1-1-2006

Design of high throughput recursive and non-recursive digital filters in one and two dimensions with Canonic Signed Digit coefficients and sub-expression elimination using Genetic Algorithm.

Tom Williams
University of Windsor

DESIGN OF HIGH THROUGHPUT RECURSIVE AND NON-RECURSIVE DIGITAL FILTERS IN ONE AND TWO DIMENSIONS WITH CANONIC SIGNED DIGIT COEFFICIENTS AND SUB-EXPRESSION ELIMINATION USING GENETIC ALGORITHM

by
Tom Williams

A Dissertation
Submitted to the Faculty of Graduate Studies and Research
through Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy at the
University of Windsor

Windsor, Ontario, Canada
2006

Your file Votre référence
ISBN: 978-0-494-35974-7
Our file Notre référence
ISBN: 978-0-494-35974-7

Abstract

In this dissertation methods of obtaining high throughput rate digital filters are examined. The use of Canonic Signed Digit (CSD) filter coefficients is established and a new chromosome coding technique is developed to enable efficient design of non-recursive filters using a Genetic Algorithm.

The new genetic algorithm approach using the proposed new coding scheme is extended to efficiently handle recursive filters using a new unstable penalty factor to handle the instability constraints imposed by such filters. A technique is presented that allows these new methods to be applied to the design of high throughput 2-D filters.

The throughput rate of CSD coefficients digital filters is further increased by the use of common sub-expression elimination. A new graphical transformation is presented that allows for optimization of the elimination of CSD-coefficient common sub-expressions in both the vertical and horizontal dimensions.

The effectiveness of the proposed methods is demonstrated with example designs and comparisons to other methods.

Dedication

To Joanne

Acknowledgments

I would like to acknowledge the guidance and support provided by Dr. Majid Ahmadi. His advice, suggestions and comments have been invaluable. I would also like to thank my committee members Dr. Chunhong Chen, Dr. Reza Laskari and Dr. Behnam Shahrrava for their invaluable comments, suggestions and feedback. A thank you also goes to Tim Johnston, Jim Smith and Dan Sooley for there ever present ideas and discussions.

Table of Contents

ABSTRACT iii

DEDICATION iv

ACKNOWLEDGMENTS v

LIST OF TABLES xii

LIST OF FIGURES xv

LIST OF ABBREVIATIONS xviii

CHAPTER 1. INTRODUCTION 1

1.1 CANONICAL SIGNED DIGIT NUMBER SYSTEM 3

1.2 LIMITING NON-ZERO BITS IN COEFFICIENTS 5

1.3 SURVEY OF FILTER DESIGN METHODOLOGIES WITH CSD COEFFICIENTS 6

1.3.1. CONVERSION 6

1.3.2. ALGEBRAIC DESIGN 7

1.3.3. SEARCH AND OPTIMIZATION 7

1.4 COMMON SUB-EXPRESSION ELIMINATION 8

1.5 ORGANIZATION OF THIS DISSERTATION 8

CHAPTER 2. GENETIC ALGORITHMS 9

2.1 INTRODUCTION 9

2.2 ORIGIN AND OPERATION....9

2.3 CLASSES OF SEARCH TECHNIQUES....11

2.4 PROBLEM CODING....13

2.4.1. FITNESS FUNCTION....14

2.4.2. REPRODUCTION....14

2.5 EXAMPLE PROBLEM....15

2.6 ANALYSIS OF THE SIMPLE THREE OPERATOR GENETIC ALGORITHM....17

2.6.1. SCHEMATA....17

2.7 THE FUNDAMENTAL THEOREM OF GENETIC ALGORITHMS....18

2.7.1. NOTATION....18

2.7.2. ANALYSIS OF THE FUNDAMENTAL THEOREM OF GENETIC ALGORITHMS....19

2.8 IMPLICIT PARALLELISM....23

2.9 ADVANCED TECHNIQUES....24

2.9.1. CROSSOVER TECHNIQUES....24

2.9.1.A) 2-POINT CROSSOVER....24

2.9.1.B) UNIFORM CROSSOVER....25

2.9.2.CROSSOVER COMPARISONS....26

2.9.3. OTHER CROSSOVER TECHNIQUES....28

2.9.4.CROSSOVER CONCLUSION....29

2.9.5. MUTATION....29

2.9.6. INVERSION AND REORDERING....30

2.9.7. DECEPTION....31

2.9.8. EPISTASIS....32

2.9.9. CHROMOSOME ALPHABETS....33

2.9.10. OPERATOR PARAMETERS....34

2.9.11. PROBLEM CONSTRAINTS AND INVALID CHROMOSOMES....34

2.9.11.A) CHROMOSOME REMAPING....36

2.10 CONCLUSION....36

CHAPTER 3. 1-D FILTER DESIGN....37

3.1 APPLICATION OF THE GA TO FILTER DESIGN....37

3.2 CSD CHROMOSOME CODING....37

3.3 EFFECTS OF CROSSOVER AND MUTATION ON CSD VALUES....38

3.4 EFFECTS OF GA DISRUPTION....41

3.5 PROPOSED DESIGN TECHNIQUE....41

3.6 NON-RECURSIVE FILTER DESIGN EXAMPLE....45

3.7 COMPARISON WITH EXISTING DESIGN..... 46

3.8 COMPARISON OF OPTIMUM SOLUTION WITH CSD CONVERSION 54

3.9 CONCLUSION..... 76

CHAPTER 4. 1-D RECURSIVE FILTER DESIGN.....77

4.1 RECURSIVE FILTERS.....77

4.2 DETERMINING FILTER STABILITY..... 78

4.3 RECURSIVE FILTER DESIGN EXAMPLE.....81

4.4 OPTIMUM GA POPULATION SIZE TEST..... 83

4.5 CONCLUSION..... 86

CHAPTER 5. TWO-DIMENSIONAL FILTERS.....87

5.1 INTRODUCTION.....87

5.2 2-D FILTERS AS CASCADED 1-D FILTERS..... 88

5.3 FIR DESIGN EXAMPLE..... 90

5.4 RECURSIVE 2-D DESIGN EXAMPLE..... 95

5.5 2-D NON-RECURSIVE FILTER EXAMPLE COMPARISON 103

5.6 CONCLUSION..... 104

CHAPTER 6. COMMON SUBEXPRESSION ELIMINATION 106

6.1 INTRODUCTION.....106

6.2 SUBEXPRESSION TYPES 106

6.3 HORIZONTAL SUB-EXPRESSION ELIMINATION WITHIN A COEFFICIENT....106

6.4 VERTICAL SUB-EXPRESSION ELIMINATION....107

6.5 HORIZONTAL SUB-EXPRESSION ELIMINATION ACROSS COEFFICIENTS....108

6.6 GRAPHICAL TRANSFORMATION....109

6.6.1. IDENTIFICATION GRAPH....109

6.7 SEARCH GRAPH....114

6.8 EXAMPLE WALK THROUGH SEARCH GRAPH....115

6.9 EXAMPLE ELIMINATION USING A GA....116

6.9.1. FITNESS FUNCTION....116

6.10 EXAMPLE....117

6.11 APPLICATION TO PREVIOUS RESULTS....119

6.12SUMMARY OF APPLICATION TO PREVIOUS RESULTS....126

6.13 CONCLUSION....126

CHAPTER 7. FUTURE WORK....127

7.1 1-D RECURSIVE FILTERS....127

7.2 2-D FILTERS....127

7.3 COMMON SUB-EXPRESSION ELIMINATION....127

CHAPTER 8. CONCLUSION....128

REFERENCES..... 129

APPENDIX A SOURCE CODE.....134

APPENDIX B DEFINITIONS.....164

VITA AUCTORIS..... 169

List of Tables

Table 1 GA Example: Initial Population 16

Table 2 GA Example: Second Population 16

Table 3 Conversion Table for CSD Length L=7 44

Table 4 Genetic Algorithm Parameters 46

Table 5 Coefficients from Optimum Design 48

Table 6 Coefficients for Filter from [36] 51

Table 7 Coefficients of Filter from Proposed GA Method 53

Table 8 Comparison Summary 53

Table 9 Infinite Precision Coefficients 57

Table 10 Coefficients with a Maximum of 6 Non-Zero Digits 58

Table 11 Coefficients with a Maximum of 5 Non-Zero Digits 60

Table 12 Coefficients with a Maximum of 4 Non-Zero Digits 64

Table 13 Coefficients with a Maximum of 3 Non-Zero Digits 67

Table 14 Coefficients with a Maximum of 2 Non-Zero Digits 69

Table 15 Coefficients with a Maximum of 1 Non-Zero Digit 72

Table 16 Filter Square Error Relative to Filter with Optimum Infinite Precision Coefficients 75

Table 17 Coefficients of Example Recursive Filter 82

Table 18 Optimum GA Population Size Test Results 84

Table 19. GA Parameters for 2-D FIR Example 90

Table 20 Coefficients of F1 and G1 for 2-D FIR Example 93

Table 21 Coefficients of F2 and G3 for 2-D FIR Example..... 94

Table 22 Coefficients of F3 and G3 for 2-D FIR Example..... 95

Table 23 GA Parameters used for the 2-D Non-Recursive Example 96

Table 24. Coefficients in Decimal and CSD Representation (where n = -1) for the 2-D Filter using Cascaded 2nd Order 1-D Filters. 98

Table 25. Coefficients in Decimal and CSD Representation (where n = -1) for 2-D Filter using Cascaded 3rd Order 1-D Filters..... 100

Table 26 Coefficients in Decimal and CSD Representation (where n = -1) for 2-D Filter using Cascaded 4th Order 1-D Filters 102

Table 27. 1-D component filter bias values..... 103

Table 28 Mean Square Error and Number of Additions Required for Coefficient Multiplication of Considered 2-D Filters..... 104

Table 29 Coefficient Stacking..... 111

Table 30 Properties of Vertices in ID Graph Example..... 112

Table 31 Edge list E'id with Edge Properties..... 113

Table 32 ID Graph Vertex Availability Table after (S6, S3) Elimination..... 116

Table 33 CSD Coefficients..... 118

Table 34 Coefficients with Eliminated Subexpressions..... 119

Table 35 Eliminated Subexpression Labeling..... 119

Table 36 Eliminated Sub-Expressions for Maximum 2 Non-Zero Digits..... 121

Table 37 Eliminated Sub-Expressions for Maximum 3 Non-Zero Digits..... 122

Table 38 Eliminated subexpressions for maximum 4 non-zero digits..... 123

Table 39 Eliminated subexpressions for maximum 5 non-zero digits..... 124

Table 40 Eliminated subexpressions for maximum 6 non-zero digits..... 125

Table 41 Summary of Application to Previous Results..... 126

List of Figures

Fig. 1.1 Shift-Add binary multiplication..... 3

Fig. 2.1 Search Techniques..... 11

Fig. 2.2 Simple Genetic Algorithm..... 12

Fig. 2.3 Simple Three Operator Genetic Algorithm 13

Fig. 2.4 Single Point Crossover..... 15

Fig. 2.5 Uniform Crossover..... 26

Fig. 3.1 Invalid CSD Crossover..... 38

Fig. 3.2 Invalid CSD Mutation..... 39

Fig. 3.3 Unlikely Mating Partners..... 40

Fig. 3.4 Impossible Mating Partners..... 40

Fig. 3.5 Response of Optimal Design using IP Coefficients..... 47

Fig. 3.6 Response of Optimal Design Converted to CSD Coefficients 49

Fig. 3.7 Response of Filter from [36]..... 50

Fig. 3.8 Response of design by proposed new method..... 52

Fig. 3.9 Response using Infinite Precision oefficients..... 55

Fig. 3.10. Response of Filter with CSD Coefficients Converted from IP with a Maximum of 6 Non-Zero Digits..... 56

Fig. 3.11 Response of Filter with CSD Coefficients Designed by GA with Maximum 6 Non-Zero Digits..... 59

Fig. 3.12 Response of Filter with CSD Coefficients Converted from IP with a Maximum of 5 Non-Zero Digit..... 61

Fig. 3.13 Response of filter with CSD Coefficients Designed by proposed GA method with a Maximum 5 of Non-Zero Digits..... 62
Fig. 3.14 Response of filter with CSD coefficients converted from IP with maximum 4 non-zero digits..... 63
Fig. 3.15 Response of filter with CSD coefficients designed by GA with maximum 4 non-zero digit..... 65
Fig. 3.16 Response of Filter with CSD Coefficients Converted from IP with a Maximum of 3 Non-Zero Digits..... 66
Fig. 3.17 Response of the Filter with CSD coefficients Designed by GA with a Maximum 3 Non-Zero Digits..... 68
Fig. 3.18 Response of the Filter with CSD Coefficients Converted from IP with a maximum of 2 Non-Zero Digits..... 70
Fig. 3.19 Response of the Filter with CSD Coefficients Designed by GA with a Maximum 2 Non-Zero Digits..... 71
Fig. 3.20 CSD Coefficients Converted from IP with a Maximum of 1 non-zero digit..... 73
Fig. 3.21 Response of the Filter with CSD Coefficients Designed by GA with a Maximum of 1 Non-Zero Digit..... 74
Fig. 4.1 Optimum Penalty Factor Determination..... 80
Fig. 4.2 z-Plane Plot of the Poles (X) and Zeros (O) of the Example Non-Recursive Filter..... 83
Fig. 4.3 Optimum GA Population Size Test Results..... 85
Fig. 5.1 Cascaded 1-D Filters Form a 2-D Filter..... 88

Fig. 5.2 Target Response..... 91

Fig. 5.3 High Throughput Filter Response..... 91

Fig. 5.4 Target Response Error..... 92

Fig. 5.5. Desired Magnitude Response of the 2-D Filter..... 96

Fig. 5.6. Amplitude Response of the 2-D Filter using Cascaded 2nd order 1-D Filters with CSD Coefficients..... 97

Fig. 5.7 Amplitude Response of the 2-D Filter using Cascaded 3rd Order 1-D filters with CSD Coefficients..... 99

Fig. 5.8 Amplitude response of the 2-D filter using cascaded 4th order 1-D filters with CSD coefficients..... 101

Fig. 6.1 ID Graph with Only Vertices..... 112

Fig. 6.2 Partial ID Graph G'id..... 113

Fig. 6.3 Completed ID Graph Gid..... 114

Fig. 6.4. Search Graph Gs 115

List of Abbreviations

0	Filter Zero
1-D	One-dimension
2-D	Two-dimensions
Bit limited	Non-zero-bit Limited
CSD	Canonical (or Canonic) Signed Digit
DSP	Digital Signal Processing
E	Set of Graph Edges
G	Graph
GA	Genetic Algorithm
ID	Identification
IP	Infinite Precision
LMS	Least mean Square
MINIMAX	Minimize the Maximum Error
N-D	N-dimensions
SVD	Singular Value Decomposition
TSP	Traveling Salesman Problem
V	Set of Graph Vertices
X	Filter Pole

CHAPTER 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 been applied in the areas of radar signal processing, speech processing, communication, biomedical image processing, and computer vision. Increasingly, DSP algorithms and devices are being found in consumer products such as home theatre , music players, cell phones, etc.

An important area of DSP is digital filtering. It is a computational process, which transforms a signal represented by an input array of numbers to another signal represented by an output array of numbers, in order to alter the signal response characteristic according to some prescribed specification.

Digital filters can be applied in one-dimension (1-D), two-dimensions (2-D), and in general N-dimensions (N-D) and can be implemented on a general-purpose computer or special-purpose hardware. The throughput of a digital filter is the rate at which an input array can be transformed into a corresponding output array. For many applications a high throughput will be required.

A 1-D recursive digital filter can be characterized by its difference equation (1.1)

$$y(nT)=\sum_{i=0}^{N} a_i x(nT-iT)-\sum_{i=1}^{M} b_i y(nT-iT) \qquad (1.1)$$

or by its transfer function (1.2). Where x(nT) is the input signal, y(nT) is the

$$H(z) = \frac{\sum_{i=0}^{N} a(i) z^{-i}}{\sum_{i=0}^{M} b(i) z^{-i}} = \frac{A(z)}{B(z)} \qquad (1.2)$$

$$for\ stability\ \ B(z) \neq 0 \ \ and \ \ |z| \geq 1$$

output signal, n is a sequential sample number, T is the sample period and a_i, b_i are the filter coefficients [1]. The magnitude and phase of the filter at a particular frequency ω is given by the transfer function when $z = e^{j\omega T}$.

Filter design is the process determining the coefficient a_i's and b_i's of the transfer function such that the magnitude or phase of the frequency spectrum of the designed filter approximates some desired response. Without loss of generality we assume M=N.

The filtering operation is performed according to the difference equation (1.1). Present and past input and past output samples are multiplied by the filter coefficient a_i's and b_i's. These products are summed to arrive at the output sample. The maximum speed at which this operation can proceed is dominated by the time needed for multiplying the samples by the filter coefficients. Decreasing the time needed for this multiplication will improve the speed of the filter thus yielding higher throughput.

Binary multiplication is performed by a shift and add operation. The multiplier is repeatedly shifted one or more bit positions and added to a partial product according to the bit pattern of the multiplicand as shown in Fig. 1.1.

(1011)·(1010)							
			1	0	1	1	
		0	0	0	0		
	1	0	1	1			
0	0	0	0				
0	1	1	0	1	1	1	

Fig. 1.1 Shift-Add binary multiplication

While shift operations execute quickly, additions are slower and comprise the bulk of the multiplication time. Since an addition is required only when a 1 bit occurs in the multiplicand, a multiplicand with fewer 1 bits will take less time to be multiplied than a multiplicand with more 1 bits.

When performing a filtering operation the multiplier is chosen to be the input or output sample and the multiplicand is one of the filter coefficients. Filters designed to have coefficients comprising a small number of binary 1 digits are able to execute faster than filters having coefficients comprising more binary 1 digits.

1.1 Canonical Signed Digit Number System

A common method [2]-[6] method for decreasing the number of binary 1 digits and hence reducing the number of additions required during multiplication is to use the Canonical Signed Digit (CSD) number system which inherently has a large number of zero digits. It is based on the signed digit number systems [7] which allows individual digits to have a sign as well as a value.

$$digit \in \left\{ -\left\lfloor \frac{r}{2} \right\rfloor, \ldots, -1, 0, 1, \ldots, \left\lfloor \frac{r}{2} \right\rfloor \right\} \tag{1.3}$$

Generally the digits of these number systems are chosen as shown in (1.3) and can have any base. As a replacement of the binary system for high speed multiplication, the ternary number system where $r = 2$ is used. This allows the digits to have values of 0, 1 or -1. Typically the -1 digit may be written as $\bar{1}$ or as the letter n (for negative 1). Here, the $\bar{1}$ form will be used for clarity in equations, while the simpler n form will be used for long compilations of CSD numbers.

In this number system, the sign and value of the overall number is determined by the weighted sum of the signed digits as shown in (1.4).

$$value = d_0 d_1 d_{2,} \ldots, d_{N-1} = \sum_{i=0}^{N-1} d_i \times 2^{-i} \tag{1.4}$$

In multiplication, the shift and add operation of the binary number system is extended to include subtraction for the case when a digit has a value of -1. Subtraction and addition are comparable in terms of speed of execution so allowing -1 digits does not hinder multiplication time yet the extra freedom offers a great potential to increase the number of zero digits used to represent a given value.

The signed-digit number system is a redundant number system, since a given value may be represented by more than one sequence of digits. For example, $0.01 = 1 \times 2^{-2} = .25$ and $0.1\bar{1} = 1 \times 2^{1} - 1 \times 2^{-2} = -.25$ are two different representations with the same value.

However, for any given value with two or more redundant representations there will be only one representation where the N-digit signed-digit number follows the constraint of (1.5).

$$d_n \times d_{n+1} = 0 \quad for \quad 0 \le n \le N-2 \tag{1.5}$$

Such a number is said to be the canonical form of the signed-digit number or simply the CSD form. Following from (1.5) is the property that the number has no adjacent non-zero digits.

Another property of the CSD form is that it has the fewest number of non-zero digits among the redundant forms. An N bit number in CSD format is able to uniquely express every value of an N bit binary number but it will never have more (N+1)/2 non-zero bits. This makes it a very desirable form for high throughput filtering.

1.2 Limiting Non-Zero Bits in Coefficients

A method of further reducing the number of non-zero digits in a coefficient is to simply place an arbitrary limit on the the number of such digits allowed. Unfortunately, this reduces the available values in the number system resulting in a loss of granularity in coefficient choice thus ultimately limiting the quality of the filters which can be designed. However, using appropriate design methods, good filters can still be designed.

For example, the designer can opt to allow no more than 3 non-zero digits within a 16 digit CSD coefficient. Such non-zero-bit limited (bit limited) CSD numbers are still technically CSD numbers since they form a subset of the CSD number set but they can no longer represent all possible values within their range.

For example, the CSD number represented by $0\bar{1}0101010000000000$ would not be allowed if we were to set a non-zero bit limit of 3. Since this uniquely represents the value -0.3359375, such a value would not be available. The closest value that could be represented would be -0.328125 having a representation of $00\bar{1}0\bar{1}0\bar{1}000000000$ which, in this case, exhibits an approximate 2.3% error from the desired value.

Coefficients in this bit-limited format are guaranteed not to have more than the given limit of non-zero digits making them well suited as operands in high speed multiplication. However, filter designs become more difficult with fewer coefficient values to choose from.

1.3 Survey of Filter Design Methodologies with CSD Coefficients

Several design methodologies have been used for designing CSD Coefficient Filters.

1.3.1. Conversion

Simply designing a filter using infinite precision numbers and converting each of the filter's coefficients to bit-limited CSD numbers is problematic at best. Due to the poor granularity of bit-limited CSD numbers, each conversion could introduce a fair amount of error. It is likely that the accumulated errors of all coefficients will detrimentally effect the filter's response. In addition, for recursive filters, a formerly stable filter may become unstable. Consequently, this method is not often used.

1.3.2. Algebraic Design

Designing a filter completely within a bit-limited CSD number system using direct algebraic filter synthesis is not possible. Bit-limited CSD number systems are not closed under normal arithmetic operations such as addition or multiplication. For example, the sum of two bit limited CSD numbers may have more non-zero digits than the limit allows placing it outside the number system. Thus no bit-limited CSD algebra exists within which direct filter synthesis calculations can be performed.

1.3.3. Search and Optimization

Some form of search and optimization is often used to design these filters. Standard optimization algorithms, such as hill climbing. will get trapped in a local maximum of the multimodal search space of a filter design.

Integer programming has been used successfully for low order filters but it tends to become impractical for higher order filters [5] due to the search space becoming extremely large.

Simulated annealing has been shown to be effective in filter design but it suffers from high computational costs [8].

A computationally efficient method which has a parallel searching capability is the Genetic Algorithm [9]. It can handle large search spaces and has been shown to work with both recursive [2]-[4] and non-recursive [6] filter designs.

To date though, this method has been hampered by its inability to efficiently handle the CSD constraints. All solutions have introduced some form of random search into an otherwise robust search method. The resulting search methods are no longer pure Genetic Algorithms but hybrids of random search and GA principles.

In this dissertation a method is presented for applying a GA to this problem with a new coding technique that makes it possible to utilize the full potential of the genetic algorithm. Many examples and comparisons are included to demonstrate the effectiveness of this method.

1.4 Common Sub-expression Elimination

The throughput of the implementation of CSD-coefficient filters is determined by the number of additions required to implement the coefficient multiplication using a shift/add procedure. The number of these additions can be reduced by avoiding redundant calculations through the use of common sub-expression elimination.

In this dissertation a method is presented for transforming this problem into one similar to the well understood Traveling Salesman problem [9]. An example using a standard GA is included to demonstrate the effectiveness of this method.

1.5 Organization of this Dissertation

This dissertation covers several topics. Chapter 1 is a general introduction. Chapter 2 continues with a detailed examination of Genetic Algorithms. Chapter 3 looks at 1-D filter non-recursive design and Chapter 4 extends this to recursive filter design. Chapter 5 covers 2-D filter design and Chapter 6 looks at common sub-expression elimination

CHAPTER 2. GENETIC ALGORITHMS

2.1 Introduction

This chapter examines the basic operation of Genetic Algorithms including the essential operations of selection, crossover and mutation. The theoretical foundations are reviewed including the fundamental theorem of Genetic Algorithms, the building block theorem and implicit parallelism. Advancements and refinements to the basic operators. as well as techniques for managing GA difficulties, are examined.

2.2 Origin and Operation

Genetic algorithms are a class of computational methods that are modeled on the mechanisms of natural evolutionary genetics. The first rigorous study of GA principles was reported by John Holland in his book Adaptation in Natural and Artificial Systems published in 1975 [10]. This work has been subsequently extended by many others. They utilize methods that are similar to the methods found in natural selection to work. These methods operate on a population of problem solutions in an effort to find the fittest individual. It is hoped that this fittest individual is at or close to the optimal solution

The technique is based on the principles of survival of the fittest. Individuals in a population must compete with each other for a limited number of resources and ultimately for survival. The most successful individuals will more likely survive and thus mate. The less successful individuals will be less likely to survive and thus will produce fewer offspring than a successful individual. This means that each succeeding generation

will more likely inherit genes from the successful individuals than from the unsuccessful ones.

Genetic algorithms borrow heavily from this natural evolutionary process to allow solutions to real world problems to evolve over many succeeding generations. Therefore, in order to artificially use the mechanisms of natural selection on a search and optimization problem, it is necessary to formulate the problem in line with that observed in nature. The solution to the problem must be expressed as a character string called a chromosome and there must also be a fitness function that can be applied to this string to determine the individual's fitness.

For example, in the design of a bridge the chromosomes may represent the size and weight of certain beams. The fitness function would calculate the strength to weight ratio of a bridge built with these beams. The GA would then be searching for the beams with the highest strength to weight ratio.

Within a population of individual solutions to a problem there are more fit and less fit solutions. Individuals are chosen from this population for mating depending upon their fitness score. Two chosen individuals are mated by cutting and splicing their chromosomes to form a new chromosome. The offspring will thus inherit features from both parents. In this way the good characteristics of a population are transferred to each succeeding population while the bad characteristics are not. This results in the most promising areas of the search space being explored. If the problem has been coded into chromosomes properly, the population will converge to an optimal solution.

The GA is robust since the only requirement for applying it to a particular problem is that the solution can be expressed as a chromosome and there exists a fitness function to evaluate this chromosome's fitness. No other information about the problem is needed. This means that a GA can be applied to a wide variety of problems including some of those where there are no other solution techniques.

A GA does not actually find a solution to a problem but instead creates new and better solutions based on existing solutions. Fortunately, the coding of a solution into a string as required by the GA allows initial solutions to be randomly generated. While GAs are not guaranteed to find the global optimum, they are good at finding good solutions in a reasonable amount of time.

2.3 Classes of Search Techniques

Genetic algorithms are a type of optimization search technique. Search techniques in general, as illustrated in Fig. 2.1, can be grouped into three broad classes [10] calculus based, enumerative and random search.

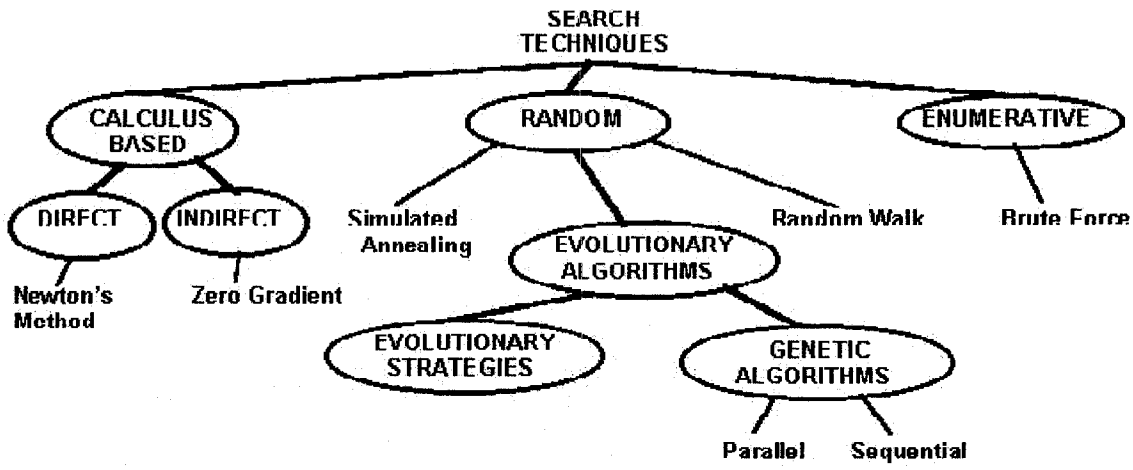

Fig. 2.1 Search Techniques

Calculus based methods include direct and indirect. Indirect is the search for the peaks of maxima by finding zero of the gradient. Direct techniques are those such as Newton's method. Random methods include simulated annealing, evolutionary strategies, genetic algorithms and the simple random walk through the search space. Enumerative methods are the brute force methods where all the solutions in the whole search space are generated.

```
BEGIN SIMPLE GENETIC ALGORITHM
Randomly generate initial population
Compute the fitness of each individual in the population
WHILE (NOT finished) DO
     // produce new generation
     FOR (population_size / 2) DO
          Reproduction:
            - Copy two parent individuals randomly selected from
            current generation using probability biased to favor the
            fittest
            - Mate these copies by randomly splitting and recombining
            them to form two new offspring
          Crossover:
            - Remove the two original parents from current generation
            - Place the two offspring into new generation
      END FOR
      Designate new generation to be current generation
      Randomly change some randomly selected individuals
      Mutation:
      Compute the fitness of each individual
      IF (population has converged) THEN
           finished
      END IF
END WHILE
END GENETIC SIMPLE ALGORITHM
```

Fig. 2.2 Simple Genetic Algorithm

The basic simple genetic algorithm. as described by Goldberg [10], is shown in Fig. 2.2. It is composed of three operators: reproduction, crossover, and mutation. These are applied to a population of individuals each of which is composed of a coded solution to

the problem. The problem must have a fitness function that can be applied to each solution to determine its merit or fitness. Applying the operators to a population results in a new population with hopefully increased average fitness as well as an increase in the fitness of the fittest individual. This process is repeated until there is no further fitness increase at which time the solution is said to have converged. The cycle is shown more graphically in Fig. 2.3.

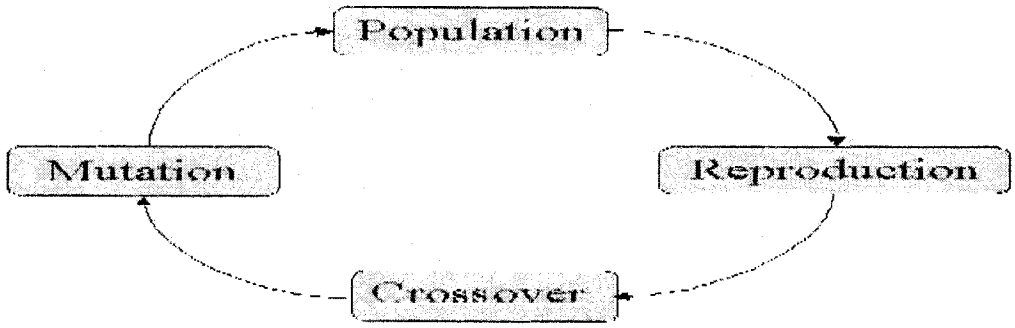

Fig. 2.3 Simple Three Operator Genetic Algorithm

2.4 Problem Coding

The problem solution must be encoded in a form suitable for use with the reproduction, crossover, and mutation operators. The solution must be configured as a string of characters called a chromosome after its biological analogue. The characters, also called genes, may be taken from any fixed alphabet that may be used to represent the problem. Early work [11] suggested that a longer string is superior to a shorter string so a low cardinality alphabet is superior to a higher one. Later work however [12],[13], shows that some problems do better with a higher cardinality chromosome. The lowest cardinality that will work is an alphabet of only two characters so a binary string is often used to

code problems. In biology, chromosomes are made of strings of four different proteins so the biological alphabet has a cardinality of four.

2.4.1. Fitness Function

The problem must also have an objective or fitness function that takes a solution chromosome and returns a value that represents a figure of merit for this solution. A higher figure of merit indicates a superior or fitter solution.

The genetic algorithm works on a population of chromosome strings each of which represent a solution to the problem. To begin, an initial population of solution strings is chosen by some method such as simple random choice. This population is applied to the objective function and each solution is assigned a fitness value. The population is then subjected to the three operators of the genetic algorithm.

2.4.2. Reproduction

Reproduction is the process of randomly selecting chromosome strings biased by their fitness value and making new chromosome strings out of them. A chromosome string with a higher fitness value will have a higher probability of being chosen for reproduction. This is analogous to the natural selection process whereby an organism that is more fit has a higher chance of surviving to reproduce.

Implementing a biased random selection in algorithmic form can be accomplished in many ways. A common method is to create a biased roulette wheel where each chromosome string is assigned a slot on the wheel but the slots are not uniform in size.

Instead each is sized in proportion to its corresponding chromosome's fitness. When the wheel is spun the chromosome string with the highest fitness will have the greatest possibility of being chosen. An entire population is chosen this way. Some may be chosen more than once. This method was used throughout this dissertation.

These selected individuals are used to create new strings through the crossover operation. In single point crossover, as shown in Fig. 2.4, a crossover point is chosen at a random position between 1 and the string length-1 which in this case is the third position. Two new chromosome strings are now created by dividing the initial chromosome strings into two sections each at the crossover point and appending the first half of the first chromosome string to the second half of the second chromosome string and vice versa.

```
First parent = ABC|DEFG          substring1 = ABC
Second parent = abc|defg         substring2 = defg

offspring1 = substring1 + substring2  =  ABCdefg
offspring2 = substring2 + substring1  =  abcDEFG
```

Fig. 2.4 Single Point Crossover

The mutation operator is now applied to the population. This operator applies a random alteration to the value of a chromosome string position. This alteration occurs with very small probability so that the likelihood of a bit actually mutating is very small. Mutation is necessary to replace genetic information that may have been lost or may never have existed in the original population.

2.5 Example Problem

The simplicity and power of genetic algorithms can best be demonstrated with the step by step solution of a simple problem. The problem shown in Table 1 and Table 2 is so

simple that it was solved by hand by Goldberg [9] using nothing more than a coin to generate random numbers. Yet even with this simplicity the improvements possible in only a single generation are readily apparent.

The problem goal is to maximize the function $f(x)=x^2$ where x is permitted to vary between 1 and 31 which is coded as a 5 bit binary number. To keep things manageable it is run with a population of only 4 individuals.

