THE APPLICATION OF GENETIC ALGORITHMS TO THE ADAPTATION OF IIR FILTERS

by

Qiang Ma, BEng

A doctoral thesis
submitted in partial fulfilment of the requirements for
the award of Doctor of Philosophy of
Loughborough University of Technology

November 1995

Supervisor: Professor Colin F. N. Cowan
Department of Electronic and Electrical Engineering

Acknowledgements

I would like to express my gratitude to Professor Colin F. N. Cowan for his guidance, support and inspiration throughout the course of the research and preparation for this thesis.

I would also like to thank Dr. Christopher Callender and many other colleagues within the Communications and Signal Processing Group for their help in the early stage of the research work.

I must thank Dr. Matthew Levin for reading through the first draft of this thesis.

Finally, I would like to thank my wife Li Kuan for encouraging me to conduct my research studies.

Abstract

The adaptation of an IIR filter is a very difficult problem due to its non-quadratic performance surface and potential instability. Conventional adaptive IIR algorithms suffer from potential instability problems and a high cost for stability monitoring. Therefore, there is much interest in adaptive IIR filters based on alternative algorithms. Genetic algorithms are a family of search algorithms based on natural selection and genetics. They have been successfully used in many different areas. Genetic algorithms applied to the adaptation of IIR filtering problems are studied in this thesis, and show that the genetic algorithm approach has a number of advantages over conventional gradient algorithms, particularly, for the adaptation of high order adaptive IIR filters, IIR filters with poles close to the unit circle and IIR filters with multi-modal error surfaces. The conventional gradient algorithms have difficulty solving these problems. Coefficient results are presented for various orders of IIR filters in this thesis. In the computer simulations presented in this thesis, the direct, cascade, parallel and lattice form IIR filter structures have been used and compared. The lattice form IIR filter structure shows its superiority over the cascade and parallel form IIR filter structures in terms of its mean square error convergence performance.

Contents

Symbols and Acronyms

a(n)	Feedforward coefficient of direct form IIR filter
AFM	Adaptive Filter Mode
A(z)	The z transform of the input x(n) of adaptive IIR filter
b(n)	Feedbackward coefficient of direct form IIR filter
B(z)	The z transform of the output y(n) of adaptive IIR filter
CHC	Cross generational elist selection, Heterogeneous recombination, and Cataclysmic mutation
dB	Decibel
d(n)	Desired signal
f_i	Fitness of individual
f(H)	Average fitness of schema H
$\overline{f}$	Average fitness of population
FIR	Finite Impulse Response
FTF	Fast Transversal Filter
GA	Genetic Algorithm
Genitor	*Gen*itic *i*mplemen*tor*
h_d	Hamming distance of parent strings
$\mathbf{h}^{\mathbf{t}}(n)$	Output coefficient vector of normalized lattice filter
H	A schema
H(z)	Transfer function

HARF	Hyperstable Adaptive Recursive Filter
$\mathbf{I}$	Identity matrix
IIR	Infinite Impulse Response
k_i	Reflection coefficient of lattice form IIR filter
LMS	Least Mean Square
L	String length
MSE	Mean Square Error
M(H, t)	Samples of a schema H contained within the population
N	Population size
NL	Normalized Lattice
o(H)	Order of schema H
P(t)	Population
p_c	Probability of crossover
p_d	Probability of destruction
p_m	Probability of mutation
p_s	Probability of survival
$\hat{\mathbf{Q}}$	Q matrix in QR method
$\hat{\mathbf{Q}}_{\mathbf{k}}$	Givens rotation matrix
$\mathbf{R}$	R matrix in QR method
RLS	Recursive Least Squares
SGA	Simple Genetic Algorithm
SIM	System Identification Mode

SPR	Strict Positive Real
$\mathbf{U_k}$	Givens rotation matrix
v_i	Output coefficient of lattice form IIR filter
v(n)	Noise signal
x(n)	Input signal to an adaptive filter
y(n)	Output signal of an adaptive filter
z^{-1}	Unit delay
α(n)	Prior error
$\beta(x)$	Linear function for selection
δ(H)	Defining length of schema H
$\theta(n)$	Coefficient of an adaptive filter
κ	A small constant for QR algorithms
λ	Forgeting factor
μ	Step size constant of LMS algorithm
ξ	Cost function
σ	Standard deviation of population fitness
τ	Step size constant
$\phi(n)$	Reflection coefficient of normalized lattice filter
$\varphi(n)$	Input vector
ψ(n)	Gradient vector

Chapter 1

Introduction

1.1 Introduction

The adaptation of an IIR filter is a very difficult problem due to its non-quadratic performance surface and instability. Conventional gradient adaptive IIR filter algorithms face either potential instability problems or the high cost of stability monitoring. In addition, when an IIR filter's pole is close to the unit circle, most of these algorithms have difficulty during the adaptation. The direct form adaptive IIR filter has poor numerical precision properties, and with high order IIR filters, it is difficult to monitor or limit the denominator coefficients to avoid instability. All of this makes alternative non-gradient algorithms and filter structures more appealing to researchers in adaptive signal processing.

Genetic Algorithms (GAs) are a family of search algorithms developed in the 60's and 70's. The basic idea for genetic algorithms originated from natural selection and genetics. They incorporate the genetic operator into the computer programming to solve the biological or non-biological problem. They have been very successful in solving many problems in biology, computer science, and engineering. These algorithms are general and robust. Our study is concerned with applying these algorithms to the adaptation of IIR filters, which has generated some encouraging results [1], [2].

The alternative IIR filter structures are cascade, parallel and lattice. The lattice structures in particular have received a great deal of attention due to their superior finite precision properties compared with the direct form structure. In addition, stability monitoring of the lattice IIR filter is extremely simple and requires almost no computation. We have paid attention to the lattice form structure and proven its superiority. This chapter begins with a review of other research applying genetic algorithms to the adaptation of IIR filters, a discussion of the motivation for the study then follows, and finally, the organization of the thesis is described.

1.2 Related Research

Early studies using an adaptive genetic algorithm to determine the optimum filter parameters of an adaptive system were carried out in [3] and [4]. [3] applied a genetic algorithm to the very simple unimodal and bimodal adaptive IIR filters, with an order one or two transfer function. [4] has applied genetic algorithms to determine the optimal control parameters.

Recently R. Nambiar and P. Mars [5] - [9] have applied genetic algorithms to the adaptation of IIR filter, in particular for high order IIR filters. These studies extended Etter's study [3], and showed some encouraging results. However, the MSE performance still needs to be improved. These studies experimented with cascade, parallel and lattice adaptive IIR filter structures - the cascade and parallel structure transfer functions are not general. For example, the high order filters were adapted as a bank of first and second order filters, with the numerator of the second order filter set to 1. In our study, we have tried to improve the MSE performance and use a general model for cascade and parallel filters.

1.3 Motivation

As has been shown, the general performance of genetic algorithms applied to the adaptation of IIR filters needs to be improved. We also need to perform more experiments on various adaptive IIR filter structures.

The first objective of this research is to find a genetic algorithm which can provide an improved MSE performance to the adaptation of IIR filters, especially when the poles of the IIR filter are close to the unit circle. The previous studies ([3] - [9]) concentrated on using the Simple Genetic Algorithm, which has many disadvantages in many applications.

The second objective is to find out which of the alternative filter structure gives the best results when applying genetic algorithms to the adaptation of IIR filters. The disadvantage of the direct form structure has been reported [10], the alternative structures being cascade, parallel and lattice.

Finally, we consider the coefficients of adaptive IIR filters, in terms of quantitative analysis, which most previous research has missed.

1.4 Outline of the Thesis

The remainder of the thesis is organized as follows:

Chapter 2. Overview of Genetic Algorithms

The concepts, mathematical foundations and operators of genetic algorithms are described in this chapter. It lays out the background of genetic algorithms for the whole thesis.

Chapter 3. Adaptive Infinite Impulse Response Filters

The two fundamental approaches to adaptive IIR filter - equation error and output error formulation - are introduced. We also outline gradient algorithms for the adaptive IIR filter.

Chapter 4. Simulations of Two Gradient IIR Filtering Algorithms

Two recent gradient approaches to the IIR filtering problem are reiterated. Computer simulation of these two schemes are conducted, the results of which are compared with our genetic algorithm results in later chapters.

Chapter 5. Applying Simple Genetic Algorithm to IIR Filters

Computer simulation results of applying the Simple Genetic Algorithm to the adaptation of IIR filtering problems are obtained. Direct, cascade and parallel IIR filter structures are used in the experiments.

Chapter 6. Applying Genitor to IIR Filters

In this chapter, computer simulation results of applying Genitor to the adaptation of IIR filtering problems are given. We experiment with a variety of IIR filtering problems - multi-modal, poles close to the unit circle, and high order IIR filter problems. We also experiment with different IIR filter structures. Several discussions based on the results in this chapter and the previous chapter are presented.

Chapter 7. Conclusions

The final chapter provides a conclusion to the results presented here - both the successes and limitations of applying genetic algorithms to IIR filtering problems. Finally, we propose a variety of topics for further investigation.

Chapter 2

Overview of Genetic Algorithms

Interest in Genetic Algorithms is expanding rapidly. Researchers have found that they can apply Genetic Algorithms to many different areas, such as biology, computer science, engineering, economics etc.. In this chapter, the background of Genetic Algorithms is presented.

2.1 Introduction

In *On the Origin of Species by Means of Natural Selection* [11], Darwin argued that all existing organisms are the modified descendants of one or a few simple ancestors that arose on Earth in the distant past - as we now know, over 3000 million years ago. He also argued that the main force driving this evolutionary change was natural selection [12]. Genetic Algorithms (GAs) are a group of search algorithms based on the mechanisms of such natural selection and genetics [13]. The basic idea and the fundamental theory were developed by Holland and his student in the 60's and early 70's at Michigan University [14], [15]. These algorithms encode a potential solution to a specific problem on a simple chromosome-like (or gene-like) data structure, and apply selection, recombination, mutation and perhaps other genetic operations to these structures, so as to preserve critical information [16].

