475001885051343100e-001	4.476131782112694500e-001
-4.825725055305786200e-001	-8.758560252152969300e-001
-4.825725055305786200e-001	8.758560252152969300e-001
4.795002987973063200e-001	-8.300651757821181400e-001
4.795002987973063200e-001	8.300651757821181400e-001
7.391231777759050800e-001	-6.102515251259516000e-001
7.391231777759050800e-001	6.102515251259516000e-001
-2.844653013099789400e-001	-9.586863367914568700e-001
-2.844653013099789400e-001	9.586863367914568700e-001
-4.936351993378996800e-001	-8.696690692295706600e-001
-4.936351993378996800e-001	8.696690692295706600e-001
-9.026911161236996800e-001	-4.302891456582940100e-001
-9.026911161236996800e-001	4.302891456582940100e-001
-3.562061829556972800e-001	-9.344073818330708200e-001
-3.562061829556972800e-001	9.344073818330708200e-001
-2.237258257538163600e-001	-9.746521199334552400e-001
-2.237258257538163600e-001	9.746521199334552400e-001
8.235462735375956900e-001	-4.903150558199255200e-001
8.235462735375956900e-001	4.903150558199255200e-001
1.613270071176471400e-001	-9.455198034632967800e-001
1.613270071176471400e-001	9.455198034632967800e-001
-8.408755732390680300e-001	5.412284825560010900e-001
-8.408755732390680300e-001	-5.412284825560010900e-001
9.430563917676236800e-001	-1.707406972066033400e-001
9.430563917676236800e-001	1.707406972066033400e-001
4.148478264414299800e-001	8.642369693534848200e-001
4.148478264414299800e-001	-8.642369693534848200e-001
5.211035583218631700e-001	-8.045740536681459400e-001
5.211035583218631700e-001	8.045740536681459400e-001
-9.999193607442558000e-001	-1.269929166550406800e-002

Table 4.2. Continued.

-9.999193607442558000e-001	1.269929166550406800e-002
-9.987082508568122500e-001	5.081170800642884800e-002
-9.987082508568122500e-001	-5.081170800642884800e-002
-8.735462854500541900e-001	-4.867410884406782600e-001
-8.735462854500541900e-001	4.867410884406782600e-001
6.677895496433029800e-002	-9.584241667173423200e-001
6.677895496433029800e-002	9.584241667173423200e-001
7.544486819304931900e-001	-5.911893517905845200e-001
7.544486819304931900e-001	5.911893517905845200e-001
1.850291143544876600e-001	9.410385747626078800e-001
1.850291143544876600e-001	-9.410385747626078800e-001
6.190511546889284600e-001	-7.318342720665232900e-001
6.190511546889284600e-001	7.318342720665232900e-001
-4.030585075585885700e-001	-9.151742126418593900e-001
-4.030585075585885700e-001	9.151742126418593900e-001
-9.739447056246292000e-001	-2.267856044499951000e-001
-9.739447056246292000e-001	2.267856044499951000e-001
-6.329573067304909700e-002	-9.979948148555491900e-001
-6.329573067304909700e-002	9.979948148555491900e-001
-6.396065793388476800e-001	7.687024285550672100e-001
-6.396065793388476800e-001	-7.687024285550672100e-001
-9.281942535900993300e-001	-3.720959924566721900e-001
-9.281942535900993300e-001	3.720959924566721900e-001
6.903168322593119500e-001	-6.649905103661552000e-001
6.903168322593119500e-001	6.649905103661552000e-001
9.533660761985040000e-001	-9.792426675604053200e-002
9.533660761985040000e-001	9.792426675604053200e-002
-9.883931248392351400e-001	1.519178421731019200e-001
-9.883931248392351400e-001	-1.519178421731019200e-001
9.580660872769222200e-001	-2.452201048083319700e-002
9.580660872769222200e-001	2.452201048083319700e-002
-2.480792415707894500e-001	-9.687397431207535000e-001
-2.480792415707894500e-001	9.687397431207535000e-001
-7.655122442216995000e-001	6.434213269286728200e-001
-7.655122442216995000e-001	-6.434213269286728200e-001
-9.538407710305344400e-001	-3.003128094502128300e-001
-9.538407710305344400e-001	3.003128094502128300e-001
5.809782138758196000e-001	7.624308635578003200e-001
5.809782138758196000e-001	-7.624308635578003200e-001
3.927570296997633900e-001	-8.745155435769693700e-001
3.927570296997633900e-001	8.745155435769693700e-001
8.107618650586566600e-001	-5.111885060419143500e-001
8.107618650586566600e-001	5.111885060419143500e-001

Table 4.2. Continued.

-6.684271638612109300e-001	7.437776056137053200e-001
-6.684271638612109300e-001	-7.437776056137053200e-001
-5.583628976825718600e-001	8.295968144174151700e-001
-5.583628976825718600e-001	-8.295968144174151700e-001
2.556654030873841100e-001	-9.241230566724455400e-001
2.556654030873841100e-001	9.241230566724455400e-001
8.976078930670013900e-001	3.359513197906257600e-001
8.976078930670013900e-001	-3.359513197906257600e-001
-6.098664008361440700e-001	7.925042417117872400e-001
-6.098664008361440700e-001	-7.925042417117872400e-001
-8.914615156709901900e-001	-4.530964202877547600e-001
-8.914615156709901900e-001	4.530964202877547600e-001
1.138933062039065900e-001	-9.528510965690609700e-001
1.138933062039065900e-001	9.528510965690609700e-001
-1.993310392221128400e-001	-9.799322103097906100e-001
-1.993310392221128400e-001	9.799322103097906100e-001
7.692851790879432500e-001	-5.717390894505592900e-001
7.692851790879432500e-001	5.717390894505592900e-001
8.586534356574326800e-001	4.258131175833673200e-001
8.586534356574326800e-001	-4.258131175833673200e-001
-7.231279907565465500e-001	6.907140573235797100e-001
-7.231279907565465500e-001	-6.907140573235797100e-001
7.836224078097899800e-001	-5.519134665967812000e-001
7.836224078097899800e-001	5.519134665967812000e-001
3.704218495740213000e-001	-8.842277692989724000e-001
3.704218495740213000e-001	8.842277692989724000e-001
-5.477919863127791400e-001	8.366145705947891600e-001
-5.477919863127791400e-001	-8.366145705947891600e-001
-3.443674709699431900e-001	9.388349401986306700e-001
-3.443674709699431900e-001	-9.388349401986306700e-001
-6.589308985045628800e-001	7.522034771230268200e-001
-6.589308985045628800e-001	-7.522034771230268200e-001
9.571253269819921500e-001	4.902758176751892400e-002
9.571253269819921500e-001	-4.902758176751892400e-002
-4.601954574657617400e-001	-8.878176281916668700e-001
-4.601954574657617400e-001	8.878176281916668700e-001
4.366787806566848100e-001	-8.533980374999776700e-001
4.366787806566848100e-001	8.533980374999776700e-001
-1.153400070029154400e-001	-9.933260707263105200e-001
-1.153400070029154400e-001	9.933260707263105200e-001
-9.902437030763029400e-001	-1.393463616953568900e-001
-9.902437030763029400e-001	1.393463616953568900e-001
5.414128580411545800e-001	-7.910386435486904700e-001

Table 4.2. Continued.

5.414128580411545800e-001	7.910386435486904700e-001
9.330972827954913100e-001	-2.187455488165105400e-001
9.330972827954913100e-001	2.187455488165105400e-001
-9.417011392636105400e-001	3.364505376866249200e-001
-9.417011392636105400e-001	-3.364505376866249200e-001
2.086720946109529900e-001	9.359838924953776100e-001
2.086720946109529900e-001	-9.359838924953776100e-001
8.357952318765807800e-001	-4.691185902792962500e-001
8.357952318765807800e-001	4.691185902792962500e-001
-7.404337370054856500e-001	6.721293633699478900e-001
-7.404337370054856500e-001	-6.721293633699478900e-001
9.583797712115493800e-001	0.000000000000000000e+000
-9.793958142797571900e-001	2.019500902978042800e-001
-9.793958142797571900e-001	-2.019500902978042800e-001
-1.628484854339060900e-001	-9.866510886792173600e-001
-1.628484854339060900e-001	9.866510886792173600e-001
6.002077340685290100e-001	-7.473763773545671900e-001
6.002077340685290100e-001	7.473763773545671900e-001
-9.960450041798283300e-001	-8.885015277648537800e-002
-9.960450041798283300e-001	8.885015277648537800e-002
-7.053469826480633900e-001	7.088622109192085400e-001
-7.053469826480633900e-001	-7.088622109192085400e-001
3.250832922973115900e-001	-9.019314350348206500e-001
3.250832922973115900e-001	9.019314350348206500e-001
9.135977054525220600e-001	2.896278421183926500e-001
9.135977054525220600e-001	-2.896278421183926500e-001
-7.736304147224183400e-001	6.336371054605448600e-001
-7.736304147224183400e-001	-6.336371054605448600e-001
9.505503139309642300e-001	-1.222829975332508600e-001
9.505503139309642300e-001	1.222829975332508600e-001
-8.608902378408900300e-001	-5.087907216039363500e-001
-8.608902378408900300e-001	5.087907216039363500e-001
-3.085421666973365800e-001	-9.512106661353781900e-001
-3.085421666973365800e-001	9.512106661353781900e-001
2.789682612169626700e-001	-9.173117841197369100e-001
2.789682612169626700e-001	9.173117841197369100e-001
9.026238513382041400e-002	-9.557992067514725600e-001
9.026238513382041400e-002	9.557992067514725600e-001
7.233191307356741100e-001	-6.289136423537065000e-001
7.233191307356741100e-001	6.289136423537065000e-001
9.272015642063485800e-001	2.425385960306551400e-001
9.272015642063485800e-001	-2.425385960306551400e-001
-9.133379191588249000e-001	-4.072024624515792600e-001

Table 4.2. Continued.

-9.133379191588249000e-001	4.072024624515792600e-001
4.351411686776529600e-002	-9.610165797717193900e-001
4.351411686776529600e-002	9.610165797717193900e-001
6.555299687581561700e-001	-6.993263980473997100e-001
6.555299687581561700e-001	6.993263980473997100e-001
-9.678693482997905400e-001	-2.514536232026515200e-001
-9.678693482997905400e-001	2.514536232026515200e-001
-3.367678895840671900e-003	-9.701336308507806400e-001
-3.367678895840671900e-003	9.701336308507806400e-001
5.004613594401914300e-001	-8.175855615924808600e-001
5.004613594401914300e-001	8.175855615924808600e-001
8.792760532834974500e-001	-3.813858350236309400e-001
8.792760532834974500e-001	3.813858350236309400e-001
-7.142872232109810400e-001	6.998526721800384400e-001
-7.142872232109810400e-001	-6.998526721800384400e-001
-4.261061892034577200e-001	-9.046731539746870600e-001
-4.261061892034577200e-001	9.046731539746870600e-001
-6.962819784804530300e-001	7.177683515197286200e-001
-6.962819784804530300e-001	-7.177683515197286200e-001
7.070468277569470500e-001	-6.471639685319049400e-001
7.070468277569470500e-001	6.471639685319049400e-001
2.042769843567140600e-002	-9.642751113081878800e-001
2.042769843567140600e-002	9.642751113081878800e-001
6.374961156308268700e-001	-7.158142664859548100e-001
6.374961156308268700e-001	7.158142664859548100e-001
3.567969540311676200e-001	0.000000000000000000e+000
Gain	0.0000122283

Table 4.3. Coefficients of the Subfilters of 495th-Order Low-Pass Filter

m	$h_m(0)$	$h_m(1)$	$h_m(2)$	$h_m(3)$	$h_m(4)$
1	1.00000000	1.00000000	0.00000000	0.00000000	0.00000000
2	1.00000000	-2.00180747	1.00000000	0.00000000	0.00000000
3	1.00000000	-3.15951146	1.00000000	0.00000000	0.00000000
4	1.00000000	0.47182144	1.00000000	0.00000000	0.00000000
5	1.00000000	1.86568788	1.00000000	0.00000000	0.00000000
6	1.00000000	1.66787312	1.00000000	0.00000000	0.00000000
7	1.00000000	0.89782044	1.00000000	0.00000000	0.00000000
8	1.00000000	1.99854649	1.00000000	0.00000000	0.00000000
9	1.00000000	1.79430076	1.00000000	0.00000000	0.00000000
10	1.00000000	1.13771960	1.00000000	0.00000000	0.00000000
11	1.00000000	1.00928428	1.00000000	0.00000000	0.00000000
12	1.00000000	1.96376563	1.00000000	0.00000000	0.00000000
13	1.00000000	0.66496053	1.00000000	0.00000000	0.00000000
14	1.00000000	1.98386665	1.00000000	0.00000000	0.00000000
15	1.00000000	0.14827073	1.00000000	0.00000000	0.00000000
16	1.00000000	1.63928004	1.00000000	0.00000000	0.00000000
17	1.00000000	0.75949939	1.00000000	0.00000000	0.00000000
18	1.00000000	1.15849254	1.00000000	0.00000000	0.00000000
19	1.00000000	1.70872206	1.00000000	0.00000000	0.00000000
20	1.00000000	1.89990053	1.00000000	0.00000000	0.00000000
21	1.00000000	0.64106339	1.00000000	0.00000000	0.00000000
22	1.00000000	1.51455361	1.00000000	0.00000000	0.00000000
23	1.00000000	1.75931864	1.00000000	0.00000000	0.00000000
24	1.00000000	1.94197300	1.00000000	0.00000000	0.00000000
25	1.00000000	0.34994570	1.00000000	0.00000000	0.00000000
26	1.00000000	1.17909953	1.00000000	0.00000000	0.00000000
27	1.00000000	1.99418868	1.00000000	0.00000000	0.00000000
28	1.00000000	1.84677334	1.00000000	0.00000000	0.00000000
29	1.00000000	1.92918708	1.00000000	0.00000000	0.00000000
30	1.00000000	0.52045489	1.00000000	0.00000000	0.00000000
31	1.00000000	1.19952550	1.00000000	0.00000000	0.00000000
32	1.00000000	1.57895814	1.00000000	0.00000000	0.00000000
33	1.00000000	0.27763760	1.00000000	0.00000000	0.00000000
34	1.00000000	1.56322336	1.00000000	0.00000000	0.00000000
35	1.00000000	0.94285574	1.00000000	0.00000000	0.00000000
36	1.00000000	0.82923433	1.00000000	0.00000000	0.00000000
37	1.00000000	1.97276241	1.00000000	0.00000000	0.00000000
38	1.00000000	1.25960224	1.00000000	0.00000000	0.00000000
39	1.00000000	1.62460100	1.00000000	0.00000000	0.00000000

Table 4.3. Continued.

40	1.00000000	1.81616357	1.00000000	0.00000000	0.00000000
41	1.00000000	0.20800460	1.00000000	0.00000000	0.00000000
42	1.00000000	1.98967105	1.00000000	0.00000000	0.00000000
43	1.00000000	1.29866100	1.00000000	0.00000000	0.00000000
44	1.00000000	1.03109970	1.00000000	0.00000000	0.00000000
45	1.00000000	1.73459047	1.00000000	0.00000000	0.00000000
46	1.00000000	1.35566135	1.00000000	0.00000000	0.00000000
47	1.00000000	0.42305942	1.00000000	0.00000000	0.00000000
48	1.00000000	1.98693294	1.00000000	0.00000000	0.00000000
49	1.00000000	1.89179467	1.00000000	0.00000000	0.00000000
50	1.00000000	0.16603999	1.00000000	0.00000000	0.00000000
51	1.00000000	1.95350484	1.00000000	0.00000000	0.00000000
52	1.00000000	0.78284730	1.00000000	0.00000000	0.00000000
53	1.00000000	1.49785071	1.00000000	0.00000000	0.00000000
54	1.00000000	1.37421410	1.00000000	0.00000000	0.00000000
55	1.00000000	1.07427570	1.00000000	0.00000000	0.00000000
56	1.00000000	0.87509673	1.00000000	0.00000000	0.00000000
57	1.00000000	1.83687044	1.00000000	0.00000000	0.00000000
58	1.00000000	0.13446290	1.00000000	0.00000000	0.00000000
59	1.00000000	0.30158478	1.00000000	0.00000000	0.00000000
60	1.00000000	1.46368320	1.00000000	0.00000000	0.00000000
61	1.00000000	1.99935346	1.00000000	0.00000000	0.00000000
62	1.00000000	1.77127383	1.00000000	0.00000000	0.00000000
63	1.00000000	1.69537765	1.00000000	0.00000000	0.00000000
64	1.00000000	0.54471937	1.00000000	0.00000000	0.00000000
65	1.00000000	0.18630833	1.00000000	0.00000000	0.00000000
66	1.00000000	1.91515879	1.00000000	0.00000000	0.00000000
67	1.00000000	1.99596245	1.00000000	0.00000000	0.00000000
68	1.00000000	1.05274711	1.00000000	0.00000000	0.00000000
69	1.00000000	0.73603514	1.00000000	0.00000000	0.00000000
70	1.00000000	1.92233391	1.00000000	0.00000000	0.00000000
71	1.00000000	1.23977418	1.00000000	0.00000000	0.00000000
72	1.00000000	0.59304661	1.00000000	0.00000000	0.00000000
73	1.00000000	1.59442561	1.00000000	0.00000000	0.00000000
74	1.00000000	1.60962543	1.00000000	0.00000000	0.00000000
75	1.00000000	0.25398027	1.00000000	0.00000000	0.00000000
76	1.00000000	1.96842648	1.00000000	0.00000000	0.00000000
77	1.00000000	0.37428044	1.00000000	0.00000000	0.00000000
78	1.00000000	1.65370280	1.00000000	0.00000000	0.00000000
79	1.00000000	1.87469926	1.00000000	0.00000000	0.00000000
80	1.00000000	0.96514501	1.00000000	0.00000000	0.00000000
81	1.00000000	0.56893060	1.00000000	0.00000000	0.00000000
82	1.00000000	0.98727040	1.00000000	0.00000000	0.00000000

Table 4.3. Continued.