String Number	Initial Population (Chromosome)	x Value (as Integer)	Objective Function Value $f(x)=x^2$	Probability of Selection	Expected Count	Actual Count (From Roulette Wheel)
1	0 1 1 0 1	13	169	0.14	0.58	1
2	1 1 0 0 0	24	576	0.49	1.97	2
3	0 1 0 0 0	8	64	0.06	0.22	0
4	1 0 0 1 1	19	361	0.31	1.23	1
Sum			1170	1.00	4.00	4.0
Average			293	0.25	1.00	1.0
Maximum			576	0.49	1.97	2.0

Table 1 GA Example: Initial Population

String Number	Mating Pool After Reproduction	Mate (Randomly Selected)	Crossover Site	New Population	x Value (Integer)	$\frac{F(x)}{x^2}$
1	0 1 1 0 | 1	2	4	0 1 1 0 0	12	144
2	1 1 0 0 | 0	1	4	1 1 0 0 1	25	625
3	1 1 | 0 0 0	4	2	1 1 0 1 1	27	729
4	1 0 | 0 1 1	3	2	1 0 0 0 0	16	256
Sum						1754
Average						439
Maximum						729

Table 2 GA Example: Second Population

In the initial population the average fitness is 293 and the best individual fitness is 576. After one generation the average fitness has increased to 439 and the best individual fitness has increased to 729.

2.6 Analysis of the Simple Three Operator Genetic Algorithm

The previous section demonstrated the mechanics of the simple three operator genetic algorithm. By mimicking nature, a procedure has been developed that seems to provide useful results. However, to understand why it works requires a theoretical analysis.

This theoretical basis for the GA's operation was first worked out by Holland [11] and later embellished by Goldberg [9]. It provides a thorough, generalized analysis of the operation of GA's. This analysis is often called the schema theorem, but its real importance is underscored by its other commonly used name: the fundamental theorem of genetic algorithms.

2.6.1. Schemata

In order to analyze the workings of genetic algorithms it is necessary to have some method of describing a subset of a string. A subset can be described using the similarity template called a schema (in plural form they are called schemata). This is a string composed of the letters of the given chromosome alphabet plus a special symbol, usually *, that is used to indicate a don't care position in the chromosome string. A schema can be thought of as a pattern matching devise as it matches the particular chromosome string

if in every location a 1 in the schema matches a 1 in the chromosome string and a 0 in the schema matches a 0 in the chromosome string and a * matches either.

For example, consider the case of a binary string of length 5. The schema * 1 0 1* matches only the chromosome strings 01010, 01011, 11010 and 11011.

2.7 The Fundamental Theorem of Genetic Algorithms

The schemata theorem or the fundamental theorem of genetic algorithms is one of the most important properties related to genetic algorithms. In order to analyze the operation of genetic algorithms it is necessary to be able to count the schemata present within a population of strings and determine which grow and which decay during each generation. This is done by considering the affect of reproduction, crossover and mutation on a particular schema. The objective is to quantifying the GA's simultaneous manipulation of a very large number of schemata.

2.7.1. Notation

In this analysis it is considered, without loss of generality, that strings are composed of characters from the binary alphabet V= {0,1}. For notational purposes strings will be referred to by capital letters and individual characters in the string by lower case letters. These individual characters may be subscripted by their position in the string as in $S=s_1s_2s_3s_4$. Populations of individual strings will be denoted as $\boldsymbol{P}_j$, j = 1, 2,. . .,n . A population existing at a time or generation t and will be denoted as $\boldsymbol{P}(t)$, where the boldface denotes a population rather than a string.

In order to describe the schema contained in individual strings and populations the three letter alphabet $V+ = \{0,1,*\}$ will be used. The additional character * is used as a don't care or wild card symbol which will match either a 0 or a 1 at any particular string position.

For a string of length l there are 3^l schemata which are defined over it. In general, for an alphabet of cardinality j there are $(j+1)^l$ schemata.

The order of a schema is denoted by $O(H)$, and is the number of fixed (as opposed to wild card) positions that it has. For example, a schema of length 5 and order 3 is 1 1 1 * *

A schema H will also have a defining length denoted by $\delta(H)$. This is the distance between the first and last fixed string position. For example the schema 1 * 1* * has a defining length $\delta(H)=2$ because the first fixed position is 1 and the last fixed position is 3 and 3-1 = 2. Similarly, the schema * 1 * * * * would have a defining length of $\delta(H)=2-2=0$.

2.7.2. Analysis of the Fundamental Theorem Of Genetic Algorithms

The previously discussed notation will be used to discuss the effect of reproduction on the expected number of schemata in the population. Suppose at a given time t there are m examples of a particular schema H contained within population $A(t)$. This can be written as $m = m(h, t)$. During reproduction a string A gets selected for copying with a probability of $p_i=f_i/\sum f_i$. Once the non-overlapping population of size n is chosen with

replacement from the population *A(t)*, there should be *m(H, t + 1)* representatives of the schema H in the population at time $t + 1$ as given by (2.1).

$$m(H,t+1)=m(H,t)\cdot n\cdot\frac{f(H)}{\Sigma f_i} \tag{2.1}$$

If $f(H)$ is the average fitness of the strings representing schema H at time t and since the average fitness of an entire population may be written as $\overline{f}=\Sigma f_i/n$ the reproductive schemata growth equation may be written as:

$$m(H,t+1)=m(H,t)\frac{f(H)}{\overline{f}} \tag{2.2}$$

This shows that a particular schema grows as the ratio of the average fitness of the schemata to the average fitness of the population. Schemata with fitness values greater than the population average will be present in greater numbers in the next generation while schemata with fitness values lower than the population average will be present in lesser numbers in the next generation. This operation is carried out with every schema in a particular population A in parallel. All schemata in the population grow or decay according to their schemata averages under the operation of reproduction. This simple operation of reproduction on the strings in a given population results in many more operations being performed on many more schemata.

Individual schema will increase or decrease in number from generation to generation depending upon their fitness values. The exact rate of this growth or decay can be determined from the Schema's difference equation. Suppose that a particular schema

remains above average by about $c\bar{f}$ with c a constant. The schema difference equation can then be rewritten as in (2.3).

$$m(H,t+1)=m(H,t)\frac{(\bar{f}+c\bar{f})}{\bar{f}}=(1+c)\cdot m(H,t). \tag{2.3}$$

Starting at $t=0$, and assuming c is constant from generation to generation, (2.4) is obtained:

$$m(H,t+1)=m(H,0)\cdot(1+c)^{t}. \tag{2.4}$$

This equation has the same form as the compound interest equation. It is a geometric progression which shows that reproduction allocates exponentially increasing or decreasing numbers of schemata. It can also be seen that for a schema that is above or below average, reproduction will allocate exponentially increasing or exponentially decreasing offspring in subsequent generations.

The reproduction operation ensures that subsequent generations will have exponentially increasing numbers of schemata that are fit and exponentially decreasing numbers of schemata that are not fit. This operation serves to concentrate the existing good solutions while eliminating some of the less good solutions. It does nothing to find new and possibly better solutions. This is the job of the crossover operator.

Crossover allows for a randomized exchange of information between strings. It creates new strings while preserving the allocation strategy pursued by reproduction. The result is an exponentially increasing or decreasing collections of particular schema throughout the population.

During the operation of crossover some schema are more likely to survive than others. For example, the schema * * * 1 0* * can only be destroyed if the crossover point is chosen so that it falls between the 1 and the zero at the 4th position. On the other hand the schema * 1 * * * * 0 can be destroyed if the crossover point falls between the 1 and the 0 at any of positions 2, 3, 4, 5 or 6. The likelihood of survival is based on the defining length of the schema and is given by the crossover survival probability p_s .

The lower bound of the crossover survival probability p_s can be calculated under simple crossover as $p_s=1-\delta(H)/(l-1)$ since the schema is likely to be destroyed whenever a crossover site within the defining length is selected from the $l-1$ possible sites. Since the crossover is itself performed by random choice with a probability p_c at a particular mating, the survival probability may be given by (2.5).

$$p_s \geq 1-p_c \cdot \frac{\delta(H)}{l-1} \tag{2.5}$$

As can be seen, this expression reduces to the previous expressions whenever $p_c=1$.

In order to calculate the number of a particular schema H expected in the next generation under the combined effect of reproduction and crossover, the combined expression in (2.6) is used.

$$m(H,t+1) \geq m(H,t) \cdot \frac{f(H)}{\bar{f}} \left[1-p_c \cdot \frac{\delta(H)}{l-1} \right] \tag{2.6}$$

The third and final operation performed by the simple genetic algorithm is mutation. This is the random alteration of a single string position with probability p_m . For a

schema H to survive, all of its fixed positions must survive. Therefore, a single character of a schema survives with the probability $1-p_m$. Since each of the mutations is statistically independent, a particular schema survives when each of the $o(H)$ fixed positions within the schema survives. Multiplying the survival probability $1-p_m$ by itself $o(H)$ times is the probability of surviving mutation as $(1-p_m)^{o(H)}$. Since the probability p_m is very small $(p_m<<1)$ this can be approximated as $1-o(H)(p_m)$.

Combining this with the previous expression gives the following relation describing the number of copies of a particular schema that can expect to survive into the next generation under all three operations of reproduction, crossover and recreation.

$$m(H,t+1)\geq m(H,t)\cdot\frac{f(H)}{\bar{f}}\left[1-p_c\cdot\frac{\delta(H)}{l-1}\right]-o(H)p_m \qquad (2.7)$$

This shows that short, low order, above average schemata receive exponentially increasing trials in subsequent generations. These schemata are often referred to as building blocks and their exponentially increasing importance to the outcome of a GA is often referred to as the building block theorem.

2.8 Implicit Parallelism

The results of the preceding analysis has been used to define a phenomenon called implicit parallelism [9] that explains the power of GAs. Here a GA, with a population of n, processes n chromosomes for each generation. During these n calculations the GA is shown to be actually processing n^3 schemata. This exponential increase in the processing capacity is credited with giving GAs their superior abilities.

2.9 Advanced Techniques

The simple genetic algorithm, as discussed so far, is the basic starting point for all genetic algorithms. Much research has gone into extending and improving the algorithm. Many of these advanced techniques have come to be used routinely.

2.9.1. Crossover Techniques

The simple GA performs crossover by making a single cut at the same location in each of the two parent chromosomes. This cut occurs somewhere between the first gene and the last gene. The cut sections are then exchanged to form two offspring. This method is somewhat simplistic and tends to destroy the building blocks that contain widely spaced genes. For this reason researchers have devised many new crossover techniques often using more than one cut point.

The effectiveness of multiple-point crossover was investigated and it was found [14] that while 2-point crossover gives an improvement, adding further crossover points reduces the performance of the GA. As additional crossover points are added, the search becomes more random because building blocks are more likely to be disrupted. The problem space is searched more thoroughly at the expense of greatly increased search time. In the extreme, the search simply becomes a random search.

2.9.1.a) 2-Point Crossover

2-point crossover has chromosomes arranged in loops by joining their ends together. Two cuts are made in the loop and the resulting segments are exchanged. From this it can

be seen that 1-point crossover is just a special case of the more general 2-point crossover where one of the cut points is fixed as falling between the last and first position. This would account for the increased performance seen when 2-point crossover is used. It is no more disruptive than 1-point crossover since they both have 2 cut points and 2-point crossover does not always destroy building blocks with widely spaced genes as is the case for 1-point crossover. That is, a chromosome treated as a loop with no beginning and no end can contain more building blocks, since they are able to wrap around at the end of the string. It is generally considered that 2-point crossover is superior to 1-point crossover.

2.9.1.b) Uniform Crossover

Another form of crossover is the n-point uniform crossover where the number of points *n* varies dynamically with each mating. In this method, a randomly generated crossover mask is used to determine which genes of an offspring come from which parent. Each gene in the first offspring is created by copying the corresponding gene from one or the other parent according to the crossover mask. Where there is a 1 in the mask, the gene is copied from the first parent, and where there is a 0 in the mask, the gene is copied from the second parent. The process is repeated with the parents exchanged to produce the second offspring. A new crossover mask is randomly generated for each pair of parents. Offspring, therefore, contain a mixture of genes from each parent. The number of effective crossing points, while not fixed, will average $L/2$ (where L is the length of the chromosome).

For example, suppose we let the first parent be an arbitrary 10-bit binary string represented by the sequence ABCDEFG where A represents the most significant bit and G the least significant bit and similarly we let the second parent be represented by abcdefg then they would mate using uniform crossover as follows:

1. A random crossover mask is chosen, (e.g. 0011101001)
2. Wherever the mask has a 1, choose the corresponding character from the first parent
3. Wherever the mask has a 0, choose the corresponding character from the second parent
4. The resulting offspring is combined as shown in Fig. 2.5
5. Reverse parents and repeat steps 2, 3, & 4 yielding the second offspring (A B c d e F g)

```
Parent      A B C D E F G          a b c d e f G
Mask        0 0 1 1 1 0 1          0 0 1 1 1 0 1
Result      - - C D E - G          a b - - - f -

Combined Offspring      a b C D E f G
```

Fig. 2.5 Uniform Crossover

2.9.2.Crossover Comparisons

Research on the different methods of crossover [15] has shown that uniform crossover produces long defining length schema which are less likely to be disrupted than those produced by 2-point crossover. But while the short defining length schemata of 2-point crossover are more likely to be disrupted, the overall amount of schemata disruption is lower.

Under 2-point crossover the defining length, and not the order of the schemata, determines the likelihood of its disruption. Under a uniform crossover, the likelihood of disruption of a given schema is based only on its order and not its defining length. This means that under a uniform crossover, the ordering of genes within a chromosome is completely irrelevant and it eliminates the need for re-ordering operators such as inversion (see Section 2.9.6 Inversion and Reordering, on page 30). Also, since the positioning of genes is immaterial there is no need to worry about coding the chromosome in such a fashion so as to create good building blocks.

In another study [16] an extensive comparison of 1-point, 2-point, multi-point and uniform crossover operators was performed. Theoretical analysis was performed in terms of positional and distributional bias on several problems. The findings indicated that there was only about a 20% difference in speed between the slowest and fastest techniques. From these results the choice of a crossover operator would seem to be relatively unimportant.

Other analysis [17] of crossover has shown that due to reduced productivity, 2-point crossover will perform poorly when the population has largely converged. Productivity is the ability of a crossover operator to produce new chromosomes that are different, thereby sampling new points in the search space. If two similar chromosomes undergo 2-point crossover, then the exchanged segments are likely to be identical causing the offspring to be identical to their parents. Under uniform crossover, this is less likely to happen.

Many problems benefit when a few parents are not crossed over at all but passed on intact. Many algorithms apply a high probability crossover rate which performs crossover most of the time but once in a while randomly passes the parents through with performing crossover.

Another operator called elitism takes the most fit individual and automatically passes it through, intact, to the next generation. This is done to make sure the best solution so far is not lost.

2.9.3. Other Crossover Techniques

A new 2-point crossover operator has been reported [18] where the offspring are checked after crossover and if they are found to be identical to their parents then crossover is repeated using two new crossover points. When tested, this new operator was found to perform slightly better than uniform crossover. This new 2-point crossover is best only when there is a large population, and that for small populations uniform crossover is best due to the increased disruption that it causes.

Several methods have been described [19]-[22] that vary the probability of crossover occurring at a particular string position. The crossover probabilities themselves become part of the chromosome so that the GA dynamically adjusts the sites that should be favored for crossover. With the crossover probabilities included as part of the chromosome, they are crossed over and passed on to descendants allowing the GA to learn which building blocks are more important than other genes in the chromosome.

2.9.4.Crossover Conclusion

Although the crossover method used has only a modest affect on the overall performance of a GA, one of the top ranked crossovers is uniform crossover. As well, under uniform crossover the ordering of genes within a chromosome is completely irrelevant. For these reason all GA's used in the examples in this dissertation will employ elitism and uniform crossover.

2.9.5. Mutation

Normally, mutation occurs with low probability and functions as a background operator [14]. It is included to allow for the searching space that may otherwise be precluded by the converging chromosomes as the genetic information is discarded during crossover. The exact amount of mutation necessary is somewhat open to debate. Too little mutation and useful schemata that are not currently in the population can never be found while too much mutation will cause the GA to degenerate into a random search.

A study [23] to determine the optimum parameters for GAs found that mutation plays a larger role than previously thought. Another study [24] compared crossover and mutation and found that each operator contained characteristics not found in the other but that each is simply a form of a more general exploration operator that modifies schemata based on available information. As the population converges, mutation plays an increasingly important role while the role of crossover diminishes.

Although it has a low probability of use and it is sometimes seen as nothing more than a background operation, mutation plays a very important part in a GA solution. Changes in the mutation rate will affect the performance more than the changes to the crossover parameters [23]. However adjusting for the optimum mutation rate is difficult as long as extremes are avoided. There is a fairly broad range of values that seem to work well in most situations.

2.9.6. Inversion and Reordering

For a GA to work effectively, the building block theorem requires that the genes be arranged in a particular order in the chromosome. To accomplish this, techniques for reordering the positions of genes have been developed. One of these techniques is inversion [11] which reverses the order of genes between two randomly chosen positions within the chromosome. To keep track of a gene's position within the chromosome some auxiliary positioning information must be maintained with each chromosome. Gene reordering is an attempt to create chromosome codings with better evolutionary potential [9].

When reordering is used, the search space is greatly expanded, since the GA is searching for a solution to the original problem and simultaneously searching for the optimum gene ordering. The extra search time spent on ordering might be better spent on the original problem.

When a uniform crossover is used, the ordering of genes is irrelevant so reordering would have absolutely no effect. Since in this dissertation uniform crossover will be used exclusively, reordering will not be performed.

2.9.7. Deception

The building block principle states that over succeeding generations there will be an increase in the number of chromosomes containing schemata that are also found in the global optimum until eventually these schemata will crossover into a single individual and the global optimum will be found. But on certain GA deceptive problems, this does not occur and schemata that are not in the global optimum increase in numbers faster than those that are.

This phenomenon has been studied in depth by many [9],[26],[27] and it has been shown that the number of chromosomes containing a particular schema will increase if the schema's fitness is higher than the average fitness of all schemata in the population. The difficulty arises if the average fitness of schemata which are not contained in the global optimum is greater than the average fitness of those which are. This class of problem is deemed to be deceptive. A GA will usually, but not always, have difficulty solving a deceptive problem.

A problem that is deceptive with one chromosome coding format may not be with a different format. Therefore the coding format can be crucial to the success or failure of applying a GA to a specific problem.

2.9.8. Epistasis

A given gene's contribution to the overall fitness of an individual may be conditional on the value of other genes in the chromosome. Such a gene would be called epistatic. In general, the amount of co-dependency among genes is termed epistasis and occurs in nature on a regular basis. For example, bats have a gene that gives them their keen hearing and another to make high pitched chirps. Either of these genes alone would not increase a bat's fitness but together they form a sonar system and have a major impact on fitness.

The amount of epistasis which occurs can vary from none to severe. An example of a problem with no epistasis is the counting of ones task where the task is to maximize the number of 1s in the binary string. In this case each gene (bit) either has a value of 1 and contributes to the fitness or has a value of 0 and doesn't. An example of moderate epistasis is the plateau function where the fitness is 1 if all the bits in a chromosome are set to 1, and zero otherwise. Here the genes do interact but only for the global optimum when they all must be 1. Severe epistasis is where the genes interact in numerous and complex ways. An example of this would be a scheduling problem where the availability of a resource is dependent on other schedules.

Problems which exhibit no epistasis or even mild epistasis are easily solved by techniques like hill-climbing and do not require a GA [28]. When the problem has moderate to severe epistasis though, observations indicate that the other, simpler techniques do not perform as well as GAs. There are ample examples[29] of GA's being successfully applied in domains of high epistasis.

It is possible to lower the epistasis of a given problem by changing the chromosome coding. This type of problem recoding is demonstrated [30] in a bin-packing problem. The chromosome of the converted problem has less epistasis than the original problem.

Having a good chromosome coding scheme is the key to having low epistasis in s given problem.

2.9.9. Chromosome Alphabets

The fundamental theorem of Genetic Algorithms suggests that the strength of a genetic algorithm lies in the implicit parallelism of the operation since the algorithm works on many schemata at the same time. Because of this it was initially believed [9] that a binary alphabet having the largest number of schemata of any alphabet, was the best. However later interpretations of schemata [12] shows that high-cardinality alphabets contain more schemata than binary alphabets and therefore offer better performance. Others [12],[13] report that some problems do better with a higher cardinality chromosome. Therefore if a particular problem can be more precisely coded in a non-binary format then that format will perform better than the ill-fitting binary format.

2.9.10. Operator Parameters

The exact values of operator parameters, such as mutation rate, population size etc., that lead to the greatest GA performance is hard to pin down. As noted in Section 2.9.5, mutation has a greater effect than crossover toward the end of a search and vice-versa.

An analysis [32] of the interacting roles of population size, crossover rate and mutation rates shows that these values are not critical as long as they are not extreme. Increasing population size makes the GA converge in fewer generations but each generation takes longer to calculate so the overall time needed changes little.

Extreme values are to be avoided but most problems will typically respond well to values for mutation rate m_r of $0.05 \leq m_r \leq 0.1$ and crossover rate c_r of $0 \leq c_r \leq 0.2$ and a population size in the hundreds.

2.9.11. Problem Constraints and Invalid Chromosomes

Many problems have so many constraints that certain chromosome values may violate one or more of these constraints. It is also possible that the number of discrete solutions to the problem is not a power of 2, so it can not be expressed exactly as a binary number. In this case, the binary number will be larger than the number of available solutions resulting in a number of chromosome values that are not valid.

The ideal solution is to use domain knowledge to prevent invalid chromosomes from being produced in the first place. But domain specific knowledge is not always available

and requiring it reduces the robustness of a pure GA. But when it is available, it eliminates the time used to process and identify invalid values.

Without domain specific knowledge, there is no guarantee that such invalid codes will not arise. Crossover and mutation will explore the whole range of the chromosome space including valid and invalid areas. To deal with this, a number of solutions have been proposed [14].

The first and easiest is to simply discard an invalid chromosome and produce a different valid chromosome. This requires that all offspring be checked for validity after they are generated and discarding any that are found to be invalid. This will result in discarding fatal chromosomes which may contain some great schemata values intermixed among the fatal gene values. This destroys the hard won schemata already found and if too many get discarded the GA will degenerate into a random search.

Another method is to simply assign an invalid chromosome a low fitness value. This makes the most sense in terms of the GA process, for indeed, such a chromosome is not fit at all. But this interferes with the essential GA processing for if the low fitness value is too low then the result is the same as discarding them, while if it is not low enough then some invalid solutions may be produced.

Another method for dealing with invalid chromosomes is chromosome remapping where invalid chromosomes are mapped onto valid ones.

2.9.11.a) Chromosome Remapping

There are currently two types of chromosome remapping in use now. The first, fixed remapping, takes a particular invalid value and either changes it to some other particular valid value or processes it as if it were that other value. While the remapping mechanism is simple and it essentially removes all invalid chromosomes from the search space, it has the disadvantage that in the remapping potentially good schemata are completely discarded and replaced by new random schemata. If this happens too frequently the GA will degenerate into a random search.

Random remapping tries to fix this shortcoming by remapping an invalid value to a randomly chosen valid value. This eliminates the representational bias problem but completely discards all parental inheritance information, instead opting to choose a random offspring when an invalid offspring is encountered. Again, if too many chromosomes get randomly remapped the GA will degenerate into a random search.

2.10 Conclusion

Genetic algorithms are an important tool to be used against the problems encountered in search and optimization. It has properties that make it superior to other techniques for many broad classes of problems. It can work when there is absolutely no problem domain knowledge other than the objective function.

CHAPTER 3. 1-D FILTER DESIGN

3.1 Application of the GA to Filter Design

Using a genetic algorithm to design a filter is simply a matter of coding a chromosome to represent the filter coefficients a_i and b_i of equations 1.1 and 1.2. The GA cycle of Fig. 2.3 starts with a population of filters whose chromosomes are randomly chosen. Each chromosome is decoded into its a_i and b_i filter coefficients and its magnitude response is determined by evaluating the transfer function of Equation 1.2 with $z=e^{j\omega T}$ over some frequency range of interest. The magnitude of the transfer function at each frequency is compared to the desired magnitude at that frequency and an error value is calculated. This value is squared and summed with the square of the error values at the other frequencies of interest to form a least mean square (LMS) error value. A fitness value for this filter is formulated as the inverse of the LMS error value. Alternately, it is sometimes desirable to minimize the maximum error in the magnitude response (MINIMAX) so the fitness value can be based on this calculation instead of the LMS value.

Some of the filters within this population are selected and mated to form a new population of filters which is likely to be more fit than the parent population. The cycle is repeated until an individual filter is found with a fitness that exceeds some desired fitness level.

3.2 CSD Chromosome Coding

Each filter in the GA search space gets coded as a chromosome. The a_i and b_i coefficient values of the filter are first coded into a string of symbols or digits to form a

partial chromosome. These are concatenated in a predetermined order to form the overall filter chromosome.

For instance, suppose $A_i = d_{1,i}d_{2,i}...d_{N,i}$ represents a coded string of digits for the N a_i coefficients and $B_i = d_{1,i}d_{2,i}...d_{M,i}$ represents the coded string of digits for the M b_i coefficients of a filter whose difference equation is (1.1). This filter's chromosome would be the concatenation of the A_i and B_i partial chromosome codes in the form $A_1A_2 ... A_NB_1B_2 ... B_M$.

Since the filter is to have coefficients in CSD format, the coding of the a_i and b_i coefficients into A_i and B_i digit strings must also preserve the CSD format. The usual approach is to code each coefficient as a sequence of ternary signed digit strings. However, a ternary signed digit string is not necessarily a CSD string as it may violate the canonical constraints of Equation (1.5) as well as any non-zero bit limit imposed by the design requirements. This coding is problematic for the genetic algorithm operators.

3.3 Effects of Crossover and Mutation on CSD Values

Parent1	00**1**|**0**0$\bar{1}$
Parent2	00**0**|**1**00
Offspring1	00**1**|**1**00

Fig. 3.1 Invalid CSD Crossover

The problem is that the genetic algorithm operation of crossover and mutation are not closed for CSD numbers in ternary digit form. For example, the two CSD parents in Fig. 3.1 undergoing single point crossover at the point shown produce an offspring that is not a CSD number.

Fig. 3.2 shows how mutation of a CSD number can have a similar result. These examples are for pure CSD numbers, for CSD numbers with non-zero bit limiting the problem is worse.

Original $00\bar{1}000\mathbf{10}10000\bar{1}01$
Mutated $00\bar{1}000\mathbf{1\bar{1}}10000\bar{1}01$

Fig. 3.2 Invalid CSD Mutation

Several approaches to maintaining the coefficients in CSD format have been proposed and all involve some form of modification of the genetic algorithm.

One approach is to fix any chromosomes which have non-CSD coefficients by converting them to CSD numbers [33]-[35]. This has been accomplished both as a straight conversion and by converting to floating point and back. While this type of approach has been shown to work, it destroys many schemata introducing serious search inefficiencies.

The implicit parallelism of a GA stems from its ability to coalesce fit schemata found over the generations into a very fit population. Converting a partial chromosome sequence into another numerically equivalent sequence destroys all of the schemata of the original sequence. To the genetic algorithm this new sequence represents a random collection of schemata causing a major disruption to the search procedure.

Another approach that has been used with some success is to modify the GA [36] so that non CSD chromosomes are never produced. In this method an offspring resulting from crossover or mutation is checked for proper CSD formatting. If the offspring fails this test it is discarded and another crossover or mutation is performed using the original

source chromosome(s). The hope is that since crossover points and mutation parameters are chosen at random another try might produce a valid result. If after several attempts it still has not produced a valid result then the operation is aborted and new chromosomes are chosen from the population.

While this may arguably be less disruptive to the GA than the previous approach of throwing out hard won schema patterns it can still be very disruptive. Under this approach there may exist parents where the likelihood of successful crossover is slim or non-existent. For example, if the parents shown in Fig. 3.3 are from a design with a non-zero bit limit of 2 then there are very few crossover points where one offspring would be valid and none where both would be valid under uniform crossover.

$parent A$ $1 0 \bar{1} 0 0 0 0 0 0 0 0 0 0 0 0 0 0$
$parent B$ $0 0 0 0 0 0 0 0 0 0 0 0 0 \bar{1} 0 1$

Fig. 3.3 Unlikely Mating Partners

The parents shown in Fig. 3.4 will never produce CSD offspring under uniform crossover. While other types of crossover exist they all have similar cases of unlikely and impossible mating partners.

$parent A$ $0 \bar{1} 0 1 0 1 0 1 0 1 0 1 0 \bar{1} 0$
$parent B$ $\bar{1} 0 1 0 1 0 \bar{1} 0 1 0 1 0 \bar{1} 0 1$

Fig. 3.4 Impossible Mating Partners

The operation of discarding offspring because they are not in proper CSD form also discards the offspring's particular mix of schemata blocking search paths that might prove useful.

3.4 Effects of GA Disruption

In each of the previous approaches the GA mechanism is compromised. Either a chromosome with completely new schemata is introduced or a combination possibly containing some good schemata is discarded. This disruption tends to make the GA lose some of its implicit parallelism and causes it to deteriorate toward a random search.

Since the typical coefficient search space is so large, a random search is very inefficient. For example, a 10th order filter using 16-bit CSD coefficients with a maximum of 3 non-zero digits has a search space of approximately 10^{38} different filters.

An exhaustive search of this space on a modern desktop PC would take about 10^{27} centuries to complete at 100 filters per second. A random search for a suitable filter would be quicker than an exhaustive search for a particular filter but even so, it will still average many centuries.

Whenever the GA mechanism is disrupted, inherited schemata are replaced by random sequences and the GA search deteriorates into a random search. The time needed to find a suitable filter also deteriorates toward the time needed for a random search. Since a random search is so slow, only a little deterioration can result in a large increase in search time.

3.5 Proposed Design Technique

In order to keep the GA from slipping toward a random search, we must avoid modifying the GA mechanism and avoid any procedure where invalid offspring must be fixed. To accomplish this a CSD chromosome coding that is closed under the operations

of crossover and mutation is used. Since the commonly used ternary string chromosome coding scheme is not suitable another must be devised.

To see how this is accomplished it is necessary to look at the fundamental attributes of a CSD number. Typically a CSD number has the attributes shown in (3.1).

$$V = d_0, d_1, \ldots, 0_{p_n-1}, \bar{z}_{p_n}, 0_{p_n+1}, \ldots, d_{L-2}, d_{L-1} \tag{3.1}$$

Here d_i is any digit at position i, 0_i is a zero digit at position i and $\bar{z}_i$ is a non-zero digit at position i. Each CSD number has a value V, a length of L digits of which N are non-zero digits $\bar{z}$ which are located at position p_n $(1 \le n \le NZ_{max})$ having a sign $S \in \{+, -\}$.

In this form the canonical constraints of equation (1.5) can be written as (3.2).

$$d_{p_n-1} = 0 \ , \ d_{p_n+1} = 0 \quad where \quad (1 \le n \le NZ_{max}) \tag{3.2}$$

For designs with an arbitrary limit on the number of non-zero digits allowed, the value of NZ_{max} can be chosen. For pure CSD values $NZ_{max} = (L+1)/2$ is the inherent CSD limit.

From this it can be seen that the problem of maintaining canonical form under crossover and mutation is a result of the position p_n in the canonical constraints of equation. So rather than basing the chromosome on a ternary encoded value which allows an unrestricted number of positions p_n , the coding itself should be based on the NZ_{max} non-zero digit positions p_n. This forces the above constraints on the chromosome making it closed under all operations including crossover and mutation.

As before, the filter coefficients a_i and b_i will be coded into partial chromosomes $A_i = d_{1,i}d_{2,i}...d_{N,i}$ and $B_i = d_{1,i}d_{2,i}...d_{M,i}$. These will be concatenated into the complete chromosome as $A_1A_2... A_NB_1B_2... B_M$.

Each A_i or B_i partial chromosome is coded as a string of NZ_{max} genes g in the form $g_1 g_2 \cdots g_{NZ_{max}}$, where each gene g designates both the position P and sign S of a non-zero digit.

The key to doing this is to abandon the binary or ternary gene strings normally used in genetic algorithms and allow genes to take on values from a much larger symbol set. As discussed in Section 2.9.9, chromosomes with a high cardinality work well for those problems suited to them. CSD filter coefficients are just such a problem since the use of small symbol sets has many drawbacks.

Consequently, we allow each gene to take on one of $2L$ symbol values where L is the number of CSD digits in each a_i or b_i filter coefficient. The $2L$ symbols are used to designate the position occupied by a non-zero digit within a CSD coefficient as well as the sign of that digit. The actual symbols used are immaterial.

As an example of this coding, consider a 7 digit CSD coefficient upon which we impose an upper limit NZ_{max} of 3 non-zero digits. Since $L=7$, we need $2L$ or 14 symbols to code each digit. If we choose 0 to 9 and A to D for the symbols we can assign them as follows: the symbols 0 to 6 indicate that a +1 exists in the CSD coefficient at position 1 to 7 respectively and the symbols 7 to D indicate that a -1 exists at position 1 to 7 respectively. Since our CSD coefficient has a maximum of 3 non-zero digits we will need

three gene digits, one for each non-zero position. Table 3 shows the symbol conversions for this case.

Symbol	0	1	2	3	4	5	6	7	8	9	A	B	C	D
Non-zero Digit Sign	+	+	+	+	+	+	+	−	−	−	−	−	−	−
Non-Zero Digit Position	1	2	3	4	5	6	7	1	2	3	4	5	6	7

Table 3 Conversion Table for CSD Length L=7

Suppose a CSD coefficient was represented by the three digit partial chromosome 2AD. From Table 1 the 2 would signify a +1 in position 3, the A would signify a -1 in position 4 and the D would signify a -1 in position 7. Therefore the CSD coefficient in question would be $00\bar{1}00\bar{1}$.

Under this coding scheme, CSD coefficients with fewer than the maximum allowed non-zero digits simply have two non-zeros occupying the same position. For example, for a 7 bit CSD with a maximum of 3 non-zero digits the partial chromosome 744 would indicate a -1 at position 1, a +1 at position 5 and another +1 also at position 5 resulting in the CSD $\bar{1}000100$ The second non-zero is simply ignored since a non-zero already exists at that location.

The canonical adjacency constraint of (1.5) is addressed in much the same fashion. If a partial chromosome digit indicates that a non-zero should be positioned adjacent to a non-zero from an earlier digit then it is simply ignored. This has the advantage of mapping any potentially non-canonical sequences to canonical sequences with fewer than the maximum number of non-zeros.

For instance, a 7 bit CSD with a maximum of 3 non-zero digits with a partial chromosome 743 would normally decode to $\bar{1}001100$ which would violate the CSD constraint. But by ignoring the 3 in 743 since it causes the problem, we get $\bar{1}001000$ which is a valid CSD number with fewer than NZ_{max} non-zero digits.

Sequences with fewer than NZ_{max} non-zero digits are very desirable so by having two mechanisms which map to this space it tends to be searched more often. This increases the likelihood of finding CSD coefficients with even fewer than the maximum allowed non-zeros.