An implementation of a genetic algorithm begins with a population of (typically random) chromosomes. These chromosomes can be represented by binary, arabic or alphabetical data. One then evaluates these structures and allocates reproductive opportunities in such a way that those chromosomes which represent a better solution to the target problem are given more chances to 'reproduce' than those chromosomes which are poorer solutions. The 'goodness' or 'fitness' of a solution is typically defined with respect to the current population.

In a classical GA, the members of the population are represented as fixed-length strings of binary digits, as shown in Figure 2.1. The length of the string L and the population size N are completely dependent on the problem. Either may range from a few tens to many thousands [17]. In genetic terms, we say that each binary string represents a chromosome, a gene or a *genotype*, each bit position is called the *locus*, and the locus value is named an *allele*. The genotype is decoded to form the *phenotype* of the individual. According to the problem we intend to solve, for example function optimization, we can convert the phenotype to the fitness (or function) value. The fitness values in Figure 2.1 (the second parentheses) are obtained through the following fitness evaluation procedure:

$$fitness = phenotype/2. \tag{2.1}$$

Sometimes we want to find the minimum, so the lowest fitness value is required; if we solve the maximum problem, the highest fitness value is required - it is problem dependent. In Chapters 5 and 6, we optimize the mean square error (MSE), so finding the lowest fitness value is the required optimization.

In a broader usage of the term, a genetic algorithm is any population-based model that uses selection and recombination operators to generate new sample points in a search space. Fitness proportionate selection, for example, which embodies the concept of 'survival of the fittest', is used to select parents from the population. Genetic recombination (crossover) is applied to pairs of parents to create offspring, which will be mutated through the mutation operation and then inserted into a new population, forming the next generation of individuals. The whole procedure is shown in Figure 2.2. We will discuss these basic genetic operators in section 2.3. In the next section we introduce the concept of schema and schema theory, which will give the mathematical background for genetic algorithms. In section 2.4, some advanced operators are introduced, and a practical example, which shows how the genetic algorithm works, is given in section 2.5.

0 0 0 1 0 1 1 1	(23)	(11.5)	1 0 1 1 1 1 0 0	(188)	(94)
1 1 0 0 0 1 0 1	(197)	(98.5)	0 1 0 0 0 1 0 0	(68)	(34)
1 0 0 1 1 0 0 0	(152)	(76)	0 1 1 1 0 1 1 1	(119)	(59.5)
0 0 0 1 0 1 1 0	(22)	(11)	1 0 1 0 1 0 1 0	(170)	(85)

Figure 2.1: A population of eight binary strings, each with a length of eight bits (phenotypes shown in the first parentheses, fitnesses in the second parentheses).

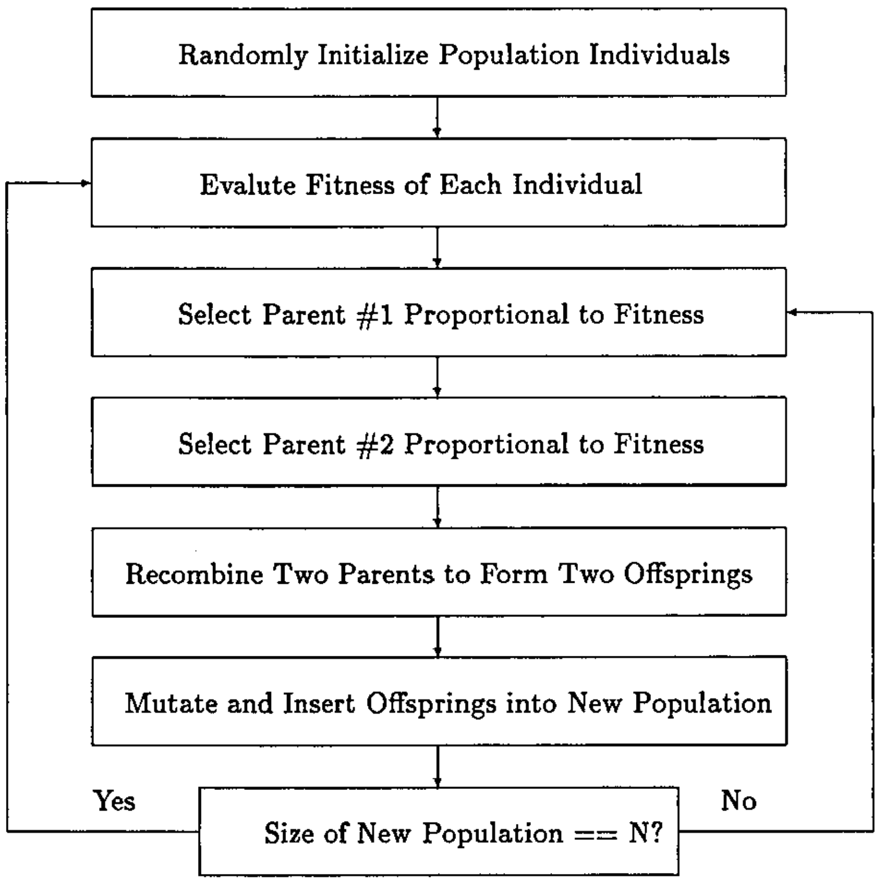
Randomly Initialize Population Individuals
Evalute Fitness of Each Individual
Select Parent #1 Proportional to Fitness
Select Parent #2 Proportional to Fitness
Recombine Two Parents to Form Two Offsprings
Mutate and Insert Offsprings into New Population
Yes
Size of New Population == N?
No

Figure 2.2: The simple genetic algorithm procedure.

2.2 Schema and Schema Theorem

The concept of schema and the schema theorem were developed by Holland in his landmark book *Adaptation In Natural and Artificial Systems* [14]. The schema theorem laid a mathematical foundation for genetic algorithms. It provides a lower bound on the change in the sampling rate for a single schema (hyperplane) from one generation to the next.

2.2.1 Schema

A schema is a similarity template describing a subset of strings with similarities at certain string positions [13]. These similarities can help guide a genetic search. Before presenting further discussion, we introduce the $*$ or *don't care* symbol for the string representation, which can either be 0 or 1, so a genetic string can be represented by the set $\{0, 1, *\}$. A string with length L represented by the set {0, 1, $*$} has 3^L different combinations, we say this string has 3^L schemata. In Figure 2.3, string 1 and string 2 have the same bit value on the second and the fourth bit position, so they can be written into the schema representation of string 3.

Given a schema $H = \{1\ 0 * * * 1\}$, we introduce two important properties of schemata:

• *defining length*, denoted by $\delta(H)$ is the distance between the first and the last specific real value (0 or 1) position, here $\delta(H)$ = 6 - 1 = 5;

• *order*, denoted by *o(H)*, is the number of fixed real value (0 or 1) positions. In the above example, *o(H)* = 3.

string 1	1 0 0 0 0
string 2	0 0 1 0 1
string 3	$* 0 * 0 *$

Figure 2.3: Illustration of schemata.

2.2.2 Genetic Search Space and Schema as Genetic Search Space Partition

Compared with traditional gradient search algorithms, such as least mean square (LMS) and recursive least squares (RLS), genetic algorithms are global search methods, because they are population based search methods. Every member of the population can be a search point, and every generation has N (population size) search points. These methods are weak, but robust and general [16]. Normally, the optimization problems are encoded into binary or alphabetical representations. If the problem is encoded into binary strings with length L, the search space is 2^L and forms an L-dimensional hypercube. The genetic algorithm samples are the corners of this hypercube (Figure 2.4). For a length 15 binary string encoding, there are $2^{15} = 32{,}768$ possible solutions in the search space, for a length 32 binary string encoding, there are $2^{32} = 2{,}147{,}483{,}648$ possible solutions in the search space. The target for the genetic algorithm is to find out which one is the best solution in the genetic search space.

Genetic algorithms can result in complex and robust searches by implicitly sampling hyperplane (schema) partitions of the search space [16]. Using the example from [16], we explain how the search space can be treated as a hypercube, and how it is partitioned. In Figure 2.4, the upper and lower cubes represent 3-dimension and 4-dimension spaces. The lower cube is called a hypercube, and is constructed by a cube 'hanging' inside the other cube. The corners are represented by strings or the search points in the search space. The numbering scheme for the lower hypercube corners is produced by adding a 0 to the upper cube corner labels as a prefix to form the corner labels of the outer cube, and a 1 to the upper cube corner labels as a prefix to form the corner labels of the inner cube. The front plane of the upper cube can be represented by the schema $\{0**\}$, the front plane of the inner and outer bottom cube can be represented by the schemata $\{00**\}$ and $\{10**\}$. In the upper cube, 8 points, 12 lines and 6 planes partition the 3-D space, making 3^3 =27 schemata in total including the hypercube (all $*$ in schema) itself.

2.2.3 Schema Theorem

Suppose at a given generation t there are M samples of a particular schema H contained within the population *P(t)*, M can be denoted by *M(H, t)*. We first consider reproduction, during which the intermediate samples are created and put in the mating pool. The selections are made according their fitness values, with the probability of a string being selected as

