2000

Design of one-dimensional and two-dimensional filters with finite-wordlength coefficients using genetic algorithms.

Alfred. Lee
University of Windsor

Design of 1-D and 2-D Filters with Finite-wordlength Coefficients Using Genetic Algorithms

by

Alfred Lee

A Thesis
Submitted to the College of Graduate Studies and Research
through the Department of Electrical Engineeringin
Partial Fulfillment of the Requirements for the
Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada

2000

0-612-62236-3

ABSTRACT

This thesis presents an enhanced version of a Genetic Algorithm to design discrete filter coefficients. This optimization based algorithm has the advantages of eliminating truncation error and quantization process in filter design, and to produce filters which are suitable for high speed signal processing applications. We have examined the usefulness of various error norms, such as Least Mean Square and Minimax, and their impact on the convergence rate and the result. We also present the application of various encoding schemes applied to the filter coefficients and their effect on obtaining optimized filters. Examples of 1-D & 2-D FIR and 2-D IIR filters are provided to illustrate the design procedures and to determine the best Genetic Algorithm combinations for digital filter design.

To My Lovely Family Members,

Mom, Dad, Linda, and Andrew.

ACKNOWLEDGMENTS

I would like to give my sincere thanks to my supervisor, Dr. M. Ahmadi. His encouragement and innovative ideas enabled me to successfully complete this thesis. His positive disposition to students and his appreciation of students' work made my school life meaningful at the University of Windsor. I would also like to thank him for the opportunity to gain invaluable experience from the presentation of our work at a conference in Victoria, British Columbia.

I would like to show my sincere appreciate to Dr. R.S. Lashkari and Dr. B. Nowrouzian for their recommendations and sincere support. They have shared their values on life and broaden my eyes to consider other engineering applications. Their friendly smile and positive attitude have always been a comfort to me.

I wish to thank my committee members, Dr. J.J Soltis, and Dr. G.A. Jullien for their comments and direction in different filter designs. They have greatly contributed to this thesis and allowed me to explore applications to the Double Base Number System, developed by V.S. Dimitrov, and other members at the University of Windsor.

I would like to extend my appreciation to my wonderful friend, Kaamran Raahemifar and his family for always being there for me in times of need. He has given me valuable suggestions throughout my Master program. He was always there to encourage me to do a better job. Qi Chen was another fine role model. His strong work ethics and research desire has inspired me to pursue the area of research in engineer. I hold fond memories of all those late nights at school working on our projects and drawing mutual support from each other. I would also like to take this opportunity to thank the VLSI group members, Zulfiqar Ahmed, Danny Conflitti, Tracy Franklin, Hossain Hajimowlana, Andy Bezaire, Roberto Muscedere, Marjan Shahkarami, Hamid Safiri, Saeid Sadeghi-Emamchaie and our wonderful secretary of Electrical Engineering, Mrs. S.A. Ouellette. They really made my two years Master program as enjoyable and productive. Many thanks especially to Rob and Andy for contributing the persudo random function generator and making it work as it should.

The last, but not the least, my family has given me their time, energy, financial support, and unbounded love. They have given everything, the best of everything to me. A simple 'thank you' may not be enough to justify the blessing I received from them. "I love you Dad, Mom, Linda, and my brother, Andrew."

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

1.1 INTRODUCTION

Digital Signal Processing (DSP) is a field of engineering that deals with the processing, enhancement, and extraction of information for discrete time data. DSP has found many applications in the area of radar signal processing [66], speech processing [45], communication [63], biomedical image processing [58], and computer vision [22]. As the cost of hardware for DSP processors has dropped drastically over the years along with the increase in processing power, these DSP algorithms are employed more frequently in many devices and equipment, such as home theatre [65], video CD players [19], etc..

One important area of development in signal processing is digital filtering. It is a computational process which transforms an input array of numbers to an output array of numbers according to a prescribed mathematical function. Digital filters can be of 1-Dimension, 2-Dimension, and in general N-Dimension. The process can be temporal, spatial, or temporal and spatial. Examples are, speech signal [62], images [51] and TV signals [5] respectively. Digital filters can be implemented as software on a digital computer, or can be implemented on a special purpose device. Implementation of digital filters on a digital computer unfortunately suffers from low throughput rate due to time required for an instruction to be fetched and executed in a Von

Neumann type machine. For this reason, most high throughput rate DSP engines are implemented using Application Specific Integrated Circuit (ASIC) processor.

Digital filters repetitively implement the three operations of add, multiply, and delay. The bottleneck in achieving high throughput rate in a DSP process is the multiplier, and various circuits that have been proposed for fast multiplication [47, 26, 15]. Many special purpose fast multipliers attempt to reduce the number of partial products being summed. Partial product reduction is often obtained using special number representations of the filter coefficients, such as Power-of-Two coefficients filter [20, 44, 67], Canonic Signed-Digit (CSD) coefficients filter [17, 59], etc..

In this thesis, methods will be presented for the design of 1-D and 2-D FIR and IIR filters with either CSD coefficient or discrete coefficients for arbitrary specifications.

1.2 CHARACTERISTIC OF FIR FILTERS

In general, there are two classes of linear digital filters, Finite-Impulse Response (FIR) and Infinite-Impulse Response (IIR). The digital FIR filter is a well-known class of stable filters. Its computation process only involves computation with past and present input data. A problem with IIR filters is that the stability is not always guaranteed, since its computation process consists of the present and past input data, as well as the past output data. The utilization of past output data has the advantage of requiring lower order IIR filter for the same specification than its FIR counterpart. This has the effect on reducing the number of operations required to complete the process which is suitable for high speed applications. Both classes of FIR and IIR filters are applicable to the 1-D and the 2-D cases.

1.2.1 One-Dimensional FIR Filter

A 1-D FIR digital filter of order N is characterized by the z-transfer function,

$$H(z) = \sum_{n=0}^{N} a_n z^{-n}, \tag{1.1}$$

where $[a_n]$ are the unknown filter coefficients. By determining the unknown coefficients, $[a_n]$, different types of filters can be designed, such as lowpass, highpass, and bandpass filters. Frequency response can be obtained by substituting $z=e^{jwT}$ into equation (1.1),

$$H(e^{j\omega T}) = \sum_{n=0}^{N} a_n e^{-jn\omega T}, \tag{1.2}$$

where ω is the angular frequency in rad/sec, and T is the sampling period in second. Linear phase response of FIR filters can be obtained by imposing symmetry on the coefficients, i.e.:

$$a_n = a_{N-n}\,. \tag{1.3}$$

In this case, equation (1.2) can be simplified to

$$H(e^{j\omega T}) = a_0 + 2 \times \sum_{n=1}^{N/2} a_n \cos(n\omega T). \tag{1.4}$$

Equation (1.4) shows that the linear phase characteristic of FIR filter reduces the number of unknown number of coefficients almost by half.

1.2.2 Techniques for 1-D FIR Filter Design

Various techniques have been developed in the past to design 1-D FIR filters. These techniques range from closed-form evaluations to iterative algorithms. One of the most common methods is the Fourier Series in conjunction with windowing functions [46]. The choice of different windows will result in different filter characteristics, in terms of transition width and stopband attenuation. The most frequently used functions are the Hann, Hamming, Blackman, and Kaiser windows [29,2]. Alternative methods in FIR filter design are the application of linear and non-linear programming methods [37,54].

1.2.3 Two-Dimensional FIR Filter

A 2-D FIR digital filter of order *N+M* is characterized by the z-transfer function,

$$H(z_1,z_2)=\sum_{n=0}^{N}\sum_{m=0}^{M} a(n,m)z_1^{-n}z_2^{-m}, \tag{1.5}$$

where a(n,m) are the unknown filter coefficients. The corresponding filter response is obtained by substituting $z_1 = e^{j\omega_1 T}, z_2 = e^{j\omega_2 T}$ into equation (1.5),

$$H(e^{j\omega_1 T},e^{j\omega_2 T})=\sum_{n=0}^{N}\sum_{m=0}^{M} a(n,m)e^{-jn\omega_1 T}e^{-jm\omega_2 T}, \tag{1.6}$$

where ω_1, ω_2 are the angular frequencies in rad/sec, and T is the sampling period in second. For a 2-D lowpass filter, octagonal symmetry [29] is highly desirable. Octagonal symmetry includes the following two characteristics:

$$1.\quad \begin{aligned} |H(e^{j\omega_1 T},e^{j\omega_2 T})| &= |H(e^{-j\omega_1 T},e^{j\omega_2 T})| \\ &= |H(e^{j\omega_1 T},e^{-j\omega_2 T})| \\ &= |H(e^{-j\omega_1 T},e^{-j\omega_2 T})|. \end{aligned} \tag{1.7}$$

2. $|H(e^{j\omega_1 T}, e^{j\omega_2 T})| = |H(e^{j\omega_2 T}, e^{j\omega_1 T})|$. (1.8)

For octagonal symmetry and linear phase, equation (1.6) can be simplified to (assume $N = M$)

$$\begin{aligned} H(e^{j\omega_1 T}, e^{j\omega_2 T}) = a(0,0) & \\ &+\sum_{n=1}^{N} 2a(n,0)[\cos(n\omega_1 T) + \cos(n\omega_2 T)] \\ &+\sum_{n=1}^{N} 4a(n,n)\cos(n\omega_1 T)\cos(n\omega_2 T) \\ &+\sum_{m=1}^{N-1}\sum_{n=m+1}^{N} 4a(n,m)[\cos(n\omega_2 T)\cos(m\omega_1 T) + \cos(m\omega_2 T)\cos(n\omega_1 T)]. \end{aligned}$$

(1.9)

1.2.4 Techniques for 2-D FIR Filter Design

One useful technique in designing 2-D FIR filters involves the transformation of a 1-D filter. A 1-D FIR filter is designed using Fourier Series in conjunction with window functions, as mentioned previously. This is then converted to a 2-D FIR filter by using the McClellan transformation [42]. Optimization techniques [7] are alternative methods in designing 2-D FIR filters. Similar to 1-D FIR filter design, the coefficients of a 2-D FIR filter are designed by minimizing an objective function [55]. This objective function is obtained as some measure of the difference between the ideal response and the designed response of the filter.

1.2.5 One-Dimensional IIR Filter

A 1-D IIR digital filter is characterized by the z-transfer function,

$$H(z) = \frac{\sum_{n=0}^{N} a_n z^{-n}}{\sum_{m=0}^{M} b_m z^{-m}}, \tag{1.10}$$

where [a$_n$] and [b$_n$] are the unknown coefficients. Various filter types [29] can be designed by determining the unknown coefficients. The corresponding frequency response is obtained by substituting $z=e^{jwT}$ into equation (1.10),

$$H(e^{j\omega T}) = \frac{\sum_{n=0}^{N} a_n \, e^{-jn\omega T}}{\sum_{m=0}^{M} b_m \, e^{-jm\omega T}}, \tag{1.11}$$

where ω is the angular frequency in rad/sec, and T is the sampling period in second.

1.2.6 Techniques for 1-D IIR Filter Design

Just as for the case of 1-D FIR filter design, there are also as many types of design techniques for 1-D IIR filters. The most traditional approach involves the discretization of an analogue filter to a digital filter [3], which are known to be Butterworth, Chebyshev, and Elliptical filters. The process of analogue design approximates the magnitude and/or the phase responses of the ideal filter specifications. The analogue filter is then converted to a digital filter by applying the bilinear transformation which is one of the most commonly used transforms [3]. Another approach to design IIR filters is iterative techniques using linear [13] or non-linear programming [39,33]. This is an optimization technique [11] where an iterative method is used in order to obtain a minimum error between the designed filter and the ideal filter specifications.

1.2.7 Two-Dimensional IIR Filter

A 2-D IIR digital filter of order $N{+}M$ is characterized by the z-transfer function,

$$H(z_1, z_2) = \frac{A(z_1, z_2)}{B(z_1, z_2)} = \frac{\sum_{m=0}^{M} \sum_{n=0}^{N} a(n,m) z_1^{-n} z_2^{-m}}{\sum_{m=0}^{M} \sum_{n=0}^{N} b(n,m) z_1^{-n} z_2^{-m}}, \tag{1.12}$$

where a(n,m) and b(n,m) are the unknown filter coefficients. From Equation (1.12), the output of the recursive filter is a function of the present and the past inputs as well as the past output of the filter. Therefore, it is possible for the output to become increasingly large despite the size of the input signal. As a result, the filter can become unstable. To maintain the filter stability, the following constraint must be satisfied on the denominator,

$$D(z_1,z_2) \neq 0 \text{ for } \bigcap_{i=1}^{2} |z_i| \geq 1. \tag{1.13}$$

The frequency response can be obtained by substituting $z_1 = e^{j\omega_1 T}, z_2 = e^{j\omega_2 T}$ into Equation (1.12).

$$H(z_1,z_2) = \frac{A(z_1,z_2)}{B(z_1,z_2)} = \frac{\sum_{m=0}^{M}\sum_{n=0}^{N} a(n,m)e^{-jn\omega_1 T}e^{-jm\omega_2 T}}{\sum_{m=0}^{M}\sum_{n=0}^{N} b(n,m)e^{-jn\omega_1 T}e^{-jm\omega_2 T}} \tag{1.14}$$

1.2.8 Techniques for 2-D IIR Filter Design

There have been various techniques proposed in the past to design 2-D recursive filters. One useful technique is to transform a designed 2-D analogue filter into a 2-D digital filter by using the double bilinear transformation [10]. This transformation preserves stability and frequency characteristic of the analogue filter. Other techniques to design 2-D recursive filters are to use linear [14] and non-linear programming approaches [1,50]. Assuming that factorization of $A(z_1,z_2)$ and / or $B(z_1,z_2)$ is possible, one may define two useful subclasses of 2-D digital IIR filters as follows:

Separable Product

In this subclass of 2-D recursive digital filter, it is assumed that factorization of both $A(z_1,z_2)$ and $B(z_1,z_2)$ into polynomials consisting of z_1 and z_2 variables is possible. Therefore, the transfer function of the 2-D filter becomes

$$H(z_1,z_2) = \frac{A_1(z_1)A_2(z_2)}{B_1(z_1)B_2(z_2)} = H_1(z_1)H_2(z_2), \qquad (1.15)$$

where

$$H_i(z_i) = \frac{A_i(z_i)}{B_i(z_i)}, \qquad \text{for i= 1,2,} \qquad (1.16)$$

and

$$A_i(z_i) = \sum_{m=0}^{M_i} a_{im} z_i^{-m}, \qquad (1.17)$$

$$B_i(z_i) = \sum_{n=0}^{N_i} b_{in} z_i^{-n}, \qquad \text{for i=1,2.} \qquad (1.18)$$

In this subclass of recursive filter, a 2-D filter design problem is reduced to that of the 1-D filter. Therefore, filters can be designed by any known procedures of 1-D IIR filters [3]. The disadvantage of this subclass is the shape of cut-off boundary, which is restricted to a rectangular one.

Separable Denominator Non-Separable Numerator

The transfer function of this subclass of filters is

$$H(z_1,z_2) = \frac{A(z_1,z_2)}{B_1(z_1)B_2(z_2)} = \frac{\sum_{i=0}^{M_1}\sum_{j=0}^{N_2} a_{ij} z_1^{-i} z_2^{-j}}{\left[\sum_{i=0}^{M_2} b_{1i} z_1^{-i}\right]\left[\sum_{j=0}^{N_2} b_{2j} z_2^{-j}\right]}. \qquad (1.19)$$

The denominator is a product of two independent functions. This subclass of 2-D filters has the advantages of designing circular, elliptical and in general non-symmetrical cut-off boundary 2-D filters [1].

1.3 IMPLEMENTATION ISSUES FOR DIGITAL FILTERS

Designed filter coefficients using the techniques described in the previous sections require quantization and truncation processes to convert the coefficients into discrete numbers for dedicated hardware realization. Each of these processes introduces error into the filter. For this reason, four types of iterative methods are often employed to design discrete coefficients filters, they are branch and bound optimization [40], discretization and re-optimization [9], simulated annealing [8], and genetic algorithm [57].

Branch and bound optimization is a global search technique that is composed of branches and nodes. For each open branch, it requires a series of computations to determine its next move by either creating more open branches or to terminate the current node. It is the most computationally intensive technique among all the four methods. Discretization and re-optimization requires less computation than branch and bound optimization. However, it only generates sub-optimum results. Simulated annealing is another optimization technique that balances convergence rate and optimality. However, simulated annealing suffers from problems in designing the 'cooling curve' that has direct effect on its performance and search result. The Genetic Algorithm is a stochastic search method that uses a similar strategy to simulated annealing, but it does not depend on a 'cooling curve'. In addition, the Genetic Algorithm utilizes multi-points search strategy that is more effective in escaping from local optimum. A number of papers have been presented in the past on digital filter design using Genetic Algorithms [57,12,61,35,20,27,28,31,4,32,64]. However, only Ashrafzadeh and Nowrouzian [6] provide a solution to the design of Canonic Signed-Digit filter coefficients. The utilization of Canonic Signed-Digit coefficients has the advantage of minimizing the use of hardware resources [59].

In this thesis, the Genetic Algorithm is used as an iterative method to manipulate coefficients in 1-D and 2-D digital filter design.

1.4 VARIOUS NUMBER SYSTEMS USED IN FILTER DESIGN

Another consideration of digital filter implementation is the choice of Number System. A digital filter can be realized by either a DSP chip or ASIC (Application Specific Integrated Circuit) design. A DSP chip is divided into two classes, fixed point and floating point computation. Each of these methodologies requires the designer to decide on the most suitable number system to implement the filter. The number system is one of the important factors affecting the overall performance of the filter, hardware resources, and power dissipation. The following are brief descriptions for each commonly used number system.

1.4.1 Signed Magnitude Number System

Each coefficient is represented by a binary string. The first digit is the sign bit followed by a binary representation of the magnitude coefficient.

$$\text{coefficient representation} = [b_0\ b_1\ b_2\ b_3\ \ldots\ b_{N-1}],$$

$$a = \sum_{i=1}^{N-1} b_i \bullet 2^{i-1} \qquad , \text{for } b_0 = 0 \qquad ,$$

where b_i is a binary bit. The first bit is reserved as a sign bit. If b_o is equal to 0, the coefficient is a positive number, and if b_o is equal to 1, the coefficient is a negative number. The negative coefficients are obtained by evaluating the two's complement [38] of the corresponding positive numbers. For example, 0.25 and -0.25 are encoded by [0 0 0 1 0] and [1 1 1 1 0] respectively.

1.4.2 Signed-Digit Number System

Each coefficient is represented by a string of ternary digits.

$$\text{coefficient representation} = [t_0\ t_1\ t_2\ \ldots\ t_{N-1}],$$

$$a = \sum_{i=0}^{N-1} t_i \times 2^{-i},$$

where t_i is the digit that would accept either -1, 1, or 0, and N is the coefficient wordlength. An example of representation is [1 0 0 $\bar{1}$ 0] that is equivalent to 0.875, where $\bar{1}$ is a symbol corresponding to a -1.

1.4.3 Power-Of-Two Number System

Each coefficient is represented by a string with only one non-zero digit in it.

$$\text{coefficient representation} = [t_0\ t_1\ t_2\ \dots\ t_{N-1}],$$

where $\sum_{i=0}^{N-1} |t_i| = 1$ or 0,

$$a = t_i \times 2^{\pm i}, \qquad \text{for } i = 0,1,2,\dots..$$

where t_i is the digit that would accept either -1, 1, or 0. Only one non-zero digit is allowed in each coefficient. An example of Power-of-two coefficient is [0 0 $\bar{1}$ 0 0] which corresponds to -0.25, where the most significant digit is located on the left handside counting as 2^0.

1.4.4 Canonic Signed-Digit Number System

Canonical Signed-Digit (CSD) number system is similar to the Signed Digit representation with the restriction of the maximum number of non-zero digit. A real number is represented by a string of ternary digits '0', '1', and '$\bar{1}$, where '$\bar{1}$' is defined as '-1'. A CSD representation is characterized has the feature that no two non-zero digits occuring adjacent to each other,

$$\text{coefficient representation} = [t_0\ t_1\ t_2\ \dots\ t_{N-1}], \qquad \text{for } t_i^2 + t_{i+1}^2 \le 1$$

$$a = \sum_{i=0}^{N-1} t_i \times 2^{-i},$$

where t_i is the ternary digit. For example, a string of [0 1 0 $\bar{1}$ 0 0] represents a continuous number of 0.375, where the most significant digit is located on the left handside counting as 2^0.

1.4.5 Double Base Number System

The Double-Base Number System (DBNS) is a recently developed number representation by Dimitrov and Jullien [16, 25]. The DBNS uses the bases 2 and 3; it is redundant and very sparse, and has a simple two-dimensional representation. An extremely sparse form of the DBNS uses a single non-zero digit to represent any real number with arbitrary precision. In this case the single digit can be identified by its co-ordinates (indices) in the two-dimensional representation space. The single digit representation is shown as below.

$$\text{coefficient representation} = s\, 2^b\, 3^t,$$

where $s \in \{-1, 0, 1\}$ and b, and t, are integers we will refer to as binary and ternary exponents respectively. We thus can represent coefficients by the 3-tuple {s, b, t}. This exponent representation means that multiplication and division are easy; addition and subtraction are more difficult. An example of a tuple {-1, 3, -2} represents the real coefficient -0.8889.

1.4.6 Comparison of Various Number Systems

In this section, we will investigate the differences between each of the number system outlined in the previous section. Each number system is implemented with six bit wordlength to illustrate their characteristic. The following diagrams show all of the available valid data within the range of zero to one according to their number system schemes.

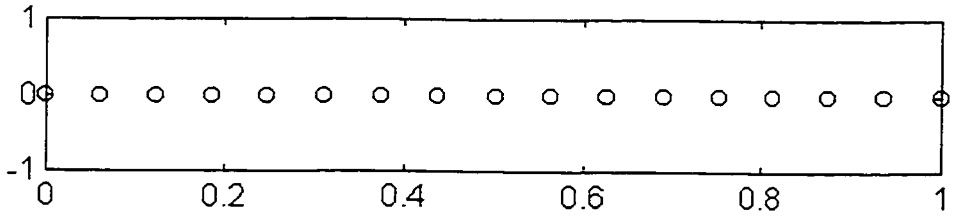

Fig.1a Signed Magnitude Number System

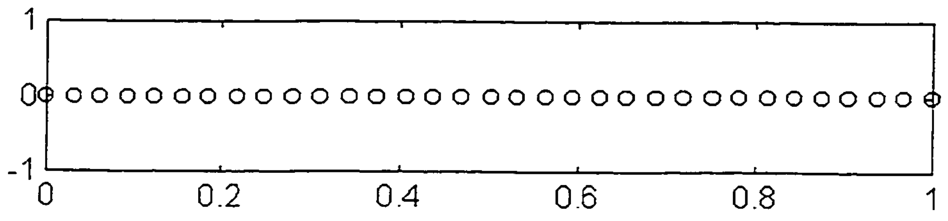

Fig.1b Signed-Digit Number System

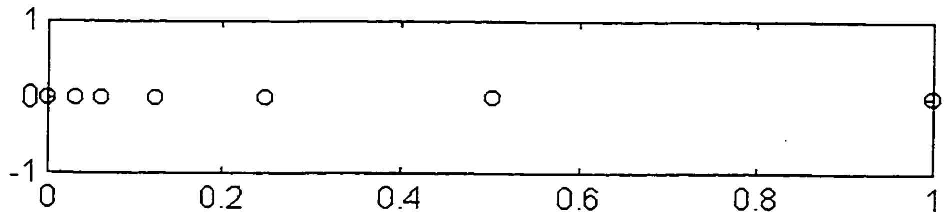

Fig.1c Power-of-Two Number System

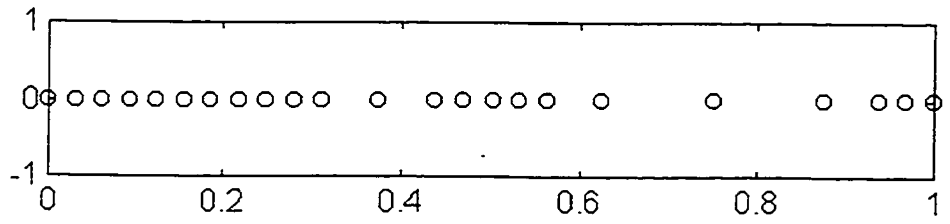

Fig.1d Canonic Signed-Digit Number System (with two non-zero digits)

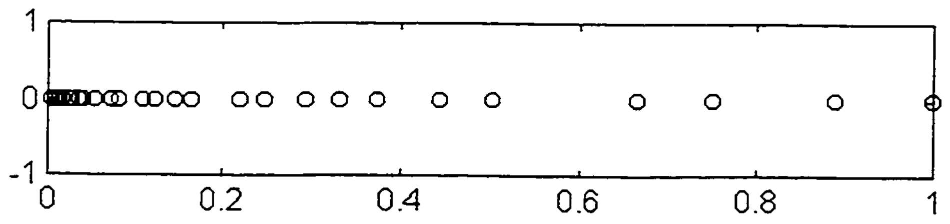

Fig.1e Double Based Number System
(using a single digit)

Signed Digit Number System (Fig.1b) has twice as much data as the Signed Magnitude (Fig.1a) with equal spacing. In contrast, Power-of-Two, CSD and DBNS, their spacing are non-uniformly distributed. They also tend to contain more data points at the lower end (near zero). The Power-of-two number system is useful for designing fast, small size filters [44,53] by eliminating some of the partial products from a full binary multiplier, but it contains the largest gaps between the range of 0.2 and 1 relative to other number systems (Fig.1c). Increasing the wordlength, however will not reduce the size of the gaps in this number system. On the other hand, with an extra non-zero digit in the representation, CSD Number System consists of more data points than Power-Of-Two. DBNS, (Fig.1e) has the advantage of generating more data at the lower end with only six bit long wordlength. In the example, the distribution of DBNS and CSD are both similar.

1.5 ORGANIZATION OF THIS THESIS

Chapter 2 presents the Genetic Algorithm and its application in digital FIR filter design. Chapter 3 presents various design examples and comparisons. Chapter 4 outlines the design of 2-D IIR and FIR filters using Genetic Algorithm. Chapter 5 provides concluding remarks.

CHAPTER 2. GENETIC ALGORITHM

2.1 INTRODUCTION TO THE GENETIC ALGORITHM

The Genetic Algorithm (GA) is a relatively new optimization technique that was originally developed by Howlland [23] in 1975. It was further modified by Goldberg [21], and others. Their primary goal was to abstract and rigorously explain the adaptive processes of natural systems and to design artificial system software that retains the important mechanisms of natural systems [21]. The characteristics of GA include multi-objective[18], exponential convergence rate, coded variables and natural selection that provide advantages in solving discrete space problems. Unlike most of the well-known optimization techniques (such as simulated annealing [30], branch and bound optimization technique [40]), GA searches a population of points rather than a single point at each iteration. This feature prevents the search from being captured by the error surface minima, and provides different search directions to seek a global solution. Another characteristic of GA is that it utilizes stochastic rather than deterministic operators which allows the GA to perform on a discontinuous space without disruption.

GA is an artificial genetic system which is based on the processes of natural selection and natural genetic [21], and which has been effectively implemented in an optimization scheme. The genetic based optimization scheme is modeled by three major operators: Reproduction, Crossover, and Mutation. Unknown variables are stored in a place, named Population, and will be manipulated by the operators consecutively as shown in the basic GA Cycle diagram, Fig.2.

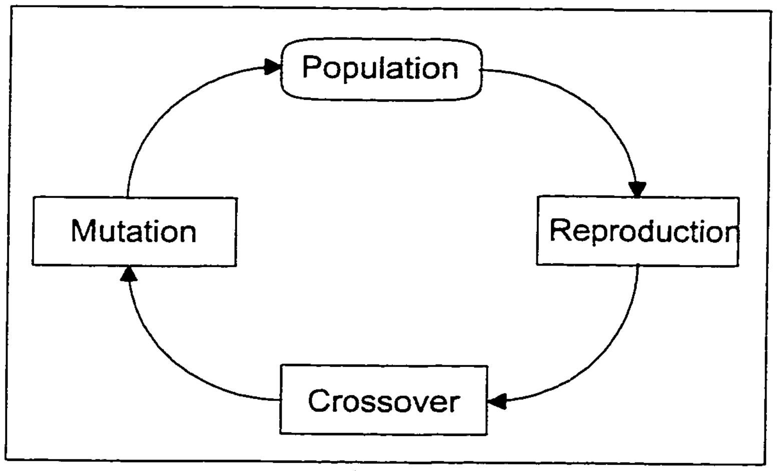

Fig.2: Genetic Algorithm Cycle

The population is a collection of *chromosomes*. Each chromosome is a bit stream of cascaded and encoded variables. For a given *parent selection scheme* within the Reproduction operator, more suitable chromosomes are picked from the population for further genetic enhancements. A chromosome is *more suitable* if it results in a smaller error cost-function. Selected chromosomes are sent to the Crossover operator to create new chromosomes by exchanging their partial bit stream. These new chromosomes are called *offsprings*. The offsprings are then passed on to the Mutation operator for further manipulation, and then returned to the population. One completed cycle is called a generation.

2.1.1 Reproduction

Reproduction is an operation to select chromosomes (encoded variables) from the population according to the parent selection scheme. Selected chromosomes are then passed to the next genetic operator, Crossover. One well known parent selection scheme is called the Roulette Wheel parent selection [60]. The basic Roulette Wheel parent selection is taken from the game of roulette wheel. It assigns a higher probability to those chromosomes which are associated with relatively high *fitness values* among the population. The fitness value is a number associated with a chromosome's measure of *suitability*. Roulette Wheel parent selection involves the following four steps:

1. Add all the fitness values.
2. Randomly select a number between zero and the sum of fitness values.
3. Return the first chromosome for which the sum of all its preceding fitness values is greater than or equal to the randomly selected number.
4. Go back to Step 1 until N_{chrom} chromosomes are selected, where N_{chrom} is the population size.

In this thesis, Roulette Wheel parent selection is used in the algorithm presented in Appendix I.

2.1.2 Crossover

The Crossover operator creates offsprings by manipulating selected chromosomes. Sub-strings of the selected chromosomes from Reproduction are exchanged with their paired partner, as shown in Figure 3.

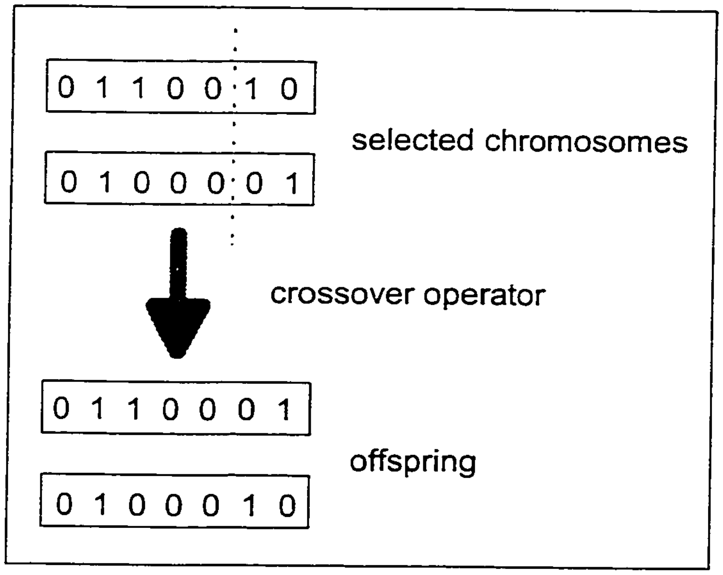

Fig.3: Crossover operator

In the above example, two chromosomes are chosen from a population. The right side of the dotted line shows the binary digits which participate in this process. By exchanging the last two digits, two new chromosomes, called offsprings, are created. Each of the offspring corresponds to a set of newly created variables. Crossover is implemented by the algorithm presented in Appendix I.

2.1.3 Mutation

Every bit contained in the offspring will be examined at this stage, and with a specified probability, certain digits will be altered from either '0' to '1' or '1' to '0' in binary encoding as shown below.

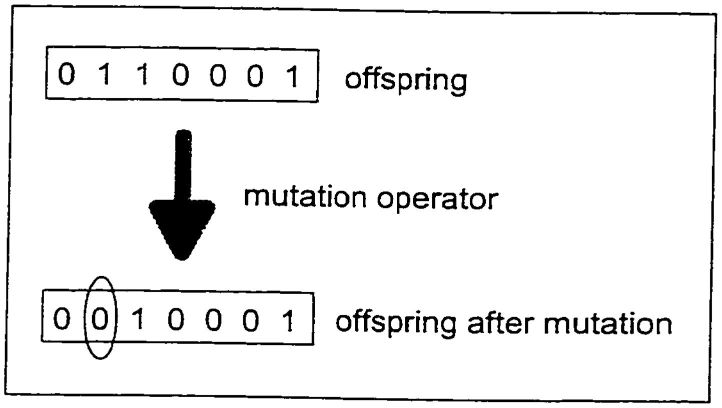

Fig.4: Mutation operator

The implementation algorithm of the Mutation operation is presented in Appendix I.

2.2 EXAMPLE OF FIR FILTER DESIGN WITH GA

A 4th order digital FIR filter with a linear phase characteristic is designed to illustrate how GA works. The design procedures are outlined as shown in the block diagram in Fig.5. It includes Initialization, Evaluation, and the three major operators (Reproduction, Crossover, and Mutation). Chromosomes are created at the initialization phase, evaluated and entered into the GA Cycle. This cycle will be terminated by a specified maximum number of iterations.

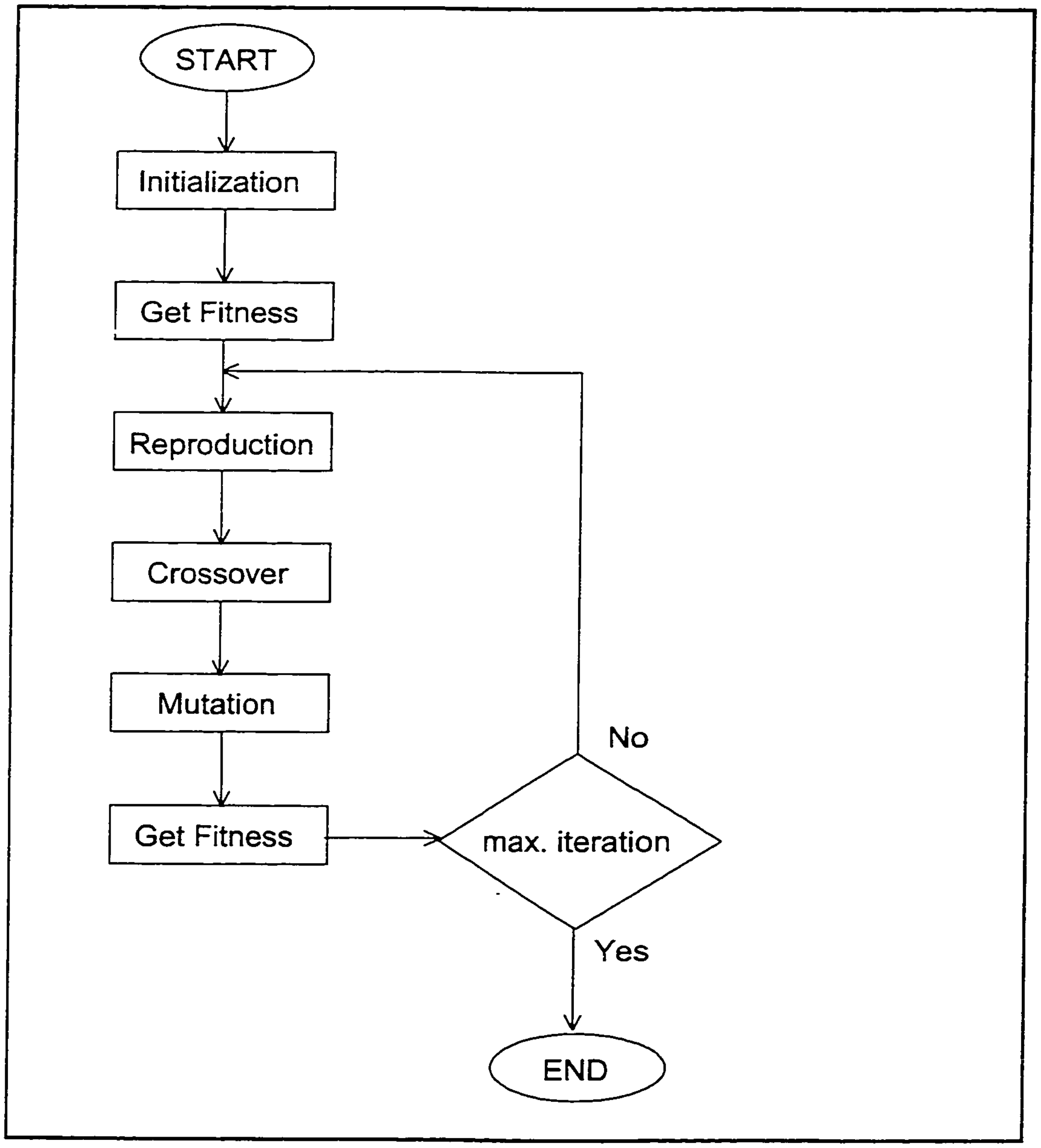

Fig.5: Block diagram for FIR filter design

Step 1. Initialization

A four-bit coefficient wordlength is chosen in this example. We modify the binary encoding for the purpose of simulation. The first digit is the sign bit. The second digit represents the number 2^{-0}. The third digit represents the number 2^{-1}, and the last digit represents the number 2^{-2}.

Binary representation of coefficient a = $[b_0\ b_1\ b_2\ b_3]$,

$$\text{and the real value of a} = \begin{cases} +\sum_{i=1}^{3} b_i \bullet 2^{i-1} & , b_0 = 0 \\ -\sum_{i=1}^{3} b_i \bullet 2^{i-1} & , b_0 = 1 \end{cases},$$

where b_i is either 0 or 1, when b_o=1 represents a negative number, when b_o=0 represent a positive number. For example, a filter contains the following coefficients,

[-0.25 -1.5 1.75 -1.5 -0.25].

A chromosome is created by first encoding them into binary strings, then cascading the encoded strings into a binary stream.

```
    [-0.25   -1.5   1.75   -1.5   -0.25]            <== filter coefficients
 => [1 0 0 1   1 1 1 0   0 1 1 1   1 1 1 0   1 0 0 1]   <== encoded coefficients
 => [1 0 0 1 1 1 1 0 0 1 1 1 1 1 1 0 1 0 0 1]          <==  a chromosome
```

Due to the even symmetry property of the filter coefficients, only three coefficients will be needed in the process.

```
    [-0.25   -1.5   1.75]            <== the first, second, and third coefficients
 => [1 0 0 1   1 1 1 0   0 1 1 1]    <== encoded coefficients
```

=> [1 0 0 1 1 1 1 0 0 1 1 1]. <== a reduced size of chromosome

The above example shows that every chromosome (and offspring) must contain exactly 12 bits. At the initialization phase, the random selection of 12 binary bits produces a chromosome. A population is the collection of these randomly selected chromosomes in Fig.6.

randomly generated chromosomes
(1) 1 0 0 1 1 0 1 1 1 0 1 1
(2) 0 0 0 1 1 0 1 0 0 0 1 0
(3) 0 1 0 0 1 0 0 0 1 1 0 0
(4) 1 0 0 1 0 1 1 1 1 0 0 0

Fig.6: Randomly select binary bits for a population of Four

Step 2. Get Fitness

Each chromosome is evaluated by the Get Fitness operator to derive a fitness value. This process involves the decoding of each chromosome into a set of continuous numbers.

	chromosomes		decoded coefficients
(1)	[1 0 0 1 1 0 1 1 1 0 1 1]	=	[-0.25 -0.75 -0.75]
(2)	[0 0 0 1 1 0 1 0 0 0 1 0]	=	[0.25 -0.5 0.5]
(3)	[0 1 0 0 1 0 0 0 1 1 0 0]	=	[-0.25 -0.0 -1.0]
(4)	[1 0 0 1 0 1 1 1 1 0 0 0]	=	[-0.25 1.75 -0.0]

Fig.7: Decode chromosomes into coefficients

Filter coefficients are obtained by expanding the decoded strings through the even-symmetry property.

	filter coefficients				
(1)	[-0.25	-0.75	-0.75	-0.75	-0.25]
(2)	[0.25	-0.5	0.5	-0.5	0.25]
(3)	[-0.25	-0.0	-1.0	-0.0	-0.25]
(4)	[-0.25	1.75	-0.0	1.75	-0.25]

Fig.8: Decoded filter coefficients

The filter coefficients are then evaluated to obtain their fitness values. The evaluation is accomplished by taking the reciprocal of the objective (error) function denoted by E_ℓ .

randomly generated chromosomes	fitness = $1/E_\ell$
(1) 1 0 0 1 1 0 1 1 1 0 1 1	0.34
(2) 0 0 0 1 1 0 1 0 0 0 1 0	0.22
(3) 0 1 0 0 1 0 0 0 1 1 0 0	0.02
(4) 1 0 0 1 0 1 1 1 1 0 0 0	0.76
	total = 1.34

Fig.9: Result of fitness evaluation

Step 3. Reproduction

Reproduction selects the chromosomes from the population according to a parent selection scheme. The Roulette Wheel parent selection scheme, presented in the previous section, will be used in this example. Its functionality includes the selection of *more suitable* chromosomes and passing them on to the next operator. This scheme will be repeated four times to obtain four selected chromosomes. The first step in the scheme is to sum up all the fitness values that are evaluated and shown to be 1.34.

In the second step, a random number is chosen between 0 and 1.34. In this example, a random number of 0.36 is chosen. The third step is to return the first chromosome if the sum of all the preceding fitness values is greater than or equal to the randomly selected number. We will now turn our attention to determining the sum of the preceding fitness values of each chromosome. The sum of the preceding fitness of each chromosome is evaluated and shown below.

chromosomes	sum of preceding fitness
(1) 1 0 0 1 1 0 1 1 1 0 1 1	0 + 0.34 = 0.34
(2) 0 0 0 1 1 0 1 0 0 0 1 0	0.34 + 0.22 = 0.56
(3) 0 1 0 0 1 0 0 0 1 1 0 0	0.56 + 0.02 = 0.58
(4) 1 0 0 1 0 1 1 1 1 0 0 0	0.58 + 0.76 = 1.34

Fig.10: Sum of preceding fitness values

Since 0.56 is the first number larger than the random number 0.36, the corresponding second chromosome is returned as a selected chromosome.

selected chromosomes
(2) 0 0 0 1 1 0 1 0 0 0 1 0

Fig.11: Select the second chromosome from the population

In order to maintain the population size of four, a replacement scheme is utilized. To obtain the second selected chromosome, we will repeat the proceeding Roulette Wheel parent selection scheme process. Thus, we need to generate another random number between 0 and 1.34. This time, lets assume the random number is 0.86. The sum of the preceding fitness values are evaluated below.

chromosomes	sum of preceding fitness
(1) 1 0 0 1 1 0 1 1 1 0 1 1	0 + 0.34 = 0.34
(2) 0 0 0 1 1 0 1 0 0 0 1 0	0.34 + 0.22 = 0.56
(3) 0 1 0 0 1 0 0 0 1 1 0 0	0.56 + 0.02 = 0.58
(4) 1 0 0 1 0 1 1 1 1 0 0 0	0.58 + 0.76 = 1.34

Fig.12: Sum of preceding fitness values

Since 1.34 is the first number larger than the random number 0.86, the fourth chromosome is chosen from the population and added to the 'selected chromosomes'.

selected chromosomes
(2) 0 0 0 1 1 0 1 0 0 0 1 0
(4) 1 0 0 1 0 1 1 1 1 0 0 0

Fig.13: Select the fourth chromosome from the population

We will repeat the Roulette Wheel parent selection procedure again. The sum of the fitness values in the population remains 1.34 with the chromosome replacement technique. Once again, we need to generate another random number between 0 and 1.34. This time, the random number is 0.27. The sum of the preceding fitness values are evaluated below.

chromosomes	sum of preceding fitness
(1) 1 0 0 1 1 0 1 1 1 0 1 1	0 + 0.34 = 0.34
(2) 0 0 0 1 1 0 1 0 0 0 1 0	0.34 + 0.22 = 0.56
(3) 0 1 0 0 1 0 0 0 1 1 0 0	0.56 + 0.02 = 0.58
(4) 1 0 0 1 0 1 1 1 1 0 0 0	0.58 + 0.76 = 1.34

Fig.14: Sum of preceding fitness values

In this case, the first chromosome is chosen from the population and it becomes the third 'selected chromosome' as shown below.

selected chromosomes
(2) 0 0 0 1 1 0 1 0 0 0 1 0
(4) 1 0 0 1 0 1 1 1 1 0 0 0
(1) 1 0 0 1 1 0 1 1 1 0 1 1

Fig.15: Select the first chromosome from the population

At this point, there is only one chromosome left to select. Again, the sum of fitness in the population remains 1.34 using the chromosome replacement scheme. A random number between 0 and 1.34 is generated. Let's assume the random number is 0.79. The sum of the preceding fitness values are evaluated below.

chromosomes	sum of preceding fitness
(1) 1 0 0 1 1 0 1 1 1 0 1 1	0 + 0.34 = 0.34
(2) 0 0 0 1 1 0 1 0 0 0 1 0	0.34 + 0.22 = 0.56
(3) 0 1 0 0 1 0 0 0 1 1 0 0	0.56 + 0.02 = 0.58
(4) 1 0 0 1 0 1 1 1 1 0 0 0	0.58 + 0.76 = 1.34

Fig.16: Sum of preceding fitness values

In this case, the fourth chromosome is chosen from the population. This completes the reproduction process. The four selected chromosomes are shown below. Note that chromosome number (4) has been chosen twice. This is the effect of using replacement scheme, such that a chromosome can be chosen more than once. The repeatedly chosen chromosomes will replace the poor fitness chromosomes in a population.

```
    selected chromosomes
(2) 0 0 0 1 1 0 1 0 0 0 1 0
(4) 1 0 0 1 0 1 1 1 1 0 0 0
(1) 1 0 0 1 1 0 1 1 1 0 1 1
(4) 1 0 0 1 0 1 1 1 1 0 0 0
```

Fig.17: Select the fourth chromosome again.

Step 4. Crossover

Every crossover operation requires a pair of chromosomes. Therefore, the four selected chromosomes are separated into two groups,.and each group contains two chromosomes. The two groups of crossover operations are illustrated below.

```
      selected chromosomes
(2)   0 0 0 1 1 0 1 0 0 : 0 1 0
(4)   1 0 0 1 0 1 1 1 1 : 0 0 0
               ⇓
      0 0 0 1 1 0 1 0 0 : 0 0 0
      1 0 0 1 0 1 1 1 1 : 0 1 0
           offsprings
```

Fig.18a: Perform Crossover for the first pair of selected chromosomes

```
      selected chromosomes
(1)   1 0 0 1 1 : 0 1 1 1 0 1 1
(4)   1 0 0 1 0 : 1 1 1 1 0 0 0
               ⇓
      1 0 0 1 1 : 1 1 1 1 0 0 0
      1 0 0 1 0 : 0 1 1 1 0 1 1
           offsprings
```

Fig.18b: Perform Crossover for the second pair of selected chromosomes

During the crossover operation, a cut-off point is randomly selected to separate each chromosome into two pieces. This is shown by the dotted line. A piece of each chromosome will then exchange with another one in its group to create offsprings. Four new offsprings are given a new label from one to four, and are shown in the following table.

	offspring
(1)	0 0 0 1 1 0 1 0 0 0 0 0
(2)	1 0 0 1 0 1 1 1 1 0 1 0
(3)	1 0 0 1 1 1 1 1 1 0 0 0
(4)	1 0 0 1 0 0 1 1 1 0 1 1

Fig.19: The resultant offspring

Step 5. Mutation

Every bit in the offsprings is examined. Under a pre-specified probability, certain bits are flipped from '1' to '0' or '0' to '1' which is indicated by the bubbles in the diagram below.

	offspring after mutation
(1)	0 0 0 1 1 0 1 0 $\textcircled{1}$ 0 0 0
(2)	1 0 0 1 0 1 1 1 1 0 1 0
(3)	1 0 0 1 $\textcircled{0}$ 1 1 1 1 0 0 0
(4)	1 0 0 $\textcircled{0}$ 0 0 1 1 1 0 1 1

Fig.20: The effect of Mutation

Step 6. Get Fitness

The output of the Mutation operator will be evaluated by the Get Fitness operator. First, offsprings are decoded into continuous numbers as illustrated at the beginning of this section.

	offsprings			decoded coefficients		
(1)	[0 0 0 1	1 0 1 0	1 0 0 0]	= [0.25	-0.5	-0.0]
(2)	[1 0 0 1	0 1 1 1	1 0 1 0]	= [-0.25	1.75	-0.5]
(3)	[1 0 0 1	0 1 1 1	1 0 0 0]	= [-0.25	1.75	-0.0]
(4)	[1 0 0 0	0 0 1 1	1 0 1 1]	= [-0.0	0.75	-0.75]

Fig.21: Decode offspring into filter coefficients

To obtain the corresponding filter coefficients, the decoded strings are expanded using the property of even-symmetry.

	filter coefficients				
(1)	[0.25	-0.5	0	-0.5	0.25]
(2)	[-0.25	1.75	-0.5	1.75	-0.25]
(3)	[-0.25	1.75	0	1.75	-0.25]
(4)	[0	0.75	-0.75	0.75	0]

Fig.22: The resultant filter coefficients

The filter coefficients are then evaluated to obtain their fitness values. The evaluation is accomplished by taking the reciprocal of the objective function. Assume the values are evaluated and shown below with each corresponding offspring.

offsprings	fitness = $1/E_t$
(1) 0 0 0 1 1 0 1 0 1 0 0 0	0.24
(2) 1 0 0 1 0 1 1 1 1 0 1 0	1.62
(3) 1 0 0 1 0 1 1 1 1 0 0 0	0.52
(4) 1 0 0 0 0 0 1 1 1 0 1 1	1.18

Fig. 23: Fitness evaluation of offspring

Up to this point, we have completed one iteration (also known as one generation) of GA operations. The above offsprings with their fitness values are the new population for the next iteration that replaces the current population. In other words, the Roulette Wheel parent selection will choose chromosomes among these four offsprings and the previous population will be discarded. To perform subsequent iterations, we need to go back to Step 3 and go through the Reproduction process again.

When the GA reaches its final iteration, we obtain a population of four chromosomes. In this example, assuming N iterations have been processed and the following population results,

offsprings	fitness = $1/E_t$
(1) 1 0 0 1 0 1 1 0 1 0 1 1	2.19
(2) 1 0 0 1 0 1 1 1 1 0 1 0	1.62
(3) 1 0 0 0 0 1 1 1 1 0 0 0	1.93
(4) 1 0 0 1 0 1 1 1 0 0 1 0	2.01

Fig.24: The resulting coefficients after N iterations

Within this population, the first offspring contains the highest fitness value which means it has the least mean square error. Thus, the first set is the solution. This set will be decoded into real coefficients,

(3) [1 0 0 1 0 1 1 0 1 0 1 1] = [-0.25 1.5 -0.75].