83	1.00000000	1.80538223	1.00000000	0.00000000	0.00000000
84	1.00000000	0.71241237	1.00000000	0.00000000	0.00000000
85	1.00000000	0.44745165	1.00000000	0.00000000	0.00000000
86	1.00000000	1.68175115	1.00000000	0.00000000	0.00000000
87	1.00000000	1.99983872	1.00000000	0.00000000	0.00000000
88	1.00000000	1.99741650	1.00000000	0.00000000	0.00000000
89	1.00000000	1.74709257	1.00000000	0.00000000	0.00000000
90	1.00000000	0.80611702	1.00000000	0.00000000	0.00000000
91	1.00000000	1.94788941	1.00000000	0.00000000	0.00000000
92	1.00000000	0.12659146	1.00000000	0.00000000	0.00000000
93	1.00000000	1.27921316	1.00000000	0.00000000	0.00000000
94	1.00000000	1.85638851	1.00000000	0.00000000	0.00000000
95	1.00000000	1.97678625	1.00000000	0.00000000	0.00000000
96	1.00000000	0.49615848	1.00000000	0.00000000	0.00000000
97	1.00000000	1.53102449	1.00000000	0.00000000	0.00000000
98	1.00000000	1.90768154	1.00000000	0.00000000	0.00000000
99	1.00000000	1.33685433	1.00000000	0.00000000	0.00000000
100	1.00000000	1.11672580	1.00000000	0.00000000	0.00000000
101	1.00000000	1.21973280	1.00000000	0.00000000	0.00000000
102	1.00000000	1.78292303	1.00000000	0.00000000	0.00000000
103	1.00000000	0.39866208	1.00000000	0.00000000	0.00000000
104	1.00000000	1.44625598	1.00000000	0.00000000	0.00000000
105	1.00000000	1.09558397	1.00000000	0.00000000	0.00000000
106	1.00000000	0.68873494	1.00000000	0.00000000	0.00000000
107	1.00000000	1.31786180	1.00000000	0.00000000	0.00000000
108	1.00000000	0.92039091	1.00000000	0.00000000	0.00000000
109	1.00000000	0.23068001	1.00000000	0.00000000	0.00000000
110	1.00000000	1.98048741	1.00000000	0.00000000	0.00000000
111	1.00000000	1.88340228	1.00000000	0.00000000	0.00000000
112	1.00000000	1.48086747	1.00000000	0.00000000	0.00000000
113	1.00000000	1.95879163	1.00000000	0.00000000	0.00000000
114	1.00000000	0.32569697	1.00000000	0.00000000	0.00000000
115	1.00000000	1.99209001	1.00000000	0.00000000	0.00000000
116	1.00000000	1.41069397	1.00000000	0.00000000	0.00000000
117	1.00000000	1.54726083	1.00000000	0.00000000	0.00000000
118	1.00000000	1.72178048	1.00000000	0.00000000	0.00000000
119	1.00000000	0.61708433	1.00000000	0.00000000	0.00000000
120	1.00000000	1.82667584	1.00000000	0.00000000	0.00000000
121	1.00000000	1.93573870	1.00000000	0.00000000	0.00000000
122	1.00000000	1.42857445	1.00000000	0.00000000	0.00000000
123	1.00000000	0.85221238	1.00000000	0.00000000	0.00000000
124	1.00000000	1.39256396	1.00000000	0.00000000	0.00000000
125	1.00000000	-3.95652300	5.91368766	-3.95652300	1.00000000

Table 4.3. Continued.

126	1.00000000	-1.45266130	2.53373770	-1.45266130	1.00000000
127	1.00000000	-3.63106166	5.29743240	-3.63106166	1.00000000
128	1.00000000	-3.78424139	5.58089042	-3.78424139	1.00000000
129	1.00000000	-2.34465621	3.37906170	-2.34465621	1.00000000
130	1.00000000	-0.57420870	2.08917458	-0.57420870	1.00000000
131	1.00000000	-3.33101932	4.77613894	-3.33101932	1.00000000
132	1.00000000	-2.81160089	3.97991535	-2.81160089	1.00000000
133	1.00000000	-1.91376780	2.92114165	-1.91376780	1.00000000
134	1.00000000	-0.96957926	2.24165972	-0.96957926	1.00000000
135	1.00000000	-3.99182334	5.98370594	-3.99182334	1.00000000
136	1.00000000	-1.26155743	2.40427644	-1.26155743	1.00000000
137	1.00000000	-3.84611277	5.69870313	-3.84611277	1.00000000
138	1.00000000	-3.92003229	5.84196197	-3.92003229	1.00000000
139	1.00000000	-3.71249167	5.44666017	-3.71249167	1.00000000
140	1.00000000	-3.54017163	5.13477646	-3.54017163	1.00000000
141	1.00000000	-2.00260775	3.00797229	-2.00260775	1.00000000
142	1.00000000	-3.08729204	4.38575869	-3.08729204	1.00000000
143	1.00000000	-3.44006624	4.96039981	-3.44006624	1.00000000
144	1.00000000	-0.67335190	2.12010442	-0.67335190	1.00000000
145	1.00000000	-3.93956611	5.88027477	-3.93956611	1.00000000
146	1.00000000	-1.73251590	2.75620453	-1.73251590	1.00000000
147	1.00000000	-2.17641144	3.18923567	-2.17641144	1.00000000
148	1.00000000	-0.27825201	2.02574235	-0.27825201	1.00000000
149	1.00000000	-3.15132819	4.48545459	-3.15132819	1.00000000
150	1.00000000	-0.77238754	2.15588008	-0.77238754	1.00000000
151	1.00000000	-2.58561492	3.67553355	-2.58561492	1.00000000
152	1.00000000	-2.88336466	4.08190643	-2.88336466	1.00000000
153	1.00000000	-3.98266071	5.96547209	-3.98266071	1.00000000
154	1.00000000	-4.00230414	6.00461435	-4.00230414	1.00000000
155	1.00000000	-2.42655184	3.47657417	-2.42655184	1.00000000
156	1.00000000	-1.64022973	2.67852392	-1.64022973	1.00000000
157	1.00000000	-3.38664254	4.86938617	-3.38664254	1.00000000
158	1.00000000	-1.06750710	2.29146176	-1.06750710	1.00000000
159	1.00000000	-3.74958947	5.51574241	-3.74958947	1.00000000
160	1.00000000	-0.47513961	2.06313849	-0.47513961	1.00000000
161	1.00000000	-3.21332275	4.58392172	-3.21332275	1.00000000
162	1.00000000	-3.58678448	5.21767939	-3.58678448	1.00000000
163	1.00000000	-3.27323332	4.68090133	-3.27323332	1.00000000
164	1.00000000	-1.54692247	2.60430458	-1.54692247	1.00000000
165	1.00000000	-3.99837269	5.99676496	-3.99837269	1.00000000
166	1.00000000	-1.82371646	2.83714673	-1.82371646	1.00000000
167	1.00000000	-2.26125759	3.28319956	-2.26125759	1.00000000
168	1.00000000	-3.89793503	5.79885091	-3.89793503	1.00000000

Table 4.3. Continued.

169	1.00000000	-0.87117162	2.19642975	-0.87117162	1.00000000
170	1.00000000	-3.49125440	5.04894120	-3.49125440	1.00000000
171	1.00000000	-2.50688949	3.57548393	-2.50688949	1.00000000
172	1.00000000	-1.35751565	2.46700469	-1.35751565	1.00000000
173	1.00000000	-3.81642327	5.64193129	-3.81642327	1.00000000
174	1.00000000	-3.97089119	5.94211195	-3.97089119	1.00000000
175	1.00000000	-1.16486148	2.34571268	-1.16486148	1.00000000
176	1.00000000	-0.37638555	2.04200954	-0.37638555	1.00000000
177	1.00000000	-3.02125691	4.28509395	-3.02125691	1.00000000
178	1.00000000	-3.87328955	5.75105580	-3.87328955	1.00000000
179	1.00000000	-0.18106743	2.01419008	-0.18106743	1.00000000
180	1.00000000	-2.73802209	3.87801331	-2.73802209	1.00000000
181	1.00000000	0.01389172	2.00372543	0.01389172	1.00000000
182	1.00000000	-2.09017542	3.09741405	-2.09017542	1.00000000
183	1.00000000	-3.67297347	5.37382624	-3.67297347	1.00000000
184	1.00000000	-2.95326596	4.18372168	-2.95326596	1.00000000
185	1.00000000	-0.08477442	2.00702514	-0.08477442	1.00000000
186	1.00000000	-2.66267640	3.77646428	-2.66267640	1.00000000
Gain	0.0000122283				

4.2 Minimum-Phase Filter Design

Even-order, zero-phase FIR filters were design using windows method or Park-McClellan method [12]. We use these filters to design the minimum-phase filters.

We design a 500^{th}-order band-pass FIR filter using the Parks-McClellan program. This is an even-order, zero-phase FIR filter. The input specifications used to obtain this filter design are list in Table 4.4. The impulse response coefficients of the minimum-phase filter are given in Table 4.5. The subfilter coefficients were calculated and are listed in Table 4.6.

The minimum-phase filter was designed with the filter order 250. The frequency response magnitude of the prototype 500^{th}-order band-pass filter is illustrated in Figure 4.2. The frequency response magnitude of the minimum-phase band-pass filter with 250^{th}-order is illustrated in Figure 4.3.

Table 4.4. Specifications For 500^{th}-Order Band-Pass Filter

Band	Lower Band Edge	Upper Band Edge	Gain	Weight
1	0.00	0.20	0.0	1.0
2	0.21	0.35	1.0	1.0
3	0.36	0.50	0.0	1.0

Table 4.5. The Impulse Response Coefficients of 250th-Order Minimum-Phase Filter

n	$h(n)$	n	$h(n)$
0	2.69550920e-002	34	5.88317640e-002
1	-1.61130134e-002	35	-2.39369089e-002
2	-6.51518374e-002	36	-2.54666975e-002
3	7.28244670e-002	37	6.30403805e-003
4	8.96386431e-002	38	-7.75811094e-003
5	-1.65554101e-001	39	3.39155272e-002
6	-5.99634592e-002	40	6.82250610e-003
7	2.45862922e-001	41	-5.01187943e-002
8	-2.65381897e-002	42	9.81766093e-003
9	-2.51451641e-001	43	2.49069737e-002
10	1.13340543e-001	44	-4.08483896e-003
11	1.68014290e-001	45	9.05687277e-003
12	-1.25993802e-001	46	-2.50169078e-002
13	-5.58307352e-002	47	-1.53946167e-002
14	4.64864166e-002	48	4.10638685e-002
15	-8.15861912e-004	49	6.79797233e-004
16	5.98652689e-002	50	-2.17019418e-002
17	-2.23405507e-002	51	1.61998336e-003
18	-1.05845305e-001	52	-1.06498683e-002
19	6.65133311e-002	53	1.78918563e-002
20	7.00145970e-002	54	2.09526498e-002
21	-5.80343645e-002	55	-3.19443014e-002
22	-1.24423534e-002	56	-7.74639362e-003
23	-8.43989176e-003	57	1.70430296e-002
24	-6.07119590e-004	58	-1.98330799e-004
25	6.79619399e-002	59	1.23159904e-002
26	-2.82582662e-002	60	-1.15249168e-002
27	-6.65024705e-002	61	-2.40143637e-002
28	4.12456160e-002	62	2.31277962e-002
29	2.17920421e-002	63	1.18349264e-002
30	-4.93503388e-003	64	-1.18672988e-002
31	6.16903037e-003	65	4.47514312e-005
32	-4.66825963e-002	66	-1.36642260e-002
33	6.46419580e-003	67	5.37996106e-003

Table 4.5. Continued.

68	2.49110745e-002	108	-1.41335069e-003
69	-1.48368438e-002	109	-1.26366291e-002
70	-1.32648418e-002	110	4.86629285e-003
71	7.01275489e-003	111	4.05350395e-003
72	-1.29469644e-003	112	6.60791753e-004
73	1.39542961e-002	113	3.15262735e-003
74	3.80372188e-004	114	-9.90073998e-003
75	-2.39493923e-002	115	-1.73143732e-003
76	7.70745764e-003	116	1.11597747e-002
77	1.25421281e-002	117	-1.81712834e-003
78	-3.26454377e-003	118	-2.47763865e-003
79	3.42963245e-003	119	-1.55069416e-003
80	-1.33194172e-002	120	-5.01202498e-003
81	-5.28544576e-003	121	8.43610213e-003
82	2.13500507e-002	122	4.12619728e-003
83	-1.90317385e-003	123	-8.84026101e-003
84	-1.03104977e-002	124	-7.12331785e-005
85	7.03872668e-004	125	6.96824770e-004
86	-5.93963763e-003	126	1.41351982e-003
87	1.12949962e-002	127	6.50091518e-003
88	9.24816784e-003	128	-6.37485512e-003
89	-1.76637917e-002	129	-5.53015604e-003
90	-2.08885157e-003	130	6.23080325e-003
91	7.21212309e-003	131	8.28698470e-004
92	2.99854930e-004	132	9.82237972e-004
93	8.20904698e-003	133	-5.18959958e-004
94	-8.51732727e-003	134	-7.24890290e-003
95	-1.18857019e-002	135	4.01458274e-003
96	1.31376606e-002	136	5.84558360e-003
97	4.28477773e-003	137	-3.87386717e-003
98	-4.06587519e-003	138	-7.24967121e-004
99	3.46267565e-006	139	-2.19263157e-003
100	-9.83308651e-003	140	-6.84617889e-004
101	5.02603525e-003	141	7.20279982e-003
102	1.30156103e-002	142	-1.73460070e-003
103	-8.68349598e-003	143	-5.43774522e-003
104	-4.82856372e-003	144	1.88981022e-003
105	1.27246088e-003	145	-4.79649344e-005
106	-1.37457357e-003	146	2.73719497e-003
107	1.04038236e-002	147	2.05188990e-003

Table 4.5. Continued.

148	-6.42402089e-003	188	-1.23422930e-003
149	-1.22099812e-004	189	1.96189659e-003
150	4.36006866e-003	190	2.10344306e-004
151	-6.40627287e-004	191	-3.25629690e-005
152	1.05881357e-003	192	-2.42101469e-004
153	-2.60297792e-003	193	-1.85787000e-003
154	-3.18235476e-003	194	1.43142608e-003
155	5.20318897e-003	195	1.39697272e-003
156	1.33590124e-003	196	-1.12636750e-003
157	-2.87020222e-003	197	-2.90840122e-004
158	1.06706390e-004	198	-4.10303380e-004
159	-1.90406899e-003	199	-7.73373387e-005
160	1.98693630e-003	200	1.87484205e-003
161	3.87329244e-003	201	-7.89379311e-004
162	-3.62251478e-003	202	-1.27921753e-003
163	-1.89288464e-003	203	4.83809937e-004
164	1.60738061e-003	204	1.34956430e-004
165	-1.72813892e-004	205	6.24563177e-004
166	2.67073339e-003	206	4.30208074e-004
167	-9.90613402e-004	207	-1.75608572e-003
168	-3.88508386e-003	208	2.23487136e-004
169	2.17904309e-003	209	9.09541878e-004
170	1.73561402e-003	210	-4.59212125e-005
171	-5.45830975e-004	211	1.87213597e-004
172	4.64241703e-004	212	-6.12459236e-004
173	-2.79480478e-003	213	-7.34169845e-004
174	1.14100733e-005	214	1.34515115e-003
175	3.56266848e-003	215	5.78964018e-005
176	-9.38465594e-004	216	-5.12788018e-004
177	-1.33406641e-003	217	-1.12305974e-004
178	-6.59844745e-005	218	-3.92905878e-004
179	-1.06357339e-003	219	4.54507459e-004
180	2.66620883e-003	220	8.90073788e-004
181	8.44361364e-004	221	-9.45274027e-004
182	-2.82259391e-003	222	-1.90128863e-004
183	1.45164957e-004	223	1.52924726e-004
184	6.38957622e-004	224	1.12897196e-004
185	3.03695732e-004	225	5.35025484e-004
186	1.62630818e-003	226	-1.55196446e-004
187	-2.08476852e-003	227	-9.09961667e-004

Table 4.5. Continued.