Since this chromosome coding has no sequences which violate either the CSD constraint or a NZ_{max} constraint, it is completely closed under the operation of crossover and mutation. In addition, it uses some search space redundancy to advantage by forcing the GA to search more often in the most desirable search space.

3.6 Non-Recursive Filter Design Example

A sixteenth order linear phase low pass FIR filter using 16 bit CSD coefficients each limited to a maximum of 3 non-zero digits was designed using the following target frequency response.

$$|H(e_{j\omega})|=\begin{cases} 1 & 0\le\omega\le 1 \\ 0 & 2\le\omega\le\pi \end{cases} \tag{3.3}$$

The genetic algorithm used the parameters shown in Table 4 and each design took approximately five minutes to complete on a 2.0 Ghz Pentium computer.

Population Size	Number of Generations	Crossover rate	Mutation Rate
500	500	.95	.05

Table 4 Genetic Algorithm Parameters

3.7 Comparison with Existing Design

To ascertain the effectiveness of the new method, a detailed comparison was performed. The same type filter was designed using Matlab's optimal least square filter design function firls. The resulting coefficients, which are accurate to 15 significant digits and are considered to be infinite precision (IP), were converted to the closest 16 digit CSD value with a maximum of 3 non-zero digits. The square error of this filter for both IP format coefficients and CSD format coefficients is compared with the filter obtained using the new method design. Also compared is an existing design [36] which utilizes the same CSD format coefficients and target frequency response.

For each filter the square error was calculated at 16384 points equally spaced points along the frequency spectrum shown in (3.4)

$$Square\ Error=\sum_{i=0}^{16383}(M_a(\pi i/16383)-M_t(\pi i/16383))^2 \tag{3.4}$$

where $M_a(\omega)$ and $M_t(\omega)$ are the actual and target magnitude responses at frequency ω respectively.

For the filter with infinite precision coefficients, the square error is 7.1306×10^{-2} and a plot of the frequency response is shown in Fig. 3.5.

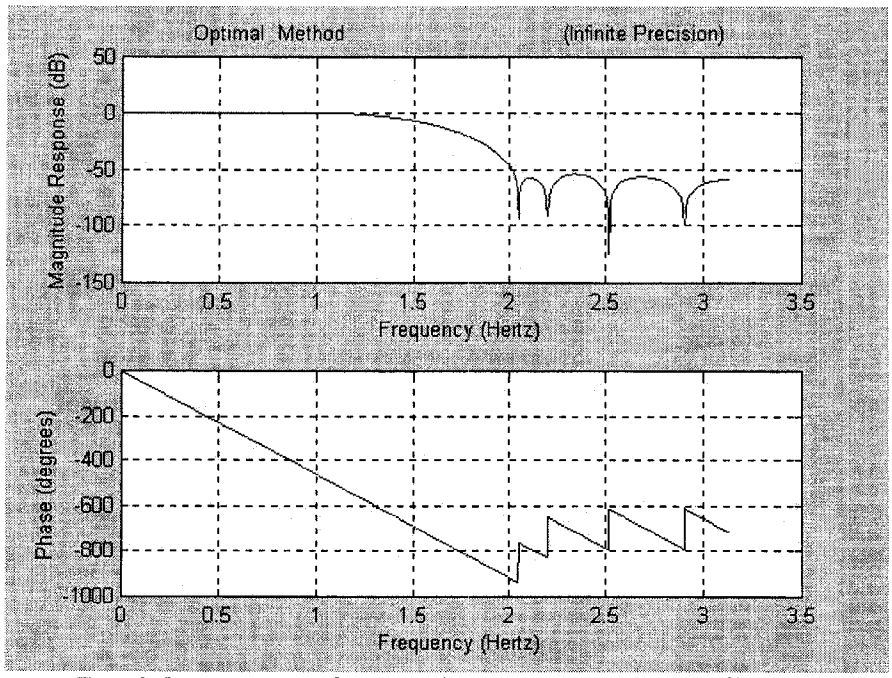

Fig. 3.5 Response of Optimal Design using IP Coefficients

Coefficient	Infinite precision (IP)	Infinite precision converted to CSD	
		CSD format	decimal format
a0	-0.00461277865458	00000000n0n01000	-0.0046386719
a1	-0.00826443773636	0000000n000n0001	-0.0082702637
a2	0.01183883759729	00000010n0000100	0.0118408203
a3	0.02875781248896	000001000n0n0000	0.0288085938
a4	-0.02035661622626	000000n0n0n00000	-0.0205078125
a5	-0.08033874590455	0000n0n00n000000	-0.0800781250
a6	0.02849524621471	00000100n0100000	0.0283203125
a7	0.30905318429880	00101000n0000000	0.3085937500
a8	0.46878831168206	01000n0000000001	0.4687805176
a9	0.30905318429880	00101000n0000000	0.3085937500
a10	0.02849524621471	00000100n0100000	0.0283203125
a11	-0.08033874590455	0000n0n00n000000	-0.0800781250
a12	-0.02035661622626	000000n0n0n00000	-0.0205078125
a13	0.02875781248896	000001000n0n0000	0.0288085938
a14	0.01183883759729	00000010n0000100	0.0118408203
a15	-0.00826443773636	0000000n000n0001	-0.0082702637
a16	-0.00461277865458	00000000n0n01000	-0.0046386719

Table 5 Coefficients from Optimum Design

For the filter with infinite precision coefficients converted to CSD format, the square error is 8.1343×10^{-2} and a plot of the frequency response is shown in Fig. 3.6.

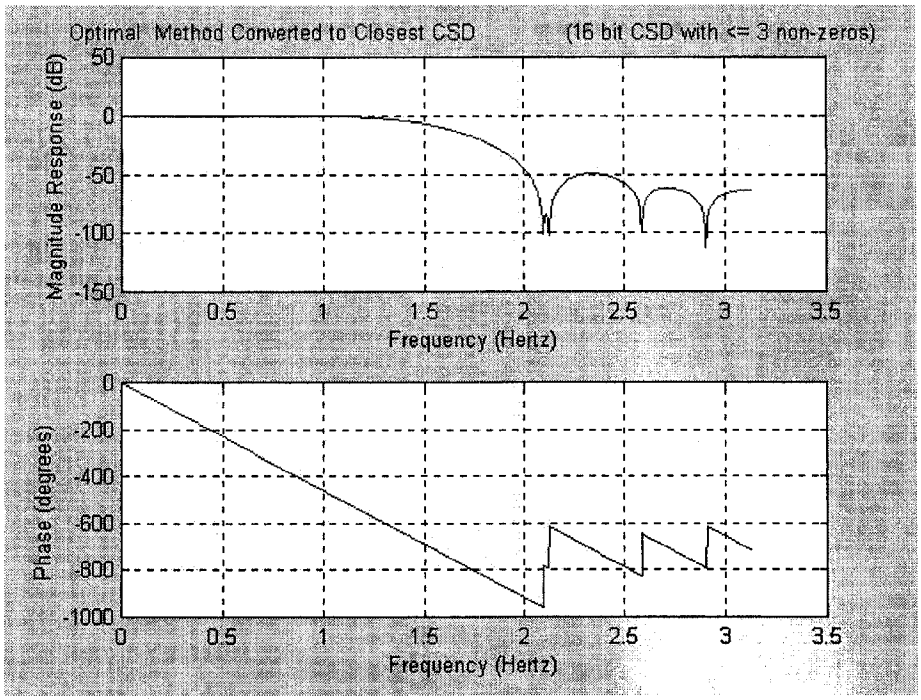

Fig. 3.6 Response of Optimal Design Converted to CSD Coefficients

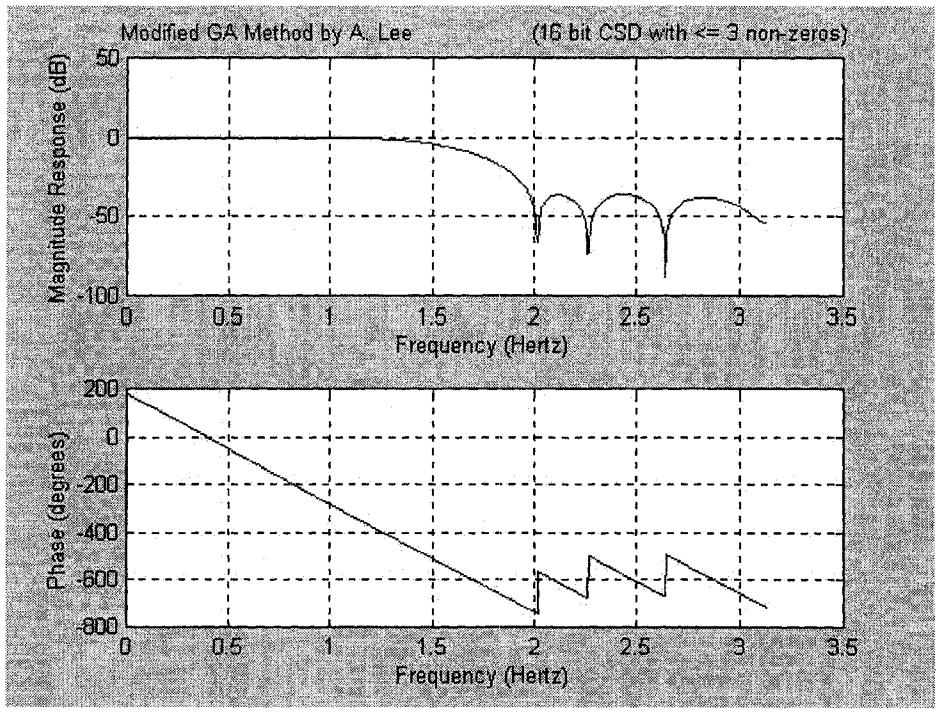

Fig. 3.7 Response of Filter from [36]

Coefficient	Decimal Value	CSD Representation (where n = -1)
a0	-0.0039062500	00000000n0000000
a1	0.0117187500	00000010n0000000
a2	0.0039062500	0000000010000000
a3	-0.0351562500	00000n00n0000000
a4	-0.0019531250	000000000n000000
a5	0.0820312500	0000101010000000
a6	-0.0009765625	0000000000n00000
a7	-0.3125000000	00n0n00000000000
a8	-0.5000000000	0n00000000000000
a9	-0.3125000000	00n0n00000000000
a10	-0.0009765625	0000000000n00000
a11	0.0820312500	0000101010000000
a12	-0.0019531250	000000000n000000
a13	-0.0351562500	00000n00n0000000
a14	0.0039062500	0000000010000000
a15	0.0117187500	00000010n0000000
a16	-0.0039062500	00000000n0000000

Table 6 Coefficients for Filter from [36]

For the filter design of [36] the square error is 98.124 $x10^{-2}$. The coefficients of this filter are given in Table 6 and a plot of the frequency response is shown in Fig. 3.7

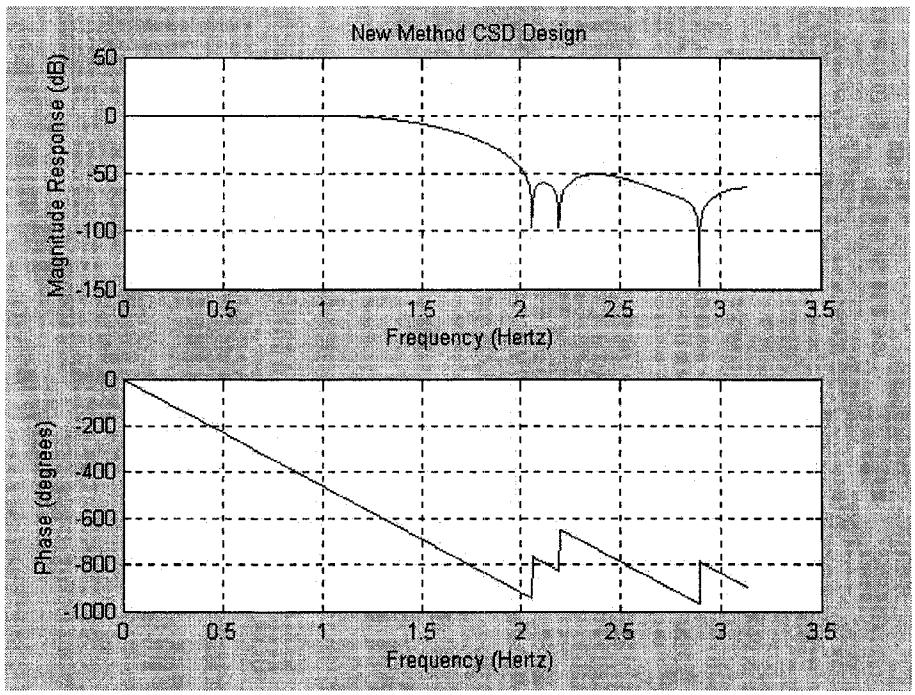

Fig. 3.8 Response of design by proposed new method

For the filter designed using the proposed new method the square error is 7.2157 $x10^{-2}$. The coefficients of this filter are given inTable 7 and a plot of the frequency response is shown in Fig. 3.8.

Coefficient	Decimal Value	CSD Representation (where n = -1)
a0	-0.0046386719	00000000n0n01000
a1	-0.0079040527	0000000n00000n01
a2	0.0114746094	00000010n000n000
a3	0.0288085938	000001000n0n0000
a4	-0.0205078125	000000n0n0n00000
a5	-0.0800781250	0000n0n00n000000
a6	0.0283203125	00000100n0100000
a7	0.3085937500	00101000n0000000
a8	0.4687805176	01000n0000000001
a9	0.3085937500	00101000n0000000
a10	0.0283203125	00000100n0100000
a11	-0.0800781250	0000n0n00n000000
a12	-0.0205078125	000000n0n0n00000
a13	0.0288085938	000001000n0n0000
a14	0.0114746094	00000010n000n000
a15	-0.0079040527	0000000n00000n01
a16	*-0.0046386719*	*00000000n0n01000*

Table 7 Coefficients of Filter from Proposed GA Method

A Performance comparison summary is shown in Table 8. The filters are compared relative to the optimal IP filter's error which is the lowest possible error.

Approach	**Absolute Error**	**Relative Error**
Infinite precision	0.071306	1
IP converted to CSD	0.081343	1.141
Lee [36]	0.98124	13.76
Proposed GA Method	0.072157	1.012

Table 8 Comparison Summary

3.8 Comparison of Optimum Solution with CSD Conversion

To determine the performance of the proposed method and to validate the bit limited CSD approach, a series of filters were designed for comparison. A filter designed using the proposed method is compared with a design obtained by converting the infinite precision coefficients of an optimal design to their nearest CSD counterparts. This was repeated 8 times with the maximum number of allowable non-zero digits in each of the 16 digit CSD coefficients ranging from 1 to 8 non-zero digits. All designs were for a 20th order non-recursive filter with the target frequency response of (3.3). The GA designs used the parameters given in Table 4 and the optimum design with infinite precision coefficients used the Matlab firls function. CSD conversions were made to the closest valued CSD with the required maximum number of non-zero digits. The time taken by the GA to complete the designs was aproximately 5 minutes on a 2 Ghz Pentium computer.

When the infinite precision coefficients are converted to the 16 bit CSD format the largest values have only 14 significant digits. A 14 digit CSD without any non-zero bit limiting can have, at most, 7 non-zero digits. In this case though, none of the converted coefficients has more that 6 non-zero digits so the results for 7 and 6 allowable non-zero digits are the same as those for 7 . Therefore, the results for 8 and 7 maximum allowable non-zero digits are the same as that for 6 maximum allowable non-zero digits and are not presented separately.

For each filter the square error was calculated at 200 equally spaced points along the frequency spectrum shown in (3.5)

$$Square\ Error = \sum_{i=0}^{199} (M_a(\pi i/199) - M_t(\pi i/199))^2 \qquad (3.5)$$

where $M_a(\omega)$ and $M_t(\omega)$ are the actual and target magnitude responses at frequency ω respectively.

For the optimum filter with infinite precision coefficients the square error is 4.4383 x10^{-5} .The frequency response is shown in Fig. 3.9 and the coefficients are given in Table 9.

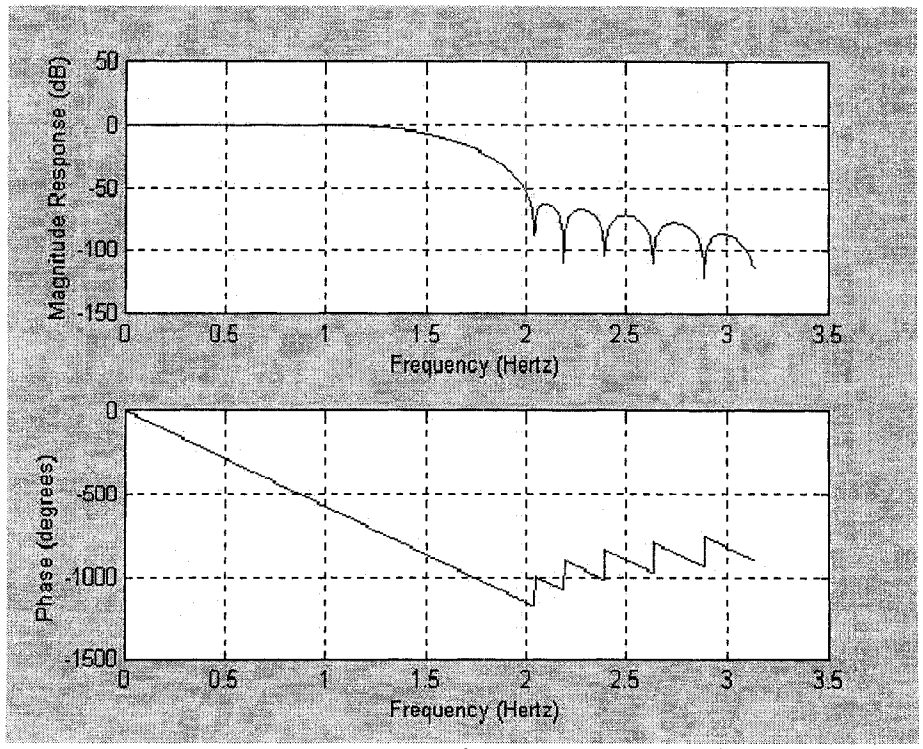

Fig. 3.9 Response using Infinite Precision Coefficients

The response of the filter with CSD coefficients converted from IP with maximum 6 non-zero digits is shown in Fig. 3.10 and the coefficients are listed in Table 10. This filter has a square error of 4.4690×10^{-5} .

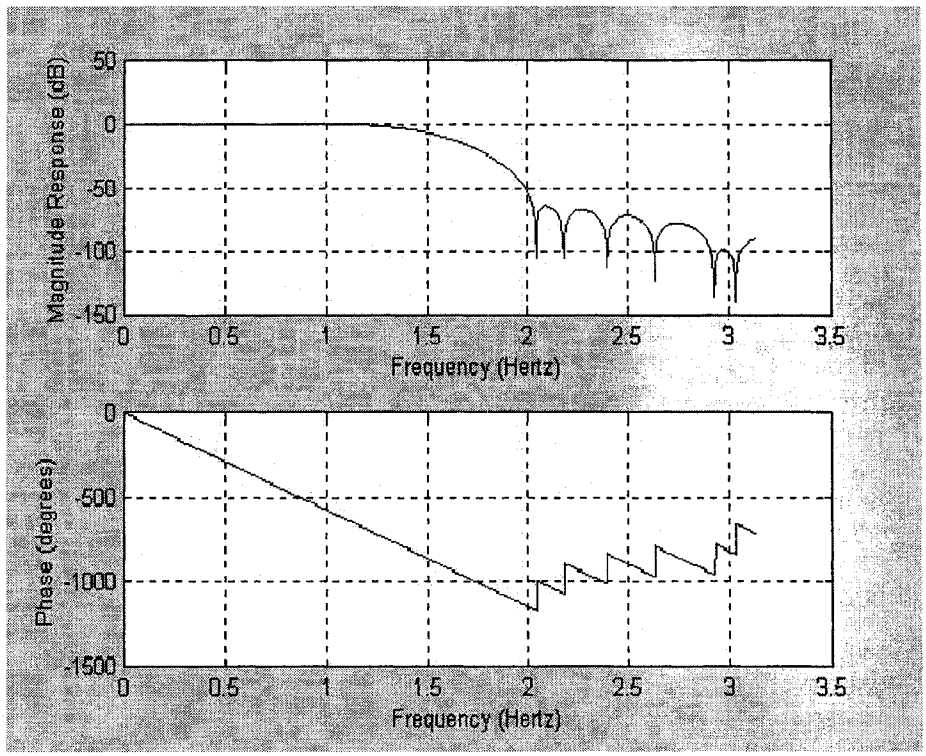

Fig. 3.10. Response of Filter with CSD Coefficients Converted from IP with a Maximum of 6 Non-Zero Digits

Coefficient	
a0	0.00174526230891
a1	0.00247787716625
a2	-0.00597187090745
a3	-0.01095329466689
a4	0.01280672911904
a5	0.03225811726060
a6	-0.02079199250683
a7	-0.08381600086404
a8	0.02730877450506
a9	0.31021651105280
a10	0.47017097295796
a11	0.31021651105280
a12	0.02730877450506
a13	-0.08381600086404
a14	-0.02079199250683
a15	0.03225811726060
a16	0.01280672911904
a17	-0.01095329466689
a18	-0.00597187090745
a19	0.00247787716625
a20	0.00174526230891

Table 9 Infinite Precision Coefficients

6 Max. Non-Zero Digits	Converted from Optimum Design		GA Design	
	Decimal	CSD (n=-1)	Decimal	CSD (n=-1)
a0	0.0017395020	000000000100n001	0.0017395020	000000000100n001
a1	0.0024719238	0000000001010001	0.0024719238	0000000001010001
a2	-0.0059814453	0000000n01000n00	-0.0059814453	0000000n01000n00
a3	-0.0109558105	000000n01010n001	-0.0109558105	000000n01010n001
a4	0.0128173828	00000010n0100100	0.0128173828	00000010n0100100
a5	0.0322570801	0000010000100001	0.0322570801	0000010000100001
a6	-0.0207824707	000000n0n0n0n00n	-0.0208129883	000000n0n0n0n0n0
a7	-0.0838012695	000n0101010010n0	-0.0838012695	000n0101010010n0
a8	0.0273132324	00000100n000000n	0.0273132324	00000100n000000n
a9	0.3102111816	001010000n0n0101	0.3102111816	001010000n0n0101
a10	0.4701843262	01000n00010n000n	0.4701538086	01000n00010n00n0
a11	0.3102111816	001010000n0n0101	0.3102111816	001010000n0n0101
a12	0.0273132324	00000100n000000n	0.0273132324	00000100n000000n
a13	-0.0838012695	000n0101010010n0	-0.0838012695	000n0101010010n0
a14	-0.0207824707	000000n0n0n0n00n	-0.0208129883	000000n0n0n0n0n0
a15	0.0322570801	0000010000100001	0.0322570801	0000010000100001
a16	0.0128173828	00000010n0100100	0.0128173828	00000010n0100100
a17	-0.0109558105	000000n01010n001	-0.0109558105	000000n01010n001
a18	-0.0059814453	0000000n01000n00	-0.0059814453	0000000n01000n00
a19	0.0024719238	0000000001010001	0.0024719238	0000000001010001
a20	0.0017395020	000000000100n001	0.0017395020	000000000100n001

Table 10 Coefficients with a Maximum of 6 Non-Zero Digits

The response of the filter with CSD coefficients from the proposed GA method with a maximum of 6 non-zero digits is shown in Fig. 3.11 and the coefficients are listed in Table 10. This filter has a square error of 4.4551 $x10^{-5}$.

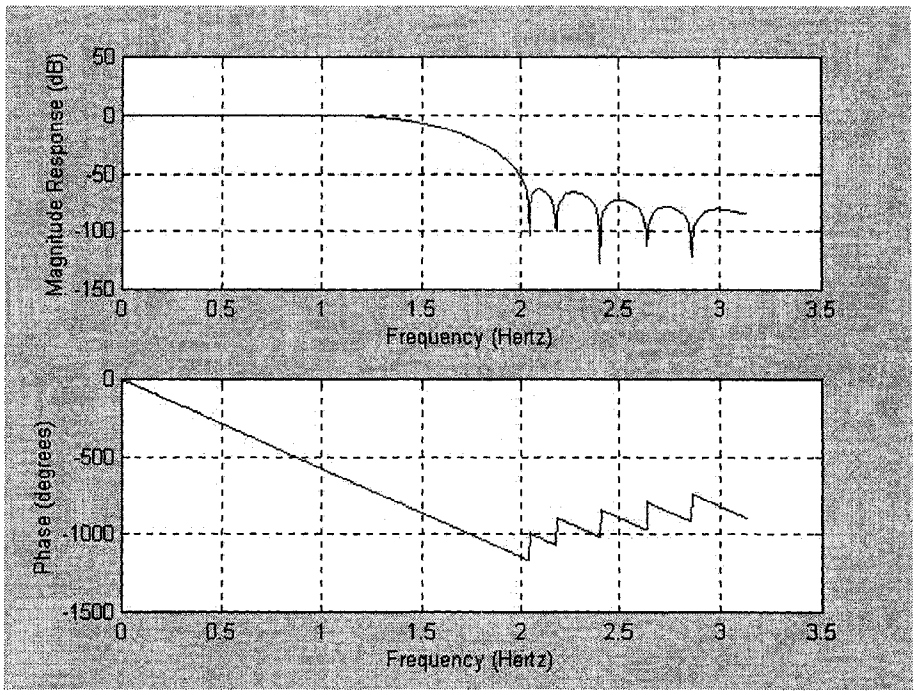

Fig. 3.11 Response of Filter with CSD Coefficients Designed by GA with Maximum 6 Non-Zero Digits

5 Max. Non-Zero Digits	Converted from Optimum Design		GA Design	
	Decimal	CSD (n=-1)	Decimal	CSD (n=-1)
a0	0.0017395020	000000000100n001	0.0017395020	000000000100n001
a1	0.0024719238	0000000001010001	0.0024719238	0000000001010001
a2	-0.0059814453	0000000n01000n00	-0.0059814453	0000000n01000n00
a3	-0.0109558105	000000n01010n001	-0.0109863281	000000n01010n000
a4	0.0128173828	00000010n0100100	0.0128173828	00000010n0100100
a5	0.0322570801	0000010000100001	0.0322875977	0000010000100010
a6	-0.0207824707	000000n0n0n0n00n	-0.0207824707	000000n0n0n0n00n
a7	-0.0838623047	000n010101000100	-0.0838623047	000n010101000100
a8	0.0273132324	00000100n000000n	0.0273132324	00000100n000000n
a9	0.3102416992	001010000n00n0n0	0.3102416992	001010000n00n0n0
a10	0.4701843262	01000n00010n000n	0.4701843262	01000n00010n000n
a11	0.3102416992	001010000n00n0n0	0.3102416992	001010000n00n0n0
a12	0.0273132324	00000100n000000n	0.0273132324	00000100n000000n
a13	-0.0838623047	000n010101000100	-0.0838623047	000n010101000100
a14	-0.0207824707	000000n0n0n0n00n	-0.0207824707	000000n0n0n0n00n
a15	0.0322570801	0000010000100001	0.0322875977	0000010000100010
a16	0.0128173828	00000010n0100100	0.0128173828	00000010n0100100
a17	-0.0109558105	000000n01010n001	-0.0109863281	000000n01010n000
a18	-0.0059814453	0000000n01000n00	-0.0059814453	0000000n01000n00
a19	0.0024719238	0000000001010001	0.0024719238	0000000001010001
a20	0.0017395020	000000000100n001	0.0017395020	000000000100n001

Table 11 Coefficients with a Maximum of 5 Non-Zero Digits

The response of the filter with CSD coefficients converted from IP with a maximum of 5 non-zero digits is shown in Fig. 3.12 and the coefficients are listed in Table 11. This filter has a square error of 4.4959×10^{-5} .

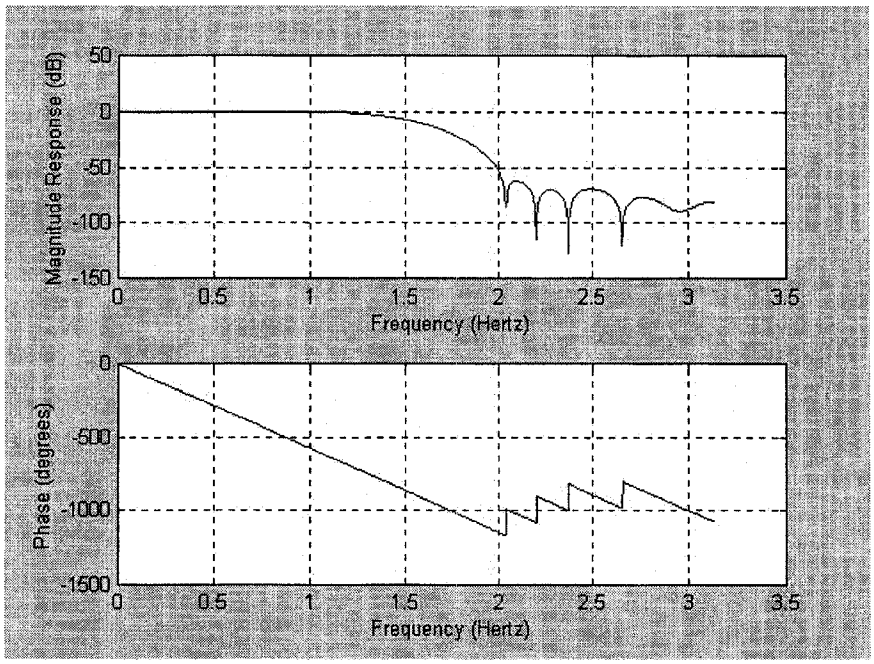

Fig. 3.12 Response of Filter with CSD Coefficients Converted from IP with a Maximum of 5 Non-Zero Digit.

The response of the filter with CSD coefficients from the proposed GA method with a maximum of 5 non-zero digits is shown in Fig. 3.13 and the coefficients are listed in Table 11. This filter has a square error of 4.4621×10^{-5} .

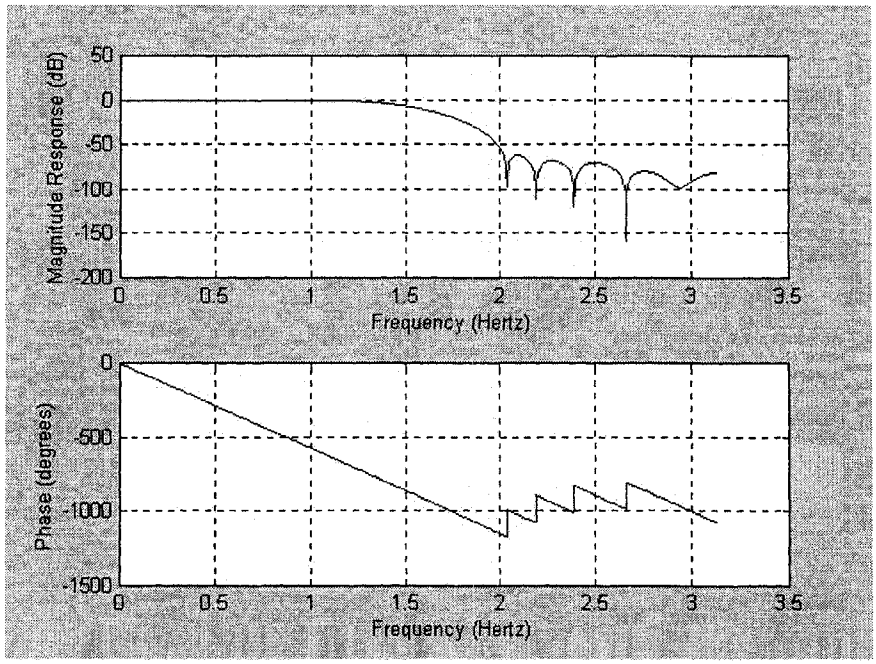

Fig. 3.13 Response of filter with CSD Coefficients Designed by proposed GA method with a Maximum 5 of Non-Zero Digits

The response of the filter with CSD coefficients converted from IP with a maximum of 4 non-zero digits is shown in Fig. 3.14 and the coefficients are listed in Table 12. This filter has a square error of 5.291×10^{-5} .

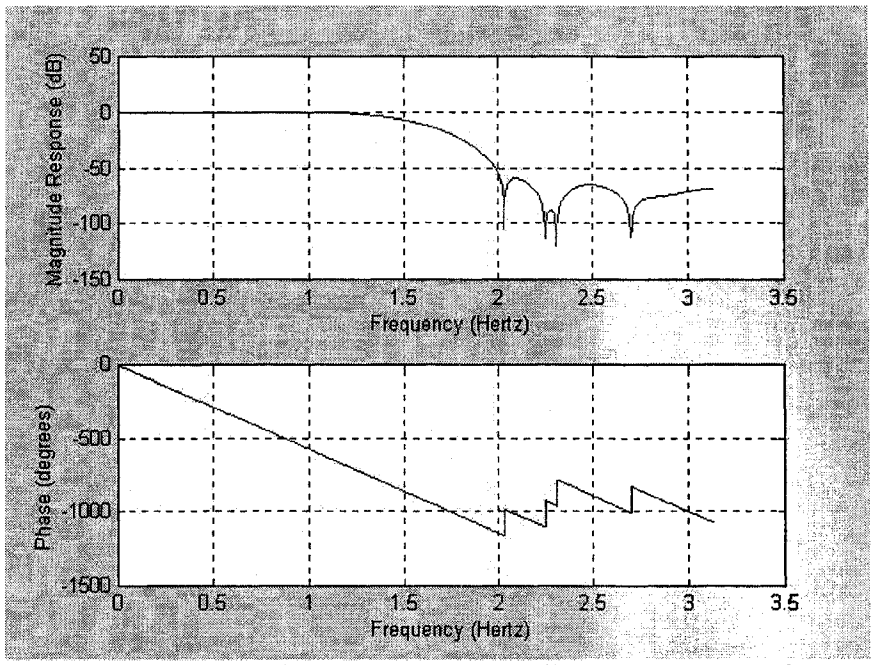

Fig. 3.14 Response of filter with CSD coefficients converted from IP with maximum 4 non-zero digits.