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}. \tag{2.2}$$

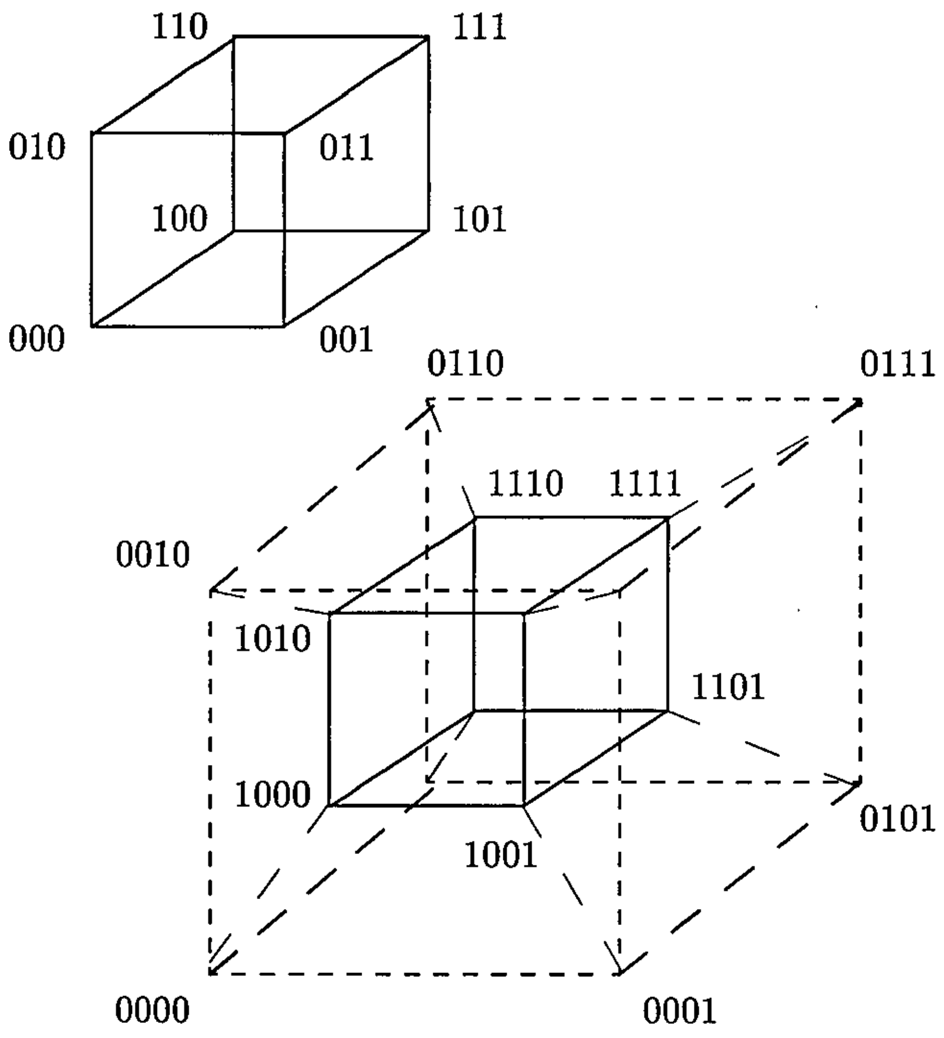

Figure 2.4: A 3-D and a 4-D hypercube. The corners of the 4-D hypercube are numbered in the same way as in the upper 3-D hypercube, except the addition of a 0 to the outer cube corner labels as a prefix, and the addition of a 1 to the inner cube corner labels as a prefix.

Therefore, after selecting samples from a population *P(t)* with size *N*, we expect to have *M(H, intermediate)* representatives of the schema *H* in the mating pool. This can be calculated by

$$M(H, intermediate) = \frac{NM(H,t)f(H)}{\sum_{j=1}^{N} f_j}, \tag{2.3}$$

where *f(H)* is the average fitness of schema *H*. This equation can also be written as

$$M(H, intermediate) = M(H,t)\frac{f(H)}{\overline{f}}, \tag{2.4}$$

with

$$\overline{f} = \frac{\sum_{j=1}^{N} f_j}{N}, \tag{2.5}$$

which is the population average fitness.

Next crossover is included. After applying crossover to the individuals in the mating pool, some individuals will survive, some individuals will die and some new individuals will be created. For a schema *H*, the bigger the defining length is, the easier it is disrupted. The probability for schema *H* being destroyed is

$$p_d = \frac{\delta(H)}{L-1}, \tag{2.6}$$

and the survival probability is

$$p_s = 1 - p_d. \tag{2.7}$$

Assuming the crossover probability is p_c, the survival probability of schema *H* can be given as

$$p_s \geq 1 - p_c\frac{\delta(H)}{L-1}. \tag{2.8}$$

Therefore, by combining reproduction and recombination, the expected representatives of the schema H in the next generation is

$$M(H,t+1) \geq M(H,t)\frac{f(H)}{\bar{f}}[1-p_c\frac{\delta(H)}{L-1}]. \tag{2.9}$$

Finally, the mutation operator is considered. Mutation could happen to every bit position in a schema. Now suppose the mutation probability is p_m, then for every single bit, the probability for it to survive over the mutation is 1 - p_m. These *o(H)* bit positions have a value of 0 or 1 in the schema *H*. If the schema survives to the next generation, the *o(H)* bits have to survive, so applying mutation to the schema *H*, the probability for this schema to survive is $(1-p_m)^{o(H)}$. Thus, the equation (2.9) can be rewritten as

$$M(H,t+1) \geq M(H,t)\frac{f(H)}{\bar{f}}[1-p_c\frac{\delta(H)}{L-1}](1-p_m)^{o(H)}. \tag{2.10}$$

This is called the schema theorem. Normally, the mutation rate p_m is very small ($p_m \ll 1$), so the schema theorem (2.10) can be further simplified as

$$M(H,t+1) \geq M(H,t)\frac{f(H)}{\bar{f}}[1-p_c\frac{\delta(H)}{L-1}][1-o(H)p_m]. \tag{2.11}$$

From this theorem, we draw the following conclusion: low defining length, low order schemata are given exponentially increasing or decreasing numbers of samples, depending on a schema's average fitness [13], and are given the special name *building blocks*. Building blocks play a important role in genetic algorithm studies, see [13] for details.

2.3 Genetic Operators

The basic genetic algorithm cycle is completed through four phases: evaluation, selection, recombination and mutation [18]. Genetic operators play a very important role in this cycle. In this section, three basic operators - selection (reproduction), crossover (recombination) and mutation are introduced.

2.3.1 Reproduction - Selection

Genetic algorithms start with a random initial population, of size N and string length L, with each population individual evaluated for its fitness value. Selection is an operator which uses the fitness value to select the fittest string. For example, for a maximum optimization problem, apply the selection operator to the population in Figure 2.1, then the second and the third strings will have the highest probability of being selected, because they have the highest fitness values (94 and 98.5). The selected individuals will be recombined and mutated, surviving to the next generation. The non-selected individuals will die out and not be included in the next generation. So under selection alone, individuals can only do one of three things: they may be born, they may live or they may die [19]. The selection phase is composed of two parts: 1) determination of the individuals' expected values; and 2) conversion of the expected values to discrete numbers of offspring [18]. The absolute difference between an individual's actual sampling probability and its expected value is defined as selection *bias* [1] There are many different selection schemes: proportionate, ranking, tournament and steady state selection. We now give some basic details about these selection schemes.

Proportionate selection describes a group of selection schemes that choose individuals for birth according to their fitness values. In these schemes, the probability of selection p of an individual from the ith class in the tth generation is calculated as

$$p_{i,t} = \frac{f_i}{\sum_{j=1}^{N} f_j}, \tag{2.12}$$

where N is the population size. Various methods have been suggested for sampling this probability distribution, including roulette wheel selection [13], [15], stochastic remainder and stochastic universal selection [18], etc.. Here we give a brief introduction to the roulette wheel selection and the stochastic remainder selection.

The roulette wheel selection is also called the Monte Carlo selection [15]. It uses the gambling roulette wheel to allocate offspring strings with slots sized according to their fitness. We list several strings in Table 2.1 together with their fitness and the percentage of individual fitness in the fitness sum. Figure 2.5 is a roulette wheel with the related string slots. We spin this wheel four times, four offspring strings are produced, and these four strings form the mating pool. So

[1]This bias is used by James Baker to analyse election efficiency in [18]. It has different definition to the one we are going to use in Chapter 6

Table 2.1: The example for roulette wheel selection.

String No.	String	Fitness	% of Total
1	0 0 0 1 0 1 1 1	12.5	6.3
2	1 1 0 0 0 1 0 1	98.5	49.7
3	1 0 0 1 1 0 0 0	76	38.4
4	0 0 0 1 0 1 1 0	11	5.6

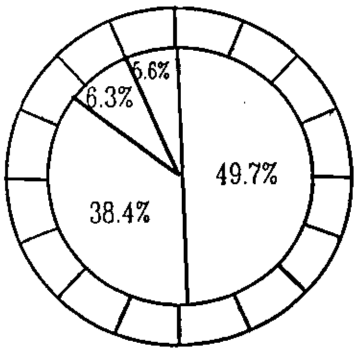

Figure 2.5: The roulette wheel reproduction.

four spins perform the whole selection procedure, and strings 2 and 3 have the highest probability of being selected.

We consider the example in [13], which maximizes the function $f(x) = x^2$ over the interval [0-31], where x is represented by five-bit strings in the GA, given in Table 2.2, showing the stochastic remainder selection. First, we calculate the mean of the fitness, which is equal to (4+576+1+361)/4 = 235.5, then we calculate the expected number of copies the string will produce in the intermediate population. For string 1, Expect = integer part of (4/235.5) = 0, its remainder = (4/235.5) - 0 = 0.017; for string 2, Expect = integer part of (576/235.5) = 2, its remainder = (576/235.5) - 2 = 0.45, etc.. Now we have produced three string at the current stage, two copies of string 2 and one copy of string 4. According to the remainder, we carry out another round of selection from the four strings: if *drand48()*[2] $\leq$ *Remainder* (roulette wheel), we choose this string, and in this case, the chosen string will be string 4. We now construct an intermediate population { 1 1 0 0 0, 1 1 0 0 0, 1 0 0 1 1, 1 0 0 1 1 }, and use a random set of selecting strings from the intermediate population to form the mating pool. The selection procedure used is the stochastic remainder selection. If after one string has been selected from the intermediate population to the mating pool, this string is dismissed from the intermediate population, we call the whole procedure stochastic remainder selection with replacement. We will use this selection scheme in the next chapter.

Ranking selection is a selection scheme in which each individual receives an

[2]drand48() is a random number generation function, generate random number between (0, 1).

Table 2.2: The example for stochastic remainder selection.

String No.	String	x	Fitness x^2	Expect	Remainder
1	0 0 0 1 0	2	4	0	0.017
2	1 1 0 0 0	24	576	2	0.45
3	0 0 0 0 1	1	1	0	0.004
4	1 0 0 1 1	19	361	1	0.53

expected number of offspring based on the rank of its performance and not on the magnitude [20]. It can control the rate of convergence, because this selection scheme can control the range of trials allocated to any single individual, so no individual receives many offspring. We will discuss this scheme further in Chapter 6. For more detailed studies, see [20].

Tournament selection is one of the most commonly used selection schemes in genetic algorithms. A group of individuals are randomly chosed from a population; select the best individual from this group for recombination, mutate, and repeat this procedure as many times as desired (usually until the mating pool is filled). Tournaments are often held between pairs (tournament size = 2) or more than two individuals [19].