The resultant filter coefficients of this example are [-0.25 1.5 -0.75 1.5 -0.25].

In the following step, we will convert the binary representation into Canonic Signed-Digit (CSD) coefficients. The conversion procedure can be found in [24]. Define b_i and C_i to be the binary and carry-in bits located at 'i'th position in the string. Let 'D' be the Canonic Signed-Digit coefficient which is defined as follow,

$$D = [d_0\ d_1\ d_2\ d_3],$$

where d_i is either {-1, 0, 1}, and the symbol for -1 is $\bar{1}$. The conversion involves the following steps:

1. Start with the least significant bit by setting the index i = 0 and initialize $C_0 = 0$. The sign bit of the binary representation will be ignored and treated it as if a positive number.
2. Examine two adjacent bits, b_{i+1} and b_i, and the carry-in C_i. If any of the two consist of a '1' into them, the carry-in next, C_{i+1}, is assigned to be a '1'.
3. Generate d_i by evaluating the equation, $d_i = b_i + C_i - 2C_{i+1}$.
4. Go back to step 2 until all the bits are completely converted.
5. If the binary representation is a negative value, invert the sign of all non-zero digits in the Signed-Digit representation.

In our example, the binary representation is [1 0 0 1 0 1 1 0 1 0 1 1]. We take the first coefficient, [1 0 0 1], and convert it into CSD coefficient. We illustrate the conversion process by using the following table. The least significant bit is at the top row. C_i and C_{i-1} are both initialized to '0' at i = 0.

i	b_{i-1}	b_i	C_i	C_{i-1}	d_i
0	0	1	0	0	1
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0

Table 1a: Transfer the first binary coefficient into CSD format

The result of conversion can be read off from the last column, D = [0 0 0 1]. Since the binary representation of this number is negative, all the non-zero digits must be inverted. The Signed-Digit representation of the first coefficient is then equal to [0 0 0 $\bar{1}$], where the least significant ternary digit, $\bar{1}$, is -2^{-2}. By going through a similar procedure the table for the second coefficeint is shown below.

i	b_{i-1}	b_i	C_i	C_{i-1}	d_i
0	1	0	0	0	0
1	1	1	0	1	$\bar{1}$
2	0	1	1	1	0
3	0	0	1	0	1

Table 1b: Transfer the second binary coefficient into CSD format

The result of the conversion is D = [1 0 $\bar{1}$ 0], where the most significant ternary digit, '1', corresponds to a 2^1, and the third digit, $\bar{1}$, is -2^{-1}. The table for the third coefficient, [1 0 1 1], is shown below.

i	b_{i-1}	b_i	C_i	C_{i-1}	d_i
0	1	1	0	1	$\bar{1}$
1	0	1	1	1	0
2	0	0	1	0	1
3	0	0	0	0	0

Table 1c: Transfer the third binary coefficient into CSD format

The outcome for the third coefficient is D = [0 1 0$\bar{1}$]. Since the binary representation of this number is negative, its Signed-Digit representation is [0 $\bar{1}$ 0 1]], where the $\bar{1}$ digit is a -2^0, and the ternary digit, '1', corresponds to a 2^{-2}. The CSD filter coefficients are equal to

$$[0\,0\,0\,\bar{1} \quad 1\,0\,\bar{1}\,0 \quad 0\,\bar{1}\,0\,1\,],$$

which is equal to [-0.25 1.5 -0.75] in continuous values.

Although there exist algorithms (e.g., [24]) to convert binary strings into CSD coefficients, the number of non-zero digits cannot be predicted nor restricted at the beginning of filter design. This problem may increase the use of hardware resources that may violate the original system specification due to unsatisfactory filter performance.

2.3 FUNDAMENTAL THEOREM OF GENETIC ALGORITHM

The mathematical model of GA provides a deeper understanding of how it behaves for each of these GA operations. The model will also provide important guidelines for choosing the mutation rate (as well as crossover rate) and develop our own parent selection scheme.

In this derivation, the GA is based on a typical chromosome referred to as a *schema* [21]. A schema (a stream) is a combination of binary digits and the symbol of '#', where '#' is the *don't care* symbol which represents either of the of the valid digits, '0' or '1', in binary encoding. For example, the following eight chromosomes belong to the same schema.

schema S = [1 # # 1 0 0 # 1]	chromosomes
	[1 1 1 1 0 0 0 1]
	[1 0 1 1 0 0 0 1]
	[1 1 1 1 0 0 1 1]
	[1 0 1 1 0 0 1 1]
	[1 1 0 1 0 0 0 1]
	[1 0 0 1 0 0 0 1]
	[1 1 0 1 0 0 1 1]
	[1 0 0 1 0 0 1 1]

A schema represents a subset of the total search spaces in the population. A group of schema will form a complete population which spans various solution spaces. On the other hand, if a particular schema does not exist in the population, those spaces will not be explored. Therefore, one of the major tasks in the GA is to create different schema to search for a global solution.

2.3.1 Genetic Algorithm Model

The mathematical model of GA separates into three phases, relative to the three genetic operators, Reproduction, Crossover, and Mutation. Each of these operators will be defined by an equation. The combination of these equations results in a full GA. With this model, we are able to predict schema in the population for the next generation which leads to an optimized solution.

Reproduction operator mathematical model

Let H(S$_i$, t) be the expected number of chromosomes belonging to the schema, S$_i$, at 't' generations, where 'i' is an index for a typical schema. The process of reproduction is to determine its effectiveness over the expected count that would allow us to predict the search space from one iteration to the next. The probability of chromosome 'i' (chromosome which belongs to S$_i$) is equal to

$$p_i = f(S_i, t) / \Sigma f_j \,,$$

where $f(S_i, t)$ is defined as the average fitness of schema S$_i$, and Σf_j represents the total fitness value within the population. For a population size of N_{chrom}, there is H$_r$(S$_i$, t +1) number of representatives of schema S$_i$ in the population at time (t + 1). The relationship of the expected count can be written as

$$H_r(S_i, t+1) = H(S_i, t) \times N_{chrom} \frac{f(S_i, t)}{\Sigma f_j}, \tag{2.1}$$

where $\frac{\Sigma f_j}{N} = f(t)$ is the average fitness of all chromosomes in the population. The above equation is simplified to

$$H_r(S_i, t+1) = H(S_i, t) \times \frac{f(S_i, t)}{f(t)}. \tag{2.2}$$

The average fitness of a schema is evaluated by summing up all its representatives then dividing it by the total number in the current population. Equation (2.2) shows that the schema S$_i$ receives an increment in the expected count for

$f(S_i,t)/f(t) > 1$. On the other hand, below average schema, $f(S_i,t)/f(t) < 1$, will receive a decrement in the expected count in the population.

Crossover operator mathematical model

The next derivation is the Crossover operator. In order to derive the Crossover operator, we must first determine the *length* of the schema, $L(S_i)$. This length is defined as the distance between two outer most valid digits (not including the *don't care* symbol). For example, $L(S_1) = 3$ and $L(S_2) = 1$, where S_1 = [# # 1 0 # 1 # # #] and S_2 = [# # # 0 1 # #]. Let 'p_c' and 'T_d' be the crossover rate and the total number of digits in a chromosome respectively. The crossover rate is defined as the probability of a chromosome that will undergo the crossover process. The utilization of a crossover rate prevents all chromosomes from undergoing crossover. This will decrease the chance of narrowing down to the same schema in a population. A random number will be generated for each selected chromosome from Reproduction. If the random number is smaller than the crossover rate, then the corresponding chromosome will pair up and perform crossover as described in the previous section. On the other hand, if the random number is larger than the specified crossover rate, the corresponding chromosome will not perform crossover and will be assigned as an offspring.

The probability of picking a cut-off point is equal to $L(S_i)/(T_d - 1)$. When the crossover rate is factored by the preceding ratio, its probability becomes

$(p_c)(L(S_i))/(T_d - 1)$.

Let $P_c(S_i)$ be the probability of the survival schema under crossover operation. Thus, one can establish the following relationship:

$$P_c(S_i) \approx 1 - p_c \frac{L(S_i)}{T_d - 1}, \tag{2.3}$$

The above Equation (2.3) shows that the schema with the longer length is more likely to be destroyed during crossover operation. In other words, the cut-off point is more likely to separate the constructed patterns.

Mathematical model for Mutation operator

We introduce a new term, *schema order* (O(S)), to develop the derivation of Mutation operator. The order of a schema is defined as the number of valid digits present in a given schema other than the *don't care* symbols, for example, $O(S_3) = 3$ and $O(S_4) = 1$, where S_3 = [# # 1 1 # # 0], and S_4 = [# # # 1 # #]. Let 'p_m' be the given probability of the mutation rate. The mutation rate is defined as the probability that a digit will not retain its value. Since the mutation operation is performed on every digit, the probability of a bit being unchanged is equal to $(1 - p_m)$. Let $P_m(S_i)$ be the probability of the survival of the schema under the mutation operation. Therefore, the chances of the schema remaining unchanged can be established by the following relationship:

$$\begin{aligned} P_m(S_i) &\approx (1 - p_m)^{O(S_i)} \\ &\approx 1 - p_m \cdot O(S_i), \text{ for } p_m \ll 1. \end{aligned} \tag{2.4}$$

Usually, designers like to keep the mutation rate small ($p_m << 1$) in order to establish a stable pattern. Equation (2.4) shows that the higher the order of a schema, the more likely it will not be retained under the mutation operation.

GA mathematical model

The mathematical model of a GA is obtained by

$$H(S_i,t+1) = H_r(S_i,t+1) \times P_c(S_i) \times P_m(S_i)$$

$$= H(S_i,t)\frac{f(S_i,t)}{f(t)} \times \left(1 - p_c\frac{L(S_i)}{T_d-1}\right) \times (1 - p_m \cdot O(S_i))$$

$$= H(S_i,t)\frac{f(S_i,t)}{f(t)} \times \left[1 - p_c\frac{L(S_i)}{T_d-1} - p_m \cdot O(S_i) + p_c p_m \frac{L(S_i)}{T_d-1}O(S_i)\right].$$

Assuming $p_c p_m \frac{L(S_i)}{T_d-1}O(S_i) << 1$, and can be neglected. $H(S_i,t+1)$ is then approximated to be

$$H(S_i,t+1) \approx H(S_i,t)\frac{f(S_i,t)}{f(S)} \times \left[1 - p_c\frac{L(S_i)}{T_d-1} - p_m \cdot O(S_i)\right]. \qquad (2.5)$$

Equation (2.5) shows that a short, low order and above average schema will receive an exponential increment of chromosomes [21]. Such a schema is used to design encoding scheme and parent selection scheme in the next sections. Equation (2.5) is simply referred to as Schema Theorem or the Fundamental Theorem of Genetic Algorithms. The probabilities of crossover and mutation are user defined values that are between zero and one. Small mutation rates have the advantage of exploring other solution spaces which are not covered by the population throughout the generations and without disturbing the established pattern.

2.4 SUMMARY

The Genetic Algorithm is an optimization scheme. Search points can be escaped from capturing by local minima (or maxima), and the characteristic equation does not need to be differentiable. The basic Genetic Algorithm involves three operators, reproduction, crossover, and mutation. Each of these operators has significant effect in optimizing a solution. For those schema which are short, low order, as well as above average, they will receive an exponential increment of chromosomes over the iterations.

CHAPTER 3. GENETIC ALGORITHM AND ITS APPLICATION IN 1-D FIR FILTER DESIGN WITH CSD COEFFICIENTS

3. MODIFICATION OF GA STRUCTURE

In this chapter, our primary focus is to present a 1-D FIR filter design techniques. The propose methodologies will produce Canonic Signed-Digit (CSD) coefficients with the use of Genetic Algorithms [34].

3.1 Modified Block Diagram

In order to produce CSD coefficients, a modified GA structure is used to compensate for the Canonic Signed-Digit constraints. These constraints are imposed by the filter coefficients, and are explained as follows:

1. The number of non-zero digits within a coefficient must be less than or equal to a prescribed number, N_{digits}.
2. The digit adjacent to a non-zero digit must be a '0'.

A modified GA structure is shown in Fig.25. The block diagram consists of the basic genetic operators (Reproduction, Crossover, and Mutation), as well as three additional components (CSD Check, Maximum Failure, and Reset Coefficient) outlined by the dotted line. These

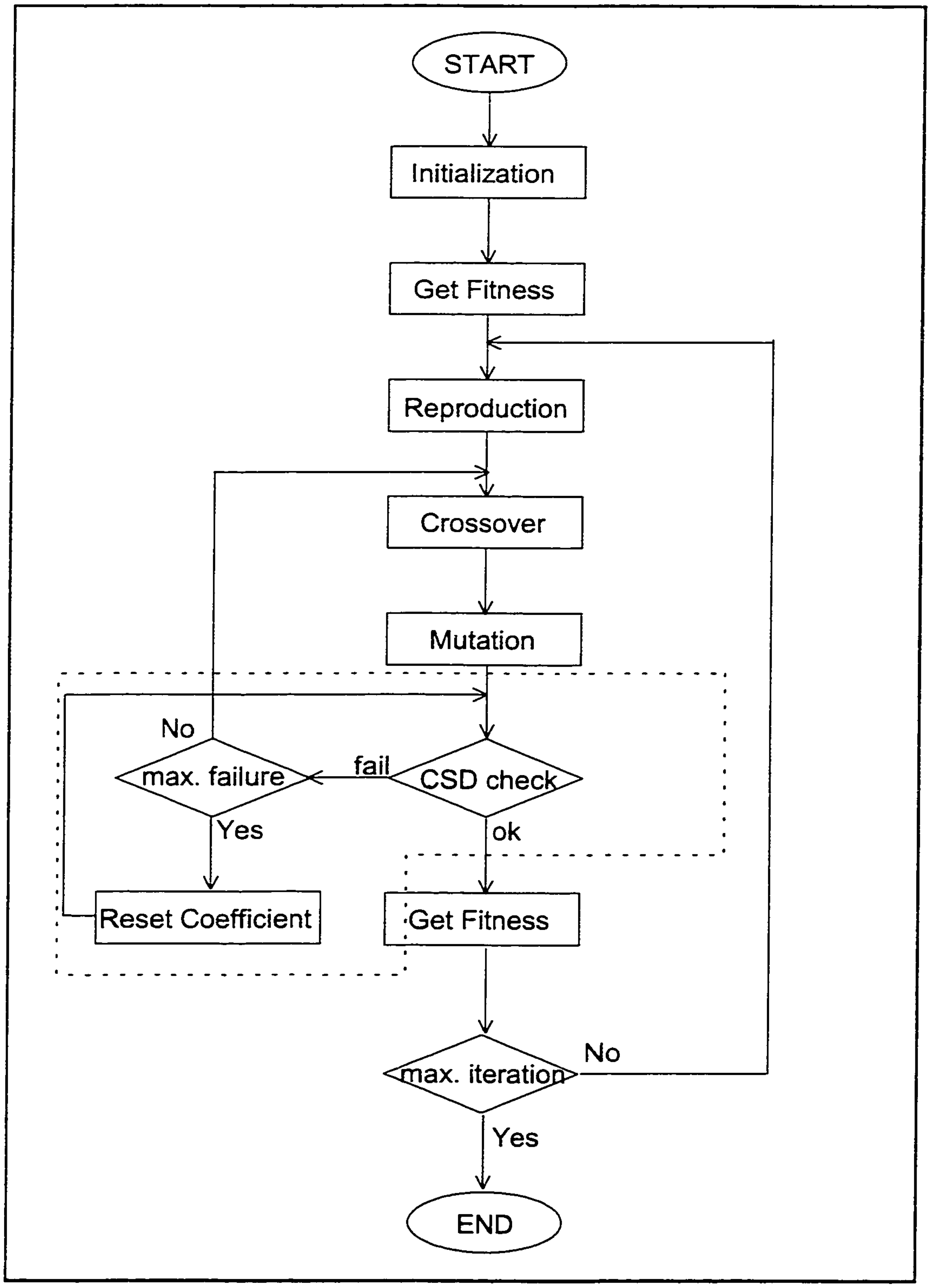

Fig.25: Modified Genetic Algorithm Block Diagram

additional components, to be explained next, will guarantee filter coefficients to be of CSD format.

Canonic Signed-Digit Check (CSD Check)

Each newly created bit stream (offspring) is examined to see if the individual coefficients satisfy the above CSD constraints. Any violation will cause the bit stream (offspring) to be considered as 'failed'. The implementation involves two phases. The first phase is to count the number of non-zero digits presented in each coefficient. The second phase is to determine if the digit adjacent to a non-zero digit is a zero. If the number of non-zero digits is greater than a specified number or the digit adjacent to a non-zero digit is not a zero, it will be considered as failed in either case. The algorithm for CSD Check is presented in Appendix I.

Maximum Failure (Max. Failure)

A specified maximum number of failures at 'CSD Check', N_{max} , ensures that individual coefficients can 'fail' no more than N_{max} times. If one of the individual coefficients has failed N_{max} times, it is directed to the 'Reset Coefficient' operator in Fig.25. Otherwise, this pair of coefficients will restore their previous values, and go through crossover and mutation again. The algorithm for Maximum Failure of a coefficient is presented in Appendix I.

Reset Coefficient

The incoming coefficient will be disregarded, and replaced by a randomly generated non-zero digit according to its encoding scheme. An algorithm is shown in Appendix I for Signed-Digit encoding scheme.

This modified GA structure ensures the identification of CSD coefficients and facilitates the exploration of new search spaces. Whenever a typical coefficient has failed to satisfy the restrictions for a certain number of times, that coefficient will be reset to derive a new solution space.

3.2 ALGORITHM ENHANCEMENT

As we have seen in the example in Section 2.2, there are three major decisions to be made in designing digital FIR filters with GA. The first one is the choice of an encoding scheme. The utilization of an inappropriate encoding scheme will cause a longer computational time, and the resultant solution may not be optimized. The rule of thumb is that the encoded stream must be well represented in the problem in a way that enhances the GA to explore new solution spaces. In addition, the length of the chromosome should be kept as small as possible, since shorter chromosomes tend to have faster convergence rate. The second decision is to choose a parent selection scheme. Selection of a parent scheme affects the convergence rate and the final solutions. A population that is slowly converging, does not always generate better solutions. The last decision is the choice of fitness evaluation: how to evaluate a chromosome. A poorly defined objective function will generate an inferior solution. Furthermore, this solution may not satisfy the design specification.

Recall that our second goal is to enhance the algorithm performance in digital filter design. In the following sections, we will discuss the encoding scheme and parent selection scheme for filter design with CSD coefficients. Filter coefficient evaluation will be left for the well known LMS and Minimax strategies.

3.2.1 Ranking Parent Selection

At the beginning of an iteration, chromosomes (filter coefficients) are generated randomly. As a result, their evaluated fitness values are usually low. A parent selection scheme, named Ranking, is developed to enhance the convergence rate. Under this selection scheme, chromosomes with similar fitness values are grouped in pairs. This may enhance the overall convergence rate at the beginning of the iterations. However, the convergence rate decreases as the process approaches the solution. This scheme involves the following two steps.

1. Rank the chromosomes in descending order according to their fitness values, e.g.. [chrom1 chrom2 chrom3 chrom4 ... chromN-1 chromN].
2. Group the structurally similar chromosomes in pairs, e.g., [(chrom1, chrom2), (chrom3, chrom4), ..., (chromN-1, chromN)], and pass them over to the crossover operation.

The Ranking parent selection aims only at a local minimum search rather than a global solution. Since GAs are stochastic search techniques, more than one trial is necessary to obtain better solutions. The Ranking parent selection scheme is implemented by the algorithm presented in Appendix I.

3.2.2 Mixed Encoding

In this section, we propose another encoding scheme, namely Mixed encoding for the design of CSD coefficient filter. The assignment of bits involves ternary digits and binary strings. The sign is represented by a signed digit, then it is followed by a binary string. Therefore, each non-zero digit from Signed-Digit encoding is re-encoded by a ternary sign bit followed by a binary representation of the non-zero digit position. The following example represents a filter coefficient wordlength of eight bits with the maximum of two non-zero digits,

$$\text{representation} = [t_{as}\ b_{a0}\ b_{a1}\ b_{a2} \qquad t_{bs}\ b_{b0}\ b_{b1}\ b_{b2}\ ].$$

$$\text{where } a = \sum_{i=0}^{2} b_{ai} \times 2^{i},$$

$$b = \sum_{i=0}^{2} b_{bi} \times 2^{i},$$

$$\text{sign}_a = t_{as},\ \text{sign}_b = t_{bs},$$

$$\text{coefficient} = \text{sign}_a 2^{-a} + \text{sign}_b 2^{-b}\ .$$

For example, consider the power-of-two numbers, $k_1 \times 2^{m1}$, and $k_2 \times 2^{m2}$. In the Mixed encoding scheme, the signed digit 'k_i', which is 1, 0, or -1, is followed by a binary representation of 'm_i'.

$$\text{coefficient } 0.421875 \Rightarrow 2^{-1} - 2^{-4}$$

$$\Rightarrow [\ 1\ 1\ 0\ 0 \qquad \bar{1}\ 0\ 0\ 1\ ]$$

In the above example, a filter coefficient is written as the sum of the power-of-two numbers, 2^{-1}, and -2^{-4}. For illustration purposes, -2^{-4} corresponds to the last four digits in which the sign and the exponent are represented by a $[\bar{1}]$, and a binary string of [0 0 1] (with the LSB on the left hand-side).

A [0] signed-digit implies a '0' digit of filter coefficient at the location corresponding to the binary stream representation. The wordlength of a filter coefficient is bounded by the maximum value of the exponent, $(m_i + 1)$. In the example, three-bit binary string can be used to encode the filter coefficient for maximum wordlength of 8 bits.

Compared to Signed-Digit encoding, an 11th order FIR filter with a 12bit wordlength, containing the maximum of two non-zero digits, the Signed-Digit encoding will result in a 12-bit Signed-Digit coefficient, and is 144 bits long for each chromosome. On the other hand, mixed encoding turns out to be 10 bits in length for each coefficient, and 110 bits long for each chromosome. As the order of the filter increases, the number of bits saved also increases.

3.2.3 Multi-Points Crossover

Single point crossover is a commonly used Crossover operator in GA as demonstrated in the example. For digital filter design, it may be better to use Multi-Points Crossover. In this operation, more than one cut-off point will be chosen in the same chromosome. This has the advantage of speeding up the convergence rate [21] and allowing the GA to actively search for new combinations of solution spaces. It is recommended to perform crossover on every coefficients. The crossover rate will be redefined towards the coefficient, instead of each selected chromosome. In this process, every pair of coefficients may go through crossover that depends on the crossover rate and the randomly generated number. To illustrate, the following two chromosomes are selected to perform crossover, designing a filter with five bits wordlength in binary encoding.

```
chrom1 = [ 0 1 0 1 1 1 1 0 0 1 0 0 0 1 1 0 0 0 1 0 1 1 1 1 1 0 1 1 0 0 ],
chrom2 = [ 0 0 1 1 1 0 0 1 0 0 0 0 1 0 0 1 1 1 1 1 0 0 1 0 1 1 1 0 1 1 ].
```

Fig.26: Selected chromosomes for crossover operation

The coefficient can be identified from the chromosomes by randomly.

```
chrom1 = [ 0 1 0 1 1 1   1 0 0 1 0 0   0 1 1 0 0 0   1 0 1 1 1 1   1 0 1 1 0 0 ],
chrom2 = [ 0 0 1 1 1 0   0 1 0 0 0 0   1 0 0 1 1 1   1 1 0 0 1 0   1 1 1 0 1 1 ].
```

Fig.27: Identify the corresponding filter coefficients

Randomly selecting a number between zero and one for each pair[1] of coefficients to see if it is less than the crossover rate. If it is less than the crossover rate, crossover will be performed on the pair of coefficients, otherwise the coefficients will be transferred to the offsprings without undergoing crossover. In our example, there are five pairs of coefficients, thus, we need to generate five random numbers. Assume the five random numbers are generated as below.

Random number = [0.43 0.67 0.82 0.36 0.91].

The first, second, and fourth numbers are smaller than the crossover rate (assume the crossover rate[2] is pre-specified to be 0.7). Therefore, only these three pairs of coefficients will undergo

[1] Coefficients are paired between two chromosomes: The first coefficient in chrom1 is grouped with the first coefficient in chrom2. The second coefficient in chrom1 is grouped with the second coefficient in chrom2, and so on.
[2] Crossover rate is assigned by a GA designer. Its value tends to be closer to 1.0 that allows the maximum spread of information between chromosomes. Assigning crossover rate to1.0 is not preferred to prevent manipulating the same set of chromosomes throughout iterations.

crossover. Next, cut-off points are randomly selected on these coefficients. The crossover points are randomly selected and indicated by the dotted lines.

chrom1 = [0 1 0 1 : 1 1   1 : 0 0 1 0 0   0 1 1 0 0 0   1 0 : 1 1 1 1   1 0 1 1 0 0],
chrom2 = [0 0 1 1 : 1 0   0 : 1 0 0 0 0   1 0 0 1 1 1   1 1 : 0 0 1 0   1 1 1 0 1 1].

Fig.28: Randomly select the cutting points

After performing the crossover operation, chromosomes are renamed as offsprings, as shown in the diagram below, Fig.29.

offspring1 = [0 1 0 1 1 0   0 0 0 1 0 0   0 1 1 0 0 0   1 1 1 1 1 1   1 0 1 1 0 0],
offspring2 = [0 0 1 1 1 1   1 1 0 0 0 0   1 0 0 1 1 1   1 0 0 0 1 0   1 1 1 0 1 1].

Fig.29: The result of Multi-points Crossover in each coefficient

The resultant offsprings of Multi-Points Crossover is shown in Fig. 30.

offspring1 = [0 1 0 1 1 0 0 0 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 1 0 1 1 0 0],
offspring2 = [0 0 1 1 1 1 1 1 0 0 0 0 1 0 0 1 1 1 1 0 0 0 1 0 1 1 1 0 1 1].

Fig.30: The resultant offspring after Multi-point Crossover

3.2.4 Fitness Scaling

At the beginning of the iteration, randomly generated chromosomes usually produce low fitness values. Scaling the fitness values can stretch out their differences and identify better chromosomes. Another advantage of fitness scaling is that it can prevent the elimination of schema with extremely low fitness values due to the large difference in fitness values. It is desirable to keep some low fitness schema in the population since they may contain some essential patterns hiding in them. The procedure of fitness scaling involves the normalization of chromosomes at each generation and scaling them to the integer space within one to twenty. The equation of Fitness Scaling is

$$\text{Scaled fitness} = \left\lceil 20 \cdot \frac{f_i}{\max(\text{population})} \right\rceil, \tag{2.6}$$

where ‘f_i’ is the fitness value, max(*) is a function that returns the maximum fitness value in the population, ‘*’.

3.2.5 Non-Uniform Mutation and Crossover Rates

In a lowpass FIR filter, the majority of the coefficients are very small similar to a sinc function. Assume Multi-Points Crossover is used, computational time will be wasted if manipulation only occurs on the Most Significant Digit (MSD) of the coefficients. Therefore, it may be more appropriate to assign non-uniform mutation and crossover rates during GA operations. Non-uniform rate refers to the dynamic probability of flipping digits (mutation) and picking cut-off points (crossover) as the iteration proceeds.

In term of Signed-Digit encoding scheme, the non-uniform mutation rate is associated with each ternary digit in the coefficient. This mutation rate dynamically changes with each generation according to the position of the digits. For example, all ternary digits receive the same mutation rate at the beginning of each iteration. But, with each succeeding generation, the exponential mutation rate (within a coefficient) shifts from the most significant digit to the least significant digit. This may facilitate the search for MSD at the beginning of the iteration by assigning an equal mutation rate to those digits for a period of time. Once the MSD pattern has been established, without disturbing the most significant digits, a fine tuning of the Least Significant Digits (LSD) is desirable. The algorithm for a non-uniform rate is outlined in Appendix I.

For example, the function of the non-uniform rate is implemented with two different alpha parameters, 0.5 and 2.0. The maximum mutation rate and encoded coefficient length is 0.1 and 8, respectively. The non-uniform rate algorithm allows one to define the mutation rate value for each digit in a coefficient string.

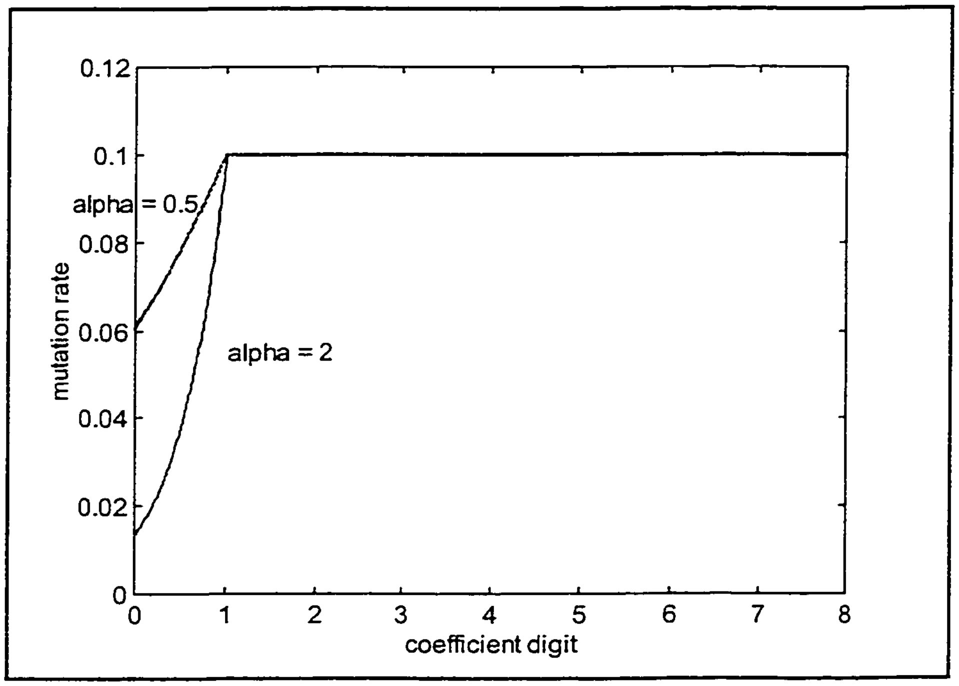

Fig.31a: Non-uniform mutation rate (completed 1/8 of iterations)

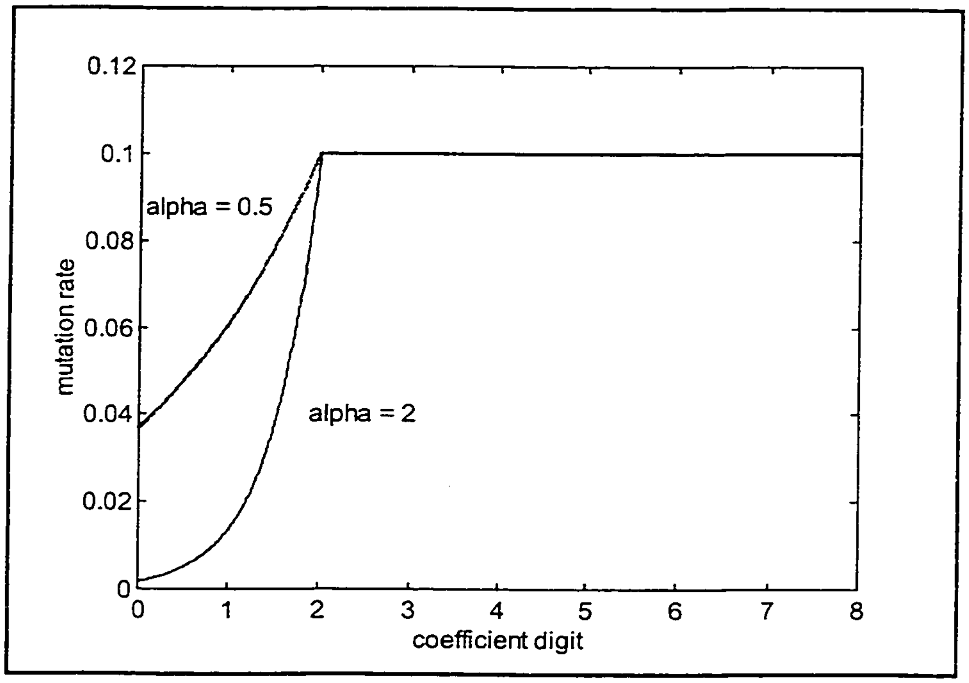

Fig.31b: Non-uniform mutation rate (completed 1/4 of iterations)

To illustrate the application of the non-uniform rate algorithm, consider Figure 31b. In this coefficient string, 0 is the most significant digit. As you move along the x-axis towards the right, the digits are known as the least significant digits. At the second digit of this specific coefficient string, the mutation rate is approximately 0.015 for alpha = 2. On the other hand, at the third digit, the mutation rate is 0.09. To illustrate the exponential function shift of the mutation rate in successive generations for a given coefficient string, the following four graphs (Figure 31c through 31f) have been provided.

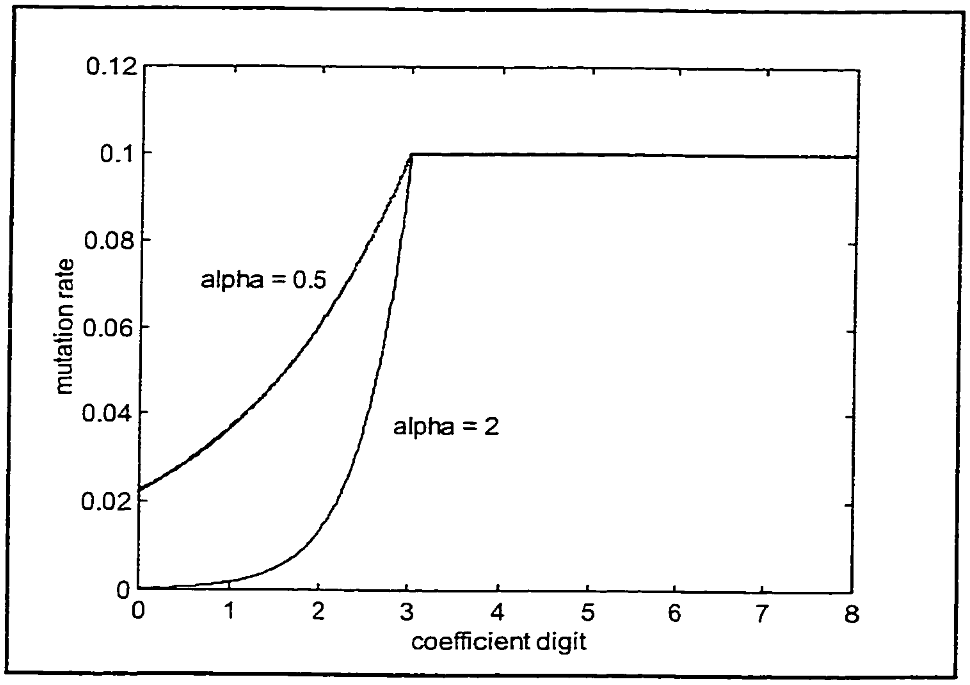

Fig.31c: Non-uniform mutation rate (completed 3/8 of iterations)

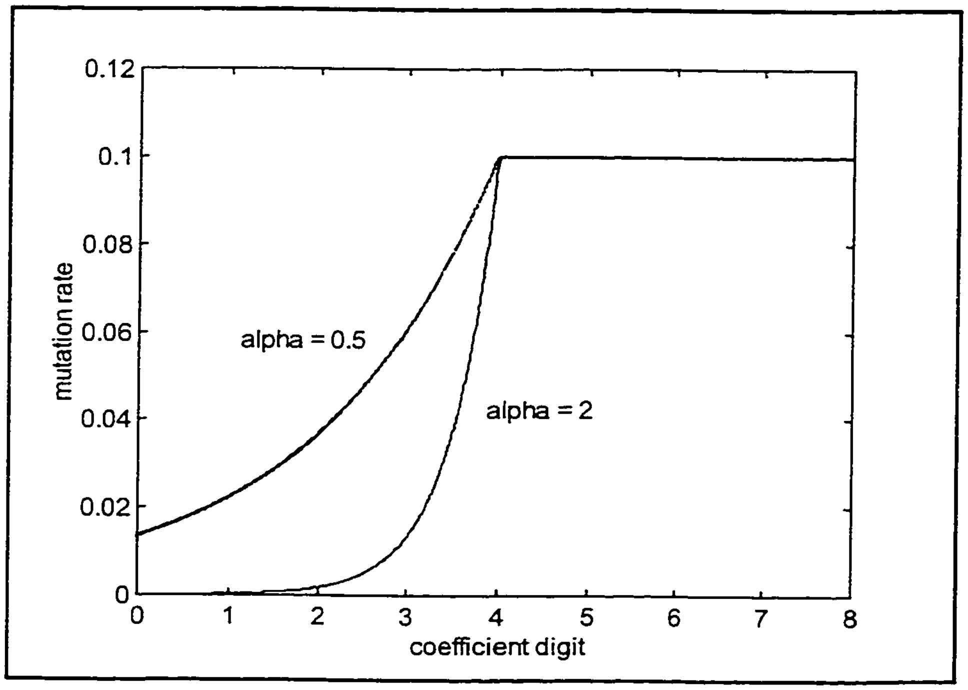

Fig.31d: Non-uniform mutation rate (completed 1/2 of iterations)

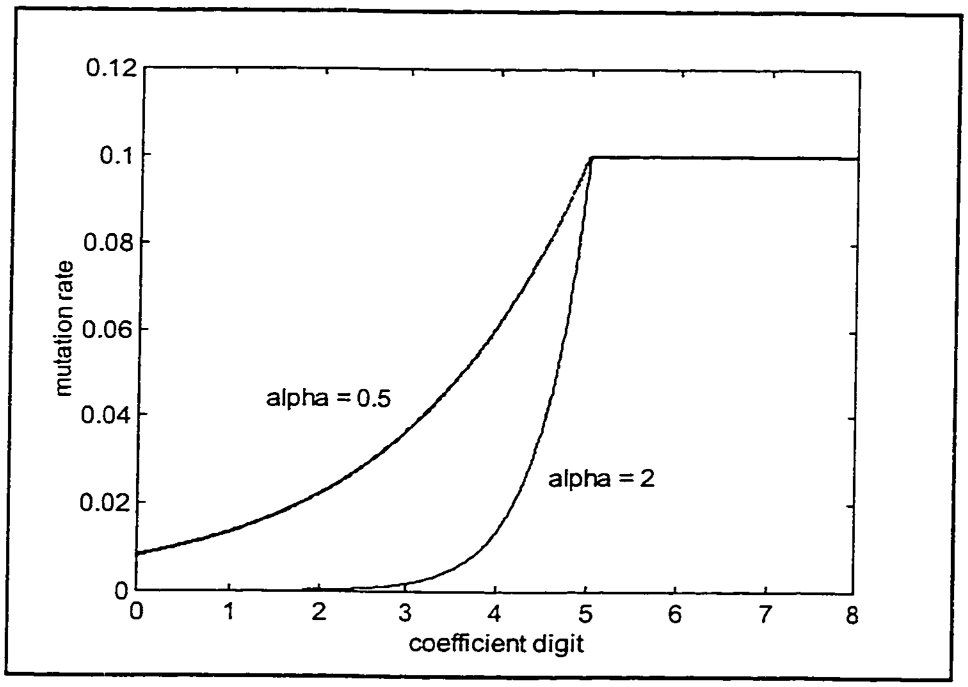

Fig.31e: Non-uniform mutation rate (completed 5/8 of iterations)

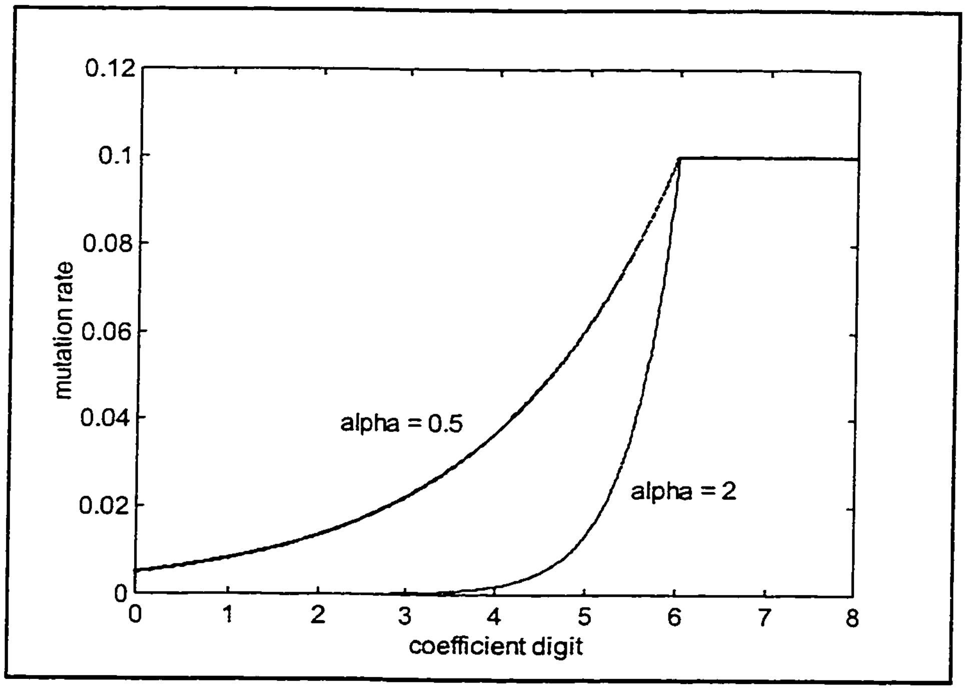

Fig.31f: Non-uniform mutation rate (completed 3/4 of iterations)

We have also applied the concept of non-uniform rate to crossover operations. The function for the non-uniform mutation rate and non-uniform crossover rate are similar, with the exception of the values on the y-axis. The maximum value of the mutation rate is 0.1 in Figures.31a through 31f and will remain constant during the iterations. However, unlike the mutation rate, the maximum value for a non-uniform crossover rate decreases linearly with each iteration. This non-uniform function will allow for more cut-off points to be chosen at the LSD at the end of each iteration cycle without disturbing the established pattern at the most significant digits. At the onset of each iteration, the crossover value should be assigned a high value to create different potential solution spaces. Whereas, near the end of the iteration, it may be desirable for the designer to maintain the constructed chromosomes and allow mutation to work towards finding the optimized solution.

3.3 1-D FIR FILTER DESIGN

In the following sections, we investigate the performance of the proposed design techniques using Genetic Algorithms with different choices of cost functions, parent selection schemes, and encoding schemes.

3.3.1 Design Formulation

For a given specification, the proposed filter design technique uses the following steps:

1. Determine the order of the filter by using Kaiser window imperical formula [3].
2. Choose the maximum number of iterations.

3. Initialize all variables that are used in Genetic Algorithm.

4. Decode each chromosome and calculate the frequency response of the design filter, $H_D(z)|_{z=e^{j\omega T}}$.

5. Calculate

$$E_{mag} = |H_D(z)| - |H_I(z)|,$$

where $H_D(z)$ and $H_I(z)$ are magnitude responses of the designed and ideal filters respectively.

6. Evaluate the cost function by using either Least Mean Square (LMS) or Minimax error function. Their equations are shown as follows:

$$\text{i) } E_{\ell_2} = \left[\Sigma E_{Mag}^2\right]^{1/2} \qquad \text{LMS,} \qquad (3.1)$$

$$\text{ii) } E_{\ell_\infty} = \max |E_{Mag}| \qquad \text{Minimax,} \qquad (3.2)$$

7. Minimize E_{ℓ_2} or E_{ℓ_∞} by using Genetic Algorithm.

8. Go back to step 4 until the specification is met, or exceed the maximum number of iterations.

The initialization of the Genetic Algorithm in step 3 needs the following data.

a). Determine the population size of Genetic Algorithm. The population size is obtained by using the equation,

$$\text{Population size} = \lceil 1.5 \times \log_2(N) \rceil.$$

Another option in determining the population size is to assign a life time value [43] to every chromosome. Any chromosome that has not been chosen for N number of times is discarded from the population. Therefore, the population size is a dynamic value controlled by the process. A large amount of chromosomes are created at the beginning of GA process. As iterations proceed, unsuitable chromosomes will 'die out', and the remaining population will lead the process to a final solution.

b). Select the parent scheme and encoding scheme to perform GA.

c). Assign mutation and crossover rates for GA process.

d). Assign a number for the Maximum Failure operator as shown in Fig.25.

3.4 1-D FIR FILTER DESIGN EXAMPLES

To illustrate the usefulness of the proposed design technique, various filters are designed with different combinations of parent selection schemes, encoding schemes, and cost functions (see Appendix II). Designed filter coefficients along with its magnitude response will be shown in these examples.

Example 1.

A 24th order lowpass FIR filter is designed with the following specification.

$$|H_I(e^{j\omega T})| = \left\{ \begin{array}{lll} 1 & \text{for} & 0 \le \omega \le 1 \; rad/sec \\ 0 & \text{for} & 2 \le \omega \le \frac{\omega_s}{2} \end{array} \right.$$

where $\omega_s = 2\pi$ rad/sec and T = 1 sec. The coefficient wordlength is restricted to be 12 bits with a maximum of 3 non-zero digits. The following combination of Genetic Algorithm is used:

Parent selection scheme: Roulette Wheel
Object function: Minimax
Encoding scheme: Signed-Digit Encoding
Population size: 230
The maximum number of CSD failure: 3
Static mutation rate: 0.03
Static crossover rate: 0.8

Fig.32 shows the magnitude response of the resultant filter. Its corresponding Canonical Signed-Digit coefficients are shown in Table 2. In this example the stopband suppression is over 90dB.

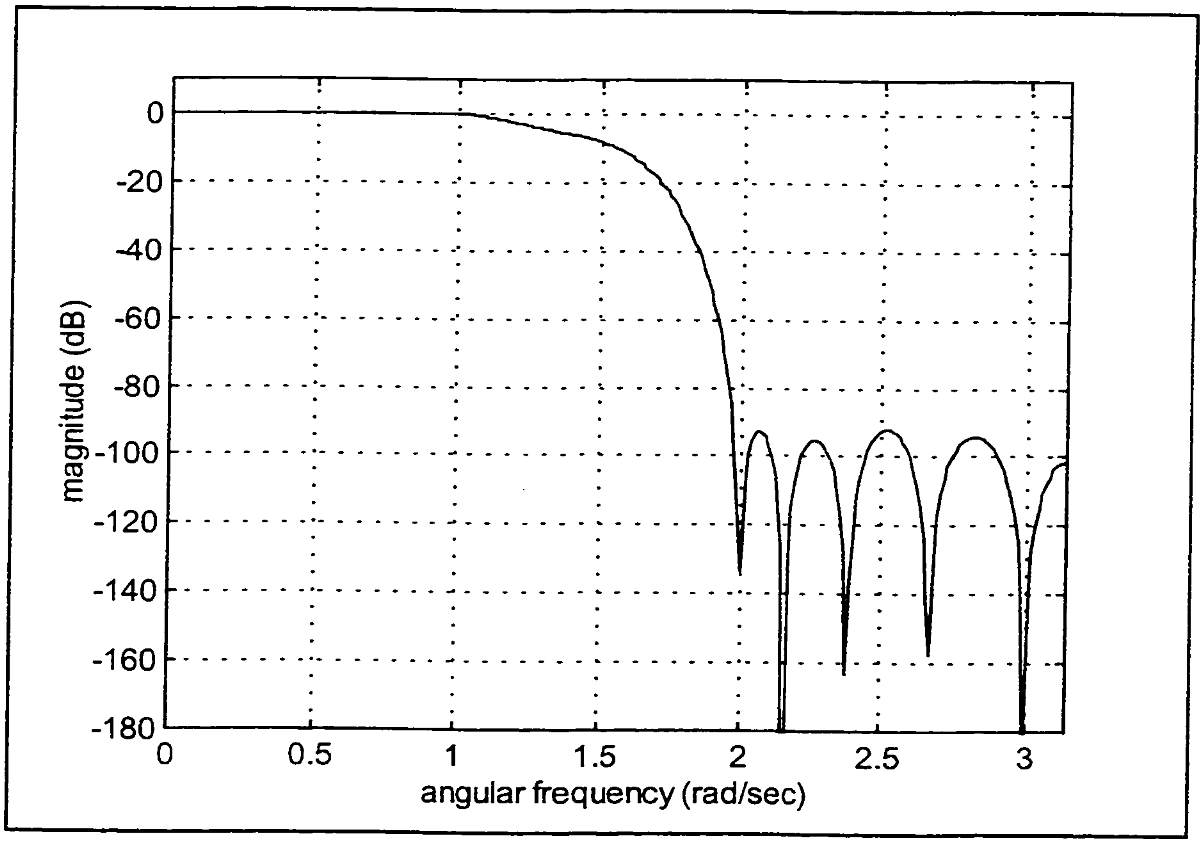

Fig. 32: A 24th order FIR filter designed by Signed-Digit Encoding, Roulette Wheel Parent Selection, and Minimax object function.

Coefficients	Values
a(0)	$2^{-7}+2^{-9}+2^{-11}$
a(1)	$2^{-7}-2^{-11}$
a(2)	$2^{-8}-2^{-10}$
a(3)	$2^{-7}+2^{-9}+2^{-11}$
a(4)	2^{-9}
a(5)	$2^{-7}+2^{-9}$
a(6)	$2^{-7}-2^{-11}$
a(7)	$2^{-6}-2^{-9}$
a(8)	$2^{-5}-2^{-7}-2^{-10}$
a(9)	$2^{-4}+2^{-7}+2^{-9}$
a(10)	$2^{-5}+2^{-11}$
a(11)	$2^{-2}-2^{-4}+2^{-7}$
a(12)	$2^{-1}+2^{-5}+2^{-10}$
a(13)	$2^{-2}-2^{-4}+2^{-7}$
a(14)	$2^{-5}+2^{-11}$
a(15)	$2^{-4}+2^{-7}+2^{-9}$
a(16)	$2^{-5}-2^{-7}-2^{-10}$
a(17)	$2^{-6}-2^{-9}$
a(18)	$2^{-7}-2^{-11}$
a(19)	$2^{-7}+2^{-9}$
a(20)	2^{-9}
a(21)	$2^{-7}+2^{-9}+2^{-11}$
a(22)	$2^{-8}-2^{-10}$
a(23)	$2^{-7}-2^{-11}$
a(24)	$2^{-7}+2^{-9}+2^{-11}$

Table 2:A 24th order 1-D FIR filter coefficients.

Example 2.