4 Max. Non-Zero Digits	Converted from Optimum Design		GA Design	
	Decimal	CSD (n=-1)	Decimal	CSD (n=-1)
a0	0.0017395020	000000000100n001	0.0017395020	000000000100n001
a1	0.0024719238	0000000001010001	0.0025024414	0000000001010010
a2	-0.0059814453	0000000n01000n00	-0.0059814453	0000000n01000n00
a3	-0.0109863281	000000n01010n000	-0.0109863281	000000n01010n000
a4	0.0128173828	00000010n0100100	0.0128173828	00000010n0100100
a5	0.0322570801	0000010000100001	0.0323791504	0000010000100101
a6	-0.0207519531	000000n0n0n0n000	-0.0207519531	000000n0n0n0n000
a7	-0.0839843750	000n010101000000	-0.0839843750	000n010101000000
a8	0.0273132324	00000100n000000n	0.0272521973	00000100n0000n01
a9	0.3103027344	001010000n00n000	0.3103027344	001010000n00n000
a10	0.4702148438	01000n00010n0000	0.4702148438	01000n00010n0000
a11	0.3103027344	001010000n00n000	0.3103027344	001010000n00n000
a12	0.0273132324	00000100n000000n	0.0272521973	00000100n0000n01
a13	-0.0839843750	000n010101000000	-0.0839843750	000n010101000000
a14	-0.0207519531	000000n0n0n0n000	-0.0207519531	000000n0n0n0n000
a15	0.0322570801	0000010000100001	0.0323791504	0000010000100101
a16	0.0128173828	00000010n0100100	0.0128173828	00000010n0100100
a17	-0.0109863281	000000n01010n000	-0.0109863281	000000n01010n000
a18	-0.0059814453	0000000n01000n00	-0.0059814453	0000000n01000n00
a19	0.0024719238	0000000001010001	0.0025024414	0000000001010010
a20	0.0017395020	000000000100n001	0.0017395020	000000000100n001

Table 12 Coefficients with a Maximum of 4 Non-Zero Digits

The response of filter with CSD coefficients from the proposed GA method with maximum 4 non-zero digits is shown in Fig. 3.15 and the coefficients are listed in Table 12. This filter has a square error of 4.8028×10^{-5} .

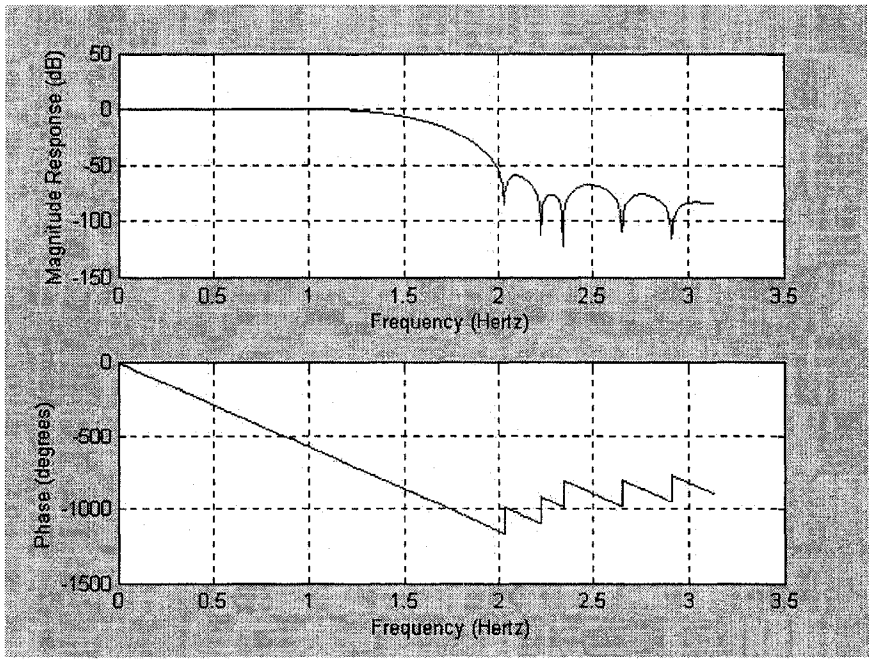

Fig. 3.15 Response of filter with CSD coefficients designed by GA with maximum 4 non-zero digit.

The response of filter with CSD coefficients converted from IP with maximum 3 non-zero digits is shown in Fig. 3.16 and the coefficients are listed in Table 13. This filter has a square error of 1.0343 $x10^{-3}$.

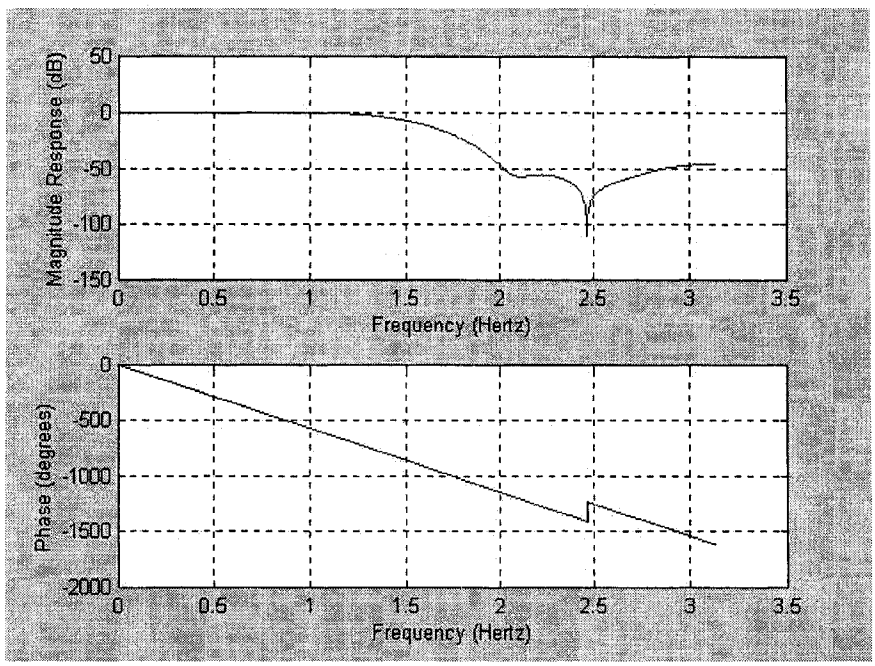

Fig. 3.16 Response of Filter with CSD Coefficients Converted from IP with a Maximum of 3 Non-Zero Digits

3 Max. Non-zero digits	Converted from optimum design		GA Design	
	Decimal	CSD (n=-1)	Decimal	CSD (n=-1)
a0	0.0017395020	000000000100n001	0.0006713867	000000000010n0n0
a1	0.0024719238	0000000001010001	0.0009155273	00000000001000n0
a2	-0.0059814453	0000000n01000n00	-0.0043334961	00000000n00n0010
a3	-0.0107421875	000000n010100000	-0.0078125000	0000000n00000000
a4	0.0126953125	00000010n0100000	0.0114746094	00000010n000n000
a5	0.0322570801	0000010000100001	0.0283203125	00000100n0100000
a6	-0.0205078125	000000n0n0n00000	-0.0205078125	000000n0n0n00000
a7	-0.0820312500	0000n0n0n0000000	-0.0800781250	0000n0n00n000000
a8	0.0273132324	00000100n000000n	0.0283203125	00000100n0100000
a9	0.3105468750	001010000n000000	0.3085937500	00101000n0000000
a10	0.4697265625	01000n0000100000	0.4686279297	01000n0000000n00
a11	0.3105468750	001010000n000000	0.3085937500	00101000n0000000
a12	0.0273132324	00000100n000000n	0.0283203125	00000100n0100000
a13	-0.0820312500	0000n0n0n0000000	-0.0800781250	0000n0n00n000000
a14	-0.0205078125	000000n0n0n00000	-0.0205078125	000000n0n0n00000
a15	0.0322570801	0000010000100001	0.0283203125	00000100n0100000
a16	0.0126953125	00000010n0100000	0.0114746094	00000010n000n000
a17	-0.0107421875	000000n010100000	-0.0078125000	0000000n00000000
a18	-0.0059814453	0000000n01000n00	-0.0043334961	00000000n00n0010
a19	0.0024719238	0000000001010001	0.0009155273	00000000001000n0
a20	0.0017395020	000000000100n001	0.0006713867	000000000010n0n0

Table 13 Coefficients with a Maximum of 3 Non-Zero Digits

The response of the filter with CSD coefficients from the proposed GA method with a maximum of 3 non-zero digits is shown in Fig. 3.17 and the coefficients are listed in Table 13. This filter has a square error of 3.4379×10^{-4} .

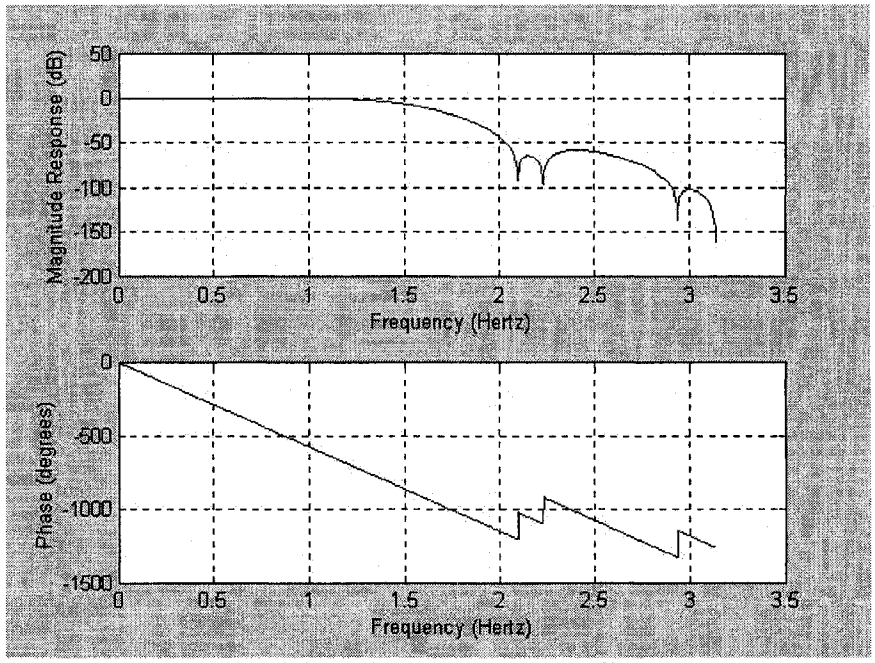

Fig. 3.17 Response of the Filter with CSD coefficients Designed by GA with a Maximum 3 Non-Zero Digits

2 Max. Non-Zero Digits	Converted from Optimum Design		GA Design	
	Decimal	CSD (n=-1)	Decimal	CSD (n=-1)
a0	0.0017089844	000000000100n000	-0.0002441406	000000000000n000
a1	0.0024414063	0000000001010000	0.0006103516	0000000000010100
a2	-0.0058593750	0000000n01000000	-0.0029296875	00000000n0100000
a3	-0.0117187500	000000n010000000	-0.0073242188	0000000n00010000
a4	0.0136718750	000000100n000000	0.0097656250	0000000101000000
a5	0.0322265625	0000010000100000	0.0273437500	00000100n0000000
a6	-0.0195312500	000000n0n0000000	-0.0195312500	000000n0n0000000
a7	-0.0781250000	0000n0n000000000	-0.0781250000	0000n0n000000000
a8	0.0273437500	00000100n0000000	0.0273437500	00000100n0000000
a9	0.3125000000	0010100000000000	0.3125000000	0010100000000000
a10	0.4687500000	01000n0000000000	0.4687500000	01000n0000000000
a11	0.3125000000	0010100000000000	0.3125000000	0010100000000000
a12	0.0273437500	00000100n0000000	0.0273437500	00000100n0000000
a13	-0.0781250000	0000n0n000000000	-0.0781250000	0000n0n000000000
a14	-0.0195312500	000000n0n0000000	-0.0195312500	000000n0n0000000
a15	0.0322265625	0000010000100000	0.0273437500	00000100n0000000
a16	0.0136718750	000000100n000000	0.0097656250	0000000101000000
a17	-0.0117187500	000000n010000000	-0.0073242188	0000000n00010000
a18	-0.0058593750	0000000n01000000	-0.0029296875	00000000n0100000
a19	0.0024414063	0000000001010000	0.0006103516	0000000000010100
a20	0.0017089844	000000000100n000	-0.0002441406	000000000000n000

Table 14 Coefficients with a Maximum of 2 Non-Zero Digits

The response of the filter with CSD coefficients converted from IP with a maximum of 2 non-zero digits is shown in Fig. 3.18 and the coefficients are listed in Table 14. This filter has a square error of 1.4394 $x10^{-2}$.

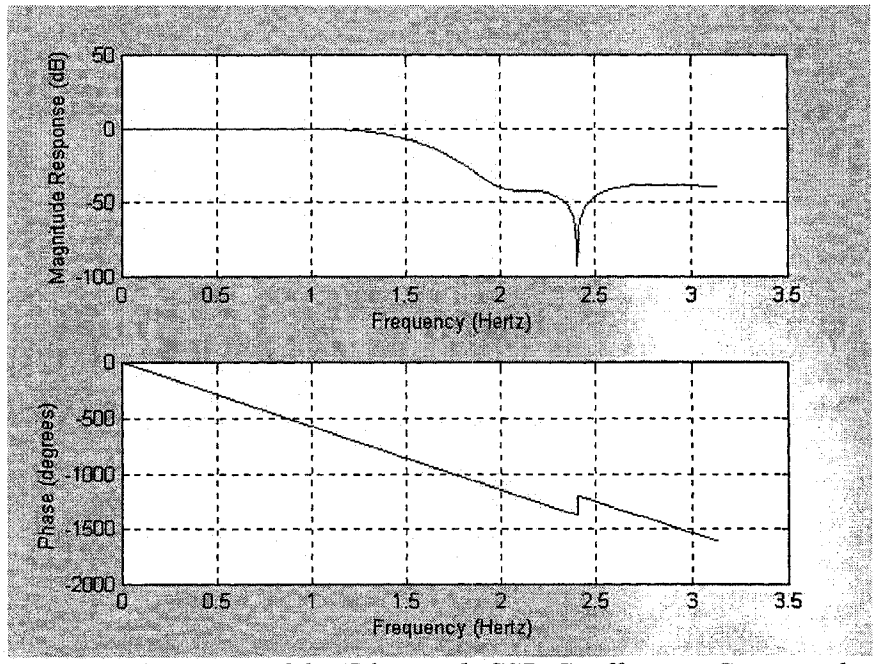

Fig. 3.18 Response of the Filter with CSD Coefficients Converted from IP with a maximum of 2 Non-Zero Digits

The response of the filter with CSD coefficients from the proposed GA method with a maximum of 2 non-zero digits is shown in Fig. 3.19 and the coefficients are listed in Table 14 . This filter has a square error of 7.8210 $x10^{-3}$.

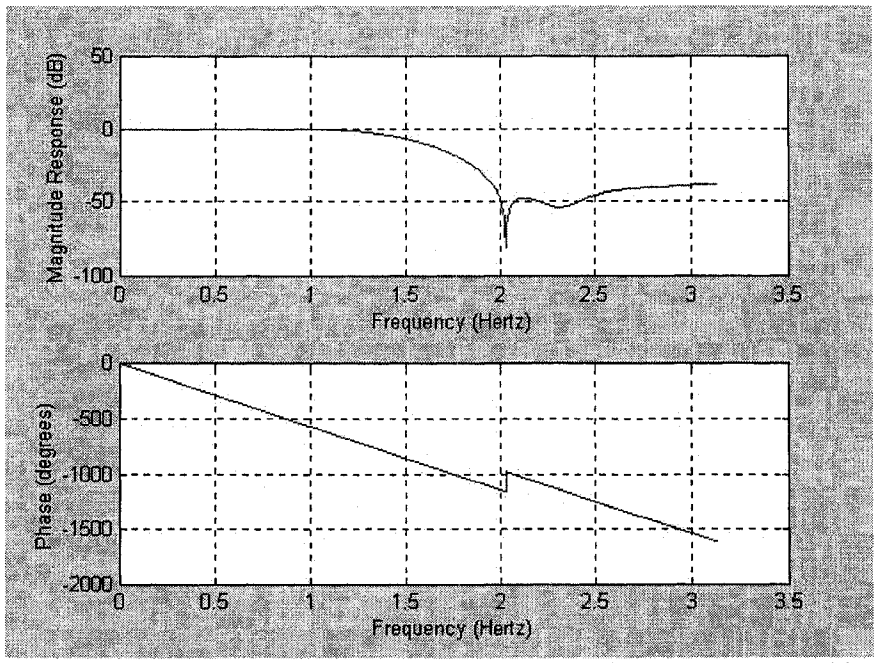

Fig. 3.19 Response of the Filter with CSD Coefficients Designed by GA with a Maximum 2 Non-Zero Digits

1 Max. Non-Zero Digits	Converted from Optimum Design		GA Design	
	Decimal	CSD (n=-1)	Decimal	CSD (n=-1)
a0	0.0019531250	0000000001000000	-0.0156250000	000000n000000000
a1	0.0019531250	0000000001000000	-0.0156250000	000000n000000000
a2	-0.0078125000	0000000n00000000	0.0312500000	0000010000000000
a3	-0.0078125000	0000000n00000000	0.0312500000	0000010000000000
a4	0.0156250000	0000001000000000	-0.0312500000	00000n0000000000
a5	0.0312500000	0000010000000000	-0.0312500000	00000n0000000000
a6	-0.0156250000	000000n000000000	0.0156250000	0000001000000000
a7	-0.0625000000	0000n00000000000	-0.0312500000	00000n0000000000
a8	0.0312500000	0000010000000000	-0.0009765625	0000000000n00000
a9	0.2500000000	0010000000000000	0.2500000000	0010000000000000
a10	0.5000000000	0100000000000000	0.5000000000	0100000000000000
a11	0.2500000000	0010000000000000	0.2500000000	0010000000000000
a12	0.0312500000	0000010000000000	-0.0009765625	0000000000n00000
a13	-0.0625000000	0000n00000000000	-0.0312500000	00000n0000000000
a14	-0.0156250000	000000n000000000	0.0156250000	0000001000000000
a15	0.0312500000	0000010000000000	-0.0312500000	00000n0000000000
a16	0.0156250000	0000001000000000	-0.0312500000	00000n0000000000
a17	-0.0078125000	0000000n00000000	0.0312500000	0000010000000000
a18	-0.0078125000	0000000n00000000	0.0312500000	0000010000000000
a19	0.0019531250	0000000001000000	-0.0156250000	000000n000000000
a20	0.0019531250	0000000001000000	-0.0156250000	000000n000000000

Table 15 Coefficients with a Maximum of 1 Non-Zero Digit

The response of the filter with CSD coefficients converted from IP with a maximum of 1 non-zero digit is shown in Fig. 3.20 and the coefficients are listed in Table 15. This filter has a square error of 1.5444.

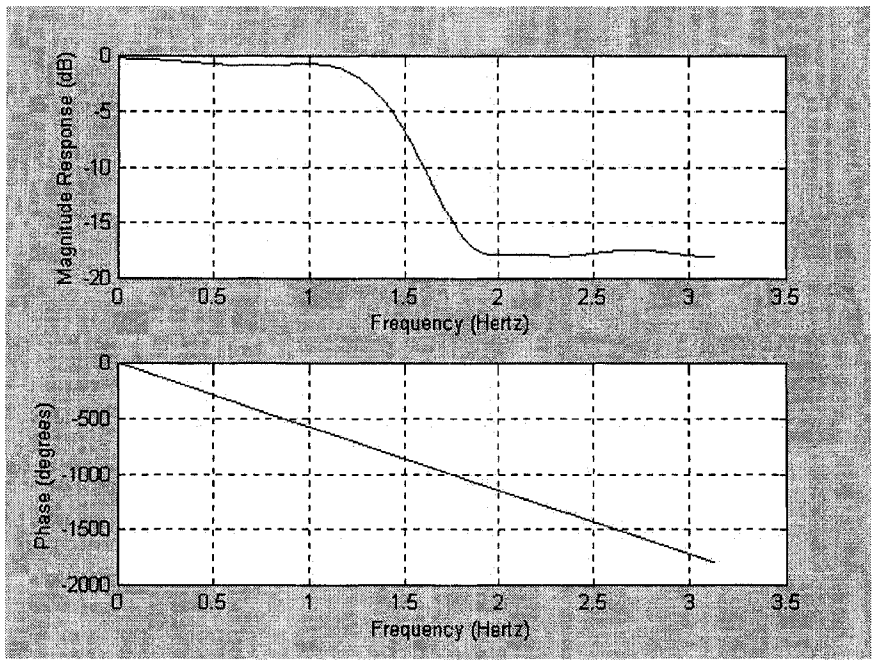

Fig. 3.20 CSD Coefficients Converted from IP with a Maximum of 1 non-zero digit

The response of the filter with CSD coefficients from the proposed GA method with a maximum of 1 non-zero digit is shown in Fig. 3.21 and the coefficients are listed in Table 15. This filter has a square error of 7.1102×10^{-1} .

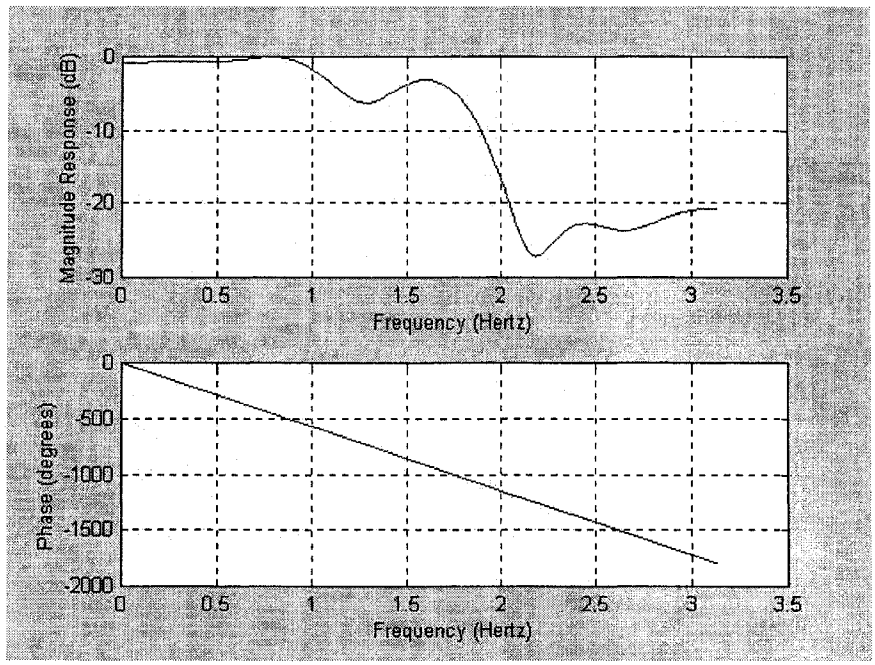

Fig. 3.21 Response of the Filter with CSD Coefficients Designed by GA with a Maximum of 1 Non-Zero Digit

A Performance comparison summary is shown in Table 16. The filters are compared relative to the optimal IP filter's error which is the lowest possible error.

Maximum number of allowed non-zero digits in any filter coefficient	Normalized Square Error (relative to optimum error of 4.4383 x10^{-5})		Improvement in Error Increase
	Filter with infinite precision coefficients converted to CSD	Filter with CSD coefficients determined by proposed GA Method	
6	1.006922	1.003792	45.2%
5	1.01297	1.00535	58.8%
4	1.19215	1.08214	57.3%
3	23.306	7.746	69.8%
2	324.30	176.2	45.8%
1	34796.6	16020	53.9%

Table 16 Filter Square Error Relative to Filter with Optimum Infinite Precision Coefficients

In all cases the GA designed filter has a lower square error than a filter converted from an infinite precision design. While all filters using CSD coefficients will have a higher error than the optimum design using infinite precision coefficients the improvement in this increase of the proposed GA method over a conversion to CSD format of the IP coefficients ranges from 45.2% to 69.8%. A specially noteworthy case is the conversion method with 6 non-zero digits compared to the proposed GA method design with 5 non-zeros. Here the GA design produces lower error while utilizing fewer non-zero digits.

In some cases the GA designed filter has a lower error than a converted filter design with more non-zero digits.

3.9 Conclusion

The method presented in this chapter maximizes the potential of a GA search for 1-D non-recursive filters with CSD coefficients. The improved functionality of the GA allows it to find filters which would have previously been difficult to find. The full capability of the genetic algorithm is now available for designing non-recursive CSD coefficients filters.

CHAPTER 4. 1-D RECURSIVE FILTER DESIGN

4.1 Recursive Filters

When the b_i filter coefficients of (1.1) or (1.2) are all 0 except for b_0 the filter is a non-recursive filter and is always stable. For this type of filter design, the preceding method works well. However, for recursive filter designs the possibility exists that the filter will not be stable. The forgoing GA approach of for non-recursive filters must be extended to handle the additional constraint. As always care must be taken to ensure that the effect on performance is minimized.

Several approaches have been proposed for handling the constraint imposed by unstable filters. The most common method [35]-[37] is to simply give any filter that is not stable a fitness value of zero. As discussed in Section 2.9.11, a filter with such a low fitness will have a very slim chance of being selected and for all intents and purposes has been rejected. Discarding an unstable filter discards all of the filter's schemata even though some of them may be quite good with possibly only a few bad ones causing the instability.

An unstable filter has inherited schemata from parents which were selected on the basis of merit. These hard won schemata could be passed on to offspring that may themselves be stable especially if the unstable parent filter is on the verge of stability. However unstable filters are definitely undesirable so some form of penalty must be applied.

4.2 Determining Filter Stability

The criterion for filter stability [1] is that the poles, which are the roots of the denominator of a filter's transfer function, B(z) in equation (1.2), must all lie within the unit circle on the z plane. If any poles are on or outside of the unit circle, then the filter is not unconditionally stable.

To ascertain a suitable penalty factor for an unstable filter it would be helpful to quantify the degree of instability. A measure, such as the sum of the distances to the unit circle for all poles outside the limit, could be used as the basis of a penalty factor. Those filters showing more instability would have a larger penalty applied to their normal fitness value.

Such a scheme would necessitate root finding for each filter to determine the pole locations. This is computationally intensive especially with large population sizes and would negate any efficiency gained as a result of not simply rejecting unstable filters.

A simple and quick method of determining filter stability is the Jury-Marsden [1] method. It does not require that any polynomial roots be found and only requires the calculation of a series of 2 by 2 determinants. Unfortunately, it is a pass/fail type of test that does not yield any measure of the amount of unstableness.

For maximum GA efficiency the fitness lowering penalty factor must strike a balance between destroying the schemata of unstable filters by outright rejection and allowing some to survive with a low enough probability that the final design will not be unstable. The optimum penalty factor is determined empirically.

A series of test designs was done for several penalty factors ranging from 2 to infinite. For each penalty factor 100 filters were designed with each design being allowed to run for 300 generations. Any unstable filter occurring in any generation had it's fitness penalized by the penalty factor using the formula fitness = fitness/penalty factor.

The fitness of each of the 100 filters for each penalty factor was averaged and the result plotted in Fig. 4.1. The number of unstable filters designed is also plotted.

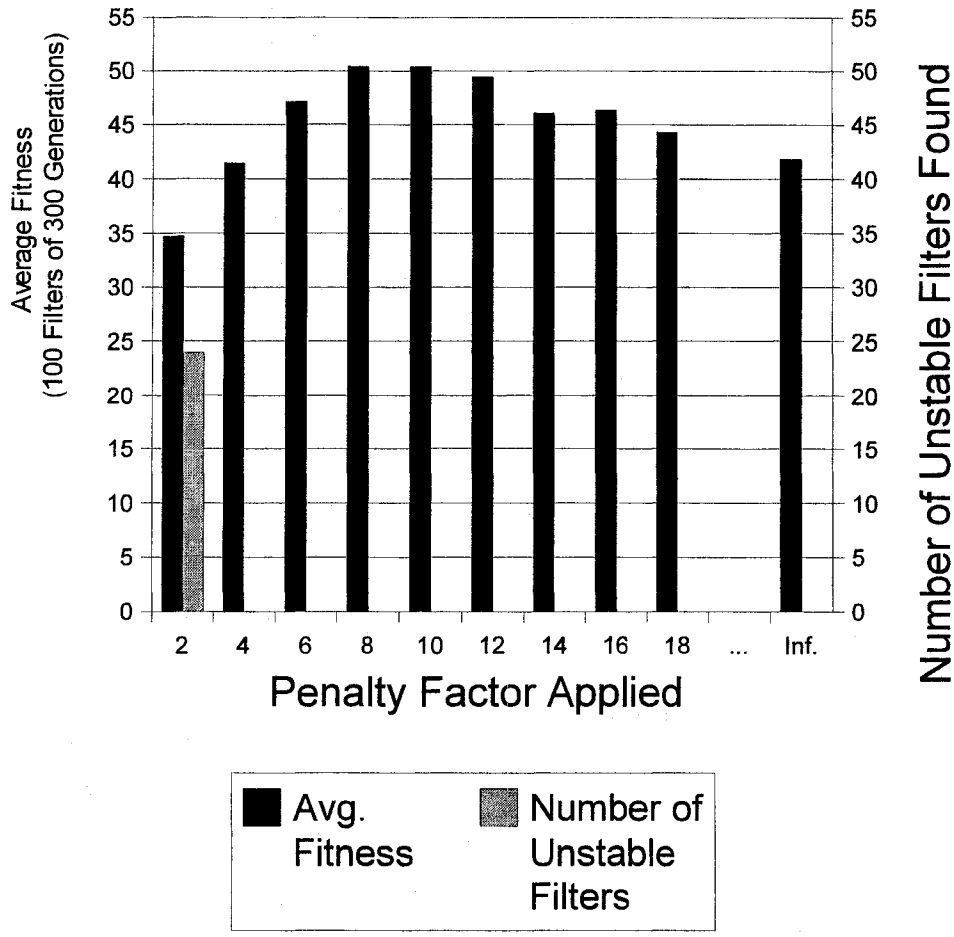

Fig. 4.1 Optimum Penalty Factor Determination

As can be seen, when using a low penalty factor of 2, the 100 filter designs had an average fitness of 35 with 24 of the 100 filters being unstable. When that was raised to 4, the average fitness increased to 41 and no unstable filters were produced. In fact, no unstable filters were produced for any penalty factor greater than 2.

From the graph it is clear that the optimum penalty factor occurs somewhere between 8 and 10. The infinite penalty factor is the same as setting the fitness value to 0 which is the method used in previous approaches. When using the optimum penalty factor the average fitness is improved by 19% over the infinite penalty factor and no unstable filters are produced.

4.3 Recursive Filter Design Example

A recursive filter was designed for the target frequency response shown in (4.1).

$$|H(e^{j\omega})|=\left\{\begin{array}{ll} 1 & 0\le\omega\le 1 \\ 0 & 2\le\omega\le\pi \end{array}\right\} \tag{4.1}$$

The unstable filter penalty factor was set at 10 and the design converged on the coefficients of Table 17. The mean square error in the frequency response taken at 16384 points along the spectrum is only 0.00855 The stability of the filter is determined by plotting the poles on the z plane in Fig. 4.2. They are all within the unit circle indicating that the filter is stable.

Coefficient	CSD Representation	Decimal
a0	000000000010n000	+0.00006103516
a1	0000000100000010	+0.00787353516
a2	0000010010100000	+0.03613281250
a3	00010n0n00000000	+0.08593750000
a4	0001000010n00000	+0.12792968750
a5	0001000000101000	+0.12622070310
a6	00010n0n00000000	+0.08593750000
a7	0000010010100000	+0.03613281250
a8	0000000100101000	+0.00903320310
a9	0000000000100n00	+0.00085449220
b0	0100000000001001	+0.50027465818
b1	0n00n00010000000	-0.55859375000
b2	100n00n000000000	+0.85937500000
b3	0n00n00100000000	-0.55468750000
b4	010n00000n000000	+0.37304687500
b5	000n00n00n000000	-0.14257812500
b6	000010n0000n0000	+0.04638671880
b7	0000000n00100000	-0.00683593750
b8	000000000000n000	-0.00024414060
b9	0000000000000001	+0.00003051758

Table 17 Coefficients of Example Recursive Filter

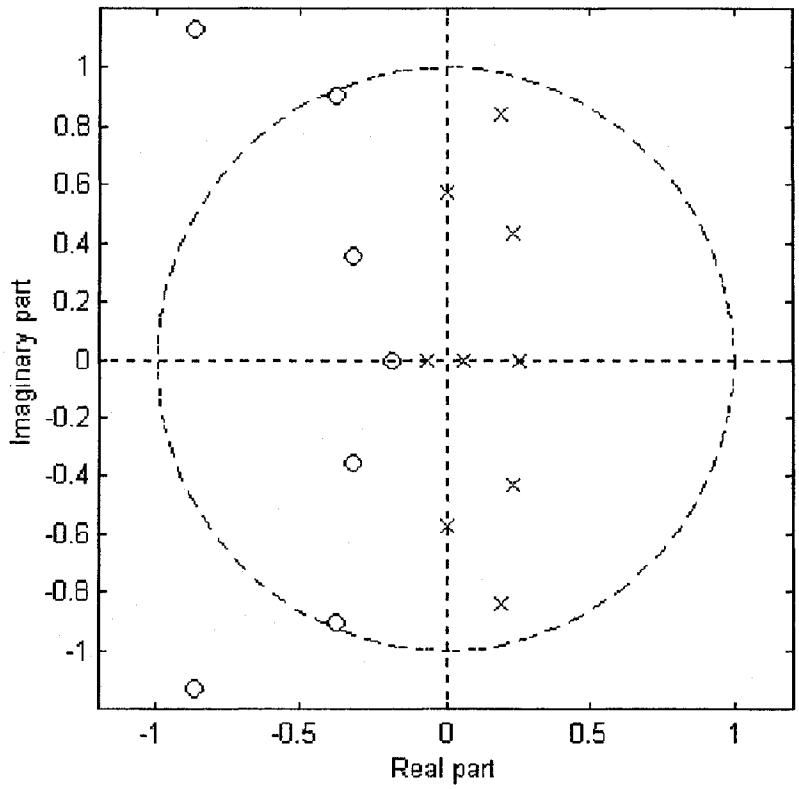

Fig. 4.2 z-Plane Plot of the Poles (X) and Zeros (O) of the Example Non-Recursive Filter

4.4 Optimum GA Population Size Test

As stated in Section 2.9.10, the exact values of operator parameters, such as population size, crossover rate and mutation rates are not critical as long as they are not extreme. To verify that statement a test was performed to measure the effects on GA performance due to changes in the GA population size.

The design of a 4th order non-recursive filter was used for all tests. It used 10 digit CSD coefficients limited to a maximum of 3 non-zero digits. The probability of crossover was

fixed at 0.9 and the rate of mutation was fixed at 0.05. The filter had the target frequency response shown in (4.2)

$$|H(e^{j\omega})|=\left\{\begin{array}{ll} 1 & 0\le\omega\le 0.3\pi \\ 0.75 & \omega=0.4\pi \\ 0.5 & \omega=\pi/2 \\ 0.25 & \omega=0.6\pi \\ 0 & 0.7\pi\le\omega\le 0.3\pi \end{array}\right\} \tag{4.2}$$

Increasing population size requires more calculations to be performed on each generation but also makes the GA converge in fewer generations. So to keep the test fair the number of calculations per filter design was fixed at 40000. Therefore as the population size is increased the number of generations the GA performs before being terminated will decrease so as to keep the number of calculations constant.