Steady state selection is another selection scheme which was used in Genitor [21], [22]. We will introduce this scheme in chapter 6.

2.3.2 Recombination - Crossover

Once parents have been selected from the population, their genetic material is combined to form the offspring. Picking a pair of strings among the selected individuals, and crossing them into one another with a probability p_c for exchanging genetic information, produces a new pair of strings. This procedure repeats a certain number of times until the full population is filled. Crossovers can be one of *1-point*, *2-point*, ..., or uniform crossover [23], [24], which exchange *1, 2, ..., or L/2* fragments of the strings. The crossover site can be chosen between *1* and *L* (string length), but normally it is randomly chosen. In Figure 2.1, if *1-point* crossover occurs between the fourth and the fifth binary position, swapping the fragments of the first and the second strings produces two offspring, the first string becoming $\{0\ 0\ 0\ 1\ 1\ 1\ 0\ 0\}$. It is this improvement in performance (fitness 14), which is the purpose of recombination. In Figure 2.6, one-point crossover occurs at locus 5; in figure 2.7, two-point crossover occurs between locus 3 and locus 7. Figure 2.8 shows the uniform crossover. We will apply these recombination procedures in Chapter 5.

2.3.3 Mutation

When applying crossover on the selected strings, some alleles on a certain locus may never be changed through the recombination operation, meaning that a certain part of the search space will not be searched. To overcome this problem, another operator - mutation - is introduced.

Mutation is the means by which fundamentally new traits are introduced into the population [17]. Mutation occurs randomly and very rarely both in natural and artificial genetic systems, but when it does, it may cause chromosomes to take on new values which have never occurred in the population before [25]. When the mutation does happen to a individual, one bit of the chromosome is chosen and set to its complementary value. This provides greater ability to ensure that every part of the search space is visited. If mutation happens to the string

$$0\ 0\ 0\ 1\ 0\ 1\ 0\ 1,$$

on locus 8, the allele on the locus will be changed to its complementary value:

$$0\ 0\ 0\ 1\ 0\ 1\ 0\ 0.$$

Typically in GAs, only one in many thousand of genotypes are affected by mutation.

The processes of evaluation for fitness, reproduction, recombination and mutation form one generation cycle in the execution of a genetic algorithm.

2.4 Some Advanced Operators

Selection, crossover and mutation are the basic operators for genetic algorithms. There are many other operators adopted from natural genetics [12] in genetic algorithms. These operators include low-level operators such as dominance, inversion, intrachromosomal duplication, deletion, translocation, and segregation. The high

before crossover	after crossover
0 0 0 0 0 0 0 0	0 0 0 0 1 1 1 1
1 1 1 1 1 1 1 1	1 1 1 1 0 0 0 0

Figure 2.6: One-point crossover, cross site at locus 5.

before crossover	after crossover
0 0 0 0 0 0 0 0	0 0 1 1 1 1 0 0
1 1 1 1 1 1 1 1	1 1 0 0 0 0 1 1

Figure 2.7: Two-point crossover, cross site at 3 and 7.

before crossover	after crossover
0 0 0 0 0 0 0 0	0 1 0 1 0 1 0 1
1 1 1 1 1 1 1 1	1 0 1 0 1 0 1 0

Figure 2.8: Uniform crossover.

level operators include migration, marriage restriction, and sharing functions [13]. We introduce two low-level operators.

Diploidy and Dominance. So far we have only discussed the simplest genetic structure: *haploid* or single chromosome genetic structure. In genetics, there exist *diploidy* and *polyploidy,* more complicated structures in which one genotype is constructed by two, or more than two, chromosome strings. For example, the following genotype (diploidy) is made up of two chromosome:

aBcDeF,

AbCdef.

On the same locus we have two different alleles, which in nature could represent different phenotypic characteristics. For example, if B represents the *blue eye* gene and b the *yellow eye* gene, then the phenotype can not express *blue eye* and *yellow eye* at the same time, it must use dominance to decide which gene is expressed in the next generation. In the above example, if the upper-case letter is dominant to lower-case letter, the next generation's expressed phenotype would be

ABCDeF.

In nature, animals and plants with diploid or polyploid structure have been the most capable of surviving, because their genetic constitution does not easily forget the lessons learned prior to previous environmental shifts [13]. The application of diploidy and dominance in genetic search can be found in [26].

Inversion. Inversion is a reordering operator - it reshufles the chromosome structure:

before inversion 1 1 **0 1 1 1** 0 1

after inversion 1 **1 1 1 1 0** 0 1.

It changes the whole genotype structure, so that a better offspring may be produced. For more detailed studies of the these operators and other operators, see [13]

2.5 Genetic Algorithm at Work - an Example

To illustrate the implementation of a simple genetic algorithm and schema processing, we will use the simple function optimization example given in [13]. This problem is a maximization problem, the first step of which to optimize the function $f(x) = x^2$ over the interval (i.e. parameter set) [0-31] is to encode the parameter set x, for example as a five digit binary string in Table 2.3 [13], generated randomly by a random number generator.

Firstly, we use proportionate selection, for example roulette wheel selection, to construct the intermediate population, which can be called the mating pool. The strings are selected according their fitness values: the expected numbers for a string being reproduced, $f_i/\overline{f}$, for first string is 0.58, the second string is 1.97, the third string is 0.22, and the fourth is 1.23. So the number of copies that the mating pool receives from the initial population strings are 1, 2, 0, and 1 respectively.

Secondly, we apply crossover to the mating pool (Tables 2.3 and 2.4 [13]). Two strings are randomly selected to mate, and the crossover sites are also randomly selected. After crossover has been applied to the pairs of strings, the new population is obtained. The average fitness of the new population is improved, so is the evolution.

Finally, we consider the involvement of mutation. Suppose the mutation rate is 0.001, then the probability for mutation to occur is 0.001*4*5 = 0.02, so in this case, no mutation occurs.

In Tables 2.3 and 2.4, by using the schema theorem (2.11), we can also perform schema processing. We chose three schemata: $H_1 = \{ 1 * * * * \}$, $H_2 = \{* 1 0 * * \}$ and $H_3 = \{ 1 * * * 0 \}$. After reproduction, the expected number of copies of the schemata are

$$M(H_1, \text{intermediate}) = 2*469/293 = 3.20,$$
$$M(H_2, \text{intermediate}) = 2*320/293 = 2.18,$$
$$M(H_3, \text{intermediate}) = 1*576/293 = 1.64.$$

After crossover and mutation, the expected number of copies of the schemata are [3]

$$M(H_1, \text{t+1}) = 2*(469/293)*(1 - 1*(0/4))(1 - 0.001*0) = 3.20,$$
$$M(H_2, \text{t+1}) = 2*(320/293)*(1 - 1*(1/4))*(1 - 0.001*1) = 1.64,$$
$$M(H_3, \text{t+1}) = 1*(576/293)*(1 - 1*(4/4))*(1 - 0.001*4) = 0.00.$$

Because of the long defining length, the crossover will usually destroy schema H_3. Therefore, we can see that string processing and schema processing produce

[3]The crossover rate $p_c = 1.0$ means that crossover would definitely happen. The mutation rate $p_m = 0.001$.

Table 2.3: String Processing and Schema Processing by Hand.

String Processing

String No.	Initial Population	x	f(x)	Expected Count	Actual Count
1	0 1 1 0 1	13	169	0.58	1
2	1 1 0 0 0	24	576	1.97	2
3	0 1 0 0 0	8	64	0.22	0
4	1 0 0 1 1	19	361	1.23	1
Sum			1170	4.00	4.0
Average			293	1.00	1.0
Max			576	1.97	2.0

Schema Processing

		Before Reproduction	
	Schema	String Representatives	Schema Average Fitness f(H)
H_1	1 * * * *	2, 4	469
H_2	* 1 0 * *	2, 3	320
H_3	1 * * * 0	2	576

Table 2.4: String Processing and Schema Processing by Hand, continuation of Table 2.3.

String Processing

Mating Pool	Mates	Swapping	New Population	x	f(x)
0 1 1 0 1	1	0 1 1 0 [1]	0 1 1 0 0	12	144
1 1 0 0 0	2	1 1 0 0 [0]	1 1 0 0 1	25	625
1 1 0 0 0	2	1 1 [0 0 0]	1 1 0 1 1	27	729
1 0 0 1 1	4	1 0 [0 1 1]	1 0 0 0 0	16	256
Sum					1754
Average					439
Max					729

Schema Processing

After Reproduction			After All Operators		
Expected Count	Actual Count	String Respresentives	Expected Count	Actual Count	String Respresentives
3.20	3	2,3,4	3.20	3	2,3,4
2.18	2	2,3	1.64	2	2,3
1.97	2	2,3	0.0	1	4

similar results.

2.6 Summary

In this chapter, we have outlined the basic concepts, mathematical foundations, and operators of the genetic algorithm. We have also given a hands-on example to show how a genetic algorithm works.

In fact, many genetic algorithm models have been introduced by researchers largely working from an experimental perspective, such as the Simple Genetic Algorithm (SGA) [13], Genitor [21], CHC [27], the Parallel Genetic Algorithm [28], etc.. Many of these genetic algorithms are application oriented, and the interest is typically in genetic algorithms as optimization tools. In this thesis, we apply genetic algorithms to the adaptation of adaptive infinite impulse response (IIR) filtering problems. We will concentrate on the Simple Genetic Algorithm and Genitor in Chapters 5 and 6.