A 24th order lowpass FIR filter is designed with the following specification.

$$|H_I(e^{j\omega T})| = \begin{cases} 0 & \text{for} \quad 0 \le \omega \le 0.7 \text{ rad/sec} \\ 1 & \text{for} \quad 1.2 \le \omega \le 1.8 \text{ rad/sec} \\ 0 & \text{for} \quad 2.3 \le \omega \le \frac{\omega_s}{2} \end{cases}$$

where $\omega_s = 2\pi$ rad/sec and T = 1 sec. The coefficient wordlength is restricted to be 12 bits with a maximum of 3 non-zero digits. The following combination of Genetic Algorithm is used:

Parent selection scheme: Ranking,
Object function: Minimax
Encoding scheme: Signed-Digit Encoding
Population size: 230
The maximum number of CSD failure: 3
Static mutation rate: 0.03
Static crossover rate: 0.8

Fig.33 shows the magnitude response of the resultant filter. Its corresponding Canonical Signed-Digit coefficients are shown in Table 3. In this example the stopband suppression is around 40dB. The poor performance of band-pass filter is due to the reduction of the transition width comparing to the previous lowpass filter examples.

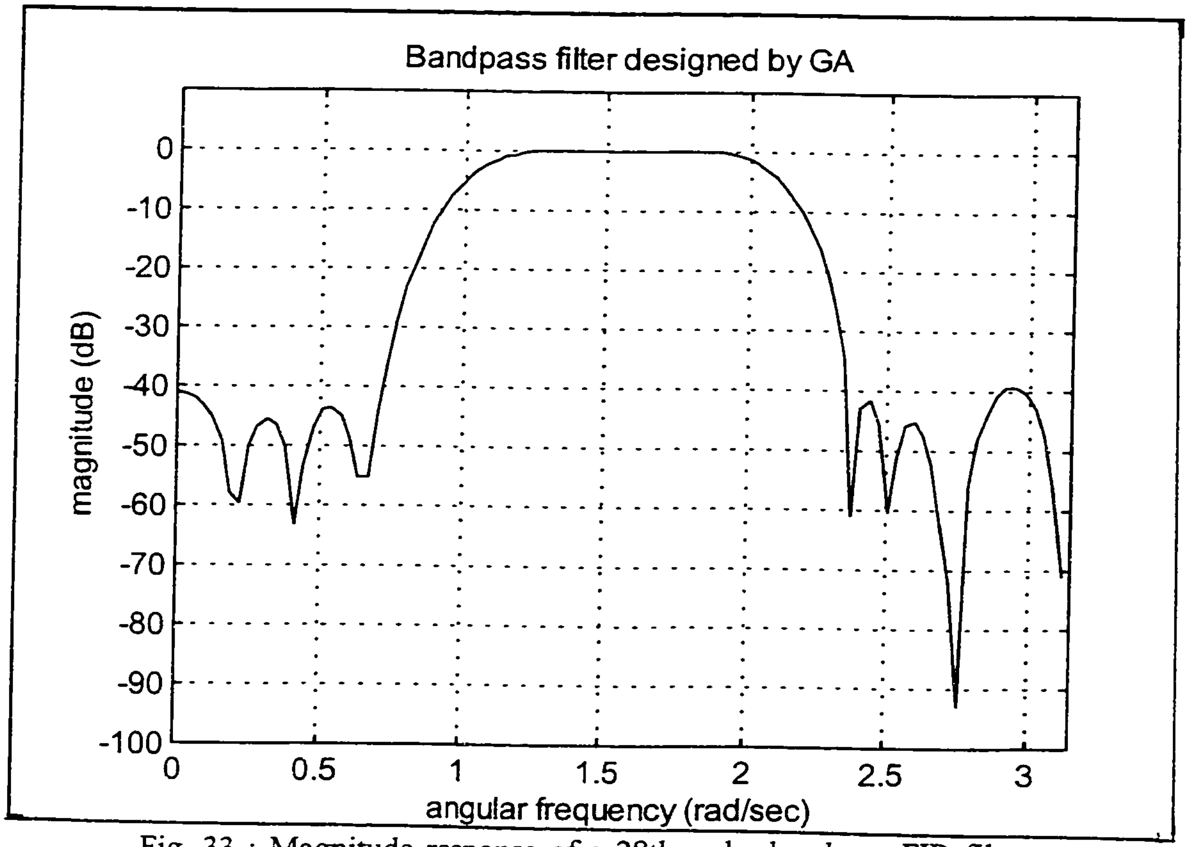

Fig. 33 : Magnitude response of a 28th order bandpass FIR filter with Signed-Digit Encoding, Ranking Parent Selection, and Minimax strategy.

coefficients	value
a(0)	2^{-7}
a(1)	$-2^{-8} + 2^{-11}$
a(2)	$-2^{-7} - 2^{-9} + 2^{-11}$
a(3)	$2^{-9} - 2^{-11}$
a(4)	$-2^{-6} - 2^{-11}$
a(5)	$2^{-9} + 2^{-11}$
a(6)	$2^{-4} - 2^{-6} + 2^{-9}$
a(7)	$-2^{-7} + 2^{-9} - 2^{-11}$
a(8)	$-2^{-5} + 2^{-7} - 2^{-11}$
a(9)	-2^{-11}
a(10)	$-2^{-3} + 2^{-6} + 2^{-9}$
a(11)	$2^{-7} + 2^{-9} - 2^{-11}$
a(12)	$2^{-2} + 2^{-5} + 2^{-11}$
a(13)	$-2^{-7} + 2^{-9} + 2^{-11}$
a(14)	$-2^{-1} + 2^{-3} + 2^{-7}$
a(15)	$-2^{-7} + 2^{-9} + 2^{-11}$
a(16)	$2^{-2} + 2^{-5} + 2^{-11}$
a(17)	$2^{-7} + 2^{-9} - 2^{-11}$
a(18)	$-2^{-3} + 2^{-6} + 2^{-9}$
a(19)	-2^{-11}
a(20)	$-2^{-5} + 2^{-7} - 2^{-11}$
a(21)	$-2^{-7} + 2^{-9} - 2^{-11}$
a(22)	$2^{-4} - 2^{-6} + 2^{-9}$
a(23)	$2^{-9} + 2^{-11}$
a(24)	$-2^{-6} - 2^{-11}$
a(25)	$2^{-9} - 2^{-11}$
a(26)	$-2^{-7} - 2^{-9} + 2^{-11}$
a(27)	$-2^{-8} + 2^{-11}$
a(28)	2^{-7}

Table 3: A 24th order 1-D FIR filter coefficients.

3.5 COMPARISON

In this section, we will present the following three types of comparisons in the proposed filter design:

1. Error-Function
2. Parent Selection Scheme
3. Filter Coefficient Encoding Scheme

3.5.1 Error-Functions Comparison, LMS vs. Minimax

From the many filter design examples ran, it is noticed that the utilization of Minimax error function generates better filters than those designed with LMS. For more direct comparisons, refer to Fig.34a, 34b, 34c, 34d. These figures show the peak error versus the number of coefficients using the LMS and Minimax error functions under different combinations of parent selection and encoding schemes. Fig.34a uses the combination of Mixed Encoding and Roulette Wheel Parent Selection. Fig.34b uses the combination of Signed-Digit Encoding and Roulette Wheel Parent Selection. Fig.34c uses the combination of Mixed Encoding and Ranking Parent Selection. Fig.34d uses the combination of Signed-Digit Encoding and Ranking Parent Selection. In all cases, Minimax error function outperforms the LMS. Due to the poor performance of LMS cost function, LMS cost function will not be our consideration for the rest of the comparisons in the following sections.

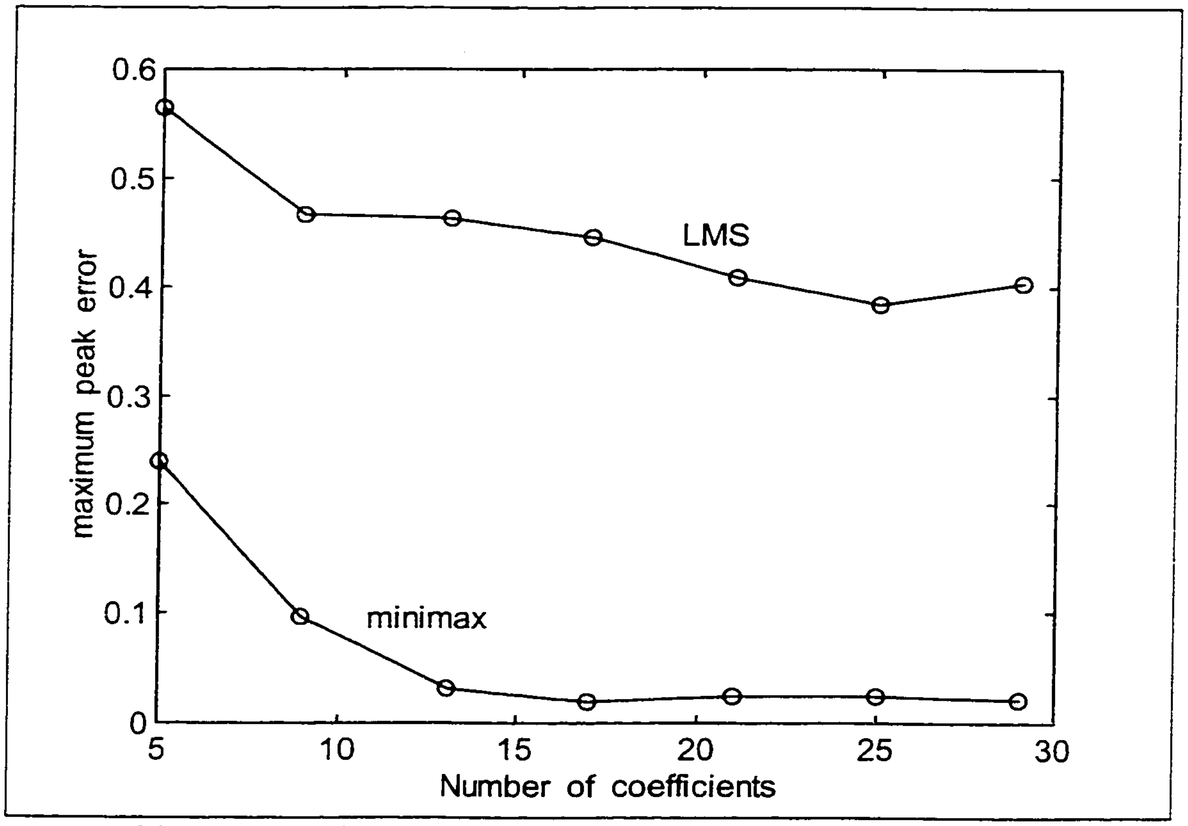

Fig. 34a : Maximum peak error comparison of LMS and Minimax errors with Mixed Encoding and Roulette Wheel Parent Selection

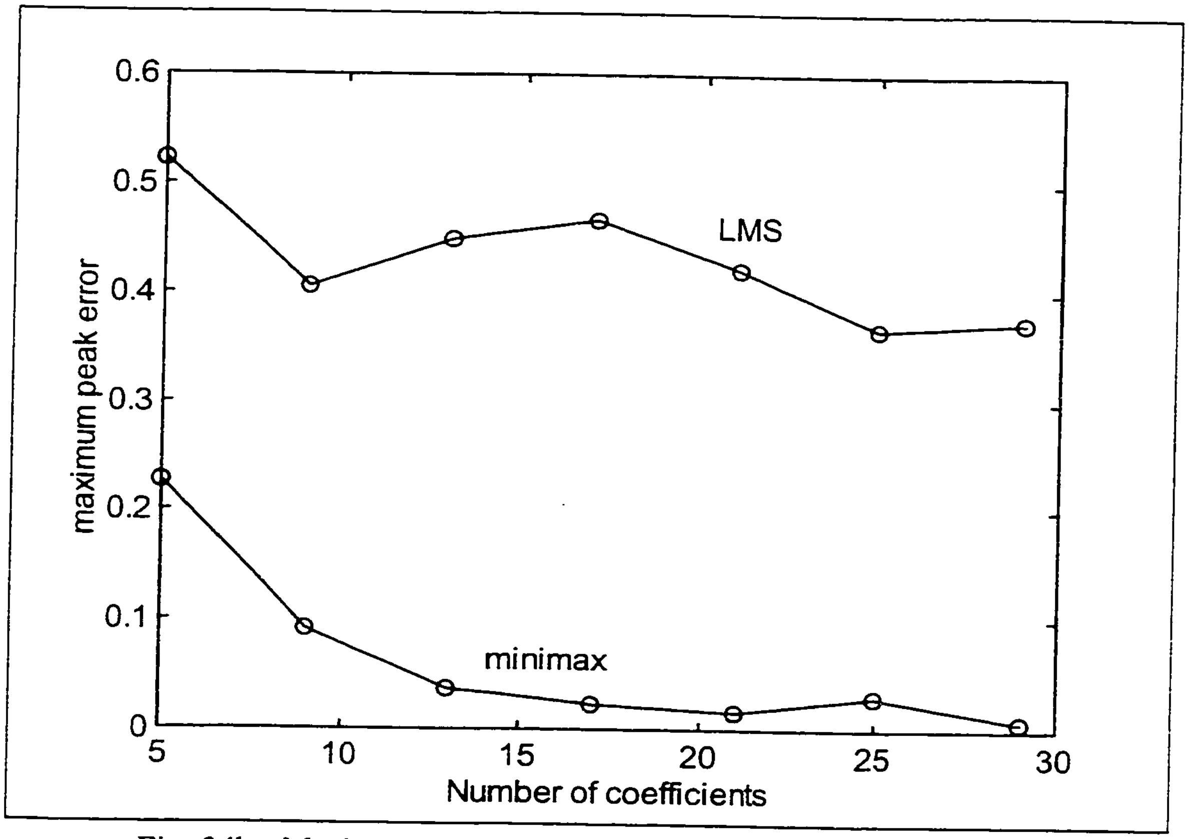

Fig. 34b : Maximum peak error comparison of LMS and Minimax errors with Signed-Digit Encoding and Roulette Wheel Parent Selection.

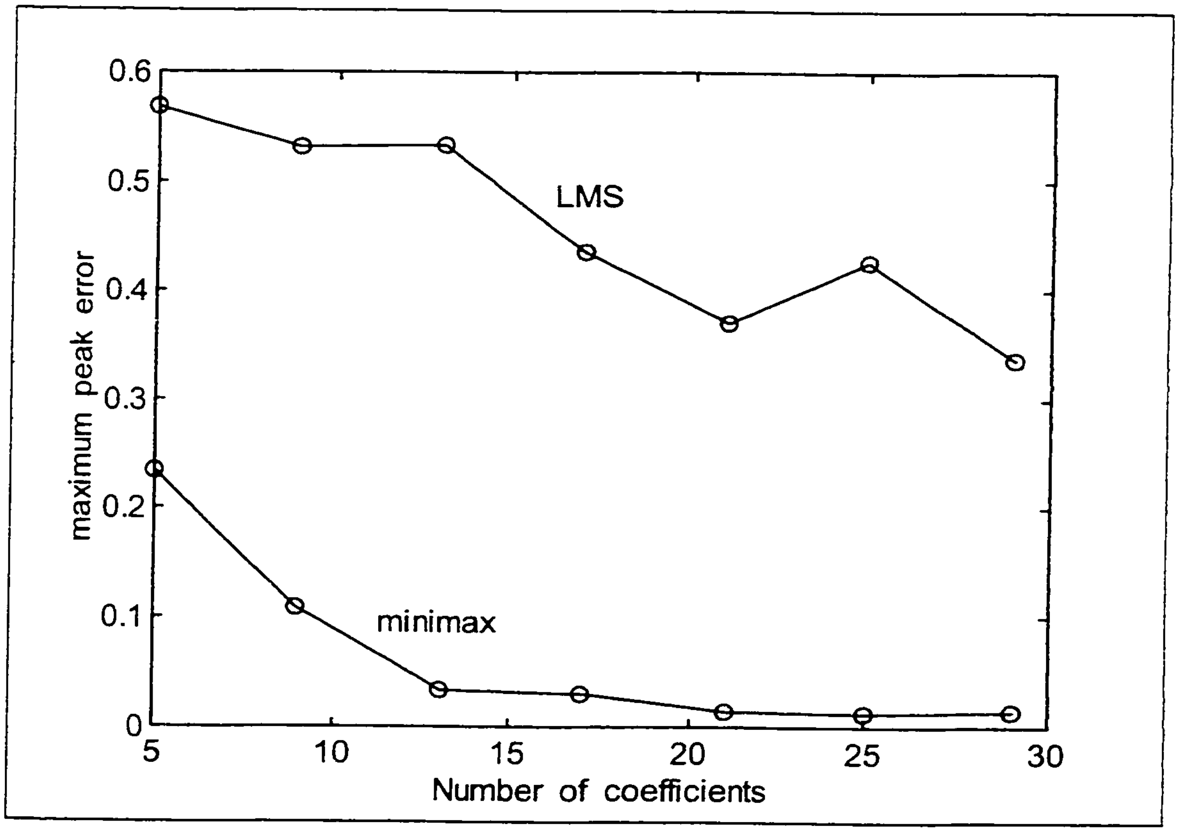

Fig. 34c : Maximum peak error comparison of LMS and Minimax errors with Mixed Encoding and Ranking Parent Selection.

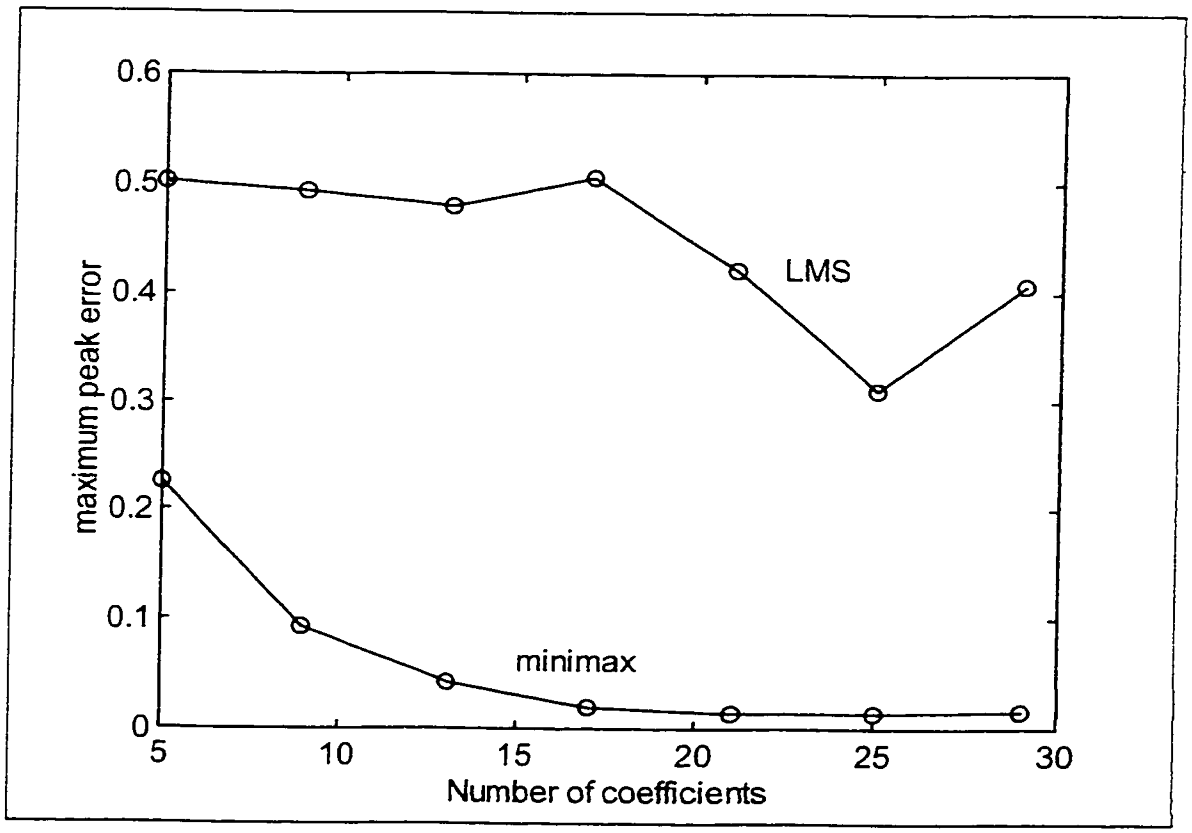

Fig. 34d : Maximum peak error comparison of LMS and Minimax errors with Signed-Digit Encoding and Ranking Parent Selection.)

3.5.2 Parent Selection Schemes Comparison

In this section, we compare the Roulette Wheel Parent Selection to the Ranking Parent Selection in terms of maximum peak error from the ideal filter and convergence rate. Fig.35a show the performance of filters designed for different parent selection schemes in terms of maximum peak error for various order of filters using Mixed Encoding and Minimax error function. This figure shows that the Ranking and Roulette Wheel Parent Selection yield similar performance. Fig.35b show the performance of filters designed for different parent selection schemes in terms of convergence rate with Mixed Encoding and Minimax error function. Fig.35b demonstrates that Ranking parent selection is more aggressive in its minimization of peak errors at the start of the iterations in comparison to the Roulette Wheel parent selection scheme. In other words, the number of peak errors decreases more quickly compared to the step-like effect of the Roulette Wheel. Fig.35c shows the performance of filters designed for different parent selection schemes in terms of maximum peak error for various orders of filters using Signed-Digit Encoding and Minimax error function. The result shows that filters designed by these two parent selection schemes have similar performance. Fig.35d shows the performance of filters designed for different parent selection schemes in terms of convergence rate with Signed-Digit Encoding and Minimax error function. It shows that both parent selection schemes have similar convergence rate.

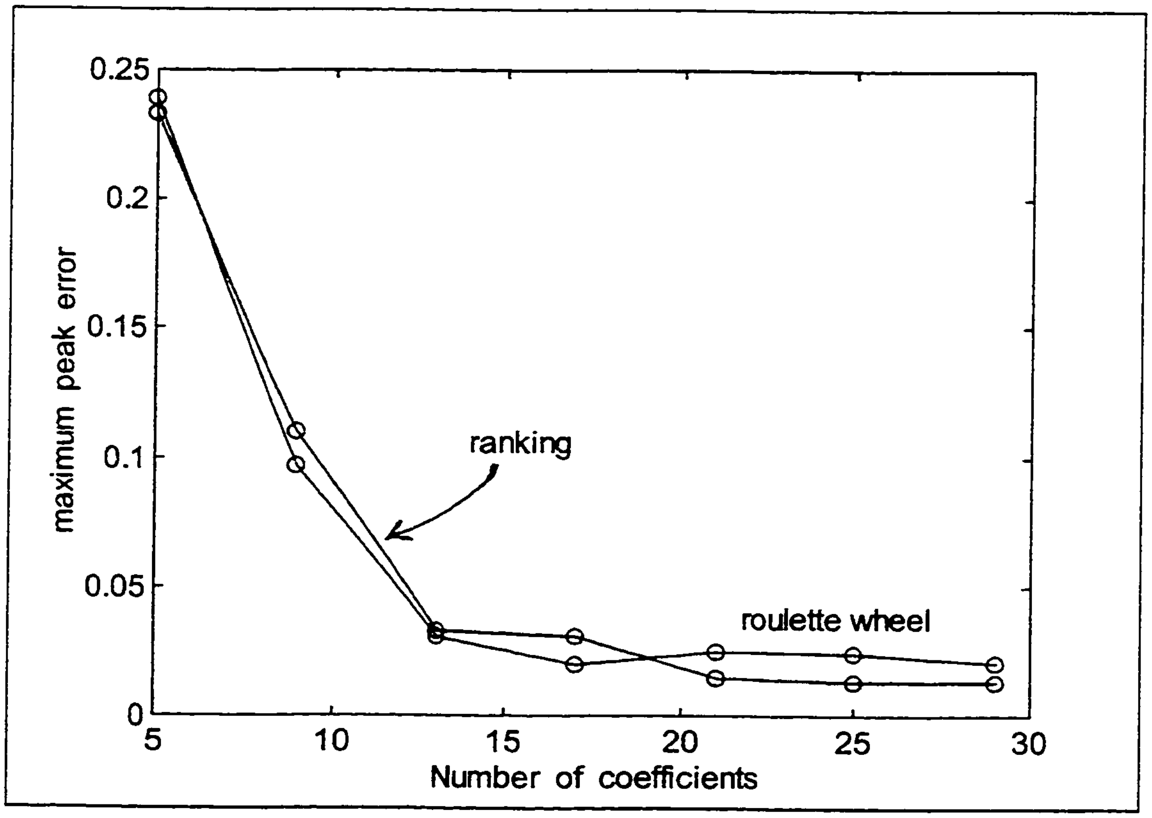

Fig. 35a : Maximum peak error comparison of Roulette Wheel Parent Selection and Ranking method with Mixed Encoding and Minimax strategy.

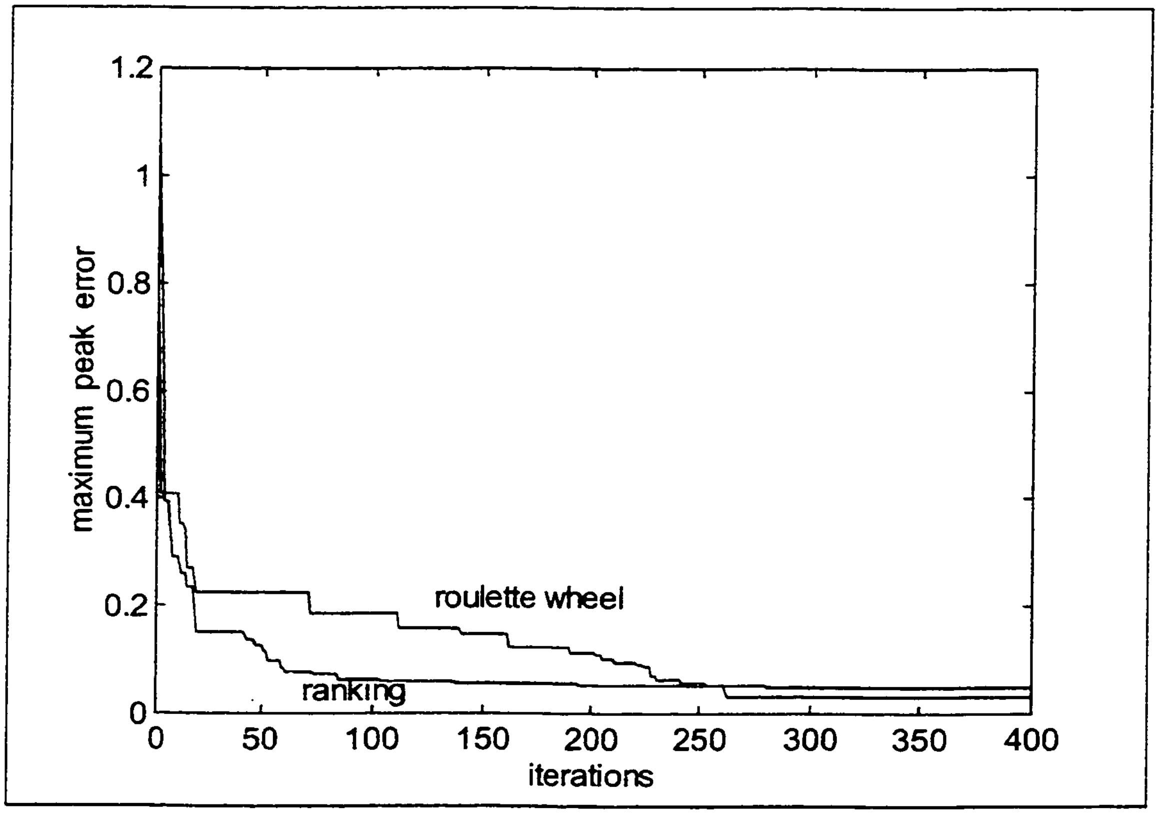

Fig. 35b : Convergence rate comparison of Roulette Wheel Parent Selection and Ranking method with Mixed Encoding and Minimax strategy.

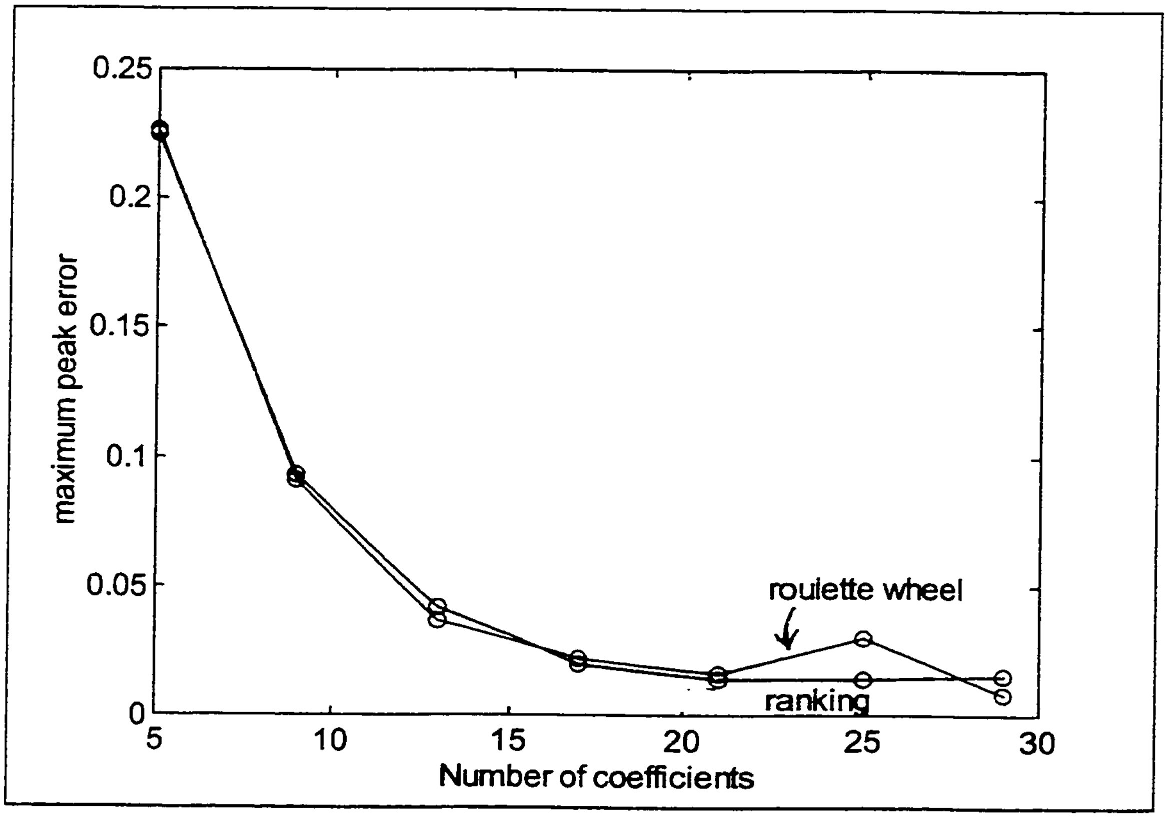

Fig. 35c : Maximum peak error comparison of Roulette Wheel Parent Selection and Ranking method with Signed-Digit Encoding and Minimax strategy.

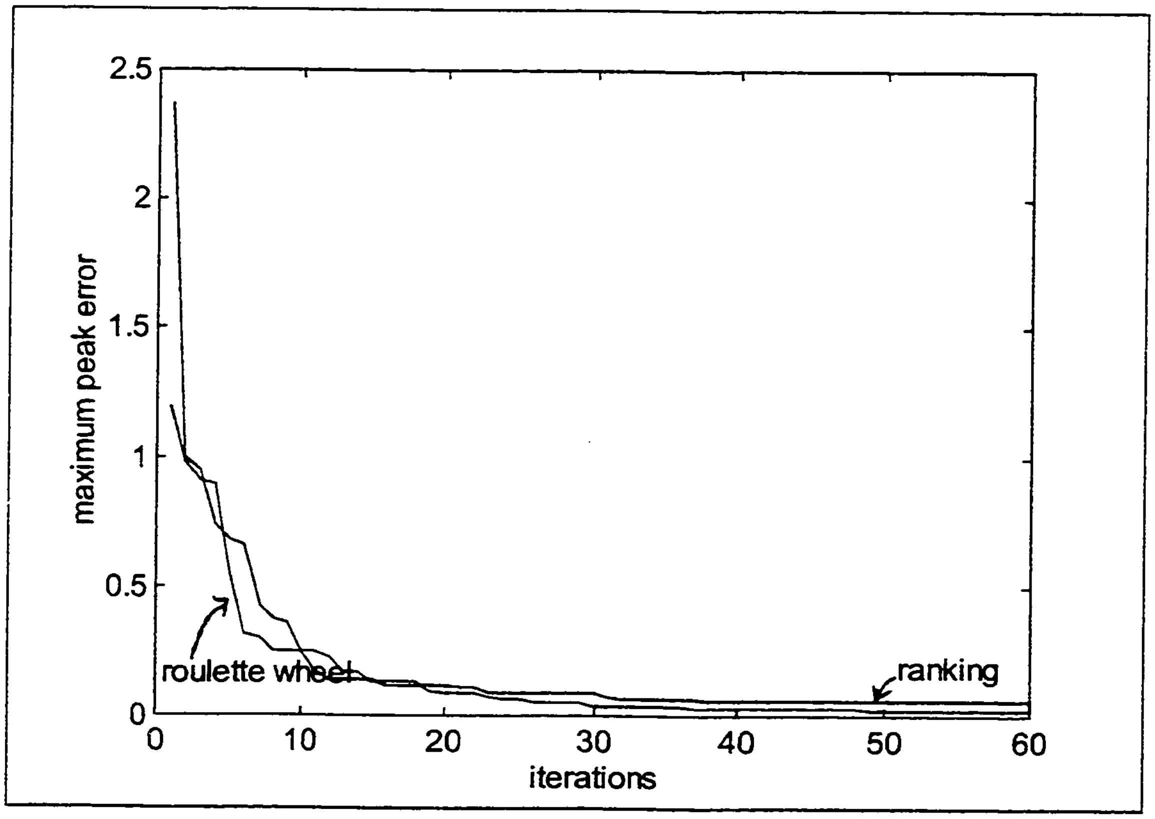

Fig. 35d : Convergence rate comparison of Roulette Wheel Parent Selection and Ranking method with Signed-Digit Encoding and Minimax strategy.

3.5.3 Filter coefficient Encoding Schemes Comparison

Fig.36a shows the comparison of Maximum peak error between Mixed Encoding and Signed-Digit Encoding schemes in terms of maximum peak error from the ideal filter using the combination of Roulette Wheel Parent Selection and Minimax error function. This figure shows that both Mixed and Signed-Digit Encoding have similar performance. Fig.36b shows the comparison of Maximum peak error between Mixed Encoding and Signed-Digit Encoding schemes in terms of convergence rate using Roulette Wheel Parent Selection and Minimax error function. This figure shows that filters designed with Signed-Digit Encoding and Roulette Wheel provide faster convergence rate than Mixed Encoding. Fig.36c shows the comparison of Maximum peak error between Mixed Encoding and Signed-Digit Encoding schemes in terms of maximum peak error from the ideal filter using the combination of Ranking Parent Selection and Minimax error function. Fig.36d shows the comparison of Maximum peak error between Mixed Encoding and Signed-Digit Encoding schemes in terms of convergence rate using Ranking Parent Selection and Minimax error function. Both Fig.36c,36d show that filters perform similarly regardless of whether a Signed-Digit or Mixed Encoding scheme is used.

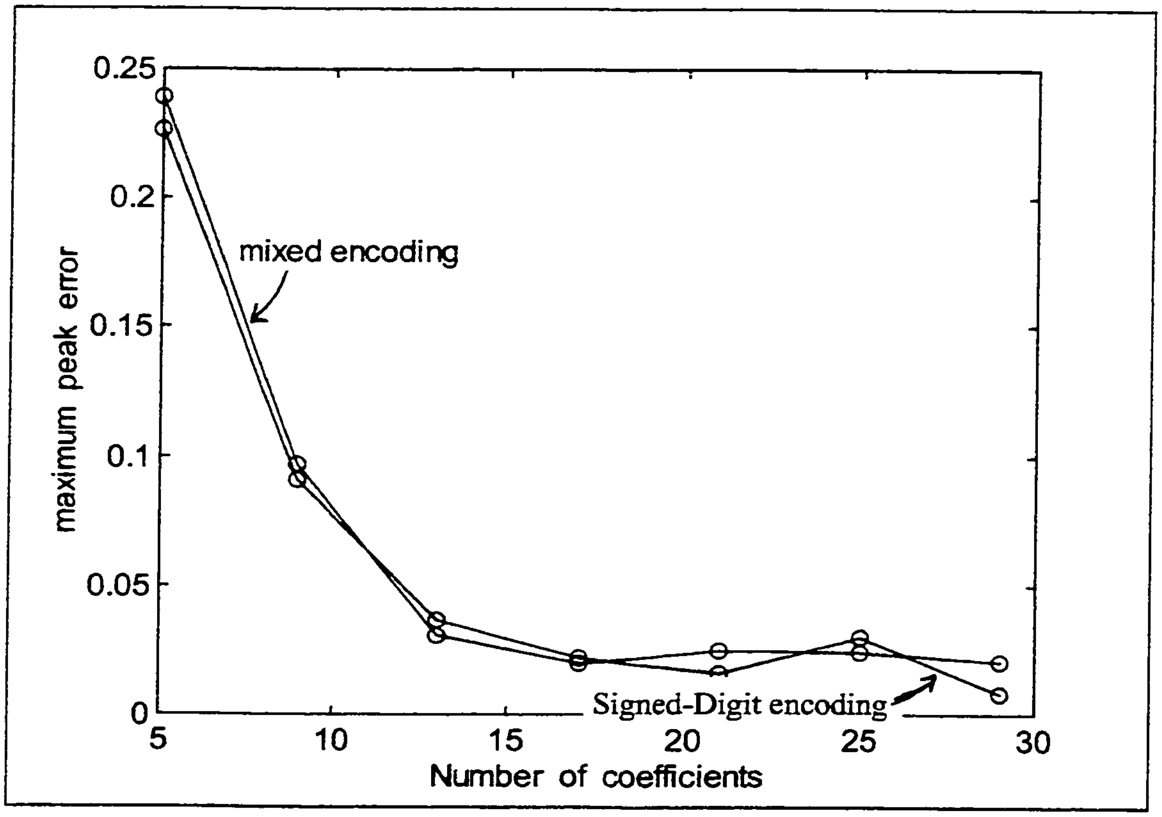

Fig. 36a : Maximum peak error comparison of Mixed Encoding and Signed-Digit Encoding method with Roulette Wheel Parent Selection and Minimax strategy.

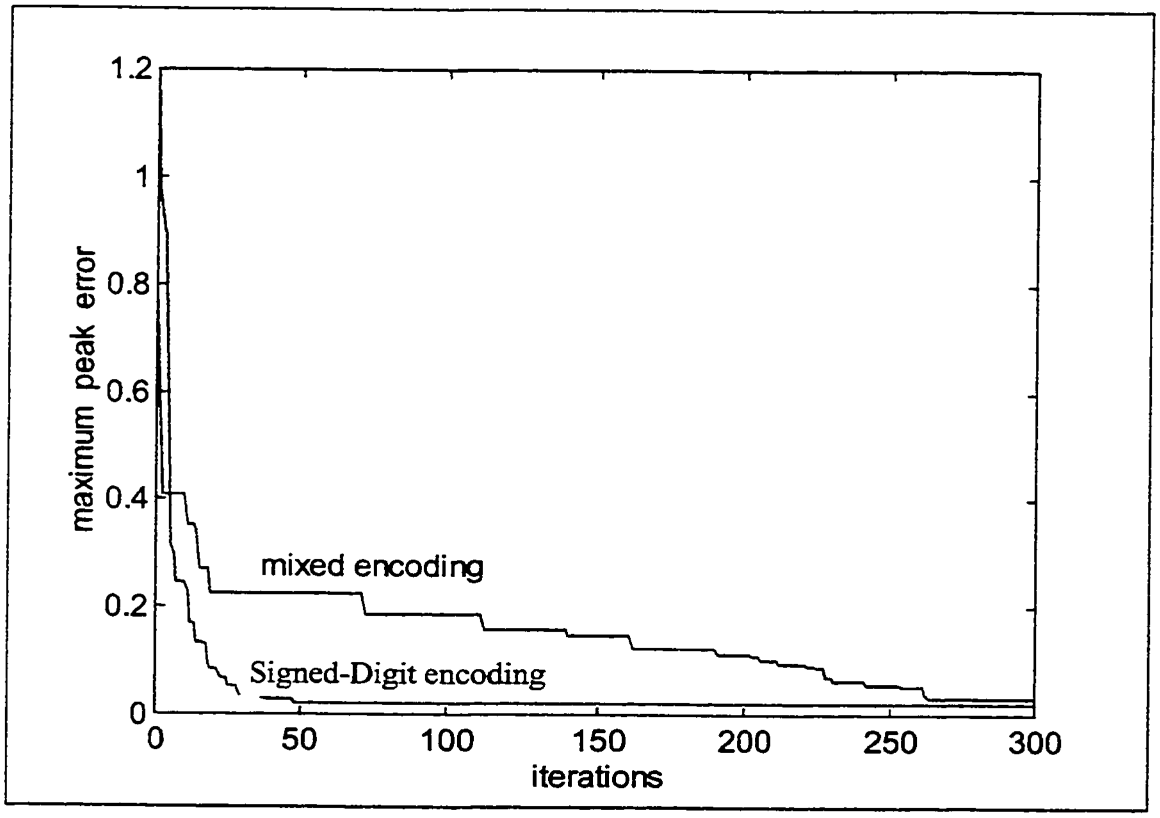

Fig. 36b: Convergence rate comparison of Mixed Encoding and Signed-Digit Encoding method with Roulette Wheel Parent Selection and Minimax strategy.

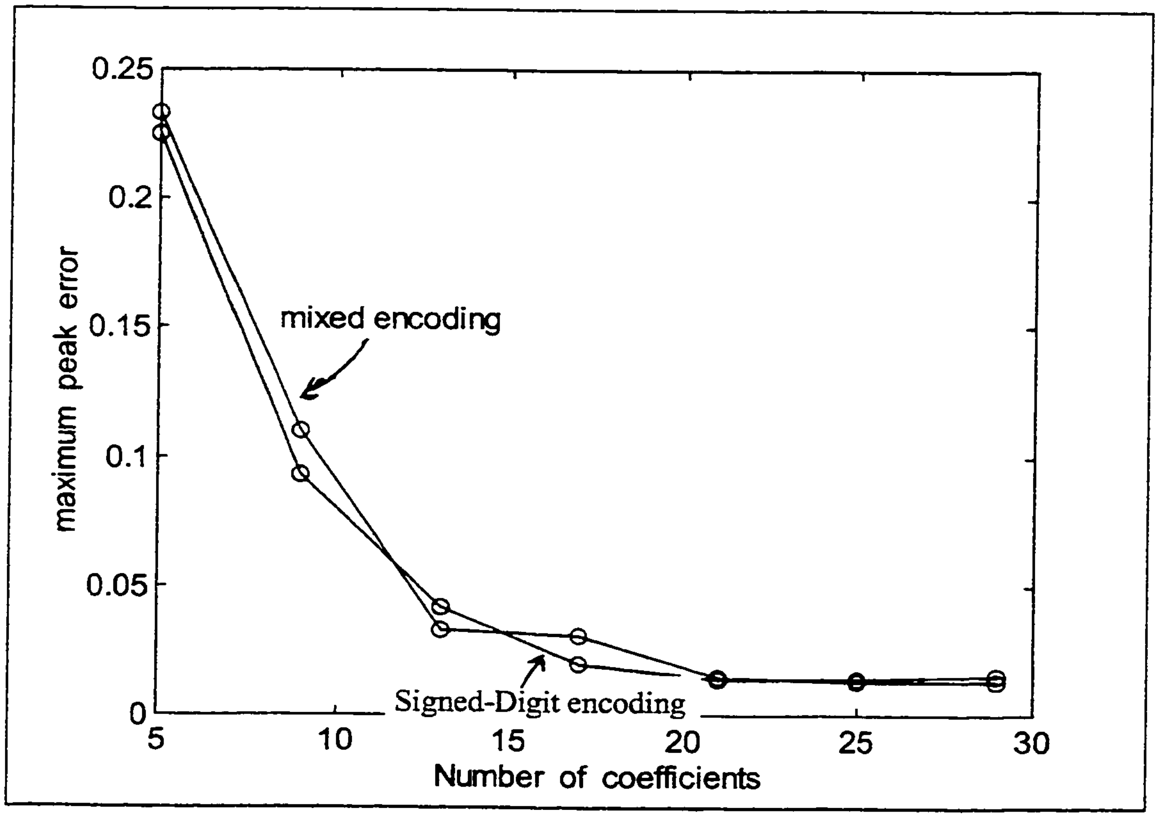

Fig. 36c : Maximum peak error comparison of Mixed Encoding and Signed-Digit Encoding method with Ranking method and Minimax strategy.

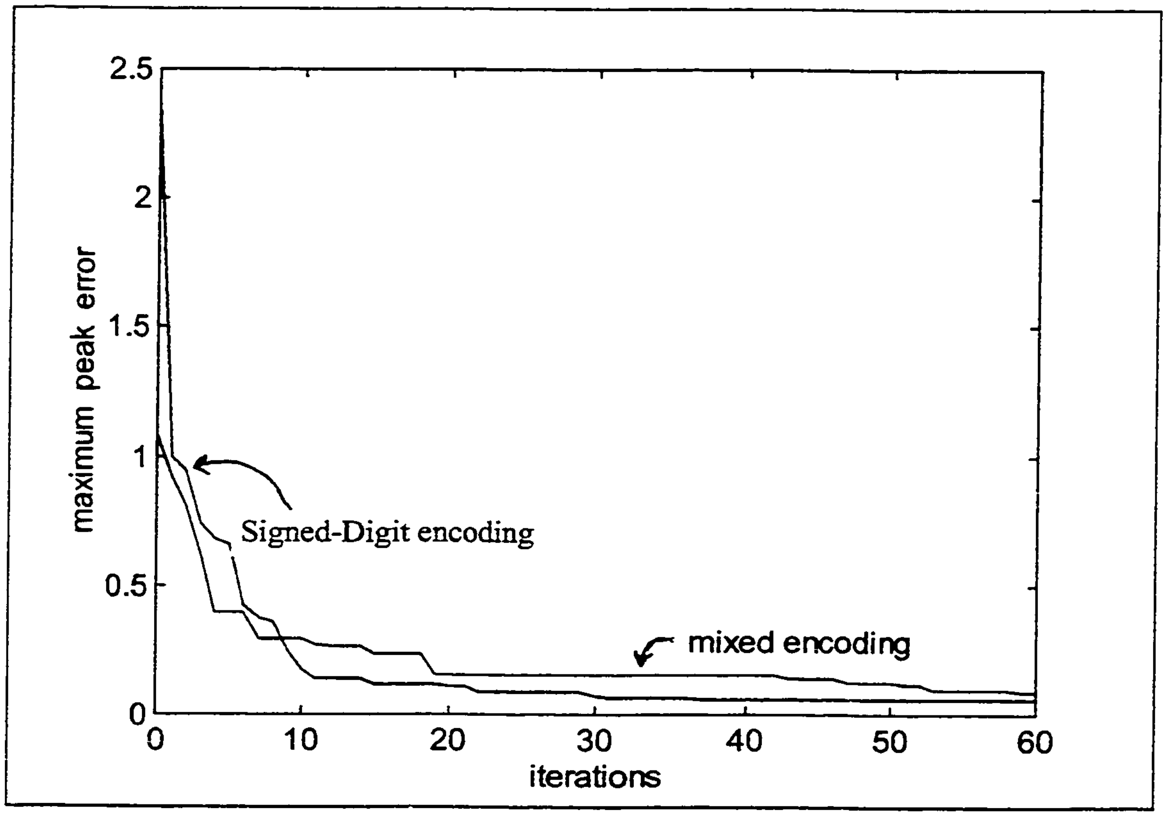

Fig. 36d : Convergence rate comparison of Mixed Encoding and Signed-Digit Encoding method with Ranking method and Minimax strategy.

We conclude that Signed-Digit Encoding with Roulette Wheel parent selection and Minimax error function produces the best result among all simulations. However, one may consider this design process relatively time consuming. Thus, we would like to pursue a filter design that takes less time to produce, but the filter response may not approach toward global optimum. In this alternative, the combination of Signed-Digit Encoding, Ranking parent selection and Minimax error function are used.

3.6 DESIGN FORMULATION OF 1-D DIGITAL FILTER WITH DBNS COEFFICIENTS

In this section, we will investigate the utilization of the single digit Double-Base Number System (DBNS) in digital filter design. The challenge of this filter design is to obtain the sign and base 2, 3 indices representation that will lead to proper filter coefficients to satisfy arbitrary specification. It is also desirable to keep the ternary index as small as possible to reduce the hardware resources, since the computation of $3^{\text{ternary index}}$ is realized by a read only memory [25]. Our design methodology will be derived from our previously proposed technique described in Section 3.3.1 with the following changes:

i). The CSD Check operator is replaced by a DBNS Check operator. For lowpass filter design, DBNS Check is defined as every decoded coefficient within a chromosome must be within the range of {-1.0 1.0}.

ii). The definition of Maximum Failure operator remains the same. If the coefficient fails more than *N* times, it will be directed to the Reset Coefficient operator. Otherwise, the coefficient will go through Crossover again.

iii). The functionality of Reset Coefficient operator also remains the same with the exception of using DBNS number. The coefficients received by the Reset Coefficient operator will be replaced by a randomly generated binary string that corresponds to a valid DBNS number.

3.6.1 Design Formulation for 1-D FIR Filter with DBNS

For a given specification (envelope), the proposed filter design technique uses the following steps:

1. Determine the order of the filter by using Kaiser window imperical formula [3].
2. Choose the maximum number of iterations.
3. Create a set of DBNS numbers, then map each of the numbers to an unique binary string.
4. Initialize all variables that are used in the Genetic Algorithm.
5. Decode each chromosome and calculate the frequency response of the design filter, $H_D(z)|_{z=e^{j\omega T}}$.
6. Calculate

   $$E_{mag} = |H_D(z)| - |H_I(z)|,$$

   where $H_D(z)$ and $H_I(z)$ are the designed and ideal filters.
7. Evaluate the cost function by using Minimax error function, $E_{L_\infty} = \max|E_{Mag}|$.
8. Minimize E_{L_∞} using the Genetic Algorithm.
9. Go back to step 5 until the specification is met, or exceed the maximum number of iterations.

The creation of the DBNS numbers in Step 3 is divided into two sub-steps:

a). Generate a all available DBNS numbers with restriction of the maximum ternary value, and the maximum bound.

b). Screen out the *redundant* data from the set with restriction of the minimum spacing and the least allowable number in the set, where minimum spacing is defined as the spacing between two DBNS numbers. Data is redundant if a truncated DBNS number can be represented by more than one combination of binary exponent and ternary exponent. Next, each of the remaining (valid) numbers will be mapped to a binary string.

Similar to the previous sections, the initialization of the Genetic Algorithm in step 4 needs the following data.

a). Determine the population size for Genetic Algorithm. The population size is obtained by using the equation,

$$\text{population size}=\lceil 1.5 \times \log_2(N) \rceil.$$

b). Select the parent scheme to perform GA.

c). Assign mutation and crossover rates for GA operations.

d). Assign a number for the Maximum Failure operator as shown in Fig.25.