228	6.90769773e-004	239	8.16886901e-004
229	1.49929049e-004	240	3.32001556e-004
230	1.12214747e-004	241	-1.02389073e-003
231	-5.28203422e-005	242	2.46204465e-004
232	-5.79968669e-004	243	-6.83677063e-004
233	-6.97518443e-005	244	9.81293553e-004
234	9.97743884e-004	245	1.18610775e-003
235	-4.61582731e-004	246	-1.29299161e-003
236	-5.68348762e-006	247	-1.63884053e-004
237	-3.67623456e-004	248	-3.10844629e-003
238	-1.87950126e-004	249	-6.84183535e-004
		250	-1.25026010e-003

Table 4.6. Coefficients of the Subfilters of 250th-Order Minimum-Phase Filter

m	$h_m(0)$	$h_m(1)$	$h_m(2)$
1	1.00000000	-1.00000000	0.00000000
2	1.00000000	0.98079048	0.00000000
3	1.00000000	-1.99939764	1.00000000
4	1.00000000	-1.99743802	1.00000000
5	1.00000000	-1.99435287	1.00000000
6	1.00000000	-1.99006640	1.00000000
7	1.00000000	-1.98419863	1.00000000
8	1.00000000	-1.97744338	1.00000000
9	1.00000000	-1.96896091	1.00000000
10	1.00000000	-1.95975421	1.00000000
11	1.00000000	-1.94936702	1.00000000
12	1.00000000	-1.93704419	1.00000000
13	1.00000000	-1.92424281	1.00000000
14	1.00000000	-1.91028234	1.00000000
15	1.00000000	-1.89418873	1.00000000
16	1.00000000	-1.87786497	1.00000000
17	1.00000000	-1.86041005	1.00000000
18	1.00000000	-1.84063655	1.00000000
19	1.00000000	-1.82088258	1.00000000
20	1.00000000	-1.79869247	1.00000000
21	1.00000000	-1.77669007	1.00000000
22	1.00000000	-1.75361745	1.00000000
23	1.00000000	-1.72794571	1.00000000
24	1.00000000	-1.70271039	1.00000000
25	1.00000000	-1.67644941	1.00000000
26	1.00000000	-1.64744102	1.00000000
27	1.00000000	-1.61911528	1.00000000
28	1.00000000	-1.58981425	1.00000000
29	1.00000000	-1.55763302	1.00000000
30	1.00000000	-1.52637683	1.00000000
31	1.00000000	-1.49420121	1.00000000
32	1.00000000	-1.45902888	1.00000000
33	1.00000000	-1.42501874	1.00000000
34	1.00000000	-1.38794292	1.00000000
35	1.00000000	-1.35218541	1.00000000
36	1.00000000	-1.31561339	1.00000000
37	1.00000000	-1.27824889	1.00000000
38	1.00000000	-1.23770598	1.00000000

Table 4.6. Continued.

39	1.00000000	-1.19877860	1.00000000
40	1.00000000	-1.15912912	1.00000000
41	1.00000000	-1.11623706	1.00000000
42	1.00000000	-1.07517415	1.00000000
43	1.00000000	-1.03346360	1.00000000
44	1.00000000	-0.99113052	1.00000000
45	1.00000000	-0.94549806	1.00000000
46	1.00000000	-0.90196198	1.00000000
47	1.00000000	-0.85788258	1.00000000
48	1.00000000	-0.81328643	1.00000000
49	1.00000000	-0.77103211	1.00000000
50	1.00000000	-0.72551145	1.00000000
51	1.00000000	-0.68532143	1.00000000
52	1.00000000	-0.64771873	1.00000000
53	1.00000000	-0.62153431	1.00000000
54	1.00000000	1.27824889	1.00000000
55	1.00000000	1.29936261	1.00000000
56	1.00000000	1.32712832	1.00000000
57	1.00000000	1.36120200	1.00000000
58	1.00000000	1.39455502	1.00000000
59	1.00000000	1.42931738	1.00000000
60	1.00000000	1.46321876	1.00000000
61	1.00000000	1.49623876	1.00000000
62	1.00000000	1.52835748	1.00000000
63	1.00000000	1.55955558	1.00000000
64	1.00000000	1.59167381	1.00000000
65	1.00000000	1.62091440	1.00000000
66	1.00000000	1.64917861	1.00000000
67	1.00000000	1.67644941	1.00000000
68	1.00000000	1.70431780	1.00000000
69	1.00000000	1.72948852	1.00000000
70	1.00000000	1.75361745	1.00000000
71	1.00000000	1.77809671	1.00000000
72	1.00000000	1.80003178	1.00000000
73	1.00000000	1.82088258	1.00000000
74	1.00000000	1.84183448	1.00000000
75	1.00000000	1.86041005	1.00000000
76	1.00000000	1.87786497	1.00000000
77	1.00000000	1.89517118	1.00000000
78	1.00000000	1.91028234	1.00000000
79	1.00000000	1.92424281	1.00000000
80	1.00000000	1.93780561	1.00000000
81	1.00000000	1.94936702	1.00000000

Table 4.6. Continued.

82	1.00000000	1.96036427	1.00000000
83	1.00000000	1.96949700	1.00000000
84	1.00000000	1.97790053	1.00000000
85	1.00000000	1.98458118	1.00000000
86	1.00000000	1.99036945	1.00000000
87	1.00000000	1.99458091	1.00000000
88	1.00000000	1.99773910	1.00000000
89	1.00000000	1.99966116	1.00000000
90	1.00000000	-0.48961456	0.94150887
91	1.00000000	1.10157713	0.93057079
92	1.00000000	0.39901300	0.91253785
93	1.00000000	0.72371328	0.91945231
94	1.00000000	1.02069482	0.92329740
95	1.00000000	-0.34959396	0.92329146
96	1.00000000	0.49421737	0.91731778
97	1.00000000	-0.11476603	0.91987162
98	1.00000000	0.85471386	0.92018705
99	1.00000000	0.07722093	0.91885830
100	1.00000000	0.58770447	0.91865366
101	1.00000000	-0.06697373	0.91962528
102	1.00000000	-0.25674118	0.92109587
103	1.00000000	0.17379489	0.91779846
104	1.00000000	1.14651397	0.94150979
105	1.00000000	0.67891839	0.91923607
106	1.00000000	0.93898687	0.92110731
107	1.00000000	0.26994415	0.91434559
108	1.00000000	-0.44113910	0.93056845
109	1.00000000	1.06087344	0.92573865
110	1.00000000	0.81163251	0.91990024
111	1.00000000	-0.16236109	0.92016613
112	1.00000000	0.22209624	0.91663906
113	1.00000000	0.02904779	0.91915315
114	1.00000000	-0.39536331	0.92573465
115	1.00000000	0.98014443	0.92193430
116	1.00000000	0.35519068	0.90825354
117	1.00000000	0.63358269	0.91898521
118	1.00000000	-0.01902501	0.91939600
119	1.00000000	0.54126194	0.91815746
120	1.00000000	0.44659063	0.91572898
121	1.00000000	0.12547682	0.91844748
122	1.00000000	0.89717368	0.92056657
123	1.00000000	-0.30339189	0.92192596
124	1.00000000	-0.20970699	0.92055107

Table 4.6. Continued.

125	1.00000000	0.76795748	0.91966497
126	1.00000000	0.31518726	0.91009767
Gain	0.0269550920		

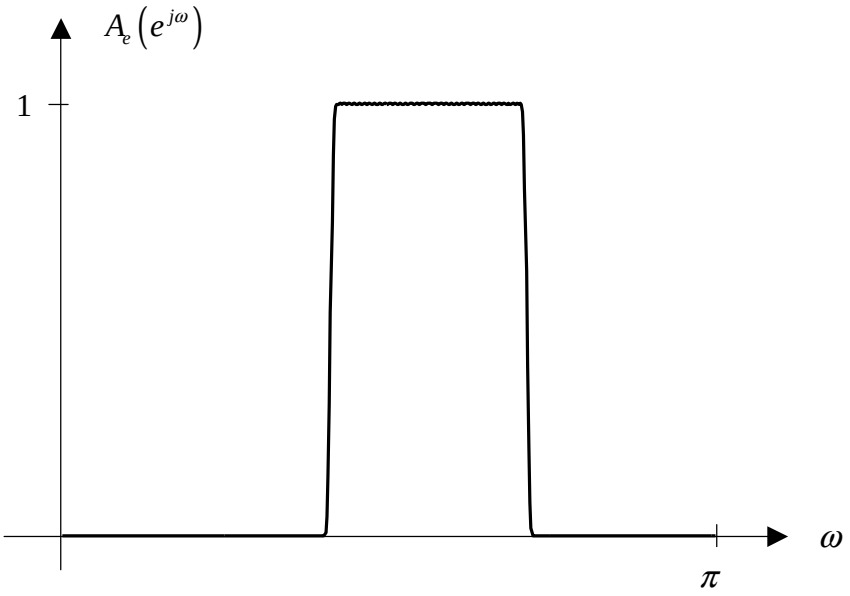

Figure 4.2. Frequency Response Magnitude of 500th-Order Band-Pass Filter

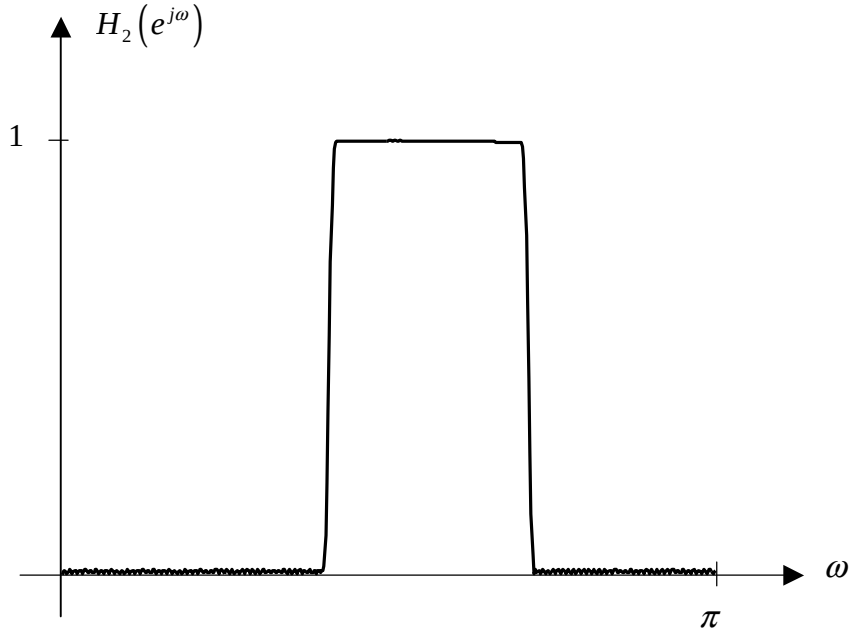

Figure 4.3. Frequency Response Magnitude of 250th-Order Minimum-Phase Band-Pass Filter

In both of these programs, if the filter order is higher than 500, some numerical instabilities have been observed in polynomial root finding. For example, if we design a 709^{th}-order FIR filter with specifications listed in Table 4.7, not all the roots of the transfer function could be found, so we can't get the correct decomposition of this filter and we can't get the correct minimum-phase filter either.

Both programs can work well at very high order FIR filter sometimes, such as a 1000^{th}-order band-pass filter with specification listed in Table 4.8. So when the FIR filter order is higher than 500, the results depend on the type and specifications of the filter.

Table 4.7. Specifications for 709^{th}-Order Low-Pass Filter

Band	Lower Band Edge	Upper Band Edge	Gain	Weight
1	0.00	0.10	1.0	1.0
2	0.11	0.50	0.0	1.0

Table 4.8. Specifications for 1000^{th}-Order Band-Pass Filter

Band	Lower Band Edge	Upper Band Edge	Gain	Weight
1	0.000	0.100	0.0	1.0
2	0.101	0.400	1.0	1.0
3	0.401	0.500	0.0	1.0

CHAPTER 5

SUMMARY AND CONCLUSIONS

There were two purposes of this research: (a) Decompose any arbitrary linear-phase FIR filter into the cascade connection of subfilters. (b) Given an even-order, linear-phase, equal-ripple FIR digital filter, design a minimum-phase FIR filter which has the square root of the magnitude response of the prototype FIR filter.

In FIR filter decomposition, the method consists of four discrete searches for the roots of the z-domain transfer function and is constrained to the upper half of the unit circle in the complex z-domain. The four search regions are

a) Search over the two points at $(1,0)$ and $(-1,0)$;

b) Search over the real axis;

c) Search over the perimeter of the unit circle;

d) Search over the interior of the unit circle.

In each case, the root value is obtained via Lang's polynomial root finding method which combines two well-known methods: Muller's method and Newton's method.

In minimum-phase filter design, the method consists of two discrete searches for the roots of the z-domain transfer function, they are

a) Search over the perimeter of the unit circle;

b) Search over the interior of the unit circle.

We modify the prototype FIR filter to get double zeros on the unit circle in the complex z-plane. Also, we use Lang's method to get the root values. We keep the zeros inside the unit circle and simple zeros on the unit circle. These zeros are treated as zeros of the minimum-phase filter. The minimum-phase FIR filter can also be decomposed into the cascade connection of subfilters.

Two computer programs implementing these two methods have been written and tested on many practical FIR filters. For filter orders up to 500, the programs can successfully implement FIR filter decomposition and minimum-phase filter design. For filter orders higher than 500, the results depend on the filter type and specifications.

The method of FIR filter decomposition should find some applications in areas involving digital filter implementations on PLD's and FPGA's. The method of minimum-phase filter design is useful when we want to design some kinds of filters with less delay.

Future work in this area should find more methods for minimum-phase filter design. We followed O. Herrmann and H. W. Schuessler's method to design the minimum-phase in this study, but there are some limits in this method. Therefore we can only design the minimum-phase filter from even-order, equal-ripple FIR filter.

REFERENCES

REFERENCES

[1] T. Parks and J. McClellan, "Chebyshev Approximation for Nonrecursive digital Filters with Linear Phase," *IEEE Trans. Circuit Theory*, Vol. 19, pages 189-194, 1972.

[2] O. Herrmann and H. W. Schuessler, "Design of nonrecursive digital filters with minimum phase," *Electron. Lett*., vol. 6, no.11, pages 329-330, 1970.

[3] Boite R. and H. Leich, "A new procedure for the design of high order minimum phaes FIR digital or CCD filters", *Signal Processing*, Vol. 3, pages 101-108, 1981.

[4] Kamp Y. and Wellekens C.J., "Optimal design of minimum phase FIR filters", *IEEE Trans. On ASSP*, Vol. 31, pages 922-926, 1983.

[5] Mian G.A. and A.P. Naider, "A fast procedure to design equiripple minimum phase FIR filters", *IEEE Trans. On Circuits and Systems*, Vol. 29, No. 5, pages 327-331, 1982.

[6] Y. Lian and Y.C. Lim, "Zeros of Linear Phase FIR Filter with Piecewise Constant Frequency Response," *Electron. Letters*, Vol. 28, pages 203-204, Jan. 1992.

[7] L. Montgomery Smith, " Decomposition of FIR Digital Filters for Realization Via the Cascade Connection of Subfilters", *IEEE Transactions On Signal Processing*. Vol. 46, No. 6, June 1998.

[8] J.G. Proakis and D.G. Manolakis, *Digital Signal Processing: Princeples, Algorithms and Applications*, 3rd ed., Upper Saddle River, NJ: Prentice-Hall, 1996.

[9] V.Y. Pan, "Solving Polynomials with Computers", *American Scientist*, Vol. 86, No. 1, pages 62-69, Jan. – Feb. 1998.

[10] Markus Lang, Bernhard-Christian Frenzel, " Polynomial Root Finding", *IEEE Signal Processing Letters*. Vol. 1, No. 10, October 1994.

[11] William H.et al. *Numerical Recipes in C*. Cambridge University Press, Cambridge, 1992.

[12] L. M. Smith, "Firdesgn.cpp," A C++ program employing digital filter design, Univ. of Tenn. Space Institute, Tullahoma, Tenn., 1998.

[13] Nachtigal, Noel M., Lothar Reichel, and Lloyd N. Trefethen. “A hybrid GMRES algorithm for nonsymmetric linear systems”. *SIAM Journal on Matrix Analysis and Applications*, 13:796-825, July 1992.