Chapter 3

Adaptive Infinite Impulse Response Filters

3.1 Introduction - Adaptive Filters and Their Applications

Adaptive filters involve the use of a programmable filter whose frequency response or transfer function is altered, or adapted, to pass without degradation the desired components of the signal and to attenuate the undesired or interfering signals, or reduce any distortion on the input signal [29]. Figure 3.1 shows an adaptive filter configuration. A signal *x(n)* is the input to an adaptive programmable filter, the adaptive filter output *y(n)* is compared with the desired signal *d(n)*, and the difference is sent to an adaptive algorithm to adjust the adaptive filter coefficients. This adjustment drives the filter output to become closer and closer to the desired

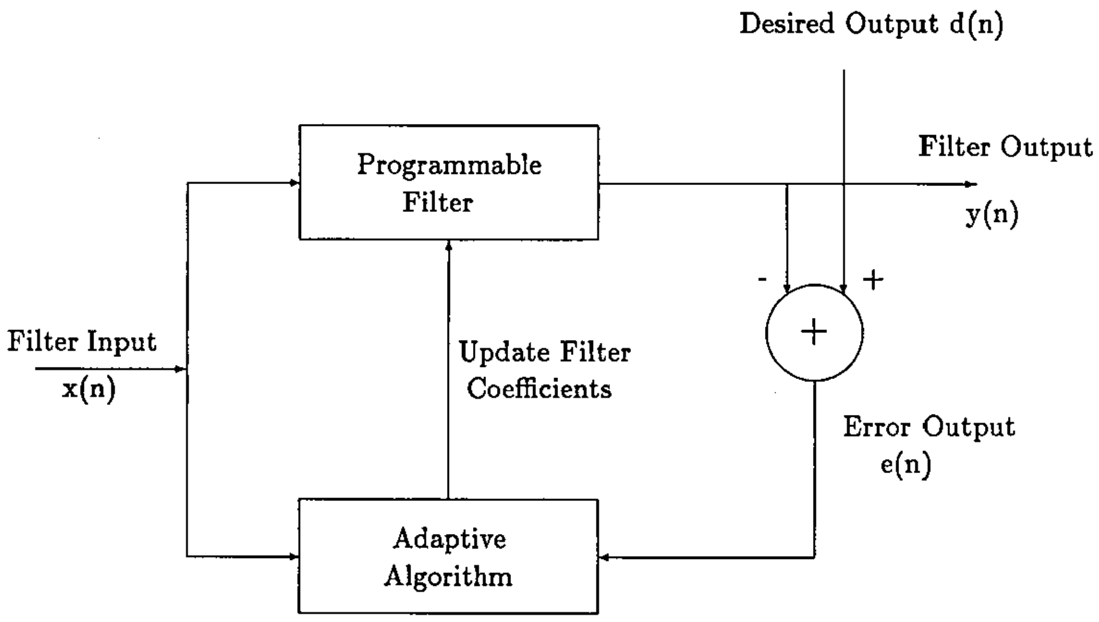

Figure 3.1: A Generic Block Diagram of an Adaptive Filter.

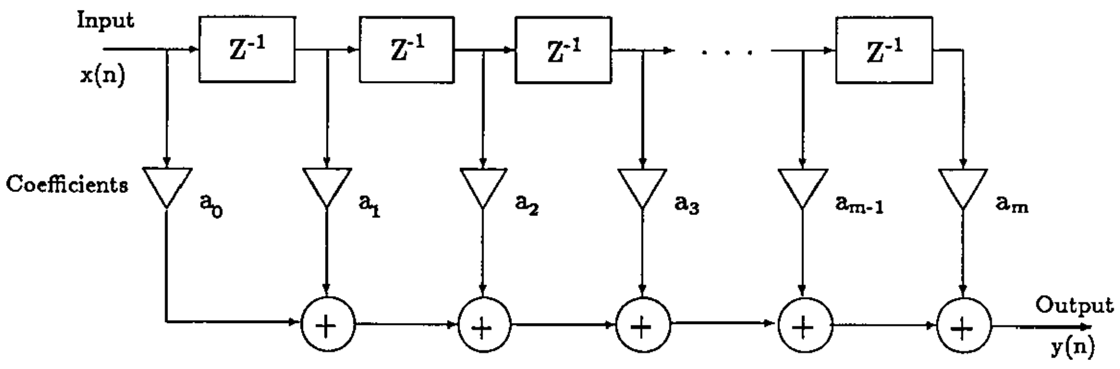

Figure 3.2: An FIR Filter Block Diagram.

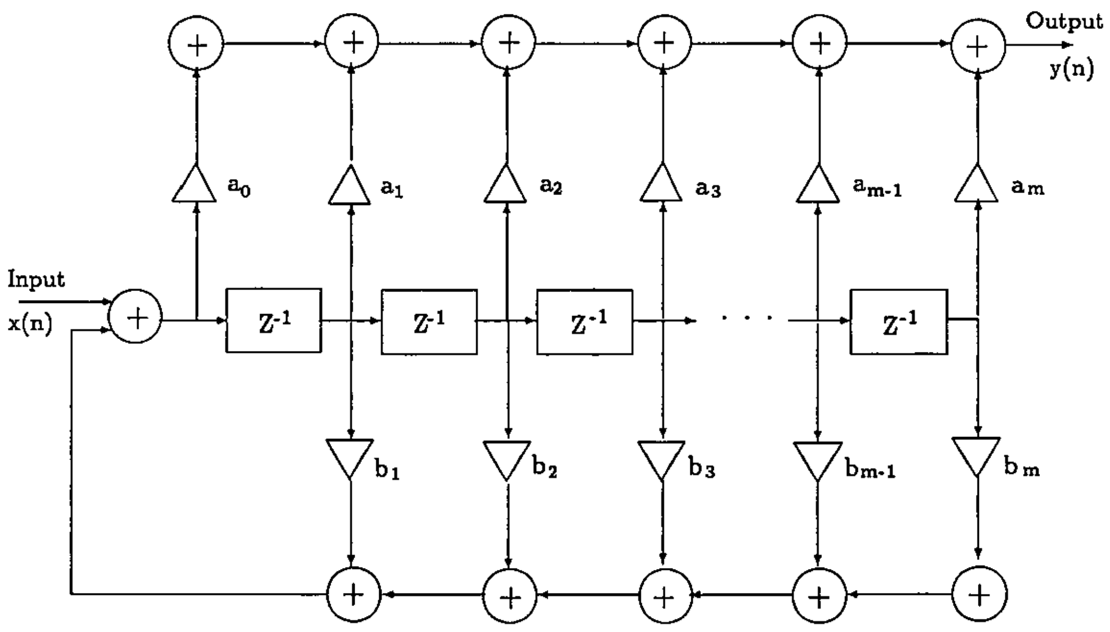

Figure 3.3: An IIR Filter Block Diagram.

output, so the unwanted components in *x(n)* are eliminated. The desirable features of adaptive filters are their ability to operate effectively in an *a priori* unknown environment and also to track time variations in the input statistics.

Linear adaptive filters can be classified as adaptive finite impulse response (FIR) filters and adaptive infinite impulse response (IIR) filters. FIR filters (Figure 3.2) are generally used in the application of adaptive filters due to their inherent stability. FIR filter algorithms such as LMS, RLS, FTF and QR-RLS, etc., [29] - [35], are well established. In particular, gradient algorithms are very suitable for adaptive FIR filtering as the error surface is quadratic and unimodal with respect to the filter coefficients.

For certain real physical systems, adaptive IIR filter (Figure 3.3) [29], [31], [36], [37] can be more economical, in the sense of lower filter order, compared to their adaptive FIR filter counterparts. However the error surface of an adaptive IIR filter can be multi-modal, making it difficult for IIR adaptation algorithms to find the global optimum. Instability is another very important issue to consider, especially when the poles are quite close to the unit circle, in which case adaptation noise can result in violation of the stability condition.

The direct form adaptive IIR filter implementation can exhibit high roundoff noise in the presence of finite precision arithmetic, and remains susceptible to quantization limit cycles [38]. If the poles of the IIR filter are close to the unit circle, for conventional gradient adaptive IIR filter algorithms, the direct form IIR filter's stability is not guaranteed. The algorithms in [39], which are variants of the Steiglitz-McBride technique [40], where the filter structure remains direct form, fail to converge for this condition.

This has motivated researchers to look for alternative structures and algorithms. Cascade, parallel and lattice structures have been documented [10], [41] and [42]. The solution given in [41] uses the LMS algorithm on parallel and cascade form adaptive IIR filters. This introduces additional saddle points into the performance surface, which are unstable solutions in the parameter space [41]. The solution given in [42] uses the LMS algorithm on the lattice form adaptive IIR filter, which maintains computational complexity $O(M^2)$ for gradient calculation. The algorithms in [10] are normalized lattice-based, the first algorithm being a reinterpretation of the Steiglitz-McBride method, while the second is a variation on the output error method, both of them of $O(M)$ complexity. The coefficients are updated by using the QR-based Gauss-Newton algorithm [10], [43] which requires many matrix computations, and for the case where the poles are extremely close to the unit circle, the algorithms fail to converge.

Genetic Algorithms and stochastic learning automata [44] are alternative solutions to the adaptive IIR filtering problems. The latter solution has been shown to be successful in tackling some IIR filtering problems [44]. In our studies, we use the genetic algorithm solution to the IIR filtering problems in Chapters 5 and 6.

Both FIR and IIR filters have been successfully applied in many areas such as prediction, communication channel equalization, echo cancellation, system identification, image processing and pattern recognition, etc.. We give a brief description of the first four applications.

- *Adaptive signal prediction* (Figure 3.4) is an adaptive system configured to perform prediction of a signal, based upon its previous values. Here the predicted or the desired signal is the current signal. This signal is fed through a delay stage into the adaptive filter. The predictor is trying to minimize the error signal, which is the difference between the desired signal and the adaptive filter output. The adaptive filter output is a combination of previous filter inputs. When the error is minimized, this adaptive filter output estimates the current input signal. An application of adaptive predictors is cancellation of periodic interference [45]. Another application is the efficient encoding of speech signals [46] - [48].

- *Adaptive channel equalization* (Figure 3.5) is inverse modeling [29], [31], [49] - [52]. For this technique, the desired adaptive filter output is an estimate of the original transmitted message sequence and the input to the adaptive filter is the received data sequence, which is subject to distortion due to transmission through the channel. The adaptive filter is updated so that the error between the equalizer output and the desired response, which is again available as either the training data or previous equalizer decisions, is minimized, hence signal distortion caused by the transmission channel is removed.