The utilization of DBNS combining with GA requires extra step (Step 3) in filter design. The functionality of Step 3 is to prepare a search domain for GA, since redundant data will degrade the GA performance.

During the screening process, adjacent DBNS representations are compared. The one which contains the least ternary value is kept on the list for use in filter design. This process is repeated until the whole list is sorted out. Small ternary value has the advantage of reducing the size of the look-up-table for filter realization purposes. The algorithms of generating DBNS numbers and screening process are both outlined in Appendix I.

3.6.2 EXAMPLE

The 1-D FIR filter design separate into two phases, obtain a list of valid DBNS number and filter optimization: A set of DBNS numbers is created by using a maximum ternary value of 200. The 'lower bound' and minimum spacing are both equal to 1.0E-9, where the 'lower bound' is the smallest number in the list. A list of screened DBNS numbers is generated and is illustrated by the following histogram.

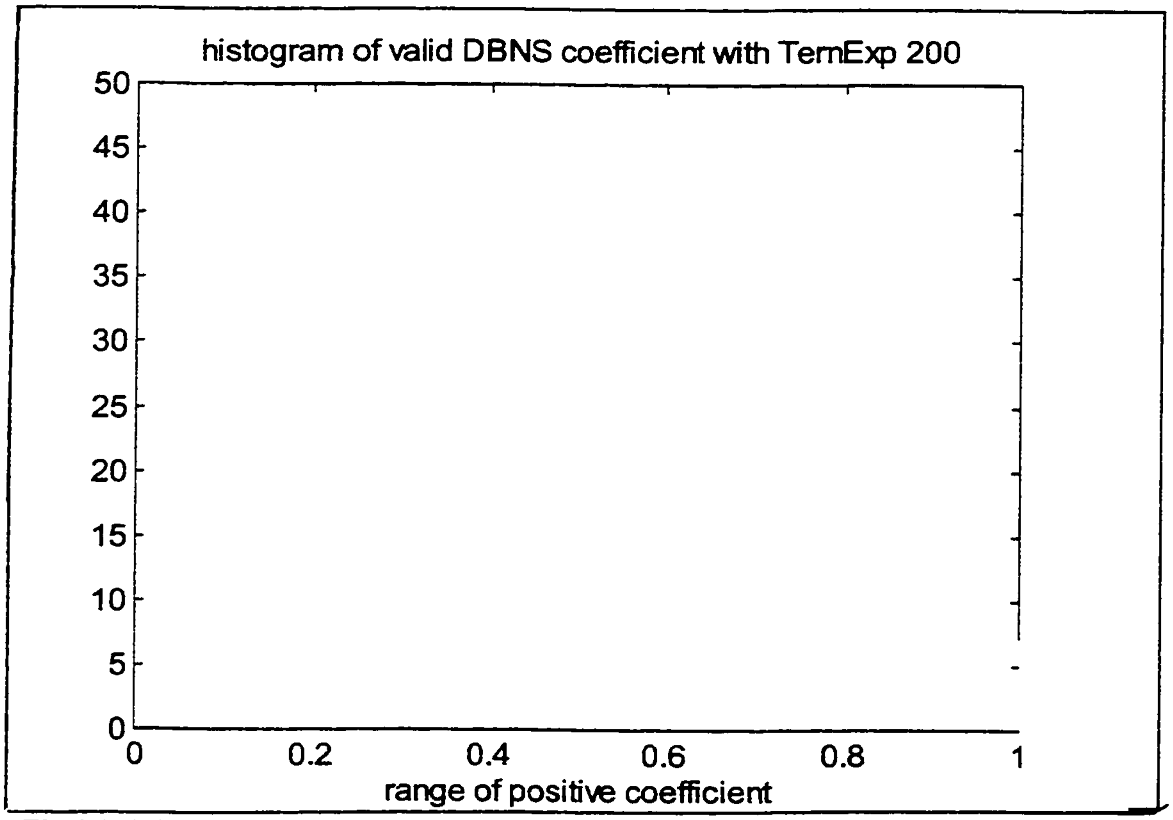

Fig.37 : Histogram of valid DBNS number with the maximum ternary value of 200.

Fig.37 is a histogram of DBNS numbers generated between 0 and 1.0. The majority of the data is very small, especially for those less than 0.1. Within a spacing of 0.1, the number of data present in each region (represented by a bar) decreases exponentially as it approaches 1.

The above data set is then encoded into binary strings through one-to-one mapping for GA operation. A 60th order lowpass FIR filter is designed with the following specification.

$$|H_I(e^{j\omega T})| = \begin{cases} 1 & \text{for} \quad 0 \le \omega \le 1 \ rad/sec \\ 0 & \text{for} \quad 2 \le \omega \le \frac{\omega_s}{2} \end{cases}$$

where $\omega_s = 2\pi$ rad/sec and T = 1 sec. Its magnitude response is shown below.

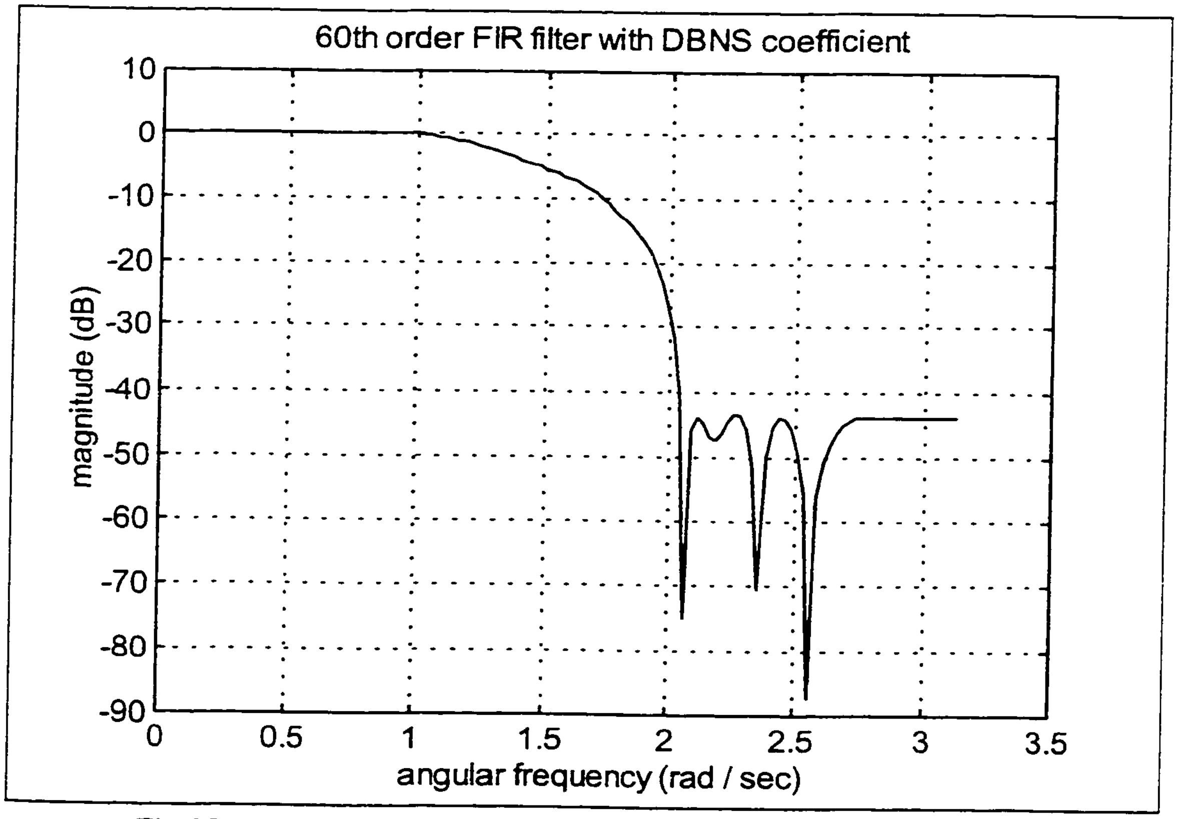

Fig.38 : Magnitude response of a 60th order lowpass filter with DBNS coefficients.

coefficients	binary exponent	ternary exponent	sign
a(0)	-106	59	1
a(1)	-59	24	-1
a(2)	-275	167	-1
a(3)	-82	39	-1
a(4)	98	-68	1
a(5)	74	-58	1
a(6)	225	-155	1
a(7)	-196	117	1
a(8)	233	-153	-1
a(9)	138	-93	-1
a(10)	278	-182	1
a(11)	-282	171	1
a(12)	-69	34	-1
a(13)	245	-169	-1
a(14)	-282	171	-1
a(15)	-233	140	-1
a(16)	-181	107	1
a(17)	51	-44	1
a(18)	-216	126	1
a(19)	282	-183	1
a(20)	-149	87	-1
a(21)	-4	-2	-1
a(22)	131	-88	1
a(23)	-114	67	1
a(24)	-136	73	-1
a(25)	208	-135	1
a(26)	-133	79	-1
a(27)	12	-10	-1
a(28)	-224	137	1
a(29)	282	-179	1
a(30)	-47	29	1

*Because of the even symmetry property of the coefficients, only half of the coefficients are shown.

Table 4: A 60 order 1-D FIR filter with DBNS coefficients.

3.7 SUMMARY

An enhanced version of Genetic Algorithm has been presented in this chapter. We have modified the Genetic Algorithm to design 1-D digital FIR filters. Three processes (CSD check, Maximum failure, Reset coefficients) are inserted into the GA block diagram to generate Canonic Signed-Digit coefficients. By utilizing the modified block diagram, numbers of low-pass filters are designed for analysis using different parent selection schemes, encoding schemes, and objective functions. We have also presented the design of 1-D FIR filter using DBNS arithmetic at the last section of this chapter. An example has been provided to illustrated the design process.

CHAPTER 4. DESIGN OF 2-D FILTERS

4. INTRODUCTION

Similar to 1-D digital filter design, the implementation of 2-D digital filters using infinite-precision coefficients introduces truncation and quantization errors. This effect may result in filter violating the desired specification and in the case of 2-D IIR filter may cause instability . The utilization of discrete coefficients with restriction on the number of non-zero digits has the advantage of eliminating these problems.

In this chapter, two techniques will be presented for 2-D digital filter designs with discrete coefficients. Each technique will be dealing with the design of 2-D FIR or IIR filters. The advantages of the proposed techniques include guaranteed filter stability, and better approximation of filter response before realization. The utilization of Canonical Signed-Digit (CSD) coefficients allows high speed implementation of the digital filter hardware. Examples will be provided to show the efficiency of each design technique.

4.1 INTRODUCTION OF 2-D FIR FILTER DESIGN

In this chapter, we propose the utilization of McClellan transformation [42,41] and Genetic Algorithms to design 2-D FIR filters [35]. McClellan transformation reduces a 2-D filter design into a 1-D filter design problem. While the application of Genetic Algorithm yields the resultant filter coefficients to have CSD format which is suitable for high speed DSP applications.

4.1.1 MCCLELLAN TRANSFORMATION

The McClellan transformation transforms a 1-D FIR filter into 2-D FIR filter while maintaining its linear phase and transition characteristics. The design technique is divided into two parts.

Part 1
Design a 1-D FIR filter of the form

$$H_D(e^{j\omega}) = \sum_{n=0}^{N} b_n \cos(n\omega), \tag{4.1}$$

where $2N$ is the order of the filter, and $\{b_n\}$ is defined as the filter coefficients .T is assumed to be equal to one. This 1-D FIR filter is then transformed to a 2-D filter of the form

$$H_D(e^{j\omega_1}, e^{j\omega_2}) = \sum_{n=0}^{N} \sum_{m=0}^{M} b_{nm} \cos(n\omega_1)\cos(m\omega_2), \tag{4.2}$$

where $2N + 2M$ is the order of the filter, and $\{b_{nm}\}$ is the filter coefficients.

using the transformation outlined as blow

$$\cos(n\omega) = \sum_{p=0}^{P} \sum_{q=0}^{Q} t_{pq} \cos(p\omega_1)\cos(q\omega_2), \tag{4.3}$$

where t_{pq} is the unknown variables.

Part 2

In this part, the coefficients of the McClellan Transformation in Equation (4.3) are calculated so that the cutoff frequency of the 1-D FIR filter is transformed to the desired cutoff boundary of the 2-D FIR filter using the GA. The frequency response of a 2-D FIR filter is obtained by substituting Equation (4.1) into Equation (4.3). In this thesis, it is assumed that P = Q = 1.

4.1.2 1-D FILTER DESIGN FORMULATION

A 1-D FIR filter transfer function is defined as

$$H_D(z) = \sum_{n=0}^{2N} a_n z^{-n}, \tag{4.4}$$

where $\{a_n\}$ are the filter coefficients, and $2N$ is the order of the filter. The corresponding frequency response is given in Equation (1.4). By minimizing the following error function,

$$\begin{aligned} E_{contour} &= E_{l_\infty}(\omega) \\ &= \max \left| |H_I(\omega)| - |H_D(\omega)| \right|, \end{aligned} \tag{4.5}$$

we obtain the filter coefficients, where $H_I(\omega)$ and $H_D(\omega)$ are defined as the frequency responses of the ideal and designed filters. For low-pass filter design, the pass-band and stopband edges of an ideal filter can be obtained directly from the specification of the 2-D filter.

4.1.3 CONTOUR DESIGN FORMULATION

A 2-D contour is characterized by Equation (4.3). First order transformation is used which is in terms of cosine function,

$$F(\omega_1,\omega_2)=t_{00}+t_{10}\cos(\omega_1)+t_{01}\cos(\omega_2) \\ +t_{11}\cos(\omega_1)\cos(\omega_2), \tag{4.6}$$

where t_{00}, t_{01}, t_{10}, and t_{11} are the unknown variables, and ω_1,ω_2 are the angular frequency in the transformed 2-D space. Equating $F(\omega_1,\omega_2)$ to $\cos(\omega)$ performs the transformation,

$$F(\omega_1,\omega_2)=\cos(n\omega), \tag{4.7}$$

where ω is the angular frequency in the 1-D space. As an example, a near circular symmetric low-pass contour contains the following three properties [48].

1. $F(\omega_1,\omega_2)=F(\omega_2,\omega_1)$.
2. $\cos(\pi)$ must be mapped to $F(\pi,\pi)$.
3. $\cos(0)$ must be mapped to $F(0,0)$.

By solving Equation (4.6) and (4.7), along with these restrictions, we obtain $-t_{00}=t_{11}$, and $t_{01}=t_{10}=0.5$, which lead to the following objective function,

$$\begin{aligned} E_{contour} &= E_{l_x}(\omega) \\ &= \max\left|\begin{matrix} \cos(\omega)-t_{00}-0.5\cos(\omega_1) \\ -0.5\cos(\omega_2)+t_{00}\cos(\omega_1)\cos(\omega_2) \end{matrix}\right|. \end{aligned} \tag{4.8}$$

By minimizing Equation (4.8) with a similar Genetic Algorithm technique presented in Chapter 3, we obtain the remaining variables, t_{00} or t_{11}.

Similar procedures presented in Chapter 3 can be used to design the 2-D filter by replacing the cost functions of the Equations (4.8) and (4.5) . The procedures include randomly creating chromosomes and evaluating them with respect to the given cost function. Evaluated chromosomes are then manipulated by the genetic operators, reproduction, crossover, and mutation to produce offspring. In order to generate CSD coefficients, created offspring must satisfy a set of rules as described in Section 2.4. After that, offspring will be evaluated to obtain their fitness value and ready to enter into the next generation. The iteration of GA manipulation will continue until the maximum iteration is reached or the filter response satisfies the specification. The 2-D filter can be obtained by inserting the resultant coefficients and variables into Equation (4.9). Equation (4.9) is the characteristic equation of 2-D FIR filter which is defined as below. It is a substitution of the transformation Equation (4.6) into Equation (4.1),

$$H(e^{j\omega_1}, e^{j\omega_2}) = \sum_{n=0}^{N} b_n \left[\begin{array}{c} t_{oo} + t_{10}\cos(\omega_1) + t_{01}\cos(\omega_2) \\ +t_{11}\cos(\omega_1)\cos(\omega_2) \end{array} \right]. \qquad (4.9)$$

4.1.4 EXAMPLE

As an example, a Genetic Algorithm is utilized to design a 24th order 1-D FIR filter, and McClellan transformation coefficients individually. The ideal 1-D FIR filter in Equation (4.4) has the characteristic of being one in the pass-band $\Re_p \in \{\omega \leq 1.0 \text{ rad/sec}\}$ and zero in the stopband $\Re_s \in \{\omega \geq 2.0 \text{ rad/sec}\}$. Two hundred ternary bit streams are randomly generated during initialization, followed by the fitness evaluation by determining their minimax errors. The genetic operations and the three additional processes are performed repeatedly on the population of chromosomes as illustrated in Fig.25. The magnitude response of the resultant 1-D FIR filter is plotted in Fig.39, where $\omega_s = 2\pi$ rad/sec. The corresponding Canonic Signed-Digit coefficients and is included in Table 5.

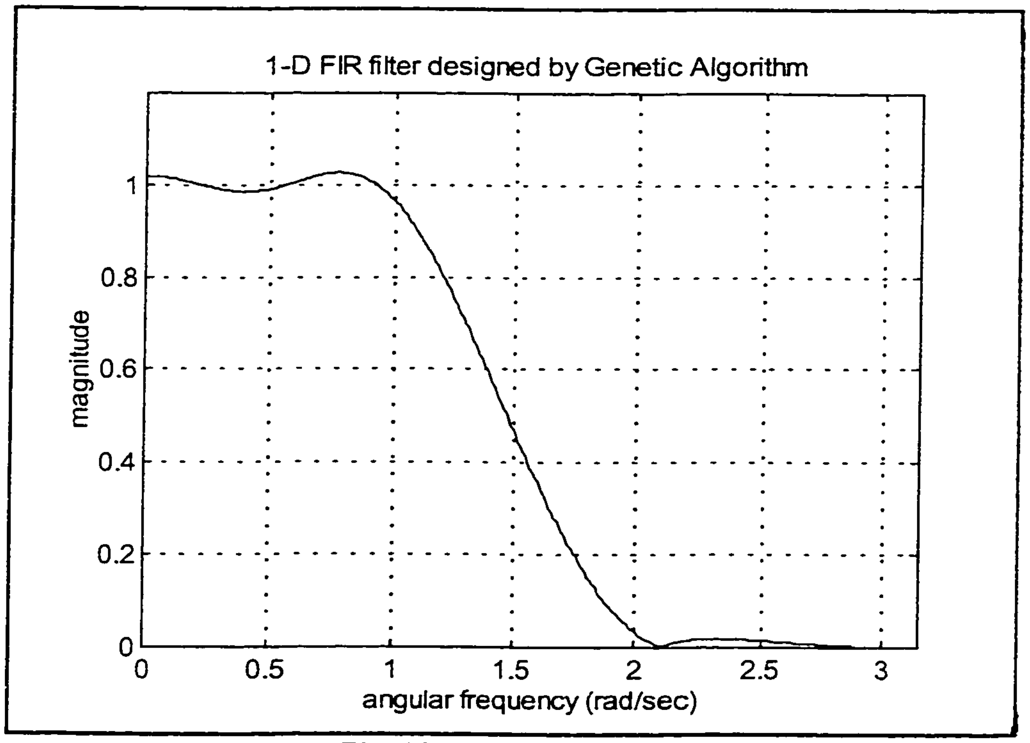

Fig. 39 : Designed 1-D FIR filter

coefficients	value
b(0)	$-2^{-1} + 2^{-3} - 2^{-6}$
b(1)	$-2^{0} - 2^{-3} + 2^{-10}$
b(2)	$-2^{-2} - 2^{-4} + 2^{-11}$
b(3)	$2^{0} - 2^{-2} - 2^{-8}$
b(4)	$-2^{-2} + 2^{-8}$
b(5)	$2^{-3} - 2^{-8} + 2^{-11}$
b(6)	$2^{0} + 2^{-8} - 2^{-11}$
b(7)	$-2^{-6} - 2^{-8} - 2^{-11}$
b(8)	$2^{-3} + 2^{-7} - 2^{-9}$
b(9)	-2^{-2}
b(10)	$-2^{0} - 2^{-2} + 2^{-4}$
b(11)	$2^{-6} + 2^{-9} - 2^{-11}$
b(12)	$2^{-1} - 2^{-7} + 2^{-11}$

Table 5 : 1-D FIR filter coefficient.

A similar technique is performed to obtain a near circular symmetric contour by determining t_{00} as described in the previous section, where fifteen ternary bit streams are used instead. The designed contours are shown in Fig.40. The corresponding Canonic Signed-Digit variables is included in Table 6.

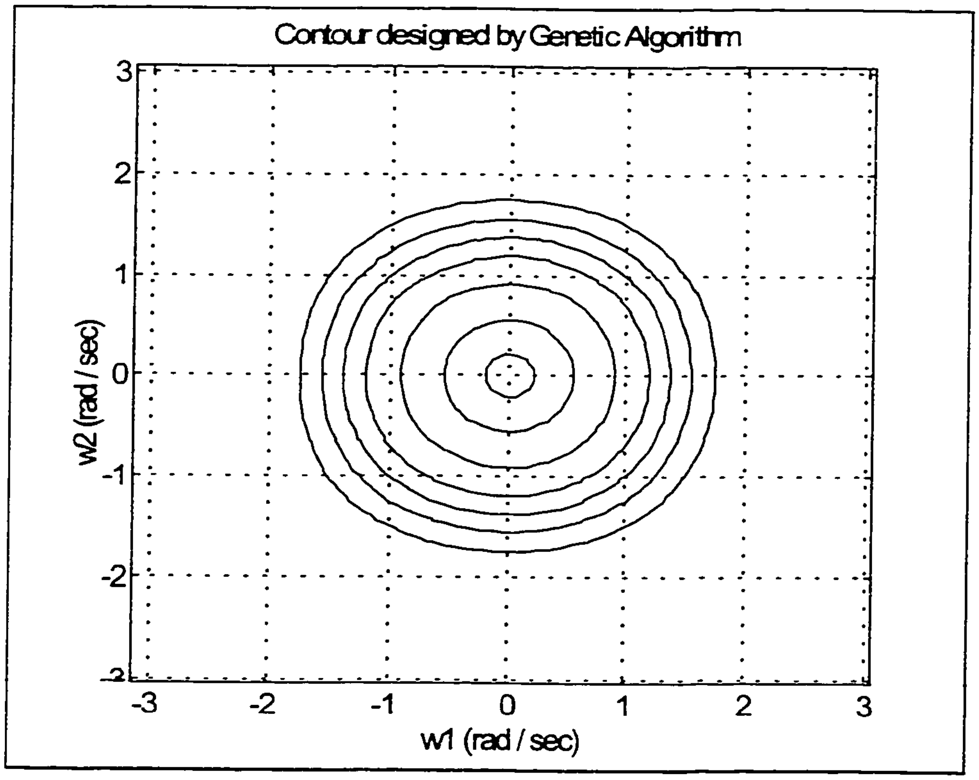

Fig. 40 : Designed circular contour

variables	value
t00	$-2^{-1} - 2^{-7} - 2^{-9}$
t01	2^{-1}
t10	2^{-1}
t11	$2^{-1} + 2^{-7} + 2^{-9}$

Table 6: McClellan transformation coefficients.

The final step, McClellan transformation is utilized to obtain a 2-D FIR filter, as shown in Fig.41. The transformed 2-D filter contains the characteristics of the 1-D filter and the circular symmetric contour.

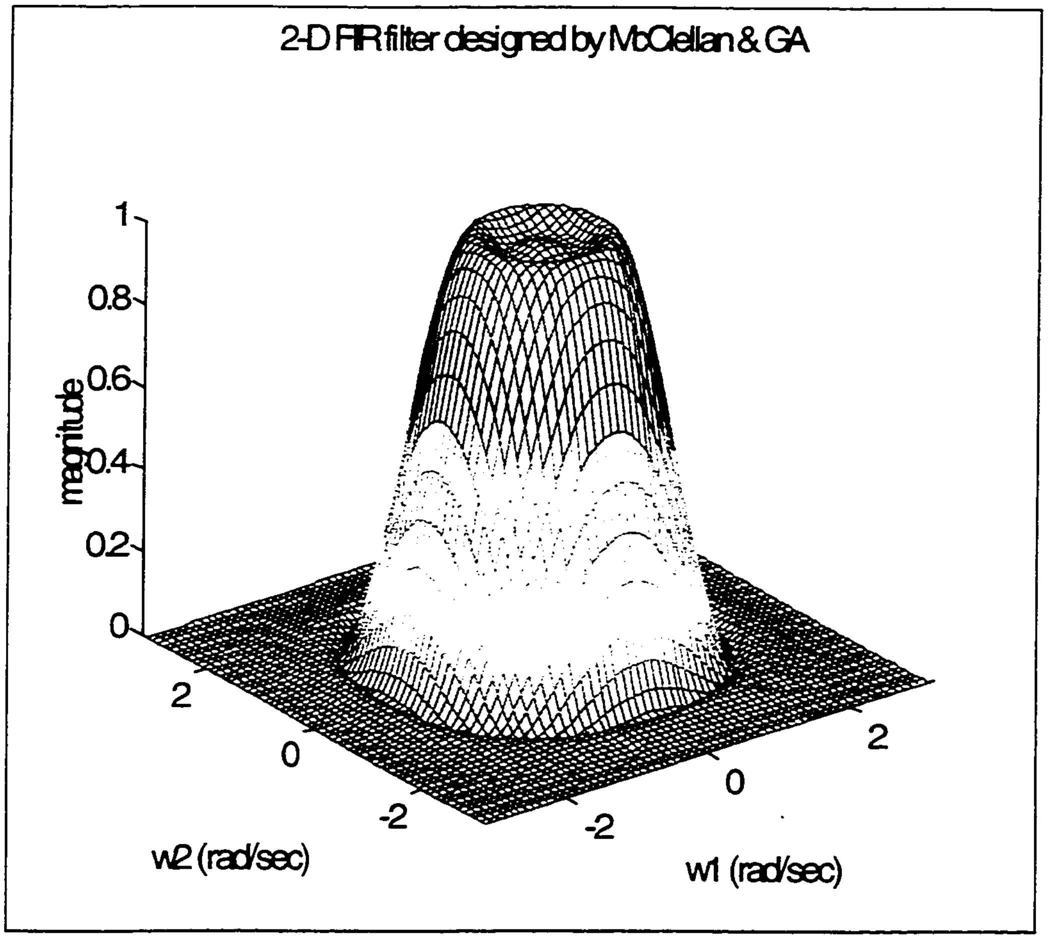

Fig. 41 : Resultant 2-D FIR filter

4.2 INTRODUCTION OF 2-D IIR FILTER DESIGN

In this section, we propose an algorithm to design 2-D IIR filter using discrete coefficients [36]. We utilize the design technique proposed by H.Safiri and M.Ahmadi [52], and enhance it further to generate discrete coefficient 2-D IIR filter while maintaining its stability. The use of the quantization and truncation process on the continuous coefficient will cause the filter to be unstable. An alternative technique will be presented to obtain discrete coefficients without the use of truncation and quantization. A low-pass filter, with fourteen bits wordlength, is provided as an example.

4.2.1 LOW-PASS FILTER STRUCTURE

In the proposed technique [52], the transfer function of a general 2-D filter has the following form:

$$\begin{aligned} H(z_1,z_2) = a + \frac{1}{2^{1+\beta}}(-1)^{a}\left[A_1(z_1,z_2) + (-1)^{I}(z_1)^{-M_1}(z_2)^{-M_2}A_2(z_1,z_2)\right] \\ \times\left[A_3(z_1,z_2) + (-1)^{J}(z_2)^{-M_3}A_4(z_1,z_2)\right]^{\beta}, \end{aligned} \tag{4.10}$$

where $A_k(z_1,z_2)$ for $1\leq k\leq 4$ the 2-D allpass filter is represented by

$$A(z_1,z_2) = z_1^{-N}z_2^{-N}\frac{D(z_1^{-1},z_2^{-1})}{D(z_1,z_2)}, \tag{4.11}$$

and $D(z_1,z_2)$ is defined as

$$D(z_1,z_2) = \sum_{i=0}^{N}\sum_{j=0}^{N} d_{ij}z_1^{-i}z_2^{-j}, \tag{4.12}$$

Parameters a, β, I, J, and M_i for $1 \le i \le 3$ are all binary values which have to be determined by the designer according to the filter type. To obtain the binary parameters of an Equation (4.10), one should consider four crucial points[3] (CP). Based on the design specification, one can use Table 7 to determine the values of the binary parameters.

CP	α	β	I	M_1	M_2	J	M_3
NONE	ZERO	ZERO	ONE	ZERO	ZERO	X	X
ALL	ZERO	ZERO	ZERO	ZERO	ZERO	X	X
1	ZERO	ONE	ZERO	ONE	ZERO	ZERO	ONE
2	ZERO	ONE	ZERO	ONE	ZERO	ONE	ONE
3	ZERO	ONE	ONE	ONE	ZERO	ZERO	ONE
4	ZERO	ONE	ONE	ONE	ZERO	ONE	ONE
1,2	ZERO	ZERO	ZERO	ONE	ZERO	X	X
1,3	ZERO	ZERO	ZERO	ZERO	ONE	X	X
1,4	ZERO	ZERO	ZERO	ONE	ONE	X	X
2,3	ZERO	ZERO	ONE	ONE	ONE	X	X
2,4	ZERO	ZERO	ONE	ZERO	ONE	X	X
3,4	ZERO	ZERO	ONE	ONE	ZERO	X	X
1,2,3	ONE	ONE	ONE	ONE	ZERO	ONE	ONE
1,2,4	ONE	ONE	ONE	ONE	ZERO	ZERO	ONE
1,3,4	ONE	ONE	ZERO	ONE	ZERO	ONE	ONE
2,3,4	ONE	ONE	ZERO	ONE	ZERO	ZERO	ONE

Table 7: Binary parameters of the general equation for different cases.

According to the above scheme, a 2-D IIR low-pass filter transfer function is defined as

$$H_{LP}(z_1,z_2) = \frac{1}{4}\left[A_1(z_1,z_2) + (z_1)^{-1}A_2(z_1,z_2)\right] \times \left[A_3(z_1,z_2) + (z_2)^{-1}A_4(z_1,z_2)\right]. \tag{4.13}$$

From Fig.42, $H_{LP}(z_1,z_2)$ is utilized using four allpass subfilters namely $A_1(z_1,z_2)$, $A_2(z_1,z_2)$, $A_3(z_1,z_2)$, and $A_4(z_1,z_2)$ along with appropriate delay elements as shown.

[3] Crucial points are located at each corner of the first quadrant 2-D z-plane. They correspond to the various pass-band and stopband boundaries.

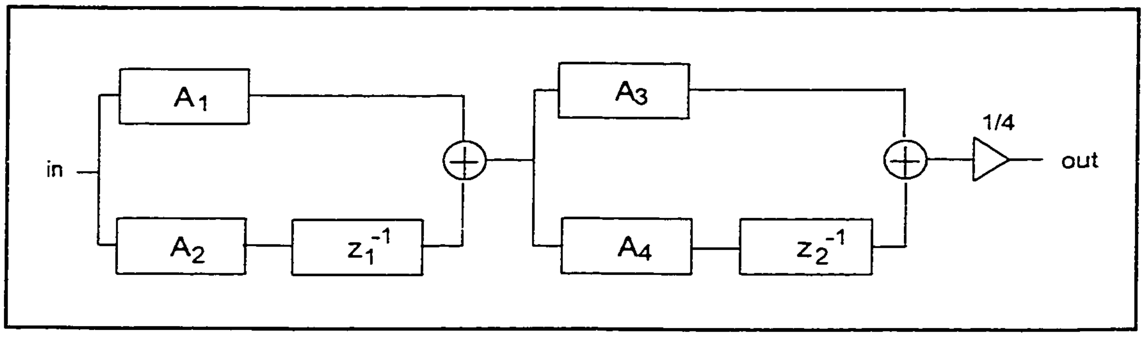

Fig.42 : Low-pass filter structure

4.2.2 FORMULATION OF THE DESIGN PROBLEM

The design of a stable 2-D IIR filter requires the use of a polynomial (transfer function) to impose restriction on filter coefficients. This polynomial is located on a 2-D s-plane that will require a transformation to map it into a 2-D *z* domain. Both, the transfer and transformation functions, are capable to maintain the stability of the filter.

A second order Very Strictly Hurwitz Polynomial (VSHP) [1] is used as a stable 2-D transfer function which contains the following form:

$$\begin{aligned} D(s_1,s_2) = {} & \left[(a-b)^2\gamma_2^2\,\delta_2^2\right]s_1s_2 + \{\gamma_2^2[(a-r)^2\sigma_2^2+\sigma_1^2]\}s_1 \\ & + \{\delta_2^2[(b-r)^2\sigma_2^2+\sigma_1^2]\}s_2 + \sigma_2^2\sigma_1^2 + g^2, \end{aligned} \tag{4.14}$$

where $\delta_2,\gamma_2,\sigma_1,\sigma_2,a,b,g,r$ are the unknown parameters, and $D(s_1,s_2)$ is in 2-D s-plane. The corresponding $D(z_1,z_2)$ is obtained through double bilinear transformation. By substituting $D(z_1,z_2)$ into Equation (4.11), allpass digital filter is then written as

$$A(z_1,z_2) = \frac{a_{00}+a_{10}z_1+a_{01}z_2+a_{11}z_1z_2}{a_{00}z_1z_2+a_{10}z_2+a_{01}z_1+a_{11}}, \tag{4.15}$$

where a_{00}, a_{10}, a_{01}, and a_{11} are the filter coefficients which are also functions of $\delta_2, \gamma_2, \sigma_1, \sigma_2, a, b, g, r$ parameters. Higher order subfilter can be obtained by cascading filters designed through Equation (4.14).

The low-pass filter transfer function is constructed by substituting Equation (4.15) into Equation (4.13). The objective function $E_M(\omega_1, \omega_2, \underline{\psi})$ is defined as:

$$E_M(\omega_1, \omega_2, \underline{\psi}) = |H_I(e^{j\omega_1 T}, e^{j\omega_2 T})| - \left|H_D\left(e^{j\omega_1 T}, e^{j\omega_2 T}, \underline{\psi}\right)\right|, \tag{4.16}$$

where H_I and H_D = $H_{LP}|(z_1 = e^{j\omega_1 T}, z_2 = e^{j\omega_2 T})$ are the magnitude response of the ideal and designed filters, and $\underline{\psi}$ is the unknown parameter vector $[\delta_2, \gamma_2, \sigma_1, \sigma_2, a, b, g, r \ldots]$. Readers are referred to [52] for the error function of the phase response. The unknown subfilter coefficients can be obtained by minimizing the following least mean square error function:

$$E_{\ell_2}(\omega_1, \omega_2, \underline{\psi}) = \left\{ \sum_{\omega_1, \omega_2 \in I_{ps}} \sum E_M^2\left(\omega_1, \omega_2, \underline{\psi}\right) \right\}^{1/2}, \tag{4.17}$$

where I_{ps} is a set of frequencies in the pass-band and stopband of the filter.

4.2.3 GENETIC ALGORITHM AND ENCODING

We propose the use of the Genetic Algorithm to optimize filter response. Optimization is performed on the unknown parameters in Equation (4.15) that will result in stable filter coefficients. In order to have control over the coefficient wordlength, a set of numbers will be predefined for the GA to search among them. The numbers are represented by the following equation.

$$p_m = \sum_{i=-K/2}^{K/2} c(i)2^i, \qquad c(i) = -1, 0, \text{ or } 1 \quad \text{and} \quad p_m \neq 0 \qquad (4.18)$$

where 'K + 1' is the wordlength of the parameter which is an even number. The sum of squares of c(i) is defined as the number of terms in p_m. Once 'K' is determined, a set of discrete numbers can be pre-calculated, $\{p_1, p_2, p_3, \ldots, p_M\}$, where 'M' is defined as the total number of valid numbers.

A potential drawback of this algorithm is that the coefficients may result in a much higher wordlength due to the small changes in the number of power-of-two terms in parameters. For example, a two term strategy results in a minimum coefficient wordlength of twelve bits, and a three term strategy results in a minimum coefficient wordlength of eighteen bits from Equation (4.14).

4.2.4 EXAMPLE

In this example, parameter wordlength of five (K=4) and the maximum of two terms are used to create a set of discrete numbers,

$$P \in \{-3.0, -2.0, -1.5, -1.0, -0.75, -0.5, -0.25, 0.25, 0.5, 0.75, 1.0, 1.5, 2.0, 3.0\}.$$

These fourteen numbers are the domain for the unknown parameters. The next step is to set-up the Genetic Algorithm to perform parameters optimization on a given ideal response (envelope). Unlike the Signed-Digit Encoding and Mixed Encoding presented in the previous chapters, we use the exact discrete numbers to be the element of chromosome. For example, a chromosome contains the following discrete numbers,

[-0.25 -2.00 -1.50 0.50 0.75 -0.25 -2.00 -1.00 1.00 1.00 -0.50 1.00].

It is more desirable to use discrete number encoding to maintain a relatively small chromosome length. The reason we can possibly do so in this example, but not in the others, is because the parameter search space is relatively smaller than the coefficient search space shown in Chapter 3, and we have correspondingly more unknown parameters than in the previous 1-D FIR filter design examples. In this example, the resultant coefficient wordlength is expected to be fourteen bits long. An ordinary Genetic Algorithm [23] with fitness scaling can be used for this design. At the beginning, 80 chromosomes are created by randomly selecting the discrete numbers. The manipulation of crossover will be performed on the chromosomes, rather than the individual coefficients. The mutation operator will check on every chromosome element and replace it by randomly choosing a number among the discrete set under a specified probability. The corresponding flow diagram is illustrated in Fig. 5.

As an example, four 4th order allpass subfilters are constructed to form a low-pass filter, $H_{LP}(z_1,z_2)$. The ideal filter in Equation (4.16) has the characteristic of being one in the pass-band

$$\Re_p \in \left\{ \sqrt{\omega_1^2+\omega_2^2} \le 0.8 \text{ rad/sec} \right\},$$

and zero in the stopband

$$\Re_s \in \left\{ \sqrt{\omega_1^2+\omega_2^2} \ge 2.0 \text{ rad/sec} \right\}.$$

The crossover and mutation rates are 0.9 and 0.1 respectively. The Roulette Wheel parent selection scheme is used in Reproduction operator. The magnitude response of the designed filter is plotted as shown below, where $\omega_s = 2\pi$ rad/sec..

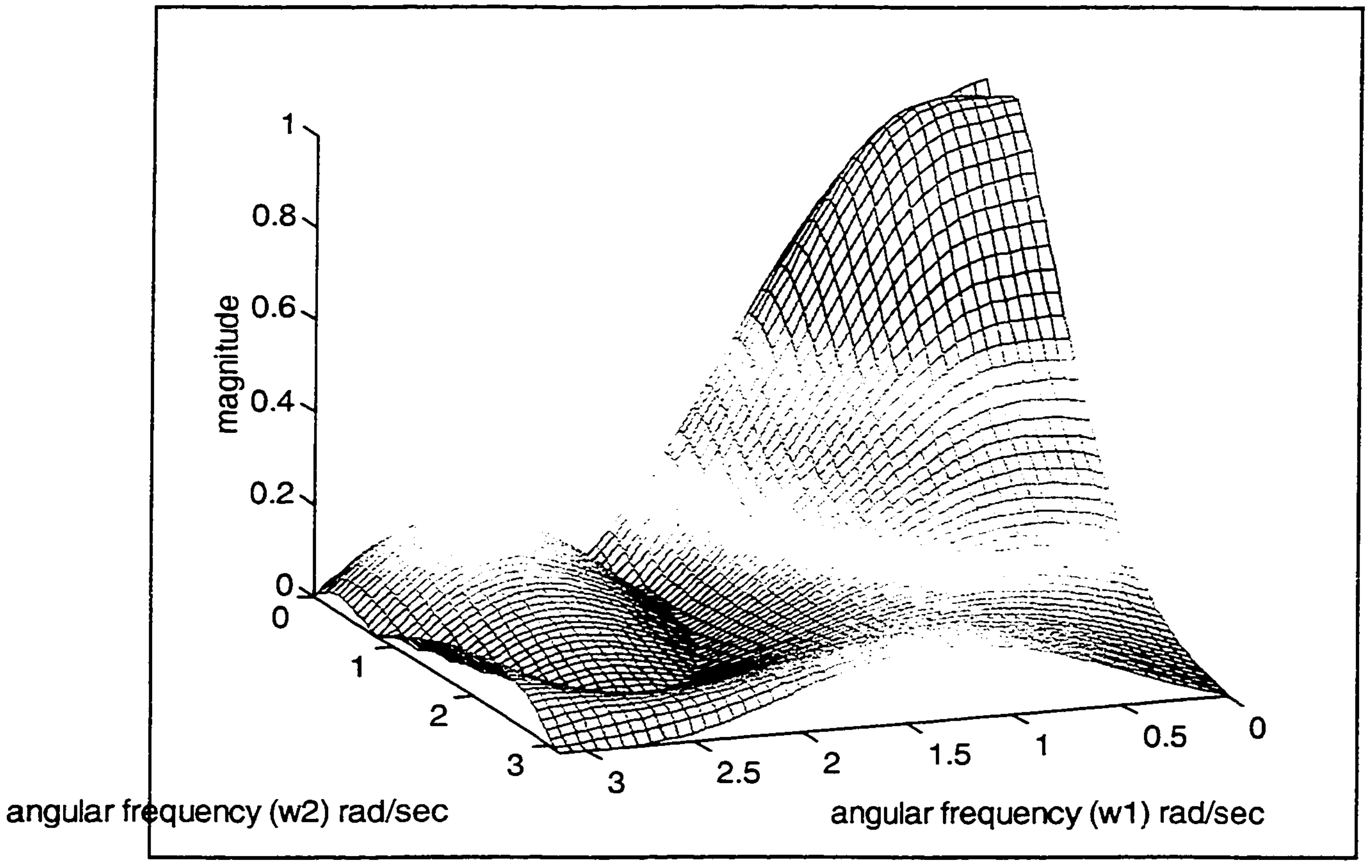

Fig.43: The magnitude response of the filter designed
with discrete coefficients.

Coefficients	Infinite-precision	16 bit wordlength
a(1,1)	34.31250000	$2^5+2+2^{-2}+2^{-4}$
a(1,2)	14.06250000	$2^3+2^2+2+2^{-4}$
a(1,3)	3.937500000	$2+2^0+2^{-1}+2^{-2}+2^{-3}+2^{-4}$
a(1,4)	3.937500000	$2+2^0+2^{-1}+2^{-2}+2^{-3}+2^{-4}$
a(1,5)	10.32812500	$2^3+2+2^{-2}+2^{-4}+2^{-6}$
a(1,6)	0.078125000	$2^{-4}+2^{-6}$
a(1,7)	-7.671875000	$-2^2-2-2^0-2^{-1}-2^{-3}-2^{-5}-2^{-6}$
a(1,8)	-1.921875000	$-2^0-2^{-1}-2^{-2}-2^{-3}-2^{-5}-2^{-6}$
a(2,1)	42.328125000	$2^5+2^3+2+2^{-2}+2^{-4}+2^{-6}$
a(2,2)	14.765625000	$2^3+2^2+2+2^{-1}+2^{-2}+2^{-6}$
a(2,3)	-25.734375000	$-2^4-2^3-2^0-2^{-1}-2^{-3}-2^{-4}-2^{-5}-2^{-6}$
a(2,4)	-17.296875000	$-2^4-2^0-2^{-2}-2^{-5}-2^{-6}$
a(2,5)	16.453125000	$2^4+2^{-2}+2^{-3}+2^{-4}+2^{-6}$
a(2,6)	1.140625000	$2^0+2^{-3}+2^{-6}$
a(2,7)	-4.359375000	$-2^2-2^{-2}-2^{-4}-2^{-5}-2^{-6}$
a(2,8)	-3.671875000	$-2^1-2^0-2^{-1}-2^{-3}-2^{-5}-2^{-6}$
a(3,1)	15.878906250	$2^3+2^2+2^1+2^0+2^{-1}+2^{-2}+2^{-3}$ $+2^{-8}$
a(3,2)	6.8085937500	$2^2+2^1+2^{-1}+2^{-2}+2^{-5}+2^{-6}+2^{-7}$ $+2^{-8}$
a(3,3)	9.5585937500	$2^3+2^0+2^{-1}+2^{-5}+2^{-6}+2^{-7}+2^{-8}$
a(3,4)	4.0039062500	2^2+2^{-8}
a(3,5)	22.203125000	$2^4+2^2+2^1+2^{-3}+2^{-4}+2^{-6}$
a(3,6)	-17.796875000	$-2^4-2^0-2^{-1}-2^{-2}-2^{-5}-2^{-6}$
a(3,7)	-13.046875000	$-2^3-2^2-2^0-2^{-5}-2^{-6}$
a(3,8)	10.953125000	$2^3+2^1+2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-6}$
a(4,1)	6.890625000	$2^2+2^1+2^{-1}+2^{-2}+2^{-3}+2^{-6}$
a(4,2)	-3.234375000	$-2^1-2^0-2^{-3}-2^{-4}-2^{-5}-2^{-6}$
a(4,3)	4.359375000	$2^2+2^{-2}+2^{-4}+2^{-5}+2^{-6}$
a(4,4)	-3.515625000	$-2^1-2^0-2^{-1}-2^{-6}$
a(4,5)	15.156250000	$2^3+2^2+2^1+2^0+2^{-3}+2^{-5}$
a(4,6)	-6.843750000	$-2^2-2^1-2^{-1}-2^{-2}-2^{-4}-2^{-5}$
a(4,7)	-8.906250000	$-2^3-2^{-1}-2^{-2}-2^{-3}-2^{-5}$
a(4,8)	5.093750000	$2^2+2^0+2^{-4}+2^{-5}$

Table 8: Discrete Coefficients with one sign bit, five integer bits, and eight fractional bits.

$$A_k(z_1,z_2)=\frac{a(k,4)\,z_1z_2+a(k,3)\,z_1+a(k,2)\,z_2+a(k,1)}{a(k,1)\,z_1z_2+a(k,2)\,z_1+a(k,3)\,z_2+a(k,4)}\times\frac{a(k,8)\,z_1z_2+a(k,7)\,z_1+a(k,6)\,z_2+a(k,5)}{a(k,5)\,z_1z_2+a(k,6)\,z_1+a(k,7)\,z_2+a(k,8)}\quad,\ 1\le k\le 4$$

4.3 SUMMARY

In this chapter, we have presented the design of 2-D FIR and IIR filters with Genetic Algorithms. By using the McClellan Transformation, a 2-D FIR filter design is simplified into two phases, 1-D filter and contour designs, and transformation. The utilization of the modified Genetic Algorithm presented in Chapter 3 can guarantee to produce Canonic Signed-Digit coefficients and contour variables. We have also presented the design of 2-D IIR filters with discrete coefficients in the previous section. The proposed design methodology of 2-D IIR filter has a unique structure of cascading all-pass subfilters to generate a low-pass filter prototype. The utilization of Horwitz Polynomial and Genetic Algorithms is presented to guarantee filter stability and to obtain discrete coefficients of all-pass subfilters. A design example has been provided by this proposed methodology.

CHAPTER 5. CONCLUSION

5. SUMMARY AND CONCLUSION

In this thesis, we have proposed the design of 1-D FIR, 2-D FIR, and 2-D IIR filters with finite-wordlength coefficients using Genetic Algorithms. We have utilized different coefficient encoding schemes, parent selection schemes, and objection functions to evaluate the performance of various techniques. Different orders of 1-D low-pass FIR filters have been presented. As the order of the filter gets higher, it requires more trials and iterations to design a filter. For this reason, Fitness Scaling, Multi-point Crossover, and Non-uniform mutation rate are proposed in this thesis as an enhancement in Genetic Algorithms. Our results have shown that minimax strategy outperforms LMS error function in all circumstances. Ranking and Roulette Wheel parent selections have similar performance in terms of attainting optimum filter response. We have also found that Signed-Digit encoding scheme has a higher convergence rate than Mixed encoding. Beside the evaluation of our proposed filter design techniques, we have also presented a methodology to design 1-D FIR filters with Double-Base number system. The utilization of this number system is possible to reduce the use of hardware resources which lead to high speed signal processing [25].

In turn, we have presented the use of iterative methods to design 2-D FIR and IIR filters. In 2-D IIR filter design, the filter structure is cascading of all-pass sub-filters. This structure has the advantage of low sensitivities towards quantization and truncation errors. For the stability reasoning, the algorithm utilizes VSHP to ensure the filter's stability. Our study has shown that discrete filter coefficients with arbitrary specification can be obtained by the use of Genetic Algorithms. The quantization of filter coefficients is removed from the design process. The resultant discrete coefficients can be further truncated into smaller wordlength. Our design experiences have confirmed that optimized filter response can be easily obtained using this filter structure with our proposed technique. Regarding the lowpass filter design, the proposed algorithm with the transfer function in Equation (4.8) can also be used to design bandpass, highpass, and bandstop filters. For 2-D FIR filter design, the proposed techniques have the advantage of eliminating coefficient round off error by eliminating the quantization process. The utilization of McClellan transformation and Genetic Algorithm have further simplified the design procedures and provided fast implementation of this transform. Our simulation results suggested that the transformation of a 1-D FIR filter can result in the design of a near circular symmetry of 2-D lowpass filter.

One of the filter design consideration using our proposed GA techniques is the number of non-zero digits must be a lot less than half of the coefficient wordlength. If this is not the case, chromosomes will be spending most of the time looping at CSD Check, Crossover, Mutation, and Maximum Failure modules. Eventually, the chromosomes will be reassigned by the Reset Coefficient module. This in turn will destroy the established schema. Fortunately, this rarely happens in our case since the hardware complexity is highly dependent on the non-zero digits in each of the coefficients. The restriction on number of non-zero digits can reduce the hardware complexity.

Genetic Algorithm is still a relatively new optimization technique. More research is needed in further investigation regarding the relationship between FIR and IIR filters design and GA to enable the design of filters exceed the 60th order.

REFERENCES

1. M. Ahmadi, "Design of 2-Dimensional Recursive Digital Filters," *Control and Dynamics System*, vol. 78, pp. 131 - 181, 1996.

2. M. Ahmadi, and A. Chottera, "Improved method for the design of 2-D FIR digital filters with circular and rectangular cut-off boundary using kaiser window", *Canadian Electrical Engineering Journal*, vol. 8, pp. 3 - 8, Jan 1983.

3. A. Antonious, *Digital Filters: Analysis, Design, and Applications*: McGraw Hill, 1993.

4. T. Arslan, D.H. Horrocks, "A Genetic Algorithm for the Design of Finite Word Length Arbitrary Response Cascaded IIR Digital Filters", *Genetic Algorithm in Engineering Systems: Innovations and Aplications*, no.414, pp.276-281, Sept. 1995.

5. Takashi Asaida, Taku Kihara, Hideaki Murayama, Masao Naito, Shige Morikawa, and Ian Sheldon,"Digital signal processing technology for broadcast TV cameras",*IEE Conference Publication International Broadcasting Convention*, pp. 442 - 447, July 1992.