APPENDICES

APPENDIX A

C++ PROGRAM FOR FIR FILTERS DECOMPOSITION

```
/*************************************************/
/*************************************************/
/*                                               */
/*          Decomposition of FIR Filter          */
/*                                               */
/*************************************************/
/*************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iostream.h>
#include "complex.cpp"
#define NPLOT 2048
#define PI 3.14159265358979363846

class Polynomial
{
public:
   void read_from_file( );
   unsigned char get_roots();
   void write_to_file( );
   void write_to_screen();
   void write_error_message( unsigned char );
   void check_value();
   ~Polynomial();

private:
  unsigned   char poly_check();
  void       quadratic(Complex *);
  unsigned   char lin_or_quad(Complex *);
  void       hornc(Complex,unsigned char );
  void       horncd(double ,double );
  int        poldef(unsigned char);
  void       monic();

// functions blow are for Newton's Method

  Complex   newton(Complex,double *);
  void      f_value1(Complex *,Complex *,Complex *,Complex );
  void      f_value2(Complex *,Complex *,Complex );

// functions blow are for Muller's Method
```

```
 Complex   muller();
 void      initialize(Complex *,double *);
 void      root_of_parabola();
 void      iteration_equation(double *);
 void      suppress_overflow();
 void      too_big_functionvalues(double * );
 void      convergence_check(int *,double,double,double);
 void      compute_function(double,double* ,double );
 void      check_x_value(Complex *,double *,int*,double , double, double,int * );
 void      root_check(double,int *,int *,int *,Complex);
 void      f_value(int ,Complex *,Complex );

 Complex  x0,x1,x2,     // common points [x0,f(x0)=P(x0)], ... [x2,f(x2)]
          f0,f1,f2,     // of parabola and polynomial
          h1,h2,        // distance between x2 and x1
          q2, *psave, *psave1;          // smaller root of parabola
 int      iter,
          nred,       // the highest exponent of the deflated polynomial
          n,N;        // original degree of the input
 int      distinct,
          indicator;
 double   data;
 double   maxerr;
 double   *matlab;
 static   Complex *p,     // coefficient vector of polynomial
                  *pred,  // coefficient vector of deflated polynom.
                  *root;  // vector of determined roots
 static   unsigned char flag ;
};

unsigned char Polynomial::flag=1;
Complex *Polynomial::p;
Complex *Polynomial::pred;
Complex *Polynomial::root;


// read coefficients stored in file FILENAME

void Polynomial :: read_from_file( )
{
   char filename[32];
   int  i;        // counter
   FILE *file_ptr;

// open file
```

```
    printf("*********************************\n");
    printf("*** FIR Filter Decompostion ***\n");
    printf("*********************************\n\n");
    printf("Enter filter file name: ");
    scanf("%s", filename);
    if ( (file_ptr = fopen( filename, "r" )) == NULL ) {
        printf( "Can't open file %s!\n", filename );
        exit(0);
    } else {
        file_ptr=fopen(filename,"r");

// read degree

   (void) fscanf( file_ptr, "%d %d %d", &n, &distinct, &indicator);

// allocate the memory

    N=n;
    p =  new Complex  [ n + 1 ];
    pred = new Complex  [ n + 1 ];
    root = new Complex  [ n ];
    psave1=p;
    psave=pred;
    flag=0;

// read coefficients

          for (int i=n-1; i>=distinct-1; i--){
                    fscanf(file_ptr,"%lf", &data);
                    if (fabs(data)<1e-8) data=0.0;
                    p[i].r=data;
                    p[i].i=0.0;
          }
          if (indicator == 0){
                              for ( i=0; i<distinct; i++){
                                        p[i].r = p[n-i-1].r;
                                        p[i].i =0.0;
                              }
          }
          else{
                              for (i=0; i<distinct; i++){
                                        p[i].r = -p[n-i-1].r;
                                        p[i].i = 0.0;
                              }
```

```
                        if (n % 2 !=0)
                                p[distinct-1].r = -p[distinct-1].r;
                }
        (void) fclose( file_ptr );
    }

        matlab = new double[N];
        for(i = n-1; i>=0; i--)
                matlab[i] = p[i].r;
}


// write the roots result to a file

void Polynomial :: write_to_file( )
{
  char   sig0,sig1,             // sign of real part
         sig2,sig3, sig4;       // sign of imaginary part
  int i,k=0;
  int count=0;
  int out_uc=0;
  int in_uc=0;
  int on_uc=0;
  int on_axis=0;
  int on_one =0;
  double *h_0, *h_1, *h_2, *h_3, *h_4;
  double *temp_0, *temp_1,*temp_2,*temp_3,*temp_4;
  FILE   *file_ptr;

  Complex *inside;
  Complex *outside;
  Complex *onunitcircle;
  Complex *onaxis;
  Complex *onone;
  Complex Hm, temp;
  Hm.r =1.0; Hm.i = 1.0;

// allocate the memory

 inside = new Complex  [ n ];
 outside= new Complex  [ n ];
 onunitcircle = new Complex  [ n ];
 onaxis = new Complex [n];
 onone  = new Complex [n];
        h_0 = new double[n];
```

```
        h_1 = new double[n];
        h_2 = new double[n];
        h_3 = new double[n];
        h_4 = new double[n];
        temp_0 = new double[n];
        temp_1 = new double[n];
        temp_2 = new double[n];
        temp_3 = new double[n];
        temp_4 = new double[n];

        for(i=0; i<n; i++){
                h_0[i]=1.0;
                h_1[i]=1.0;
                h_2[i]=1.0;
                h_3[i]=0.0;
                h_4[i]=0.0;
        }

// generate output file

   file_ptr = fopen( "roots_FIR.dat", "w" );
   fprintf(file_ptr,"  %6d \n", n );
   for ( i = 0; i < n; i++ ) {
     sig1 = (root[i].r >= 0) ? ' ' : '-';
     sig2 = (root[i].i >= 0) ? ' ' : '-';
     fprintf( file_ptr, " %c%.18e     %c%.18e\n",
             sig1, fabs(root[i].r), sig2, fabs(root[i].i) );
   }
   (void) fclose( file_ptr );

   file_ptr = fopen( "roots_FIR.txt", "w" );
   fprintf(file_ptr, "                Decomposition of FIR Filter\n");
   fprintf(file_ptr, "                  Roots of the FIR Filter\n");
   fprintf(file_ptr, "                   Filter Order = %d\n\n", n);
   fprintf(file_ptr, "        Real Part                  Imaginary Part\n");

  for ( i = 0; i < n; i++ ) {
     sig1 = (root[i].r >= 0) ? ' ' : '-';
     sig2 = (root[i].i >= 0) ? ' ' : '-';
     fprintf( file_ptr, " %c%.18e     %c%.18e\n",
             sig1, fabs(root[i].r), sig2, fabs(root[i].i) );
   }
  (void) fclose( file_ptr );

        for ( i = 0; i < n; i++ ) {
```

```
                if ( sqrt(root[i].r*root[i].r + root[i].i*root[i].i ) < (1-1E-8)  &&
                        (root[i].i > 0)){
                    inside[in_uc].r = root[i].r;
                    inside[in_uc].i = root[i].i;
                    in_uc++;
                }
        }

// save zeroes outside the unit circle. (Above - Half)

        for ( i = 0; i < n; i++ ) {
                if ( sqrt(root[i].r*root[i].r + root[i].i*root[i].i )-1 > (1E-8) &&
       (root[i].i > 0)){
                    outside[out_uc].r = root[i].r;
                    outside[out_uc].i = root[i].i;
                    out_uc++;
                }
        }

// save zeroes on the unit circle.

        for ( i = 0; i < n; i++ ) {
                if ( sqrt(root[i].r*root[i].r + root[i].i*root[i].i) < (1+1E-8) &&
                     sqrt(root[i].r*root[i].r + root[i].i*root[i].i) >(1-1E-8)
                        && (fabs(root[i].i)!=0)){
                        onunitcircle[on_uc].r = root[i].r;
                        onunitcircle[on_uc].i = root[i].i;
                        on_uc++; }
        }

// save zeroes on the Real Axis.(All )

        for ( i = 0; i < n; i++ ) {
                if( fabs(root[i].i)==0  && ( (fabs(root[i].r)>1+1E-8)||
                    ( fabs(root[i].r)<1-1E-8) )){
                    onaxis[on_axis].r = root[i].r;
                    onaxis[on_axis].i = root[i].i;
                    on_axis++;
                }
        }

// save zeroes on the Real Axis are +1 or -1.

        for ( i = 0; i < n; i++ ) {
                if (  ( (fabs(root[i].i)==0) && (fabs(root[i].r)<1+1E-8) &&
```

```
                (fabs(root[i].r)>1-1E-8) )){
                    onone[on_one].r = root[i].r;
                    onone[on_one].i = root[i].i;
                    on_one++;
                }
        }

// get the subfilter frequency response

        file_ptr = fopen( "FIR_subfilter_response.dat", "w" );
        for (i=0; i<on_one; i++){
                if( (onone[i].r-1)>-1E-8 && (onone[i].r-1)< 1E-8 ){
                        h_1[i] = -1;
                }
                temp_0[count] = h_0[i];
                temp_1[count] = h_1[i];
                temp_2[count] = temp_3[count] = temp_4[count] = 0.0;
                count++;
        }
        for (i=0; i<on_axis; i++){
                if ( fabs(onaxis[i].r)>1 )
                        k++;
                else if (fabs(onaxis[i].r)<1){
                    if(onaxis[i].r!=0 || onaxis[i].i!=0) {
                        h_1[i+on_one-k] = -(onaxis[i].r + 1/onaxis[i].r);
                        temp_0[count] = h_0[i+on_one-k];
                        temp_1[count] = h_1[i+on_one-k];
                        temp_2[count] = h_2[i+on_one];
                        temp_3[count] = temp_4[count] = 0.0;
                        count++;
                    }
                }
        }
        for (i=0; i<on_uc; i++){
                if (onunitcircle[i].i<0)
                        k++;
                else if (onunitcircle[i].i>0){
                        h_1[i+on_one+on_axis-k] =
                                -2*onunitcircle[i].r/sqrt(onunitcircle[i].r
                                *onunitcircle[i].r + onunitcircle[i].i*onunitcircle[i].i);
                        temp_0[count] = h_0[i+on_one+on_axis-k];
                        temp_1[count] = h_1[i+on_one+on_axis-k];
                        temp_2[count] = h_2[i+on_one+on_axis];
                        temp_3[count] = temp_4[count] = 0.0;
                        count++;}}
```

```
        for (i=0; i<in_uc; i++){
                double r;
                r = sqrt(inside[i].r*inside[i].r + inside[i].i*inside[i].i);
                h_2[i+on_one+on_axis+on_uc -k] = r*r+1/(r*r) +
                     4*inside[i].r*inside[i].r/(inside[i].r*inside[i].r + inside[i].i*inside[i].i);
                h_1[i+on_one+on_axis+on_uc -k] = -2*(r+1/r)*inside[i].r/sqrt(inside[i].r*
                     inside[i].r + inside[i].i*inside[i].i);
                temp_0[count] = h_0[i+on_one+on_axis+on_uc -k];
                temp_1[count] = h_1[i+on_one+on_axis+on_uc -k];
                temp_2[count] = h_2[i+on_one+on_axis+on_uc -k];
                temp_3[count] = h_1[i+on_one+on_axis+on_uc -k];
                temp_4[count] = 1;
                count++;
        }
        fprintf(file_ptr, "%d\n", count);
        fprintf(file_ptr, "5\n");
        for (i=0; i<count; i++){
                fprintf(file_ptr, "%.8f  %.8f  %.8f  %.8f  %.8f\n", temp_0[i],temp_1[i],
                temp_2[i], temp_3[i],temp_4[i]);
        }
        (void) fclose( file_ptr );

// save to human readable file

        file_ptr = fopen( "FIR_subfilter_response.txt", "w" );
        fprintf(file_ptr,"              The Subfilters of FIR Filter\n\n");
        fprintf(file_ptr, "            The Number of the Subfilters = %d\n", count);
        fprintf(file_ptr, "   The Number of impulse response coefficients per subfilter =
                           5\n\n");
        fprintf(file_ptr," m      hm(0)       hm(1)       hm(2)       hm(3)       hm(4)\n");
        fprintf(file_ptr,"=============================================
                          ====================\n");
        for (i=0; i<count; i++){
                sig0 = (temp_0[i] >= 0) ? ' ' : '-';
                sig1 = (temp_1[i] >= 0) ? ' ' : '-';
                sig2 = (temp_2[i] >= 0) ? ' ' : '-';
                sig3 = (temp_3[i] >= 0) ? ' ' : '-';
                sig4 = (temp_4[i] >= 0) ? ' ' : '-';
               fprintf(file_ptr, "%3d  %c%.8f  %c%.8f  %c%.8f  %c%.8f  %c%.8f\n",
                        i+1,
                        sig0,fabs(temp_0[i]),sig1,fabs(temp_1[i]),sig2,fabs(temp_2[i]),
                        sig3, fabs(temp_3[i]),sig4, fabs(temp_4[i]));
        }
        (void) fclose( file_ptr );
    delete [] inside;
```

```
   delete [] outside;
   delete [] onunitcircle;
   delete [] onaxis;
   delete [] onone;
   delete [] temp_0;
   delete [] temp_1;
   delete [] temp_2;
   delete [] temp_3;
   delete [] temp_4;
   delete [] h_0;
   delete [] h_1;
   delete [] h_2;
   delete [] h_3;
   delete [] h_4;
}


// write the result data file name to monitor screen

void Polynomial :: write_to_screen( )
{
        printf("\n\nProgram finished. The results were saved in the follow files:\n\n");
        printf("roots_FIR.dat              --------- Roots ang Gain of the FIR filter\n");
        printf("roots_FIR.txt              --------- readable text type file\n");
        printf("FIR_subfilter_response.dat   --------- subfilters and Gain\n");
        printf("FIR_subfilter_response.txt   --------- readable text type file\n\n");
}


// write error message

void Polynomial :: write_error_message( unsigned char error )
{
   printf( "Error %d occured!\n", (int) error );
   switch ( error ) {
     case 1:
          printf( "Power of polynomial lower null!\n" );
          break;
     case 2:
          printf( "Polynomial is a null vector!\n" );
          break;
     case 3:
          printf( "Polynomial is a constant unequal null!\n" );
          break;
   }
```

```
}


// get the roots we want

unsigned char Polynomial :: get_roots()
{
   const  double DBL_EPSILON = 2.2204460492503131E-16;
   Complex  ns;                  // root determined by Muller's method
   int      i;                   // counter
   double newerr;
   unsigned char error; // indicates an error in poly_check
   int      red,
            diff;                // number of roots at 0
            n-=1;
   nred = n;                     // At the beginning: degree defl. polyn. =
                                 // degree of original polyn.
   maxerr=0.;

//check input of the polynomial and make some changes if there are "0" in inputs

    error = poly_check();
    diff  = (n-nred);   // reduce polynomial, if roots at 0
    p    += diff;       // the pointer should change
    n     = nred;

// some errors such like all inputs are Null or "0"

    if (error)
        return error;

// speical case,polynomial is linear or quadratic,
//such like ax+b=0 or ax^2 + bx + c=0
// we can  find the result directly and don't need to use Muller & Newton Method

    if (lin_or_quad(p)==0) {
        n += diff;     // remember roots at 0
        maxerr = DBL_EPSILON;
        return 0;
    }

  monic();

// Prepare for the input of Muller
```

```
    for (i=0;i<=n;i++)
            pred[i]=p[i];
    do {

// Muller method

          ns = muller();

// Newton method

           root[nred-1] = newton(ns,&newerr);
          if (newerr>maxerr)
                    maxerr = newerr;

         red = poldef(flag);
         pred += red;       // forget lowest coefficients
         nred -= red;       // reduce degree of polynomial

    } while (nred>2);
                             // last one or two roots
    (void) lin_or_quad(pred);

    if (nred==2) {
                    if(Cabs(root[1])<=1){
                          root[1] = newton(root[1],&newerr);
                          if (newerr>maxerr)
                                    maxerr = newerr;
                    }
    }
  if(Cabs(root[0])<=1)
       root[0] = newton(root[0],&newerr);
            n += diff;              // remember roots at 0

            if (maxerr < 9e-5){
                    printf("\n...\n");
                    return 0;
            }
            else{
               printf(" Root finding  failed, program will exit ...\n");
               exit(0);
            }
    return 0;
}
```

```
// monic() computes monic polynomial for original polynomial

void Polynomial ::monic()

{
    double factor;          // stores absolute value of the coefficient
                            // with highest exponent
    int    i;               // counter variable

    factor=1./Cabs(p[n]);     // factor = |1/pn|
    if ( factor!=1.)          // get monic pol., when |pn| != 1
    for (i=0;i<=n;i++)
            p[i] *= factor;
}


// poly_check() check the formal correctness of input

unsigned char Polynomial :: poly_check()
{
    int  i = -1,
         j;
    unsigned char notfound=1;

//degree of polynomial less than zero,return error

    if (n<0) return 1;

// ex. sometimes the degree is 5, but the polynomial is "0,0,3,4,2",
// so its degree shoule be 3

    for (j=0;j<=n;j++) {
        if(Cabs(p[j])!=0.)
            i=j;
    }

// olynomial is a null

    if (i==-1) return 2;

// polynomials are all "0"

    if (i==0) return 3;

// get new exponent of polynomial
```

```
    n=i;
    i=0;

// i --> how many "0" in the input exponent polynomial

 do {
        if (Cabs(p[i])==0.)
            i++;
        else
            notfound=0; //FALSE
    } while (i<=n && notfound);

        if (i==0) {           // no '0',original degree=deflated degree
            nred = n;
            return 0;
        } else {              // there are '0',  store roots at 0
            for (j=0;j<=i-1;j++)
            root[n-j-1] = Complex(0.,0.);
            nred = n-i;      // reduce degree of deflated polynomial