- *Adaptive noise cancellation* (Figure 3.6) is an adaptive processing system to cancel interference [53]. For this system, a signal, s, is corrupted by interference n, resulting in the combined signal, $s+n$. A correlated, but distorted, estimate of this noise, $\overline{n}$, is also available. So the corrupted signal, $s+n$, is fed to the desired input and the estimated noise is fed to the adaptive filter input. By minimizing the difference between the two signals, in term of error, the adaptive filter is

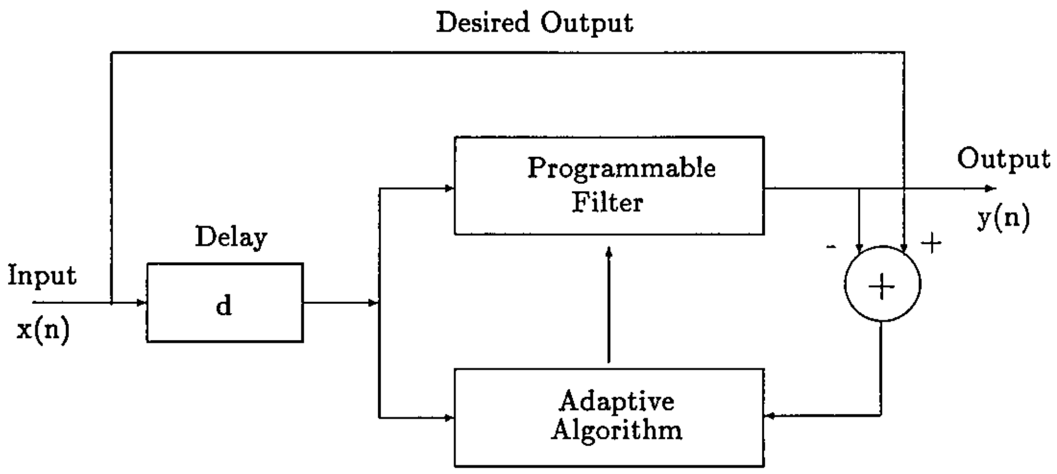

Figure 3.4: Adaptive Prediction.

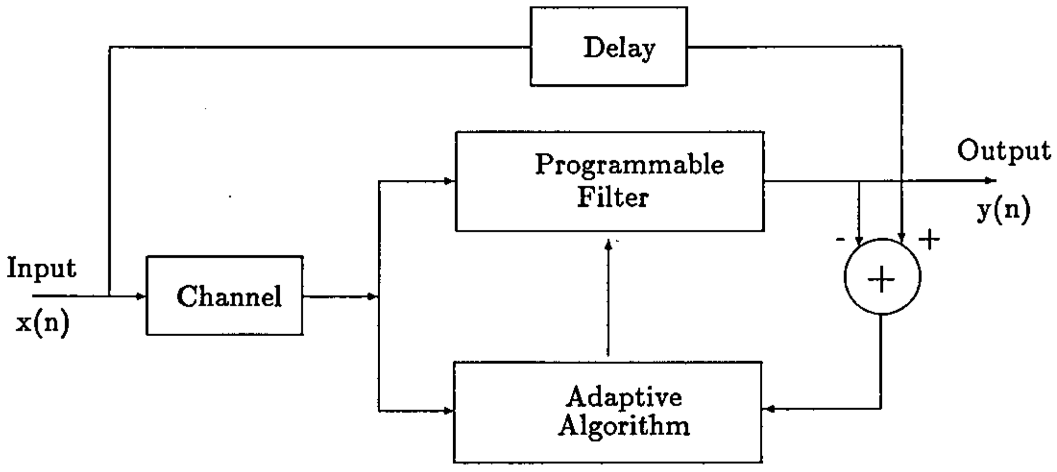

Figure 3.5: Adaptive Channel Equalization.

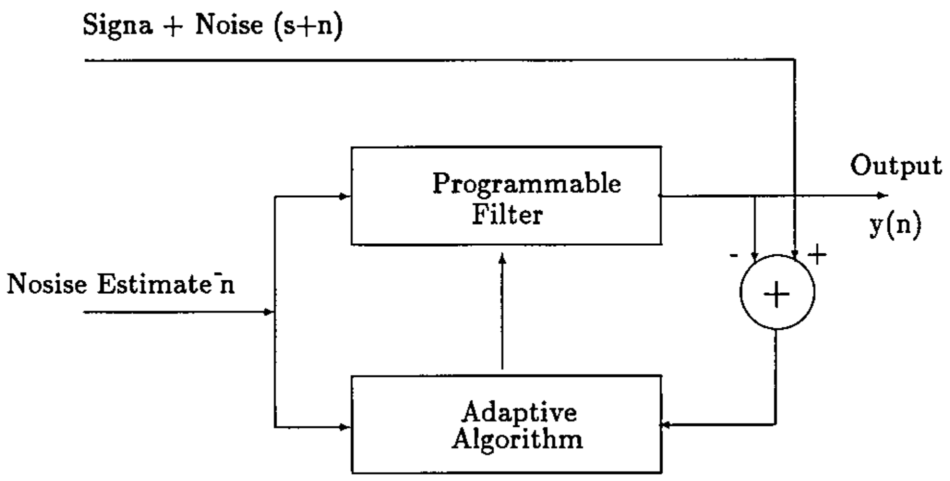

Figure 3.6: Adaptive Noise Cancellation.

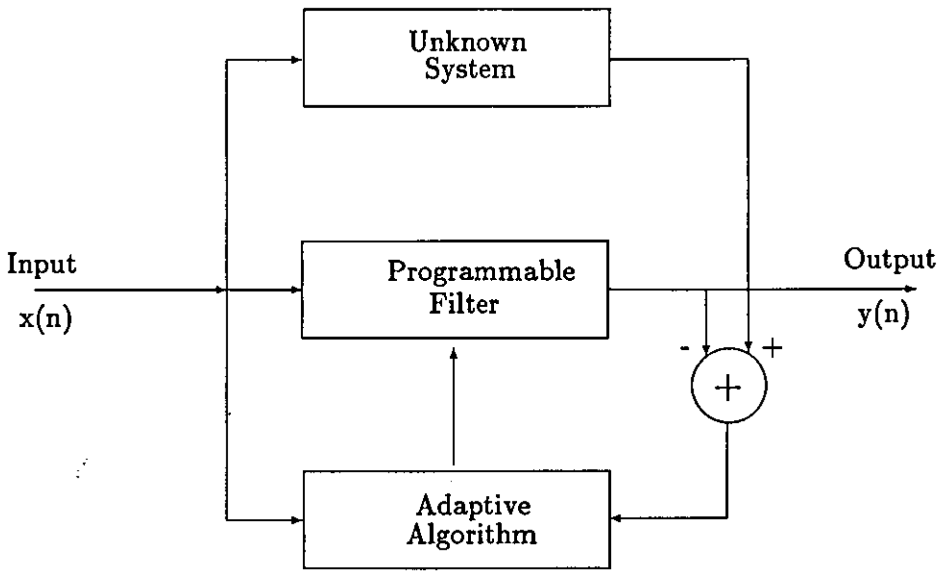

Figure 3.7: Adaptive System Identification.

configured to estimate the actual noise. The noise estimate may then be subtracted from the noisy signal, resulting in an estimate of the original signal. A typical example of the application of this technique is the cancellation of additive noise from speech signals [54].

• *Adaptive system identification* [29], [32], [43] is shown in Figure 3.7. The aim of this adaptive filter is to find a system with a transfer function which closely approximates to the unknown system's transfer function. A signal x(n) is fed into the unknown system and also the adaptive filter. The output, which the unknown system gives in response to this input, is the desired response of the adaptive filter and so is fed into the desired response input. By minimizing the difference of two system's output, the adaptive filter learns to respond like the unknown system. The parameters of the adaptive filter try to pertain to the unknown system's. The output of the unknown system may be corrupted by a small amount of noise, so that it can not be identified exactly. Throughout this thesis, system identification is used as the main system configuration.

3.2 Equation-Error and Output-Error Adaptive IIR Filters

Fundamentally, there have been two approaches to adaptive IIR filtering that correspond to different formulations of the prediction error. They are known as equation-error and output-error formulations [36], [37].

3.2.1 Equation-Error Formulation

In the equation-error formulation [55], the delayed desired response *d(n-1)* and the input *x(n)* are fed into the filter, to generate an estimate of *d(n)*. It can be characterized by the nonrecursive difference equation

$$y_e(n) = \sum_{k=0}^{M} a_k(n)x(n-k) + \sum_{k=1}^{M} b_k(n)d(n-k), \tag{3.1}$$

where $a_k(n)$ and $b_k(n)$ are the adjustable coefficients. Alternatively, this formulation can be rewritten as:

$$y_e(n) = A(n,z)x(n) + B(n,z)d(n), \tag{3.2}$$

where the polynomials in z represent time-varying filters and are defined by

$$A(n,z) = \sum_{k=0}^{M} a_k(n)z^{-k} \quad and \quad B(n,z) = \sum_{k=1}^{M} b_k(n)z^{-k}. \tag{3.3}$$

This formulation is depicted in Figure 3.8 [36]. It does not have feedback, hence it is simply operated by all zero and all pole filters, and the corresponding algorithms are well understood. The difference with the FIR filter is that the FIR filter is strictly an all zero model since *B(n, q)* = *0*. The equation-error approach can lead to biased estimates of the coefficients [36], in that the converged coefficients obtained with this approach are generally different from those generated by the output-error formulation. The error $e_e(n)$ = *d(n)* - $y_e(n)$ is a linear function of the coefficients, so that the mean square error (MSE) is a quadratic function with a single global minimum.