Each population size was run 100 times and the total Elapsed time and the average square error of each filter design was tabulated as shown in Table 18.

Population Size	Number of Generations	Number of Calculations per Filter	Elapsed Time for 100 Filter (Seconds)	Avg. Filter Square Error (x 10^{-2})
100	400	40000	307.91	1.350
200	200	40000	298.55	1.348
300	133	40000	303.25	1.366
400	100	40000	310.11	1.326
500	80	40000	319.49	1.390

Table 18 Optimum GA Population Size Test Results

The results are shown graphically in Fig. 4.3.

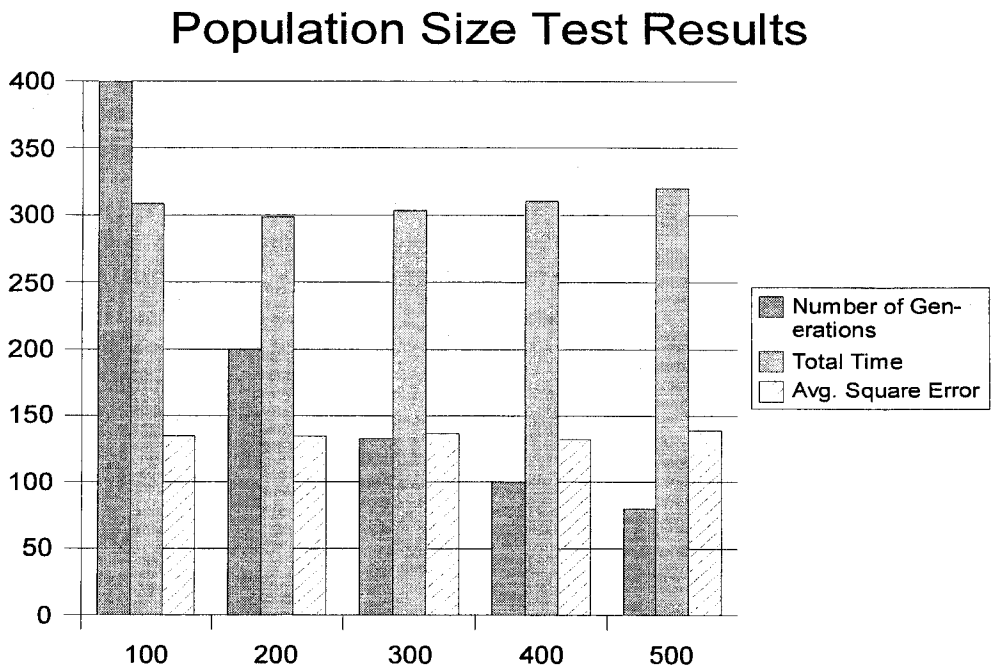

Fig. 4.3 Optimum GA Population Size Test Results

As can be seen the average square error changes very little with population size changes. The variation is less than 5% and there is no clear pattern or trend as to which population size is best.

An interesting result though is the total time taken. The pattern would seem to indicate that even though the number of calculations is held constant, the total time required is not. A population size of 200 appears to be the optimum with gradual slowing for the population sizes on either side of this.

However, since the total elapsed time variation is less than 7% for a population size change of 500% it can be concluded that population size has very little effect on GA performance as long as extreme values are avoided.

4.5 Conclusion

The method presented in this chapter extends the capabilities of a GA search for 1-D filters with CSD coefficients to include recursive filters. An efficient method for handling unstable filters has been demonstrated allowing the full capability of the genetic algorithm to be used for designing non-recursive and recursive CSD coefficient filters.

CHAPTER 5. TWO-DIMENSIONAL FILTERS

5.1 INTRODUCTION

Two-dimensional (2-D) filters have the transfer function given in (5.1)

$$H(z_1, z_2) = \frac{\sum_{k=0}^{M}\sum_{l=0}^{N} a_{k,l}\, z_1^{-k}, z_2^{-l}}{\sum_{k=0}^{M}\sum_{l=0}^{N} b_{k,l}\, z_1^{-k}, z_2^{-l}} = \frac{A(z_1, z_2)}{B(z_1, z_2)} \tag{5.1}$$

Where $z_1 = e^{j\omega_1 T}$, $z_2 = e^{j\omega_2 T}$, T is the sample period and $a_{k,l}$ and $b_{k,l}$ are the coefficients of the numerator and denominator of the filter respectively [38],[39].

For the transfer function to be stable:

$$B(z_1, z_2) \neq 0, \quad \bigcap_{i=1}^{2} |z_i| \geq 1 \tag{5.2}$$

2-D filters can be implemented by direct calculation of the difference equation (5.3).

$$y(mT, nT) = \sum_{k=0}^{M}\sum_{l=0}^{N} a_{k,l}\, x(mT - kT, nT - lT) - \sum_{k=0}^{M}\sum_{\substack{l=0 \\ k+l\neq 0}}^{N} b_{k,l}\, y(mT - kT, nT - lT) \tag{5.3}$$

High throughput two-dimensional (2-D) filters can be achieved using a two stage approach. First, the 2-D filter is reformulated as a series of cascaded one-dimensional (1-D) filters which reduces the number of overall multipliers needed. Then the component 1-D filters are implemented using CSD coefficients allowing for efficient coefficient multiplication.

5.2 2-D Filters as Cascaded 1-D Filters

It has been shown [40],[41] that a 2-D filter can be implemented as M parallel sections of two cascaded 1-D filters in z_1 and z_2 as shown in Fig. 5.1.

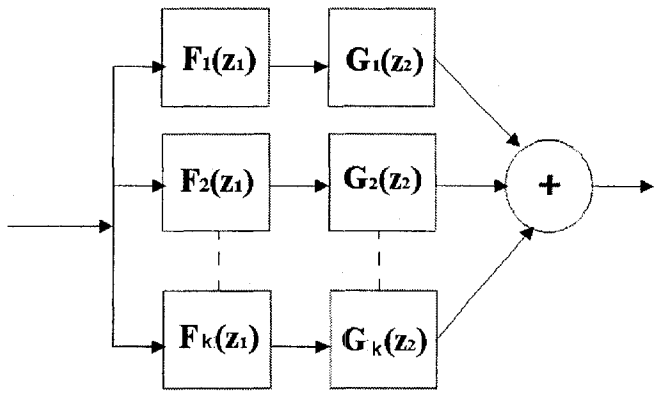

Fig. 5.1 Cascaded 1-D Filters Form a 2-D Filter

Direct implementation of a 2-D filter of order N x N requires $(N + 1)^2$ multipliers, whereas for parallel sections of cascaded 1-D filter implementations the requirement is $2k(N+1)$ multipliers. Since $N>2k$ a hardware saving will be achieved. Such a filter is suitable for a pipelined, high throughput implementation, which is an advantage over the direct implementation of 2-D filters.

Let $A=\{a_{p,q}\}$ be the desired sampled amplitude response of a 2-D filter where

$$a_{p,q}=|H(e^{j\pi u_p}, e^{j\pi v_q})| \qquad 1\le p\le P, 1\le q\le Q \tag{5.4}$$

and u_p and v_q are the normalized frequencies given by:

$$u_p=\frac{p-1}{P-1} \qquad v_q=\frac{q-1}{Q-1}$$

such that $0\le u_p\le 1, 0\le v_q\le 1$

The SVD of matrix A is

$$A=\sum_{i=1}^{r}\sigma_i u_i v_i' \tag{5.5}$$

where $\sigma_1\geq\sigma_2\geq\sigma_3\geq\sigma_4\geq\cdots\geq\sigma_r$ are the singular values of A, u_i and v_i are unit vectors and r is the rank of **A** .

If $\phi_i=\sigma_i^{1/2}u_i$ and $\gamma_i=\sigma_i^{1/2}v_i$ then

$$A=\sum_{i=1}^{r}\phi_i\gamma_i \tag{5.6}$$

These $\phi_i\gamma_i$ are the sampled amplitude response characteristics of the 1-D filters $F_i(z_1)$ and $G_i(z_2)$ which can be used to form the 2-D filter A by cascading F and G in parallel branches.

When the number of parallel branches $k=r$ the filter will be a minimal square error approximation of the desired 2-D response. The number of parallel branches may be arbitrarily reduced such that $k<r$ to further reduce the number of multiplications required. This will result in higher filter error but since the higher order branches contribute little to the filter response it may be a worthwhile compromise.

The transfer function of the designed filter using this approach will be in the form

$$H(z_1,z_2)=\sum_{i=1}^{k}X_i(z_1)Y_i(z_2) \tag{5.7}$$

where $k\leq r$ and the orders of filters X_i and Y_i are considered to be identical and equal to N for i = 1,2,3, ... ,k.

For the first branch, none of the sampled amplitude responses in the vectors ϕ_i and γ_i can have negative values. This is not true for the subsequent branches. Each vector ϕ_i and γ_i (i >1) are dealt with individually. A positive value equal to each vector's most negative value is added to all elements of that vector. This shifts the whole vector up so that it has no negative members. These shifts must be accounted for in the filter implementation [41].

5.3 FIR Design Example

Design a 2-D filter with the target frequency response $|M_d(\omega_1,\omega_2)|$ shown in (5.8). A graph of this response is shown in Fig. 5.2.

$$M_d(\omega_1,\omega_2)=|H(e^{j\omega_1},e^{j\omega_2})|=\begin{cases}1 & 0\le\sqrt{\omega_1^2+\omega_2^2}\le 1 \text{ rad/sec}\\ e^{-2(\sqrt{\omega_1^2+\omega_2^2}-1)}, & otherwise\end{cases} \tag{5.8}$$

The filter designed utilized M=3 for three parallel sections. Each parallel section has 2 linear phase 1-D filters of 41st order. Each of the 21 unique coefficients for these 1-D filters was calculated as a 20-digit CSD number with a maximum limit of 3 non-zero coefficients. The genetic algorithm used parameters given in Table 19 and each 1-D design took approximately four minutes to complete on a 2.0 Ghz Pentium computer. Fig. 5.3 shows the magnitude response $|M(\omega_1,\omega_2)|$ of the designed filter.

Population Size	Number of Generations	Crossover Rate	Mutation Rate
500	1000	.95	.05

Table 19. GA Parameters for 2-D FIR Example

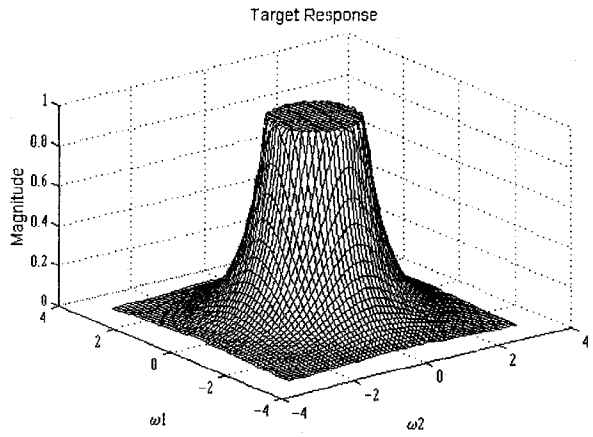

Fig. 5.2 Target Response

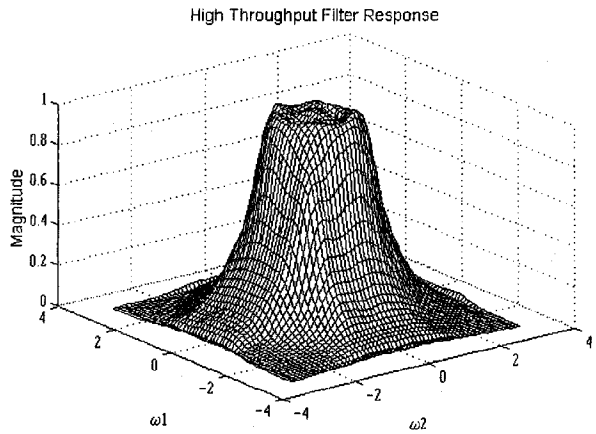

Fig. 5.3 High Throughput Filter Response

The closeness of this approach can be seen by the error given in Fig. 5.4. The greatest error occurs in two regions of the transition slope where the response is not entirely circular. The other areas show very little error.

This error could be reduced in a couple of ways. Another parallel branch could be added to 2-D implementation, the order of the 1-D component filters could be increased or the number of allowable CSD non-zero digits could be increased allowing better granularity for coefficient selection. All of these methods will increase the number of additions required to perform all of the required multiplication.

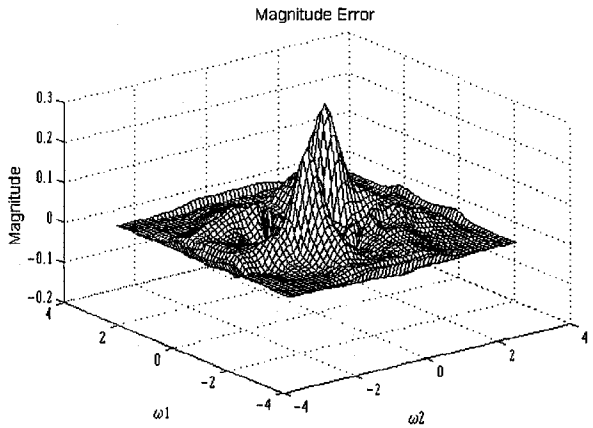

Fig. 5.4 Target Response Error

The CSD filter coefficients in both CSD format and decimal value for the six 1-D (F1,G1,F2,G2,F3,G3) filters are given in Table 20,Table 21 and Table 22.

Coefficient	F1 (decimal)	F1 (CSD) (n=-1)	G1 (decimal)	G1 (CSD) (n=-1)
a0	0.0024490356	0000000010100000100	-0.0005493164	00000000000n00n00000
a1	0.0002479553	0000000000010000010	-0.0012512207	0000000000n0n00n0000
a2	0.0009689331	0000000001000000n00	-0.0009727478	0000000000n000000010
a3	-0.0076904297	0000000n000001000000	0.0075645447	000000010000n00000n0
a4	-0.0019683838	000000000n000000n000	0.0031738281	0000000010n010000000
a5	-0.0012817383	0000000000n0n0n00000	0.0014038086	00000000010n00n00000
a6	0.0236816406	0000010n000010000000	-0.0239257813	00000n01000n00000000
a7	0.0160522461	00000010000100n00000	-0.0161151886	000000n0000n0000000n
a8	-0.0390548706	00000n0n000000000100	0.0390625000	0000010100000000000
a9	-0.2812423706	00n00n00000000000100	0.2812500000	0010010000000000000
a10	-0.4726562500	0n000100n00000000000	0.4688720703	01000n00000001000000
a11	-0.2812423706	00n00n00000000000100	0.2812500000	0010010000000000000
a12	-0.0390548706	00000n0n000000000100	0.0390625000	0000010100000000000
a13	0.0160522461	00000010000100n00000	-0.0161151886	000000n0000n0000000n
a14	0.0236816406	0000010n000010000000	-0.0239257813	00000n01000n00000000
a15	-0.0012817383	0000000000n0n0n00000	0.0014038086	00000000010n00n00000
a16	-0.0019683838	000000000n000000n000	0.0031738281	0000000010n010000000
a17	-0.0076904297	0000000n000001000000	0.0075645447	000000010000n00000n0
a18	0.0009689331	0000000001000000n00	-0.0009727478	0000000000n000000010
a19	0.0002479553	0000000000010000010	-0.0012512207	0000000000n0n00n0000
a20	0.0024490356	0000000010100000100	-0.0005493164	00000000000n00n00000

Table 20 Coefficients of F1 and G1 for 2-D FIR Example.

Coefficient	F2 (decimal)	F2 (CSD) (n=-1)	G2 (decimal)	G2 (CSD) (n=-1)
a0	0.0019493103	0000000010000000n0	-0.0000629425	00000000000000n0000n
a1	0.0010375977	0000000001000100000	-0.0097656250	0000000n0n0000000000
a2	-0.0018920898	000000000n0000100000	0.0000381470	0000000000000010100
a3	-0.0039138794	00000000n00000000n00	0.0151367188	00000010000n00000000
a4	-0.0038452148	00000000n00000100000	0.0273456573	00000100n00000000001
a5	-0.0076751709	0000000n000001001000	0.0019454956	0000000010000000n00
a6	-0.0024337769	000000000n0n00000100	-0.0322265625	00000n0000n000000000
a7	0.0312213898	000001000000000n0001	-0.0937500000	000n0100000000000000
a8	0.1259765625	0001000001000000000	-0.0703048706	0000n00n000000000100
a9	0.2812576294	0010010000000000100	-0.0161094666	000000n0000n00000010
a10	-0.0783691406	0000n0n00000n0000000	0.3125000000	0010100000000000000
a11	0.2812576294	0010010000000000100	-0.0161094666	000000n0000n00000010
a12	0.1259765625	0001000001000000000	-0.0703048706	0000n00n000000000100
a13	0.0312213898	000001000000000n0001	-0.0937500000	000n0100000000000000
a14	-0.0024337769	000000000n0n00000100	-0.0322265625	00000n0000n000000000
a15	-0.0076751709	0000000n000001001000	0.0019454956	0000000010000000n00
a16	-0.0038452148	00000000n00000100000	0.0273456573	00000100n00000000001
a17	-0.0039138794	00000000n00000000n00	0.0151367188	00000010000n00000000
a18	-0.0018920898	000000000n0000100000	0.0000381470	0000000000000010100
a19	0.0010375977	0000000001000100000	-0.0097656250	0000000n0n0000000000
a20	0.0019493103	0000000010000000n0	-0.0000629425	00000000000000n0000n

Table 21 Coefficients of F2 and G3 for 2-D FIR Example

Coefficient	F3 (decimal)	F3 (CSD) (n=-1)	G3 (decimal)	G3 (CSD) (n=-1)
a0	-0.0000591278	000000000000000n00001	0.0004940033	0000000000010000010n
a1	0.0018920898	000000000010000n00000	0.0039443970	0000000100000010100
a2	0.0156230927	00000010000000000000n	0.0117263794	00000010n00000000100
a3	0.0001564026	0000000000000101010	-0.0048828125	00000000n0n000000000
a4	-0.0273475647	00000n001000000000n0	-0.0273437500	00000n00100000000000
a5	-0.0388183594	00000n0n000010000000	-0.0546913147	0000n0010000000000n0
a6	-0.0468730927	0000n010000000000001	-0.0478515625	0000n01000n000000000
a7	-0.0155029297	000000n0000001000000	-0.0156097412	000000n0000000001000
a8	0.0312194824	000001000000000n0000	0.0234375000	0000010n000000000000
a9	0.0137939453	000000100n0001000000	0.0273361206	00000100n00000000n00
a10	-0.2811889648	00n00n00000000100000	-0.2656269073	00n000n000000000000n
a11	0.0137939453	000000100n0001000000	0.0273361206	00000100n00000000n00
a12	0.0312194824	000001000000000n0000	0.0234375000	0000010n000000000000
a13	-0.0155029297	000000n0000001000000	-0.0156097412	000000n0000000001000
a14	-0.0468730927	0000n010000000000001	-0.0478515625	0000n01000n000000000
a15	-0.0388183594	00000n0n000010000000	-0.0546913147	0000n0010000000000n0
a16	-0.0273475647	00000n001000000000n0	-0.0273437500	00000n00100000000000
a17	0.0001564026	0000000000000101010	-0.0048828125	00000000n0n000000000
a18	0.0156230927	00000010000000000000n	0.0117263794	00000010n00000000100
a19	0.0018920898	000000000010000n00000	0.0039443970	0000000100000010100
a20	-0.0000591278	000000000000000n00001	0.0004940033	0000000000010000010n

Table 22 Coefficients of F3 and G3 for 2-D FIR Example

5.4 Recursive 2-D Design Example

Three 2-D filters with the following magnitude specifications were designed:

$$M_d(\omega_1,\omega_2)=|H(e^{j\omega_1},e^{j\omega_2})|=\begin{cases}1, & if\ \sqrt{\omega_1^2+\omega_2^2}\ \le\ 0.08\pi\\ 0.5, & if\ 0.08\pi \le \sqrt{\omega_1^2+\omega_2^2}\ \le\ 0.12\pi\\ 0, & otherwise.\end{cases} \tag{5.9}$$

This target magnitude $|M_d(\omega_1,\omega_2)|$ is plotted in Fig. 5.5.

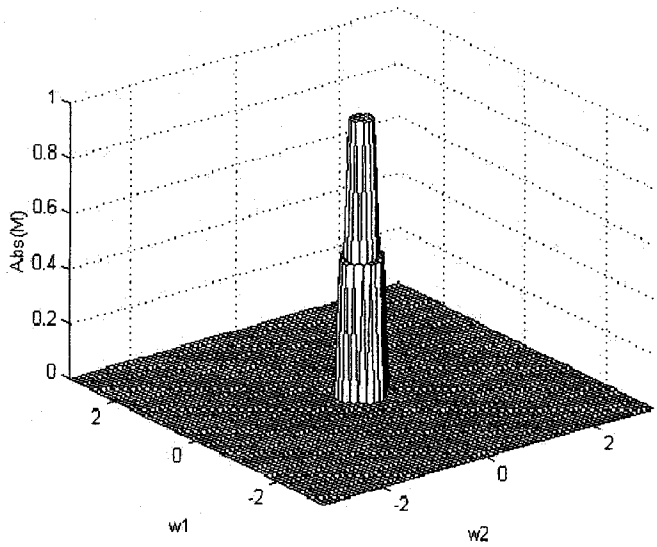

Fig. 5.5. Desired Magnitude Response of the 2-D Filter.

All 2-D filters utilized k=3 for three parallel sections with each section composed of two 1-D non-recursive filters. The three example 2-D designs are identical except for the order of the component 1-D filters. They are composed of either all 2nd order, all 3rd order or all 4th order 1-D component filters. Each of the coefficients for these 1-D filters is a 16-digit CSD number with a maximum limit of 3 non-zero digits.

The genetic algorithm used the parameters given in .Table 23.

Population Size	Number of Generations	Crossover rate	Mutation Rate
500	500	.95	.05

Table 23 GA Parameters used for the 2-D Non-Recursive Example

The magnitude response $|M(\omega_1, \omega_2)|$ for the 2-D filter composed of all 2nd order 1-D filters is shown inFig. 5.6 and the coefficients are given in Table 24.

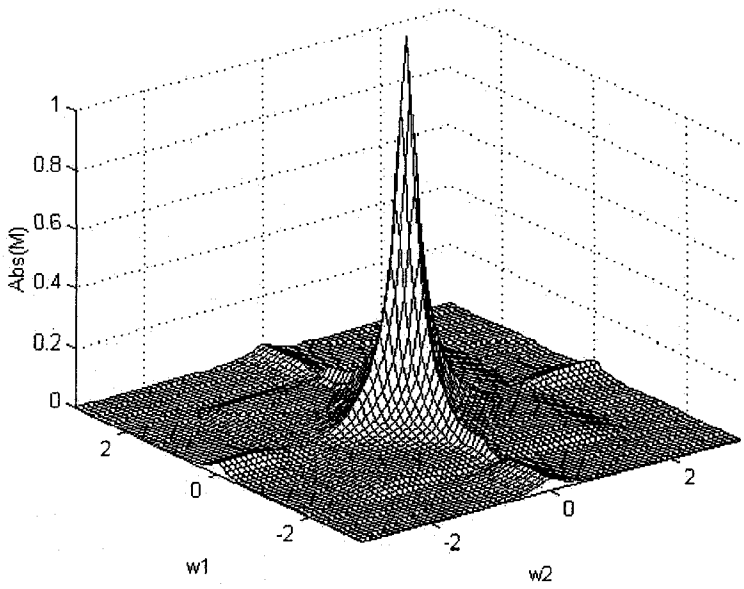

Fig. 5.6. Amplitude Response of the 2-D Filter using Cascaded 2nd order 1-D Filters with CSD Coefficients.

	F1: CSD	value	G1:CSD	value
a0	0000n01000n00000	-0.0478515625	0000n01000n00000	-0.0478515625
a1	0000000010101000	0.0051269531	0000000010101000	0.0051269531
a2	0000n0100n000000	-0.0488281250	0000n0100n000000	-0.0488281250
b0	10000n0n00000000	0.9609375000	10000n0n00000000	0.9609375000
b1	n00n0n0000000000	-1.1562500000	n00n0n0000000000	-1.1562500000
b2	00100100n0000000	0.2773437500	00100100n0000000	0.2773437500
	F2: CSD	value	G2:CSD	value
a0	00n0n0000n000000	-0.3144531250	0000010101000000	0.0410156250
a1	0000n00n0n000000	-0.0722656250	0000010000100100	0.0323486328
a2	00n0101000000000	-0.1718750000	000100000n010000	0.1235351562
b0	10000n0000000n00	0.9686279297	10n00n0000000000	0.7187500000
b1	010100000n000000	0.6230468750	0100n00000100000	0.4384765625
b2	0100000010000n00	0.5037841797	0010001010000000	0.2695312500
	F3: CSD	value	G3:CSD	value
a0	000n0n000000n000	-0.1564941406	01000n0n00000000	0.4609375000
a1	00n00000n0010000	-0.2534179688	0n000100000n0000	-0.4692382812
a2	00n000n000001000	-0.2653808594	00000n0000000n00	-0.0313720703
b0	100000n0n0000000	0.9804687500	1010000100000000	1.2578125000
b1	0010101000000000	0.3281250000	n0000n0000000100	-1.0311279297
b2	0100n00n00000000	0.4296875000	000n0n0000100000	-0.1552734375

Table 24. Coefficients in Decimal and CSD Representation (where n = -1) for the 2-D Filter using Cascaded 2nd Order 1-D Filters.

The magnitude response $|M(\omega_1, \omega_2)|$ for the 2-D filter composed of all 3rd order 1-D filters is shown in Fig. 5.7 and the coefficients are given in Table 25.

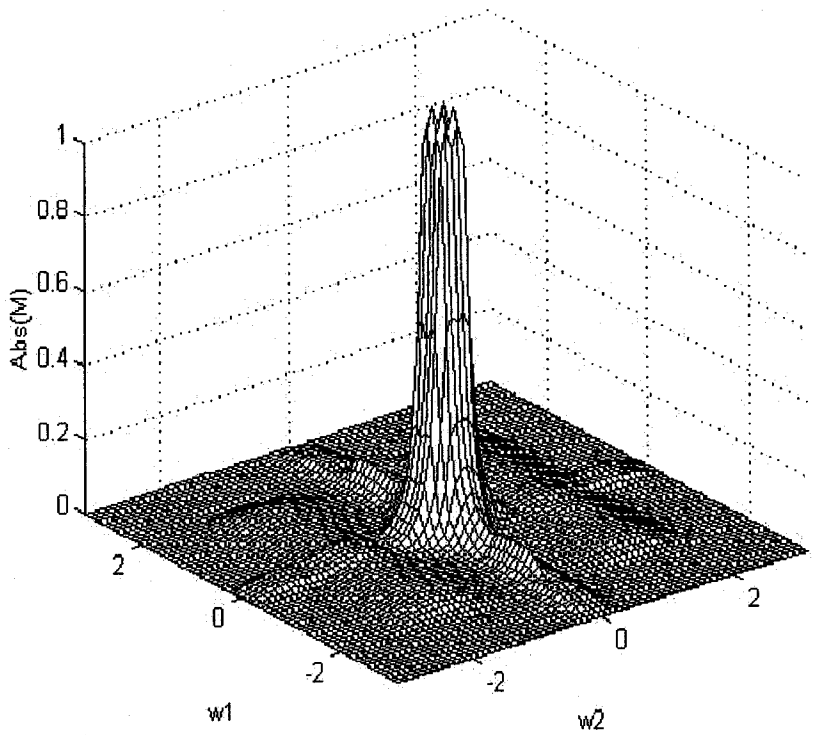

Fig. 5.7 Amplitude Response of the 2-D Filter using Cascaded 3rd Order 1-D filters with CSD Coefficients

	F1: CSD	value	G1:CSD	value
a0	00000n0000101000	-0.0300292969	00000n0000101000	-0.0300292969
a1	0000000001000010	0.0020141602	0000000001000010	0.0020141602
a2	00000000100000n0	0.0038452148	00000000100000n0	0.0038452148
a3	00000n0000010010	-0.0307006836	00000n0000010010	-0.0307006836
a4	100010n000000000	1.0468750000	100010n000000000	1.0468750000
b0	n0n00000n0000000	-1.2539062500	n0n00000n0000000	-1.2539062500
b2	00n00000000n0n00	-0.2506103515	00n00000000n0n00	-0.2506103515
b3	0100001000n00000	0.5146484375	0100001000n00000	0.5146484375
	F2: CSD	value	G2:CSD	value
a0	0001010010000000	0.1601562500	0000001010100000	0.0205078125
a1	00000n0n0n000000	-0.0410156250	00000010n0100000	0.0126953125
a2	00100100n0000000	0.2773437500	0000101000001000	0.0783691406
a3	00n000100n000000	-0.2363281250	0000000010000n00	0.0037841797
a4	100000100n000000	1.0136718750	0100n01000000000	0.4531250000
b0	0000n00101000000	-0.0527343750	00100000n000n000	0.2458496094
b2	0000010101000000	0.0410156250	0001000000010010	0.1255493164
b3	00n0n0n000000000	-0.3281250000	000000n000101000	-0.0144042969
	F3: CSD	value	G3:CSD	value
a0	00000100n0n00000	0.0263671875	0n00000010100000	-0.4951171875
a1	00n0000100000n00	-0.2423095703	010000n000n00000	0.4833984375
a2	00n010n000000000	-0.2031250000	000000n010100000	-0.0107421875
a3	000n0n0n00000000	-0.1640625000	00000n0n0n000000	-0.0410156250
a4	100000n0n0000000	0.9804687500	10100000000000n0	1.2499389648
b0	00010n0100000000	0.1015625000	n00n00n000000000	-1.1406250000
b2	0100000n0n000000	0.4902343750	00000000n000n010	-0.0040893554
b3	00n0100100000000	-0.1796875000	0000000100000n0n	0.0076599121

Table 25. Coefficients in Decimal and CSD Representation (where n = -1) for 2-D Filter using Cascaded 3rd Order 1-D Filters.

The magnitude response $|M(\omega_1, \omega_2)|$ for the example filter composed of all 4th order 1-D filters is shown in Fig. 5.8 and the coefficients are given in Table 26.

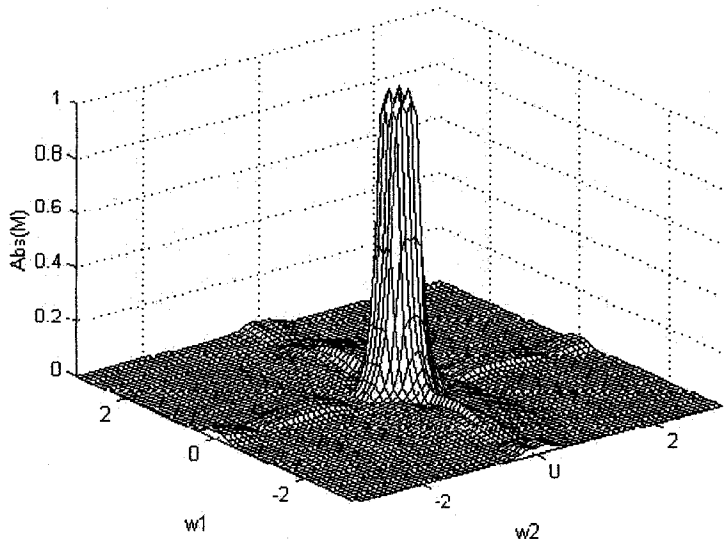

Fig. 5.8 Amplitude response of the 2-D filter using cascaded 4th order 1-D filters with CSD coefficients.

	F1: CSD	value	G1:CSD	value
a0	00000n0001010000	-0.0288085938	00000n0001010000	-0.0288085938
a1	00000000n0n0n000	-0.0051269531	00000000n0n0n000	-0.0051269531
a2	00000000100000n0	0.0038452148	00000000100000n0	0.0038452148
a3	0000000n00n00n00	-0.0089111328	0000000n00n00n00	-0.0089111328
a4	00000n0010000100	-0.0272216797	00000n0010000100	-0.0272216797
b0	100000n0000n0000	0.9838867188	100000n0000n0000	0.9838867188
b2	n0000000100n0000	-0.9965820312	n0000000100n0000	-0.9965820312
b3	0n01000000n00000	-0.3759765625	0n01000000n00000	-0.3759765625
b4	0010100000000010	0.3125610352	0010100000000010	0.3125610352

b5	0001001010000000	0.1445312500	0001001010000000	0.1445312500
	F2: CSD	value	G2:CSD	value
a0	0000001000n00n00	0.0145263672	00000000n00n000n	-0.0044250488
a1	0100000000000n00	0.4998779297	0010000n0n000000	0.2402343750
a2	0n00000010000000	-0.4960937500	00n0000000n0000n	-0.2510070801
a3	0000000101010000	0.0102539062	000000000100n00n	0.0016784668
a4	0000001000n0n000	0.0144042969	0000000100010001	0.0083312988
b0	1000000001001000	1.0021972656	1000000n0n000000	0.9902343750
b2	n000100000000n000	-0.9377441406	n00000n0n0000000	-1.0195312500
b3	00000000000n00n0	-0.0005493164	0000000010000001	0.0039367676
b4	0000000100010001	0.0083312988	0000001010100000	0.0205078125
b5	0000000n0n0000n0	-0.0098266602	0000101010000000	0.0820312500
	F3: CSD	value	G3:CSD	value
a0	000100000n000000	0.1230468750	0000000010010000	0.0043945312
a1	000n000n0n000000	-0.1347656250	00000100n00n0000	0.0268554688
a2	00n0001000001000	-0.2341308594	00000000n0n000n0	-0.0049438477
a3	000n00n0n0000000	-0.1445312500	0n00100000010000	-0.4370117188
a4	00000000n0n00000	-0.0048828125	010000000n0n0000	0.4975585938
b0	1000000000001000	1.0002441406	1010000100000000	1.2578125000
b2	0000001010001000	0.0197753906	n00000n0n0000000	-1.0195312500
b3	0000101000100000	0.0791015625	00000n0000100100	-0.0301513672
b4	0000n0n0n0000000	-0.0820312500	000001000000n000	0.0310058594
b5	0000n00n0n000000	-0.0722656250	000n010100000000	-0.0859375000

Table 26 Coefficients in Decimal and CSD Representation (where n = -1) for 2-D Filter using Cascaded 4th Order 1-D Filters

If the 2-D target response is specified as a complex frequency response then after SVD transformation the 1-D component target responses will also be specified as complex frequency responses [40]. Consequently, the magnitude of the target responses of the 1-D

filters can never be negative. In the example filter however, the 2-D target response $|M_d(\omega_1,\omega_2)|$ is specified for magnitude only. After SVD transformation the 1-D filter responses are also specified as magnitude only. In this case, 1-D filters F_n and G_n (n>1) can have a negative magnitude specification [41]. To compensate, a bias value is added to shift the target magnitude response up making all values positive. An inverse bias must be used in the filter implementation to compensate. This will require four additions beyond those required for coefficient multiplication. The bias values used in the example design are given in Table 27.