            return 0;
        }
}

// calculates the roots of a quadratic polynomial ax^2+bx+c=0

void Polynomial :: quadratic(Complex *p)
{
    Complex  discr,       // discriminate
             Z1,Z2,       // numerators of the quadratic formula
             N;           // denominator of the quadratic formula

                          // discr = p1^2-4*p2*p0
          discr=p[1]*p[1]-4*p[2]*p[0];
                          // Z1 = -p1+sqrt(discr)
          Z1=-p[1]+Csqrt(discr);
                          // Z2 = -p1-sqrt(discr)
          Z2=-p[1]-Csqrt(discr);
                          // N  = 2*p2
          N=2*p[2];
          root[0]=Z1/N;
          root[1]=Z2/N;
}
```

```
// lin_or_quad() calculates roots of lin. or quadratic equation

unsigned char Polynomial :: lin_or_quad(Complex *p)
{
    if (nred==1) {                      // root = -p0/p1
                root[0]=-p[0]/p[1];
                return 0;               // and return no error
    } else if (nred==2) {               // quadratic polynomial
                quadratic(p);
                return 0;                       // return no error
    }
    return 1;                           // nred>2 => no roots were calculated
}


// Horner method to deflate one root

void Polynomial :: hornc(Complex x0, unsigned char flag)
{
    int     i;
    Complex help1;              // help variable

    if ((flag&1)==0)            // real coefficients
        for(i=nred-1; i>0; i--)
            pred[i].r += (x0.r*pred[i+1].r);
    else                        // complex coefficients
        for (i=nred-1; i>0; i--) {
            CMUL(help1,pred[i+1],x0);
            CADD(pred[i],help1,pred[i]);
        }
}


// Horner method to deflate two roots

void Polynomial :: horncd(double a,double b)
{
    int i;
    pred[nred-1].r += pred[nred].r*a;
    for (i=nred-2; i>1; i--)
        pred[i].r += (a*pred[i+1].r+b*pred[i+2].r);
}


// main routine to deflate polynomial
```

```
int Polynomial :: poldef(unsigned char flag)
{
    double   a,
             b;
    Complex x0;                    // root to be deflated
    x0 = root[nred-1];
    if (x0.i!=0.)                  // x0 is complex
        flag |=2;

    if (flag==2) {                 // real coefficients and complex root
        a = 2*x0.r;                // => deflate x0 and Conjg(x0)
        b = -(x0.r*x0.r+x0.i*x0.i);
        root[nred-2]=Conjg(x0);// store second root = Conjg(x0)
        horncd(a,b);
        return 2;                  // two roots deflated
    } else {
        hornc(x0,flag);            // deflate only one root
        return 1;
    }
}


// Newton's method

Complex Polynomial::newton(Complex ns,double *dxabs)
{
  const  int    ITERMAX_1 = 20 ;
  const  double DBL_EPSILON = 2.2204460492503131E-16;
  const  double BOUND= sqrt(DBL_EPSILON);

// if the imaginary part of the root is smaller than BOUND => real root

  const  int NOISEMAX =5;
  const  int FACTOR =5;
  const  double FVALUE = 1E36;
  double fabsmin=FVALUE,
           eps = DBL_EPSILON;

  Complex    x0,             // iteration variable for x-value
             xmin,           // best x determined in newton()
               f,            // P(x0)
              df,            // P'(x0)
              dx,            // P(x0)/P'(x0)
         dxh;            // temperary variable dxh = P(x0)/P'(x0)
```

```
    int     noise  =0;
    x0   = ns;                  // initial estimation = from Muller method
    xmin = x0;                  // initial estimation for the best x-value
    dx   = Complex(1.,0.);      // initial value: P(x0)/P'(x0)=1+j*0
    *dxabs = Cabs(dx);

    for (iter=0;iter<ITERMAX_1;iter++) {
                f_value1(p,&f,&df,x0); // f=P(x0), df=P'(x0)
                 if(Cabs(f)<fabsmin){
                        xmin=x0;
                        fabsmin = Cabs(f);
                        noise =0;
                 }
                 if (Cabs(df)!=0.) {      // calculate new dx
                        dxh=f/df;
                        if (Cabs(dxh) < *dxabs * FACTOR){
                                dx=dxh;
                                *dxabs = Cabs(dx);
                          }
      }
       if (Cabs(xmin)!=0.) {
            if(*dxabs/Cabs(xmin)<eps || noise==NOISEMAX){
              if (fabs(xmin.i)<BOUND && flag==0) {
                      xmin.i=0.;   // if imag. part<BOUND, let's it=0
                       }
             *dxabs = *dxabs/Cabs(xmin);
             return xmin;             // return best approximation
          }
       }

// x0 = x0 - P(x0)/P'(x0)

      x0-=dx;
      noise++;
    }
    if (fabs(xmin.i)<BOUND && flag==0)
        xmin.i=0.;

//if imag. part<BOUND , let's it=0

    if(Cabs(xmin)!=0.)
          *dxabs=*dxabs/Cabs(xmin);
    return xmin;              // return best xmin until now
}
```

```
void Polynomial :: f_value1(Complex *p,Complex *f,Complex *df,Complex x0)
{
    int      i;                    // counter
    Complex help1;                 // temperary variable
    *f  = p[n];

    COMPLEXM(*df,0.,0.);
    for (i=n-1; i>=0; i--) {
            *df = (*df) * x0 + (*f);
            *f  = (*f)  * x0 + p[i];
        }
}


void Polynomial :: f_value2(Complex *f,Complex *df,Complex x0)
{
    int      i;                    // counter
    Complex help1;                 // temperary variable
    *f  = psave[nred];

    COMPLEXM(*df,0.,0.);
    for (i=nred-1; i>=0; i--) {
            *df=(*df)*x0 + (*f);
            *f= (*f)*x0 +psave[i];
        }
}


// Muller's method

Complex Polynomial :: muller()
{
          const int    ITERMAX =  150;   // max. number of iteration steps
          const double FVALUE = 1e36;    // initialisation of |P(x)|^2
          const double DBL_EPSILON =2.2204460492503131E-16;
          const double NOISESTART = DBL_EPSILON*1e2;
          const int    NOISEMAX = 5;
          double   h2abs,             // h2abs=|h2| h2absnew=distance between old and new x2
                   f1absq,            // f1absq=|f1|^2       used for check
                   f2absq=FVALUE,     // f2absq=|f2|^2       used for check
                   f2absqb=FVALUE,    // f2absqb=|P(xb)|^2   used for check
             epsilon;
  int      seconditer=0,             // second iteration, when root is too bad
```

```
  noise=0,                    // noise counter
  rootd=0;
  Complex xb;                 // best x-value
  initialize(&xb,&epsilon);   // initialize x0,x1,x2,h1,h2,q2,*xb

//use Horner's Method, get f0=P(x0), f1=P(x1), f2=P(x2)

    f_value(nred,&f0,x0);
    f_value(nred,&f1,x1);
    f_value(nred,&f2,x2);
do {
    do {

// get q2  (     q2=2C/B(+-)sqr(B^2-4AC)   )

        root_of_parabola();

// store values for the next iteration

        x0 = x1;
        x1 = x2;
        h2abs = Cabs(h2);   // |x2-x1|

// get the result from Muller's method:  x2=x2-(x2-x1) * 2C/B(+-)sqr(B^2-4AC)

        iteration_equation(&h2abs);

// store P(x) values for the next iteration

        f0 = f1;
        f1 = f2;
        f1absq = f2absq;
        compute_function(f1absq,&f2absq,epsilon);

// check if the new x2 is best enough , these two checks are necessary

        check_x_value(&xb,&f2absqb,&rootd,f1absq,f2absq,epsilon,&noise);

        if (fabs((Cabs(xb)-Cabs(x2))/Cabs(xb))<NOISESTART)
           noise++;
    } while ((iter<=ITERMAX)&& (!rootd)&& (noise<=NOISEMAX));
    seconditer++;

    root_check(f2absqb,&seconditer,&rootd,&noise,xb);
```

```
} while (seconditer==2);
    return xb;              // return best x value
}


// initializing routine

void Polynomial :: initialize( Complex *xb, double *epsilon)
{
  const double DBL_EPSILON =2.2204460492503131E-16;
  x0 = Complex(0.,1.);                          // x0 = 0 + j*1
  x1 = Complex(0.,-1.);                         // x1 = 0 - j*0
  x2 = Complex(1./sqrt(2),1./sqrt(2));      // x2 = (1 + j*1)/sqrt(2)
  h1=x1-x0;
  h2=x2-x1;                             // h2 = x2 - x1
  q2=h2/h1;                             // q2 = h2/h1
  *xb     = x2;                         // best initial x-value = zero
  *epsilon = 5*DBL_EPSILON;
  iter    = 0;                          // reset iteration counter
}


// root of Muller's parabola------q2

void Polynomial :: root_of_parabola(void)
{
   Complex A2,B2,C2,
               discr,
               N1,N2;
   const double DBL_EPSILON = 2.2204460492503131E-16;

// A2 = q2(f2 - (1+q2)f1 + f0q2)
// B2 = q2[q2(f0-f1) + 2(f2-f1)] + (f2-f1)
// C2 = (1+q2)f[2]

   A2=q2*(f2-(1+q2)*f1 +f0*q2);
   B2=q2*(q2*(f0-f1) + 2*(f2-f1) )  + (f2-f1);
   C2=(1+q2)*f2;

// discr = B2^2 - 4A2C2

    discr=B2*B2 - 4*A2*C2;

// denominators of q2
```

```
            N1=B2-Csqrt(discr);
            N2=B2+Csqrt(discr);

// choose denominater with largest modulus

    if (Cabs(N1)>Cabs(N2) && Cabs(N1)>DBL_EPSILON)
        q2=(-2)*C2/N1;
    else if (Cabs(N2)>DBL_EPSILON)
        q2=(-2)*C2/N2;
    else
        q2 = Complex(cos(iter),sin(iter));
}


// main iteration equation: x2 = h2*q2 + x2

void Polynomial :: iteration_equation(double *h2abs)

{
    double h2absnew,                // Absolute value of the new h2
           help;                    // help variable
    const double MAXDIST = 1e3;
    h2 *= q2;
    h2absnew = Cabs(h2);            // distance between old and new x2

    if (h2absnew > (*h2abs*MAXDIST)) { // maximum relative change
        help = MAXDIST/h2absnew;
        h2 *= help;
        q2 *= help;
    }
    *h2abs = h2absnew;              // actualize old distance for next iteration
    x2 += h2;
}


// use Horner's method to get P(x)

void Polynomial :: f_value(int n,Complex *f,Complex x0)
{
    int      i;
    Complex help1;
    *f  = pred[n];

// compute  P(x0)
```

```
        for (i=n-1; i>=0; i--) {

// use Horner's method

            CMUL(help1,*f,x0);   // *f = p[i] + *f * x0
            CADD(*f,help1,pred[i]);
        }
}


// check of too big function values

void Polynomial :: too_big_functionvalues(double *f2absq)
{
        const double DBL_MAX = 1.7976931348623157E+308;
        const double BOUND4 = sqrt(DBL_MAX)/1e4;
        if ((fabs(f2.r)+fabs(f2.i))>BOUND4)        // limit |f2|^2
                *f2absq = fabs(f2.r)+fabs(f2.i);
        else
                *f2absq = (f2.r)*(f2.r)+(f2.i)*(f2.i);
}


void Polynomial::suppress_overflow()
{
    int         kiter;                  // internal iteration counter
    unsigned char loop;                 // loop = FALSE => terminate loop
    double  help;                       // help variable
    const    double KITERMAX = 1e3;
    const    double DBL_MAX = 1.7976931348623157E+308;
    const    double BOUND4 = sqrt(DBL_MAX)/1e4;
    const    double BOUND6 = log10(BOUND4)-4;
    kiter = 0;                          // reset iteration counter
    do {
        loop=0;                         // initial estimation: no overflow
        help = Cabs(x2);                // help = |x2|
        if (help>1. && fabs(nred*log10(help))>BOUND6) {
            kiter++;                    // if |x2|>1 and |x2|^nred>10^BOUND6
            if (kiter<KITERMAX) { // then halve the distance between
                h2=.5*h2;               // new and old x2
                q2=.5*q2;
                x2=x2-h2;
                loop=1;
            } else
                kiter=0;
```

```
        }
    } while(loop);
}



// Muller's modification to improve convergence

void Polynomial::convergence_check(int *overflow,double f1absq,double
            f2absq,double epsilon)
{
   const int CONVERGENCE = 100;
   const int ITERMAX = 150;
   if ((f2absq>(CONVERGENCE*f1absq)) && (Cabs(q2)>epsilon) &&
    (iter<ITERMAX)) {
        q2 *= .5;               // in case of overflow:
        h2 *= .5;               // halve q2 and h2; compute new x2
        x2 -= h2;
        *overflow = 1;
    }
}


// compute P(x2) and make some checks

void  Polynomial ::compute_function(double f1absq,double *f2absq,double epsilon)
{
    int    overflow;      // overflow = TRUE  => overflow occures
                          // overflow = FALSE => no overflow occures

    do {
       overflow =0;    // initial estimation: no overflow

                          // suppress overflow
       suppress_overflow();

                          // calculate new value => result in f2
       f_value(nred,&f2,x2);

                          // check of too big function values
       too_big_functionvalues(f2absq);

                          // increase iterationcounter
       iter++;
```

```
                                // Muller's modification to improve convergence
        convergence_check(&overflow,f1absq,*f2absq,epsilon);
    } while (overflow);
}


// check if the new x2 the best approximation

void Polynomial :: check_x_value(Complex *xb,double *f2absqb,int *rootd,
                double f1absq,double f2absq,double epsilon,int *noise)
{
     const double BOUND1 = 1.01;
     const double BOUND2 = 0.99;
     const double BOUND3 = 0.01;
     if ((f2absq<=(BOUND1*f1absq)) && (f2absq>=(BOUND2*f1absq))) {
                                // function-value changes slowly
        if (Cabs(h2)<BOUND3) {  // if |h[2]| is small enough =>
           q2 *= 2;             // double q2 and h[2]
           h2 *= 2;
        } else {                // otherwise: |q2| = 1 and
                                // h[2] = h[2]*q2
                q2 = Complex(cos(iter),sin(iter));
                h2 = h2*q2;
        }
    } else if (f2absq<*f2absqb) {
        *f2absqb = f2absq;      // the new function value is the
        *xb      = x2;          // best approximation
        *noise   = 0;           // reset noise counter
        if ((sqrt(f2absq)<epsilon) && (Cabs((x2-x1)/x2))<epsilon)
            *rootd = 1;
     }
}


void Polynomial ::root_check(double f2absqb,int *seconditer,
             int *rootd,int *noise,Complex xb)
{
    Complex df;                     // df=P'(x0)
    const double BOUND7 = 1e-5;
    if ((*seconditer==1) && (f2absqb>0)) {
        f_value2(&f2,&df,xb);         // f2=P(x0), df=P'(x0)
       if (Cabs(f2)/(Cabs(df)*Cabs(xb))>BOUND7) {

// start second iteration with new initial estimations
```

```
            x0 = Complex(-1./sqrt(2),1./sqrt(2));
            x1 = Complex(1./sqrt(2),-1./sqrt(2));
            x2 = Complex(-1./sqrt(2),-1./sqrt(2));

            f_value(nred,&f0,x0);

            f_value(nred,&f1,x1);

            f_value(nred,&f2,x2);
            iter = 0;               // reset iteration counter
            (*seconditer)++;        // increase seconditer
            *rootd = 0;             // no root determined
            *noise = 0;             // reset noise counter
          }
      }
}


void Polynomial ::check_value()
{
  int k, m, i;
  double omega, W_Re, W_Im, T_Re, T_Im,  H_mag, upper, lower, G;
  double s_H_mag;
  double s_T_Re, s_T_Im, s_H_Re, s_H_Im, s_temp;
  Complex H, temp;
  double *test2;
  double *h;
  test2=new double[NPLOT];
  FILE *file_ptr;
  H.r = 1;H.i =0;

  upper=lower=0.0;
  h = new double [N+1];

  for (i=N-1; i>=0; i--)
          h[i] = matlab[i];

  for ( k = 0; k < NPLOT; k++ ) {
      omega = PI * ((double) k) / ((double) NPLOT);
      W_Re =   cos( omega );
      W_Im = - sin( omega );
      T_Re =   cos( 2*omega );
      T_Im = - sin( 2*omega );
      s_T_Re = 1.0;
      s_T_Im =0.0;
```

```
s_H_Re = h[0];
s_H_Im = 0.0;

for ( m = 0; m < N-1 ; m++ ) {
        if (root[m].i == 0.0){
                temp.r = 1-root[m].r*W_Re;
                temp.i = root[m].r*W_Im;
         }
        else{
                temp.r = 1-(2*root[m].r*W_Re)+
                          (root[m].r*root[m].r+(root[m].i*root[m].i))*T_Re;
                 temp.i = (root[m].r * root[m].r + (root[m].i * root[m].i))*T_Im
                           -2*root[m].r*W_Im;
                   m++;
             }
                 H*=temp;
         }

        for(i=1; i<N; i++){
                 s_temp = s_T_Re * W_Re - s_T_Im * W_Im;
                 s_T_Im = s_T_Re * W_Im + s_T_Im * W_Re;
                 s_T_Re = s_temp;
                 if ( i<distinct){
                          s_temp = h[i];
                 }
                 else{
                          s_temp = h[N -1 -i];
                          if(indicator == 1) s_temp = -s_temp;
                 }
                 s_H_Re += s_temp * s_T_Re;
                 s_H_Im += s_temp * s_T_Im;
         }
         s_H_mag = sqrt(s_H_Re*s_H_Re + s_H_Im*s_H_Im);

         H_mag = sqrt( H.r * H.r + H.i * H.i );
         H.r = 1; H.i =0 ;

         upper += s_H_mag *H_mag;
         lower += H_mag*H_mag;
         test2[k] = s_H_mag;
        }
    G = upper/lower;

    file_ptr = fopen("roots_FIR.dat","a");
    fprintf(file_ptr,"%.10f\n", G);
```

```
        fclose( file_ptr );

        file_ptr = fopen("FIR_subfilter_response.dat","a");
        fprintf(file_ptr,"%.10f\n", G);
        fclose( file_ptr );

        file_ptr = fopen("roots_FIR.txt","a");
        fprintf(file_ptr,"Gain = %.10f\n", G);
        fclose( file_ptr );

        file_ptr = fopen("FIR_subfilter_response.txt","a");
        fprintf(file_ptr,"Gain = %.10f\n", G);
        fclose( file_ptr );

        delete [] matlab;
        delete [] h;
        delete [] test2;
}

Polynomial :: ~Polynomial()
{
        delete [] psave1;
        delete [] root;
        delete [] psave;
}

char main(void)
{
    Polynomial Poly;
    unsigned char error;
    Poly.read_from_file( );
    error=Poly.get_roots();
    if ( !error ) {
                    Poly.write_to_file();
                    Poly.write_to_screen();
                    Poly.check_value();
          }
    else {
                    Poly.write_error_message( error );
                    return 1;
          }
    return 0;
}
```