6. F. Ashrafzadeh, B. Nowrouzian, "Crossover and Mutation in Genetic Algorithms Employing Canonical Signed-Digit Number System", *Proceeding of the 4th Midwest Symposium on Circuits and Systems*, pp.702-705, Aug. 1997.

7. Acyl Benslimane, and Pierre Siohan, "Algorithm for the design of optimal finite wordlength 2-D FIR digital filters", *Proceedings - ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 553 - 556, 1986.

8. N. Benvenuto, M. Marchesi, A. Uncini, "Applications of Simulated Annealing for the Design of Special Digital Filters", *IEEE Transcations on Signal Processing*, Vol.40, No.2, pp.323-333, February 1992.

9. M. T. Boraie, M. Ahmadi, S. Erfani, and V. Ramachandran,"Design of 2-dimensional recursive digital filters with integer coefficients by successive discretization and reoptimization", *Midwest Symposium on Circuits and Systems Conference Proceedings - 28th Midwest Symposium on Circuits and Systems*, pp. 294 - 296, 1985.

10. L. T. Bruton, and N. R. Bartley, "General-purpose computer program for the design of two-dimensional recursive filters - 2DFIL", *Circuits, Systems, and Signal Processing*, vol. 3, pp. 243 - 264, 1984.

11. B.D. Bunday, *Basic Optimization Methods*: Arnold, 1984.

12. R. Cemes, and Ait-Boudaoud, "Genetic approach to design of multiplierless FIR filters", *Electron. Lett.*, vol.29, no.24, pp.2090-2091, Nov. 1993.

13. A. Chottera, G.A. Jullien, "A Linear Programming Approach to Recursive Dgital Filter Design with Linear Phase," IEEE trans. on Circuits&Sytems, vol:CAS-29, no.3, March 1982

14. A.T. Chottera, and G.A. Jullien, "Design of Two-Dimensional Recursive Digital Filters Using Linear Programming", *IEEE Trans. on Circuits And Systems*, vol. CAS-29, no.12, pp.817-826, Dec 1982.

15. William J. Dally, "High-performance VLSI quaternary serial multiplier", *Proceedings - 1987 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pp. 649 - 653, 1987.

16. V. S. Dimitrov, G.A. Jullien and W.C. Miller, 1999, "Theory and Applications of the Double-Base Number System", *IEEE Trans. Computers*, Vol. 48, 10, pp. 1098-1106.

17. R. J. Fone, K. K. Pang, and D. B. Keogh, "Analysis of the canonic signed digit coding technique for implementing digital filters without multipliers ", *20th International Electronics Convention & Exhibition of the Institution of Radio and Electronics Engineers Australia, Digest of Papers*, pp. 1000 - 1003, 1985.

18. C.M. Fonseca, P.J. Fleming, "Genetic Algorithm for Multiobjective Optimization: Formulation, Discussion and Generalization", *Dept. Automatic Control and Systems Eng. University of Sheffield, Sheffeld S1 4DU, U.K.*

19. Hong Gao, Liren Zhao, Duanyi Xu, and Timothy J. Connors, "Video CD playback board for audio CD player", *Proceedings of SPIE - The International Society for Optical Engineering Fourth Int. Symposium on Optical Storage*, vol. 2931, pp. 185 - 189, April 1996.

20. Paolo Gentili, Francesco Piazza, Aurelio Uncini, "Efficient Genetic Algorithm Design for Power-Of-Two FIR filters", *Proceeding - IEEE International Conference on Acoustic, Speech, and Signal Processing*, vol.2, pp.1268-1271, May 1995.

21. D.E. Goldberg, *Genetic algorithm in search, optimization, and machine learning*, Addison: Wesley, 1989.

22. John Goodenough, Richard J. Meacham, Jonathan D. Morris, N. Luke Seed, and Peter A. Ivey, "Single chip video signal processing architecture for image processing, coding, and computer vision", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 5, pp. 436 - 444, Oct 1995.

23. J. Holland: 'Adaption in natural and artifical systems' (University of Michigan Press, 1975).

24. K. Hwang, *Computer Arithmetic, Principles, Architecture, and Design*. New York: Wiley, 1979.

25. G.A. Jullien, V.S. Dimitrov, B. Li, W.C. Miller, A. Lee, and M. Ahmadi, "A Hybrid DBNS Processor for DSP Computation", ISCAS'99, I-5-8.

26. Michitaka Kameyama,Shoji Kawahito,and Tatsuo Higuchi, "Multiplier chip with multiple-valued bidirectional current-mode logic circuits", *Computer,* vol. 21, pp. 43 - 56, Apr 1988.

27. M. Kawamata, J. Imakubo, T. Higuchi, "Optimal Design Method of Separable-Denominator Two-Dimensional Digital Filters Based on a Genetic Algorithm", *Electronic and Communications in Japan*, Part 3. vol.78, no.12, pp.70-80, 1995.

28. M. Kawamata, J. Imakubo, T. Higuchi, "Optimal Design Method of 2-D IIR Digital Filters Based on a Simple Genetic Algorithm", *IEEE*, pp.780-784, 1994.

29. R. King, M. Ahmadi, R. Gorgui-Nguib, A. Kwabwe, M. Azimi-Sadjadi, *Digital Filtering in One and Two Dimensions Design and Applicaions*: Plenum, 1989.

30. S. Kirkpatrick, C.D. Gelatt, Jr., M.P. Vecchi, "Optimization by Simulated Annealing", *Science*, vol.220, no.4598, pp.671-679, May 1983.

31. M. Kishida, N. Hamada, "Design of 2-D IIR Filter Using the Genetic Algorithm", *IEICE Trans. Fundamentals*, vol.E79-A, no.1, pp.131-133, Jan.1996.

32. A. Kosir, J.F. Tasic, "Genetic Algorithms and Filtering", *Genetic Algorithm in Engineering Systems: Innovations and Aplications*, no.414, pp.343-348, Sept. 1995.

33. A.Lee, M.Ahmadi, V.Ramachandran, C.S.Gargour, "Design of Fractional Delay Operator", *Proc. of the World Automation Congress*, pp657-662, May1998.

34. A.Lee, M.Ahmadi, G.A.jullien, R.S.Lashkari, W.C.Miller, "Design of 1-D FIR Filters with Genetic Algorithms", *Proc. of IEEE ISCAS 1999.* Vol.III, pp 295-298.

35. A.Lee, M.Ahmadi, R.S.Lashkari, "Design of 2-D FIR Filter With McClellan Transformation and Genetic Algorithms", Proc. *of 2nd Inter. Symp. on Artificial Intelligence Adaptive System.*, pp51-58, March 1999.

36. A.Lee, M.Ahmadi, R.S.Lashkari, "Design of Stable 2-D Recursive Filters Using Power-of-Two Coefficients", Proc. *of IEEE international Conf. on Electronics Circuits and Systems*, pp409-412, Sept. 1998.

37. Yong Ching Lim, "Efficient special purpose linear programming for FIR filter design", *IEEE Transactions on Acoustics, Speech, and Signal Processing,* vol. ASSP-31, pp. 963 - 968, August 1983.

38. M. Morrid Mano, Computer Engineering Hardware Design, Prentice-Hall, 1988.

39. F.C. Marshall,"Approximation of general nonrecursive filters using nonlinear programming", *Nav Undersea Cent, San Diego, Calif Source: Metallurgia Italiana Asilomar Conf on Circuits, Syst, and Comput, 7th, Annu, Conf Rec,*pp.518-521,Nov 1973.

40. Thomas G. Marshall, and Yeunung Chen, "Branch and bound algorithm for selecting quantization order in cascade FIR digital filter design", *National Bureau of Standards, Special Publication Midwest Symp on Circuits and Syst*, pp. 494 - 498, June 1979.

41. Wolfgang F.G. Mecklenbrauker, Russell M. Mersereau, "McClellan Transformations for Two-Dimensional Digital FIltering: II - Implementation", *IEEE Trans. Circuits and Syst.*, vol. CAS-23, no.7, pp.414-422, July 1976.

42. Russell M. Mersereau, Wolfgang F.G. Mecklenbrauker, "McClellan Transformations for Two-Dimensional Digital FIltering: I - Design", *IEEE Trans. Circuits and Syst.*, vol.CAS-23, no.7, pp.405-413, July 1976.

43. Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs, third, Revised and Extended Edition*: Springer, 1996.

44. W. J. Oh, Y.H. Lee, "A Design and Implementation of Programmable Multiplierless FIR Filters with Power-of-Two Coefficients", *Proceeding - IEEE International Symposium on Circuits and Systems*, vol.1, pp.88-91, May 1993.

45. Hideki Ohashi, Nobuyuki Tanaka, Mitsuo Fujimoto, Takashi Kuroda, and Tetsuji Yamaguchi, "Development of a DSP for speech signal processing and applications", *Digest of Technical Papers - IEEE International Conference on Consumer Electronics Proceedings of the 1998 17th Conference on Consumer Electronics*, pp. 412 - 413, Jun 2-4 1998.

46. A.V. Oppenheim, R.W. Schafer, *Discrete-Time Signal Processing*: Prentice Hall, 1989.

47. Michael S. Paterson, and Uri Zwick, "Shallow multiplication circuits", *Proceedings - Symposium on Computer Arithmetic Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pp. 28 - 34, Jun 1991.

48. Soo-Chang Pei, Jong-Jy Shyu, "Design of 2-D FIR Digital Filters by McClellan Transformation and Least Squares Eigencontour Mapping", *IEEE Trans. Circuits and Syst. - II: Analog and Digital Signal Processing*, vol.40, no.9, pp.546- ,Sept. 1993.

49. V. Ramachandran, C.S. Gargour, M. Ahmadi, M.T.H. Boraie, "Direct Design of Recursive Digital Filters Based on a New Stability Test", *Journal of the Franklin Institute Pergamon Press Ltd.*, vol.318, no.6, pp.407-413, Dec. 1984.

50. V. Ramachandran, M. Ahmadi, and S. Golikeri, "Min-max design of 2-D recursive digital filters satisfying prescribed magnitude and group delay specifications", *Digital Techniques in Simulation, Communication and Control, Proceedings of the IMACS European Meeting*, pp.129-132, 1985.

51. F. Russo, "Recent advances in fuzzy techniques for image enhancement", *IEEE Instrumentation & Measurement Magazine*, vol. 1, pp. 29 - 31, Dec 1998.

52. H. Safiri, M. Ahmadi, and V. Ramachandran, "Design of Stable, Causal 2-D Digital Filters Using Real Coefficient 2-D All-Pass Building Blocks," *IEEE Trans. Circuits Syst.*, vol. 44, no. 5, pp. 409 - 412, May 1997.

53. H. Samueli, "An Improved Search Algorithm for the Design of Multiplierless FIR filters with Powers-of-Two Coefficients", *IEEE Trans. Circuits and Syst.*, vol.36, no.7, pp.1044-1047, July 1989.

54. Akira Shindo, Yoshihiro Sasajima, Tomoichi Takahashi, Tomonori Aoyama, and Sadayasu Ono, "Nonlinear phase FIR filter design techniques", *Proceedings - IEEE International Symposium on Circuits and Systems Proc IEEE Int Symp Circuits Syst,* pp. 9 - 12, July 1979.

55. P. Siohan, D. Pele, C. Roche, "Iterative algorithm for the minimax approximation of two-dimensional FIR digital filters", *Proceedings - IEEE International Symposium on Circuits and Systems IEEE International Symposium on Circuits and Systems 1989, the 22nd ISCAS*, v 3, pp.1684-1687,May 1989 .

56. S. Sriranganathan, D.R. Bull, D.W. Redmill, "Design of 2-D Multiplierless FIR filters Using Genetic Algorithms", *Genetic Algorithms in Engineering Systems:Innovatoins and Applications*, no.414, pp.282-286, Sept. 1995.

57. D. Suckley, "Genetic Algorithm in the Design of FIR filters", *IEE Proceeding-G*, vol.138, no.2, pp.234-238, April 1991.

58. G.K. Sun, and J.F. Young, "Hydration reactions in autoclaved DSP cements", *Advances in Cement Research*, vol. 5, pp.163 - 169, Octomber 1993.

59. K. Suzuki, H. Ochi, S. Kinjo, "Design of FIR filter using CSD with minimum number of registres", *Proceedings of the 1996 IEEE Asia Pacific Conference on Circuits and Systems*, pp.227-230, Nov 1996.

60. K.S. Tang, K.F. Man, S. Kwong, Q. He, "Genetic Algorithms and their Applications", *IEEE signal proceesing magazine*, pp.22-37, Nov. 1996.

61. G.Wade, A. Roberts, G. Williams, "Multiplier-less FIR filter design using a genetic algorithm", *IEE Proc.-Vis. Image Signal Process.*, vol.141, no.3, pp.175-180, June 1994.

62. Eric A. Wan, and Alex T. Nelson, "Removal of noise from speech using the dual EKF algorithm", *Proceedings Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 1, pp. 381 - 384, May 1998,

63. Stephen G. Wilson,"Digital Moudulation and Coding", 1996, Prentice-Hall Inc..

64. D.J. Xu, M.L. Daley, "Design of Optimal Digital Filter Using a Parallel Genetic Algorithm", *IEEE Trans. Circuits and Syst. - II: Analog and Digital Signal Processing*, vol.42, no.10, pp.673-675, Oct. 1995.

65. M. Yasuda, T. Nishio, M. Toyokura, A. Kawamura, M. Fujiwara, and T. Akiyama, "MPEG2 video decoder and AC-3 audio decoder LSIS for DVD player", *Digest of Technical Papers - IEEE*, pp.114 - 115, June 1997.

66. T. S.Yu, A. Gruber, and P. Wang, "Adaptive doppler parameter measurement system for EW applications", *IEEE 1989 National Aerospace and Electronics Conference*, vol. 3, pp. 1153 - 1158, May 1989.

67. Q. Zhao, Y. Tadokoro, "A Simple Design of FIR Filters with Power-of-Two Coefficients", *IEEE Trans. Cirucits and Syst.*, vol.35, no.5, pp.566-570, May 1988.

APPENDIX I

Roulette Wheel Parent implementation algorithm

```
*selected_chrom = Roulette_Wheel_Parent_Selection(*fitness)

        /* chromosomes selection performs N number of times */
        for i = 0 to (N number of chromosome)-1
                total = Sum(*fitness);
                anumber = Random(0, total);

                acc = 0;
                /* accumult 'acc' til it is larger than or equal to total */
                for j = 0 to (N number of chromosome) - 1
                        acc = acc + *(fitness + j);
                        if ( acc >= anumber)
                                Store_Chrom(j, *selected_chrom);
                                break;
                        end if
                end for
        end for

end Roulette_Wheel_Parent_Selection
```

where

Roulette_Wheel_Parent_Selection(*fitness)

- is a procedure which takes a parameter that contains a pointer to the fitness values. Selected chromosomes indexes will be stored in an array represented by a pointer named *selected_chrom.

Sum(*fitness)

- is a procedure that takes on a parameter which is a pointer that contains the fitness value of each corresponding chromosomes. The output value is the sum of all fitness values that is stored in a variable named total.

Store_Chrom(j, *selected_chrom)

- is a procedure that places the selected index, 'j', into the array, named by the pointer selected_chrom.

Crossover operator implementation algorithm

```
**offspring = Crossover(**chrom, *selected_chrom)
    index = 0;

    /* perform crossover for ½ N number of times */
    loop while(index < (N number of chromosome))

        /* temp_a and temp_b are the index of selected chromosomes */
        temp_a = selected_chrom(index);
        temp_b = selected_chrom(index + 1);
        *offspring = Exchange_Digits(*chrom[temp_a], *chrom[temp_b]);
        index = index + 2;

    end loop

end Crossover
```

where

**chrom - is a double pointer that corresponds to a population.

*selected_chrom - is a pointer that contains the order of selection from Reproduction operator.

The Exchange_Digits(*,*) procedure is implemented as follow:

```
**offspring = Exchange_Digits(*chromA, *chromB)
        static int index = 0;

        /* select a cutting point between the chromosomes */
        anumber = Random(1, LengthOfChrom - 1);

        /* obtain probability for both chromosomes, *chromA and *chromB*/
        **offspring= Copy(*chromA, *chromB, index);
        probA = Random(0,1);
        probB = Random(0,1);

        /* exchange (LengthOfChrom - anumber) number of digits */
        for j = anumber to LengthOfChrom - 1

                /* compare if the corresponding digits are the same */
                if (offspring[index][j] != offspring[index + 1][j])

                        /* exchange the digit if probA is under a pre-specified */
                        /* CROSSOVer_RATE */
                        if (probA < CROSSOVER_RATE)
                                offspring[index][j] = chromB[j];
                        end if

                        /* exchange the digit if probB is under a pre-specified */
                        /* CROSSOVer_RATE */
                        if (probB < CROSSOVER_RATE)
                                offspring[index + 1][j] = chromA[j];
                        end if

                end if

        end for

        index = index + 2;
        if (index >= (N number of chromosome) - 1)
                index = 0;
        end if

end Crossover_Operation
```

Mutation operator implementation algorithm

```
**offspring Mutation(**offspring)

        /* Mutation performs on N number of offspring */
        for i=0 to (N number of chromosome) - 1

                /* Mutation operator check on every digit */
                for j=0 to LengthOfChrom - 1

                        /* generate a random number between 0 and 1 */
                        anumber = random(0,1);

                        /* if the 'anumber' is within the pre-specifed */
                        /* MUTATION_RATE, change the digit polarity */
                        if ( anumber < MUTATION_RATE )
                                offspring[i][j] = !(offspring[i][j]);
                        end if
                end for
        end for

end Mutation
```

The implementation of CSD Check operator

```
char CSD_Check(string *coefficient, int Nzero)

        /* No two adjacent non-zero digit next to each other*/
        string answer = NULL;
        for i=0 to (coefficient_word length) - 2
                for j=1 to (coefficient_word length) - 1
                        if ( coefficient(i) * coefficient(j) != 0)
                                answer = fail;
                                break;
                        end if
                end for
                if ( answer == fail ) break;
        end for

        /* The number of non-zero digit must be less than or equal to Nzero*/
        int sum = 0;
        for i=0 to (coefficient_word length) - 1
                sum = sum + abs( coefficient(i) );
        end for

        if ( answer != fail )
                if (sum <= Nzero)
                        answer = pass;
                else
                        answer = fail;
                end if
        end if

        return (answer)

end CSD_check
```

The implementation of Maximum Failure operator

```
*string MAX_failure(string *coefficient, int N_fail, int Nmax)

        /* No coefficient should be failed for more than Nmax times*/
        string answer = NULL;
        if ( N_fail < Nmax )
                N_fail = N_fail + 1;
                answer = coefficient;           /* coefficient is a pointer */
        else
                answer = fail;
        end if

        return (answer)

end MAX_check
```

The implementation of Reset Coefficient operator

```
*string Reset_coefficient(string *coefficient, int N_fail, int Nmax)

        /* Randomly selecting ternary bit */
        N_fail = 0;
        string answer = NULL;
        for j=0 to Nmax
                for i=0 to (word length - 1)
                        index = random( 0, word length - 1 )
                        coefficient[index] = random( -1, 1 );
                end for
        end for

        return (coefficient, N_fail)

end Reset_coefficient
```

where

random(x,y) - is a random function generator that will return a number from x to y inclusive.

Ranking Parent Selection algorithm implementation

```
*selected_chrom = Ranking_Parent_Selection(*fitness)

        int checked[word length];
        /* Initialize checked[] array */
        for i=0 to (word length - 1)
                checked[i] = 0;
        end for

        /* Perform ranking according to fitness value */
        for i = 0 to (N number of chromosome)-1
                int maximum = 0;
                for j = 0 to (N number of chromosome)-1
                        if ( fitness[j] > maximum ) and ( checked[j] != 0 )
                                maximum = j;
                                checked(j) = 1;
                        end if
                end for
                Store_Chrom(j, *selected_chrom);
        end for

end Roulette_Wheel_Parent_Selection
```

The implementation algorithm of non-uniform rate

```
for digit = 1 to word length
    if {floor(word length × percentage) < digit }
        non-uniform rate = max_rate × exp(-1 × alpha ×
                (floor(word length × percentage) - digit ))
    else
        non-uniform rate = max_rate
    end if
end for
```

where

max_rate - is the maximum mutation rate; alpha is the control parameter for the exponential function;

wordlength - is the length of each encoded coefficient; 'percentage' are the percentage of completion of GA iterations;

'digit' - is defined as the current manipulating coefficient digit;

floor(*) - is a function that will return a nearest integer number that is smaller than '*'.

APPENDIX II

The appendix II contains the simulation results proposed in this thesis. Each design example presented below uses different combination of encoding schemes, objective functions and reproduction algorithms. With the addition of CSD check, Maximum failure, and Reset coefficients functions, the resultant of Canonic Signed-Digit coefficients are guaranteed.

Example 1.

A 24th order lowpass FIR filter is designed with the following specification.

$$|H_I(e^{j\omega T})| = \left\{ \begin{array}{lll} 1 & \text{for} & 0 \le \omega \le 1 \; rad/sec \\ 0 & \text{for} & 2 \le \omega \le \frac{\omega_s}{2} \end{array} \right.$$

where $\omega_s = 2\pi$ rad/sec and T = 1 sec. The coefficient wordlength is restricted to be 12 bits with a maximum of 3 non-zero digits. The following combination of Genetic Algorithm is used:

Parent selection scheme: Ranking,
Object function: Minimax
Encoding scheme: Signed-Digit Encoding
Population size: 230
The maximum number of CSD failure: 3
Static mutation rate: 0.03
Static crossover rate: 0.8

Fig.44 shows the magnitude response of the resultant filter. Its corresponding Canonic Signed-Digit coefficients are shown in Table 9. In this example the stopband suppression is over 80dB.

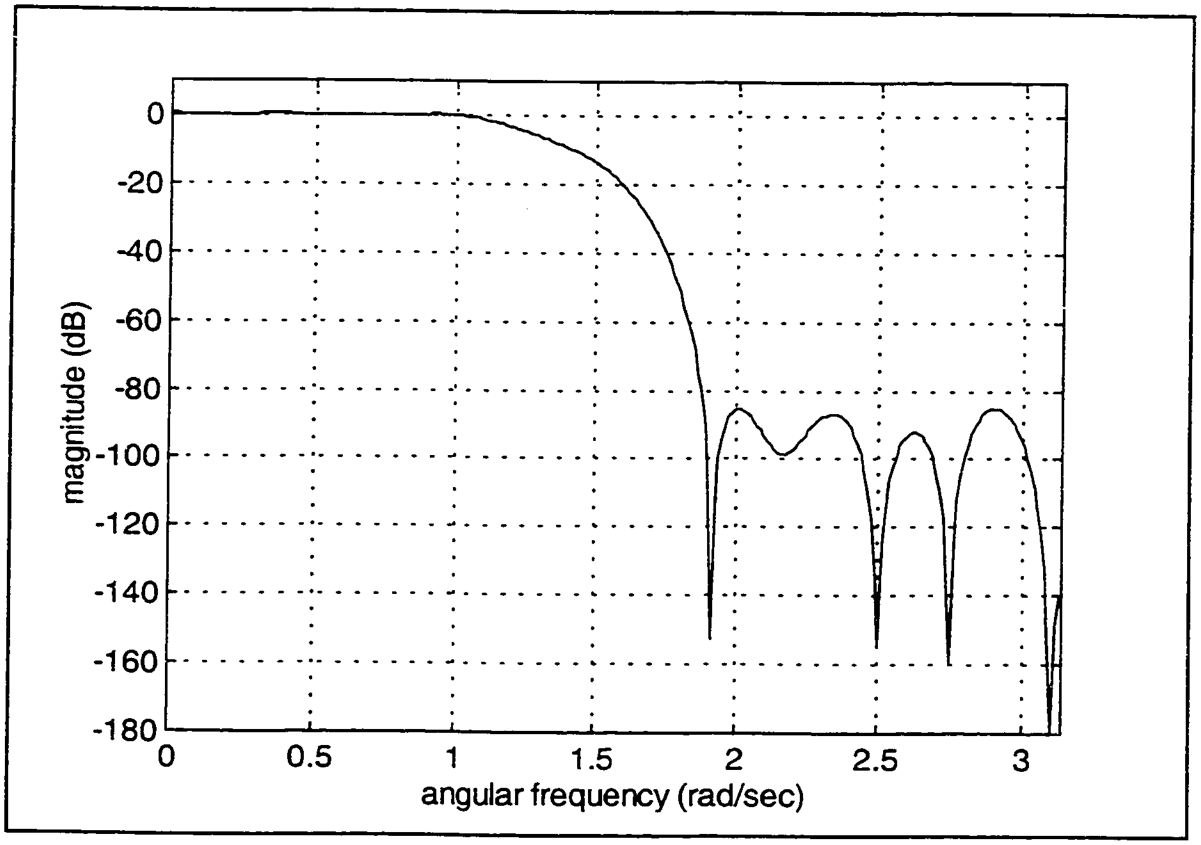

Fig. 44: A 24th order FIR filter designed by Signed-Digit Encoding, Ranking Parent Selection, and Minimax object function.

Coefficients	Values
a(0)	$2^{-4}+2^{-9}$
a(1)	$2^{-9}-2^{-11}$
a(2)	$2^{-7}-2^{-9}$
a(3)	$2^{-9}-2^{-11}$
a(4)	$2^{-9}+2^{-11}$
a(5)	$2^{-7}-2^{-9}-2^{-11}$
a(6)	$2^{-8}-2^{-10}$
a(7)	$2^{-5}+2^{-9}+2^{-11}$
a(8)	$2^{-6}-2^{-10}$
a(9)	$2^{-4}+2^{-6}+2^{-10}$
a(10)	$2^{-5}+2^{-8}+2^{-10}$
a(11)	$2^{-2}-2^{-4}+2^{-10}$
a(12)	$2^{-1}+2^{-5}-2^{-9}$
a(13)	$2^{-2}-2^{-4}+2^{-10}$
a(14)	$2^{-5}+2^{-8}+2^{-10}$
a(15)	$2^{-4}+2^{-6}+2^{-10}$
a(16)	$2^{-6}-2^{-10}$
a(17)	$2^{-5}+2^{-9}+2^{-11}$
a(18)	$2^{-8}-2^{-10}$
a(19)	$2^{-7}-2^{-9}-2^{-11}$
a(20)	$2^{-9}+2^{-11}$
a(21)	$2^{-9}-2^{-11}$
a(22)	$2^{-7}-2^{-9}$
a(23)	$2^{-9}-2^{-11}$
a(24)	$2^{-4}+2^{-9}$

Table 9: A 24th order 1-D FIR filter coefficients.

Example 2.

A 20th order lowpass FIR filter is designed with the following specification.

$$|H_I(e^{j\omega T})| = \begin{cases} 1 & \text{for} \quad 0 \le \omega \le 1 \; rad/sec \\ 0 & \text{for} \quad 2 \le \omega \le \frac{\omega_s}{2} \end{cases}$$

where $\omega_s = 2\pi$ rad/sec and T = 1 sec. The coefficient wordlength is restricted to be 12 bits with a maximum of 3 non-zero digits. The following combination of Genetic Algorithm is used:

Parent selection scheme: Ranking,
Object function: LMS
Encoding scheme:Signed-Digit Encoding
Population size: 200
The maximum number of CSD failure: 3
Static mutation rate: 0.03
Static crossover rate: 0.8

Fig.45 shows the magnitude response of the resultant filter. Its corresponding Canonic Signed-Digit coefficients are shown in Table 10. In this example the stopband suppression is over 90dB. However, the passband edge has been rounded off at 1.0 rad/sec..

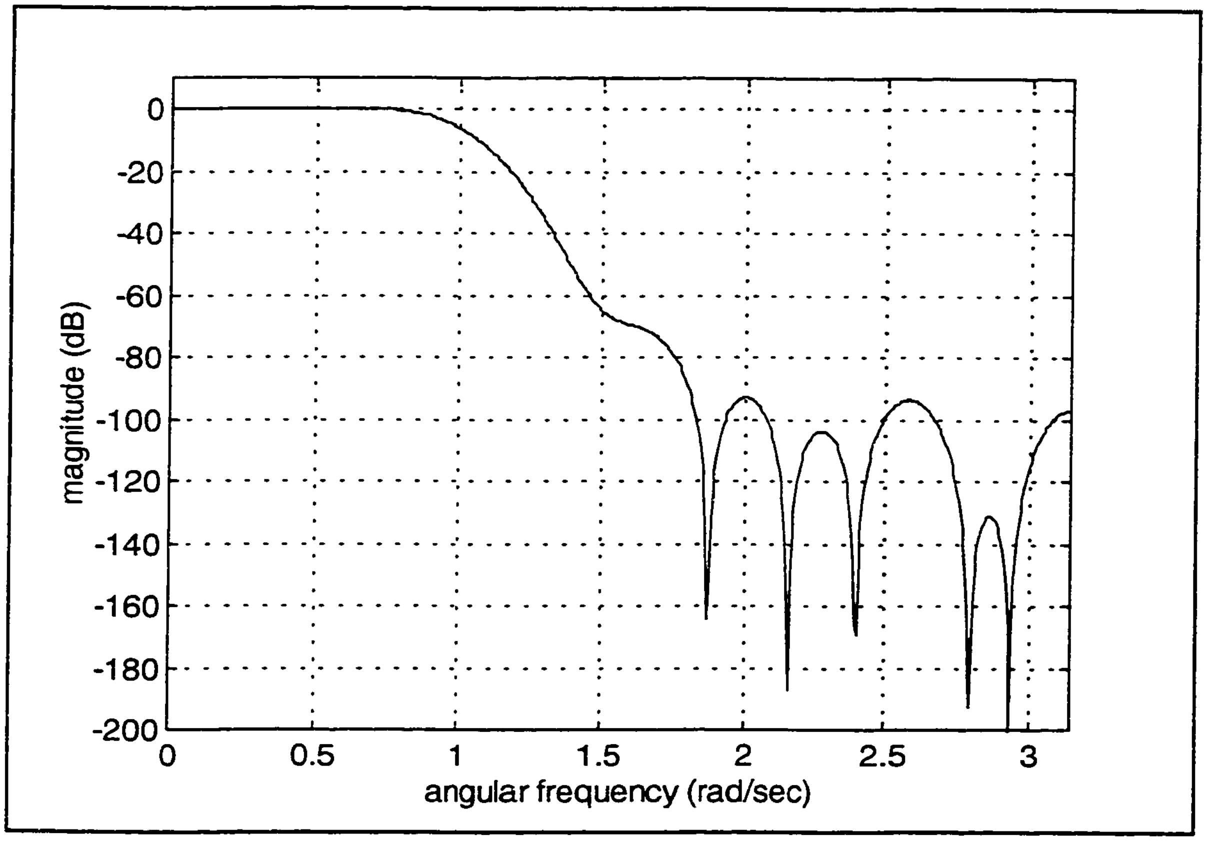

Fig. 45: A 20th order FIR filter designed by Signed-Digit Encoding, Ranking Parent Selection, and LMS object function.

Coefficients	Values
a(0)	$-2^{-7}+2^{-11}$
a(1)	$-2^{-8}-2^{-11}$
a(2)	$2^{-7}+2^{-9}$
a(3)	$2^{-6}+2^{-8}+2^{-10}$
a(4)	$2^{-7}+2^{-9}$
a(5)	$-2^{-5}+2^{-8}$
a(6)	$-2^{-4}+2^{-8}+2^{-10}$
a(7)	$-2^{-6}-2^{-8}-2^{-10}$
a(8)	$2^{-3}-2^{-6}+2^{-8}$
a(9)	$2^{-2}+2^{-5}+2^{-9}$
a(10)	$2^{-1}-2^{-3}-2^{-6}$
a(11)	$2^{-2}+2^{-5}+2^{-9}$
a(12)	$2^{-3}-2^{-6}+2^{-8}$
a(13)	$-2^{-6}-2^{-8}-2^{-10}$
a(14)	$-2^{-4}+2^{-8}+2^{-10}$
a(15)	$-2^{-5}+2^{-8}$
a(16)	$2^{-7}+2^{-9}$
a(17)	$2^{-6}+2^{-8}+2^{-10}$
a(18)	$2^{-7}+2^{-9}$
a(19)	$-2^{-8}-2^{-11}$
a(20)	$-2^{-7}+2^{-11}$

Table 10: A 20th order 1-D FIR filter coefficients.

Example 3.

A 12th order lowpass FIR filter is designed with the following specification.

$$|H_I(e^{j\omega T})| = \begin{cases} 1 & \text{for} \quad 0 \le \omega \le 1 \; rad/sec \\ 0 & \text{for} \quad 2 \le \omega \le \frac{\omega_s}{2} \end{cases}$$

where $\omega_s = 2\pi$ rad/sec and T = 1 sec. The coefficient wordlength is restricted to be 12 bits with a maximum of 3 non-zero digits. The following combination of Genetic Algorithm is used:

Parent selection scheme: Roulette Wheel
Object function: LMS
Encoding scheme:Signed-Digit Encoding
Population size: 120
The maximum number of CSD failure: 3
Static mutation rate: 0.03
Static crossover rate: 0.8

Fig.46 shows the magnitude response of the resultant filter. Its corresponding Canonic Signed-Digit coefficients are shown in Table 11. In this example the stopband suppression is over 60dB. However, the passband edge has been rounded off at 1.0 rad/sec. similar to Fig.45.

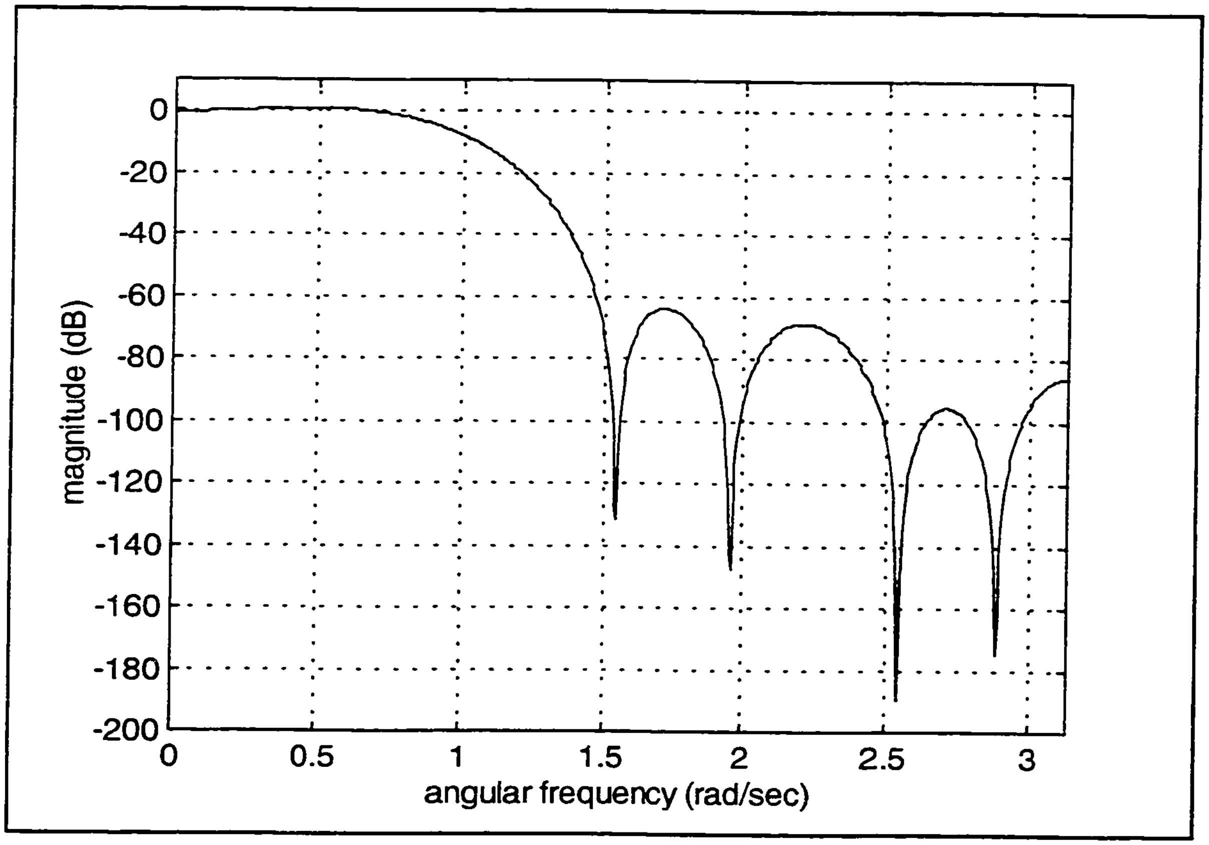

Fig. 46: A 12th order FIR filter designed by Signed-Digit Encoding, Roulette Wheel Parent Selection, and LMS object function.

Coefficients	Values
a(0)	2^{-7}
a(1)	$2^{-5}+2^{-7}$
a(2)	$2^{-4}+2^{-8}+2^{-11}$
a(3)	$2^{-6}+2^{-11}$
a(4)	$2^{-3}-2^{-8}-2^{-10}$
a(5)	$2^{-2}+2^{-5}$
a(6)	$2^{-1}-2^{-3}-2^{-6}$
a(7)	$2^{-2}+2^{-5}$
a(8)	$2^{-3}-2^{-8}-2^{-10}$
a(9)	$2^{-6}+2^{-11}$
a(10)	$2^{-4}+2^{-8}+2^{-11}$
a(11)	$2^{-5}+2^{-7}$
a(12)	2^{-7}

Table 11: A 12th order 1-D FIR filter coefficients.

Example 4.

A 16th order lowpass FIR filter is designed with the following specification.

$$|H_I(e^{j\omega T})| = \begin{cases} 1 & \text{for} \quad 0 \le \omega \le 1 \; rad/sec \\ 0 & \text{for} \quad 2 \le \omega \le \frac{\omega_s}{2} \end{cases}$$

where $\omega_s = 2\pi$ rad/sec and T = 1 sec. The coefficient wordlength is restricted to be 12 bits with a maximum of 3 non-zero digits. The following combination of Genetic Algorithm is used:

Parent selection scheme: Roulette Wheel
Object function: Minimax
Encoding scheme: Mixed Encoding
Population size: 160
The maximum number of CSD failure: 3
Static mutation rate: 0.03
Static crossover rate: 0.8

Fig.47 shows the magnitude response of the resultant filter. Its corresponding Canonic Signed-Digit coefficients are shown in Table 12. In this example the stopband suppression is over 80dB with only 16th order.

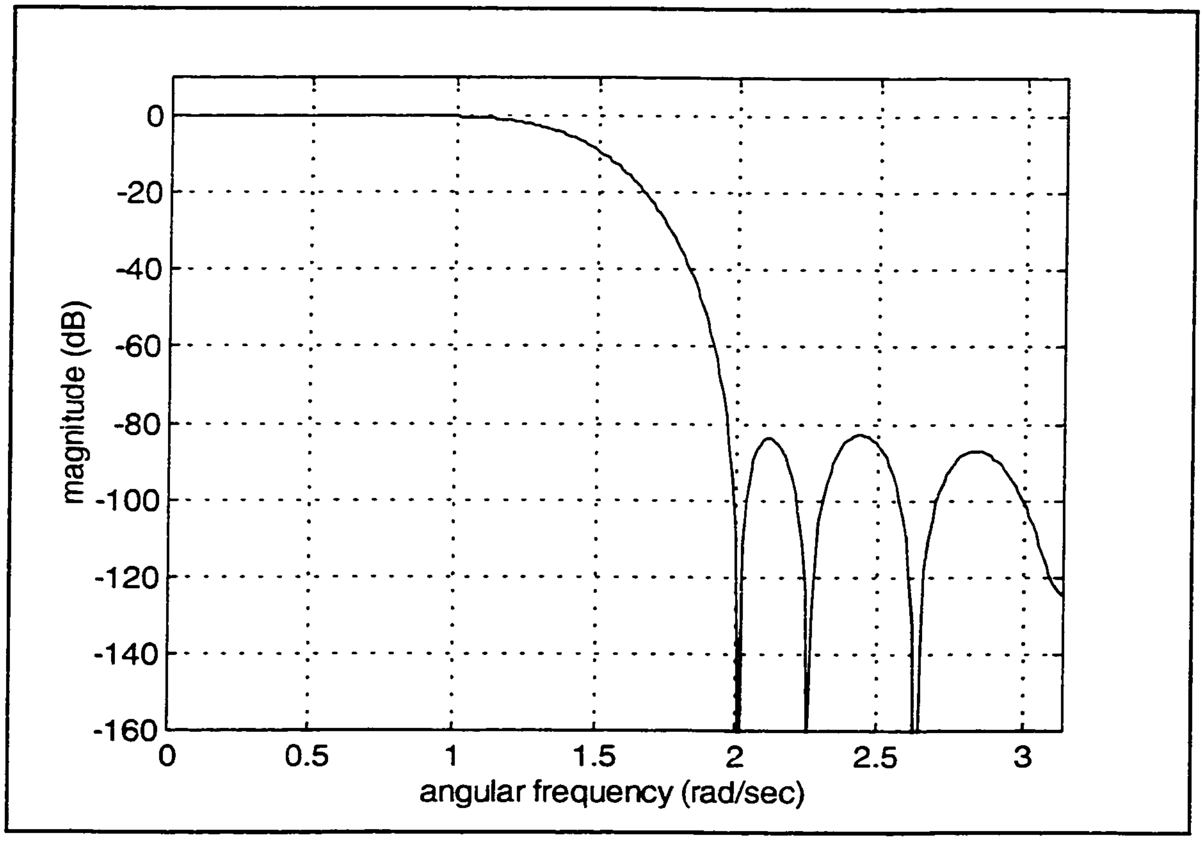

Fig. 47: A 16th order FIR filter designed by Mixed Encoding, Roulette Wheel Parent Selection, and Minimax object function.

Coefficients	Values
a(0)	-2^{-8}
a(1)	$-2^{-8}+2^{-6}$
a(2)	2^{-8}
a(3)	$-2^{-8}-2^{-5}$
a(4)	-2^{-9}
a(5)	$2^{-8}+2^{-6}+2^{-4}$
a(6)	-2^{-10}
a(7)	$-2^{-4}-2^{-2}$
a(8)	-2^{-1}
a(9)	$-2^{-4}-2^{-2}$
a(10)	-2^{-10}
a(11)	$2^{-8}+2^{-6}+2^{-4}$
a(12)	-2^{-9}
a(13)	$-2^{-8}-2^{-5}$
a(14)	2^{-8}
a(15)	$-2^{-8}+2^{-6}$
a(16)	-2^{-8}

Table 12: A 16th order 1-D FIR filter coefficients.

Example 5.

A 16th order lowpass FIR filter is designed with the following specification.

$$|H_I(e^{j\omega T})| = \begin{cases} 1 & \text{for} \quad 0 \le \omega \le 1 \; rad/sec \\ 0 & \text{for} \quad 2 \le \omega \le \frac{\omega_s}{2} \end{cases}$$

where $\omega_s = 2\pi$ rad/sec and T = 1 sec. The coefficient wordlength is restricted to be 12 bits with a maximum of 3 non-zero digits. The following combination of Genetic Algorithm is used:

Parent selection scheme: Ranking
Object function: Minimax
Encoding scheme: Mixed Encoding
Population size: 160
The maximum number of CSD failure: 3
Static mutation rate: 0.03
Static crossover rate: 0.8

Fig.48 shows the magnitude response of the resultant filter. Its corresponding Canonic Signed-Digit coefficients are shown in Table 13. In this example the stopband suppression is over 80dB. This combination of Genetic Algorithm produce similar filter response to one shown in Fig.47.

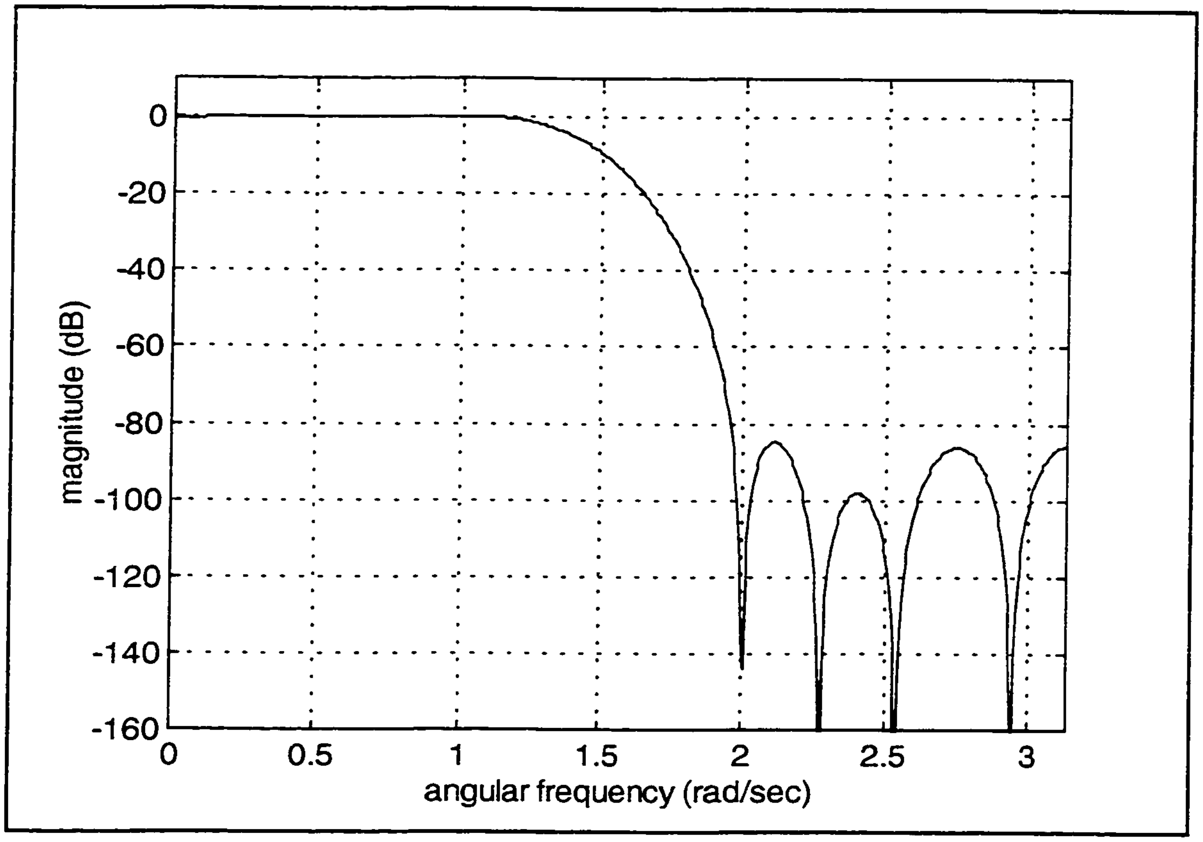

Fig. 48: A 16th order FIR filter designed by Mixed Encoding, Ranking Parent Selection, and Minimax object function.

Coefficients	Values
a(0)	2^{-9}
a(1)	$2^{-10}-2^{-6}$
a(2)	-2^{-11}
a(3)	$2^{-8}+2^{-5}$
a(4)	-2^{-11}
a(5)	$2^{-8}+2^{-5}-2^{-3}$
a(6)	-2^{-10}
a(7)	$2^{-4}+2^{-2}$
a(8)	2^{-1}
a(9)	$2^{-4}+2^{-2}$
a(10)	-2^{-10}
a(11)	$2^{-8}+2^{-5}-2^{-3}$
a(12)	-2^{-11}
a(13)	$2^{-8}+2^{-5}$
a(14)	-2^{-11}
a(15)	$2^{-10}-2^{-6}$
a(16)	2^{-9}

Table 13: A 16th order 1-D FIR filter coefficients.

Example 6.

In this example, we have included the Non-uniform Mutation and Crossover rates, and Fitness Scaling in the GA process. The addition of these features enhance the Reproduction operator to effectively identify more suitable chromosomes. A 60th order FIR filter is designed to illustrate the usefulness of these features. Every fitness value is scaled between one and twenty discrete space. The specification is shown below.

$$|H_I(e^{j\omega T})| = \{ \begin{array}{lll} 1 & \text{for} & 0 \le \omega \le 1 \; rad/sec \\ 0 & \text{for} & 2 \le \omega \le \frac{\omega_s}{2} \end{array}$$

where $\omega_s = 2\pi$ rad/sec and T = 1 sec. The coefficient wordlength is restricted to be 24 bits with a maximum of 3 non-zero digits. The following combination of Genetic Algorithm is used:

Parent selection scheme: Roulette Wheel	
Object function: Minimax	
Encoding scheme:Signed-Digit Encoding	
Population size: 2000	
The maximum number of CSD failure: 3	
Maximum mutation rate: 0.1	(alpha = 0.5)
Crossover rate: starting at 1.0, ending at 0.7	(alpha = 0.5)

Fig.49 shows the magnitude response of the resultant filter. Its corresponding Canonic Signed-Digit coefficients are shown in Table 14. In this higher order filter example, the stopband suppression is just over 90dB.

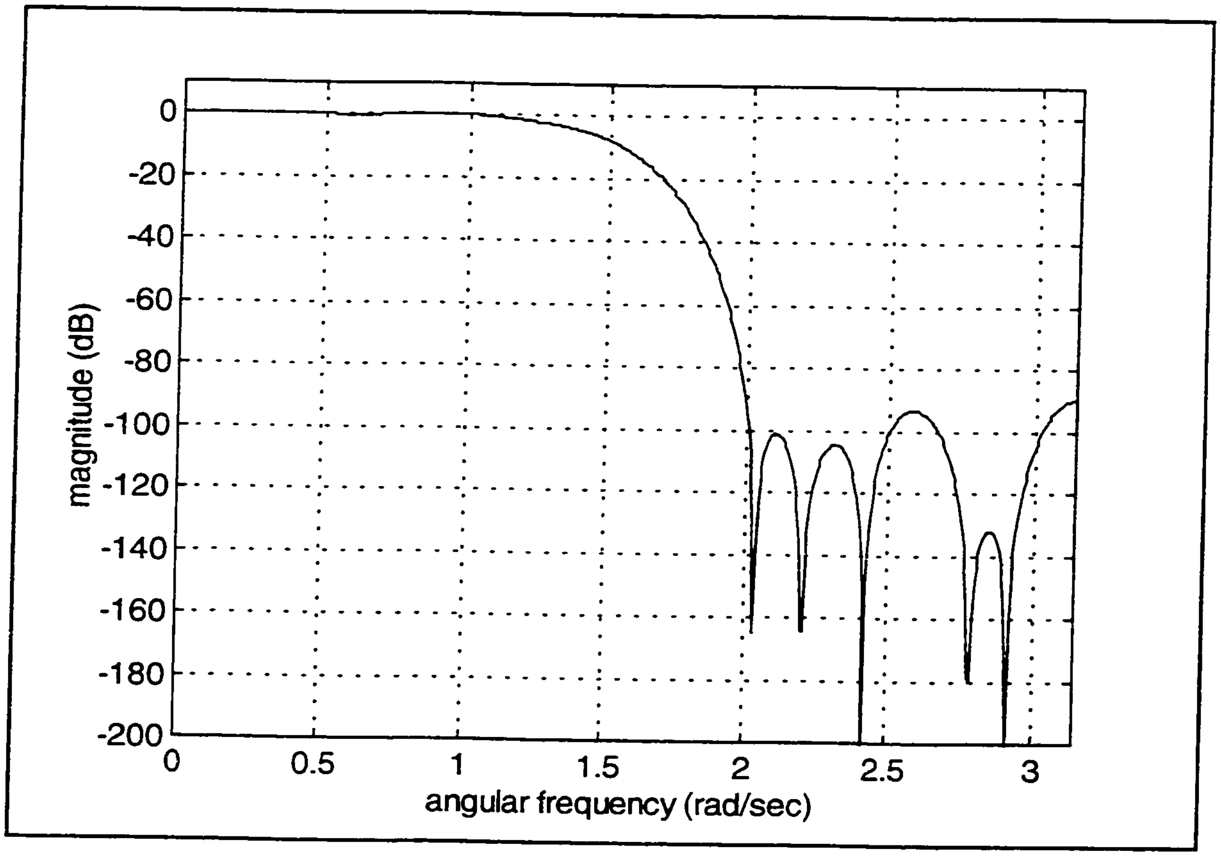

Fig.49 : Magnitude response of a 60th order lowpass FIR filter with Signed-Digit Encoding, Roulette Wheel Parent Selection, and Minimax strategy.