Equation (3.1) can also be compactly written as the inner product

$$y_e(n) = \theta^T(n)\varphi_e(n), \tag{3.4}$$

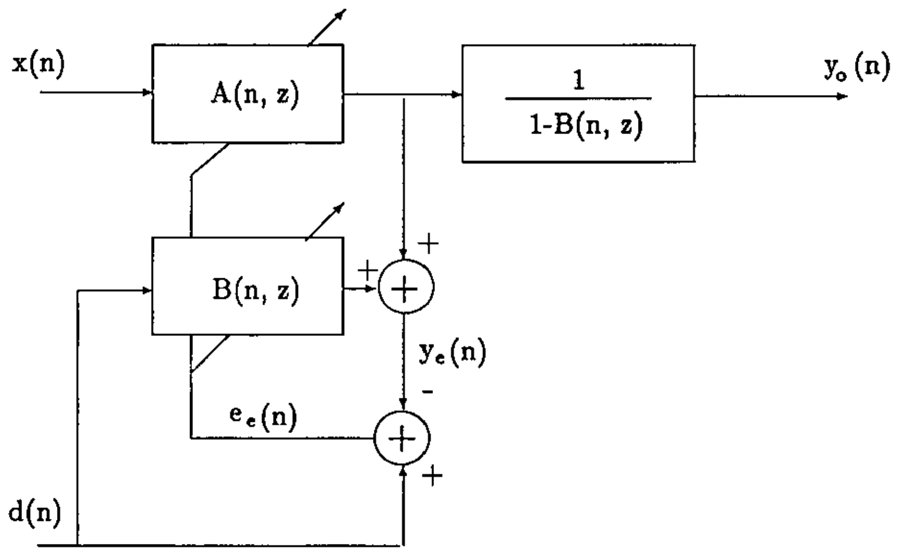

Figure 3.8: Equation-error formulation

where the coefficient vector $\theta(n)$ and the signal vector $\varphi_e(n)$ each have length *(2M+1)* elements and are defined as

$$\theta(n) = [a_0(n), ..., a_M(n), b_1(n), ..., b_M(n)]^T \tag{3.5}$$

$$\varphi_e(n) = [x(n), ..., x(n-M), d(n-1), ..., d(n-M)]^T. \tag{3.6}$$

The equation (3.4) is a linear regression and can be solved by using the LMS or RLS algorithms. The LMS (Least-mean-square) algorithm [30] is a recursive gradient-descent method that searches for the minimum of the mean square error; the RLS (recursive-least-square) algorithm [32] recursively minimizes a least-squares criterion.

3.2.2 Output-Error Formulation

Figure 3.3 shows a direct form output-error adaptive IIR filter. This output-error formulation can be characterized by the following recursive difference quation:

$$y_o(n) = \sum_{k=0}^{M} a_k(n)x(n-k) + \sum_{k=1}^{M} b_k(n)y_o(n-k), \tag{3.7}$$

In this formulation, the input signal *x(n)* and the previous output signals are fed into the filter to generate the current output signal. Because this formulation depends on the feedback of output signals, it has a greater complexity due to the nonlinearity compared with equation-error approach. The equation (3.7) can be rewritten as

$$y_o(n) = \frac{A(n,z)}{1 - B(n,z)}x(n), \tag{3.8}$$

and Figure 3.3 can be redraw as Figure 3.9.

Equation (3.7) or (3.8) can also be written as the inner product

$$y_o(n) = \theta^T(n)\varphi_o(n), \tag{3.9}$$

where the coefficient vector is given by equation (3.5) and the signal vector by

$$\varphi_o(n) = [x(n), ..., x(n-M), y_o(n-1), ..., y_o(n-M)]^T. \tag{3.10}$$

Clearly, we can see that the filter output $y_o(n)$ is a nonlinear function of the coefficients $\theta(n)$, the reason being that the output $y_o(n-k)$ of $\varphi_o(n)$ depends on previous coefficient values. The error $e_o(n)$ = *d(n)* - $y_o(n)$ is also a nonlinear function of the coefficients - the mean square error function is not a quadratic

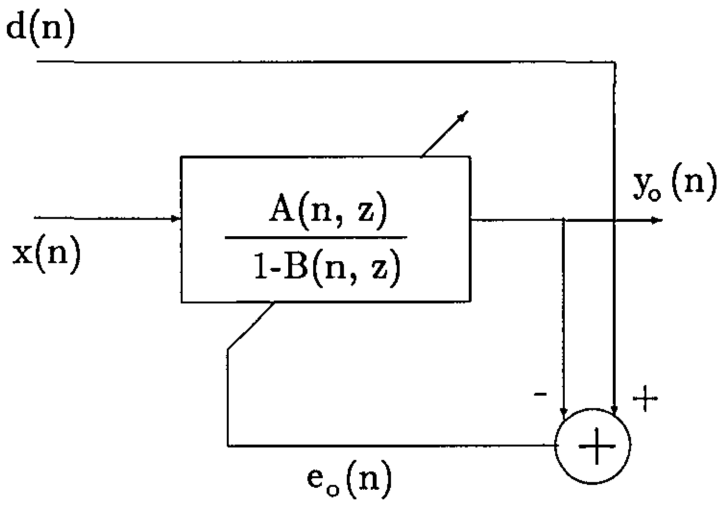

Figure 3.9: Output error formulation.

function and could have multiple minima [56]. Adaptive algorithms that are based on gradient-search methods (LMS or RLS) could converge to one of these local solutions, resulting in suboptimal performance and inaccurate estimate of the coefficients [36]. We will discuss these matters in the later chapters. In this thesis, we mostly use the output-error formulation for our studies.

3.3 Adaptive IIR Filter Algorithms

Most of the adaptive IIR filter adaptation algorithms are gradient based. These adaptive filtering algorithms revolve around a generic coefficient update formula

$$\theta(n+1) = \theta(n) + \mu(n)\alpha(n)\psi(n) \tag{3.11}$$

or in Gauss-Newton form [43]:

$$\theta(n+1) = \theta(n) + [\sum_{k=0}^{n} \lambda^{n-k}\psi(k)\psi^t(k)]^{-1}\psi(n)\alpha(n), 0 \ll \lambda \ll 1. \tag{3.12}$$

where $\theta(n)$ is the coefficient vector, $\alpha(n)$ represents an *a priori* error signal, $\mu(n)$ is a sequence of step size parameters, λ is the forgetting factor and $\psi(n)$ is a gradient vector, whose components are ordered in one-to-one correspondence with the elements of $\theta(n)$. The recursive adaptive algorithm in [43] adapts the filter coefficients to minimize the MSE (mean-square-error) cost function $\xi = \mathrm{E}[e^2(n)]$, where $e(n) = d(n) - y(n)$. Because ξ is generally unknown or the signals are nonstationary, the algorithm is designed to minimize ξ at each instant of time, and the instantaneous estimate of $\xi(n)$ is given by $\xi(n) \approx e^2(n)$. The gradient is defined as

$$\psi(n) \equiv \nabla_\theta \xi \approx \frac{\partial \xi(n)}{\partial \theta(n)} = -e(n)\nabla_\theta y(n), \tag{3.13}$$

where $y(n)$ is the adaptive filter output.

By using the gradient-descent method and $\xi = \mathrm{E}[e^2(n)]$ to evaluate the gradient vector $\psi(n)$, the equation (3.11) becomes the LMS algorithm [32]. By using the formulation

$$\xi = \sum_{k=1}^{n} \lambda^{n-k}|e(k)|^2, \tag{3.14}$$

to evaluate the gradient vector $\psi(n)$, the equation (3.12) becomes the RLS algorithm [29], [32], where λ is the forgetting factor.

The earliest study on IIR filters using gradient algorithms can be traced back to 1976: Feintuch [57] applied the LMS algorithm to IIR filters which triggered a rebuttal [58] - [60] as well as new interest in adaptive IIR filtering. This work [57] - [60] in adaptive IIR filtering was mainly restricted to extending Widrow's LMS method of adaptive FIR filtering based on gradient search techniques. As discussed in [39], these algorithms may have guaranteed global convergence only for the unimodal error surface. This severely limits their usefulness.

Another existing family of adaptive IIR filtering algorithms is represented by the group of algorithms based on the concept of hyperstability [29], [37], [61] - [63]. Among these, the hyperstable adaptive recursive filter (HARF) was proven asymptotically convergent under the 'strict positive real' (SPR) assumption [62]. However, the SPR requirement is a major obstacle in the practical application of HARF.

Recently, two approaches [10], [39] claim that they can either solve the multimodal case or the poles close to the unit circle case, and have a reduced computational complexity. We will choose these two approaches to compare with our genetic algorithm approach. The simulation results based on these two approaches are given in Chapter 4.

3.4 Summary

In this chapter, we have given a brief introduction to adaptive filters and their applications. We have described the fundamental approaches to adaptive IIR filters, namely are the equation-error formulation and output-error formulation, and their properties. We have given a generic adaptive algorithm formula, and also reviewed the conventional adaptive IIR filter algorithms, their convergence and stability properties. Based on the two more recent approaches mentioned in this chapter, we present the results of various simulations in the following chapter, which will lay a background for comparison with genetic algorithms in Chapters 5 and 6.

Chapter 4

Simulations of Two Gradient IIR Filter Algorithms

4.1 Introduction

In Chapter 3, we briefly introduced adaptive IIR filter algorithms: the gradient based algorithms [57] - [60], which may only have guaranteed global convergence for the unimodal error surface, and the hyperstable adaptive recursive filter algorithms which require the SPR condition, a major obstacle in practical applications. A new adaptive IIR filter developed by H. Fan and W. K. Jenkins [39] overcomes the previous gradient IIR filter algorithm's problems. But stability is not guaranteed by this algorithm. Thus, in theory, a stability monitoring device has to be incorporated into these algorithms [39]. For the case of the filter's poles approaching the unit circle, we found this algorithm failed to converge.

Another algorithm developed by P.A. Regalia [10] has a number of advantages over traditional gradient algorithms, especially multimodal and high order IIR filter problems. We also found this algorithm failed to solve an IIR filter problem when the poles are extremely close to the unit circle. In sections 4.3 and 4.4 we will present some experimental results to prove our claims.