APPENDIX B

C++ PROGRAM FOR MINIMUM-PHASE FILTER DESIGN

```
/*******************************************************/
/*******************************************************/
/*                                                     */
/*              Minimum-Phase Filter Design            */
/*                                                     */
/*******************************************************/
/*******************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iostream.h>
#include <conio.h>
#include "complex.cpp"
#define NPLOT 2048
#define PI 3.14159265358979363846

RootCoeff(double G)
{
        int i,j,N_roots, ind, L;
        double x,y;
        Complex dum1;
        Complex *root, *coeff;
        Complex  **a, *dum2, *x_out, *aaa;
        FILE * file_ptr;

        dum1 = Cmplx(0.0, 0.0);

        file_ptr=  fopen("roots_minimum.dat", "r");
        fscanf(file_ptr, "%d", &N_roots);

        root = new Complex [N_roots+2];
        dum2 = new Complex [N_roots+2];
        x_out= new Complex [N_roots+2];
        aaa  = new Complex [N_roots+2];

        a= new Complex *[N_roots+3];
        for(j=1; j<=N_roots+2; j++)    {
            a[j]=new Complex[N_roots+2];
          }

        for(i=1; i<=N_roots; i++){
                fscanf(file_ptr, "%le", &x);
                fscanf(file_ptr, "%le", &y);
```

```
            root[i]=Cmplx(x,y);
 }
 root[N_roots+1]=root[N_roots];
 fclose(file_ptr);

 for (i=1; i<= N_roots+1; i++){
      for(j=1; j<=N_roots; j++){
           a[i][j] = root[j];
     }
}

 for(j=1; j<=N_roots; j++)
         a[1][j] = Cabs(a[1][j]);

 for(i=1; i<=N_roots; i++){
     if(Cabs(dum1)<Cabs(root[i])){
           dum1= root[i];
           ind = i;
     }
 }

 for(i=1; i<=N_roots+1; i++){
      dum2[i]=a[i][1];
 }

 for(i=1; i<=N_roots+1; i++){
      a[i][1]=a[i][ind];
 }
 for(i=1; i<=N_roots+1; i++){
     a[i][ind]=dum2[i];
}

 x_out[1] =  a[N_roots][ind];

for(j=2; j<=N_roots;j++)
      a[2][j] = Cabs( a[2][j]-x_out[1] );

dum1 = Cmplx(0.0, 0.0);              // initialize dum=0, aaa=1
for(i=0;i<N_roots; i++){
   aaa[i].r =1.0;
   aaa[i].i =0.0;
}

for(L=2; L<=N_roots-1; L++){
   dum1.r=0;
```

```
            dum1.i=0;
            for(j=L; j<=N_roots; j++){
                        aaa[j].r=1; aaa[j].i=0;
                        for(i=1; i<=L; i++){
                                    aaa[j] *= a[i][j];
                        }
            }

  for (j=L; j<=N_roots; j++){
              if(Cabs(dum1) < Cabs(aaa[j]) ){
                                    dum1 = aaa[j];
                                    ind = j-L+1;
                        }
  }

  ind = ind +L-1;

  if (L!=ind){
            for(i=1; i<=N_roots+1; i++){
                        dum2[i] = a[i][L];
                        a[i][L] = a[i][ind];
                        a[i][ind] = dum2[i];
            }
 }

x_out[L] = a[N_roots][L];

for(j=L+1; j<=N_roots; j++){
            a[L+1][j]  = Cabs( a[L+1][j] - x_out[L]  );
 }
 }

  for(j=1; j<=N_roots; j++)
            x_out[j] = a[N_roots+1][j];

  for(i=0; i<N_roots; i++)
           root[i]=x_out[i+1];
           coeff = new Complex[N_roots +1];
           coeff[0] = Cmplx(1.0, 0.0);
           for(j=1; j<=N_roots; j++)
                      coeff[j] = Cmplx(0.0, 0.0);
           for(i =0; i<N_roots; i++){
                      for(j=i+1; j>0; j--){
                                 coeff[j] -= root[i] * coeff[j-1];
                      }
```

```
        }
        delete []root;

        file_ptr = fopen ("coefficient.dat", "w");
        fprintf(file_ptr, "%6d \n", N_roots +1 );

        for (i=0; i< N_roots +1 ; i++){
                fprintf(file_ptr, "%16.8e  \n", G*Re(coeff[i]));
        }
        fclose (file_ptr);

        file_ptr = fopen ("coefficient.txt", "w");
        fprintf(file_ptr, "       Minimum-Phase Filter Design\n\n");
        fprintf(file_ptr, "         Filter Length = %d \n\n", N_roots +1 );
        fprintf(file_ptr, "       Impulse Response Coefficients\n");

        for (i=0; i< N_roots +1 ; i++){
                fprintf(file_ptr, "         h(%3d) = %16.8e  \n", i, G*Re(coeff[i]));
    }
        fclose (file_ptr);
        delete [] coeff;
        delete [] dum2;
        delete [] x_out;
        delete [] aaa;
        for(j=1; j<=N_roots+2; j++)
                delete [] a[j];
        delete []a;
        return (0);
}


class Polynomial
{
public:
        void read_from_file( );
        unsigned char get_roots();
        void write_to_file( );
        void write_to_screen();
        void write_error_message( unsigned char );
        void check_value();
        void draw_magnitude();
        void After_Modified();
        ~Polynomial();

private:
```

```
unsigned char poly_check();
void         quadratic(Complex *);
unsigned char lin_or_quad(Complex *);
void         hornc(Complex,unsigned char );
void         horncd(double ,double );
int          poldef(unsigned char);
void         monic();

// functions used are for Newton's Method

Complex      newton(Complex,double *);
void         f_value1(Complex *,Complex *,Complex *,Complex );
void         f_value2(Complex *,Complex *,Complex );

// functions used are for Muller's Method

Complex   muller();
void         initialize( Complex *,double *);
void         root_of_parabola();
void         iteration_equation(double *);
void         suppress_overflow();
void         too_big_functionvalues( double * );
void         convergence_check(int *,double,double,double);
void         compute_function(double,double* ,double );
void         check_x_value(Complex *,double *,int*,double , double, double,int * );
void         root_check(double,int *,int *,int *,Complex);
void         f_value(int ,Complex *,Complex );
Complex      x0,x1,x2,      // common points [x0,f(x0)=P(x0)], ... [x2,f(x2)]
             f0,f1,f2,      // of parabola and polynomial
             h1,h2,         // distance between x2 and x1
             q2, *psave, *psave1;          // smaller root of parabola
int     iter,
        nred,        // the highest exponent of the deflated polynomial
        n,N;         // original degree of the input
int     distinct,
         indicator,
         N_half_root;

double   data;
double   maxerr;
double   delta;
double   *matlab;
double   *half_root_r, *half_root_i;
static   Complex  *p,      // coefficient vector of polynomial
                  *pred,  // coefficient vector of deflated polynom.
```

```
                        *root;  // vector of determined roots
  static unsigned char flag ;
};

unsigned char Polynomial::flag=1;
Complex *Polynomial::p;
Complex *Polynomial::pred;
Complex *Polynomial::root;

void Polynomial :: read_from_file( )
{
   char filename[32];
   int  i, k;                              // counter
   double H_Re, T_Re, omega;
   FILE *file_ptr;

   delta = 1.0;

// open original filter file and calculate the ripple:

 printf("************************************\n");
 printf("*** Minimum-Phase Filter Design ***\n");
 printf("************************************\n\n");
 printf("enter  file name which includes the coefficients of the original filter:");
 scanf("%s", filename);

 file_ptr = fopen ( filename, "r");
 fscanf( file_ptr, "%d %d %d", &n, &distinct, &indicator);

 if (n%2 ==0){
          printf("The order of the filter should be even!\n");
          exit(0);
 }
 if(indicator != 0){
          printf("The coefficient of the filter should be symmetric!\n");
          exit(0);
 }
 N=n;
 p =    new Complex  [ n + 1 ];
 pred = new Complex  [ n + 1 ];
 root = new Complex  [ n ];
 psave=pred;
 psave1=p;
 flag=0;
```

```
// read coefficients
   for (i=n-1; i>=distinct-1; i--){
                    fscanf(file_ptr,"%lf", &data);
                    p[i].r=data;
                    p[i].i=0.0;
          }
          if (indicator == 0){
                    for ( i=0; i<distinct; i++){
                                        p[i].r = p[n-i-1].r;
                                        p[i].i =0.0;
                    }
          }
          else{
                    for (i=0; i<distinct; i++){
                                        p[i].r = -p[n-i-1].r;
                                        p[i].i = 0.0;
                              }
                    if (n % 2 !=0)
                                        p[distinct-1].r = -p[distinct-1].r;
          }
          fclose (file_ptr);

// calcullate the ripple

          for(k=0; k<NPLOT; k++){
                              omega = PI * ((double) k) / ((double) NPLOT);
                              H_Re = p[n/2].r;
                              for ( i=1; i<=n/2; i++){
                                        T_Re = p[n/2-i].r * cos( omega * i);
                                        H_Re += 2 * T_Re;
                              }
                              if (delta > H_Re)
                                        delta = H_Re;  // H_Re is the magnitude of original filter
                              }
         delta = -delta;
          file_ptr=fopen(filename,"r");

// read degree

    (void) fscanf( file_ptr, "%d %d %d", &n, &distinct, &indicator);

// read coefficients

           for (i=n-1; i>=distinct-1; i--){
                    fscanf(file_ptr,"%lf", &data);
```

```
                p[i].r=data/(1+ delta);
                p[i].i=0.0;
        }
        p[distinct-1] = (data + delta)/(1 + delta);
        if (indicator == 0){
                for ( i=0; i<distinct; i++){
                        p[i].r = p[n-i-1].r;
                        p[i].i =0.0;
                }
        }
        else{
                for (i=0; i<distinct; i++){
                        p[i].r = -p[n-i-1].r;
                        p[i].i = 0.0;
                }
                if (n % 2 !=0)
                        p[distinct-1].r = -p[distinct-1].r;
        }
        (void) fclose( file_ptr );

        matlab = new double[N];
        for(i = n-1; i>=0; i--)
                matlab[i] = p[i].r;
}

void Polynomial :: write_to_file( )
{
        int i,k;
        double *h_0, *h_1, *h_2;
        temple = new Complex [n];
        h_0 = new double[n];
        h_1 = new double[n];
        h_2 = new double[n];
        for(i=0; i<n; i++){
                h_0[i]=1.0;
                h_1[i]=1.0;
                h_2[i]=0.0;
        }
        half_root_r = new double[N];
        half_root_i = new double[N];
        temple[0] = root[0];
        k=1;
        for ( i=1; i<n; i++){
                temple[k] = root[i];
                if (temple[k-1].r == root[i].r )
```

```
                        k--;
                k++;
        }
        for(i=0; i<k; i++){
                half_root_r[i] = temple[i].r;
                half_root_i[i] = temple[i].i;
        }
        N_half_root = k;
        delete [] temple;
}


// write message to monitor screen

void Polynomial :: write_to_screen( )
{
        printf("\n\nProgram finished. The results were saved in the follow files:\n\n");
        printf("roots_minimum.dat          \n");
        printf("roots_minimum.txt          --------- Roots of the minimum filter\n\n");
        printf("mini_subfilter_response.dat \n");
        printf("mini_subfilter_response.txt --------- Subfilters and Gain of Mini-phase
                filter\n\n");
        printf("coefficient.dat            \n");
        printf("coefficient.txt            --------- Impulse response of Mini-phase filter\n\n");
}

// write error message

void Polynomial :: write_error_message( unsigned char error )
{
   printf( "Error %d occured!\n", (int) error );
   switch ( error ) {
     case 1:
          printf( "Power of polynomial lower null!\n" );
          break;
     case 2:
          printf( "Polynomial is a null vector!\n" );
          break;
     case 3:
          printf( "Polynomial is a constant unequal null!\n" );
          break;
   }
}

// get the roots we want
```

```
unsigned char Polynomial :: get_roots()
{
    const  double DBL_EPSILON = 2.2204460492503131E-16;
    Complex  ns;                // root determined by Muller's method
    int      i;      int fail = 0;   // counter
    double newerr, data;
    unsigned char error;       // indicates an error in poly_check
    int      red,
             diff;              // number of roots at 0

    n-=1;
    nred = n;                   // At the beginning: degree defl. polyn. =
Reduce:                         // degree of original polyn.
    maxerr=0.;
    error = poly_check();
    diff  = (n-nred);           // reduce polynomial, if roots at 0
    p    += diff;               // the pointer should change
    n     = nred;

// some errors such like all inputs are Null or "0"

    if (error)
        return error;

// speical case,polynomial is linear or quadratic, such like ax+b=0 or ax^2 + bx + c=0
// we can  find the result directly and don't need to use Muller & Newton Method

    if (lin_or_quad(p)==0) {
        n += diff;              // remember roots at 0
       maxerr = DBL_EPSILON;
        return 0;
    }

 monic();

// Prepare for the input of Muller

    for (i=0;i<=n;i++)
                    pred[i]=p[i];
do {

// Muller method

          ns = muller();
```

```
// Newton method

        root[nred-1] = newton(ns,&newerr);

        if (newerr>maxerr)
                maxerr = newerr;

        red = poldef(flag);
        pred += red;        // forget lowest coefficients
        nred -= red;        // reduce degree of polynomial
    } while (nred>2);
                            // last one or two roots
    (void) lin_or_quad(pred);

    if (nred==2) {
                if(Cabs(root[1])<=1){
                        root[1] = newton(root[1],&newerr);
                        if (newerr>maxerr)
                                maxerr = newerr;
                }
    }
    if(Cabs(root[0])<=1)
    root[0] = newton(root[0],&newerr);
         n += diff;             // remember roots at 0

    if (fail == 0 && maxerr < 9e-5){
                printf("\n...\n");
                return 0;
        }

    if (fail == 1 && maxerr < 9e-5){
                printf("\n.....\n");
                return 0;
        }

    else{
                if (fail == 1 && maxerr > 9e-4) {
                        printf("err==%.18e\n", maxerr);
                        printf("Sorry, Roots finding failed, program will exit...\n");
                        exit(0);
                }
                printf("err  ==%.18e\n", maxerr);
                printf("Roots finding is not very good,  wait.....\n");
```

```
                    for(i = N-1; i>=0; i--){
                       data = floor ( matlab[i]*10e8 );
                       data /= 10e8;
                       p[i].r = data;
                    }
               fail = 1;
               goto Reduce;
            }
}

//monic() computes monic polynomial for original polynomial

void Polynomial ::monic()
{
    double factor;   // stores absolute value of the coefficient
                     // with highest exponent
    int    i;        // counter variable

    factor=1./Cabs(p[n]);      // factor = |1/pn|
    if ( factor!=1.)           // get monic pol., when |pn| != 1
       for (i=0;i<=n;i++)
            p[i] *= factor;
}


// poly_check() check the formal correctness of input

unsigned char Polynomial :: poly_check()
{
    int  i = -1,
         j;
    unsigned char notfound=1;

//degree of polynomial less than zero,return error

    if (n<0) return 1;

// ex. sometimes the degree is 5, but the polynomial is "0,0,3,4,2", so its degree is 3

    for (j=0;j<=n;j++) {
        if(Cabs(p[j])!=0.)
            i=j;
    }

// olynomial is a null
```

```
    if (i==-1) return 2;

// polynomials are all "0"

    if (i==0) return 3;

// get new exponent of polynomial

    n=i;
    i=0;

// i --> how many "0" in the input exponent polynomial

          do {
                if (Cabs(p[i])==0.)
                   i++;
                else
                   notfound=0; //FALSE
    } while (i<=n && notfound);

        if (i==0) {           // no '0',original degree=deflated degree
           nred = n;
           return 0;
        } else {              // there are '0',  store roots at 0
           for (j=0;j<=i-1;j++)
                root[n-j-1] = Complex(0.,0.);
           nred = n-i;       // reduce degree of deflated polynomial
           return 0;
        }
}

// calculates the roots of a quadratic polynomial ax^2+bx+c=0

void Polynomial :: quadratic(Complex *p)
{
          Complex  discr,       // discriminate
                     Z1,Z2,     // numerators of the quadratic formula
                       N;       // denominator of the quadratic formula
                                // discr = p1^2-4*p2*p0
          discr=p[1]*p[1]-4*p[2]*p[0];
                                // Z1 = -p1+sqrt(discr)
          Z1=-p[1]+Csqrt(discr);
                                // Z2 = -p1-sqrt(discr)
          Z2=-p[1]-Csqrt(discr);
```

```
        N=2*p[2];
        root[0]=Z1/N;
        root[1]=Z2/N;
}

//lin_or_quad() calculates roots of lin. or quadratic equation

unsigned char Polynomial :: lin_or_quad(Complex *p)
{
        if (nred==1) {                  // root = -p0/p1
                root[0]=-p[0]/p[1];
                return 0;               // and return no error
        } else if (nred==2) {           // quadratic polynomial
                quadratic(p);
                return 0;               // return no error
        }

        return 1;                       // nred>2 => no roots were calculated