Coefficients	values
a(0)	$-2^{-10} + 2^{-16} + 2^{-21}$
a(1)	$2^{-10} + 2^{-17} + 2^{-23}$
a(2)	$-2^{-13} + 2^{-16} - 2^{-23}$
a(3)	$-2^{-9} + 2^{-14} - 2^{-23}$
a(4)	$2^{-11} - 2^{-17} - 2^{-21}$
a(5)	$2^{-9} + 2^{-18} - 2^{-23}$
a(6)	$-2^{-11} + 2^{-13}$
a(7)	$-2^{-13} - 2^{-15}$
a(8)	$2^{-10} - 2^{-16} + 2^{-23}$
a(9)	$-2^{-10} + 2^{-15} - 2^{-19}$
a(10)	$-2^{-13} - 2^{-18} + 2^{-22}$
a(11)	$2^{-9} + 2^{-16} - 2^{-18}$
a(12)	$-2^{-20} - 2^{-22}$
a(13)	$-2^{-9} + 2^{-15} + 2^{-20}$
a(14)	$2^{-11} - 2^{-14} - 2^{-22}$
a(15)	$-2^{-15} - 2^{-17} - 2^{-23}$
a(16)	$-2^{-11} - 2^{-14} + 2^{-21}$
a(17)	$2^{-8} - 2^{-15} + 2^{-21}$
a(18)	$2^{-13} + 2^{-17} - 2^{-23}$
a(19)	$-2^{-7} - 2^{-14} - 2^{-21}$
a(20)	$-2^{-14} + 2^{-18} - 2^{-23}$
a(21)	$2^{-7} - 2^{-19} - 2^{-22}$
a(22)	$2^{-13} - 2^{-15} + 2^{-18}$
a(23)	$2^{-13} + 2^{-17} + 2^{-23}$
a(24)	$-2^{-16} + 2^{-18} + 2^{-23}$
a(25)	$-2^{-5} + 2^{-7} - 2^{-11}$
a(26)	$-2^{-14} - 2^{-16} + 2^{-22}$
a(27)	$2^{-4} + 2^{-6} - 2^{-16}$
a(28)	$2^{-16} + 2^{-18} - 2^{-23}$
a(29)	$-2^{-2} - 2^{-4} + 2^{-8}$
a(30)	$-2^{-1} - 2^{-14} + 2^{-18}$

*Because of the even symmetry property of the coefficients, only half of the coefficients are shown.

Table 14: A 60th order 1-D FIR filter coefficients.

APPENDIX III

The appendix III contains the source code of Genetic Algorithm in the application of filter design with arbitory specification. Design algorithm is shown in Appendix I. Each of the following sections consists at least three source files (other than the DBNS and 2-D IIR coefficient designs): steps.cpp, allchrom.cpp, and random.cpp. And each section corresponds to a different combination of Genetic Algorithm.

> 'Steps.cpp' contains the orgination of Genetic Algorithm and the cost-function of filter design.
>
> 'Allchrom.cpp' contains the encoding and decoding scheme, and the operations of crossover, mutation, CSD check, max. failure, and reset coefficient.
>
> 'Random.cpp' is a random function generator that is shown in section 1 only.

A filter is designed by inputing the filter specification at the beginning of 'allchrom.cpp' file, complie it with the command:

```
CC -lm steps.cpp <output filename>.
```

The arguments required to run this program is shown as below:

```
steps.exe fun iter popsize Pmut Pcro Ncoef <output filename>.
```

where

fun - must be either 'm' or 'l' that corresponds to minimax or least mean squre strategy.

iter - is the maximum number of iterations.

popsize - specify the population size.

Pmut - is the propability of mutation rate ranging from 0 to 100.

Pcro - is the propabiligy of crossover rate ranging from 0 to 100.

Ncoef - is the effective number of coefficient that is evaluated by

Ncoef = (desire filter order) / 2 + 1

<output filename> - specified file to store the resultant filter coefficients.

Section 1

This section involves the combination of signed-digit encoding with roulette wheel parent selection.

RANDOM.CPP

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>

#ifndef _GETSEED_H
#define _GETSEED_H \
struct timeval tp; \
gettimeofday(&tp, NULL); \
srand((tp.tv_sec % 8388608 ) * 256 + tp.tv_usec % 256);
#endif


unsigned int ANUMBER(void){
        static int init=0;
        if (!init) { _GETSEED_H; init++; }
        return( rand() );
}

float UNI(void){
        return( ANUMBER() / 32767.0 );            //2^15-1
}
```

ALLCHROM.CPP

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

#define NCSDdigit 3
#define Nbit        12      //8
#define T           1.0     //sampling frequency
#define maxFreq 3.1416
#define Npoints 80 //# of sampling points
#define stopbd      2.0     //either stopbd or ALPHA is used
#define ALPHA   6.0
#define passbd      1.0

typedef struct {
        signed int bit:2;
} CSDbit;

typedef struct {
        CSDbit coef[50][Nbit];
        double fitness;
} CHMtype;
```

```
//This class will deal with all the single bit things
class AllChrom {
private:
            CHMtype *chrom, *offsprg, bestChrom;
      double pmut, pcross;
            int    Nchrom, Ncoef;
            FILE *outptr;
            char outfilename[22];

public:
      AllChrom(int,double,double,char*, int);
            ~AllChrom();
            double decodeChrom(int,int);                    //which chrom, which gene
      double decodeBest(int);
            int randBit();
            int randBit(int);
            int CSDcheck(int,int);                          //which offsprg, which gene
            void report();
            void DOcrossover(int,int ,int);                 //which 2 chrom, which gene, which gene, which 2 offsprg
            void DOmutation(int,int);
            void ArandChrom(int);                           //which chrom
            void GETbestChrom();        //usually is 1 for index
            void chrom2offsprg(int,int);
      void offsprg2chrom(int,int);
            void reloadGene(int,int,int,int);
            void preloadChrom();
            void ArandGene(int,int);
            void AsmallGene(int,int);
            void OUTchrom(int);
            void scaleFitness();

            void INchromfitness(int loc, double val)        { chrom[loc].fitness = val; }
            double OUTchromfitness(int loc)         { return(chrom[loc].fitness); }
            void INoffsprgfitness(int loc, double val)      { offsprg[loc].fitness = val; }
            double OUToffsprgfitness(int loc)               { return(offsprg[loc].fitness); }
            double OUTbestfitness()                         { return(bestChrom.fitness); }
};

void AllChrom::scaleFitness(){
            int i, value;
            double minN, maxN;

            //Find maximum and minimum
            minN = 999.9;
            maxN = -10.0;
            for(i = 0; i < Nchrom; i++){
                        if( minN > OUTchromfitness(i) ) minN = OUTchromfitness(i);
                        if( maxN < OUTchromfitness(i) ) maxN = OUTchromfitness(i);
            }

            //Perform scaling
            for( i = 0; i < Nchrom; i++){
//                      value = (int) ceil( 100.0 * exp(-2.0 * (OUTchromfitness(i) - minN)/(maxN - minN)) );
                        value = (int) ceil( 100.0 * (OUTchromfitness(i) - minN)/(maxN - minN) );
                        INchromfitness(i, value);
            }
}


int AllChrom::randBit(){
```

```
            int numb;

            numb = ANUMBER() % 3;
            numb = numb - 1;
            return( numb );
}

int AllChrom::randBit(int except){
            int numb = except;

            while (numb == except){
                        numb = ANUMBER() % 3;
                        numb = numb - 1;
            }
            return( numb );
}

AllChrom::AllChrom(int Nchromosome, double pmutation, double pcrossover, char *fname, int Ncoeff){
            int i;

            Ncoef = Ncoeff;
            Nchrom = Nchromosome;
            pmut = pmutation;
      pcross = pcrossover;

            chrom = new CHMtype[Nchrom];
            if ( chrom == NULL){
                        printf("\nNot enough memory for *chrom");
                        exit(-1);
            }

            offsprg = new CHMtype[Nchrom];
            if ( offsprg == NULL){
                        printf("\nNot enough memory for *offsprg");
                        exit(-1);
            }

            bestChrom.fitness = 0.0;

            for(i = 0; i < strlen(fname); i++) outfilename[i] = fname[i];
}

AllChrom::~AllChrom(){
            delete[] chrom;
      delete[] offsprg;
}

void AllChrom::OUTchrom(int loc){
            int i,j;

            for ( j=0 ; j < Ncoef; j++){
                        for (i =0; i < Nbit; i++)
                                    printf("%3d",chrom[loc].coef[j][i].bit);
            }
}


//decodeChrom
double AllChrom::decodeChrom(int chrom_loc, int coef_loc){
            double anumber;
            int n;

            anumber = 0;
            for (n = 0; n < Nbit; n++){
                        if ( chrom[chrom_loc].coef[coef_loc][n].bit == 1 )
                                    anumber = anumber + pow(2, -1.0*(n));
                        else if ( chrom[chrom_loc].coef[coef_loc][n].bit == -1 )
                                    anumber = anumber - pow(2, -1.0*(n));
                        else { }
            }
            return(anumber);
}


//decodeBestchrom
double AllChrom::decodeBest(int coef_loc){
            double anumber;
            int n;

            anumber = 0;
            for (n = 0; n < Nbit; n++){
                        if ( bestChrom.coef[coef_loc][n].bit == 1 )
                                    anumber = anumber + pow(2, -1.0*(n));
                        else if ( bestChrom.coef[coef_loc][n].bit == -1 )
                                    anumber = anumber - pow(2, -1.0*(n));
                        else { }
            }
            return(anumber);
}


void AllChrom::report(){
            int i,j,flag;

            outptr = fopen( outfilename ,"w+");
            if ( outptr == NULL){
                        printf("cannot open output file");
            exit(-1);
      }

            for(i = 0; i < Ncoef; i++){
                        for(j = 0; j < Nbit; j++) fprintf(outptr, "%3d",bestChrom.coef[i][j].bit);
                        fprintf(outptr, "\n");
            }
            for(i = 1; i < Ncoef; i++){
                        for(j = 0; j < Nbit; j++) fprintf(outptr, "%3d",bestChrom.coef[Ncoef - i - 1][j].bit);
                        fprintf(outptr,"\n");
            }

      fprintf(outptr,"\n");
            for(i = 0; i < Ncoef; i++){
                        fprintf(outptr,"\n%18.15f",decodeBest(i));

            }
      for ( j = 1; j < Ncoef; j++){
                        fprintf(outptr,"\n%18.15f",decodeBest(Ncoef - j - 1));
      }


            fprintf(outptr, "\n\nThe best fitness is %f", bestChrom.fitness);
            fclose(outptr);
 }

 void AllChrom::DOcrossover(int loc1, int loc2, int coef_loc){
            int apoint, i, j;
            int abit;

            if ( UNI() < pcross ){
                        apoint = ANUMBER() % (Nbit - 2);
                        apoint++;

                        for ( i = 0; i < apoint ; i++){
                                    abit = chrom[loc1].coef[coef_loc][i].bit;
                                    chrom[loc1].coef[coef_loc][i].bit = chrom[loc2].coef[coef_loc][i].bit;
                                    chrom[loc2].coef[coef_loc][i].bit = abit;
                        }
            }
 }

 void AllChrom:: DOmutation(int loc, int coefloc){
            int i,temp;

            for (i=0; i < Nbit; i++){
//          if ( UNI() < pmut ){
            if ( pmut > (UNI() * exp(-3.0 * (1.0 - i / (Nbit - 1)))) ){
                                    //check the original one
                                    temp = chrom[loc].coef[coefloc][i].bit;
                                    chrom[loc].coef[coefloc][i].bit = randBit(temp);
            }
      }

 }

 int AllChrom::CSDcheck(int loc, int geneloc){
            int sumbit = 0;
            int i , a,b,c,rvalue = 1;

            for( i = 0; i < Nbit; i++) sumbit = sumbit + abs(chrom[loc].coef[geneloc][i].bit);

            if ( sumbit <= NCSDdigit ){
                        for ( i = 1 ; i < (Nbit - 1); i++){
                                    a = abs(chrom[loc].coef[geneloc][i].bit);
                                    b = abs(chrom[loc].coef[geneloc][i - 1].bit);
                                    c = abs(chrom[loc].coef[geneloc][i + 1].bit);

                                    if ( a & b ) goto JUMP;
                                    if ( a & c ) goto JUMP;
                        }
                        rvalue = 0;
            }

JUMP:     return(rvalue);
}

void AllChrom::reloadGene(int a, int loc1, int loc2, int coefloc){
            int i;

            for( i =0; i < Nbit; i++){
                        chrom[a].coef[coefloc][i].bit = offsprg[loc1].coef[coefloc][i].bit;
                        chrom[a + 1].coef[coefloc][i].bit = offsprg[loc2].coef[coefloc][i].bit;
            }
```

```
}

void AllChrom::preloadChrom(){
            int i;
            for( i =0; i < Nchrom; i++) chrom2offsprg(i , i);
}


void AllChrom::ArandChrom(int loc){                         //which chrom
            int i,j,k;
            int luckybit;

            for (j=0; j < Ncoef; j++){
                        for (i=0; i < Nbit; i++) chrom[loc].coef[j][i].bit = 0;

                        luckybit = ANUMBER() % Nbit;
                        chrom[loc].coef[j][luckybit].bit = randBit(0);
            }
}




//For offsprg
void AllChrom::ArandGene(int loc, int coef_loc){
            int i;
            int luckybit;

            for( i = 0; i < Nbit; i++)
                        chrom[loc].coef[coef_loc][i].bit = 0;
            luckybit = ANUMBER() % Nbit;
            chrom[loc].coef[coef_loc][luckybit].bit = randBit(0);
}

void AllChrom::AsmallGene(int loc, int coef_loc){
            int i;
            for( i = 0; i < (Nbit - 1); i++) chrom[loc].coef[coef_loc][i].bit = 0;
            chrom[loc].coef[coef_loc][Nbit - 1].bit = randBit(0);
}


//loc usually is 1
void AllChrom::GETbestChrom(){       //usually is 1 for index
            int j,i,loc;

            loc = -10;
            for( i =0 ; i < Nchrom; i++){
                        if ( bestChrom.fitness < chrom[i].fitness ) loc = i;
            }

            if ( loc >= 0 ){
                        bestChrom.fitness = chrom[loc].fitness;
                        for(j=0;j < Ncoef; j++){
                                    for(i=0; i < Nbit; i++) bestChrom.coef[j][i].bit = chrom[loc].coef[j][i].bit;
                }
            }
}

//don't include fitness
void AllChrom::chrom2offsprg(int chrom_loc, int offsprg_loc){
            int i,j;
            for(j=0;j < Ncoef; j++){
                    for(i=0; i < Nbit; i++)
                            offsprg[offsprg_loc].coef[j][i].bit = chrom[chrom_loc].coef[j][i].bit;
                    offsprg[offsprg_loc].fitness = chrom[chrom_loc].fitness;
        }
 }

 //include fitness
 void AllChrom::offsprg2chrom(int offsprg_loc , int chrom_loc){
            int i,j;

            for(j=0;j < Ncoef; j++){
                    for(i=0; i < Nbit; i++)
                            chrom[chrom_loc].coef[j][i].bit = offsprg[offsprg_loc].coef[j][i].bit;
                    chrom[chrom_loc].fitness = offsprg[offsprg_loc].fitness;
        }
 }
```

STEPS.CPP

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include "allchrom.cpp"

class Steps : public AllChrom {
private:
            int Noffsprg, Nchrom, Ncoef;
            int *ordering;
            char ob;

public:
        Steps(int,double,double,char, char*,int);
        ~Steps(){ }
            void presetChrom();                              //watch out for preload
        void reproduce();           //duplicate the chrom to offsprg
            void GETchromfit();   //passing decoded coefficients to fitness
            double fitness(double*);
            void CMCfix();
};


Steps::Steps(int Nchromosome, double pmutation, double pcrossover, char ob_ch, char *name, int
Ncoeff):AllChrom(Nchromosome, pmutation,pcrossover, name, Ncoeff){

            Ncoef = Ncoeff;
            Nchrom = Nchromosome;
            Noffsprg = Nchrom;

            ordering = new int [Nchrom];
            if (ordering == NULL){
                    printf("\nno memory for ordering");
                    exit(-1);
            }

            ob = ob_ch;
}
```

```
 void Steps::presetChrom(){
             int j,i;

             for(j = 0 ; j < Nchrom; j++){
             ArandChrom(j);
             }

             //************check for the initial data **************
/*           for ( j=0; j < Nchrom; j++){
                     OUTchrom(j);
                     printf("\n");
             }
             scanf("%d",&i);
*/
}


//duplicate and set the Noffsprg
void Steps::reproduce(){
      int i, count;
            double rawfitness, total, sum;

            total = -1.0E-7;
            for( i = 0; i < Nchrom; i++) total = total + OUTchromfitness(i);


            for( i = 0; i < Noffsprg; i++){
                      rawfitness = UNI() * total;
                      sum = OUTchromfitness(0);
                      count = 0;
                      while( sum < rawfitness ){
                                count++;
                                sum = sum + OUTchromfitness(count);
                      }

                      ordering[i] = count;
            }
}


void Steps::CMCfix(){
            int coef_count,j;
            int flag1, flag2 , count;

            preloadChrom();

            for ( j = 0; j < Noffsprg; j = j + 2){
                      for ( coef_count = 0; coef_count < Ncoef; coef_count++){

                                //check for CSD format
                                count = 0;
                                flag2 = 1;

                                while( (flag2) && (count < 2) ){
                                          reloadGene(j,ordering[j], ordering[j + 1],
coef_count);

                                          DOcrossover(j, j + 1, coef_count);
                                          DOmutation(j, coef_count);
                                          DOmutation(j + 1, coef_count);
```

```
                                        flag1 = CSDcheck(j,    coef_count);
                                        if ( flag1 == 0) flag2 = CSDcheck(j + 1, coef_count);
                                        count++;
                                }

                                //if all five chances fail
                                if ( (flag1 != 0) || (flag2 != 0) ){
                                        flag2 = CSDcheck(j +1, coef_count);
                                        if (flag1 != 0 ) ArandGene(j, coef_count);
                                        if (flag2 != 0 ) ArandGene(j +1,coef_count);
                                }

                                //check for zero coefficients
                                if ( decodeChrom(j, coef_count) == 0.0 )
                                        ArandGene(j, coef_count);
                                if ( decodeChrom(j + 1, coef_count) == 0.0 )
                                        ArandGene(j + 1, coef_count);
                        }
                }
        }

void Steps::GETchromfit(){   //passing decoded coefficients to fitness
        int j,i;
        double *coeff, val;

        coeff = new double[Ncoef];
        if ( coeff == NULL){
                printf("\nNot enough memory for *coeff");
                exit(-1);
        }

        for( j = 0; j < Nchrom; j++){
                for( i=0; i < Ncoef; i++){
                coeff[i] = decodeChrom(j,i);
                }

                val = fitness(coeff);
                INchromfitness(j,val);
    }

        delete[] coeff;
}

double Steps::fitness(double *coeff){
        int i,j, flag , count;
        double err,w, delta_w, magH;
        double H, anumber, temp;
        double sum, return_val, max_err;

    delta_w = maxFreq / Npoints;

    err = 0;
        max_err = 0;
        for (w = 0 ; w < maxFreq + delta_w; w = w + delta_w){

                //at one frequency of the first half symmetry coefficients
        count = 0;

                H = coeff[Ncoef - 1];
                for ( j = 1; j < Ncoef; j++){
                        H = H + 2.0 * coeff[Ncoef - j - 1] * cos(w * T * j);
```

```
                    }

                    H = fabs(H);


                    //minmax or LMS
                    if ( ob == 'l'){

                              //exp decay transision region
                              if ( w < passbd )
                                        err = err + fabs( 1.0 - H);
                              else
                                        err = err + fabs( exp(-1.0*ALPHA*(w - passbd)) - H );

                    } else if(ob == 'm') {

                              //get the magnitude response error
                              if ( w < passbd + delta_w) err = fabs( 1.0 - H );
                              else if ( w > stopbd - delta_w ) err = fabs(H);
                              else { }
                              if( max_err < err ) max_err = err;

                    } else {
                              printf("\ncannot get the objective function\n");
                              exit(-1);
                    }

          }

          if (ob == 'l') return_val  =  1.0 / err;
          else return_val  =  1.0 / max_err;

          return( return_val );

}

//***************************************************************************

class final:public Steps {
private:
          int MaxIter;

public:
      final(int,int,double,double,char,char*,int);
          ~final(){ }
          void process();
};

final::final(int MaxIteration, int chrom_size, double mut_prob, double cross_prob, char ob_ch, char *name, int
Ncoeff):Steps(chrom_size,mut_prob, cross_prob, ob_ch, name, Ncoeff){
          MaxIter = MaxIteration;
}


void final::process(){
          int Citer;

      //get the random chromosome
          presetChrom();
```

```
            //calcuate the fitness of each chromosome
            GETchromfit();

            Citer = 0;
            while ( Citer < MaxIter ) {
                    Citer++;

                //find Noffsprg and duplicate the chrom to offsprg
                            reproduce();

                            //crossover, mutation, and CSD checkNfix
                            CMCfix();

                            //calcuate fitness of each offsprg
                            GETchromfit();

                //store the best chromosome comparing to the history
                            GETbestChrom();

                            printf("\n%d          %f", Citer, OUTbestfitness() );

            }
            printf("\n");
            report();
}

main(int argsize, char **arg){
            int popsize, iterations;
            double mutation, crossover;
            int Ncoeff;

//          arg[1] - m for minimax or l for LMS
//          arg[6] - output filename

            if ( argsize == 8 ){
                    iterations = atoi(arg[2]);
                    popsize   = atoi(arg[3]);
                    mutation  = ((double) atoi(arg[4])) / 100.0;
                    crossover = ((double) atoi(arg[5])) / 100.0;
                    Ncoeff    = atoi(arg[6]);

//                  printf("\niteration is %d popsize is %d ", iterations, popsize );
//                  printf("mutation is %f crossover is %f\n", mutation, crossover );
//                  printf("The order of the filter is %d", Ncoeff - 1);

                    final object( iterations, popsize, mutation, crossover, *arg[1], arg[7], Ncoeff);
                    object.process();
            }
            else printf("\nwrong argument");
}
```

Section 2

This section involves the combination of signed-digit encoding with ranking parent selection.

ALLCHROM.CPP

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

#define NCSDdigit 3
#define Nbit        12      //8
#define T           1.0     //sampling frequency
#define maxFreq 3.1416
#define Npoints 70 //# of sampling points
#define stopbd      2.0     //either stopbd or ALPHA is used
#define ALPHA  6.0
#define passbd      1.0

typedef struct {
        signed int bit:2;
} CSDbit;

typedef struct {
        CSDbit coef[50][Nbit];
        double fitness;
} CHMtype;


//This class will deal with all the single bit things
class AllChrom {
private:
        CHMtype *chrom, *offsprg, bestChrom;
    double pmut, pcross;
        int    Nchrom, Ncoef;
        FILE *outptr;
        char outfilename[22];

public:
    AllChrom(int,double,double,char*, int);
        ~AllChrom();
        double decodeChrom(int,int);                //which chrom, which gene
    double decodeBest(int);
        int randBit();
        int randBit(int);
        int CSDcheck(int,int);                      //which offsprg, which gene
        void report();
        void DOcrossover(int,int ,int);             //which 2 chrom, which gene, which gene, which 2 offsprg
        void DOmutation(int,int);
        void ArandChrom(int);                       //which chrom
        void GETbestChrom();        //usually is 1 for index
        void chrom2offsprg(int,int);
    void offsprg2chrom(int,int);
        void reloadGene(int,int,int);
        void preloadChrom();
        void ArandGene(int,int);
        void AsmallGene(int,int);
        void OUTchrom(int);
        void scaleFitness();

        void INchromfitness(int loc, double val)        { chrom[loc].fitness = val; }
        double OUTchromfitness(int loc)         { return(chrom[loc].fitness); }
        void INoffsprgfitness(int loc, double val)      { offsprg[loc].fitness = val; }
        double OUToffsprgfitness(int loc)               { return(offsprg[loc].fitness); }
        double OUTbestfitness()                         { return(bestChrom.fitness); }
```

```
 };

 void AllChrom::scaleFitness(){
            int i, value;
            double minN, maxN;

            //Find maximum and minimum
            minN = 999.9;
            maxN = -10.0;
            for(i = 0; i < Nchrom; i++){
                        if ( fabs(OUTchromfitness(i) - ceil(OUTchromfitness(i)) ) < 1.0E-8 ){
                        if( minN > OUTchromfitness(i) ) minN = OUTchromfitness(i);
                        if( maxN < OUTchromfitness(i) ) maxN = OUTchromfitness(i);
                        }
            }

            //Perform scaling
            for( i = 0; i < Nchrom; i++){
                        if ( fabs(OUTchromfitness(i) - ceil(OUTchromfitness(i)) ) < 1.0E-8 ){
//                      value = (int) ceil( 100.0 * exp(-2.0 * (OUTchromfitness(i) - minN)/(maxN - minN)) );
                        value = (int) ceil( 100.0 * (OUTchromfitness(i) - minN)/(maxN - minN) );
                        INchromfitness(i, value);
                        }
            }
}


int AllChrom::randBit(){
            int numb;

            numb = ANUMBER() % 3;
            numb = numb - 1;
            return( numb );
}

int AllChrom::randBit(int except){
            int numb = except;

            while (numb == except){
                        numb = ANUMBER() % 3;
                        numb = numb - 1;
            }
            return( numb );
}

AllChrom::AllChrom(int Nchromosome, double pmutation, double pcrossover, char *fname, int Ncoeff){
            int i;

            Ncoef = Ncoeff;
            Nchrom = Nchromosome;
            pmut = pmutation;
      pcross = pcrossover;

            chrom = new CHMtype[Nchrom];
            if ( chrom == NULL){
                        printf("\nNot enough memory for *chrom");
                        exit(-1);
            }

            offsprg = new CHMtype[Nchrom];
            if ( offsprg == NULL){
                printf("\nNot enough memory for *offsprg");
                exit(-1);
        }

        bestChrom.fitness = 0.0;

        for(i = 0; i < strlen(fname); i++) outfilename[i] = fname[i];
}

AllChrom::~AllChrom(){
        delete[] chrom;
    delete[] offsprg;
}

void AllChrom::OUTchrom(int loc){
        int i,j;

        for ( j=0 ; j < Ncoef; j++){
                for (i =0; i < Nbit; i++)
                        printf("%3d",chrom[loc].coef[j][i].bit);
        }
}



//decodeChrom
double AllChrom::decodeChrom(int chrom_loc, int coef_loc){
        double anumber;
        int n;

        anumber = 0;
        for (n = 0; n < Nbit; n++){
                if ( chrom[chrom_loc].coef[coef_loc][n].bit == 1 )
                        anumber = anumber + pow(2, -1.0*(n));
                else if ( chrom[chrom_loc].coef[coef_loc][n].bit == -1 )
                        anumber = anumber - pow(2, -1.0*(n));
                else { }
        }
        return(anumber);
}


//decodeBestchrom
double AllChrom::decodeBest(int coef_loc){
        double anumber;
        int n;

        anumber = 0;
        for (n = 0; n < Nbit; n++){
                if ( bestChrom.coef[coef_loc][n].bit == 1 )
                        anumber = anumber + pow(2, -1.0*(n));
                else if ( bestChrom.coef[coef_loc][n].bit == -1 )
                        anumber = anumber - pow(2, -1.0*(n));
                else { }
        }
        return(anumber);
}


void AllChrom::report(){
        int i,j,flag;
```

```
            outptr = fopen( outfilename ,"w+");
            if ( outptr == NULL){
                        printf("cannot open output file");
            exit(-1);
       }

            for(i = 0; i < Ncoef; i++){
                        for(j = 0; j < Nbit; j++) fprintf(outptr, "%3d",bestChrom.coef[i][j].bit);
                        fprintf(outptr, "\n");
            }
            for(i = 1; i < Ncoef; i++){
                        for(j = 0; j < Nbit; j++) fprintf(outptr, "%3d",bestChrom.coef[Ncoef - i - 1][j].bit);
                        fprintf(outptr,"\n");
            }

       fprintf(outptr,"\n");
            for(i = 0; i < Ncoef; i++){
                        fprintf(outptr,"\n%18.15f",decodeBest(i));

            }
       for ( j = 1; j < Ncoef; j++){
                        fprintf(outptr,"\n%18.15f",decodeBest(Ncoef - j - 1));
       }


            fprintf(outptr, "\n\nThe best fitness is %f", bestChrom.fitness);

            fclose(outptr);
}

void AllChrom::DOcrossover(int loc1, int loc2, int coef_loc){
            int apoint, i, j;
            int abit;

            if ( UNI() < pcross ){
                        apoint = ANUMBER() % (Nbit - 2);
                        apoint++;

                        for ( i = 0; i < apoint : i++){
                                    abit = chrom[loc1].coef[coef_loc][i].bit;
                                    chrom[loc1].coef[coef_loc][i].bit = chrom[loc2].coef[coef_loc][i].bit;
                                    chrom[loc2].coef[coef_loc][i].bit = abit;
                        }
            }
}

void AllChrom:: DOmutation(int loc, int coefloc){
            int i,temp;

            for (i=0; i < Nbit; i++){
            if ( UNI() < pmut ){

                                    //check the original one
                                    temp = chrom[loc].coef[coefloc][i].bit;
                                    chrom[loc].coef[coefloc][i].bit = randBit(temp);
            }
       }

}
```

```
 int AllChrom::CSDcheck(int loc, int geneloc){
            int sumbit = 0;
            int i , a,b,c,rvalue = 1;

            for( i = 0; i < Nbit; i++) sumbit = sumbit + abs(chrom[loc].coef[geneloc][i].bit);

            if ( sumbit <= NCSDdigit ){
                        for ( i = 1 ; i < (Nbit - 1); i++){
                                    a = abs(chrom[loc].coef[geneloc][i].bit);
                                    b = abs(chrom[loc].coef[geneloc][i - 1].bit);
                                    c = abs(chrom[loc].coef[geneloc][i + 1].bit);

                                    if ( a & b ) goto JUMP;
                                    if ( a & c ) goto JUMP;
                        }
                        rvalue = 0;
            }

JUMP:    return(rvalue);
}

void AllChrom::reloadGene(int loc1, int loc2, int coefloc){
            int i;

            for( i =0; i < Nbit; i++){
                        chrom[loc1].coef[coefloc][i].bit = offsprg[loc1].coef[coefloc][i].bit;
                        chrom[loc2].coef[coefloc][i].bit = offsprg[loc2].coef[coefloc][i].bit;
            }
}

void AllChrom::preloadChrom(){
            int i;
            for( i =0; i < Nchrom; i++) chrom2offsprg(i , i);
}


void AllChrom::ArandChrom(int loc){                          //which chrom
            int i,j,k;
            int luckybit;

            for (j=0; j < Ncoef; j++){
                        for (i=0; i < Nbit; i++) chrom[loc].coef[j][i].bit = 0;

                        luckybit = ANUMBER() % Nbit;
                        chrom[loc].coef[j][luckybit].bit = randBit(0);
            }
}

//For offsprg
void AllChrom::ArandGene(int loc, int coef_loc){
            int i;
            int luckybit;

            for( i = 0; i < Nbit; i++)
                        chrom[loc].coef[coef_loc][i].bit = 0;
            luckybit = ANUMBER() % Nbit;
            chrom[loc].coef[coef_loc][luckybit].bit = randBit(0);
}

void AllChrom::AsmallGene(int loc, int coef_loc){
            int i;
```

```
        for( i = 0; i < (Nbit - 1); i++) chrom[loc].coef[coef_loc][i].bit = 0;
        chrom[loc].coef[coef_loc][Nbit - 1].bit = randBit(0);
}


//loc usually is 1
void AllChrom::GETbestChrom(){      //usually is 1 for index
        int j,i,loc;

        loc = -10;
        for( i =0 ; i < Nchrom; i++){
                if ( bestChrom.fitness < chrom[i].fitness ) loc = i;
        }

        if ( loc >= 0 ){
                bestChrom.fitness = chrom[loc].fitness;
                for(j=0;j < Ncoef; j++){
                        for(i=0; i < Nbit; i++) bestChrom.coef[j][i].bit = chrom[loc].coef[j][i].bit;
            }
        }
}

//don't include fitness
void AllChrom::chrom2offsprg(int chrom_loc, int offsprg_loc){
        int i,j;

        for(j=0;j < Ncoef; j++){
                for(i=0; i < Nbit; i++)
                        offsprg[offsprg_loc].coef[j][i].bit = chrom[chrom_loc].coef[j][i].bit;
                offsprg[offsprg_loc].fitness = chrom[chrom_loc].fitness;
    }
}

//include fitness
void AllChrom::offsprg2chrom(int offsprg_loc , int chrom_loc){
        int i,j;

        for(j=0;j < Ncoef; j++){
                for(i=0; i < Nbit; i++)
                        chrom[chrom_loc].coef[j][i].bit = offsprg[offsprg_loc].coef[j][i].bit;
                chrom[chrom_loc].fitness = offsprg[offsprg_loc].fitness;
    }
}
```

STEPS.CPP

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include "allchrom.cpp"

class Steps : public AllChrom {
private:
        int Noffsprg, Nchrom, Ncoef;
        int *ordering;
        char ob;

public:
    Steps(int,double,double,char, char*, int);
    ~Steps(){ }
```

```
		void presetChrom();				//watch out for preload
	void reproduce();		//duplicate the chrom to offsprg
		void GETchromfit();	//passing decoded coefficients to fitness
		double fitness(double*);
		void CMCfix();
};


Steps::Steps(int Nchromosome, double pmutation, double pcrossover, char ob_ch, char *name, int
Ncoeff):AllChrom(Nchromosome, pmutation,pcrossover, name, Ncoeff){
		int i;

		Nchrom = Nchromosome;
		Noffsprg = Nchrom;
		Ncoef = Ncoeff;

		ordering = new int [Nchrom];
		if (ordering == NULL){
			printf("\nno memory for ordering");
			exit(-1);
		}

		ob = ob_ch;
		for ( i =0; i < Nchrom; i++) ordering[i] = i;
}

void Steps::presetChrom(){
		int j,i;

		for(j = 0 ; j < Nchrom; j++){
		ArandChrom(j);
		}

		//************check for the initial data **************
/*		for ( j=0; j < Nchrom; j++){
			OUTchrom(j);
			printf("\n");
		}
		scanf("%d",&i);
*/
}


//duplicate and set the Noffsprg
void Steps::reproduce(){
	int loc, i, j, *checked;
		double value;

		checked = new int[Nchrom];
		if ( checked == NULL){
			printf("\nNot enough memory for *chrom");
			exit(-1);
		}

		Noffsprg = ANUMBER() % ( Nchrom - 2 );
		Noffsprg = Noffsprg + 2;
		if ((Noffsprg % 2) != 0 ) Noffsprg++;

		for(i = 0; i < Nchrom; i++) checked[i] = 0;

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

```
                value = 0;
                for( j = 0; j < Nchrom; j++){
                        if ( (checked[j] != 1) && ( value < OUTchromfitness(j) ) ){
                                value = OUTchromfitness(j);
                                loc = j;
                        }
                }
                ordering[i] = loc;
                checked[loc] = 1;
        }

        delete[] checked;
}


void Steps::CMCfix(){
        int coef_count,j;
        int flag1, flag2 , count;

        preloadChrom();

        for ( j = 0; j < Noffsprg; j = j + 2){
                for ( coef_count = 0; coef_count < Ncoef; coef_count++){

                        //check for CSD format
                        count = 0;
                        flag2 = 1;

                        while( (flag2) && (count < 2) ){
                                reloadGene(ordering[j], ordering[j + 1], coef_count);

                                DOcrossover(ordering[j], ordering[j + 1], coef_count);
                                DOmutation(ordering[j], coef_count);
                                DOmutation(ordering[j + 1], coef_count);

                                flag1 = CSDcheck(ordering[j],     coef_count);
                                if ( flag1 == 0) flag2 = CSDcheck(ordering[j + 1], coef_count);
                                count++;
                        }

                        //if all five chances fail
                        if ( (flag1 != 0) || (flag2 != 0) ){
                                flag2 = CSDcheck(ordering[j +1], coef_count);
                                if (flag1 != 0 ) ArandGene(ordering[j], coef_count);
                                if (flag2 != 0 ) ArandGene(ordering[j +1], coef_count);
                        }

                        //check for zero coefficients
                        if ( decodeChrom(ordering[j], coef_count) == 0.0 )  ArandGene(ordering[j], coef_count);
                        if ( decodeChrom(ordering[j + 1], coef_count) == 0.0 ) ArandGene(ordering[j + 1], coef_count);
                }
        }

        //refill the rest of the extra space
        coef_count = 0;
        for ( j = Noffsprg; j < Nchrom; j++){
                offsprg2chrom( ordering[coef_count], ordering[j]);
                coef_count++;
        }
}
```

```
 void Steps::GETchromfit(){   //passing decoded coefficients to fitness
            int j,i;
            double *coeff, val;

            coeff = new double[Ncoef];
            if ( coeff == NULL){
                        printf("\nNot enough memory for *coeff");
                        exit(-1);
            }

            for( j = 0; j < Noffsprg; j++){
                        for( i=0; i < Ncoef; i++){
                        coeff[i] = decodeChrom(ordering[j],i);
                        }

                        val = fitness(coeff);
                        INchromfitness(ordering[j],val);
            }
            delete[] coeff;
}

double Steps::fitness(double *coeff){
            int i,j, flag , count;
            double err,w, delta_w;
            double H, temp;
            double return_val, max_err;

      delta_w = maxFreq / Npoints;

      err = 0;
            max_err = 0;
            for (w = 0 ; w < maxFreq + delta_w; w = w + delta_w){

                        //at one frequency of the first half symmetry coefficients
            count = 0;

                        H = coeff[Ncoef - 1];
                        for ( j = 1; j < Ncoef; j++){
                                    H = H + 2.0 * coeff[Ncoef - j - 1] * cos(w * T * j);
                        }

                        H = fabs(H);


                        //minmax or LMS
                        if ( ob == 'l'){

                                    //exp decay transision region
                                    if ( w < passbd )
                                                err = err + fabs( 1.0 - H);
                                    else
                                                err = err + fabs( exp(-1.0*ALPHA*(w - passbd)) - H );

                        } else if(ob == 'm') {

                                    //get the magnitude response error
                                    if ( w < passbd + delta_w)   err = fabs( 1.0 - H );
                                    else if ( w > stopbd - delta_w )  err = fabs(H);
                                    else { }
                                    if( max_err < err ) max_err = err;
```

```
                } else {
                        printf("\ncannot get the objective function\n");
                        exit(-1);
                }

        }

        if (ob == 'l') return_val = 1.0 / err;
        else return_val = 1.0 / max_err;

        return( return_val );

}


//*****************************************************************************

class final:public Steps {
private:
        int MaxIter;

public:
     final(int,int,double,double,char,char*, int);
        ~final(){ }
        void process();
};

final::final(int MaxIteration, int chrom_size, double mut_prob, double cross_prob, char ob_ch, char *name, int
Ncoeff):Steps(chrom_size,mut_prob, cross_prob, ob_ch, name, Ncoeff){
        MaxIter = MaxIteration;
}


void final::process(){
        int Citer;

    //get the random chromosome
        presetChrom();

        //calcuate the fitness of each chromosome
        GETchromfit();

        scaleFitness();

        Citer = 0;
        while ( Citer < MaxIter ){
                Citer++;

            //find Noffsprg and duplicate the chrom to offsprg
                        reproduce();

                        //crossover, mutation, and CSD checkNfix
                        CMCfix();

                        //calcuate fitness of each offsprg
                        GETchromfit();

            //store the best chromosome comparing to the history
                        GETbestChrom();

                        scaleFitness();
                        printf("\n%d          %f", Citer, OUTbestfitness() );

        }
        printf("\n");
        report();
}


main(int argsize, char **arg){
        int popsize, iterations;
        double mutation, crossover;
        int Ncoeff;

//      arg[1] - m for minimax or l for LMS
//      arg[6] - output filename

        if ( argsize == 8 ){
                iterations = atoi(arg[2]);
                popsize    = atoi(arg[3]);
                mutation   = ((double) atoi(arg[4])) / 100.0;
                crossover  = ((double) atoi(arg[5])) / 100.0;
                Ncoeff     = atoi(arg[6]);

                final object( iterations, popsize, mutation, crossover, *arg[1], arg[7], Ncoeff);
                object.process();
        }
        else printf("\nwrong argument");
}
```

Section 3

This section involves the combination of mixed encoding with roulette wheel parent selection.

ALLCHROM.CPP

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

#define Ndigit   5
#define Nbit        3
#define MAX_WL   31
#define T              1.0      //sampling frequency
#define maxFreq 3.1416
#define Npoints 70 //# of sampling points
#define stopbd      2.0      //either stopbd or ALPHA is used
#define ALPHA  6.0
#define passbd      1.0

typedef struct {
        unsigned int bit:1;
} BINARY;

typedef struct {
        signed int sbit:2;
} CSDbit;

typedef struct {
        BINARY coef[50][Ndigit][Nbit];
        CSDbit signbit[50][Ndigit];
        double fitness;
} CHMtype;


//This class will deal with all the single bit things
class AllChrom {
private:
        CHMtype *chrom, *offsprg, bestChrom;
    double pmut, pcross;
        int    Nchrom, Ncoef;
        FILE *outptr;
        char outfilename[22];

public:
    AllChrom(int,double,double,char*, int);
        ~AllChrom();
        double decodeChrom(int,int);
        int decodeChrom(int,int, int);
    double decodeBest(int);
    int decodeBest(int, int);
        int randBit();
        int randBit(int);
        int CSDcheck(int,int);                      //which offsprg, which gene
        void report();
        void DOcrossover(int,int ,int);             //which 2 chrom, which gene, which gene, which 2 offsprg
```

```
		void DOmutation(int,int);
		void ArandChrom(int);			//which chrom
		void GETbestChrom();	//usually is 1 for index
		void chrom2offsprg(int,int);
	void offsprg2chrom(int,int);
		void reloadGene(int, int,int,int);
		void preloadChrom();
		void ArandGene(int,int);
		int randBit3(int);
		void orderDigits();
		void exchangeDigit(int, int, int,int);

		void INchromfitness(int loc, double val)		{ chrom[loc].fitness = val; }
		double OUTchromfitness(int loc)	{ return(chrom[loc].fitness); }
		void INoffsprgfitness(int loc, double val)		{ offsprg[loc].fitness = val; }
		double OUToffsprgfitness(int loc)		{ return(offsprg[loc].fitness); }
		double OUTbestfitness()			{ return(bestChrom.fitness); }
};

int AllChrom::randBit(){
		int numb;

		numb = ANUMBER() % 2;
		return( numb );
}

int AllChrom::randBit(int except){
		if (except == 1) return(0);
		else return(1);
}


int AllChrom::randBit3(int except){
		int numb = except;

		while (numb == except){
				numb = ANUMBER() % 3;
				numb = numb - 1;
		}
		return( numb );
}

AllChrom::AllChrom(int Nchromosome, double pmutation, double pcrossover, char *fname, int Ncoeff){
		int i;

		Ncoef = Ncoeff;
		Nchrom = Nchromosome;
		pmut = pmutation;
	pcross = pcrossover;

		chrom = new CHMtype[Nchrom];
		if ( chrom == NULL){
				printf("\nNot enough memory for *chrom");
				exit(-1);
		}

		offsprg = new CHMtype[Nchrom];
		if ( offsprg == NULL){
				printf("\nNot enough memory for *offsprg");
				exit(-1);
		}
```

```
            bestChrom.fitness = 0.0;

            for(i = 0; i < strlen(fname); i++) outfilename[i] = fname[i];
}

AllChrom::~AllChrom(){
            delete[] chrom;
      delete[] offsprg;
}

//decodeChrom
double AllChrom::decodeChrom(int chrom_loc, int coef_loc){
            double anumber, ternery;
            int asign;
            int n,i;

            anumber = 0;

            for ( i=0; i < Ndigit; i++){
                        asign = chrom[chrom_loc].signbit[coef_loc][i].sbit;
                        if ( asign != 0) {
                                    ternery = decodeChrom(chrom_loc, coef_loc, i);
                                    anumber = anumber + asign * pow( 2.0 , (-1.0 * ternery) );
                        }
            }

            return(anumber);
}

int AllChrom::decodeChrom(int chrom_loc, int coef_loc, int t_value){
            double anumber, ternery;
            int asign;
            int n,k;

            ternery = 0;
            asign = chrom[chrom_loc].signbit[coef_loc][t_value].sbit;

            if( asign != 0 ){
                        for ( k = 0; k < Nbit; k++){
                                    if ( chrom[chrom_loc].coef[coef_loc][t_value][k].bit == 1 ) ternery = ternery + pow(2, k);
                        }
            }

            return(ternery);
}


//decodeBestchrom
double AllChrom::decodeBest(int coef_loc){
            double anumber, ternery;
            int asign;
            int n,i;

            anumber = 0;

            for ( i=0; i < Ndigit; i++){
                        asign = bestChrom.signbit[coef_loc][i].sbit;
                        if( asign != 0 ){
                                    ternery = decodeBest(coef_loc, i);
```

```
                              anumber = anumber + asign * pow(2.0 , (-1.0 * ternery) );
                        }
            }

            return(anumber);
}


int AllChrom::decodeBest(int coef_loc, int t_value){
            double anumber, ternery;
            int asign;
            int n,k;

            ternery = 0;
            asign = bestChrom.signbit[coef_loc][t_value].sbit;

            if( asign != 0 ){
                        for ( k = 0; k < Nbit; k++){
                                    if ( bestChrom.coef[coef_loc][t_value][k].bit == 1 ) ternery = ternery + pow(2, k);
                        }
            }

            return(ternery);
}



void AllChrom::report(){
            int i,j,flag,k;

            outptr = fopen( outfilename ,"w+");
            if ( outptr == NULL){
                        printf("cannot open output file");
            exit(-1);
      }

            fprintf(outptr,"\n\n");
            for(j = 0; j < Ncoef; j++){
                        for (k=0; k < Ndigit; k++){
                                    fprintf(outptr,"%4d#", bestChrom.signbit[j][k].sbit);
                                    fprintf(outptr,"%2d", decodeBest(j, k) );
                        }
                        fprintf(outptr,"\n");
            }
            for(j = 1; j < Ncoef; j++){
                        for (k=0; k < Ndigit; k++){
                                    fprintf(outptr,"%4d#", bestChrom.signbit[Ncoef - j - 1][k].sbit);
                                    fprintf(outptr,"%2d", decodeBest( (Ncoef - 1 - j), k) );
                        }
                        fprintf(outptr,"\n");
            }


            fprintf(outptr,"\n\n");
            for(j = 0; j < Ncoef; j++) fprintf(outptr,"%f\n",decodeBest(j) );
            for(j = 1 ; j < Ncoef; j++) fprintf(outptr,"%f\n",decodeBest(Ncoef - 1 - j ) ) ;

            fprintf(outptr, "\n\nThe best fitness is %f", bestChrom.fitness);

            fclose(outptr);
}
```

```
 void AllChrom::DOcrossover(int loc1, int loc2, int coef_loc){
            int apoint,i,j, k;
       int abit, asign1, asign2;

            for (k = 0; k < Ndigit; k++){
                    if (UNI() < pcross){

                            asign1 = chrom[loc1].signbit[coef_loc][k].sbit;
                            asign2 = chrom[loc2].signbit[coef_loc][k].sbit;

                            if( asign1 != asign2 ){
                                    if ( UNI() < 0.05 ){
                                            chrom[loc1].signbit[coef_loc][k].sbit = randBit3(5);
                                            chrom[loc2].signbit[coef_loc][k].sbit = randBit3(5);
                                    }
                            }

                            asign1 = chrom[loc1].signbit[coef_loc][k].sbit;
                            asign2 = chrom[loc2].signbit[coef_loc][k].sbit;

                            apoint = ANUMBER() % (Nbit - 2);
                            apoint ++;

                            for ( i = 0; i < apoint ; i++){
                                    abit = chrom[loc1].coef[coef_loc][k][i].bit;
                                    chrom[loc1].coef[coef_loc][k][i].bit = chrom[loc2].coef[coef_loc][k][i].bit;
                                    chrom[loc2].coef[coef_loc][k][i].bit = abit;
                            }

                    }
            }
}

void AllChrom:: DOmutation(int loc, int coefloc){
            int i,temp,j,k,asign;

            for ( k = 0; k < Ndigit; k++){
                    if ( UNI() < pmut ) chrom[loc].signbit[coefloc][k].sbit = randBit3(5);
                    asign = offsprg[loc].signbit[coefloc][k].sbit;
                    if ( asign != 0 ){
                            for (i=0; i < Nbit; i++){
                                  if ( UNI() < pmut ){

                                            //check the original one
                                            temp = chrom[loc].coef[coefloc][k][i].bit;
                                            chrom[loc].coef[coefloc][k][i].bit = randBit(temp);
                            }
                            }
                    }
       }
}

void AllChrom::orderDigits(){
            int i,j,k, l,store[Ndigit];

            for(i = 0; i < Nchrom; i++){
                    for( j = 0; j < Ncoef; j++){

                            for( k = 0; k < Ndigit; k++) store[k] = decodeChrom(i, j, k);
                            for(l = 0; l < (Ndigit - 1); l++){
                                    for(k = l + 1; k < Ndigit; k++){
                                            if( store[k] > store[l] ) exchangeDigit(i,j,l, k);
                                    }
                            }
                    }
            }
    }

    void AllChrom::exchangeDigit(int a, int b, int c, int d){
            int temp[Nbit], temp2, i;

            for( i = 0; i < Nbit; i++) temp[i] = chrom[a].coef[b][c][i].bit;
            for( i = 0; i < Nbit; i++) chrom[a].coef[b][c][i].bit = chrom[a].coef[b][d][i].bit;
            for( i = 0; i < Nbit; i++) chrom[a].coef[b][d][i].bit = temp[i];

            temp2 = chrom[a].signbit[b][c].sbit;
            chrom[a].signbit[b][c].sbit = chrom[a].signbit[b][d].sbit;
            chrom[a].signbit[b][d].sbit = temp2;
    }


    int AllChrom::CSDcheck(int loc, int geneloc){
            int sum,i,j,numbplus,numbminu, rvalue, store[Ndigit];

            rvalue = 1;
            for(i = 0; i < Ndigit; i++) store[i] = decodeChrom(loc, geneloc, i);

            for(i=0; i < Ndigit; i++){ if ( store[i] >= MAX_WL ) goto JUMP; }

            for(i = 0; i < (Ndigit - 1); i++){
                    numbplus = store[i] + 1;
                    numbminu = store[i] - 1;
                    for ( j = i + 1; j < Ndigit; j++){
                            if ( (numbplus == store[j]) || ( numbminu == store[j]) ) goto JUMP;
                            if ( store[i] ==  store[j] ) goto JUMP;
                    }
            }

            rvalue = 0;