F2	G2	F3	G3
0.51452313093353	0.24328398326778	0.13827000193691	0.40588223422716

Table 27. 1-D component filter bias values

5.5 2-D Non-Recursive Filter Example Comparison

To check the results the 2-D designs the results using the proposed GA technique are compared with the design in [42]. While this design is also a non-recursive filter, has the same target frequency response $|M_d(\omega_1,\omega_2)|$ and utilized a GA for the design it was not specifically intended for a high throughput implementation and utilizes full range 32 bit binary coefficients. The frequency response $|M(\omega_1,\omega_2)|$ of the filter in [42] is reproduced in . The mean square error of the example filters and the comparison filter taken at 50 equally spaced points $P_{x,y}$, in each dimension $(-\pi \le x, y \le \pi)$ is given in Table 28.

	Design of [42]	Design using 2nd Order Components	Design using 3rd Order Components	Design using 4th Order Components
Mean Square Error ($x10^{-3}$)	2.048	1.899	0.9810	0.6847
Number of additions	240 (average)	106	138	168

Table 28 Mean Square Error and Number of Additions Required for Coefficient Multiplication of Considered 2-D Filters

Normally it would be expected that the high throughput filter with bit-limited CSD coefficients would not be able to match the mean square error of a filter using full range 32 bit binary coefficients. The results however, prove otherwise. The example 2-D filter using second order 1-D filters has a slightly better mean square error than [42] but requires only 106 additions to perform the coefficient multiplication. As the remaining two examples show, this error can be further lowered at the expense of more additions. For the example using forth order 1-D filters, the error is a third of that in [42] and yet still only requires 168 additions for coefficient multiplication.

Since the design in [42] was not intended as a high throughput design, no attempt was made to quantify the actual number of additions required. Instead an average figure is calculated which applies to any filter such as this with fifteen coefficients using 32 bit binary coefficients. Since on average a 32 bit binary number will have 16 ones, on average 15 coefficients will require 240 additions.

5.6 Conclusion

The approach presented in this chapter for the design of high throughput 2-D filters uses several strategies for increasing performance. The design of the 2-D filters as a

parallel arrangement of 1-D filters provides for high throughput, pipelined, parallel processing and permits the use of high performance 1-D filters with CSD coefficients. This method can produce 2-D filters with better response characteristics and reduced computational requirements.

CHAPTER 6. COMMON SUBEXPRESSION ELIMINATION

6.1 Introduction

Further reduction in the number of additions required for the multiplication of CSD filter coefficients can be gained through the use of common subexpression elimination [43],[44]. When a portion of an expression (subexpression) occurs more than once it can be calculated once and the result used wherever that subexpression occurs within the expression. Since removing one subexpression may preclude the removal of others, identifying which sub-expressions are most advantageous to remove is a difficult search and optimization problem.

6.2 Subexpression Types

Sub-expression can be any bit pattern within a CSD coefficient but the most commonly occurring ones [44] are the 2-bit CSD subexpressions such as $101, 10\bar{1}, 1001,$ etc. which have a non-zero on each end and one or more zeros in the middle. 2-bit CSD sub-expressions may be common within the coefficients (horizontal sub-expressions) or between coefficients (vertical sub-expressions).

6.3 Horizontal Sub-Expression Elimination within a Coefficient

Sub-expressions can be found and eliminated horizontally within a CSD coefficient [44],[46]. As an example, suppose we wish to calculate (6.1).

$$y=(1010101)x \tag{6.1}$$

Using shift/add in place of multiplication this would become (6.2) which requires three additions.

$$y = x + x \ll 2 + x \ll 4 + x \ll 6 \tag{6.2}$$

The 2-bit sub-expression $1\,0\,1$ appears more than once (three times but only two are in a position to be eliminated). This can be algebraically re-arranged as in (6.3).

$$y = x + x \ll 2 + (x + x \ll 2) \ll 4 \tag{6.3}$$

This clearly shows the sub-expression $x + x \ll 2$ $(1\,0\,1)$ occurring twice. By pre-calculating this sub-expression as $s = x + x \ll 2$ our expression becomes $y = s + s \ll 4$ which requires a total of only 2 additions including the one for calculating sub-expression*s*. This represents a 1/3 savings in the number of additions required.

6.4 Vertical Sub-Expression Elimination

Sub-expressions can also be found and eliminated in the vertical dimension [47] when the coefficients are staked. As an example of vertical sub-expression elimination, suppose we wish to calculate (6.4).

$$\begin{aligned} y = &(1\,0\,0\,\bar{1}\,0\,\mathbf{1})x[0] \\ &+(\mathbf{1}\,0\,1\,0\,0\,\mathbf{1})x[-1] \end{aligned} \tag{6.4}$$

Here the 2-bit common sub-expression, shown in bold, is the pair of 1's appearing vertically in the first bit position when the coefficients are stacked as shown. This sub-expression appears again in the last bit position.

This is calculated as in (6.5) for a total of 5 total additions.

$$\begin{aligned} y = &x[0] \;\; - x[0] \ll 2 \;\; + x[0] \ll 5 \\ &+ x[-1] + x[-1] \ll 3 + x[-1] \ll 5 \end{aligned} \tag{6.5}$$

This can be more readily shown by rearranging (6.5) to get (6.6).

$$y=(x[0]+x[-1])+(x[0]+x[-1])\ll 5-x[0]\ll 2+x[-1]\ll 3 \qquad (6.6)$$

Here the common sub-expression is $s=x[0]+x[-1]$. By pre-calculating s our original expression becomes (6.7).

$$y=s+s\ll 5-x[0]\ll 2+x[-1]\ll 3 \qquad (6.7)$$

This has a total of 4 total additions including the one used in calculating s for a reduction of one addition.

6.5 Horizontal Sub-Expression Elimination Across Coefficients

Horizontal sub-expressions can be eliminated across coefficients if the appropriate input delay is taken into consideration. In (6.4) of the previous example there also appears the 2-bit horizontal sub-expression 10001 in both coefficients. In this example, x[0] is the current input value and x[-1] is the previous input value that has been delayed. To eliminate this we can pre-calculate the 10001 sub-expression and simply delay the result until it is needed. Rearranging (6.4) yields (6.8).

$$y=(x[0]+x[0]\ll 5)-x[0]\ll 2+(x[-1]+x[-1]\ll 5)+x[-1]\ll 3 \qquad (6.8)$$

Pre-calculating $s[0]=x[0]+x[0]\ll 5$ this becomes

$$y=s[0]-x[0]\ll 2+s[-1]+x[-1]\ll 3 \qquad (6.9)$$

where s[-1] is the delayed sub-expression value from the previous calculation. Eliminating this sub-expression results in a saving of 1 addition operation.

Note that eliminating the vertical sub-expression in (6.4) will preclude eliminating the horizontal sub-expression across coefficients and vice-versa.

As well, in the first example $y=(1010101)x$ the sub-expression 101 appears 3 times but only the two that do not share a non-zero digit with the other can be eliminated.

In general, the elimination of a 2-bit sub-expression will remove the sub-expression's two terminating non-zero digits from participation in any other subexpression. These potential sub-expressions are therefore no longer available for elimination.

6.6 Graphical Transformation

There has been some study as to whether it is better to remove the vertical subexpression [44], horizontal subexpressions [47]. A comparison of these methods [48] determined that the best approach varies by problem type. Without predetermined knowledge about which type to eliminate, the best approach would be to search for both vertical and horizontal simultaneously to find the best combination for the particular problem at hand

To this end, a new graphical method of identifying sub-expressions and potential elimination paths is proposed to optimally eliminate both vertical and horizontal sub-expressions. The method employs a two-step graphical transformation which converts the problem into one very similar to the much studied traveling salesman problem [1] where well known methods such as a Genetic Algorithm can be applied.

6.6.1. Identification Graph

The first transformation step is the identification (ID) graph $G_{id}=(V_{id}, E_{id})$. This graph is similar to the admissibility graph of [49] but it has been extended to include both

vertical and horizontal subexpressions and only contains the information necessary for subexpression identification.

To create this graph the CSD coefficients are first stacked vertically for easy identification of the horizontal and vertical sub-expression dimensions. Then the graph vertices are created according to (6.10).

$$V_{id}=\{\text{non-zero digits in all coefficients}\} \tag{6.10}$$

Next a partial ID graph $G'_{id}=(V_{id}, E'_{id})$ with edges E'_{id} representing all possible vertical and horizontal sub-expressions are defined as $E'_{id}=E_h+E_v$ where

$$E_h=\{(V_a, V_b)\forall V_a, V_b \text{ within the same coefficient}\} \tag{6.11}$$

and

$$E_v=\{(V_a, V_b)\forall V_a, V_b \text{ with the same vertical bit location}\} \tag{6.12}$$

Therefore, E_h and E_v are the edges for fully connected sub-graphs in the horizontal and vertical dimensions respectfully.

The edges are next labeled with their properties. They have a start and end vertices V_a and V_b as all edges do. They have a type: horizontal or vertical. They have a polarity to indicate if the sign of their non-zero digit vertices have matching signs. For example, edges representing 101 or -10-1 have positive polarity while those representing 10-1 and -101 have negative polarity. Horizontal types also have a bit position separation distance to indicate their length and vertical types have a coefficient pair (c1,c2) and a bit position.

The final ID graph edges E_{id} representing only the common sub-expressions is now formed from E'_{id}. The edges that do not share the same (common) properties with at least one other edge are removed to leave only the common subexpressions. The final edges are determined by

$$E_{id}=E'_{id}-E_{unique} \tag{6.13}$$

where $E_{unique}=\{\text{edges with unique properties}\}$.

To illustrate this, suppose a filter has the CSD coefficients

$$c_0=1010\bar{1}001, c_1=10000101 \tag{6.14}$$

To get the actual dimensions to match the nomenclature they are first stacked vertically as in Table 29.

c_0	$1010\bar{1}001$
c_1	10000101

Table 29 Coefficient Stacking

The vertices of G_{id} $V_{id}=\{V_1 \ldots V_7\}$ representing the non-zero digits of the coefficients are formed and have the properties shown in Table 30.

Vertex (Non-zero digit)	Digit Polarity	Coefficient	digit Position
V_1	+1	0	8
V_2	+1	0	6
V_3	-1	0	4
V_4	+1	0	1
V_5	+1	1	8
V_6	+1	1	3
V_7	+1	1	1

Table 30 Properties of Vertices in ID Graph Example

These are graphed to get Fig. 6.1.

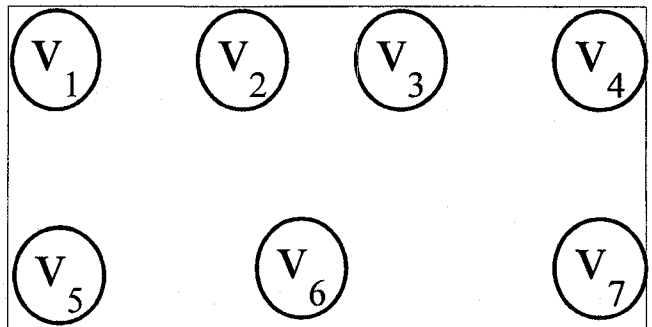

Fig. 6.1 ID Graph with Only Vertices

The edges E'_{id} representing all vertical and horizontal subexpressions are now added to get the partial ID graph G'_{id} shown in Fig. 6.2.

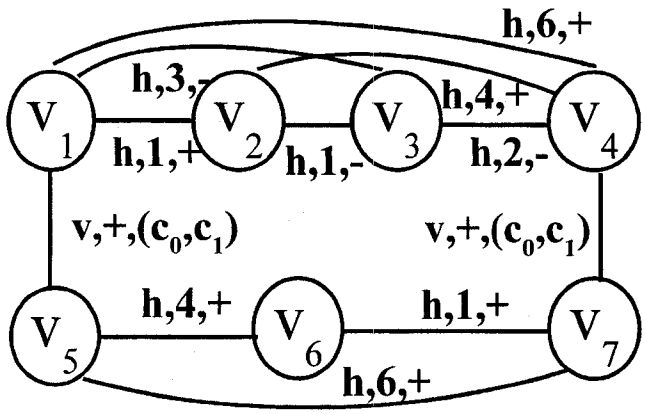

Fig. 6.2 Partial ID Graph G'_{id}

The properties of each edge is tabulated in Table 31 and added to each edge of the graph.

Edge	(V_a,V_b)	Type	Polarity	Length	(C_1,C_2)
1	(1,2)	h	+	1	----
2	(1,3)	h	-	3	----
3	(1,4)	h	+	6	----
4	(1,5)	v	+	----	(c_0,c_1)
5	(2,3)	h	-	1	----
6	(2,4)	h	+	4	-----
7	(3,4)	h	-	2	-----
8	(4,7)	v	+	----	(c_0,c_1)
9	(5,6)	h	+	4	----
10	(5,7)	h	+	6	----
11	(6,7)	h	+	1	----

Table 31 Edge list E'_{id} with Edge Properties

The edges that do not share the same (common) properties with at least one other edge are removed from the edge list. The completed ID graph G_{id} containing edges representing only common vertical and horizontal subexpressions results as shown in Fig. 6.3.

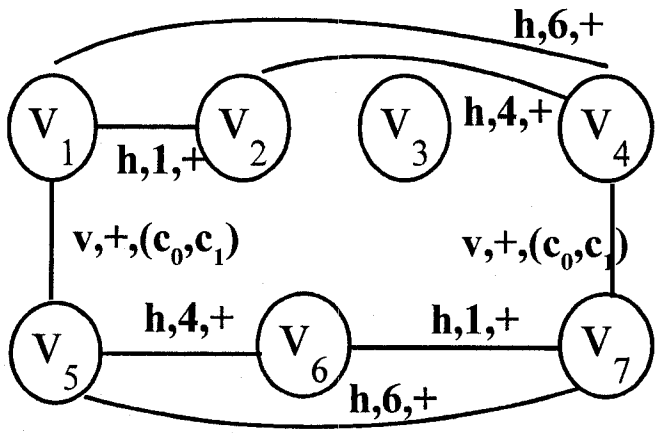

Fig. 6.3 Completed ID Graph G_{id}

6.7 Search Graph

With the ID graph complete it is used to form the search graph $G_s=(V_s, E_s)$. Each edge of the ID graph becomes a vertex of the search graph E_{id} -> V_s.

Since each vertex represents a subexpression, a Hamiltonian walk through the vertices will yield one possible elimination scheme. It is now possible to use a search and optimization technique to find the Hamiltonian walk through G_s which yields the greatest subexpression elimination.

This is very similar to the Traveling Salesman Problem (TSP) where the verticees represent cities and the edges represent the distance between cities and the object is to find the Hamiltonian walk that produces the smallest sum of edge distances.

The major difference here is that the distances are all known in advance for the TSP but with the sub-expression elimination problem traversing a sub-expression edge may not yield a reduction in additions immediately if at all. Only when an identical sub-expression is also traversed can an elimination occur and even then it may not happen.

A subexpression can only participate in an elimination if neither of its non-zero digits (vertices of G_{id}) have been allocated to a previous elimination. So depending on the path taken, the edge with identical properties may not be available. This difference will preclude some optimization methods such as those based on gradients but others such as the genetic algorithm are not affected.

To illustrate, taking the edges of G_{id} of Table 31 and using them as vertices S_1 to S_8 in G_s we get the serach graph of Fig. 6.4.

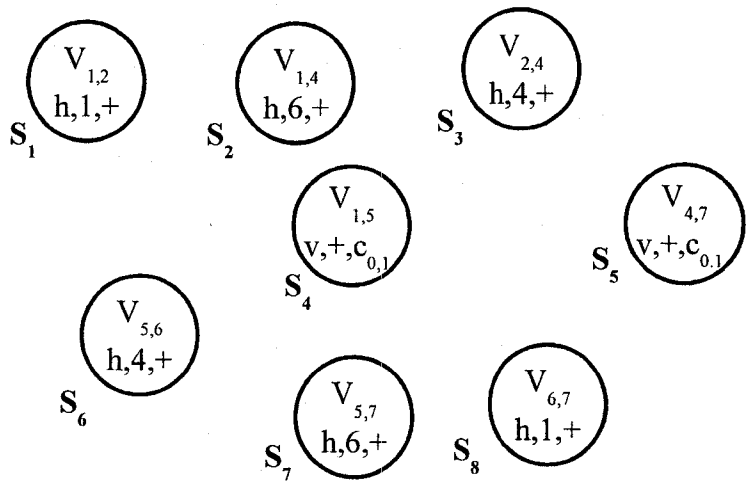

Fig. 6.4. Search Graph G_s

6.8 Example Walk Through Search Graph

One possible Hamiltonian walk is $S_6,S_7,S_3S_5,S_8,S_4,S_1,S_2$ as shown in . Here, as the path goes from S_6 to S_7, an examination shows that they have different properties and are therefore not common subexpressions and can not be eliminated. Then S_3 is traversed which has identical properties to S_6 (h,4,+) making them candidates for elimination.

To ascertain if these candidates can be eliminated a check of the availability of their non-zero digits as represented by their ID graph vetices $V_{a,b}$ is required. This is kept in an

availability table which initially shows all vertices are available. Since the table shows none of the non-zero digits of S_6, or S_3 has been allocated to other eliminations, these two subexpressions are eliminated. Their ID graph vertices $V_{5,6}$ and $V_{2,4}$ are now marked as unavailable as shown in Table 32 to prevent their non-zero digits from participating in any subsequent eliminations.

V_{id} 1	Available
V_{id} 2	Not Available
V_{id} 3	Available
V_{id} 4	Not Available
V_{id} 5	Not Available
V_{id} 6	Not Available
V_{id} 7	Available

Table 32 ID Graph Vertex Availability Table after (S_6, S_3) Elimination

As the walk continues, common pairs (S_5,S_4), (S_1,S_8) and (S_2,S_7) are found but in each case at least one of the non-zero digits specified by their ID graph vertices has already been used by the (S_6,S_3) elimination.

6.9 Example Elimination using a GA

As an example, a genetic algorithm (GA) is used to eliminate sub-expressions of a typical filter. A standard GA designed for the TSP is used [1]. The chromosome is simply an ordered list of edges which form a Hamiltonian walk through the graph. The only change was to the fitness function which returns a value based on the number of sub-expressions eliminated instead of the distance the salesman must travel.

6.9.1. Fitness Function

At the start of the fitness evaluation all vertices from the ID graph G_{id} are listed in a table and marked as available. These represent the available non-zero digits on the end of each 2-bit sub-expression to be eliminated. Traversing the search graph G_s through the Hamiltonian walk specified by the GA chromosome we get the first or next edge. The vertices in the availability table are marked as available then we mark the edge as an elimination candidate and mark its vertices as taken otherwise we mark the edge as not eliminated. We increment the occurrence count for an edge with this ones properties (type, polarity and length or coefficient pair). We repeat for all edges in order.

The fitness value is then determined by summing the n occurrence count (OC) values as shown in (6.15).

$$fitness = \sum_{i=1}^{n} (OC_i - 1, \quad if\ OC_i > 1 \quad else\ \ 0) \tag{6.15}$$

During the GA run it is not necessary to know which edges have been eliminated, only how many. However, on the final GA run we need to show which edges have been eliminated. This is done by repeating the walk again only this time we print out all the candidate edges that have an associated occurrence count of 2 or greater since these represent the sub-expression that will be eliminated.

6.10 Example

The filter chosen is a 10^{th} order FIR with the CSD 16 digit coefficients bit limited to 3 non-zero digits maximum as shown in Table 33. There are 27 non-zero digits in the coefficients requiring 26 additions for coefficient multiplication. The circled sub-

expressions are the ones eliminated. In this case even with an already low number of non-zero digits and additional 6 or 23.1% were eliminated.

Coefficient	CSD Representation (n=-1)
a 0	0100n0100000
a 1	n0n00n000000
a 2	00n0n0n00000
a 3	0100n0100000
a 4	0010n000n000
a 5	000n00000000
a 6	0000n00n0010
a 7	0000001000n0
a 8	00000010n0n0
a 9	000000000001
a 10	0000001000n0

Table 33 CSD Coefficients

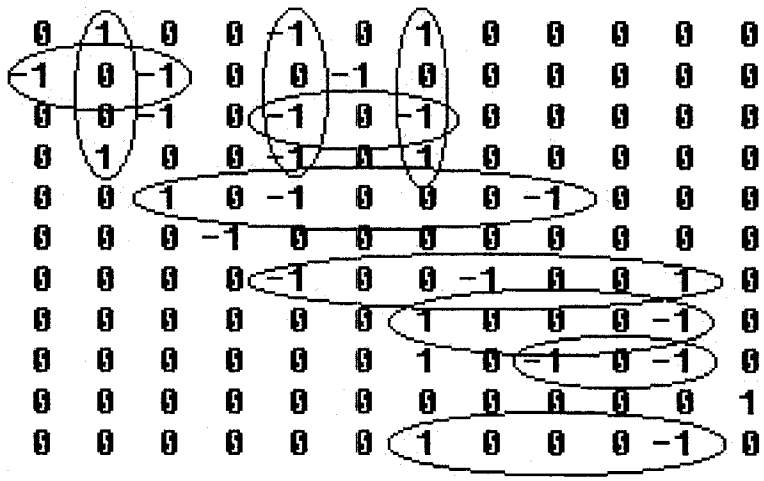

Table 34 Coefficients with Eliminated Subexpressions

6.11 Application to Previous Results

The method was used on the 1-D filters designed in Chapter 3 to determine the amount of reduction that can be expected from filters with bit-limited CSD coefficients.

In the results that follow, the original CSD format coefficients a_0 to a_{10} are shown and the then the same coefficients are given with the eliminated subexpression labeled on each of its terminating non-zero digits. The labeling is according to Table 35.

Label	Meaning
Subexpression Polarity	+ for matching non-zero digit polarity
	- for opposite non-zero digit polarity
Size	Number of digits between terminating non-zero digits (if H type)
	Digit position counting from the most significant digit (if V type)
Type	H for horizontal subexpression
	V for vertical subexpression

Table 35 Eliminated Subexpression Labeling

For example the subexpression 1001 would be shown as +2H 0 0 +2H since it is a horizontal type with matching polarity and 2 digits between the termination non-zero digits. There will always be two or more subexpressions with the same labeling and each pair represents one less addition (each triple represents two less, etc.).

Table 36 shows the eliminated subexpressions for the filter with a maximum of 2 non-zero digits. The total number of non-zero digits is 21 requiring 20 additions for coefficient multiplication. After the elimination the number of additions was reduced by 6 or 30.0%.

CSD Coefficients																
a0	0	1	0	0	0	-1	0	0	0	0	0	0	0	0	0	0
a1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
a2	0	0	0	0	0	1	0	0	-1	0	0	0	0	0	0	0
a3	0	0	0	0	-1	0	-1	0	0	0	0	0	0	0	0	0
a4	0	0	0	0	0	0	-1	0	-1	0	0	0	0	0	0	0
a5	0	0	0	0	0	1	0	0	-1	0	0	0	0	0	0	0
a6	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
a7	0	0	0	0	0	0	0	-1	0	0	0	1	0	0	0	0
a8	0	0	0	0	0	0	0	0	-1	0	1	0	0	0	0	0
a9	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0
a10	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0
Coefficients with eliminated subexpressions labeled																
a0	0	-3H	0	0	0	-3H	0	0	0	0	0	0	0	0	0	0
a1	0	0	+1H	0	+1H	0	0	0	0	0	0	0	0	0	0	0
a2	0	0	0	0	0	+6V	0	0	+9V	0	0	0	0	0	0	0
a3	0	0	0	0	1H	0	1H	0	0	0	0	0	0	0	0	0
a4	0	0	0	0	0	0	1H	0	1H	0	0	0	0	0	0	0
a5	0	0	0	0	0	+6V	0	0	+9V	0	0	0	0	0	0	0
a6	0	0	0	0	0	0	0	+1H	0	+1H	0	0	0	0	0	0
a7	0	0	0	0	0	0	0	-3H	0	0	0	-3H	0	0	0	0
a8	0	0	0	0	0	0	0	0	-1	0	1	0	0	0	0	0
a9	0	0	0	0	0	0	0	0	0	0	0	+1H	0	+1H	0	0
a10	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0	0

Table 36 Eliminated Sub-Expressions for Maximum 2 Non-Zero Digits

Table 37 shows the eliminated subexpressions for the filter with a maximum of 3 non-zero digits. The total number of non-zero digits is 20 requiring 29 additions for coefficient multiplication. After the elimination the number of additions was reduced by 7 or 24.1%.

CSD Coefficients																
a0	0	1	0	0	0	-1	0	0	0	0	0	0	0	-1	0	0
a1	0	0	1	0	1	0	0	0	-1	0	0	0	0	0	0	0
a2	0	0	0	0	0	1	0	0	-1	0	1	0	0	0	0	0
a3	0	0	0	0	-1	0	-1	0	0	-1	0	0	0	0	0	0
a4	0	0	0	0	0	0	-1	0	-1	0	-1	0	0	0	0	0
a5	0	0	0	0	0	1	0	0	-1	0	1	0	0	0	0	0
a6	0	0	0	0	0	0	1	0	-1	0	0	0	-1	0	0	0
a7	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0
a8	0	0	0	0	0	0	0	0	-1	0	0	-1	0	0	1	0
a9	0	0	0	0	0	0	0	0	0	0	1	0	0	0	-1	0
a10	0	0	0	0	0	0	0	0	0	0	1	0	-1	0	-1	0
Coefficients with eliminated subexpressions labeled																
a0	0	-3H	0	0	0	-3H	0	0	0	0	0	0	0	-1	0	0
a1	0	0	-5H	0	1	0	0	0	-5H	0	0	0	0	0	0	0
a2	0	0	0	0	0	+6V	0	0	+9V	0	+11V	0	0	0	0	0
a3	0	0	0	0	+1H	0	+1H	0	0	-1	0	0	0	0	0	0
a4	0	0	0	0	0	0	+1H	0	+1H	0	-1	0	0	0	0	0
a5	0	0	0	0	0	+6V	0	0	+9V	0	+11V	0	0	0	0	0
a6	0	0	0	0	0	0	-5H	0	-1	0	0	0	-5H	0	0	0
a7	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0	0
a8	0	0	0	0	0	0	0	0	-5H	0	0	-1	0	0	-5H	0
a9	0	0	0	0	0	0	0	0	0	0	-3H	0	0	0	-3H	0
a10	0	0	0	0	0	0	0	0	0	0	-3H	0	-1	0	-3H	0

Table 37 Eliminated Sub-Expressions for Maximum 3 Non-Zero Digits

Table 38 shows the eliminated subexpressions for the filter with a maximum of 4 non-zero digits. The total number of non-zero digits is 41 requiring 40 additions for coefficient multiplication. After the elimination the number of additions was reduced by 13 or 32.5 percent.

CSD Coefficients																
a0	0	1	0	0	0	-1	0	0	0	1	0	-1	0	0	0	0
a1	0	0	1	0	1	0	0	0	0	-1	0	0	-1	0	0	0
a2	0	0	0	0	0	1	0	0	-1	0	0	0	0	-1	0	1
a3	0	0	0	-1	0	1	0	1	0	1	0	0	0	0	0	0
a4	0	0	0	0	0	0	-1	0	-1	0	-1	0	-1	0	0	0
a5	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	1
a6	0	0	0	0	0	0	1	0	-1	0	1	0	0	1	0	0
a7	0	0	0	0	0	0	-1	0	1	0	1	0	-1	0	0	0
a8	0	0	0	0	0	0	0	-1	0	1	0	0	0	-1	0	0
a9	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0
a10	0	0	0	0	0	0	0	0	0	1	0	0	-1	0	0	1
Coefficients with eliminated subexpressions labeled																
a0	0	1	0	0	0	-1	0	0	0	-1H	0	-1H	0	0	0	0
a1	0	0	+1H	0	+1H	0	0	0	0	+2H	0	0	+2H	0	0	0
a2	0	0	0	0	0	-2H	0	0	-2H	0	0	0	0	-1H	0	-1H
a3	0	0	0	-1H	0	-1H	0	+1H	0	+1H	0	0	0	0	0	0
a4	0	0	0	0	0	0	+1H	0	+1H	0	+1H	0	+1H	0	0	0
a5	0	0	0	0	0	1	0	0	0	0	1	0	0	+1H	0	+1H
a6	0	0	0	0	0	0	-1H	0	-1H	0	+2H	0	0	+2H	0	0
a7	0	0	0	0	0	0	-1H	0	-1H	0	-1H	0	-1H	0	0	0
a8	0	0	0	0	0	0	0	-1H	0	-1H	0	0	0	-1	0	0
a9	0	0	0	0	0	0	0	0	0	1	0	+2H	0	0	+2H	0
a10	0	0	0	0	0	0	0	0	0	-2H	0	0	-2H	0	0	1

Table 38 Eliminated subexpressions for maximum 4 non-zero digits

Table 39 shows the eliminated subexpressions for the filter with a maximum of 5 non-zero digits. The total number of non-zero digits is 37 requiring 36 additions for coefficient multiplication. After the elimination the number of additions was reduced by 10 or 27.8 percent.

CSD Coefficients																
a0	0	0	0	0	0	1	0	0	-1	0	0	0	0	0	0	-1
a1	0	0	0	-1	0	1	0	1	0	1	0	0	0	1	0	0
a2	0	0	0	0	0	0	-1	0	-1	0	-1	0	-1	0	0	-1
a3	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0
a4	0	0	0	0	0	0	1	0	-1	0	1	0	0	1	0	0
a5	0	0	0	0	0	0	-1	0	1	0	1	0	-1	0	0	0
a6	0	0	0	0	0	0	0	-1	0	1	0	0	0	-1	0	0
a7	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1
a8	0	0	0	0	0	0	0	0	0	1	0	0	-1	0	0	1
a9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
a10	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1
Coefficients with eliminated subexpressions labeled																
a0	0	0	0	0	0	1	0	0	+6H	0	0	0	0	0	0	+6H
a1	0	0	0	-1H	0	-1H	0	+5H	0	1	0	0	0	+5H	0	0
a2	0	0	0	0	0	0	+3H	0	+3H	0	+3H	0	+3H	0	0	-1
a3	0	0	0	0	0	1	0	0	0	0	3H	0	0	0	3H	0
a4	0	0	0	0	0	0	+6H	0	-1H	0	-1H	0	0	+6H	0	0
a5	0	0	0	0	0	0	-1H	0	-1H	0	-1H	0	-1H	0	0	0
a6	0	0	0	0	0	0	0	-1H	0	-1H	0	0	0	-1	0	0
a7	0	0	0	0	0	0	0	0	0	1	0	+3H	0	0	0	+3H
a8	0	0	0	0	0	0	0	0	0	+5H	0	0	-1	0	0	+5H
a9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
a10	0	0	0	0	0	0	0	0	0	1	0	+3H	0	0	0	+3H

Table 39 Eliminated subexpressions for maximum 5 non-zero digits

Table 40 shows the eliminated subexpressions for the filter with a of maximum 6 non-zero digits. The total number of non-zero digits is 46 requiring 45 additions for coefficient multiplication. After the elimination the number of additions was reduced by 14 or 31.1 percent.

CSD Coefficients																
a0	0	1	0	0	0	-1	0	0	0	1	0	-1	0	0	0	-1
a1	0	0	1	0	1	0	0	0	0	-1	0	-1	0	1	0	1
a2	0	0	0	0	0	1	0	0	-1	0	0	0	0	0	-1	0
a3	0	0	0	-1	0	1	0	1	0	1	0	0	1	0	-1	0
a4	0	0	0	0	0	0	-1	0	-1	0	-1	0	-1	0	0	-1
a5	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
a6	0	0	0	0	0	0	1	0	-1	0	1	0	0	1	0	0
a7	0	0	0	0	0	0	-1	0	1	0	1	0	-1	0	0	1
a8	0	0	0	0	0	0	0	-1	0	1	0	0	0	-1	0	0
a9	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	1
a10	0	0	0	0	0	0	0	0	0	1	0	0	-1	0	0	1
Coefficients with eliminated subexpressions labeled																
a0	0	1	0	0	0	+9H	0	0	0	-1H	0	-1H	0	0	0	+9H
a1	0	0	+1H	0	+1H	0	0	0	0	+1H	0	+1H	0	+1H	0	+1H
a2	0	0	0	0	0	-2H	0	0	-2H	0	0	0	0	0	-1	0
a3	0	0	0	-1H	0	-1H	0	+1H	0	+1H	0	0	-1H	0	-1H	0
a4	0	0	0	0	0	0	+1H	0	+1H	0	+1H	0	+1H	0	0	-1
a5	0	0	0	0	0	+9H	0	0	0	0	1	0	0	0	0	+9H
a6	0	0	0	0	0	0	-1H	0	-1H	0	1	0	0	1	0	0
a7	0	0	0	0	0	0	-1H	0	-1H	0	-1H	0	-1H	0	0	1
a8	0	0	0	0	0	0	0	-1H	0	-1H	0	0	0	-1	0	0
a9	0	0	0	0	0	0	0	0	0	+1H	0	+1H	0	0	0	1
a10	0	0	0	0	0	0	0	0	0	-2H	0	0	-2H	0	0	1

Table 40 Eliminated subexpressions for maximum 6 non-zero digits

6.12 Summary of Application to Previous Results

Shown in Table 41 is a summary of the results of common subexpression elimination on each of the filters. The reduction ranges from 24.1 to over 32.5 %. Even for the filter with only two non-zero digits per coefficient the method was able to reduce the number of additions required by 30%.

Filter (by non-zero digit count)	Original Addition Count	Reduction (additions)	Additions Required After Elimination	Reduction (%)
6	45	14	31	31.1%
5	36	10	26	27.7%
4	40	13	27	32.5%
3	29	7	22	24.1%
2	20	6	14	30.0%

Table 41 Summary of Application to Previous Results

6.13 Conclusion

The method presented in this chapter allows for the efficient and simultaneous elimination of both vertical and horizontal common subexpressions occurring within a filter's coefficient multiplication expressions. A computational savings of up to 31% has been demonstrated.

CHAPTER 7. FUTURE WORK

Several areas discussed in this dissertation deserve further examination for possible improvements in filter throughput.

7.1 1-D Recursive Filters

The penalty factor was empirically determined using a single representative example. More examples using different filter designs should be taken to see if what if any effects it has on the optimum penalty factor value.

7.2 2-D Filters

The 2-D filters in this dissertation used three parallel branches and all were treated equally. It is possible that some branches are more important to the overall filter that others. Studies should be taken to ascertain the contribution of each branch and to see if it may be possible to use lower order 1-D component filters on the less important branches.