}

//Horner method to deflate one root

void Polynomial :: hornc(Complex x0, unsigned char flag)
{
    int     i;
    Complex help1;      // help variable

    if ((flag&1)==0)    // real coefficients
        for(i=nred-1; i>0; i--)
            pred[i].r += (x0.r*pred[i+1].r);
        else                // complex coefficients
            for (i=nred-1; i>0; i--) {
                CMUL(help1,pred[i+1],x0);
                CADD(pred[i],help1,pred[i]);
            }
}

//Horner method to deflate two roots

void Polynomial :: horncd(double a,double b)
{
    int i;
    pred[nred-1].r += pred[nred].r*a;
    for (i=nred-2; i>1; i--)
```

```
        pred[i].r += (a*pred[i+1].r+b*pred[i+2].r);
}

// main routine to deflate polynomial

int Polynomial :: poldef(unsigned char flag)
{
    double   a, b;
    Complex x0;          // root to be deflated
    x0 = root[nred-1];
    if (x0.i!=0.)        // x0 is complex
        flag |=2;

    if (flag==2) {       // real coefficients and complex root
        a = 2*x0.r;      // => deflate x0 and Conjg(x0)
        b = -(x0.r*x0.r+x0.i*x0.i);
        root[nred-2]=Conjg(x0); // store second root = Conjg(x0)
        horncd(a,b);
        return 2;        // two roots deflated
    } else {
        hornc(x0,flag);  // deflate only one root
        return 1;
    }
}

Complex Polynomial::newton(Complex ns,double *dxabs)
{
  const  int    ITERMAX_1 = 20 ;
  const  double DBL_EPSILON = 2.2204460492503131E-16;
  const  double BOUND= sqrt(DBL_EPSILON);

 // if the imaginary part of the root is smaller than BOUND => real root

  const  int NOISEMAX =5;
  const  int FACTOR =5;
  const  double FVALUE = 1E36;
  double fabsmin=FVALUE,
                eps = DBL_EPSILON;

  Complex       x0,            // iteration variable for x-value
                xmin,          // best x determined in newton()
                f,             // P(x0)
                df,            // P'(x0)
                dx,            // P(x0)/P'(x0)
                dxh;           // temperary variable dxh = P(x0)/P'(x0)
```

```
    int     noise  =0;

    x0   = ns;                    // initial estimation = from Muller method
    xmin = x0;                    // initial estimation for the best x-value
    dx   = Complex(1.,0.);        // initial value: P(x0)/P'(x0)=1+j*0
    *dxabs = Cabs(dx);

     for (iter=0;iter<ITERMAX_1;iter++) {
            f_value1(p,&f,&df,x0);   // f=P(x0), df=P'(x0)

            if(Cabs(f)<fabsmin){
                        xmin=x0;
                        fabsmin = Cabs(f);
                        noise =0;
            }

             if (Cabs(df)!=0.) {    // calculate new dx
                        dxh=f/df;
                        if (Cabs(dxh) < *dxabs * FACTOR){
                                dx=dxh;
                                *dxabs = Cabs(dx);
                    }
               }

         if (Cabs(xmin)!=0.) {
              if(*dxabs/Cabs(xmin)<eps || noise==NOISEMAX){
                    if (fabs(xmin.i)<BOUND && flag==0) {
                         xmin.i=0.;   // if imag. part<BOUND, let's it=0
                     }
                     *dxabs = *dxabs/Cabs(xmin);
                     return xmin;     // return best approximation
               }
         }

// x0 = x0 - P(x0)/P'(x0)

         x0-=dx;
         noise++;
    }
    if (fabs(xmin.i)<BOUND && flag==0)
         xmin.i=0.;

//if imag. part<BOUND , lets it equals to zero

 if(Cabs(xmin)!=0.)
```

```
        *dxabs=*dxabs/Cabs(xmin);
return xmin;                        // return best xmin until now
}


void Polynomial :: f_value1(Complex *p,Complex *f,Complex *df,Complex x0)
{
     int      i;                    // counter
    Complex help1;                 // temperary variable
    *f  = p[n];

    COMPLEXM(*df,0.,0.);

    for (i=n-1; i>=0; i--) {
            *df = (*df) * x0 + (*f);
            *f  = (*f)  * x0 + p[i];
    }
}

void Polynomial :: f_value2(Complex *f,Complex *df,Complex x0)
{
    int      i;                    // counter
    Complex help1;                 // temperary variable
    *f  = psave[nred];

    COMPLEXM(*df,0.,0.);
        for (i=nred-1; i>=0; i--) {
            *df=(*df)*x0 + (*f);
            *f= (*f)*x0 +psave[i];
        }
}


Complex Polynomial :: muller()
{
          const int    ITERMAX =  150;      // max. number of iteration steps
          const double FVALUE = 1e36;    // initialisation of |P(x)|^2
          const double DBL_EPSILON =2.2204460492503131E-16;
          const double NOISESTART = DBL_EPSILON*1e2;
          const int    NOISEMAX = 5;
          double  h2abs,             // h2abs=|h2| h2absnew=distance between old and new x2
                  f1absq,            // f1absq=|f1|^2        used for check
                  f2absq=FVALUE,           // f2absq=|f2|^2        used for check
                  f2absqb=FVALUE,          // f2absqb=|P(xb)|^2    used for check
                  epsilon;
```

```
        int     seconditer=0,       // second iteration, when root is too bad
                noise=0,            // noise counter
                rootd=0;
        Complex xb;                 // best x-value
        initialize(&xb,&epsilon);   // initialize x0,x1,x2,h1,h2,q2,*xb

//use Horner's Method, get f0=P(x0), f1=P(x1), f2=P(x2)

        f_value(nred,&f0,x0);
        f_value(nred,&f1,x1);
        f_value(nred,&f2,x2);
do {
   do {

// get q2  (    q2=2C/B(+-)sqr(B^2-4AC)   )

       root_of_parabola();

// store values for the next iteration

        x0 = x1;
        x1 = x2;
        h2abs = Cabs(h2);   // |x2-x1|

// get the result from Muller's method:  x2=x2-(x2-x1) * 2C/B(+-)sqr(B^2-4AC)

        iteration_equation(&h2abs);

// store P(x) values for the next iteration

        f0 = f1;
        f1 = f2;
        f1absq = f2absq;
        compute_function(f1absq,&f2absq,epsilon);

// check if the new x2 is best enough , these two checks are necessary

        check_x_value(&xb,&f2absqb,&rootd,f1absq,f2absq,epsilon,&noise);

        if (fabs((Cabs(xb)-Cabs(x2))/Cabs(xb))<NOISESTART)
                noise++;
   } while ((iter<=ITERMAX)&& (!rootd)&& (noise<=NOISEMAX));
   seconditer++;

   root_check(f2absqb,&seconditer,&rootd,&noise,xb);
```

```
} while (seconditer==2);
    return xb;              // return best x value
}

// initializing routine

void Polynomial :: initialize( Complex *xb, double *epsilon)
{
    const double DBL_EPSILON =2.2204460492503131E-16;
    x0 = Complex(0.,1.);                        // x0 = 0 + j*1
    x1 = Complex(0.,-1.);                       // x1 = 0 - j*0
    x2 = Complex(1./sqrt(2),1./sqrt(2));        // x2 = (1 + j*1)/sqrt(2)
    h1=x1-x0;
    h2=x2-x1;                                   // h2 = x2 - x1
    q2=h2/h1;                                   // q2 = h2/h1
    *xb      = x2;                              // best initial x-value = zero
    *epsilon = 5*DBL_EPSILON;
     iter     = 0;                              // reset iteration counter
}

// root of Muller's parabola------q2

void Polynomial :: root_of_parabola(void)
{
    Complex A2,B2,C2,
                discr,
                N1,N2;
   const double DBL_EPSILON = 2.2204460492503131E-16;

// A2 = q2(f2 - (1+q2)f1 + f0q2)
// B2 = q2[q2(f0-f1) + 2(f2-f1)] + (f2-f1)
// C2 = (1+q2)f[2]

          A2=q2*(f2-(1+q2)*f1 +f0*q2);
           B2=q2*(q2*(f0-f1) + 2*(f2-f1) )  + (f2-f1);
           C2=(1+q2)*f2;

// discr = B2^2 - 4A2C2

           discr=B2*B2 - 4*A2*C2;

// denominators of q2

           N1=B2-Csqrt(discr);
```

```
            N2=B2+Csqrt(discr);

// choose denominater with largest modulus

    if (Cabs(N1)>Cabs(N2) && Cabs(N1)>DBL_EPSILON)
                    q2=(-2)*C2/N1;
    else if (Cabs(N2)>DBL_EPSILON)
                    q2=(-2)*C2/N2;
    else
                    q2 = Complex(cos(iter),sin(iter));
}


// main iteration equation: x2 = h2*q2 + x2

void Polynomial :: iteration_equation(double *h2abs)
{
    double h2absnew,        // Absolute value of the new h2
           help;            // help variable

   const double MAXDIST = 1e3;
    h2 *= q2;
    h2absnew = Cabs(h2);   // distance between old and new x2

    if (h2absnew > (*h2abs*MAXDIST)) { // maximum relative change
        help = MAXDIST/h2absnew;
        h2 *= help;
        q2 *= help;
    }
    *h2abs = h2absnew;      // actualize old distance for next iteration
    x2 += h2;
}

// use Horner's method to get P(x)

void Polynomial :: f_value(int n,Complex *f,Complex x0)
{
    int     i;
    Complex help1;
    *f  = pred[n];

// compute  P(x0)

            for (i=n-1; i>=0; i--) {
```

```
// use Horner's method

                CMUL(help1,*f,x0);   // *f = p[i] + *f * x0
                CADD(*f,help1,pred[i]);
        }
}


// check of too big function values

void Polynomial :: too_big_functionvalues(double *f2absq)
{
        const double DBL_MAX = 1.7976931348623157E+308;
        const double BOUND4 = sqrt(DBL_MAX)/1e4;
        if ((fabs(f2.r)+fabs(f2.i))>BOUND4)        // limit |f2|^2
                *f2absq = fabs(f2.r)+fabs(f2.i);
        else
                *f2absq = (f2.r)*(f2.r)+(f2.i)*(f2.i);
}


void Polynomial::suppress_overflow()
{
    int         kiter;                  // internal iteration counter
    unsigned char loop;                 // loop = FALSE => terminate loop
    double       help;                  // help variable
    const    double KITERMAX = 1e3;
    const    double DBL_MAX = 1.7976931348623157E+308;
    const    double BOUND4 = sqrt(DBL_MAX)/1e4;
    const    double BOUND6 = log10(BOUND4)-4;
    kiter = 0;                          // reset iteration counter
    do {
        loop=0;                         // initial estimation: no overflow
        help = Cabs(x2);                // help = |x2|
        if (help>1. && fabs(nred*log10(help))>BOUND6) {
            kiter++;                    // if |x2|>1 and |x2|^nred>10^BOUND6
            if (kiter<KITERMAX) { // then halve the distance between
                h2=.5*h2;               // new and old x2
                q2=.5*q2;
                x2=x2-h2;
                loop=1;
            } else
                kiter=0;
        }
    } while(loop);
```

```
}



// Muller's modification to improve convergence

void Polynomial::convergence_check(int *overflow,double f1absq,double
f2absq,double epsilon)

{
        const int CONVERGENCE = 100;
        const int ITERMAX = 150;
        if ((f2absq>(CONVERGENCE*f1absq)) && (Cabs(q2)>epsilon) &&
           (iter<ITERMAX)) {
                q2 *= .5; // in case of overflow:
                h2 *= .5; // halve q2 and h2; compute new x2
                x2 -= h2;
               *overflow = 1;
         }
}


//compute P(x2) and make some checks

void  Polynomial ::compute_function(double f1absq,
                double *f2absq,double epsilon)

{
   int    overflow;      // overflow = TRUE  => overflow occures
                         // overflow = FALSE => no overflow occures

   do {
      overflow =0;     // initial estimation: no overflow

            // suppress overflow
      suppress_overflow();

            // calculate new value => result in f2

     f_value(nred,&f2,x2);

            // check of too big function values
      too_big_functionvalues(f2absq);

            // increase iterationcounter
```

```
        iter++;

                // Muller's modification to improve convergence
        convergence_check(&overflow,f1absq,*f2absq,epsilon);
    } while (overflow);
}


// is the new x2 the best approximation?

void Polynomial :: check_x_value(Complex *xb,double *f2absqb,
                int *rootd,double f1absq,double f2absq,double epsilon,int *noise)
{
     const double BOUND1 = 1.01;
     const double BOUND2 = 0.99;
     const double BOUND3 = 0.01;
     if ((f2absq<=(BOUND1*f1absq)) && (f2absq>=(BOUND2*f1absq))) {
                                // function-value changes slowly
        if (Cabs(h2)<BOUND3) {  // if |h[2]| is small enough =>
                q2 *= 2;  // double q2 and h[2]
                h2 *= 2;
        } else {                // otherwise: |q2| = 1 and
                                // h[2] = h[2]*q2
                q2 = Complex(cos(iter),sin(iter));
                h2=h2*q2;
        }
    } else if (f2absq<*f2absqb) {
        *f2absqb = f2absq;// the new function value is the
        *xb      = x2;          // best approximation
        *noise   = 0;           // reset noise counter
        if ((sqrt(f2absq)<epsilon) && (Cabs((x2-x1)/x2))<epsilon)
            *rootd = 1;
     }
}


void Polynomial ::root_check(double f2absqb,int *seconditer,
            int *rootd,int *noise,Complex xb)
{
    Complex df;                                 // df=P'(x0)
    const double BOUND7 = 1e-5;
    if ((*seconditer==1) && (f2absqb>0)) {
        f_value2(&f2,&df,xb);                   // f2=P(x0), df=P'(x0)
       if (Cabs(f2)/(Cabs(df)*Cabs(xb))>BOUND7) {
           // start second iteration with new initial estimations
```

```
            x0 = Complex(-1./sqrt(2),1./sqrt(2));
            x1 = Complex(1./sqrt(2),-1./sqrt(2));
            x2 = Complex(-1./sqrt(2),-1./sqrt(2));

            f_value(nred,&f0,x0);
            f_value(nred,&f1,x1);
            f_value(nred,&f2,x2);
            iter = 0;               // reset iteration counter
            (*seconditer)++;        // increase seconditer
            *rootd = 0;             // no root determined
            *noise = 0;             // reset noise counter
          }
      }
}


void Polynomial :: After_Modified()
{
  int k, m, i;
  double omega, W_Re, W_Im, T_Re, T_Im, H_Re,  H_mag, upper, lower;//, G;
  double s_H_mag;
  Complex H, temp;
  double *test2;
  double *h;

    test2=new double[NPLOT];
    H.r = 1;H.i =0;
    upper=lower=0.0;
    h = new double [N+1];
    for (i=N-1; i>=0; i--)
              h[i] = matlab[i];

   for ( k = 0; k < NPLOT; k++ ) {
              omega = PI * ((double) k) / ((double) NPLOT);
              W_Re =   cos( omega );
              W_Im = - sin( omega );
              T_Re =   cos( 2*omega );
              T_Im = - sin( 2*omega );
              for ( m = 0; m < N-1 ; m++ ) {
                    if (root[m].i == 0.0){
                              temp.r = 1-root[m].r*W_Re;
                              temp.i = root[m].r*W_Im;
              }
              else{
                        temp.r = 1-(2*root[m].r*W_Re)+
```

```
                        (root[m].r*root[m].r+(root[m].i*root[m].i))*T_Re;
                temp.i = (root[m].r * root[m].r + (root[m].i * root[m].i))*T_Im
                        -2*root[m].r*W_Im;
                m++;
            }
            H*=temp;
        }
        H_Re = h[N/2];
        for ( i = 1; i < N/2; i++ ) {
                T_Re = h[N/2 - i] * cos(omega*i);
                 H_Re += 2 * T_Re;
         }
          s_H_mag = H_Re;
          H_mag = sqrt( H.r * H.r + H.i * H.i );
          H.r = 1; H.i =0 ;
          upper += s_H_mag *H_mag;
          lower += H_mag*H_mag;
          test2[k]=H_mag;
   }
  delete []h;
  delete []test2;
}


void Polynomial::draw_magnitude()
{
        int i,n, k, remember_k,  zero_sum, zero_sum_1;
        int m=0, mini=0, single_zero1, single_zero2, sub=0, sub_1=0;
        int y1;                         // the number ofzeros inside unit circle
        int former_sub, former_sub_1;
        double omega, T_Re, H_Re,  data1;
        double delta = .2;
        double *H_mag;
        double *pp;
        double *dip;
        double *w;
        double *h_0, *h_1, *h_2;
        char   sig1,          // sign of real part
               sig2;          // sign of imagrary part
        Complex *temp_for_onuc;
        Complex *On_UnitCircle;
        Complex *NOT_OnUC;
        Complex *roots_inside;
        Complex *roots_minimum;
        Complex *roots_sub;
```

```
Complex *roots_sub_1;

pp = new double [N];
H_mag = new double [NPLOT+1];
dip = new double [N];
w = new double [N];
temp_for_onuc = new Complex [N];
On_UnitCircle = new Complex [N];
NOT_OnUC = new Complex[N];
roots_inside = new Complex [N];
roots_minimum = new Complex [N];
roots_sub = new Complex[N];
roots_sub_1 = new Complex[N];

FILE   *file_ptr;
double angle_real;  // angle of zero
double *angle_all;  // angle of zeros on the unit circle and inside the unitcircle.

single_zero1= single_zero2 =0;

for (i=N-1; i>=0; i--)
        pp[i] = matlab[i];
        for(k=0; k<NPLOT; k++){
                omega = PI * ((double) k) / ((double) NPLOT);
                H_Re = pp[N/2];
                for ( i=1; i<=N/2; i++){
                        T_Re = pp[N/2-i] * cos( omega * i);
                        H_Re += 2 * T_Re;
                }
                H_mag[k]=H_Re;

// check if the first point is a dip. it's linear and symmetric

                if( k==1 && (fabs(H_mag[0]) < 1e-3) &&
                  (H_mag[0]<H_mag[1]) ){
                        dip[m] = H_mag[0];
                         w[m] = 0;              // omega = 0
                         m++;
                         single_zero1++;
                }

  // from the second to the last point (except last one)

                if (k>=2) {
                      if(H_mag[k] < delta) {
```

```
                                    if( H_mag[k-1]<H_mag[k] && H_mag[k-1]<H_mag[k-2]
                                        && fabs(H_mag[k-1]) < 1e-3){

// dip is the distance to the horizenal line, can be positive or negative

                                            dip[m] = H_mag[k-1];