JUMP:       return(rvalue);
}

void AllChrom::reloadGene(int a, int loc1, int loc2, int coefloc){
            int i, k;

            for( k =0; k < Ndigit; k++){
                    for( i =0; i < Nbit; i++){
                            chrom[a].coef[coefloc][k][i].bit = offsprg[loc1].coef[coefloc][k][i].bit;
                    }
                    chrom[a].signbit[coefloc][k].sbit = offsprg[loc1].signbit[coefloc][k].sbit;
            }
            for( k =0; k < Ndigit; k++){
                    for( i =0; i < Nbit; i++){
                            chrom[a + 1].coef[coefloc][k][i].bit = offsprg[loc2].coef[coefloc][k][i].bit;
                    }
                    chrom[a + 1].signbit[coefloc][k].sbit = offsprg[loc2].signbit[coefloc][k].sbit;
            }
}

void AllChrom::preloadChrom(){
```

```
            int i;
            for( i =0; i < Nchrom; i++) chrom2offsprg(i , i);
 }


 void AllChrom::ArandChrom(int loc){                              //which chrom
            int i,j,k;
            int luckybit;

            for (j=0; j < Ncoef; j++){
                       for( k = 0; k < Ndigit; k++){
                                  for (i=0; i < Nbit; i++) chrom[loc].coef[j][k][i].bit = randBit();
                                  while( decodeChrom(loc, j, k) >= MAX_WL ){ for (i=0; i < Nbit; i++) chrom[loc].coef[j][k][i].bit
= randBit(); }
                       }
            }

            for ( j=0; j < Ncoef; j++){
                       for ( k =0; k < Ndigit; k++) chrom[loc].signbit[j][k].sbit = 0;
                       luckybit = ANUMBER() % Ndigit;
                       chrom[loc].signbit[j][luckybit].sbit = randBit3(0);
            }
}




//For offsprg
void AllChrom::ArandGene(int loc, int coef_loc){
           int i,k, sum;
           int luckybit;

           sum = 0;
           for ( i = 0; i < Ndigit; i++ ) sum =  sum + abs( chrom[loc].signbit[coef_loc][i].sbit );

           if ( sum == 0 ){
                      luckybit = ANUMBER() % Ndigit;
                      chrom[loc].signbit[coef_loc][luckybit].sbit = randBit3(0);
                      while( decodeChrom(loc, coef_loc, luckybit) >= MAX_WL ) {
                                 for (i=0; i < Nbit; i++) chrom[loc].coef[coef_loc][luckybit][i].bit = randBit();
                      }
           }
           else{
                      for ( k =0; k < Ndigit; k++) chrom[loc].signbit[coef_loc][k].sbit = 0;
                      luckybit = ANUMBER() % Ndigit;
                      chrom[loc].signbit[coef_loc][luckybit].sbit = randBit3(0);
                      while( decodeChrom(loc, coef_loc, luckybit) >= MAX_WL ) {
                                 for (i=0; i < Nbit; i++) chrom[loc].coef[coef_loc][luckybit][i].bit = randBit();
                      }
           }
}


//loc usually is 1
void AllChrom::GETbestChrom(){      //usually is 1 for index
           int j,i,loc,k;

           loc = -10;
           for( i =0 ; i < Nchrom; i++){
                      if ( bestChrom.fitness < chrom[i].fitness ) loc = i;
           }
```

```
        if( loc >= 0 ){
                bestChrom.fitness = chrom[loc].fitness;
                for(j=0;j < Ncoef; j++){
                        for(k=0; k < Ndigit; k++){
                                for(i=0; i < Nbit; i++) bestChrom.coef[j][k][i].bit = chrom[loc].coef[j][k][i].bit;
                                bestChrom.signbit[j][k].sbit = chrom[loc].signbit[j][k].sbit;
                        }
            }
        }
}

void AllChrom::chrom2offsprg(int chrom_loc, int offsprg_loc){
        int i,j,k;

        for(j=0;j < Ncoef; j++){
                for(k=0; k < Ndigit; k++){
                        for(i=0; i < Nbit; i++)
                                offsprg[offsprg_loc].coef[j][k][i].bit = chrom[chrom_loc].coef[j][k][i].bit;
                        offsprg[offsprg_loc].signbit[j][k].sbit = chrom[chrom_loc].signbit[j][k].sbit;
                }
        }
        offsprg[offsprg_loc].fitness = chrom[chrom_loc].fitness;
}

//include fitness
void AllChrom::offsprg2chrom(int offsprg_loc , int chrom_loc){
        int i,j,k;

        for(j=0;j < Ncoef; j++){
                for(k=0; k < Ndigit; k++){
                        for(i=0; i < Nbit; i++)
                                chrom[chrom_loc].coef[j][k][i].bit = offsprg[offsprg_loc].coef[j][k][i].bit;
                        chrom[chrom_loc].signbit[j][k].sbit = offsprg[offsprg_loc].signbit[j][k].sbit;
                }
        }
        chrom[chrom_loc].fitness = offsprg[offsprg_loc].fitness;
}
```

STEPS.CPP

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include "allchrom.cpp"

class Steps : public AllChrom {
private:
        int Noffsprg, Nchrom, Ncoef;
        int *ordering;
        char ob;

public:
    Steps(int,double,double,char, char*,int);
    ~Steps(){ }
        void presetChrom();                             //watch out for preload
    void reproduce();       //duplicate the chrom to offsprg
        void GETchromfit();  //passing decoded coefficients to fitness
        double fitness(double*);
        void CMCfix();
```

```
};


Steps::Steps(int Nchromosome, double pmutation, double pcrossover, char ob_ch, char *name, int
Ncoeff):AllChrom(Nchromosome, pmutation,pcrossover, name, Ncoeff){

        Ncoef = Ncoeff;
        Nchrom = Nchromosome;
        Noffsprg = Nchrom;

        ordering = new int [Nchrom];
        if (ordering == NULL){
                printf("\nno memory for ordering");
                exit(-1);
        }

        ob = ob_ch;
}

void Steps::presetChrom(){
        int j,i;

        for(j = 0 ; j < Nchrom; j++){
        ArandChrom(j);
        }

        //************check for the initial data **************
/*      for ( j=0; j < Nchrom; j++){
                OUTchrom(j);
                printf("\n");
        }
        scanf("%d",&i);
*/
}


//duplicate and set the Noffsprg
void Steps::reproduce(){
    int i, count;
        double rawfitness, total, sum;

        total = -1.0E-7;
        for( i = 0; i < Nchrom; i++) total = total + OUTchromfitness(i);


        for( i = 0; i < Noffsprg; i++){
                rawfitness = UNI() * total;
                sum = OUTchromfitness(0);
                count = 0;
                while( sum < rawfitness ){
                        count++;
                        sum = sum + OUTchromfitness(count);
                }

                ordering[i] = count;
        }
}


void Steps::CMCfix(){
        int coef_count,j;
```

```
            int flag1, flag2 , count;

            preloadChrom();

            for ( j = 0; j < Noffsprg; j = j + 2){
                        for ( coef_count = 0; coef_count < Ncoef; coef_count++){

                                    //check for CSD format
                                    count = 0;
                                    flag2 = 1;

                                    while( (flag2) && (count < 6) ){
                                                reloadGene(j,ordering[j], ordering[j + 1], coef_count);

                                                DOcrossover(j, j + 1, coef_count);
                                                DOmutation(j, coef_count);
                                                DOmutation(j + 1, coef_count);

                                                flag1 = CSDcheck(j,     coef_count);
                                                if ( flag1 == 0) flag2 = CSDcheck(j + 1, coef_count);
                                                count++;
                                    }

                                    //if all five chances fail
                                    if ( (flag1 != 0) || (flag2 != 0) ){
                                                flag2 = CSDcheck(j +1, coef_count);
                                                if (flag1 != 0 ) ArandGene(j, coef_count);
                                                if (flag2 != 0 ) ArandGene(j +1, coef_count);
                                    }

                                    //check for zero coefficients
                                    if ( decodeChrom(j, coef_count) == 0.0 )
                                                ArandGene(j, coef_count);
                                    if ( decodeChrom(j + 1, coef_count) == 0.0 )
                                                ArandGene(j + 1, coef_count);
                        }
            }
}

void Steps::GETchromfit(){   //passing decoded coefficients to fitness
            int j,i;
            double *coeff, val;

            coeff = new double[Ncoef];
            if ( coeff == NULL){
                        printf("\nNot enough memory for *coeff");
                        exit(-1);
            }

            for( j = 0; j < Nchrom; j++){
                        for( i=0; i < Ncoef; i++){
                        coeff[i] = decodeChrom(j,i);
                        }

                        val = fitness(coeff);
                        INchromfitness(j,val);
      }

            delete[] coeff;
}
```

```
 double Steps::fitness(double *coeff){
            int i,j, flag , count;
            double err,w, delta_w, magH;
            double H, anumber, temp;
            double sum, return_val, max_err;

       delta_w = maxFreq / Npoints;

       err = 0;
             max_err = 0;
             for (w = 0 ; w < maxFreq + delta_w; w = w + delta_w){

                       //at one frequency of the first half symmetry coefficients
             count = 0;


                       H = coeff[Ncoef - 1];
                       for ( j = 1; j < Ncoef; j++){
                                 H = H + 2.0 * coeff[Ncoef - j - 1] * cos(w * T * j);
                       }

                       H = fabs(H);


                       //minmax or LMS
                       if ( ob == 'l'){

                                 //exp decay transision region
                                 if ( w < passbd )
                                           err = err + fabs( 1.0 - H);
                                 else
                                           err = err + fabs( exp(-1.0*ALPHA*(w - passbd)) - H );

                       } else if(ob == 'm') {

                                 //get the magnitude response error
                                 if ( w < passbd + delta_w)  err = fabs( 1.0 - H );
                                 else if ( w > stopbd - delta_w ) err = fabs(H);
                                 else { }
                                 if( max_err < err ) max_err = err;

                       } else {
                                 printf("\ncannot get the objective function\n");
                                 exit(-1);
                       }

            }

            if (ob == 'l') return_val  =  1.0 / err;
            else return_val  =  1.0 / max_err;

            return( return_val );

}

//****************************************************************************

class final:public Steps {
private:
            int MaxIter;
```

```
public:
      final(int,int,double,double,char,char*,int);
            ~final() { }
            void process();
};

final::final(int MaxIteration, int chrom_size, double mut_prob, double cross_prob, char ob_ch, char *name, int
Ncoeff):Steps(chrom_size,mut_prob, cross_prob, ob_ch, name, Ncoeff){
            MaxIter = MaxIteration;
}


void final::process(){
            int Citer;

      //get the random chromosome
            presetChrom();

            orderDigits();

            //calcuate the fitness of each chromosome
            GETchromfit();

            Citer = 0;
            while ( Citer < MaxIter ){
                        Citer++;

                  //find Noffsprg and duplicate the chrom to offsprg
                                    reproduce();

                                    //crossover, mutation, and CSD checkNfix
                                    CMCfix();

                                    orderDigits();

                                    //calcuate fitness of each offsprg
                                    GETchromfit();

                  //store the best chromosome comparing to the history
                                    GETbestChrom();

                                    printf("\n%d          %f", Citer, OUTbestfitness() );

            }
            printf("\n");
            report();
}

main(int argsize, char **arg){
            int popsize, iterations;
            double mutation, crossover;
            int Ncoeff;

//          arg[1] - m for minimax or l for LMS
//          arg[6] - output filename

            if ( argsize == 8 ){
                        iterations = atoi(arg[2]);
                        popsize    = atoi(arg[3]);
                        mutation   = ((double) atoi(arg[4])) / 100.0;
```

```
                    crossover  = ((double) atoi(arg[5])) / 100.0;
                    Ncoeff     = atoi(arg[6]);

//                  printf("\niteration is %d popsize is %d ", iterations, popsize );
//                  printf("mutation is %f crossover is %f\n", mutation, crossover );
//                  printf("The order of the filter is %d", Ncoeff - 1);

                    final object( iterations, popsize, mutation, crossover, *arg[1], arg[7], Ncoeff);
                    object.process();
          }
          else printf("\nwrong argument");
}
```

Section 4

This section involves the combination of mixed encoding with ranking parent selection.

ALLCHROM.CPP

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

#define Ndigit   3  //3
#define Nbit         4
#define MAX_WL   12
#define T              1.0       //sampling frequency
#define maxFreq 3.1416
#define Npoints 70 //# of sampling points
#define stopbd      2.0       //either stopbd or ALPHA is used
#define ALPHA  6.0
#define passbd      1.0

typedef struct {
        unsigned int bit:1;
} BINARY;

typedef struct {
        signed int sbit:2;
} CSDbit;

typedef struct {
        BINARY coef[50][Ndigit][Nbit];
        CSDbit signbit[50][Ndigit];
        double fitness;
} CHMtype;


//This class will deal with all the single bit things
class AllChrom {
private:
        CHMtype *chrom, *offsprg, bestChrom;
    double pmut, pcross;
        int    Nchrom, Ncoef;
        FILE *outptr;
        char outfilename[22];

public:
    AllChrom(int,double,double,char*, int);
        ~AllChrom();
        double decodeChrom(int,int);
        int decodeChrom(int,int, int);
    double decodeBest(int);
    int decodeBest(int, int);
        int randBit();
        int randBit(int);
        int CSDcheck(int,int);                      //which offsprg, which gene
        void report();
        void DOcrossover(int,int ,int);             //which 2 chrom, which gene, which gene, which 2 offsprg
```

```
            void DOmutation(int,int);
            void ArandChrom(int);                          //which chrom
            void GETbestChrom();        //usually is 1 for index
            void chrom2offsprg(int,int);
        void offsprg2chrom(int,int);
            void reloadGene(int,int,int);
            void preloadChrom();
            void ArandGene(int,int);
            int randBit3(int);
            void orderDigits();
            void exchangeDigit(int, int, int,int);

            void INchromfitness(int loc, double val)         { chrom[loc].fitness = val; }
            double OUTchromfitness(int loc)        { return(chrom[loc].fitness); }
            void INoffsprgfitness(int loc, double val)       { offsprg[loc].fitness = val; }
            double OUToffsprgfitness(int loc)                { return(offsprg[loc].fitness); }
            double OUTbestfitness()                          { return(bestChrom.fitness); }
};

int AllChrom::randBit(){
            int numb;

            numb = ANUMBER() % 2;
            return( numb );
}

int AllChrom::randBit(int except){
            if (except == 1) return(0);
            else return(1);
}


int AllChrom::randBit3(int except){
            int numb = except;

            while (numb == except){
                        numb = ANUMBER() % 3;
                        numb = numb - 1;
            }
            return( numb );
}

AllChrom::AllChrom(int Nchromosome, double pmutation, double pcrossover, char *fname, int Ncoeff){
            int i;

            Ncoef = Ncoeff;
            Nchrom = Nchromosome;
            pmut = pmutation;
        pcross = pcrossover;

            chrom = new CHMtype[Nchrom];
            if ( chrom == NULL){
                        printf("\nNot enough memory for *chrom");
                        exit(-1);
            }

            offsprg = new CHMtype[Nchrom];
            if ( offsprg == NULL){
                        printf("\nNot enough memory for *offsprg");
                        exit(-1);
            }
            bestChrom.fitness = 0.0;

            for(i = 0; i < strlen(fname); i++) outfilename[i] = fname[i];
}

AllChrom::~AllChrom(){
            delete[] chrom;
      delete[] offsprg;
}

//decodeChrom
double AllChrom::decodeChrom(int chrom_loc, int coef_loc){
            double anumber, ternery;
            int asign;
            int n,i;

            anumber = 0;

            for ( i=0; i < Ndigit; i++){
                        asign = chrom[chrom_loc].signbit[coef_loc][i].sbit;
                        if ( asign != 0) {
                                    ternery = decodeChrom(chrom_loc, coef_loc, i);
                                    anumber = anumber + asign * pow( 2.0 , (-1.0 * ternery) );
                        }
            }

            return(anumber);
}

int AllChrom::decodeChrom(int chrom_loc, int coef_loc, int t_value){
            double anumber, ternery;
            int asign;
            int n,k;

            ternery = 0;
            asign = chrom[chrom_loc].signbit[coef_loc][t_value].sbit;

            if( asign != 0 ){
                        for ( k = 0; k < Nbit; k++){
                                    if ( chrom[chrom_loc].coef[coef_loc][t_value][k].bit == 1 ) ternery = ternery + pow(2, k);
                        }
            }

            return(ternery);
}



//decodeBestchrom
double AllChrom::decodeBest(int coef_loc){
            double anumber, ternery;
            int asign;
            int n,i;

            anumber = 0;

            for ( i=0; i < Ndigit; i++){
                        asign = bestChrom.signbit[coef_loc][i].sbit;
                        if( asign != 0 ){
                                    ternery = decodeBest(coef_loc, i);
```

```
                                        anumber = anumber + asign * pow(2.0 , (-1.0 * ternery) );
                                }
                        }

                        return(anumber);
        }


        int AllChrom::decodeBest(int coef_loc, int t_value){
                        double anumber, ternery;
                        int asign;
                        int n,k;

                        ternery = 0;
                        asign = bestChrom.signbit[coef_loc][t_value].sbit;

                        if( asign != 0 ){
                                for ( k = 0; k < Nbit; k++){
                                        if ( bestChrom.coef[coef_loc][t_value][k].bit == 1 ) ternery = ternery + pow(2, k);
                                }
                        }

                        return(ternery);
        }



        void AllChrom::report(){
                        int i,j,flag,k;

                        outptr = fopen( outfilename ,"w+");
                        if ( outptr == NULL){
                                printf("cannot open output file");
                        exit(-1);
                }

                        fprintf(outptr,"\n\n");
                        for(j = 0; j < Ncoef; j++){
                                for (k=0; k < Ndigit; k++){
                                        fprintf(outptr,"%4d#", bestChrom.signbit[j][k].sbit);
                                        fprintf(outptr,"%2d", decodeBest(j, k) );
                                }
                                fprintf(outptr,"\n");
                        }
                        for(j = 1; j < Ncoef; j++){
                                for (k=0; k < Ndigit; k++){
                                        fprintf(outptr,"%4d#", bestChrom.signbit[Ncoef - j - 1][k].sbit);
                                        fprintf(outptr,"%2d", decodeBest( (Ncoef - 1 - j), k) );
                                }
                                fprintf(outptr,"\n");
                        }


                        fprintf(outptr,"\n\n");
                        for(j = 0; j < Ncoef; j++) fprintf(outptr,"%f\n",decodeBest(j) );
                        for(j = 1 ; j < Ncoef; j++) fprintf(outptr,"%f\n",decodeBest(Ncoef - 1 - j ) ) ;

                        fprintf(outptr, "\n\nThe best fitness is %f", bestChrom.fitness);

                        fclose(outptr);
        }
```

```
void AllChrom::DOcrossover(int loc1, int loc2, int coef_loc){
        int apoint,i,j, k;
    int abit, asign1, asign2;

        for (k = 0; k < Ndigit; k++){
                if (UNI() < pcross){

                        asign1 = chrom[loc1].signbit[coef_loc][k].sbit;
                        asign2 = chrom[loc2].signbit[coef_loc][k].sbit;

                        if( asign1 != asign2 ){
                                if ( UNI() < 0.05 ){
                                        chrom[loc1].signbit[coef_loc][k].sbit = randBit3(5);
                                        chrom[loc2].signbit[coef_loc][k].sbit = randBit3(5);
                                }
                        }

                        asign1 = chrom[loc1].signbit[coef_loc][k].sbit;
                        asign2 = chrom[loc2].signbit[coef_loc][k].sbit;

                        apoint = ANUMBER() % (Nbit - 2);
                        apoint ++;

                        for ( i = 0; i < apoint ; i++){
                                abit = chrom[loc1].coef[coef_loc][k][i].bit;
                                chrom[loc1].coef[coef_loc][k][i].bit = chrom[loc2].coef[coef_loc][k][i].bit;
                                chrom[loc2].coef[coef_loc][k][i].bit = abit;
                        }

                }
        }
}

void AllChrom:: DOmutation(int loc, int coefloc){
        int i,temp,j,k,asign;

        for ( k = 0; k < Ndigit; k++){
                if ( UNI() < pmut ) chrom[loc].signbit[coefloc][k].sbit = randBit3(5);
                asign = offsprg[loc].signbit[coefloc][k].sbit;
                if ( asign != 0 ){
                        for (i=0; i < Nbit; i++){
                            if ( UNI() < pmut ){

                                        //check the original one
                                        temp = chrom[loc].coef[coefloc][k][i].bit;
                                        chrom[loc].coef[coefloc][k][i].bit = randBit(temp);
                        }
                        }
                }
    }
}

void AllChrom::orderDigits(){
        int i,j,k, l,store[Ndigit];

        for(i = 0; i < Nchrom; i++){
                for( j = 0; j < Ncoef; j++){

                        for( k = 0; k < Ndigit; k++) store[k] = decodeChrom(i, j, k);
                        for(l = 0; l < (Ndigit - 1); l++){
```

```
                                        for(k = l + 1; k < Ndigit; k++){
                                                if( store[k] > store[l] ) exchangeDigit(i,j,l, k);
                                        }
                                }
                        }
                }
        }

        void AllChrom::exchangeDigit(int a, int b, int c, int d){
                int temp[Nbit], temp2, i;

                for( i = 0; i < Nbit; i++) temp[i] = chrom[a].coef[b][c][i].bit;
                for( i = 0; i < Nbit; i++) chrom[a].coef[b][c][i].bit = chrom[a].coef[b][d][i].bit;
                for( i = 0; i < Nbit; i++) chrom[a].coef[b][d][i].bit = temp[i];

                temp2 = chrom[a].signbit[b][c].sbit;
                chrom[a].signbit[b][c].sbit = chrom[a].signbit[b][d].sbit;
                chrom[a].signbit[b][d].sbit = temp2;
        }

        int AllChrom::CSDcheck(int loc, int geneloc){
                int sum,i,j,numbplus,numbminu, rvalue, store[Ndigit];

                rvalue = 1;
                for(i = 0; i < Ndigit; i++) store[i] = decodeChrom(loc, geneloc, i);

                for(i=0; i < Ndigit; i++){ if ( store[i] >= MAX_WL ) goto JUMP; }

                for(i = 0; i < (Ndigit - 1); i++){
                        numbplus = store[i] + 1;
                        numbminu = store[i] - 1;
                        for ( j = i + 1; j < Ndigit; j++){
                                if ( (numbplus == store[j]) || ( numbminu == store[j]) ) goto JUMP;
                                if ( store[i] == store[j] ) goto JUMP;
                        }
                }

                rvalue = 0;

        JUMP:   return(rvalue);
        }

        void AllChrom::reloadGene(int loc1, int loc2, int coefloc){
                int i, k;

                for( k =0; k < Ndigit; k++){
                        for( i =0; i < Nbit; i++){
                                chrom[loc1].coef[coefloc][k][i].bit = offsprg[loc1].coef[coefloc][k][i].bit;
                        }
                        chrom[loc1].signbit[coefloc][k].sbit = offsprg[loc1].signbit[coefloc][k].sbit;
                }
                for( k =0; k < Ndigit; k++){
                        for( i =0; i < Nbit; i++){
                                chrom[loc2].coef[coefloc][k][i].bit = offsprg[loc2].coef[coefloc][k][i].bit;
                        }
                        chrom[loc2].signbit[coefloc][k].sbit = offsprg[loc2].signbit[coefloc][k].sbit;
                }
        }

        void AllChrom::preloadChrom(){
                int i;
```

```
            for( i =0; i < Nchrom; i++) chrom2offsprg(i , i);
 }


 void AllChrom::ArandChrom(int loc){                          //which chrom
            int i,j,k;
            int luckybit;

            for (j=0; j < Ncoef; j++){
                      for( k = 0; k < Ndigit; k++){
                                for (i=0; i < Nbit; i++) chrom[loc].coef[j][k][i].bit = randBit();
                                while( decodeChrom(loc, j, k) >= MAX_WL ){ for (i=0; i < Nbit; i++) chrom[loc].coef[j][k][i].bit
= randBit(); }
                      }
            }

            for ( j=0; j < Ncoef; j++){
                      for ( k =0; k < Ndigit; k++) chrom[loc].signbit[j][k].sbit = 0;
                      luckybit = ANUMBER() % Ndigit;
                      chrom[loc].signbit[j][luckybit].sbit = randBit3(0);
            }
}




//For offsprg
void AllChrom::ArandGene(int loc, int coef_loc){
            int i,k, sum;
            int luckybit;

            sum = 0;
            for ( i = 0; i < Ndigit; i++ ) sum =  sum + abs( chrom[loc].signbit[coef_loc][i].sbit );

            if ( sum == 0 ){
                      luckybit = ANUMBER() % Ndigit;
                      chrom[loc].signbit[coef_loc][luckybit].sbit = randBit3(0);
                      while( decodeChrom(loc, coef_loc, luckybit) >= MAX_WL ) {
                                for (i=0; i < Nbit; i++) chrom[loc].coef[coef_loc][luckybit][i].bit = randBit();
                      }
            }
            else{
                      for ( k =0; k < Ndigit; k++) chrom[loc].signbit[coef_loc][k].sbit = 0;
                      luckybit = ANUMBER() % Ndigit;
                      chrom[loc].signbit[coef_loc][luckybit].sbit = randBit3(0);
                      while( decodeChrom(loc, coef_loc, luckybit) >= MAX_WL ) {
                                for (i=0; i < Nbit; i++) chrom[loc].coef[coef_loc][luckybit][i].bit = randBit();
                      }
            }
}


//loc usually is 1
void AllChrom::GETbestChrom(){      //usually is 1 for index
            int j,i,loc,k;

            loc = -10;
            for( i =0 ; i < Nchrom; i++){
                      if ( bestChrom.fitness < chrom[i].fitness ) loc = i;
            }
            if ( loc >= 0 ){
                        bestChrom.fitness = chrom[loc].fitness;
                        for(j=0;j < Ncoef; j++){
                                    for(k=0; k < Ndigit; k++){
                                                for(i=0; i < Nbit; i++) bestChrom.coef[j][k][i].bit = chrom[loc].coef[j][k][i].bit;
                                                bestChrom.signbit[j][k].sbit = chrom[loc].signbit[j][k].sbit;
                                    }
                }
            }
}

void AllChrom::chrom2offsprg(int chrom_loc, int offsprg_loc){
            int i,j,k;

            for(j=0;j < Ncoef; j++){
                        for(k=0; k < Ndigit; k++){
                                    for(i=0; i < Nbit; i++)
                                                offsprg[offsprg_loc].coef[j][k][i].bit = chrom[chrom_loc].coef[j][k][i].bit;
                                    offsprg[offsprg_loc].signbit[j][k].sbit = chrom[chrom_loc].signbit[j][k].sbit;
                        }
            }
            offsprg[offsprg_loc].fitness = chrom[chrom_loc].fitness;
}

//include fitness
void AllChrom::offsprg2chrom(int offsprg_loc , int chrom_loc){
            int i,j,k;

            for(j=0;j < Ncoef; j++){
                        for(k=0; k < Ndigit; k++){
                                    for(i=0; i < Nbit; i++)
                                                chrom[chrom_loc].coef[j][k][i].bit = offsprg[offsprg_loc].coef[j][k][i].bit;
                                    chrom[chrom_loc].signbit[j][k].sbit = offsprg[offsprg_loc].signbit[j][k].sbit;
                        }
            }
            chrom[chrom_loc].fitness = offsprg[offsprg_loc].fitness;
}
```

STEPS.CPP

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include "allchrom.cpp"

class Steps : public AllChrom {
private:
            int Noffsprg, Nchrom, Ncoef;
            int *ordering;
            char ob;

public:
      Steps(int,double,double,char, char*, int);
      ~Steps(){ }
            void presetChrom();                                    //watch out for preload
      void reproduce();           //duplicate the chrom to offsprg
            void GETchromfit();   //passing decoded coefficients to fitness
            double fitness(double*);
            void CMCfix();
};
```

```
Steps::Steps(int Nchromosome, double pmutation, double pcrossover, char ob_ch, char *name, int
Ncoeff):AllChrom(Nchromosome, pmutation,pcrossover, name, Ncoeff){
            int i;

            Nchrom = Nchromosome;
            Noffsprg = Nchrom;
            Ncoef = Ncoeff;

            ordering = new int [Nchrom];
            if (ordering == NULL){
                        printf("\nno memory for ordering");
                        exit(-1);
            }

            ob = ob_ch;
            for ( i =0; i < Nchrom; i++) ordering[i] = i;
}

void Steps::presetChrom(){
            int j,i;

            for(j = 0 ; j < Nchrom; j++){
            ArandChrom(j);
            }

            //************check for the initial data **************
/*          for ( j=0; j < Nchrom; j++){
                        OUTchrom(j);
                        printf("\n");
            }
            scanf("%d",&i);
*/
}


//duplicate and set the Noffsprg
void Steps::reproduce(){
      int loc, i, j, *checked;
            double value;

            checked = new int[Nchrom];
            if ( checked == NULL){
                        printf("\nNot enough memory for *chrom");
                        exit(-1);
            }

            Noffsprg = ANUMBER() % ( Nchrom - 2 );
            Noffsprg = Noffsprg + 2;
            if ((Noffsprg % 2) != 0 ) Noffsprg++;

            for(i = 0; i < Nchrom; i++) checked[i] = 0;
            for(i = 0; i < Nchrom; i++){
                        value = 0;
                        for( j = 0; j < Nchrom; j++){
                                    if ( (checked[j] != 1) && ( value < OUTchromfitness(j) ) ){
                                                value = OUTchromfitness(j);
                                                loc = j;
                                    }
                        }
```

```
                ordering[i] = loc;
                checked[loc] = 1;
        }

        delete[] checked;
}


void Steps::CMCfix(){
        int coef_count,j;
        int flag1, flag2 , count;

        preloadChrom();

        for ( j = 0; j < Noffsprg; j = j + 2){
                for ( coef_count = 0; coef_count < Ncoef; coef_count++){

                        //check for CSD format
                        count = 0;
                        flag2 = 1;

                        while( (flag2) && (count < 5) ){
                                reloadGene(ordering[j], ordering[j + 1], coef_count);

                                DOcrossover(ordering[j], ordering[j + 1], coef_count);
                                DOmutation(ordering[j], coef_count);
                                DOmutation(ordering[j + 1], coef_count);

                                flag1 = CSDcheck(ordering[j],     coef_count);
                                if ( flag1 == 0) flag2 = CSDcheck(ordering[j + 1], coef_count);
                                count++;
                        }

                        //if all five chances fail
                        if ( (flag1 != 0) || (flag2 != 0) ){
                                flag2 = CSDcheck(ordering[j +1], coef_count);
                                if (flag1 != 0 ) ArandGene(ordering[j], coef_count);
                                if (flag2 != 0 ) ArandGene(ordering[j +1], coef_count);
                        }

                        //check for zero coefficients
                        if ( decodeChrom(ordering[j], coef_count) == 0.0 )  ArandGene(ordering[j], coef_count);
                        if ( decodeChrom(ordering[j + 1], coef_count) == 0.0 ) ArandGene(ordering[j + 1], coef_count);
                }
        }

        //refill the rest of the extra space
        coef_count = 0;
        for ( j = Noffsprg; j < Nchrom; j++){
                offsprg2chrom( ordering[coef_count], ordering[j]);
                coef_count++;
        }
}

void Steps::GETchromfit(){   //passing decoded coefficients to fitness
        int j,i;
        double *coeff, val;

        coeff = new double[Ncoef];
        if ( coeff == NULL){
                printf("\nNot enough memory for *coeff");
                exit(-1);
        }

        for( j = 0; j < Noffsprg; j++){
                for( i=0; i < Ncoef; i++){
                coeff[i] = decodeChrom(ordering[j],i);
                }

                val = fitness(coeff);
                INchromfitness(ordering[j],val);
        }
        delete[] coeff;
}

double Steps::fitness(double *coeff){
        int i,j, flag , count;
        double err,w, delta_w, magH;
        double H, anumber, temp;
        double sum, return_val, max_err;

    delta_w = maxFreq / Npoints;

    err = 0;
        max_err = 0;
        for (w = 0 ; w < maxFreq + delta_w; w = w + delta_w){

                //at one frequency of the first half symmetry coefficients
        count = 0;


                H = coeff[Ncoef - 1];
                for ( j = 1; j < Ncoef; j++){
                        H = H + 2.0 * coeff[Ncoef - j - 1] * cos(w * T * j);
                }

                H = fabs(H);


                //minmax or LMS
                if ( ob == 'l'){

                        //exp decay transision region
                        if ( w < passbd )
                                err = err + fabs( 1.0 - H);
                        else
                                err = err + fabs( exp(-1.0*ALPHA*(w - passbd)) - H );

                } else if(ob == 'm') {

                        //get the magnitude response error
                        if ( w < passbd + delta_w)  err = fabs( 1.0 - H );
                        else if ( w > stopbd - delta_w ) err = fabs(H);
                        else { }
                        if( max_err < err ) max_err = err;

                } else {
                        printf("\ncannot get the objective function\n");
                        exit(-1);
                }

        }
```

```
            if (ob == 'l') return_val  =  1.0 / err;
            else return_val  =  1.0 / max_err;

            return( return_val );

    }


    //***************************************************************************

    class final:public Steps {
    private:
                int MaxIter;

    public:
           final(int,int,double,double,char,char*, int);
                ~final() { }
                void process();
    };

    final::final(int MaxIteration, int chrom_size, double mut_prob, double cross_prob, char ob_ch, char *name, int
    Ncoeff):Steps(chrom_size,mut_prob, cross_prob, ob_ch, name, Ncoeff){
                MaxIter = MaxIteration;
    }


    void final::process(){
                int Citer;

          //get the random chromosome
                presetChrom();

                //ordering the digits
                orderDigits();

                //calcuate the fitness of each chromosome
                GETchromfit();

                Citer = 0;
                while ( Citer < MaxIter ){
                        Citer++;

                   //find Noffsprg and duplicate the chrom to offsprg
                                reproduce();

                                //crossover, mutation, and CSD checkNfix
                                CMCfix();

                                //ordering the digits
                                orderDigits();

                                //calcuate fitness of each offsprg
                                GETchromfit();

                   //store the best chromosome comparing to the history
                                GETbestChrom();

                                printf("\n%d          %f", Citer, OUTbestfitness() );

               }
```

```
        printf("\n");
        report();
}


main(int argsize, char **arg){
        int popsize, iterations;
        double mutation, crossover;
        int Ncoeff;

//      arg[1] - m for minimax or l for LMS
//      arg[6] - output filename

        if ( argsize == 8 ){
                iterations = atoi(arg[2]);
                popsize    = atoi(arg[3]);
                mutation   = ((double) atoi(arg[4])) / 100.0;
                crossover  = ((double) atoi(arg[5])) / 100.0;
                Ncoeff     = atoi(arg[6]);

//              printf("\niteration is %d popsize is %d ", iterations, popsize );
//              printf("mutation is %f crossover is %f\n", mutation, crossover );
//              printf("The order of the filter is %d", Ncoeff - 1);

                final object( iterations, popsize, mutation, crossover, *arg[1], arg[7], Ncoeff);
                object.process();
        }
        else printf("\nwrong argument");
}
```

Section 5

This section shows the source code for DBNS filter design with screening process.

The GA program (MAIN PROGRAM) is written in a PC machine. Therefore, the executable can only run in DOS under window environment. Its execution command is as follow:

```
C:> GA4 <ob> <N> <Nchrom> <pmut> <pcross> <Ncoef> <delta_error>
```

where

<ob> is the choice of objective function, "m" for minimax error or "l" for lease mean square error.

<N> is the maximum number of iterations, must be an integer number.

<Nchrom> is the population size, must be an integer number.

<pmut> is the mutation rate that should be kept between 0.01 to 0.2.

<pcross> is the crossover rate that should be kept between 0.5 to 0.95.

<Ncoef> filter order is determined as 2*(Ncoef - 1), Ncoef must be an integer number.

<delta_error> is determined as the passband ripple and stopband losses, eg. delta_error = 0.05.

If any key on the keyboard is touched during execution, this program will terminate and report the coefficients into an file named "output.dat". The format of the output data file is given as "signbit# binary Exponent ternary Exponent", and followed by a list of coefficients in continuous value. A "n200.dat" file is generated using the 'screening.cpp' program as input data that should be kept in the same directory as the exec. program. This input data is generated by determined all the available DBNS numbers between zero and one with the restriction of maximum ternary exponent, then the "screening process" carries out to remove the redundant data.

SCREENING.CPP

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define DUMMY 131072

typedef struct {
        double numb;
        int  bexp;
        int  texp;
} DBNStype;

DBNStype data[DUMMY];
FILE *ptr;

void addlist(int b, int t, int N){
        data[N].numb = exp( b * log(2.0) + t * log(3.0) );
        data[N].bexp = b;
        data[N].texp = t;
}

void sortlist(int N){
        double a;
        int b,t, i,j;

        for(i = 0; i < (N - 1); i++){
                for(j = i + 1; j < N; j++){
                        if( data[i].numb > data[j].numb ){
                                a = data[i].numb;
                                b = data[i].bexp;
                                t = data[i].texp;
                                data[i].numb = data[j].numb;
                                data[i].bexp = data[j].bexp;
                                data[i].texp = data[j].texp;
                                data[j].numb = a;
                                data[j].bexp = b;
                                data[j].texp = t;
                        }
                }
        }
}

void cutdum(int N){
        int flag = 1;
        int i,j;

        while(flag){
                flag = 0;
                for(i = 0; i < (N - 1); i++){
                        if( (data[i].texp != 3) && ( data[i].bexp != 3)){
                        for( j = i + 1; j < N; j++){

                                if( (data[j].texp != 3) && ( data[j].bexp != 3)){
                                        if( fabs(data[i].numb - data[j].numb) < 1.0E-9 ){
                                                if( data[i].texp < data[j].texp ) {
                                                        data[j].texp = 3;
                                                        data[j].bexp = 3;
                                                }
                                                else {
```

```
                                                            data[i].texp = 3;
                                                            data[i].texp = 3;
                                                    }
                                                    flag = 1;
                                            }
                                    }

                            }

                            }

                    }
            }
    }

    main(int argsize, char **arg){
            int TExp, BExp, t, b, count;
            int upbound, lowbound, gg, k;
            double temp;

            if ( argsize != 3 ) exit(-1);
            TExp = atoi(arg[1]);
            BExp = (int)( (1.0 + TExp * log(3.0)) / log(2.0)) + 1;


            upbound = 0;
            lowbound = -20.0;



            count = 0;
            for(t = (-1 * TExp); t <= TExp; t++){
                    for(b = (-1 * BExp); b <= BExp; b++){
                            if ( (abs(b) + abs(t)) != 0 ){
                                    temp = b * log(2.0) + t * log(3.0);
                                    if ((temp < upbound) && (temp > lowbound)){
                                            addlist(b,t,count);
                                            count++;
                                    }
                            }
                            else
                            if ( count > (DUMMY - 5) ){
                                    printf("\nOver the storage\n");
                                    exit(-1);
                            }
                    }
            }
            count++;

            sortlist(count);
            cutdum(count);

            ptr = fopen(arg[2],"w+");
            if ( ptr == NULL ) {
                    printf("\nCannot open file\n");
                    exit(-1);
            }

            gg = 1;
            for(k=0; k < count; k++){
                    if ((data[k].bexp == 0) && (data[k].texp == 0) )
                    fprintf(ptr,"%15.11f %6d %6d\n", 1.0, 0, 0);
                else if ( (data[k].bexp != 3) && (data[k].texp != 3) ){
                    fprintf(ptr,"%15.11f %6d %6d\n", data[k].numb, data[k].bexp, data[k].texp);
                    gg++;
                }
                else { }
        }
        printf("\nTotal number of data is %d\n",gg);

        free(data);
        fclose(ptr);
}
```

MAIN PROGRAM

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<string.h>
//#include<conio.h>
#include<time.h>

#define T              1.0                  //sampling frequency
#define maxFreq        3.1416
#define Npoints        47                   //# of sampling points
#define stopbd         1.8                  //either stopbd or ALPHA is used
#define ALPHA          5.0
#define passbd         1.2
#define   maxFilterOrder 50

/********************/
/* define data type */
/********************/
typedef struct {
        unsigned int bit:1;
} aBIT;

typedef struct {
        aBIT coef[maxFilterOrder][14];      //the last bit is the sign bit for double base
        float fitness;
        int             absfitness;
} aCHROM;

typedef struct {
        double  val;
        int             binExp;
        int             terExp;
        aBIT     sign;
} DATA;

/********************/
/* Global Variables */
/********************/
float    pmut, pcross, errZone;
DATA     *coeff;
int             Ncoef, Nchrom;
int             *ordering;
FILE     *inptr, *outptr;
DATA     *data;
```

```
 aCHROM          *chrom, *offsprg,Bchrom;
 char     ob;

 /***************************/
 /* Procedure and function */
 /***************************/

 int ANUMBER(){
         return( rand() );
 }

 float UNI(){
                 return ( (float)rand() / (float)RAND_MAX );
 }

 void presetChrom(){
         int i,j,k1,k2,k3,k4,k;

         for (i = 0; i < Nchrom; i++){
                 for (j = 0; j < Ncoef; j++){
                         for (k = 0; k < 14;k++) chrom[i].coef[j][k].bit = 0;
                 }
         }

         for (i = 0; i < Nchrom; i++){
                 for (j = 0; j < Ncoef; j++){
                         for (k = 0; k < 14;k++) offsprg[i].coef[j][k].bit = 0;
                 }
         }

         for (i = 0; i < Nchrom; i++){
                 for (j = 0; j < Ncoef; j++){
                         k1 = ANUMBER() % 14;
                         chrom[i].coef[j][k1].bit = 1;

                         k2 = ANUMBER() % 14;
                         chrom[i].coef[j][k2].bit = 1;

                         k3 = ANUMBER() % 14;
                         chrom[i].coef[j][k3].bit = 1;

                         k4 = ANUMBER() % 14;
                         chrom[i].coef[j][k4].bit = 1;
                 }
         }

}

void InsertBestChrom(int maxi, float maxFit, int mini, float minFit){
        int i, j;

        if ( maxFit > Bchrom.fitness ){
                for (j=0; j < Ncoef; j++){
                        for (i=0; i < 14; i++) {
                                if (Bchrom.coef[j][i].bit != chrom[maxi].coef[j][i].bit){
                                        Bchrom.coef[j][i].bit = !Bchrom.coef[j][i].bit;
                                }
                        }
                }
                Bchrom.fitness = maxFit;
        } else {
```

```
                    for (j=0; j < Ncoef; j++){
                              for (i=0; i < 14; i++) {
                                        if (Bchrom.coef[j][i].bit != chrom[mini].coef[j][i].bit){
                                                  chrom[mini].coef[j][i].bit = !chrom[mini].coef[j][i].bit;
                                        }
                              }
                    }
                    chrom[mini].fitness = Bchrom.fitness;
          }
}

void scaleFitness(){
          int i,indexMax,indexMin;
          double minN, maxN;
          static int flag = 0;

          //Find maximum and minimum
          minN = 999.9;
          maxN = -10.0;
          for(i = 0; i < Nchrom; i++){
                    if( minN > chrom[i].fitness ){ minN = chrom[i].fitness; indexMin = i;}
                    if( maxN < chrom[i].fitness ){ maxN = chrom[i].fitness; indexMax = i;}
          }

          if ( flag != 0 )
                    InsertBestChrom(indexMax,maxN,indexMin,minN);
          else flag++;

          //Perform scaling
          for( i = 0; i < Nchrom; i++)
                    chrom[i].absfitness = (int) ceil( 80.0 * (chrom[i].fitness - minN)/(maxN - minN) ) + 5;
}


void DOcrossover(int chrom1_loc, int chrom2_loc, int offsprg1_loc, int offsprg2_loc, int coef_loc){
          int apoint, i;

          if ( UNI() < pcross ){
                    apoint = ANUMBER() % (14 - 1);
                    apoint++;

                    for ( i = 0; i < apoint ; i++){
                              if (chrom[chrom1_loc].coef[coef_loc][i].bit != offsprg[offsprg1_loc].coef[coef_loc][i].bit){
                                        offsprg[offsprg1_loc].coef[coef_loc][i].bit = chrom[chrom1_loc].coef[coef_loc][i].bit;
                              }
                              if (chrom[chrom2_loc].coef[coef_loc][i].bit != offsprg[offsprg2_loc].coef[coef_loc][i].bit){
                                        offsprg[offsprg2_loc].coef[coef_loc][i].bit = chrom[chrom2_loc].coef[coef_loc][i].bit;
                              }
                    }

                    for ( i = apoint; i < 14 ; i++){
                              if (chrom[chrom1_loc].coef[coef_loc][i].bit != offsprg[offsprg2_loc].coef[coef_loc][i].bit){
                                        offsprg[offsprg2_loc].coef[coef_loc][i].bit = chrom[chrom1_loc].coef[coef_loc][i].bit;
                              }
                              if (chrom[chrom2_loc].coef[coef_loc][i].bit != offsprg[offsprg1_loc].coef[coef_loc][i].bit){
                                        offsprg[offsprg1_loc].coef[coef_loc][i].bit = chrom[chrom2_loc].coef[coef_loc][i].bit;
                              }
                    }
          }
}
```

```
void DOmutation(int loc){
            int i,j;

            for (i=0; i < Ncoef; i++){
                        for (j = 0; j < 14; j++){
                  if ( UNI() < pmut )
                                                      offsprg[loc].coef[i][j].bit = !offsprg[loc].coef[i][j].bit;
                        }
            }
}

void reproduce(){
   int i, count,j;
            float rawfitness, total, sum;

            total = -0.5;
            for( j = 0; j < Nchrom; j++) total = total + (float)chrom[j].absfitness;

            for( i = 0; i < Nchrom; i++){

                        rawfitness = UNI() * total;
                        j = ANUMBER() % Nchrom;
                        sum = (float)chrom[j].absfitness;
                        count = j;
                        while(sum < rawfitness){
                                    count++;
                                    if (count == Nchrom) count = 0;
                                    sum = sum + (float)chrom[count].absfitness;
                        }
                        ordering[i] = count;
            }
/*
            for(i=0;i<Nchrom;i++) printf("%d ",chrom[i].absfitness);
            printf("\n");
            for(i=0;i<Nchrom;i++) printf("%d ",ordering[i]);
            printf("\n");
            getch();
*/
}

//decodeChrom
int decodeChrom(int chrom_loc, int coef_loc){
            int  temp;

            temp = 0;
            if (chrom[chrom_loc].coef[coef_loc][0].bit != 0) temp = temp + 1;
            if (chrom[chrom_loc].coef[coef_loc][1].bit != 0) temp = temp + 2;
            if (chrom[chrom_loc].coef[coef_loc][2].bit != 0)temp = temp + 4;
            if (chrom[chrom_loc].coef[coef_loc][3].bit != 0)temp = temp + 8;
            if (chrom[chrom_loc].coef[coef_loc][4].bit != 0)temp = temp + 16;
            if (chrom[chrom_loc].coef[coef_loc][5].bit != 0)temp = temp + 32;
            if (chrom[chrom_loc].coef[coef_loc][6].bit != 0)temp = temp + 64;
            if (chrom[chrom_loc].coef[coef_loc][7].bit != 0)temp = temp + 128;
            if (chrom[chrom_loc].coef[coef_loc][8].bit != 0)temp = temp + 256;
            if (chrom[chrom_loc].coef[coef_loc][9].bit != 0)temp = temp + 512;
            if (chrom[chrom_loc].coef[coef_loc][10].bit != 0)temp = temp + 1024;
            if (chrom[chrom_loc].coef[coef_loc][11].bit != 0)temp = temp + 2048;
            if (chrom[chrom_loc].coef[coef_loc][12].bit != 0)temp = temp + 4096;

            if (chrom[chrom_loc].coef[coef_loc][13].bit == 1)
```

```
                    return( -temp );
            else return(temp);
}



float fitness(){
            int j, count;
            float err,w, delta_w;
            double H, temp;
            float return_val, max_err;

   delta_w = (float)maxFreq / (float)Npoints;

   err = 0;
            max_err = 0;
            for (w = 0 ; w < maxFreq + delta_w; w = w + delta_w){

                        //at one frequency of the first half symmetry coefficients
      count = 0;

                        H = coeff[Ncoef - 1].val;
                        for ( j = 1; j < Ncoef; j++){
                                    H = H + 2.0 * coeff[Ncoef - j - 1].val * cos(w * T * j);
                        }

                        H = fabs(H);


                        //minmax or LMS
                        if ( ob == 'l'){

                                    //exp decay transision region
                                    if ( w < passbd )
                                                temp = fabs( 1.0 - H);
                                    else
                                                temp = fabs( exp(-1.0*ALPHA*(w - passbd)) - H );

                                    if (temp > errZone) err = err + temp - errZone;

                        } else if(ob == 'm') {

                                    //get the magnitude response error
                                    if ( w < passbd )
                                                err = fabs( 1.0 - H );
                                    else if ( w > stopbd  )
                                                err = fabs(H);
                                    else { }

                                    if (err < errZone ) err = 0;
                                    else err = err - errZone;

                                    if( max_err < err ) max_err = err;

                        } else {
                                    printf("\ncannot get the objective function\n");
                                    exit(-1);
                        }

            }
            if (ob == '1') return_val  =  1.0 / err;
            else return_val  =  1.0 / max_err;

            return( return_val );

    }

    void GETchromfit(){    //passing decoded coefficients to fitness
            int j,i, loc;

            for( j = 0; j < Nchrom; j++){
                    for( i=0; i < Ncoef; i++){
                            loc = decodeChrom(j,i);
                            if (loc < 0){
                                    coeff[i].val = - data[-loc].val;
                                    coeff[i].binExp = data[-loc].binExp;
                                    coeff[i].terExp = data[-loc].terExp;
                                    coeff[i].sign.bit = 1;
                            }else {
                                    coeff[i].val = data[loc].val;
                                    coeff[i].binExp = data[loc].binExp;
                                    coeff[i].terExp = data[loc].terExp;
                                    coeff[i].sign.bit = 0;
                            }

                    }
                    chrom[j].fitness = fitness();
      }
    }

int decodeBChrom(int coef_loc){
            int  temp;

            temp = 0;
            if (Bchrom.coef[coef_loc][0].bit != 0) temp = temp + 1;
            if (Bchrom.coef[coef_loc][1].bit != 0) temp = temp + 2;
            if (Bchrom.coef[coef_loc][2].bit != 0)temp = temp + 4;
            if (Bchrom.coef[coef_loc][3].bit != 0)temp = temp + 8;
            if (Bchrom.coef[coef_loc][4].bit != 0)temp = temp + 16;
            if (Bchrom.coef[coef_loc][5].bit != 0)temp = temp + 32;
            if (Bchrom.coef[coef_loc][6].bit != 0)temp = temp + 64;
            if (Bchrom.coef[coef_loc][7].bit != 0)temp = temp + 128;
            if (Bchrom.coef[coef_loc][8].bit != 0)temp = temp + 256;
            if (Bchrom.coef[coef_loc][9].bit != 0)temp = temp + 512;
            if (Bchrom.coef[coef_loc][10].bit != 0)temp = temp + 1024;
            if (Bchrom.coef[coef_loc][11].bit != 0)temp = temp + 2048;
            if (Bchrom.coef[coef_loc][12].bit != 0)temp = temp + 4096;

            if (Bchrom.coef[coef_loc][13].bit == 1)
                    return( -temp );

            return(temp);
}

void report(){
            int i,j,sign, temp;
            float maxfitness;

            /* open output file */
            outptr = fopen( "output.dat","w+");
            if ( outptr == NULL){ printf("cannot open output file"); exit(-1); }
            /* decode the coefficient */
            for( i=0; i < Ncoef; i++){
                        temp = decodeBChrom(i);
                        if (temp < 0){
                                    coeff[i].val = - data[-temp].val;
                                    coeff[i].binExp = data[-temp].binExp;
                                    coeff[i].terExp = data[-temp].terExp;
                                    coeff[i].sign.bit = 1;
                        }else {
                                    coeff[i].val = data[temp].val;
                                    coeff[i].binExp = data[temp].binExp;
                                    coeff[i].terExp = data[temp].terExp;
                                    coeff[i].sign.bit = 0;
                        }
                        Bchrom.fitness = fitness();
            }

            /* printout the coefficient and exponents values */
            for(i = 0; i < Ncoef; i++){
                        if (coeff[i].sign.bit != 0) sign = -1;
                        else sign = 1;
                        fprintf(outptr, "%5d# %8d %8d",sign, coeff[i].binExp, coeff[i].terExp);
                        fprintf(outptr, "\n");
            }
            for(i = 1; i < Ncoef; i++){
//                      if ( bestChrom.coefB2[Ncoef - i - 1][NbitB2 - 1].bit != 0 ) sign = -1;
//                      else sign = 1;
                        if (coeff[Ncoef - i - 1].sign.bit != 0) sign = -1;
                        else sign = 1;
                        fprintf(outptr, "%5d# %8d %8d",sign, coeff[Ncoef - i - 1].binExp, coeff[Ncoef - i - 1].terExp);
                        fprintf(outptr,"\n");
            }

   fprintf(outptr,"\n");
            for(i = 0; i < Ncoef; i++){
                        fprintf(outptr,"\n%18.15f",coeff[i].val);

            }
   for ( j = 1; j < Ncoef; j++){
                        fprintf(outptr,"\n%18.15f",coeff[Ncoef - j - 1].val);
   }

            fprintf(outptr, "\n\nThe best fitness is %f", Bchrom.fitness);
            fclose(outptr);
}

/**********************
** Main Program *******
**********************/
void main(int argsize, char **arg){
            int Citer, chromctr, coefctr, MaxIter,index1,index2;
            aCHROM *temp;

            if ( argsize != 8 ){ printf("\nwrong number of arguments!!"); exit(-1);}

            ob                          = *arg[1];
            MaxIter             = atoi(arg[2]);
            Nchrom              = atoi(arg[3]);
            pmut                = (float)atof(arg[4]);
            pcross              = (float)atof(arg[5]);
```

```
Ncoef              = atoi(arg[6]);
errZone            = (float)atof(arg[7]);

srand( (unsigned)time( NULL ) );

/* restriction on the population size */
if((Nchrom % 2) != 0 ) Nchrom--;

/* create space for chromosome */
chrom = new aCHROM[Nchrom];
if (chrom == NULL) {printf("don't have enough space, chrom\n"); exit(-1);}

/* create space for offsprg */
offsprg = new aCHROM[Nchrom];
if (offsprg == NULL) {printf("don't have enough space, offsprg\n"); exit(-1);}

/* create space for ordering */
ordering = new int[Nchrom];
if (ordering == NULL) {printf("don't have enough space, ordering\n"); exit(-1);}

/* create space for coeff for fitness evaluation use */
coeff = new DATA[Ncoef];
if ( coeff == NULL){ printf("\nNot enough memory for *coeff"); exit(-1); }

/* read all available data */
inptr = fopen( "n200.dat" ,"r");
if (inptr == NULL) { printf("cannot open readin file/n"); exit(-1); }
data = new DATA[8192];

if (data == NULL) {printf("don't have enough space, data\n"); exit(-1);}

for (int i=0; i < 8192; i++)
        fscanf(inptr,"%lf %d %d",&data[i].val,&data[i].binExp,&data[i].terExp);
fclose(inptr);

presetChrom();

//calcuate the fitness of each chromosome
GETchromfit();

Bchrom.fitness = -1.0;
Bchrom.absfitness = -1;
Citer = 0;
scaleFitness();
while ( (Citer < MaxIter) && !_kbhit() ){
        Citer++;

        reproduce();

        //crossover, mutation
        for (chromctr = 0; chromctr < Nchrom; chromctr = chromctr + 2){
                index1 = ordering[chromctr];
                index2 = ordering[chromctr + 1];
                for (coefctr = 0; coefctr < Ncoef; coefctr++)
                        DOcrossover(index1, index2, chromctr, chromctr + 1, coefctr);
                DOmutation(chromctr);
                DOmutation(chromctr + 1);
        }

        /* exchange pointer */
        temp = chrom;
```

```
                chrom = offsprg;
                offsprg = temp;

                //calcuate fitness of each offsprg
                GETchromfit();
                scaleFitness();
                if ((Citer % 30) == 0 ) printf("\n%d       %f", Citer, Bchrom.fitness );
        }

        if (_kbhit()) _getch();
        printf("\n");
        report();

        delete[] chrom;
        delete[] data;
        delete[] coeff;
        delete[] offsprg;
}
```

Section 6

This section shows the source code for 2-D IIR filter design using Genetic Algorithm.