7.3 Common Sub-expression Elimination

The amount of common subexpression elimination is dependent on bit pattern of coefficients so it may be possible to integrate common subexpression elimination into CSD design algorithm. Such a scheme would use the amount of CSE reduction as part of GA search criteria along with error during design process.

In this dissertation only 2-bit common sub-expressions were considered. It may be possible to gain some additional elimination through the use of n-bit common sub-expression elimination.

CHAPTER 8. CONCLUSION

In this dissertation, high throughput digital filters have been achieved by using CSD filter coefficients and eliminating as many common sub-expressions as possible

Genetic Algorithms have been successfully applied to the design of CSD coefficient filters through the use of a proposed new chromosome coding technique that eliminates the problems previously encountered in using GAs to design such filters.

A new unstable penalty factor has been empirically determined that allows Genetic Algorithms to efficiently handle the unstable filter constraint inherent in recursive filter design. A techniques has been presented that allows these methods to be used to create high throughput 2-D filters

A proposed new graphical transformation allows for optimization of the elimination of CSD- coefficient common sub-expressions in both the vertical and horizontal dimensions. This improves coefficient multiplication efficiency resulting in increased filter throughput.

This proposed new techniques have been shown to work on both recursive and non-recursive filters for both 1-D and 2-D filters. These effectiveness new methods has been demonstrated with example designs and comparisons to other methods.

References

[1] A. Antoniou, Digital Filters: Analysis and Design. New York: McGraw-Hill, 1979.

[2] Ashrafzadeh and B. Nowrouzian, "Crossover and Mutation in Genetic Algorithms Employing Canonical Signed-Digit Number System," *Proceedings of the 1997 Midwest Symposium on Circuits and Systems*, pp. 702–705, Aug. 1997.

[3] A. T. G. Fuller, B. Nowrouzian.and F. Ashrafzade,. "Optimization of FIR digital filters over the canonical signed-digit coefficient space using genetic algorithm," *Proceedings of the 1998 Midwest Symposium on Circuits and Systems*, pp. 456-459 , Aug. 1998.

[4] A. Lee, M. Ahmadi, G.A Julien, W.C. Miller, R.S. Lashkari. "Digital Filter Design Using Genetic Algorithms," *Proceedings of IEEE DFSP'98*, pp. 34–38, Victoria, B.C., June 1998.

[5] G. Wade, A. Roberts and G. Williams, "Multiplier-less FIR Filter Design Uising a Genetic Algorithm," *IEE Proc-Vis. Image Signal Process*, Vol. 141, No.3,pp. 175-180, June 1994.

[6] Li Liang; Ahmadi, M.; Sid-Ahmed, M.; Wallus, K, "Design of canonical signed digit IIR fiilters using genetic algorithm," *Signals, Systems & Computers, 2003 The Thrity-Seventh Asilomar Conference on , Volume: 2* , pp:2043-2047, Nov. 2003.

[7] K. Hwang, Computer Arithmetics Principles, Architecture, and Design. John Wiley & Sons, 1979.

[8] N Benvenuto, N.; M. Marchesi, "Digital Filters Design by Simulated Annealing," *Circuits and Systems, IEEE Transactions on , Volume: 36 , Issue: 3* , pp.:459-460, March 1989.

[9] D.E. Goldberg, Genetic Algorithms in Search Optimization and Machine Learning. Addison-Wesley, 1989.

[10] D.E. Goldberg and J. Richardson, "Genetic algorithms with sharing for multimodal function optimization", *In J.J. Grefenstette, editor, Proceedings of the Second International Conference on Genetic Algorithms, pages 41--49.* Lawrence Erlbaum Associates, 1987.

[11] J.H. Holland. Adaptation in Natural and Artificial Systems. MIT Press, 1975.

[12] J. Antonisse, "A new interpretation of schema notation that overturns the binary encoding constraint," In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 86--91. Morgan Kaufmann, 1989.

[13] P. Field. "A Multary Theory for Genetic Algorithms: Unify Binary and Nonbinary Problem Representations," Ph.D. thesis, University of London, 1996.

[14] K. DeJong, "Genetic algorithms: A 10 year perspective.", In J.J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 169--177. Lawrence Erlbaum Associates, 1985.

[15] G. Syswerda, "Uniform crossover in genetic algorithms," In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2-9. Morgan Kaufmann, 1989.

[16] L.J. Eshelman, R. Caruna, and J.D. Schaffer, "Biases in the crossover landscape," In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 10--19. Morgan Kaufmann, 1989.

[17] W.M. Spears and K. DeJong, "An analysis of multi-point crossover," In G.J.E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 301--315. Morgan Kaufmann, 1991.

[18] K. DeJong and W.M. Spears, "An analysis of the interacting roles of population size and crossover in genetic algorithms," In H.-P. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, pages 38--47. Springer-Verlag, 1990.

[19] J.D. Schaffer and A. Morishma, "An adaptive crossover distribution mechanism for genetic algo rithms," In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms,* pages 36-40. Lawrence Erlbaum Associates, 1987.

[20] J.H. Holland, "Genetic algorithms and classifier systems: foundations and future directions," In J.J. Grefenstette, editor, P*roceedings of the Second International Conference on Genetic Algorithms*, pages 82-89. Lawrence Erlbaum Associates, 1987.

[21] Y. Davidor, "A genetic algorithm applied to robot trajectory generation," In L. Davis, editor, Handbook of Genetic Algorithms, chapter 12, pages 144--165. Van Nostrand Reinhold, 1991.

[22] S.J. Louis and G.J.E. Rawlins, "Designer genetic algorithms: Genetic algorithms in structure design," In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 53--60. Morgan Kaufmann, 1991.

[23] J.D. Schaffer, R.A. Caruna, Eshelman L.J., and R. Das, "A study of control parameters affecting online performance of genetic algorithms for function optimization," In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 51-60. Morgan Kaufmann, 1989.

[24] William M. Spears, "Crossover or mutation?", In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms*, 2, pages 221-237. Morgan Kaufmann, 1993.

[24] J.D. Schaffer and L.J. Eshelman, "On crossover as an evolutionarily viable strategy," In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 61-68. Morgan Kaufmann, 1991.

[26] D.E. Goldberg, "Simple genetic algorithms and the minimal, deceptive problem." In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, chapter 6, pages 74--88. Pitman, 1987.

[27] John J. Grefenstette, "Deception considered harmful," In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms*, 2, pages 75--91. Morgan Kaufmann, 1993.

[28] L. Davis, "Bit climbing, representational bias and test suite design," In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 18—23.

[29] L. Davis and S. Coombs, "Genetic algorithms and communication link speed design: theoretical considerations," In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 252-256. Lawrence Erlbaum Associates, 1987.

[30] L. Davis, "Applying adaptive algorithms to epistatic domains," *In 9th Int. Joint Conf. on AI,* pages 162-164, 1985.

[31] C.Z. Janikow and Z. Michalewicz, "An experimental comparison of binary and floating point representations in genetic algorithms," In R.K. Belew and L.B. Booker, editors, P*roceedings of the Fourth International Conference on Genetic Algorithms*, pages 31--36. Morgan Kaufmann, 1991.

[32] K. DeJong and W.M. Spears, "An analysis of the interacting roles of population size and crossover in genetic algorithms," In H.-P. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, pages 38--47. Springer-Verlag, 1990.

[33] Ashrafzadeh and B. Nowrouzian, "Crossover and Mutation in Genetic Algorithms Employing Canonical Signed-Digit Number System," P*roceedings of the 1997 Midwest Symposium on Circuits and Systems*, pp. 702–705, Aug. 1997.

[34] A. T. G. Fuller, B. Nowrouzian.and F. Ashrafzade, "Optimization of FIR digital filters over the canonical signed-digit coefficient space using genetic algorithm," *Proceedings of the 1998 Midwest Symposium on Circuits and Systems, pp. 456-459 , Aug. 1998.*

[35] Li Liang; Ahmadi, M.; Sid-Ahmed, M.; Wallus, "Design of canonical signed digit IIR fiilters using genetic algorithm," Signals, Systems & Computers, 2003 The Thrity-Seventh Asilomar Conference on ,Volume: 2 , pp:2043-2047, Nov. 2003.

[36] A. Lee, M. Ahmadi, G.A Julien, W.C. Miller, R.S. Lashkari, "Digital Filter Design Using Genetic Algorithms," *Proceedings of IEEE DFSP'98*, pp. 34–38, Victoria, B.C., June 1998.

[37] G. Wade, A. Roberts and G. Williams, "Multiplier-less FIR Filter Design Uising a Genetic Algorithm," I*EE Proc-Vis. Image Signal Process, Vol. 141, No.3*, pp. 1-180, June 1994.

[38] W. S. Lu and A. Antoniou, "Two Dimensional Digital Filters" New York: Marcel Dekker, 1992.

[39] R. King, M. Ahmadi, R. Gorgui-Naguib, A. Kwabwe and M. Azimi-Sadjadi, "Digital Filtering in One and Two Dimensions; Design and Applications" Plenum press, 1989.

[40] W. S. Lu and A. Antoniou, "Application of the singular value decomposition to the design of two-dimensional digital filters," in *Control and Dynamic Systems*, ed. by C.T. Leondes, vol. 69, pp.181-210, Academic Press, San Diego, 1995.

[41] A. Antoniou, W.S. Lu, "Design of Two-Dimensional Digital Filters by Using the Singular Value Decomposition," I*EEE Trans. CAS-34*, pp.1191-1198, 1987.

[42] N. E. Mastorakis, I. F. Gonos, and M. N. S. Swamy, "Design of two-dimensional recursive filters using genetic algorithms," IEEE Trans. Circuits Syst. I, vol. 50, pp. 634–639, May 2003.[43] M. Potkonjak, M. B. Shrivasta and P. A. Chandrakasan, "Multiple constant multiplications: Efficient and versatile framework and algorithms for exploring common subexpression elimination, IEEE Trans. Computer-Aided Design, vol. 15, no. 2, pp. 151-161, Feb. 1996.

[44] R. I. Hartley, "Subexpression sharing in filters using canonic signed digit multipliers," IEEE Trans. Ckts. Syst. II, vol. 43, pp. 677-688, Oct. 1996.

[45] R. Pasko, P. Schaumont, V. Derudder, S. Vernalde, and D. Durackova, "A new algorithm for elimination of common subexpressions," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Syst., vol. 18, no. 1, pp. 58-68, January 1999.

[46] M. M. Peiro, E. I. Boemo, and L. Wanhammar, "Design of high-speed multiplierless filters using a nonrecursive signed common subexpression algorithm," IEEE Trans. Ckts. Syst. II, vol. 49, no. 3, pp. 196-203, March 2002.

[47] Y. Jang and S.Yang, Low-power CSD linear phase FIR filter structure using vertical common subexpression, Electronics Letters, vol. 38, no. 15, pp. 777-779, July 2002.

[48] A.P. Vinod and E. M-K. Lai, "Comparison of the Horizontal and the Vertical Common Subexpression Elimination Methods for Realizing Digital Filters", Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS), May 2005, Kobe, Japan.

[49] F. Xu, C. H. Chang and C. C. Jong, "Efficient Algorithms for Common Subexpression Elimination in Digital Filter Design," in *Proc. IEEE Int. Conf. on Speech, Acoustics, and Signal Processing*, Montreal, Canada, Vol. V, pp. 137-140, May, 2004.

Appendix A Source Code

The following is the C++ source code for the Genetic Algorithm.

File of unit1.h

```
//---------------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <system.hpp>
#include "sgr_data.hpp"
#include "sgr_def.hpp"
#include <Menus.hpp>
#include <ExtCtrls.hpp>
#include"Chromosome.h"
#define MINMAXdef 0
#define LMSdef 1
#define STABLE 0
#define UNSTABLE 1
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:            // IDE-managed Components
   TLabel *Label1;
   TLabel *Label2;
   TLabel *Label3;
   TButton *Button1;
   TLabel *Label4;
   TButton *Button2;
   Tsp_XYPlot *sp_XYPlot1;
   Tsp_XYLine *sp_XYLine1;
   TButton *Button3;
   TMainMenu *MainMenu1;
   TPopupMenu *PopupMenu1;
   TMenuItem *File1;
   TMenuItem *Exit1;
   TMenuItem *FileMenu;
   TMenuItem *Exit;
   TMenuItem *FilterMenu;
   TMenuItem *HelpMenu;
   TLabel *Label5;
   TLabel *Label6;
   TLabel *Label7;
   TLabel *Label8;
   TMenuItem *LPFilter;
   TMenuItem *HighPassFilter1;
   TMenuItem *SettingsMenu;
   TMenuItem *PassBandFilter1;
   TMenuItem *Coefficients2;
```

```
    TMenuItem *Algorithm2;
    TMenuItem *LowPass1;
    TMenuItem *HighPass1;
    TMenuItem *BandPass1;
    TMenuItem *Notch1;
    TMenuItem *Arbitrary1;
    TMenuItem *Frequency1;
    TMenuItem *Radssec1;
    TMenuItem *Hz1;
    TLabel *Label9;
    void __fastcall Button1Click(TObject *Sender);
    void __fastcall Button2Click(TObject *Sender);
    void __fastcall Button3Click(TObject *Sender);
    void __fastcall Coefficients2Click(TObject *Sender);
    void __fastcall Algorithm2Click(TObject *Sender);
    void __fastcall ArbitraryFilter1Click(TObject *Sender);
    void __fastcall LowPass1Click(TObject *Sender);
    void __fastcall Arbitrary1Click(TObject *Sender);
    void __fastcall FormClose(TObject *Sender, TCloseAction &Action);
private:    // User declarations

public:
            // User declarations
    char elitism ;
    int FitnessType;
            int xoverType;  // 0 = uniform, 1 = 1 point, 2 = 2 point
            int popSize,chromCard,chromLength,NumGens;
    float desiredFitness;
    int numFreqPoints;
            double mutRate,crossRate;
            String fitFunc, chromDecode;
    int numUniqueCoeffs,Ndigits,NcsdOnes,numRecursiveCoeffs,numAi,numBi;
    double PassBandStop;
    double StopBandStart;
    double ws;
    int order;
    bool linearPhase;
    int SaveCSDs(Chromosome *Best,int Gen);
    int stats(char *buffer1);
    int unstablePenalty;
    double *fresp;
    double *target;

    __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//extern void sgenrand(unsigned long);
extern "C" void sgenrand(unsigned long);
extern "C" double genrand(void);
//---------------------------------------------------------------------------
#endif
```

File Unit1.cpp:

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
```

```
//#include "FitFunc.h"
#include "ArbitraryForm.h"
#include "Unit2.h"
#include "Unit3.h"
#include "Unit4.h"
#include "Unit5.h"
#include "Unit1.h"
#include <stdlib.h>
#include "MTRand.h"
#define GC_DEBUG
#include "gc_cpp.h"
#include "GA.h"
#include <sys\timeb.h>


//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma link "sgr_data"
#pragma link "sgr_def"
#pragma resource "*.dfm"
//---------------------------------------------------------------------------

#define UNSTABLEPENALTY 9
int numTrials = 100;
int unstablePenalty = UNSTABLEPENALTY;
GA *theGaPtr;
Chromosome *GlobalBest = NULL;
int GenCount,numAi,numBi;
TForm1 *Form1;
bool quit;
char buff2[60];
double SDweights[41]= {1.0, 5.e-1, 2.5e-01, 1.25e-01, 6.25e-02, 3.125e-02, 1.5625e-02, 7.8125e-03, 3.90625e-03, 1.953125e-03,
9.765625e-04, 4.882812e-04, 2.441406e-04, 1.220703e-04, 6.103516e-05, 3.051758e-05, 1.525879e-05, 7.629395e-06, 3.814697e-
06, 1.907349e-06, 9.536743e-07, 4.768372e-07, 2.384186e-07, 1.192093e-07, 5.960464e-08, 2.980232e-08, 1.490116e-08,
7.450581e-09, 3.72529e-09, 1.862645e-09, 9.313226e-10, 4.656613e-10, 2.328306e-10, 1.164153e-10, 5.820766e-11, 2.910383e-
11, 1.455192e-11, 7.275958e-12, 3.637979e-12, 1.818989e-12, 9.094947e-13};
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
// Default Filter Values

order = 20; // 10; //20; //21; // filter order
// set numRecCoeffs to 0 for FIR
numRecursiveCoeffs = 4; //5 for 4th order //10; //2; // Number of b[i]'s (a[i]'s = order - numRecursiveCoeffs)
linearPhase= false; //true;
ws = M_PI * 2.0;

// CSD Values
Ndigits= 16; // 20; // Number of CSD digits
NcsdOnes = 3; // 10;//3; // Number of of non-zeros allowed in CSD


// GA settings
elitism = 1; // 0 = off 1=on;
xoverType = 0;
popSize = 500;
mutRate = 0.05;
crossRate = 0.95;
```

```
NumGens = 500; //20000;  //0;
desiredFitness = 9999999;      // set to a 9999999 for unlimited
//FitnessType = MINMAXdef;
FitnessType = LMSdef;    // Least mean Square error
// Set up Mersenne Twister MT19937 Pseudorandom Number Generator
randomize();
unsigned long seed = random(32000);    // use built in PNG to get seed for MT19937
sgenrand(seed);
unstablePenalty = UNSTABLEPENALTY;

}
//---------------------------------------------------------------------------

void __fastcall TForm1::Button1Click(TObject *Sender)
{

        // Calc some Variables
        if (linearPhase && numRecursiveCoeffs == 0)
          numUniqueCoeffs = ((order-1)/2) +1; // Number of unique filter coefficients
        else
          numUniqueCoeffs = order;
        numBi =  numRecursiveCoeffs;
        numAi = numUniqueCoeffs-numBi;
        numUniqueCoeffs = numAi + numBi;
                                chromLength = numUniqueCoeffs*NcsdOnes;
        chromCard = Ndigits*2;
        FilterSpec *theFilterSpec = new (GC) FilterSpec(
              target, fresp, ws, numFreqPoints, numUniqueCoeffs);
FILE *p;
int nonZeroCount;
p = fopen ("0Timeouts.txt" ,"w");

sprintf(buff2,"CSD digits = %d, Non-zeros = %d\n",Ndigits,NcsdOnes);
fprintf(p,"Results for  %s",buff2);
fprintf(p,"time (seconds), Pbripple,  Sbgain,  Number of Non-zeroes, Number of Gens\n");
Form1->Label9->Caption = buff2;
flushall();
for (int i=1;i<=numTrials && quit == false;i++) {
  //float diff;
  struct timeb tm;
  float start,stop;
  ftime(&tm);
  start = (tm.time -1048196000)+ tm.millitm/1000.0;
        GA *theGa = new (GC) GA( elitism, xoverType,
                                                popSize,chromCard,chromLength,NumGens,desiredFitness,
                                                mutRate, crossRate,theFilterSpec,chromDecode,
              numUniqueCoeffs,NcsdOnes,Ndigits);
        theGaPtr=theGa;
        theGa->PerformGA(&GlobalBest);
        delete theGa;
  ftime(&tm);
  stop = (tm.time - 1048196000) + tm.millitm/1000.0;
  double Pbripple,Sbgain;
  Pbripple = Sbgain =0;  //GlobalBest->getFreqStats(&Pbripple,&Sbgain);
  nonZeroCount = SaveCSDs(GlobalBest,i);
  fprintf(p,"Trial %d, %f %f %f %d %d\n", i, stop-start,Pbripple,Sbgain,nonZeroCount,theGa->generationCount);
  //SaveCSDs(GlobalBest,i);
  //fopen ("0Timeouts.txt" ,"w");
  fflush(p);
}
```

```
 fclose(p);
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
if (theGaPtr->pause) theGaPtr->pause = false;
      else theGaPtr->pause = true;
}
//---------------------------------------------------------------------------



void __fastcall TForm1::Button3Click(TObject *Sender)
{
   if (GlobalBest == NULL ) return;
   ResultsForm->Visible = true;


   //double value;
   int numpoints = numFreqPoints; //200;
   ResultsForm->sp_XYPlot1->BufferedDisplay = true;
   ResultsForm->sp_XYLine1->Clear();
   ResultsForm->sp_XYLine2->Clear();
   Freq *fr = new (GC) Freq[numpoints];
   fr = GlobalBest->FreqResp(numpoints);
   for (int i = 0; i < numpoints; i++) {
        ResultsForm->sp_XYLine1->AddXY(fr[i].w,fr[i].mag);
        ResultsForm->sp_XYLine2->AddXY(fr[i].w,Form1->target[i]);
   }
   ResultsForm->sp_XYPlot1->Paint();
   int c,j=0,i=0;
   char *Buffer = new (GC) char[5000]; //Create Buffer for display
   stats(Buffer);
    char *lineBuffer = new (GC) char[50];
    ResultsForm->Memo1->Lines->Clear();
    for (c=Buffer[i++];c!='\0';c=Buffer[i++]) {
     if(c=='\n') {
         lineBuffer[j]='\0';
         ResultsForm->Memo1->Lines->Add(lineBuffer);
         j=0;
     }
     else          {
        lineBuffer[j++]=(char)c;
     }
    }
    ResultsForm->Memo1->Lines->EndUpdate();

}
//---------------------------------------------------------------------------
void setGlobalBest(Chromosome *p){
    GlobalBest=p;
    }

int TForm1::stats(char *display){
  double value=0,values[100];
  int nonzeroes=0,count=0,i;
  char *Buffer = new (GC) char[500];  //Creates Buffer dynamic variable
  double **CSDs = GlobalBest->toCSDGenotype();;
   sprintf(Buffer,"At Generation %d: \n",theGaPtr->generationCount);strcat (display, Buffer);
   sprintf(Buffer,"Best Fitness %f: \n",theGaPtr->bestFitness);strcat (display, Buffer);
```

```
for (int nc = 0; nc<GlobalBest->numUniqueCoeffs; nc++) {
   sprintf(Buffer,"CSD %2d: ",nc);strcat (display, Buffer);
   for (int i = 0; i<GlobalBest->Ndigits; i++) {
      if  (CSDs[nc][i] == 0)
         strcat (display, " 0");
      if  (CSDs[nc][i] == 1) {
         strcat (display, " 1");
         value = value +  SDweights[i];
         nonzeroes++;
      }
      if  (CSDs[nc][i] == -1) {
         strcat (display, "-1");
         value = value -  SDweights[i];
         nonzeroes++;
      }
   }
   sprintf(Buffer," %21.18f: \n",value);strcat (display, Buffer);
   values[count++] = value;
   value=0;
}
int nuc;
if (linearPhase) {  // do the rest of the Coeffs
    sprintf(Buffer,"Linear Phase Filter: Repeat Coeffs\n"); strcat (display, Buffer);
    if (order%2 == 0 ) nuc = GlobalBest->numUniqueCoeffs-1;  // even order --> odd # of coeffs -> use middle coeff only once
    else   nuc = GlobalBest->numUniqueCoeffs;  // odd order --> even # of coeffs -->  use all coeffs twice
    for (int nc = nuc-1; nc>=0; nc--) {
        sprintf(Buffer,"CSD %2d: ",nc);strcat (display, Buffer);
        value=0;
        for (int i = 0; i<GlobalBest->Ndigits; i++) {
            if  (CSDs[nc][i] == 0)
               strcat (display, " 0");
            if  (CSDs[nc][i] == 1) {
               strcat (display, " 1");
               value = value +  SDweights[i];
               nonzeroes++;
            }
            if  (CSDs[nc][i] == -1) {
               strcat (display, "-1");
               value = value -  SDweights[i];
               nonzeroes++;
            }
        }
        sprintf(Buffer,"%21.18f: \n",value);strcat (display, Buffer);
        values[count++] = value;
        value=0;
    }

 }
 sprintf(Buffer,"The total number of non-zero digits is %d\n",nonzeroes);
 strcat (display, Buffer);
 if (linearPhase) {
  sprintf(Buffer,"linearPhase = [ ");
  for (i=count/2;i<count-1;i++) {
    sprintf(Buffer,"%15.10f ; ",values[i]);
    strcat (display, Buffer);
  }
  for (i=0;i<count/2;i++) {
    sprintf(Buffer,"%15.10f ; ",values[i]);
    strcat (display, Buffer);
  }
```

```
    sprintf(Buffer,"] \n");
    strcat (display, Buffer);
    sprintf(Buffer,"freqz(linearPhase,1,1024,2*pi); filterSqErr(linearPhase,1)\n");
    strcat (display, Buffer);
   }

   double *Coeffs;
   Coeffs = GlobalBest->toCSDPhenotype();
   if (!linearPhase && numBi==0) {
    sprintf(Buffer,"Coeffs = [ ");
    strcat (display, Buffer);
    for (i=0;i<count;i++) {
      sprintf(Buffer,"%f ; ",Coeffs[i]);
      strcat (display, Buffer);
    }
    sprintf(Buffer,"] \n");
    strcat (display, Buffer);
    sprintf(Buffer,"freqz(Coeffs,1,1024,2*pi); [r,f]=freqz(Coeffs,1,1024,2*pi);Err = sum((phi1-abs(r)).^2)\n");
    strcat (display, Buffer);
   }

   //IIR
   if (numBi > 0 ) {
     sprintf(Buffer,"Ai = [ ");
     strcat (display, Buffer);
     for (i=0;i<numAi;i++) {
       sprintf(Buffer,"%15.12f ",Coeffs[i]);
       strcat (display, Buffer);
     }
     sprintf(Buffer,"] \n");
     strcat (display, Buffer);

     sprintf(Buffer,"Bi = [ ");
     strcat (display, Buffer);
     for (i=0;i<numBi;i++) {
       sprintf(Buffer,"%15.12f ",Coeffs[i+numAi]);
       strcat (display, Buffer);
     }
     sprintf(Buffer,"] \n");
     strcat (display, Buffer);
     sprintf(Buffer,"freqz(Ai,Bi,1024,2*pi);figure; zplane(Ai,Bi); [r,f]=freqz(Ai,Bi,1024,2*pi);Err = sum((phi1-abs(r)).^2)\n");
     strcat (display, Buffer);

if (GlobalBest->stable == STABLE)
       sprintf(Buffer,"This filter is stable\n");
     else
       sprintf(Buffer,"This filter is unstable\n");
     strcat (display, Buffer);
   }

 return 0;
 }

void __fastcall TForm1::Coefficients2Click(TObject *Sender)
{
CSDSettings->Visible = true;
CSDSettings->Edit1->Text= AnsiString(Ndigits);
CSDSettings->Edit2->Text= AnsiString(NcsdOnes);
}
//---------------------------------------------------------------------------
```

```
void __fastcall TForm1::Algorithm2Click(TObject *Sender)
{
AlgorithmSettings->Visible = true;
AlgorithmSettings->Edit1->Text= AnsiString(popSize);
AlgorithmSettings->Edit2->Text= AnsiString(NumGens);
AlgorithmSettings->Edit3->Text= AnsiString(numFreqPoints);
AlgorithmSettings->Edit4->Text= AnsiString(mutRate);
AlgorithmSettings->Edit5->Text= AnsiString(crossRate);
if (FitnessType == LMSdef)
   AlgorithmSettings->RadioButton2->Checked = true;
else
   AlgorithmSettings->RadioButton1->Checked = true;;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::ArbitraryFilter1Click(TObject *Sender)
{
Arbitrary->Visible = true;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::LowPass1Click(TObject *Sender)
{
LPFilterSettings->Visible = true;
LPFilterSettings->Edit1->Text= AnsiString(PassBandStop);
LPFilterSettings->Edit2->Text= AnsiString(StopBandStart);
LPFilterSettings->Edit3->Text= AnsiString(ws);
LPFilterSettings->Edit4->Text= AnsiString(order);
}
//---------------------------------------------------------------------------

void __fastcall TForm1::Arbitrary1Click(TObject *Sender)
{
 Arbitrary->Visible = true;
}
//---------------------------------------------------------------------------


void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
 if (MessageDlg("Close application ?", mtConfirmation, TMsgDlgButtons() << mbYes << mbNo,0) == mrYes){
   quit = true;
   theGaPtr->quit = true;
   Action = caFree;
 }
 else
   Action = caMinimize;

}
//---------------------------------------------------------------------------
int TForm1::SaveCSDs(Chromosome *Best,int Run){
   // put CSD's in a file and return the number of non-zero digits
   if (GlobalBest == NULL ) return 0;
   int nonZeroCount=0;

   //double value=0.0;
   FILE *p;
   char name[80],Buffer1[5000];
   sprintf(name,"CSDout\\run%d.txt",Run);
```

```
  p = fopen (name,"w");
  fprintf(p,"Results for  %s",buff2);
  fprintf(p,"The best filter for Generation %d has these CSDs:\n",Run);

   stats(Buffer1);
   fprintf(p,"%s",Buffer1);
   fclose(p);
   return nonZeroCount;
}
```

File Unit2.h:

```
//---------------------------------------------------------------------------
#ifndef Unit2H
#define Unit2H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include "sgr_data.hpp"
#include "sgr_def.hpp"
//---------------------------------------------------------------------------
class TResultsForm : public TForm
{
__published:            // IDE-managed Components
   Tsp_XYPlot *sp_XYPlot1;
   Tsp_XYLine *sp_XYLine1;
   Tsp_XYLine *sp_XYLine2;
   TButton *Button1;
   TMemo *Memo1;
   TLabel *Label1;
   TLabel *Label2;
   TLabel *Label3;
   void __fastcall Button1Click(TObject *Sender);
private:    // User declarations
public:                 // User declarations
   __fastcall TResultsForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TResultsForm *ResultsForm;
//---------------------------------------------------------------------------
#endif
```

File Unit2.cpp:

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit2.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma link "sgr_data"
#pragma link "sgr_def"
```

```
#pragma resource "*.dfm"
TResultsForm *ResultsForm;
//---------------------------------------------------------------------------
__fastcall TResultsForm::TResultsForm(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TResultsForm::Button1Click(TObject *Sender)
{
Close();
}
//---------------------------------------------------------------------------
```

File Unit3.h:

```
//---------------------------------------------------------------------------
#ifndef Unit3H
#define Unit3H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class TLPFilterSettings : public TForm
{
__published:            // IDE-managed Components
    TLabel *Label1;
    TLabel *Label2;
    TLabel *Label3;
    TButton *Button1;
    TEdit *Edit1;
    TButton *Button2;
    TEdit *Edit2;
    TEdit *Edit3;
    TLabel *Label4;
    TLabel *Label5;
    TEdit *Edit4;
    void __fastcall Button1Click(TObject *Sender);
    void __fastcall Button2Click(TObject *Sender);
private:    // User declarations
public:                 // User declarations
    __fastcall TLPFilterSettings(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TLPFilterSettings *LPFilterSettings;
//---------------------------------------------------------------------------
#endif
```

File Unit3.cpp:

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
```

```
#include "Unit3.h"
#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TLPFilterSettings *LPFilterSettings;
//---------------------------------------------------------------------------
__fastcall TLPFilterSettings::TLPFilterSettings(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------------


void __fastcall TLPFilterSettings::Button1Click(TObject *Sender)
{
Form1->PassBandStop = Edit1->Text.ToDouble();
Form1->StopBandStart = Edit2->Text.ToDouble();
Form1->ws = Edit3->Text.ToDouble();
Form1->order = Edit4->Text.ToDouble();
Close();
}
//---------------------------------------------------------------------------

void __fastcall TLPFilterSettings::Button2Click(TObject *Sender)
{
Close();
}
//---------------------------------------------------------------------------
```

File Unit4.h:

```
//---------------------------------------------------------------------------
#ifndef Unit4H
#define Unit4H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class TCSDSettings : public TForm
{
__published:            // IDE-managed Components
    TLabel *Label1;
    TLabel *Label2;
    TEdit *Edit1;
    TEdit *Edit2;
    TButton *Button2;
    TButton *Button1;
    void __fastcall Button1Click(TObject *Sender);
    void __fastcall Button2Click(TObject *Sender);
private:    // User declarations
public:                 // User declarations
    __fastcall TCSDSettings(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TCSDSettings *CSDSettings;
```

```
//---------------------------------------------------------------------------
#endif
```

File Unit4.cpp:

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit4.h"
#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TCSDSettings *CSDSettings;
//---------------------------------------------------------------------------
__fastcall TCSDSettings::TCSDSettings(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------------------------------

void __fastcall TCSDSettings::Button1Click(TObject *Sender)
{


Form1->Ndigits = Edit1->Text.ToDouble();  // Number of CSD digits
Form1->NcsdOnes = Edit2->Text.ToDouble();
Close();
}
//---------------------------------------------------------------------------

void __fastcall TCSDSettings::Button2Click(TObject *Sender)
{
Close();
}
//---------------------------------------------------------------------------
```

File Unit5.h:

```
//---------------------------------------------------------------------------
#ifndef Unit5H
#define Unit5H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class TAlgorithmSettings : public TForm
{
__published:            // IDE-managed Components
  TLabel *Label1;
  TLabel *Label2;
  TLabel *Label3;
  TLabel *Label4;
```

```
    TLabel *Label5;
    TEdit *Edit1;
    TEdit *Edit2;
    TEdit *Edit3;
    TEdit *Edit4;
    TEdit *Edit5;
    TButton *Button1;
    TButton *Button2;
    TGroupBox *GroupBox1;
    TRadioButton *RadioButton1;
    TRadioButton *RadioButton2;
    void __fastcall Button1Click(TObject *Sender);
    void __fastcall Button2Click(TObject *Sender);
    void __fastcall RadioButton1Click(TObject *Sender);
    void __fastcall RadioButton2Click(TObject *Sender);
private:    // User declarations
public:                 // User declarations
    __fastcall TAlgorithmSettings(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TAlgorithmSettings *AlgorithmSettings;
//---------------------------------------------------------------------------
#endif
```

File Unit5.cpp:

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit5.h"
#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TAlgorithmSettings *AlgorithmSettings;
//---------------------------------------------------------------------------
__fastcall TAlgorithmSettings::TAlgorithmSettings(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TAlgorithmSettings::Button1Click(TObject *Sender)
{
Form1->popSize = Edit1->Text.ToDouble();  // Number of CSD digits
Form1->NumGens = Edit2->Text.ToDouble();
Form1->numFreqPoints = Edit3->Text.ToDouble();
Form1->mutRate = Edit4->Text.ToDouble();
Form1->crossRate = Edit5->Text.ToDouble();
Close();
}
//---------------------------------------------------------------------------
void __fastcall TAlgorithmSettings::Button2Click(TObject *Sender)
{
Close();
}
//---------------------------------------------------------------------------
```

```
void __fastcall TAlgorithmSettings::RadioButton1Click(TObject *Sender)
{
Form1->FitnessType = MINMAXdef;
}
//---------------------------------------------------------------------------

void __fastcall TAlgorithmSettings::RadioButton2Click(TObject *Sender)
{
Form1->FitnessType = LMSdef;
}
//---------------------------------------------------------------------------
```

File ArrayGen.h:

```
#ifndef Array_Gen_H
#define Array_Gen_H
#define GC_DEBUG
#include "gc_cpp.h"
#include <complex.h>


class ArrayGen:public gc {
  public:
    int **new2D(int **,int,int);
    double **new2D(double **,int,int);
    float **new2D(float **,int,int);
    complex<double>** new2D(complex<double>**,int,int);
    void del2D(int **p,int m);
    void del2D(double **p,int m);
    void del2D(float **p,int m);

};

#endif
```

File ArrayGen.cpp:

```
#include "ArrayGen.h"
#include <vcl.h>
#include <exception>
#include <iostream.h>


// Allocates and dealloacates 2D Arrays
complex<double>** ArrayGen::new2D(complex<double>**p,int m,int n) {
 try {                    // TEST FOR EXCEPTIONS.
   p  = new (GC) complex<double> *[m];
   for (int j = 0; j < m; j++)
     p[j]= new (GC) complex<double>[n];
 }
 catch (std::bad_alloc) {  // bad_alloc THROWN.
    Application->MessageBox("Out of Memory","Memory Error",IDOK);
    }
  return p;
}
int** ArrayGen::new2D(int **p,int m,int n) {
```

```
 try {                    // TEST FOR EXCEPTIONS.
   p  = new (GC) int *[m];
   for (int j = 0; j < m; j++)
      p[j]= new (GC) int[n];
 }
 catch (std::bad_alloc) {  // bad_alloc THROWN.
     Application->MessageBox("Out of Memory","Memory Error",IDOK);
    }
   return p;
}

double** ArrayGen::new2D(double **p,int m,int n) {
 try {
   p  = new (GC) double *[m];
   for (int j = 0; j < m; j++)
      p[j]= new (GC) double[n];
 }
 catch (std::bad_alloc) {  // bad_alloc THROWN.
     Application->MessageBox("Out of Memory","Memory Error",IDOK);
 }
   return p;
}

float** ArrayGen::new2D(float **p,int m,int n) {
 try {
   p  = new (GC) float *[m];
   for (int j = 0; j < m; j++)
      p[j]= new (GC) float[n];
 }
 catch (std::bad_alloc) {  // bad_alloc THROWN.
     Application->MessageBox("Out of Memory","Memory Error",IDOK);
 }
  return p;
}

void ArrayGen::del2D(float **p,int m) {
  for (int i = 0; i < m;  i++)
     delete[] p[i];             // STEP 1: DELETE THE COLUMNS
  delete[] p;                   // STEP 2: DELETE THE ROWS

}

void ArrayGen::del2D(int **p,int m) {
  for (int i = 0; i < m;  i++)
     delete[] p[i];             // STEP 1: DELETE THE COLUMNS
  delete[] p;                   // STEP 2: DELETE THE ROWS

}

void ArrayGen::del2D(double **p,int m) {
  for (int i = 0; i < m;  i++)
     delete[] p[i];             // STEP 1: DELETE THE COLUMNS
  delete[] p;                   // STEP 2: DELETE THE ROWS


}
```

File Chromosome.h:

```
#ifndef SUPPORT_H
#define SUPPORT_H
extern "C" void sgenrand(unsigned long);
extern "C" double genrand(void);
#include <system.hpp>
#define GC_DEBUG
#include "gc_cpp.h"
#include "FiltSpec.h"
#include <complex.h>
#define STABLE 0
#define UNSTABLE 1
#define MAXORDER 100
class Freq :public gc{
  public:
    double w;
    double mag;
    double phase;
};

class Chromosome: public gc  {
          private:
          unsigned char *gene;
            int cardinality, Length;
    double  fitness;
    int tmp;
    FilterSpec *theFilterSpec;
    double getFitness(Chromosome *c);
    complex<double> e2jwT(double w);
        // relative fitness  = fitness / sum(population fitness)
          double  rfitness;
    double phenotype();
          // cumulative relative fitness (from chrom[0] to this chrom) for roulette wheel
    double  cfitness;
    int NcsdOnes; // Number of of non-zeros allowed in CSD
    double FreqError(double *Coeffs) ;
  public:
    bool fitnessEvaluated;
    double rfitnessGet();
    double cfitnessGet();
    void rfitnessSet(double value);
    void cfitnessSet(double value);
                String toString(void);
    Chromosome(int cc,int cr,FilterSpec *ff, int nc,
      int ndigits,int ncsdones);
    Chromosome();
    ~Chromosome();
                int getChromosomeLength();
    double getFitness();
    void copyChromosome(Chromosome *c);
                Chromosome *cloneChromosome();
        void initializeChromosomeRandom();
          void clearChromosome();
    unsigned char getGene(int locus);
    void setGene(int locus, unsigned char allele);
    void mutateGene(int locus);
    String toGenotype();
```

```
    String toPhenotype();
    double evalChromosome();
    int numUniqueCoeffs;  // Number of filter coefficients
    int Ndigits; // Number of CSD digits
    double *toCSDPhenotype();
    // **toCSDGenotype() calculates CSD string as 2d array:
    // an array of array of ints where ints = 0,1 or -1
     double **toCSDGenotype();
    Freq *FreqResp(int numpoints);
    Freq *FreqResp(FilterSpec *theFilterSpec);
    int stable;

};

/*************************************************************/
/* Crossover: performs crossover of the two selected parents. */
/*************************************************************/
class Crossover:public gc {
  public:
    int cromLength,xoverType;
    bool *mask;
    bool bit;
    unsigned char xoPoint1, xoPoint2;
    int locus;
    Crossover(int cl, int xot);
    Crossover();
    ~Crossover();
    void xOver(Chromosome *one, Chromosome *two);
};
class MyRandom :public gc{

  public:
    double dbl();  // Returns a random double in [0,1).
    bool boolean();// Return a random boolean (false or true).
    unsigned char integer(int n); // Return a random integer from 1 to n inclusive.

};
/*************************************************************/
/* Selection function: Standard proportional selection for   */
/* maximization problems incorporating elitist model - makes  */
/* sure that the best member survives.                 */
/*************************************************************/
// also called the roulette wheel method
class Selection :public gc{
          public:
    void select(Chromosome **population,
      Chromosome **nextPopulation, int populationSize);
};

#endif
```

File Chromosome.cpp:

```
#include "Chromosome.h"
#include <math.h>
#define GC_DEBUG
#include "gc_cpp.h"
MyRandom MyRandom1;
```

```
double Chromosome::rfitnessGet() {
        return rfitness;
}
double Chromosome::cfitnessGet() {
        return cfitness;
}
void Chromosome::rfitnessSet(double value) {
        rfitness = value;
        }
void Chromosome::cfitnessSet(double value) {
                cfitness = value;
}
String Chromosome::toString() {
                return this->toGenotype();
}
Chromosome::Chromosome() {
        gene = new (GC) unsigned char[Length];
        fitnessEvaluated = false;
        fitness = rfitness = cfitness = 0;

}
Chromosome::Chromosome(int cc,int cr,FilterSpec *ff, int nc,
    int ndigits, int ncsdones) {
  cardinality = cc;
        Length = cr;
      theFilterSpec = ff;
        gene = new (GC) unsigned char[Length];
      fitnessEvaluated = false;
                fitness = rfitness = cfitness = 0;
  numUniqueCoeffs = nc; // Number of filter coefficients
  Ndigits=ndigits; // Number of CSD digits
  NcsdOnes = ncsdones; // Number of of non-zeros allowed in CSD
  stable=STABLE;
}
Chromosome::~Chromosome() {
        delete[] gene;
}
int Chromosome::getChromosomeLength() {
        return Length;
}
double Chromosome::getFitness() {
                if (!fitnessEvaluated)  {
                        fitness = evalChromosome();
    // System.err.println("getFitness: negative fitness so quit");
    // System.exit(1);
    fitnessEvaluated=true;
                }
                return fitness;
}
void Chromosome::copyChromosome(Chromosome *c) { // copy this to c
                Chromosome *destination = c;
                destination->fitness  = this->fitness;
                destination->rfitness = this->rfitness;
                destination->cfitness = this->cfitness;
                fitnessEvaluated = this->fitnessEvaluated;
                for (int  locus = 0;  locus < Length;  locus++) {
                        destination->gene[ locus] = this->gene[locus];
                }
}
void Chromosome::initializeChromosomeRandom() {
```

```
                    for (int locus = 0; locus < Length; locus++) {
                            gene[locus] = MyRandom1.integer(cardinality);
                    }
                    fitnessEvaluated = false;          // set this again since we
                    fitness = rfitness = cfitness = 0;  // may be called more than once
}
unsigned char Chromosome::getGene(int locus) {
                    return gene[locus];
}
void Chromosome::setGene(int locus, unsigned char allele) {
                    gene[locus] = allele;
                    fitnessEvaluated = false;
                    fitness = rfitness = cfitness = 0;
}
void Chromosome::mutateGene(int locus) {
                    // randomize this gene
                    gene[locus] = MyRandom1.integer(cardinality);
                    fitnessEvaluated = false;
                    fitness = rfitness = cfitness = 0;
}
String Chromosome::toGenotype() {
                    String genotype = "";
        for (int locus = 0; locus < Length; locus++) {
                            genotype += AnsiString(gene[locus]);
                    }
                    return genotype;
}
double Chromosome::phenotype() {
                    // evaluate chromosome as a number
                    double value = 0;

            for (int locus = 0; locus < Length; locus++) {
                    value += pow(cardinality, (double)locus )  * gene[Length-locus-1];
                    }
                    return value;
}
String Chromosome::toPhenotype() {
     String x = "x";
                    return x + AnsiString(phenotype());;
}
/*****************************************************************/
/* Crossover: performs crossover of the two selected parents. */
/*****************************************************************/
Crossover::Crossover (int cl, int xot) {
                    cromLength = cl;
                    xoverType= xot; // 0 = uniform, 1 = 1 point, 2 = 2 point
                    mask = new (GC) bool[cromLength];
                    bit =true;
}
Crossover::Crossover () {
                    mask = new (GC) bool[cromLength];
                    bit =true;
}
Crossover::~Crossover() {
     delete mask;
}
void Crossover::xOver(Chromosome *one, Chromosome *two) {
                    unsigned char  temp;
     // create xo mask
                    xoPoint2 = MyRandom1.integer(cromLength-1);
```

```
                    if (xoverType == 1 ) {
                            // one point: start cut at zero
                            xoPoint1 =0;
                    }
                    else {
                            xoPoint1 = MyRandom1.integer(cromLength-1);
                    }
                    if (xoverType == 0 ) {
                            for (locus = 0; locus < cromLength; locus++) {
                                    mask[locus]= MyRandom1.boolean();
                            }
                    }
                    else {
                            if (xoPoint1 > xoPoint2 ) {
                                    //reverse order and reverse bit flag;
                                    bit = false;
                                    temp =xoPoint2;
                                    xoPoint2=xoPoint1;
                                    xoPoint1=temp;
                            }
                            for (locus = 0; locus < cromLength; locus++)
                                    mask[locus] = bit;
                            for (locus = xoPoint1; locus < xoPoint2; locus++)
                                    mask[locus] = !bit;
                    }
            // Do the Crossover using the mask
        for (int locus = 0; locus < cromLength; locus++) {
                    if (mask[locus]) {
                      // swap
                    temp = one->getGene(locus);
                      one->setGene(locus, two->getGene(locus));
                    two->setGene(locus, temp);
                      }
        }
}
double MyRandom::dbl() {
  // Return a random double in [0,1).
  return genrand();
}
bool MyRandom::boolean() {
   // Return a random boolean (false or true).
  if (genrand() >= 0.5) return true;
  return false;
}
unsigned char MyRandom::integer(int n) {
   // Return a random integer (char) from 0 to n-1 inclusive.
   unsigned char i;
   i=(unsigned char)(genrand()*n);   // genrand() returns a double in [0,1]
   if (i==n) i=0; //On the chance the interval [0,1] really does include 1
   return i;
}
/*************************************************************/
/* Selection function: Standard proportional selection for   */
/* maximization problems incorporating elitist model - makes  */
/* sure that the best member survives.                */
/*************************************************************/
// also called the roulette wheel method
void Selection::select(Chromosome **population,
        Chromosome **nextPopulation, int populationSize){
   double p, sum = 0;
```

```
    int i;
    // find total fitness of the population
    for (i = 0; i < populationSize; i++) {
            sum += population[i]->getFitness();
    }
    // calculate relative fitness
    for (i = 0; i < populationSize; i++) {
            population[i]->rfitnessSet(population[i]->getFitness()/sum);
    }
    population[0]->cfitnessSet(population[0]->rfitnessGet());
    // calculate cumulative fitness
    for (i = 1; i < populationSize; i++) {
            population[i]->cfitnessSet(population[i-1]->cfitnessGet() + population[i]->rfitnessGet());
    }
    // finally select survivors using cumulative fitness.
    for (i = 0; i < populationSize; i++) {
            p = MyRandom1.dbl();
      if (p < population[0]->cfitnessGet())  {
        population[0]->copyChromosome(nextPopulation[i]);
        //System.err.println("replacing 0 with  " + i);
      }
                        else {
        for (int j = 0; j < populationSize; j++) {
          if (p >= population[j]->cfitnessGet()
                                                  && p < population[j+1]->cfitnessGet()) {
            population[j+1]->copyChromosome(nextPopulation[i]);
          }
        }
      }
    }
}
```

File FiltSpec.h:

```
#ifndef FILTSPEC_H
#define FILTSPEC_H
#include <math.h>
#define GC_DEBUG
#include "gc_cpp.h"
#include <complex.h>
class FilterSpec: public gc {
  private:
    void calcFilterZs(void);
    int numUniqueCoeffs;
    double freqStart;
  public:
    bool linearPhase;
    bool  skipTransitionBand;
    double *targetResp;
    int numFreqPoints;
     FilterSpec(double *targetResp, double *targetFreqs, double ws, int nep,int nc);
    double FiltSquareError(double w,double mag);
    double T;  // sample period
    double ws;   //sample freq.
    complex<double> **zsum;
    double *FilterZfreqs;
};
#endif
```

File FiltSpec.cpp:

```
#include <math.h>
#include <complex.h>
#define GC_DEBUG
#include "gc_cpp.h"
#include "FiltSpec.h"
#include "ArrayGen.h"
#include"Unit1.h"
FilterSpec::FilterSpec(double *tr, double *tf, double w, int nep,int nc) {
        targetResp = tr;
        FilterZfreqs = tf;
        ws = w;
        T= (2.0 * M_PI)/ws;
        numFreqPoints = nep;
        numUniqueCoeffs=nc;       // number of coeff's
        calcFilterZs();
        freqStart=0;
}
void FilterSpec::calcFilterZs(void)  {
    double w;
    int n,i,NumTotalCoeffs;
    complex<double> z,zScale;
    if (linearPhase && Form1->numBi == 0) NumTotalCoeffs = Form1->order;  // reverse of  ((order-1)/2) +1
    else NumTotalCoeffs=numUniqueCoeffs;
    ArrayGen *A = new (GC) ArrayGen;
    zsum = A->new2D(zsum,numFreqPoints,NumTotalCoeffs) ;
    for (i = 0; i < numFreqPoints; i++) {
                                w = FilterZfreqs[i] ;  // frpassBandStart + increment * (double)i;
                                z= exp(complex<double>(0,T*w));
                                if (linearPhase && Form1->numBi == 0 )  {// Linear Phase FIR
        for (n=0; n < NumTotalCoeffs; n++) {
         zsum[i][n] = pow(z,n) + pow(z,-n);
        }
        zScale = pow(z,-NumTotalCoeffs);
        zsum[i][n] *=  zScale;
                        }
      if (!linearPhase && Form1->numBi == 0 ) {  //FIR
        for (n=0; n < NumTotalCoeffs; n++) {
          zsum[i][n] =  pow(z,-n);
        }
      }
      if (Form1->numBi > 0 ) {
        int max = Form1->numBi;
        if (Form1->numAi > max) max =Form1->numAi;  //IIR
        for (n=0; n < max; n++) {
          zsum[i][n] =  pow(z,-n);
        }
      }

    }
}
```

File FitFunc.h:

```
//---------------------------------------------------------------------------
#ifndef FitFuncH
```

```
#define FitFuncH
//---------------------------------------------------------------------------
#define MINMAXdef 0
#define LMSdef 1
#endif
```

File FitFunc.cpp:

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "FitFunc.h"
#include "ArrayGen.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#include "Chromosome.h"
#include <math.h>
#define GC_DEBUG
#include "gc_cpp.h"
#include <Math.h>
#include <Complex.h>
#include "Unit1.h"
int juryMarCheck(double B[],int);
int juryMarChck2(double C[],int,int);

double Chromosome::evalChromosome() {
      double *Coeffs;
      Coeffs = toCSDPhenotype();
      return FreqError(Coeffs);
}

double **Chromosome::toCSDGenotype() {   // to ternary bit string
            int g;
              complex<float> z,cn,cmn,z2n,z2mn,H;
     double **CSDs;
     ArrayGen *AG = new (GC)ArrayGen;
     // Calc Coeffs
     CSDs=NULL;
     CSDs = AG->new2D(CSDs,numUniqueCoeffs,Ndigits);
     int *OneLoc = new (GC) int[NcsdOnes];
     int *OneVal = new (GC) int[NcsdOnes];
     for (int nc = 0; nc<numUniqueCoeffs; nc++) {
       for (int i = 0; i<NcsdOnes; i++) {
         g  =  gene[i + nc * NcsdOnes ];    // gene goes from 0 to (2*NcsdOnes -1) cardinality -1
         g -= Ndigits;
         if (g < 0 ) {
           OneVal[i]= -1;
           OneLoc[i] = abs(g) -1; // -16 to -1 become 15 to 0
         }
         else {
           OneVal[i]= 1;
           OneLoc[i] = g;
         }
       }
       for (int i = 0; i<Ndigits; i++) {
         CSDs[nc][i] = 0;
       }
       CSDs[nc][OneLoc[0]] = OneVal[0];
```

```
        for (int i=1;i< NcsdOnes;i++) {
          if (OneLoc[i] == 0)  {
            if (CSDs[nc][0] == 0 && CSDs[nc][1] == 0 )
              CSDs[nc][OneLoc[i]] = OneVal[i];
          }
          else {
            if (OneLoc[i] == Ndigits-1) {
              if (CSDs[nc][Ndigits-1] == 0 && CSDs[nc][Ndigits-2] == 0 )
                CSDs[nc][OneLoc[i]] = OneVal[i];
            }
            else {
              if (CSDs[nc][OneLoc[i]] == 0 &&
                CSDs[nc][OneLoc[i]-1] == 0 &&
                CSDs[nc][OneLoc[i]+1] == 0)
               CSDs[nc][OneLoc[i]] = OneVal[i];
            }
          }
        }
      }
      return CSDs;
}
double *Chromosome::toCSDPhenotype() {
      // Convert CSD's to Floating point value
      double *Coeffs = new (GC) double[numUniqueCoeffs];
      double **CSDs = toCSDGenotype();

      for (int nc = 0; nc<numUniqueCoeffs; nc++) {
        Coeffs[nc] = 0.0;
        for (int i = 0; i<Ndigits; i++) {
          Coeffs[nc] += ((double)CSDs[nc][i])/(pow(2.0,i));
        }
      }
      return Coeffs;
}


Freq *Chromosome::FreqResp(int numpoints) {
                numpoints = Form1->numFreqPoints; e
   FilterSpec *theFilterSpec2 = new (GC) FilterSpec(
              Form1->target, Form1->fresp, Form1->ws, numpoints, numUniqueCoeffs);
  return FreqResp(theFilterSpec2);
}

Freq *Chromosome::FreqResp(FilterSpec *theFilterSpec) {
  double w;;
  int i,n;
  complex<double> H;
  Freq *fr = new (GC) Freq[theFilterSpec->numFreqPoints];
  double *Coeffs  = toCSDPhenotype();
  double mag;
          for (i = 0; i < theFilterSpec->numFreqPoints; i++) {
      w = theFilterSpec->FilterZfreqs[i];
      fr[i].w=w;
      complex <double> sumA = complex<double>(0.0,0.0);
      complex <double> sum = complex<double>(0.0,0.0);
          for (n=0; n < Form1->numAi; n++) {
                                        sumA += theFilterSpec->zsum[i][n] * Coeffs[n];
                    }
      complex <double> sumB = complex<double>(0.0,0.0);
              if (Form1->numBi > 0 )   { // IIR
```

```
            for (n=0; n < Form1->numBi; n++) {
                                        sumB += theFilterSpec->zsum[i][n] * Coeffs[n+Form1->numAi];
                                }
              stable = juryMarCheck(Coeffs + Form1->numAi,Form1->numBi);
          }
          else  { // FIR
             sumB =  complex<double>(1.0,0.0);
          }
          sum = sumB;
          if ( abs(sum) == 0 )  {
               sum = complex<double>(.00000000001,0.0);
          }
          sum = sumA / sum;
          H=sum;
          fr[i].mag = mag = abs(H);
                        fr[i].phase = arg(H);
      }                 */
      return fr;
}
double Chromosome::FreqError(double *ff) {
      int i ;
      if (Form1->FitnessType == LMSdef) {
        Freq *fr = FreqResp(theFilterSpec);
        double error=0.0;
        for (i = 0; i < theFilterSpec->numFreqPoints; i++) {
          error +=  pow((theFilterSpec->targetResp[i] - fr[i].mag),2.0);
        }
        if (stable == UNSTABLE)
          error *= Form1->unstablePenalty; //5; 2;  // double error;
        return 1.0/error;
       }
       else {
        double Pbr,Sbg,PBdiff,SBdiff;
        Pbr=Sbg = 0 ;
        PBdiff = pow(10.0,Pbr/20.0);
        SBdiff = pow(10.0,Sbg/20.0);
        return 1.0 /(PBdiff + 3*SBdiff);
       }
}
int juryMarCheck(double B[],int n) {
   int i,minus1Flag;
   double D1=0;
   int N = n-1; // n is number of Coeffs, N is highest index (going 0 to n-1)
   for (i=0;i<=N;i++) {
     D1= D1 + B[i];
   }
   if (D1 <= 0 )
     return UNSTABLE;

    D1=0;
    minus1Flag=+1;
    for (i=N;i>=0;i--) {      // Add in reverse order, from B(N) to B(0)
      D1= D1 + minus1Flag*B[i]; // D(-1) = B(N) - B(N-1) + B(N-2) ...
      minus1Flag = minus1Flag * -1;
    }
    D1 = D1 * minus1Flag *-1;  // D1 * (-1) to the N   (minus flag == -1 to the N-1)
    if (D1 <=0) return UNSTABLE;

    if (n<3)
      return STABLE;
```

```
    return juryMarChck2(B,n,1);


}

int juryMarChck2(double c[],int n,int first) {
    int i;
    double d[100];
    int N = n-1; // n is number of Coeffs, N is highest index (going 0 to n-1)
    if (first)  {  // first call, B0> abs(BN)
      if ( c[0] <= fabs(c[N]) )
          return UNSTABLE;
    }
    else {      // not first call, abs(C0) > abs (CN-1) (N was already decremented in recursive call)
      if (fabs(c[0])<= fabs(c[N]))
          return UNSTABLE;
    }
    if (N==2)  // down to r0, r1, r2
      return STABLE;
    else {
      for (i=0; i<= N-1; i++) {
        d[i] = c[0]*c[i] - c[N-i] * c[N];
      }
      return juryMarChck2(d,n-1,0);
    }
}
```

File GA.h:

```
#ifndef GA_H
#define GA_H
#include "Chromosome.h"
#include "FiltSpec.h"
#include <system.hpp>

class GA:public gc {
            // Initial GA parameters
            private:
      char elitism;
      int xoverType;
      int populationSize,chromCard,chromLength,numGenerations;
      float desiredFitness;
      double mutationRate,crossoverRate;
      String chromDecode;
      FilterSpec *aFilterSpec;
      int  theBestGeneration; // generation where theBest first showed up

      Chromosome **population;
      Chromosome  **nextPopulation;

      //Selection        *theSelection;
      //Crossover        *theCrossover;
      int Xrange;
      int numUniqueCoeffs; // Number of filter coefficients
      int NcsdOnes;   // Number of of non-zeros allowed in CSD
      int Ndigits;   // Number of CSD digits
      double Yupper;
```

```
    double Ylower;
            //private double _count = 0.0;
    void genPlot();


        public:
      GA(char e,int xot,int ps,int chc,int chl,int ng, float df,
         double mr, double xor, FilterSpec *ff, String chd,
         int nc, int ncsdOnes, int ndigits);
      bool pause;
      bool quit;
      void PerformGA(Chromosome **GlobalBestPtr);
      Chromosome *theBest;
      int                       generationCount;    // generation counter
      double bestFitness;
      double PBR,SBG;


};

#endif
```

File GA.cpp:

```
#include "GA.h"
#include"Unit1.h"
#define GC_DEBUG
#include "gc_cpp.h"
#include<stdio.h>
#include<math.h>
          GA::GA(char e, int xot, int ps, int chc, int chl, int ng, float df,
                                  double mr, double xor,FilterSpec *ff, String chd,
      int nc, int ncsdOnes, int ndigits)  {
                    generationCount = 0;
    Xrange = 10;
    elitism = e;
                    Yupper = 0.1;
    Ylower = 0.0;
    xoverType = xot;
                    populationSize = ps;
                    chromCard = chc;
                    chromLength = chl;
                    numGenerations = ng;
    desiredFitness =df;
                    mutationRate = mr;
                    crossoverRate = xor;
                    aFilterSpec = ff;
                    chromDecode = chd;
    numUniqueCoeffs=nc;    // number of
    NcsdOnes=ncsdOnes;  // number of allowed non-zeros in csd
    Ndigits=ndigits;   // number of ternary digits
    quit = false;
                    populationSize = ps;
                    mutationRate = mr;
                    numGenerations = ng;
    // Set up and initialize the GA structures
                    theBest = new (GC) Chromosome(chromCard,chromLength,aFilterSpec,
                numUniqueCoeffs,Ndigits,NcsdOnes);
```

```
        population = new (GC)  Chromosome*[populationSize];
        nextPopulation = new (GC)  Chromosome*[populationSize];
        for (int j = 0; j < populationSize; j++) {
                        population[j] = new (GC) Chromosome(chromCard,
                    chromLength,aFilterSpec,numUniqueCoeffs,Ndigits,NcsdOnes);
                        population[j]->initializeChromosomeRandom();
                        nextPopulation[j] = new (GC) Chromosome(chromCard,
                    chromLength,aFilterSpec,numUniqueCoeffs,Ndigits,NcsdOnes);
                }

}
        void GA::PerformGA(Chromosome **GlobalBestPtr ) {
          bool done = false;
        pause=false;
        MyRandom MyRandom1;
        Selection *theSelection = new (GC) Selection();
        Crossover *theCrossover = new (GC) Crossover(chromLength, xoverType);

        *GlobalBestPtr = population[0];
        Form1->SaveCSDs (population[0],generationCount);


        if (!done) {
                        // getBest() Search the population for the individual with the highest fitness
                        int BestSoFarIndex = 0;     // index of the current best individual
                        double BestSoFar = population[BestSoFarIndex]->getFitness();
                        double fitness;
        Form1->sp_XYPlot1->BufferedDisplay = true;
        Form1->sp_XYLine1->Clear();
                        for (int i = 1; i < populationSize; i++) {
                                        if ((fitness = population[i]->getFitness()) > BestSoFar ) {
                                        BestSoFarIndex = i;
                                        BestSoFar = fitness;
                        }
                        }
                        // once the best member in the population is found, copy the genes
                        population[BestSoFarIndex]->copyChromosome(theBest);
        double currentBestFitness = 0;
        Form1->SaveCSDs (theBest,generationCount);
                        while ((quit == false) && ((generationCount <= numGenerations
              || numGenerations == 0 ))
              && ( desiredFitness > currentBestFitness )) {

                                        generationCount++;
                                        theSelection->select(population, nextPopulation, populationSize);
                                        Chromosome **temp = population;
                                        population = nextPopulation;
                                        nextPopulation = temp;
                                        // crossover();
                                        int one = -1;
                                        int first  = 0; // count of the number of members chosen
                                        for (int i = 0; i < populationSize; ++i) {
                                                        if (MyRandom1.dbl() < crossoverRate) {
                                                                        ++first;
                                                                        if (first % 2 == 0) {
                                                                        theCrossover->xOver(population[one], population[i]);
                                                                        } else one = i;
                                                        }
                                        }
                                        if (elitism == 1 || (elitism==2 && generationCount != 2000000  )) { // && generationCount >30)))
```

```
{
                        // keep the best by replacing one individual
                        theBest->copyChromosome(population[0]);
                }
                        for (int i = 0; i < populationSize; i++) {
                                for (int j = 0; j < chromLength; j++) {
                                if (MyRandom1.dbl() < mutationRate) {
                                population[i]->mutateGene(j);
                                }
                        }
                        }
                        // update The Best
                        currentBestFitness = population[0]->getFitness();
                        double next = -1;
                        int currentBest = 0; // index of the current best individual

                        for (int j = 1; j < populationSize; j++) {
                                if ((next = population[j]->getFitness()) > currentBestFitness ) {
                                currentBest = j;
                                        currentBestFitness = next;
                                }

                        }
                        genPlot();
        *GlobalBestPtr = theBest;
        if (currentBestFitness > theBest->getFitness() || generationCount == 2000000 ) { //* 0.1 ) {
                        population[currentBest]->copyChromosome(theBest);
                        theBestGeneration = generationCount;
            if (generationCount > 3) //100)
                Form1->SaveCSDs (theBest,generationCount);
                        }
                    // done so print findings
        Form1->Update();
        Application->ProcessMessages();
            }
          }
}
        void GA::genPlot() {
        bestFitness=0.0;    // best population fitness
        double sum=0.0;         // total population fitness
        double fitness;
                for (int i = 0; i < populationSize; i++) {
                fitness = population[i]->getFitness();
                        if (bestFitness < fitness) bestFitness = fitness;
                sum += fitness;
                }
                bestFitness = theBest->getFitness();
    PBR=SBG=0; //theBest->getFreqStats(&PBR,&SBG);
    Form1->Label1->Caption = bestFitness;
    Form1->Label7->Caption = PBR;
    Form1->Label8->Caption = SBG;
    Form1->Label2->Caption = generationCount;
    Form1->Update();
    Form1->sp_XYLine1->AddXY(generationCount, bestFitness);
    Form1->sp_XYPlot1->Paint();
}
```

File ArbitraryForm.h:

```
//---------------------------------------------------------------------------
#ifndef ArbitraryFormH
#define ArbitraryFormH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ComCtrls.hpp>
#include <ExtCtrls.hpp>
#include <ToolWin.hpp>
#include <ImgList.hpp>
//---------------------------------------------------------------------------
class TArbitrary : public TForm
{
__published:            // IDE-managed Components
   TToolBar *ToolBar1;
   TToolButton *ToolButton1;
   TToolButton *ToolButton2;
   TImageList *ImageList1;
   TToolButton *ToolButton3;
   TImage *Image1;
   void __fastcall Panel1Click(TObject *Sender);
private:    // User declarations
public:                 // User declarations
   __fastcall TArbitrary(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TArbitrary *Arbitrary;
#endif
```

File ArbitraryForm.cpp:

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "ArbitraryForm.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TArbitrary *Arbitrary;
//---------------------------------------------------------------------------
__fastcall TArbitrary::TArbitrary(TComponent* Owner)
   : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TArbitrary::Panel1Click(TObject *Sender)
{
     int X=2,Y=6;
     Image1->Canvas->MoveTo(0, 0);
          Image1->Canvas->LineTo(X, Y);

}
```

Appendix B Definitions

Allele

Each gene occupies a specific character location within the chromosome string. Each gene position may take a character value called an allele.

Canonic Signed Digit (CSD) Number

Also known as **Canonical Signed Digit;** a number composed of 0,1 and -1 digits whose format follows the canonic constraint and has the fewest non-zero digits of any signed digit representation

Chromosome

A data structure consisting of a character string of coded task parameters.

Converged

A gene is said to have converged when 95% of the chromosomes in the population all contain the same allele for that gene. A population is said to have converged when all genes have converged.

It is commonly used to mean that the GA has slowed to a point that it doesn't seem to be finding new, better solutions.

Crossover

A reproduction operator which forms a new chromosome by combining parts of each of two parent chromosomes.

Deception

The condition where mating leads to reduced overall population fitness, rather than increased fitness. Proposed by Goldberg[gol89a] as a reason for the failure of gas on many tasks.

Elitism

A mechanism which is used to ensure that the chromosome of the most highly fit member of the population is passed on to the next generation without being altered by genetic operators. Using elitism ensures that the maximum fitness of the population can never reduce from one generation to the next. Elitism usually brings about a more rapid convergence of the population.

Epistasis

The interaction between different genes in a chromosome. It is the extent to which the contribution to fitness of one gene depends on the values of other genes.

Exploitation

The process of using information gathered from previously visited points in the search space to determine which places might be profitable to visit next.

Exploration

The process of visiting entirely new regions of a search.

Fitness

A value assigned to an individual which indicates how well the individual solves the task at hand. A fitness function is used to map a chromosome to a fitness value.

Gene

A position on a chromosome which usually holds the encoded value of a single parameter.

Generation

An iteration the creation of a new population by means of reproduction operators.

Genetic drift

Gene value changes resulting from chance rather than selection in a population over many generations.

Individual

A single member of a population which contains a chromosome representing a potential solution to the problem under consideration.

Mutation

A reproduction operator which forms a new chromosome by making random alterations to the values of genes. It usually occurs with low probability.

Non-Recursive filter

A filter whose output depends only on input values. It is always stable.

Offspring

An individual generated by any process of reproduction.

Optimization

The process of iteratively improving the quality of a solution to a problem as determined by a specified objective function.

Parent

An individual which takes part in reproduction to generate one or more other individuals, known as offspring.

Population

A group of individuals which interact together by mating to produce offspring.

Recursive filter

A filter whose output depends on input values and previous output values. It is not guaranteed to be stable.

Reproduction

The creation of a new individual from two parents.

Schema

A pattern of gene values in a chromosome, which may include

`don't care' states.

Schema theorem

The fundamental theorem of genetic algorithms. It says that a GA gives exponentially increasing reproductive trials to above average schemata. The rate of schema processing in the population is very high, leading to a phenomenon known as implicit parallelism. This gives a GA with a population of size N an implicit processing factor of N^3.

Selection

The process by which some individuals in a population are chosen for reproduction. It is biased in favor of individuals with higher fitness.

Vita Auctoris

Tom Williams obtained a Bachelor of Computer Science (Honours) degree from the University of Windsor in 1994 and a Master of Science, Computer Science degree in 1999. He is currently employed as an Information Technology Administrator at St. Clair College of Applied Arts and Technology. He is a candidate for a Doctor of Philosophy degree at the University of Windsor and hopes to graduate in 2006.