                                            w[m] =  PI * ((double) (k-1)) / ((double) NPLOT);
                                            m++;
                                            }
                                    }
                            }
                    }
// check if the last point is a dip

                    if( (fabs(H_mag[NPLOT-1]) < 1e-3) && (H_mag[NPLOT-1] <
                        H_mag[NPLOT-2])) {
                            dip[m] = H_mag[NPLOT-1];



                            w[m] = PI;
                            m++;
                            single_zero2++;
                    }

            for(n=0; n<m; n++){
                    if( (w[n]!=0) && (w[n]!=PI) ){
                            roots_minimum[mini].r = cos(w[n]);
                            roots_minimum[mini].i = sin(w[n]);
                            roots_sub[sub] = roots_minimum[mini];
                            sub++;
                            mini++;
                            roots_minimum[mini].r = cos(w[n]);
                            roots_minimum[mini].i = -sin(w[n]);
                            mini++;
                    }
                    else{
                            roots_minimum[mini].r = cos(w[n]);
                            roots_minimum[mini].i = 0.0;
                            roots_sub[sub] = roots_minimum[mini];
                            sub++;
                            mini++;
                    }
            }
```

```
//  save the subfilter response for on Unit Circle first

        h_0 = new double [ N ];
        h_1 = new double [ N ];
        h_2 = new double [ N ];

        former_sub = sub;
        for(i=0; i<sub;  i++){   // on uc
                data1 = sqrt(roots_sub[i].r*roots_sub[i].r + roots_sub[i].i*roots_sub[i].i);
                if ( (roots_sub[i].r+1)>-1E-8 && (roots_sub[i].r+1)< 1E-8 &&
                    roots_sub[i].i==0){
                        h_0[i] = 1.0;
                        h_1[i] = 1.0;
                        h_2[i] = 0.0;
                }
                else if ( (roots_sub[i].r-1)>-1E-8 && (roots_sub[i].r-1)< 1E-8 &&
                     roots_sub[i].i==0){
                        h_0[i] = 1.0;
                        h_1[i] = -1.0;
                        h_2[i] = 0.0;
                }
                else{
                        h_0[i] = 1;
                        h_1[i] = -2*data1*roots_sub[i].r/data1;
                        h_2[i] = data1*data1;
                }
        }

// put all roots (on unit circle and very close to unit circle) to temp_for_onuc[]

        i = N_half_root;
        angle_all = new double[i];
        for(k=0; k<i; k++){
                data = half_root_r[k];
                data1= half_root_i[k];
                temp_for_onuc[k].r = data;
                temp_for_onuc[k].i = data1;
                if(data ==0 && data1==0)
                        angle_all[k] = 4.444;    // zero at middle point,
                else
                        angle_all[k] = acos(data/sqrt(data*data + data1*data1));
        }

        for(n=2*m-1; n>=0; n--)
                w[2*n]=w[2*n+1]=w[n];
```

```
        zero_sum = 2*m - single_zero1 - single_zero2;
        if ( single_zero1 ==1 && single_zero2 == 1)
                zero_sum = 2*m - 1;

// zero_sum : exactly number of zeros on unit circle  (N/2)

         for(n=0; n < zero_sum ; n++){
                On_UnitCircle[n].r = 10000.00;        // pickup the first one
                angle_real = 8.1;                     // initialize the first angle unbelievable huge
                for(k=0; k<i; k++){

// choose the angle of the roots closest to dip roots angle (cos (w[n]) )

                if ((fabs(angle_all[k] - w[n] ) <= fabs(angle_real - w[n])) ){
                        if( fabs(angle_all[k] - w[n] ) == fabs(angle_real - w[n])){
                                if (fabs(temp_for_onuc[k].r - cos(w[n]) ) <
                                      fabs(On_UnitCircle[n].r - cos(w[n]))){
                                        On_UnitCircle[n] = temp_for_onuc[k];
                                        remember_k = k;
                                        }
                                }
                        else{
                                On_UnitCircle[n] = temp_for_onuc[k];
                                remember_k = k;
                                angle_real =
acos(On_UnitCircle[n].r/sqrt(On_UnitCircle[n].r*On_UnitCircle[n].r +
On_UnitCircle[n].i*On_UnitCircle[n].i));
                                }
                }
        }
// we want to get two zeroes, after got the first one, should delete it and then look for
//second one
// after find both, should restore the original roots_inside_and_on

                angle_all[remember_k] = 9.1;

                n++ ;



                On_UnitCircle[n] = 10000.00;
                angle_real = 5.1;
                for(k=0; k<i; k++){
                        if ((fabs(angle_all[k] - w[n])<= fabs(angle_real - w[n])) ){
```

```
                                if( fabs(angle_all[k] - w[n] ) == fabs(angle_real - w[n])){
                                        if (fabs(temp_for_onuc[k].r - cos(w[n])) <
                                            fabs(On_UnitCircle[n].r - cos(w[n]))){
                                                On_UnitCircle[n] = temp_for_onuc[k];
                                                remember_k = k;
                                        }
                                  }
                                else{
                                        On_UnitCircle[n] = temp_for_onuc[k];
                                        remember_k = k;
                                        angle_real =
                        acos(On_UnitCircle[n].r/sqrt(On_UnitCircle[n].r*On_UnitCircle[n].r +
                        On_UnitCircle[n].i*On_UnitCircle[n].i));
                                }
                        }
                }
                angle_all[remember_k] = 5.1;
        }
  zero_sum_1 = 0;

// real zeros on unit circle

        if(zero_sum % 2 ==1)
                zero_sum++;
        for(n=0; n<zero_sum; n++){
                if(On_UnitCircle[n].i == 0){
                        roots_sub_1[sub_1] = On_UnitCircle[n];
                        sub_1++;
                        zero_sum_1++;
                }
                else{
                        roots_sub_1[sub_1] = On_UnitCircle[n];
                        sub_1++;
                        zero_sum_1+=2;
                }
        }

  former_sub_1 = sub_1;

// find the roots inside the unit circle

  int y=0;
  i = N_half_root;
  for(k=0; k<i; k++){  // put all roots (on uc and inside uc) to temp_for_onuc[]
                data = half_root_r[k];
```

```
                data1= half_root_i[k];
                if ( (data*data + data1*data1)<=1 ){
                        temp_for_onuc[y].r = data;
                        temp_for_onuc[y].i = data1;
                        y++;
                 }
   }
// look for the zeros inside the unit circle

        y1=0;
        for(k=0; k<y; k++){             // all roots inside & on the unit circle
                int temple_1 = 0;
                for(n=0; n<zero_sum; n++){
                        if(temp_for_onuc[k].r != On_UnitCircle[n].r )
                                temple_1++;
                }
                if(temple_1 == zero_sum ){
                if((temp_for_onuc[k].r*temp_for_onuc[k].r+temp_for_onuc[k].i*
                  temp_for_onuc[k].i)==1)
                        printf(".");
                roots_inside[y1]=temp_for_onuc[k];
                if(roots_inside[y1].i == 0){
                        roots_minimum[mini] = roots_inside[y1];
                        roots_sub[sub] = roots_minimum[mini];
                        sub++;
                        roots_sub_1[sub_1] = roots_minimum[mini];
                        sub_1++;
                        mini++;
                        y1++;
                }
                else{
                        roots_minimum[mini] = roots_inside[y1];
                        roots_sub[sub] = roots_minimum[mini];
                        sub++;
                        roots_sub_1[sub_1] = roots_minimum[mini];
                        sub_1++;
                        mini++;
                        roots_minimum[mini].r = roots_inside[y1].r;
                        roots_minimum[mini].i = -roots_inside[y1].i;
                        mini++;
                        y1 += 2;
                        }
                }
        }
```

```
// write the roots of the minimum-phase filter to a filter.

        file_ptr = fopen( "roots_minimum.dat", "w" );
        fprintf(file_ptr, "%d\n", mini);
        for(k=0; k<mini; k++){
                sig1 = (roots_minimum[k].r >= 0) ? ' ' : '-';
                sig2 = (roots_minimum[k].i >= 0) ? ' ' : '-';
                fprintf(file_ptr,"%c%.16e     %c%.16e\n",sig1, fabs(roots_minimum[k].r),
                                                   sig2, fabs(roots_minimum[k].i));
        }
        fclose(file_ptr);

        file_ptr = fopen( "roots_minimum.txt", "w" );
        fprintf(file_ptr, "                Minimum-Phase Filter Design\n");
        fprintf(file_ptr, "                  Filter Order = %d\n\n", mini);
        fprintf(file_ptr, "        Real Part                  Imaginary Part\n");
        for(k=0; k<mini; k++){
                sig1 = (roots_minimum[k].r >= 0) ? ' ' : '-';
                sig2 = (roots_minimum[k].i >= 0) ? ' ' : '-';
                fprintf(file_ptr,"%c%.16e     %c%.16e\n",sig1, fabs(roots_minimum[k].r),
                                                   sig2, fabs(roots_minimum[k].i));
        }
        fclose(file_ptr);

// get subfilter impulse response

        for(i= former_sub; i< sub; i++){   // inside and on axis
                data1 = sqrt(roots_sub[i].r*roots_sub[i].r + roots_sub[i].i*roots_sub[i].i);
                if(roots_sub[i].i == 0){
                        h_0[i] = 1.00;
                        h_1[i] = -roots_sub[i].r;
                        h_2[i] = 0.00;
                }
                else{
                        h_0[i] = 1.00;
                        h_1[i] = -2*data1*roots_sub[i].r/data1;;
                        h_2[i] = data1*data1;
                }
        }
        file_ptr = fopen("mini_subfilter_response.dat","w");
        m = 0;
        mini =0;
        for(k=0; k<i; k++){
                if(h_2[k] == 0){
                        mini++;
```

```
        }
        else{
                m++;
        }
}
fprintf(file_ptr, "%d\n", mini+m);
fprintf(file_ptr, "3\n");
m=0;
mini=0;

for(k=0; k<i; k++){
        if(h_2[k] == 0){
                fprintf(file_ptr, "%3.8f  %16.8f  %8d\n",h_0[k], h_1[k], h_2[k]);
                mini++;
        }
        else{
                pp[m] = h_0[k];
                H_mag[m] = h_1[k];
                w[m] = h_2[k];
                m++;
        }
}
for(k=0; k<m; k++)
        fprintf(file_ptr, "%3.8f  %16.8f   %16.8f\n",pp[k], H_mag[k], w[k]);
fclose (file_ptr);

file_ptr = fopen("mini_subfilter_response.txt","w");
fprintf(file_ptr, "          The Subfilters of Minimum-Phase Filter\n\n");
fprintf(file_ptr, "            The Number of the Subfilters = %d\n", mini+m);
fprintf(file_ptr, "   The Number of impulse response coefficients per subfilter =
                    3\n\n");
fprintf(file_ptr,"  m        Hm(0)             Hm(1)             Hm(2)\n");
fprintf(file_ptr,"==========================================
                  ==========\n");
m = 0;
mini =0;
for(k=0; k<i; k++){
        if(h_2[k] == 0){
                fprintf(file_ptr, "%3d  %16.8f  %16.8f   %16.8f\n",mini+1, h_0[k],
                        h_1[k], h_2[k]);
                mini++;
        }
        else{
                pp[m] = h_0[k];
                H_mag[m] = h_1[k];
```

```
                w[m] = h_2[k];
                m++;
        }
}
for(k=0; k<m; k++)
        fprintf(file_ptr, "%3d  %16.8f  %16.8f   %16.8f\n",k+mini+1,pp[k],
                H_mag[k], w[k]);
fclose (file_ptr);



for(i=0; i<former_sub_1;  i++){   // on the unit circle
        if (roots_sub_1[i].i == 0){
                h_0[i] = -2.0;
                h_1[i] = 1.0;
                h_2[i] = 0.0;
        }
        else{
                data1 = sqrt(roots_sub_1[i].r*roots_sub_1[i].r +
                        roots_sub_1[i].i*roots_sub_1[i].i);
           h_0[i] = -2*roots_sub_1[i].r/data1;
           h_1[i] = 1.0;
           h_2[i] = 0.0;
        }
}
for(i= former_sub_1; i< sub_1; i++){   // inside and on the axis
        data1 = sqrt(roots_sub_1[i].r*roots_sub_1[i].r +
                 roots_sub_1[i].i*roots_sub_1[i].i);
         if(roots_sub_1[i].i == 0){
                h_0[i] = -(data1 + 1/data1);
                h_1[i] = 1.0;
                h_2[i] = 0.0;
        }
        else{
                h_0[i] = data1*data1 + 4*roots_sub_1[i].r
                        *roots_sub_1[i].r/(data1*data1)+ 1/(data1*data1);
                h_1[i] = -2*(data1+1/data1)*roots_sub_1[i].r/data1;
                h_2[i] = 1.0;
        }
}
delete [] pp;
delete [] H_mag;
delete [] w;
delete [] On_UnitCircle;
delete [] temp_for_onuc;
```

```
        delete [] NOT_OnUC;
        delete [] roots_minimum;
        delete [] roots_sub;
        delete [] angle_all;
        delete [] h_0;
        delete [] h_1;
        delete [] h_2;
        delete [] roots_sub_1;
}

void Polynomial ::check_value()
{
        int k, m, i;
        double omega, W_Re, W_Im, T_Re, T_Im, H_Re,  H_mag,  upper, lower,G;
        double  s_H_mag;
        Complex H, temp;
        double test,test3;
        double *test2;
        double *h;

        test2=new double[NPLOT];
        FILE *file_ptr;
        H.r =1; H.i =0;
        upper=lower=0.0;

        int I;
        file_ptr = fopen( "roots_minimum.dat", "r" )  ;
        fscanf( file_ptr, "%d ", &I);
        root = new Complex [N];
        for(m=0; m<N; m++)
                root[m].r = root[m].i =0.0;
        for(m=0; m<I; m++){
             fscanf(file_ptr,"%lf  %lf", &test, &test3);
             root[m].r=test;
             root[m].i=test3;
        }
        fclose(file_ptr);

        h = new double [N+1];
        for (i=N-1; i>=0; i--)
             h[i] = matlab[i];

        for ( k = 0; k < NPLOT; k++ ) {
             omega = PI * ((double) k) / ((double) NPLOT);
             W_Re =   cos( omega );
```

```
            W_Im = - sin( omega );
            T_Re =   cos( 2*omega );
            T_Im = - sin( 2*omega );

            for ( m = 0; m < I ; m++ ) {
                  if (root[m].i == 0.0){
                        temp.r = 1-root[m].r*W_Re;
                        temp.i = root[m].r*W_Im;
                  }
                  else{
                        temp.r = 1-(2*root[m].r*W_Re)+
                              (root[m].r*root[m].r+(root[m].i*root[m].i))*T_Re;
                        temp.i = (root[m].r * root[m].r + (root[m].i * root[m].i))*T_Im
                               -2*root[m].r*W_Im;
                        m++;
                  }
                  H*=temp;
            }
            H_Re = h[N/2];
            for ( I = 1; i < N/2; i++ ) {
                  T_Re = h[N/2 - i] * cos(omega*i);
                  H_Re += 2 * T_Re;
            }
           s_H_mag = H_Re;
          H_mag = sqrt( H.r * H.r + H.i * H.i );
          H.r = 1; H.i =0;
          upper += sqrt(fabs(s_H_mag));
          lower += H_mag;
          test2[k]=H_mag;
}

          G=upper/lower;
          RootCoeff(G);

          file_ptr = fopen("roots_minimum.dat","a");
          fprintf(file_ptr,"%.10f\n", G);
          fclose( file_ptr );

          file_ptr = fopen("roots_minimum.txt","a");
          fprintf(file_ptr,"\nGain = %.10f\n", G);
          fclose( file_ptr );

          file_ptr = fopen("mini_subfilter_response.dat","a");
          fprintf(file_ptr,"%.10f\n", G);
          fclose( file_ptr );
```

```
        file_ptr = fopen("mini_subfilter_response.txt","a");
        fprintf(file_ptr,"Gain = %.10f\n", G);
        fclose( file_ptr );

        delete []h;
        delete []test2;
}

Polynomial :: ~Polynomial()
{
        delete [] psave1;
        delete [] root;
        delete [] psave;
}

char main(void)
{
    Polynomial Poly;
    unsigned char error;
    Poly.read_from_file( );
    error=Poly.get_roots();
    if ( !error ) {
                    Poly.write_to_file();
                    Poly.After_Modified();
                    Poly.draw_magnitude();
                    Poly.check_value();
                    Poly.write_to_screen();
          }
     else {
                     Poly.write_error_message( error );
                     return 1;
          }
    return 0;
    }
```

VITA

Mr. Wei Su was born and raised in QingDao, China. He attended elementary school and high school in QingDao until 1990. In 1990 he entered Tianjin Polytechnic University after a competitive examination. During his time in the university he majored in Mechanical and Electrical Engineering. From 1994 to 2000, he worked in QingDao and Beijing as an electrical engineer. In Fall 2000 he was awarded a graduated research assistantship in University of Tennessee Space Institute and came to the Unit States to pursue a Master's Degree in Electrical Engineering. He completed the masters program at UTSI in Summer 2002. The master's degree was received in August 2002.