MAIN PROGRAM

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<conio.h>
#include<string.h>
#include<time.h>

//********************************
//control room
//********************************
#define T                   1.0                 //sampling frequency
#define maxfreq             3.14159265358979
#define  maxFilterOrder 64
#define MAX_WORD            14                  //38, 14
#define alpha_tran 3.0
#define   Nstep1    120      //62
#define   Nstep2    120      //62
#define   passband  0.8
#define stopband   2.2
//********************************

/********************/
/* define data type */
/********************/
typedef struct {
        float coef[maxFilterOrder];
        float fitness;
        int             absfitness;
} aCHROM;


/////////////////// define complex number and class //////////
typedef struct{
                float re,im;
} cnumber;

//////////////////////////////////////////////////////////////


/********************/
/* Global Variables */
/********************/
float     pmut, pcross;
int                 Ncoef, Nchrom, nvars;
int                 *ordering;
FILE      *inptr, *outptr;
aCHROM              *chrom, *offsprg,Bchrom;
float  coef[33];  //used by the fitness evaluation
float     domain[MAX_WORD];
```

```
cnumber cadd(cnumber a,cnumber b){
            cnumber c;

            c.re = a.re + b.re;
            c.im = a.im + b.im;
      return(c);
}

cnumber cadd3c1r(cnumber a,cnumber b,cnumber c,float anumber){
            cnumber e;

            e.re = a.re + b.re + c.re  + anumber;
            e.im = a.im + b.im + c.im;
      return(e);
}

cnumber csub(cnumber a,cnumber b){
            cnumber c;

            c.re = a.re - b.re;
            c.im = a.im - b.im;
      return(c);
}

cnumber cmulti(cnumber a, cnumber b){
            cnumber c;

            c.re = a.re * b.re - a.im * b.im;
            c.im = a.re * b.im + a.im * b.re;
            return(c);
}

cnumber cmulti1r2c(float anumber, cnumber a, cnumber b){
            cnumber c;

   c = cmulti(a,b);
            c.re = c.re * anumber;
            c.im = c.im * anumber;
            return(c);
}

cnumber cmulti1r1c(float anumber, cnumber a){
            cnumber c;

            c.re = a.re * anumber;
            c.im = a.im * anumber;
            return(c);
}

cnumber cdivid(cnumber a, cnumber b){
            cnumber c;
            float scalar;

            scalar = (b.re)*(b.re) + (b.im)*(b.im);
            if (scalar == 0) scalar = 1.0E-7;
            c.re = (a.re * b.re + a.im * b.im)/scalar;
            c.im = (a.im * b.re - a.re * b.im)/scalar;
            return(c);
}
```

```
 cnumber cexpj(float a){
              cnumber c;

              c.re = cos(a);
              c.im = sin(a);
       return(c);
 }

 float cmag(cnumber a){
              float c;

              c = (a.re)*(a.re) + (a.im)*(a.im);
              c = sqrt(c);
              return(c);
 }

 /**************************/
 /* Procedure and function */
 /**************************/

 int ANUMBER(){
              return( rand() );
 }

 float UNI(){
                            return ( (float)rand() / (float)RAND_MAX );
 }

 void presetChrom(){
              short i,j, temp;

              for (i = 0; i < Nchrom; i++){
                            for (j = 0; j < Ncoef; j++)
                                          offsprg[i].coef[j] = domain[0];
              }

              for (i = 0; i < Nchrom; i++){
                            for (j = 0; j < Ncoef; j++){
                                          temp = ANUMBER() % MAX_WORD;
                                          chrom[i].coef[j] = domain[temp];
                            }
              }
}

void InsertBestChrom(int maxi, float maxFit, int mini, float minFit){
              int j;

              if ( maxFit > Bchrom.fitness ){
                            for (j=0; j < Ncoef; j++){
                                          Bchrom.coef[j]  =  chrom[maxi].coef[j];
                            }
                            Bchrom.fitness = maxFit;
              } else {
                            for (j=0; j < Ncoef; j++){
                                          chrom[mini].coef[j] = Bchrom.coef[j];
                            }
                            chrom[mini].fitness = Bchrom.fitness;
              }
}

void scaleFitness(){
	int i,indexMax,indexMin;
	float minN, maxN;
	static int flag = 0;

	//Find maximum and minimum
	minN = 999.9;
	maxN = -10.0;
	for(i = 0; i < Nchrom; i++){
		if( minN > chrom[i].fitness ){ minN = chrom[i].fitness; indexMin = i;}
		if( maxN < chrom[i].fitness ){ maxN = chrom[i].fitness; indexMax = i;}
	}
	if ( flag != 0 )
		InsertBestChrom(indexMax,maxN,indexMin,minN);
	else flag++;

	//Perform scaling
	for( i = 0; i < Nchrom; i++){
		chrom[i].absfitness = (int) ceil( 20.0 * (chrom[i].fitness - minN)/(maxN - minN) ) + 1;
	}
}


void DOcrossover(int chrom1_loc, int chrom2_loc, int offsprg1_loc, int offsprg2_loc, int coef_loc){
	int apoint, i;

	if ( UNI() < pcross ){
		apoint = ANUMBER() % (Ncoef - 1);
		apoint++;

		for ( i = 0; i < apoint ; i++){
			if (chrom[chrom1_loc].coef[i] != offsprg[offsprg1_loc].coef[i]){
				offsprg[offsprg1_loc].coef[i] = chrom[chrom1_loc].coef[i];
			}
			if (chrom[chrom2_loc].coef[i] != offsprg[offsprg2_loc].coef[i]){
				offsprg[offsprg2_loc].coef[i] = chrom[chrom2_loc].coef[i];
			}
		}

		for ( i = apoint; i < Ncoef ; i++){
			if (chrom[chrom1_loc].coef[i] != offsprg[offsprg2_loc].coef[i]){
				offsprg[offsprg2_loc].coef[i] = chrom[chrom1_loc].coef[i];
			}
			if (chrom[chrom2_loc].coef[i] != offsprg[offsprg1_loc].coef[i]){
				offsprg[offsprg1_loc].coef[i] = chrom[chrom2_loc].coef[i];
			}
		}
	}
}


void DOmutation(int loc){
	short i,temp;

	for (i=0; i < Ncoef; i++){
		if ( UNI() < pmut ){
			temp = ANUMBER() % MAX_WORD;
			while (domain[temp] == offsprg[loc].coef[i]){
				temp = ANUMBER() % MAX_WORD;
			}
			offsprg[loc].coef[i] = domain[temp];
		}
```

```
            }
    }


    void reproduce(){
      int i, count,j;
            float rawfitness, total, sum;

            total = -0.5;
            for( j = 0; j < Nchrom; j++) total = total + (float)chrom[j].absfitness;

            for( i = 0; i < Nchrom; i++){

                    rawfitness = UNI() * total;
                    j = ANUMBER() % Nchrom;
                    sum = (float)chrom[j].absfitness;
                    count = j;
                    while(sum < rawfitness){
                            count++;
                            if (count == Nchrom) count = 0;
                            sum = sum + (float)chrom[count].absfitness;
                    }
                    ordering[i] = count;
            }
    }

    float fitness(int j)
    {
            int n = nvars,i;
            float  A1a, A1b, A1gamma, A1delta, A1r,        A1roll_1, A1roll_2, A1g;
            float  A2a, A2b, A2gamma, A2delta, A2r,        A2roll_1, A2roll_2, A2g;
            float  B1a, B1b, B1gamma, B1delta, B1r,        B1roll_1, B1roll_2, B1g;
            float  B2a, B2b, B2gamma, B2delta, B2r,        B2roll_1, B2roll_2, B2g;
            float  C1a, C1b, C1gamma, C1delta, C1r,        C1roll_1, C1roll_2, C1g;
            float  C2a, C2b, C2gamma, C2delta, C2r,        C2roll_1, C2roll_2, C2g;
            float  D1a, D1b, D1gamma, D1delta, D1r,        D1roll_1, D1roll_2, D1g;
            float  D2a, D2b, D2gamma, D2delta, D2r,        D2roll_1, D2roll_2, D2g;

      float  A1coef1, A1coef2, A1coef3, A1coef4, A2coef1, A2coef2, A2coef3, A2coef4;
      float  B1coef1, B1coef2, B1coef3, B1coef4, B2coef1, B2coef2, B2coef3, B2coef4;
      float  C1coef1, C1coef2, C1coef3, C1coef4, C2coef1, C2coef2, C2coef3, C2coef4;
      float  D1coef1, D1coef2, D1coef3, D1coef4, D2coef1, D2coef2, D2coef3, D2coef4;
      float  dum;

            float w1,w2, max_err, mag, temperr;
            float stepsize1,stepsize2,err,region;

            cnumber A,B,C,D,A1,A2,B1,B2,C1,C2,D1,D2,H;
            cnumber z1,z2,pure1, tempz1,tempz2,tempApluszlB,tempCplusz2D;
            cnumber temp1,temp2,temp3,temp5,temp6,temp7,tempnum,tempdenum;

            A1a=chrom[j].coef[0];
            A1b=chrom[j].coef[1];
            A1gamma=chrom[j].coef[2];
            A1delta=chrom[j].coef[3];
            A1r=chrom[j].coef[4];
            A1roll_1=chrom[j].coef[5];
            A1roll_2=chrom[j].coef[6];
            A1g=chrom[j].coef[7];

            A2a=chrom[j].coef[8];
```

```
A2b=chrom[j].coef[9];
A2gamma=chrom[j].coef[10];
A2delta=chrom[j].coef[11];
A2r=chrom[j].coef[12];
A2roll_1=chrom[j].coef[13];
A2roll_2=chrom[j].coef[14];
A2g=chrom[j].coef[15];

B1a=chrom[j].coef[16];
B1b=chrom[j].coef[17];
B1gamma=chrom[j].coef[18];
B1delta=chrom[j].coef[19];
B1r=chrom[j].coef[20];
B1roll_1=chrom[j].coef[21];
B1roll_2=chrom[j].coef[22];
B1g=chrom[j].coef[23];

B2a=chrom[j].coef[24];
B2b=chrom[j].coef[25];
B2gamma=chrom[j].coef[26];
B2delta=chrom[j].coef[27];
B2r=chrom[j].coef[28];
B2roll_1=chrom[j].coef[29];
B2roll_2=chrom[j].coef[30];
B2g=chrom[j].coef[31];

C1a=chrom[j].coef[32];
C1b=chrom[j].coef[33];
C1gamma=chrom[j].coef[34];
C1delta=chrom[j].coef[35];
C1r=chrom[j].coef[36];
C1roll_1=chrom[j].coef[37];
C1roll_2=chrom[j].coef[38];
C1g=chrom[j].coef[39];

C2a=chrom[j].coef[40];
C2b=chrom[j].coef[41];
C2gamma=chrom[j].coef[42];
C2delta=chrom[j].coef[43];
C2r=chrom[j].coef[44];
C2roll_1=chrom[j].coef[45];
C2roll_2=chrom[j].coef[46];
C2g=chrom[j].coef[47];

D1a=chrom[j].coef[48];
D1b=chrom[j].coef[49];
D1gamma=chrom[j].coef[50];
D1delta=chrom[j].coef[51];
D1r=chrom[j].coef[52];
D1roll_1=chrom[j].coef[53];
D1roll_2=chrom[j].coef[54];
D1g=chrom[j].coef[55];

D2a=chrom[j].coef[56];
D2b=chrom[j].coef[57];
D2gamma=chrom[j].coef[58];
D2delta=chrom[j].coef[59];
D2r=chrom[j].coef[60];
D2roll_1=chrom[j].coef[61];
D2roll_2=chrom[j].coef[62];
D2g=chrom[j].coef[63];
```

```
//************************************************

A1coef1= (A1a-A1b)*(A1a-A1b)*A1gamma*A1gamma*A1delta*A1delta;
A1coef2= A1gamma*A1gamma*((A1a-A1r)*(A1a-A1r)*A1roll_2*A1roll_2+A1roll_1*A1roll_1);
A1coef3= A1delta*A1delta*((A1b-A1r)*(A1b-A1r)*A1roll_2*A1roll_2+A1roll_1*A1roll_1);
A1coef4= A1roll_1*A1roll_1*A1roll_2*A1roll_2+A1g*A1g;

A2coef1= (A2a-A2b)*(A2a-A2b)*A2gamma*A2gamma*A2delta*A2delta;
A2coef2= A2gamma*A2gamma*((A2a-A2r)*(A2a-A2r)*A2roll_2*A2roll_2+A2roll_1*A2roll_1);
A2coef3= A2delta*A2delta*((A2b-A2r)*(A2b-A2r)*A2roll_2*A2roll_2+A2roll_1*A2roll_1);
A2coef4= A2roll_1*A2roll_1*A2roll_2*A2roll_2+A2g*A2g;

B1coef1= (B1a-B1b)*(B1a-B1b)*B1gamma*B1gamma*B1delta*B1delta;
B1coef2= B1gamma*B1gamma*((B1a-B1r)*(B1a-B1r)*B1roll_2*B1roll_2+B1roll_1*B1roll_1);
B1coef3= B1delta*B1delta*((B1b-B1r)*(B1b-B1r)*B1roll_2*B1roll_2+B1roll_1*B1roll_1);
B1coef4= B1roll_1*B1roll_1*B1roll_2*B1roll_2+B1g*B1g;

B2coef1= (B2a-B2b)*(B2a-B2b)*B2gamma*B2gamma*B2delta*B2delta;
B2coef2= B2gamma*B2gamma*((B2a-B2r)*(B2a-B2r)*B2roll_2*B2roll_2+B2roll_1*B2roll_1);
B2coef3= B2delta*B2delta*((B2b-B2r)*(B2b-B2r)*B2roll_2*B2roll_2+B2roll_1*B2roll_1);
B2coef4= B2roll_1*B2roll_1*B2roll_2*B2roll_2+B2g*B2g;

C1coef1= (C1a-C1b)*(C1a-C1b)*C1gamma*C1gamma*C1delta*C1delta;
C1coef2= C1gamma*C1gamma*((C1a-C1r)*(C1a-C1r)*C1roll_2*C1roll_2+C1roll_1*C1roll_1);
C1coef3= C1delta*C1delta*((C1b-C1r)*(C1b-C1r)*C1roll_2*C1roll_2+C1roll_1*C1roll_1);
C1coef4= C1roll_1*C1roll_1*C1roll_2*C1roll_2+C1g*C1g;

C2coef1= (C2a-C2b)*(C2a-C2b)*C2gamma*C2gamma*C2delta*C2delta;
C2coef2= C2gamma*C2gamma*((C2a-C2r)*(C2a-C2r)*C2roll_2*C2roll_2+C2roll_1*C2roll_1);
C2coef3= C2delta*C2delta*((C2b-C2r)*(C2b-C2r)*C2roll_2*C2roll_2+C2roll_1*C2roll_1);
C2coef4= C2roll_1*C2roll_1*C2roll_2*C2roll_2+C2g*C2g;

D1coef1= (D1a-D1b)*(D1a-D1b)*D1gamma*D1gamma*D1delta*D1delta;
D1coef2= D1gamma*D1gamma*((D1a-D1r)*(D1a-D1r)*D1roll_2*D1roll_2+D1roll_1*D1roll_1);
D1coef3= D1delta*D1delta*((D1b-D1r)*(D1b-D1r)*D1roll_2*D1roll_2+D1roll_1*D1roll_1);
D1coef4= D1roll_1*D1roll_1*D1roll_2*D1roll_2+D1g*D1g;

D2coef1= (D2a-D2b)*(D2a-D2b)*D2gamma*D2gamma*D2delta*D2delta;
D2coef2= D2gamma*D2gamma*((D2a-D2r)*(D2a-D2r)*D2roll_2*D2roll_2+D2roll_1*D2roll_1);
D2coef3= D2delta*D2delta*((D2b-D2r)*(D2b-D2r)*D2roll_2*D2roll_2+D2roll_1*D2roll_1);
D2coef4= D2roll_1*D2roll_1*D2roll_2*D2roll_2+D2g*D2g;

dum = 2/T;

coef[0] =  A1coef1*dum*dum + A1coef2*dum + A1coef3*dum + A1coef4;
coef[1] = -A1coef1*dum*dum + A1coef2*dum - A1coef3*dum + A1coef4;
coef[2] = -A1coef1*dum*dum - A1coef2*dum + A1coef3*dum + A1coef4;
coef[3] =  A1coef1*dum*dum - A1coef2*dum - A1coef3*dum + A1coef4;

coef[4] =  A2coef1*dum*dum + A2coef2*dum + A2coef3*dum + A2coef4;
coef[5] = -A2coef1*dum*dum + A2coef2*dum - A2coef3*dum + A2coef4;
coef[6] = -A2coef1*dum*dum - A2coef2*dum + A2coef3*dum + A2coef4;
coef[7] =  A2coef1*dum*dum - A2coef2*dum - A2coef3*dum + A2coef4;

coef[8] =  B1coef1*dum*dum + B1coef2*dum + B1coef3*dum + B1coef4;
coef[9] = -B1coef1*dum*dum + B1coef2*dum - B1coef3*dum + B1coef4;
coef[10] = -B1coef1*dum*dum - B1coef2*dum + B1coef3*dum + B1coef4;
coef[11] =  B1coef1*dum*dum - B1coef2*dum - B1coef3*dum + B1coef4;

coef[12] =  B2coef1*dum*dum + B2coef2*dum + B2coef3*dum + B2coef4;
```

```
        coef[13] = -B2coef1*dum*dum + B2coef2*dum - B2coef3*dum + B2coef4;
        coef[14] = -B2coef1*dum*dum - B2coef2*dum + B2coef3*dum + B2coef4;
        coef[15] =  B2coef1*dum*dum - B2coef2*dum - B2coef3*dum + B2coef4;

        coef[16] =  C1coef1*dum*dum + C1coef2*dum + C1coef3*dum + C1coef4;
        coef[17] = -C1coef1*dum*dum + C1coef2*dum - C1coef3*dum + C1coef4;
        coef[18] = -C1coef1*dum*dum - C1coef2*dum + C1coef3*dum + C1coef4;
        coef[19] =  C1coef1*dum*dum - C1coef2*dum - C1coef3*dum + C1coef4;

        coef[20] =  C2coef1*dum*dum + C2coef2*dum + C2coef3*dum + C2coef4;
        coef[21] = -C2coef1*dum*dum + C2coef2*dum - C2coef3*dum + C2coef4;
        coef[22] = -C2coef1*dum*dum - C2coef2*dum + C2coef3*dum + C2coef4;
        coef[23] =  C2coef1*dum*dum - C2coef2*dum - C2coef3*dum + C2coef4;

        coef[24] =  D1coef1*dum*dum + D1coef2*dum + D1coef3*dum + D1coef4;
        coef[25] = -D1coef1*dum*dum + D1coef2*dum - D1coef3*dum + D1coef4;
        coef[26] = -D1coef1*dum*dum - D1coef2*dum + D1coef3*dum + D1coef4;
        coef[27] =  D1coef1*dum*dum - D1coef2*dum - D1coef3*dum + D1coef4;

        coef[28] =  D2coef1*dum*dum + D2coef2*dum + D2coef3*dum + D2coef4;
        coef[29] = -D2coef1*dum*dum + D2coef2*dum - D2coef3*dum + D2coef4;
        coef[30] = -D2coef1*dum*dum - D2coef2*dum + D2coef3*dum + D2coef4;
        coef[31] =  D2coef1*dum*dum - D2coef2*dum - D2coef3*dum + D2coef4;

////////////////////////////
        pure1.re = 1.0;
        pure1.im = 0.0;
        stepsize1 = maxfreq/Nstep1;
        stepsize2 = maxfreq/Nstep2;
        ////////////////////////////
        temperr = 0;
        for (i = 0; i < 32; i++){
                if (fabs(coef[i]) > 64.0)
                        temperr += fabs(coef[i]) - 64.0;
        }

        max_err = -1.0;
        for (w1=-1.0*maxfreq; w1 < (maxfreq + stepsize1); w1=w1 +stepsize1){
                for (w2=0; w2 < (maxfreq + stepsize2); w2=w2 + stepsize2){
                        region = sqrt(w1*w1 + w2*w2);
                        z1=cexpj(T*w1);
                        z2=cexpj(T*w2);

                        temp1 = cmulti(z1,z2);
                        temp1 = cmulti1r1c(coef[0],temp1);
                        temp2 = cmulti1r1c(coef[1],z1);
                        temp3 = cmulti1r1c(coef[2],z2);

                        temp5 = cmulti(z1,z2);
                        temp5 = cmulti1r1c(coef[3],temp5);
                        temp6 = cmulti1r1c(coef[2],z1);
                        temp7 = cmulti1r1c(coef[1],z2);

                        tempdenum =  cadd3c1r(temp1,temp2,temp3,coef[3]);
                        tempnum   =  cadd3c1r(temp5,temp6,temp7,coef[0]);
                        A1= cdivid(tempnum,tempdenum);

                        temp1 = cmulti(z1,z2);
                        temp1 = cmulti1r1c(coef[4],temp1);
                        temp2 = cmulti1r1c(coef[5],z1);
                        temp3 = cmulti1r1c(coef[6],z2);
```

```
temp5 = cmulti(z1,z2);
temp5 = cmulti1r1c(coef[7],temp5);
temp6 = cmulti1r1c(coef[6],z1);
temp7 = cmulti1r1c(coef[5],z2);

tempdenum  =  cadd3c1r(temp1,temp2,temp3,coef[7]);
tempnum =            cadd3c1r(temp5,temp6,temp7,coef[4]);
A2= cdivid(tempnum,tempdenum);

temp1 = cmulti(z1,z2);
temp1 = cmulti1r1c(coef[8],temp1);
temp2 = cmulti1r1c(coef[9],z1);
temp3 = cmulti1r1c(coef[10],z2);

temp5 = cmulti(z1,z2);
temp5 = cmulti1r1c(coef[11],temp5);
temp6 = cmulti1r1c(coef[10],z1);
temp7 = cmulti1r1c(coef[9],z2);

tempdenum  =  cadd3c1r(temp1,temp2,temp3,coef[11]);
tempnum =            cadd3c1r(temp5,temp6,temp7,coef[8]);
B1= cdivid(tempnum,tempdenum);

temp1 = cmulti(z1,z2);
temp1 = cmulti1r1c(coef[12],temp1);
temp2 = cmulti1r1c(coef[13],z1);
temp3 = cmulti1r1c(coef[14],z2);

temp5 = cmulti(z1,z2);
temp5 = cmulti1r1c(coef[15],temp5);
temp6 = cmulti1r1c(coef[14],z1);
temp7 = cmulti1r1c(coef[13],z2);

tempdenum  =  cadd3c1r(temp1,temp2,temp3,coef[15]);
tempnum =            cadd3c1r(temp5,temp6,temp7,coef[12]);
B2= cdivid(tempnum,tempdenum);


//////////////////////////////////////////////////
temp1 = cmulti(z1,z2);
temp1 = cmulti1r1c(coef[16],temp1);
temp2 = cmulti1r1c(coef[17],z1);
temp3 = cmulti1r1c(coef[18],z2);

temp5 = cmulti(z1,z2);
temp5 = cmulti1r1c(coef[19],temp5);
temp6 = cmulti1r1c(coef[18],z1);
temp7 = cmulti1r1c(coef[17],z2);

tempdenum  =  cadd3c1r(temp1,temp2,temp3,coef[19]);
tempnum =            cadd3c1r(temp5,temp6,temp7,coef[16]);
C1= cdivid(tempnum,tempdenum);

temp1 = cmulti(z1,z2);
temp1 = cmulti1r1c(coef[20],temp1);
temp2 = cmulti1r1c(coef[21],z1);
temp3 = cmulti1r1c(coef[22],z2);

temp5 = cmulti(z1,z2);
temp5 = cmulti1r1c(coef[23],temp5);
```

```
                        temp6 = cmulti1r1c(coef[22],z1);
                        temp7 = cmulti1r1c(coef[21],z2);

                        tempdenum  = cadd3c1r(temp1,temp2,temp3,coef[23]);
                        tempnum =           cadd3c1r(temp5,temp6,temp7,coef[20]);
                        C2= cdivid(tempnum,tempdenum);

                        temp1 = cmulti(z1,z2);
                        temp1 = cmulti1r1c(coef[24],temp1);
                        temp2 = cmulti1r1c(coef[25],z1);
                        temp3 = cmulti1r1c(coef[26],z2);

                        temp5 = cmulti(z1,z2);
                        temp5 = cmulti1r1c(coef[27],temp5);
                        temp6 = cmulti1r1c(coef[26],z1);
                        temp7 = cmulti1r1c(coef[25],z2);

                        tempdenum  = cadd3c1r(temp1,temp2,temp3,coef[27]);
                        tempnum =           cadd3c1r(temp5,temp6,temp7,coef[24]);
                        D1= cdivid(tempnum,tempdenum);

                        temp1 = cmulti(z1,z2);
                        temp1 = cmulti1r1c(coef[28],temp1);
                        temp2 = cmulti1r1c(coef[29],z1);
                        temp3 = cmulti1r1c(coef[30],z2);

                        temp5 = cmulti(z1,z2);
                        temp5 = cmulti1r1c(coef[31],temp5);
                        temp6 = cmulti1r1c(coef[30],z1);
                        temp7 = cmulti1r1c(coef[29],z2);

                        tempdenum  = cadd3c1r(temp1,temp2,temp3,coef[31]);
                        tempnum =           cadd3c1r(temp5,temp6,temp7,coef[28]);
                        D2= cdivid(tempnum,tempdenum);

                        A=cmulti(A1,A2);
                        B=cmulti(B1,B2);
                        C=cmulti(C1,C2);
                        D=cmulti(D1,D2);

tempz1 = cdivid(pure1 , z1);
tempz2 = cdivid(pure1 , z2);

                        tempApluszlB = cadd( A, cmulti(tempz1,B));
                        tempCplusz2D = cadd( C, cmulti(tempz2,D));

                        H = cmulti1r2c(0.25, tempApluszlB, tempCplusz2D);

                        if ( region <= passband ){
                                err = fabs(1.0 - cmag(H));
                                if( max_err < err ) max_err = err;
                        }
                        else if ( region <= stopband){
                                mag = exp(-1.0* alpha_tran * (region - passband));
                                err = fabs(mag - cmag(H));
                                if( (max_err < err) && (err > 0.2)) max_err = err;
                        }
                        else {
                                err = fabs(cmag(H));
                                if( max_err < err ) max_err = err;
                        }
```

```
                    }
   }

          return (1.0 / (max_err + temperr));
}

void GETchromfit(){   //passing decoded coefficients to fitness
          int j;

          for( j = 0; j < Nchrom; j++)
                    chrom[j].fitness = fitness(j);
}

void report(){
          int i;
          float  A1a, A1b, A1gamma, A1delta, A1r,       A1roll_1, A1roll_2, A1g;
          float  A2a, A2b, A2gamma, A2delta, A2r,       A2roll_1, A2roll_2, A2g;
          float  B1a, B1b, B1gamma, B1delta, B1r,       B1roll_1, B1roll_2, B1g;
          float  B2a, B2b, B2gamma, B2delta, B2r,       B2roll_1, B2roll_2, B2g;
          float  C1a, C1b, C1gamma, C1delta, C1r,       C1roll_1, C1roll_2, C1g;
          float  C2a, C2b, C2gamma, C2delta, C2r,       C2roll_1, C2roll_2, C2g;
          float  D1a, D1b, D1gamma, D1delta, D1r,       D1roll_1, D1roll_2, D1g;
          float  D2a, D2b, D2gamma, D2delta, D2r,       D2roll_1, D2roll_2, D2g;

  float  A1coef1, A1coef2, A1coef3, A1coef4, A2coef1, A2coef2, A2coef3, A2coef4;
  float  B1coef1, B1coef2, B1coef3, B1coef4, B2coef1, B2coef2, B2coef3, B2coef4;
  float  C1coef1, C1coef2, C1coef3, C1coef4, C2coef1, C2coef2, C2coef3, C2coef4;
  float  D1coef1, D1coef2, D1coef3, D1coef4, D2coef1, D2coef2, D2coef3, D2coef4;

          float dum;

          /* open output file */
          outptr = fopen( "output.dat","w+");
          if ( outptr == NULL){ printf("cannot open output file"); exit(-1); }

          /* printout the coefficient and exponents values */
          for(i = 0; i < Ncoef; i++){
                    fprintf(outptr, "%12.8f,\n",Bchrom.coef[i]);
          }
          fprintf(outptr,"\n\n");

          A1a=Bchrom.coef[0];
          A1b=Bchrom.coef[1];
          A1gamma=Bchrom.coef[2];
          A1delta=Bchrom.coef[3];
          A1r=Bchrom.coef[4];
          A1roll_1=Bchrom.coef[5];
          A1roll_2=Bchrom.coef[6];
          A1g=Bchrom.coef[7];

          A2a=Bchrom.coef[8];
          A2b=Bchrom.coef[9];
          A2gamma=Bchrom.coef[10];
          A2delta=Bchrom.coef[11];
          A2r=Bchrom.coef[12];
          A2roll_1=Bchrom.coef[13];
          A2roll_2=Bchrom.coef[14];
          A2g=Bchrom.coef[15];

          B1a=Bchrom.coef[16];
          B1b=Bchrom.coef[17];
```

```
B1gamma=Bchrom.coef[18];
B1delta=Bchrom.coef[19];
B1r=Bchrom.coef[20];
B1roll_1=Bchrom.coef[21];
B1roll_2=Bchrom.coef[22];
B1g=Bchrom.coef[23];

B2a=Bchrom.coef[24];
B2b=Bchrom.coef[25];
B2gamma=Bchrom.coef[26];
B2delta=Bchrom.coef[27];
B2r=Bchrom.coef[28];
B2roll_1=Bchrom.coef[29];
B2roll_2=Bchrom.coef[30];
B2g=Bchrom.coef[31];

C1a=Bchrom.coef[32];
C1b=Bchrom.coef[33];
C1gamma=Bchrom.coef[34];
C1delta=Bchrom.coef[35];
C1r=Bchrom.coef[36];
C1roll_1=Bchrom.coef[37];
C1roll_2=Bchrom.coef[38];
C1g=Bchrom.coef[39];

C2a=Bchrom.coef[40];
C2b=Bchrom.coef[41];
C2gamma=Bchrom.coef[42];
C2delta=Bchrom.coef[43];
C2r=Bchrom.coef[44];
C2roll_1=Bchrom.coef[45];
C2roll_2=Bchrom.coef[46];
C2g=Bchrom.coef[47];

D1a=Bchrom.coef[48];
D1b=Bchrom.coef[49];
D1gamma=Bchrom.coef[50];
D1delta=Bchrom.coef[51];
D1r=Bchrom.coef[52];
D1roll_1=Bchrom.coef[53];
D1roll_2=Bchrom.coef[54];
D1g=Bchrom.coef[55];

D2a=Bchrom.coef[56];
D2b=Bchrom.coef[57];
D2gamma=Bchrom.coef[58];
D2delta=Bchrom.coef[59];
D2r=Bchrom.coef[60];
D2roll_1=Bchrom.coef[61];
D2roll_2=Bchrom.coef[62];
D2g=Bchrom.coef[63];

//*************************************************

A1coef1= (A1a-A1b)*(A1a-A1b)*A1gamma*A1gamma*A1delta*A1delta;
A1coef2= A1gamma*A1gamma*((A1a-A1r)*(A1a-A1r)*A1roll_2*A1roll_2+A1roll_1*A1roll_1);
A1coef3= A1delta*A1delta*((A1b-A1r)*(A1b-A1r)*A1roll_2*A1roll_2+A1roll_1*A1roll_1);
A1coef4= A1roll_1*A1roll_1*A1roll_2*A1roll_2+A1g*A1g;

A2coef1= (A2a-A2b)*(A2a-A2b)*A2gamma*A2gamma*A2delta*A2delta;
A2coef2= A2gamma*A2gamma*((A2a-A2r)*(A2a-A2r)*A2roll_2*A2roll_2+A2roll_1*A2roll_1);
```

```
A2coef3= A2delta*A2delta*((A2b-A2r)*(A2b-A2r)*A2roll_2*A2roll_2+A2roll_1*A2roll_1);
A2coef4= A2roll_1*A2roll_1*A2roll_2*A2roll_2+A2g*A2g;

B1coef1= (B1a-B1b)*(B1a-B1b)*B1gamma*B1gamma*B1delta*B1delta;
B1coef2= B1gamma*B1gamma*((B1a-B1r)*(B1a-B1r)*B1roll_2*B1roll_2+B1roll_1*B1roll_1);
B1coef3= B1delta*B1delta*((B1b-B1r)*(B1b-B1r)*B1roll_2*B1roll_2+B1roll_1*B1roll_1);
B1coef4= B1roll_1*B1roll_1*B1roll_2*B1roll_2+B1g*B1g;

B2coef1= (B2a-B2b)*(B2a-B2b)*B2gamma*B2gamma*B2delta*B2delta;
B2coef2= B2gamma*B2gamma*((B2a-B2r)*(B2a-B2r)*B2roll_2*B2roll_2+B2roll_1*B2roll_1);
B2coef3= B2delta*B2delta*((B2b-B2r)*(B2b-B2r)*B2roll_2*B2roll_2+B2roll_1*B2roll_1);
B2coef4= B2roll_1*B2roll_1*B2roll_2*B2roll_2+B2g*B2g;

C1coef1= (C1a-C1b)*(C1a-C1b)*C1gamma*C1gamma*C1delta*C1delta;
C1coef2= C1gamma*C1gamma*((C1a-C1r)*(C1a-C1r)*C1roll_2*C1roll_2+C1roll_1*C1roll_1);
C1coef3= C1delta*C1delta*((C1b-C1r)*(C1b-C1r)*C1roll_2*C1roll_2+C1roll_1*C1roll_1);
C1coef4= C1roll_1*C1roll_1*C1roll_2*C1roll_2+C1g*C1g;

C2coef1= (C2a-C2b)*(C2a-C2b)*C2gamma*C2gamma*C2delta*C2delta;
C2coef2= C2gamma*C2gamma*((C2a-C2r)*(C2a-C2r)*C2roll_2*C2roll_2+C2roll_1*C2roll_1);
C2coef3= C2delta*C2delta*((C2b-C2r)*(C2b-C2r)*C2roll_2*C2roll_2+C2roll_1*C2roll_1);
C2coef4= C2roll_1*C2roll_1*C2roll_2*C2roll_2+C2g*C2g;

D1coef1= (D1a-D1b)*(D1a-D1b)*D1gamma*D1gamma*D1delta*D1delta;
D1coef2= D1gamma*D1gamma*((D1a-D1r)*(D1a-D1r)*D1roll_2*D1roll_2+D1roll_1*D1roll_1);
D1coef3= D1delta*D1delta*((D1b-D1r)*(D1b-D1r)*D1roll_2*D1roll_2+D1roll_1*D1roll_1);
D1coef4= D1roll_1*D1roll_1*D1roll_2*D1roll_2+D1g*D1g;

D2coef1= (D2a-D2b)*(D2a-D2b)*D2gamma*D2gamma*D2delta*D2delta;
D2coef2= D2gamma*D2gamma*((D2a-D2r)*(D2a-D2r)*D2roll_2*D2roll_2+D2roll_1*D2roll_1);
D2coef3= D2delta*D2delta*((D2b-D2r)*(D2b-D2r)*D2roll_2*D2roll_2+D2roll_1*D2roll_1);
D2coef4= D2roll_1*D2roll_1*D2roll_2*D2roll_2+D2g*D2g;

dum = 2/T;

coef[0] =  A1coef1*dum*dum + A1coef2*dum + A1coef3*dum + A1coef4;
coef[1] = -A1coef1*dum*dum + A1coef2*dum - A1coef3*dum + A1coef4;
coef[2] = -A1coef1*dum*dum - A1coef2*dum + A1coef3*dum + A1coef4;
coef[3] =  A1coef1*dum*dum - A1coef2*dum - A1coef3*dum + A1coef4;

coef[4] =  A2coef1*dum*dum + A2coef2*dum + A2coef3*dum + A2coef4;
coef[5] = -A2coef1*dum*dum + A2coef2*dum - A2coef3*dum + A2coef4;
coef[6] = -A2coef1*dum*dum - A2coef2*dum + A2coef3*dum + A2coef4;
coef[7] =  A2coef1*dum*dum - A2coef2*dum - A2coef3*dum + A2coef4;

coef[8] =  B1coef1*dum*dum + B1coef2*dum + B1coef3*dum + B1coef4;
coef[9] = -B1coef1*dum*dum + B1coef2*dum - B1coef3*dum + B1coef4;
coef[10] = -B1coef1*dum*dum - B1coef2*dum + B1coef3*dum + B1coef4;
coef[11] =  B1coef1*dum*dum - B1coef2*dum - B1coef3*dum + B1coef4;

coef[12] =  B2coef1*dum*dum + B2coef2*dum + B2coef3*dum + B2coef4;
coef[13] = -B2coef1*dum*dum + B2coef2*dum - B2coef3*dum + B2coef4;
coef[14] = -B2coef1*dum*dum - B2coef2*dum + B2coef3*dum + B2coef4;
coef[15] =  B2coef1*dum*dum - B2coef2*dum - B2coef3*dum + B2coef4;

coef[16] =  C1coef1*dum*dum + C1coef2*dum + C1coef3*dum + C1coef4;
coef[17] = -C1coef1*dum*dum + C1coef2*dum - C1coef3*dum + C1coef4;
coef[18] = -C1coef1*dum*dum - C1coef2*dum + C1coef3*dum + C1coef4;
coef[19] =  C1coef1*dum*dum - C1coef2*dum - C1coef3*dum + C1coef4;

coef[20] =  C2coef1*dum*dum + C2coef2*dum + C2coef3*dum + C2coef4;
```

```
    coef[21] = -C2coef1*dum*dum + C2coef2*dum - C2coef3*dum + C2coef4;
    coef[22] = -C2coef1*dum*dum - C2coef2*dum + C2coef3*dum + C2coef4;
    coef[23] =  C2coef1*dum*dum - C2coef2*dum - C2coef3*dum + C2coef4;

    coef[24] =  D1coef1*dum*dum + D1coef2*dum + D1coef3*dum + D1coef4;
    coef[25] = -D1coef1*dum*dum + D1coef2*dum - D1coef3*dum + D1coef4;
    coef[26] = -D1coef1*dum*dum - D1coef2*dum + D1coef3*dum + D1coef4;
    coef[27] =  D1coef1*dum*dum - D1coef2*dum - D1coef3*dum + D1coef4;

    coef[28] =  D2coef1*dum*dum + D2coef2*dum + D2coef3*dum + D2coef4;
    coef[29] = -D2coef1*dum*dum + D2coef2*dum - D2coef3*dum + D2coef4;
    coef[30] = -D2coef1*dum*dum - D2coef2*dum + D2coef3*dum + D2coef4;
    coef[31] =  D2coef1*dum*dum - D2coef2*dum - D2coef3*dum + D2coef4;

    for(i = 0; i < 32 ; i++){
            fprintf(outptr, "%25.20f,\n",coef[i]);
    }

    fprintf(outptr, "\n\nThe best fitness is %f", Bchrom.fitness);
    fclose(outptr);
}


/**********************
** Main Program *******
**********************/
void main(int argsize, char **arg){
    int Citer, chromctr, coefctr, MaxIter,index1,index2;
    aCHROM *temp;

    if ( argsize != 5 ){ printf("\nwrong number of arguments!!"); exit(-1);}

    MaxIter         = (int)atoi(arg[1]);
    Nchrom          = (int)atoi(arg[2]);
    pmut            = (float)atof(arg[3]);
    pcross          = (float)atof(arg[4]);
    Ncoef           = 64;
    nvars           = Ncoef;

    srand( (unsigned)time( NULL ) );

    /* restriction on the population size */
    if((Nchrom % 2) != 0 ) Nchrom--;

    /* create space for chromosome */
    chrom = new aCHROM[Nchrom];
    if (chrom == NULL) {printf("don't have enough space, chrom\n"); exit(-1);}

    /* create space for offsprg */
    offsprg = new aCHROM[Nchrom];
    if (offsprg == NULL) {printf("don't have enough space, offsprg\n"); exit(-1);}

    /* create space for ordering */
    ordering = new int[Nchrom];
    if (ordering == NULL) {printf("don't have enough space, ordering\n"); exit(-1);}

/*  domain[0] =-7.0;
    domain[1] =-6.0;
    domain[2] =-5.0;
    domain[3] =-4.0;
    domain[4] =-3.5;
```

```
            domain[5] =-3.0;
            domain[6] =-2.5;
            domain[7] =-2.0;
            domain[8] =-1.75;
            domain[9] =-1.5;
            domain[10] =-1.25;
            domain[11] =-1.0;
            domain[12] =-0.875;
            domain[13] =-0.75;
            domain[14] =-0.625;
            domain[15] =-0.5;
            domain[16] =-0.375;
            domain[17] =-0.25;
            domain[18] =-0.125;
            domain[19] =0.125;
            domain[20] =0.25;
            domain[21] =0.375;
            domain[22] =0.5;
            domain[23] =0.625;
            domain[24] =0.75;
            domain[25] =0.875;
            domain[26] =1.0;
            domain[27] =1.25;
            domain[28] =1.5;
            domain[29] =1.75;
            domain[30] =2.5;
            domain[31] =2.0;
            domain[32] =3.0;
            domain[33] =3.5;
            domain[34] =4.0;
            domain[35] =5.0;
            domain[36] =6.0;
            domain[37] =7.0;
*/
            domain[0] = -3.0;
            domain[1] =-2.0;
            domain[2] =-1.5;
            domain[3] =-1.0;
            domain[4] =-0.75;
            domain[5] =-0.5;
            domain[6] =-0.25;
            domain[7] =0.25;
            domain[8] =0.5;
            domain[9] =0.75;
            domain[10] =1.0;
            domain[11] =1.5;
            domain[12] =2.0;
            domain[13] =3.0;

  //get the random chromosome
            presetChrom();

            //calcuate the fitness of each chromosome
            GETchromfit();
            Bchrom.fitness = -1.0;
            Bchrom.absfitness = -1;
            Citer = 0;
            scaleFitness();
            while ( (Citer < MaxIter) && !kbhit() ){
                        Citer++;
```

```
        //find Noffsprg and duplicate the chrom to offsprg
                        reproduce();

                        //crossover, mutation
                        for (chromctr = 0; chromctr < Nchrom; chromctr = chromctr + 2){
                                index1 = ordering[chromctr];
                                index2 = ordering[chromctr + 1];
                                for (coefctr = 0; coefctr < Ncoef; coefctr++)
                                        DOcrossover(index1, index2, chromctr, chromctr + 1, coefctr);
                                DOmutation(chromctr);
                                DOmutation(chromctr + 1);
                        }

                        /* exchange pointer */
                        temp = chrom;
                        chrom = offsprg;
                        offsprg = temp;

                        //calcuate fitness of each offsprg
                        GETchromfit();
                        scaleFitness();
//                      if ((Citer % 30) == 0 )
                                printf("\n%d          %f", Citer, Bchrom.fitness );
                }

                if ( kbhit() ) getch();
                printf("\n");
                report();

                delete[] chrom;
                delete[] offsprg;
                delete[] ordering;
}
```

VITA AUCTORIS

Alfred Lee was born in Hong Kong in 1971. He immigrated to Canada in 1989. He obtained his Ontario Secondary School Diploma from Sir John A. Macdonald Secondary School in 1991. He received his Bachelor of Engineering in Computer Engineering from McMaster University in 1996. He is currently a candidate for the degree of Master of Applied Science at the University of Windsor. He is currently employed as a Software Engineer in SWI Systemware consultant company, in Toronto, Canada.

PUBLICATION OUT OF THE THESIS

1. A.Lee, M.Ahmadi, V.Ramachandran, C.S.Gargour, "Design of Fractional Delay Operator", *Proc. of the World Automation Congress*, pp657-662, May1998.

2. A.Lee, M.Ahmadi, G.A.Jullien, W.C.Miller, R.S.Lashkari, "Digital Filter Design Using Genetic Algorithm", *Proc. of 1998 IEEE Symp.on Advances in Digital Filtering and Signal Processing*, pp34-38, June 1998.

3. A.Lee, M.Ahmadi, R.S.Lashkari, "Design of Stable 2-D Recursive Filters Using Power-of-Two Coefficients", *Proc. of IEEE international Conf. on Electronics Circuits and Systems*, pp409-412, Sept. 1998.

4. A.Lee, M.Ahmadi, R.S.Lashkari, "Design of 2-D FIR Filter With McClellan Transformation and Genetic Algorithms", *Proc. of 2nd Inter. Symp. on Artificial Intelligence Adaptive System.*, pp51-58, March 1999.

5. A.Lee, M.Ahmadi, G.A.jullien, R.S.Lashkari, W.C.Miller, "Design of 1-D FIR Filters with Genetic Algorithms", *Proc. of IEEE ISCAS 1999.* Vol.III, pp 295-298.