4.2 The Steiglitz-McBride Identification Technique

Both of the IIR filtering schemes we describe are based on the Steiglitz-McBride identification technique [40], [64]. This technique was developed by K. Steiglitz and L. E. Mcbride in 1965. A brief introduction to this technique will be given before we proceed on to discussing the algorithms. The Steiglitz-McBride identification technique is shown diagrammatically in Figure 4.1. It is an iterative technique. An initial estimate of the unknown system's denominator polynomial $D_n(z)$ is used as a prefilter for both the input and output sequences relating the unknown system transfer function. The prefiltered signals are fed to the numerator and denominator polynomials $N_{n+1}(z)$ and $D_{n+1}(z)$ to minimize a typically quadratic measure of the error $e(n)$. The prefilters are updated to $1/D_{n+1}(z)$ for the next sample instant, and the procedure continues by seeking $N_{n+2}(z)$ and $D_{n+2}(z)$. If convergence is obtained, that is, $D_{n+1}(z) = D_n(z)$, Figure 4.1 becomes Figure 4.2, which is in the form of a converged IIR filter whose output is subtracted from the unknown system output to produce the error signal. We will

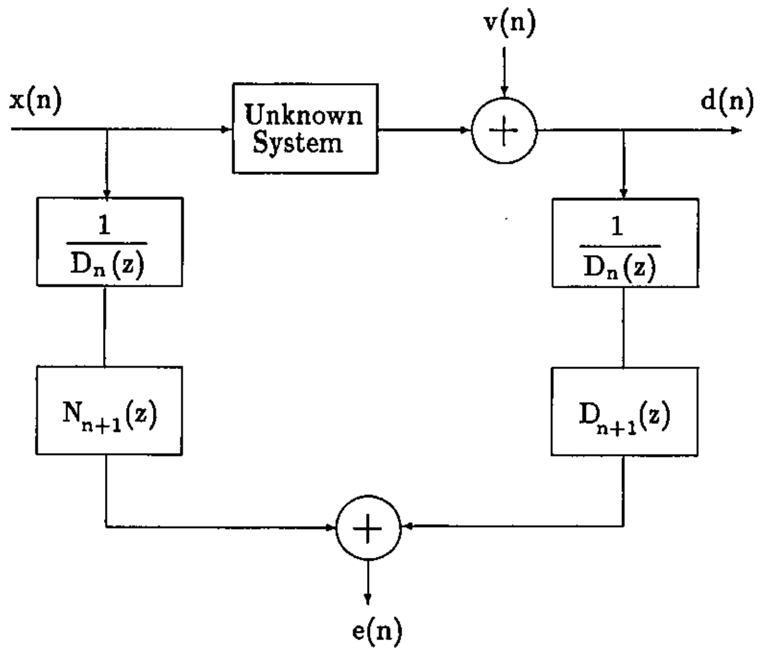

Figure 4.1: The Steiglitz-McBride identification scheme.

see how this technique works in both schemes.

4.3 Fan's Algorithms

The algorithms in [39] are a family of stochastic approximation variants of the Steiglitz-McBride identification scheme. Suppose we have the system model of Figure 4.3, which is described by the following equations

$$w(n) = \sum_{i=0}^{n_a} a_i(n)x(n-i) + \sum_{i=1}^{n_b} b_i(n)w(n-i), \tag{4.1}$$

$$d(n) = w(n) + v(n), \tag{4.2}$$

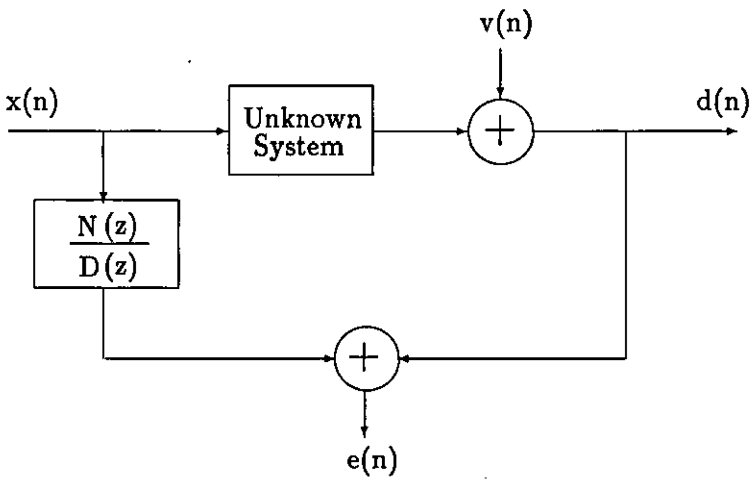

Figure 4.2: The equivalent system of Fig. 1 at any stationary point.

$$y(n) = \sum_{i=0}^{\overline{n}_a} a_i(n)x(n-i) + \sum_{i=1}^{\overline{n}_b} b_i(n)y(n-i), \tag{4.3}$$

$$e(n) = d(n) - y(n). \tag{4.4}$$

Fan's model (Figure 4.4) is obtained by adding three prefilters to the system identification model (Figure 4.3). Here

$$A = \sum_{i=0}^{n_a} a_i(n)z^{-i} \; and \; B = \sum_{i=1}^{n_b} b_i(n)z^{-i}, \tag{4.5}$$

$$\overline{A} = \sum_{i=0}^{\overline{n}_a} \overline{a}_i(n)z^{-i} \; and \; \overline{B} = \sum_{i=1}^{\overline{n}_b} \overline{b}_i(n)z^{-i}. \tag{4.6}$$

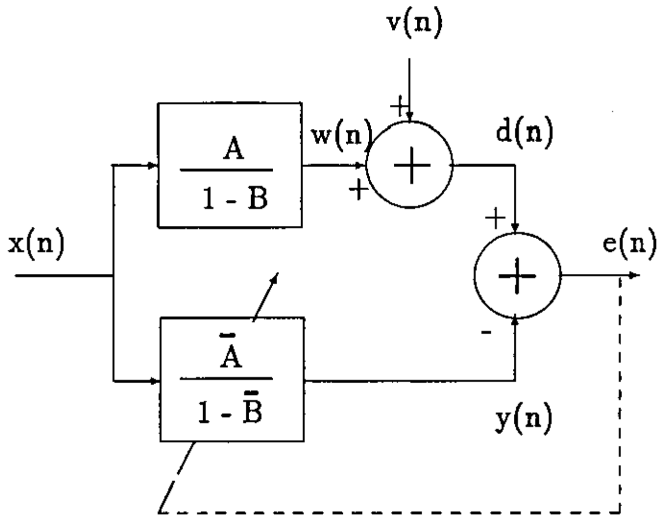

Figure 4.3: System identification mode.

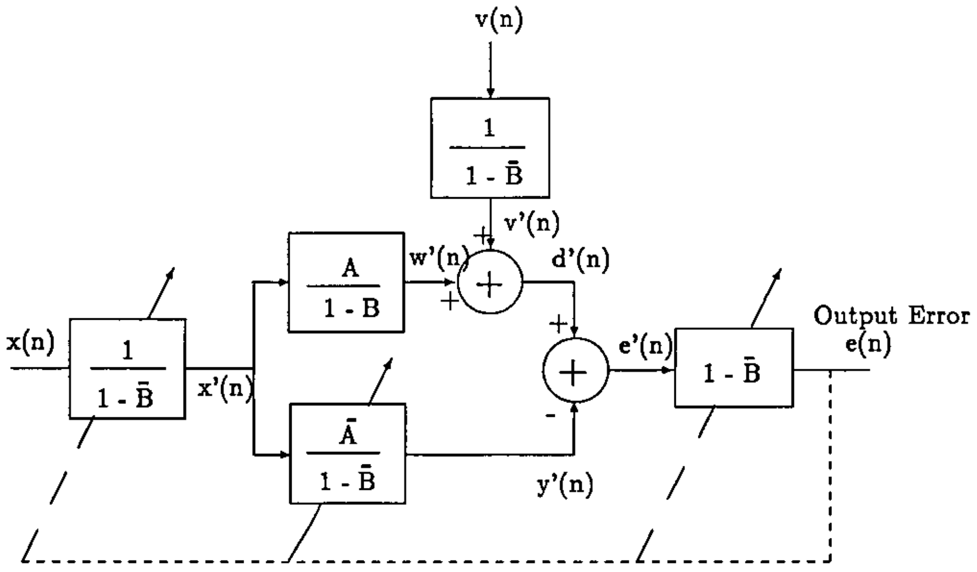

Figure 4.4: System identification mode of Fan's algorithm.

Fan's system identification model algorithm (SIM) is given by table 3.1 in which

Table 3.1: Fan's SIM algorithm.

$$\bar{a}_i(n+1) = \bar{a}_i(n) + \tau e(n)x'(n-i), \quad i = 0, 1, \cdots, \bar{n}_a \tag{4.7}$$

$$\bar{b}_j(n+1) = \bar{b}_j(n) + \tau e(n)d'(n-j), \quad j = 1, 2, \cdots, \bar{n}_b \tag{4.8}$$

$$x'(n) = x(n) + \sum_{j=1}^{\bar{n}_b} \bar{b}_j(n)x'(n-j) \tag{4.9}$$

$$e(n) = e'(n) - \sum_{j=1}^{\bar{n}_b} \bar{b}_j(n)e'(n-j) \tag{4.10}$$

$$e'(n) = d'(n) - y'(n) \tag{4.11}$$

$$w'(n) = \sum_{i=0}^{n_a} a_i(n)x'(n-i) + \sum_{j=1}^{n_b} b_j(n)w'(n-j) \tag{4.12}$$

$$d'(n) = w'(n) + v'(n) \tag{4.13}$$

$$v'(n) = v(n) + \sum_{j=1}^{\bar{n}_b} \bar{b}_j(n)v'(n-j) \tag{4.14}$$

$$y'(n) = \sum_{i=0}^{\bar{n}_a} \bar{a}_i x'(n-i) + \sum_{j=1}^{\bar{n}_b} \bar{b}_j(n)y'(n-j) \tag{4.15}$$

τ is a constant. If τ is too large, the adaptive filter coefficients will go beyond the stable region and the filter will be unstable [39]. Fan also presents the adaptive filter model (AFM) and the independent filter (IF) algorithm.

We conducted three experiments using this algorithm. In the first one, we chose the dynamic plant transfer function [56] as

$$H_p(z) = \frac{1.0}{1.0 - 1.2z^{-1} + 0.6z^{-2}} \tag{4.16}$$

and the adaptive filter as

$$H_a(z) = \frac{\overline{a}_0(n)}{1.0 - \overline{b}_1(n)z^{-1} - \overline{b}_2(n)z^{-2}}. \tag{4.17}$$

The converged MSE results for this unimodal case are shown in Figure 4.5. The adaptive filter's coefficients, which are obtained by using this algorithm, are $\overline{a}_0(n)$ = 1.000000, $\overline{b}_1(n)$ = 1.200000, and $\overline{b}_2(n)$ = -0.600000.

The MSE results of the second experiment are shown in Figure 4.6. This is a example considered by Johnson [58], where

$$H_p(z) = \frac{0.05 - 0.4z^{-1}}{1.0 - 1.1314z^{-1} + 0.25z^{-2}} \tag{4.18}$$

$$H_a(z) = \frac{\overline{a}(n)}{1.0 - \overline{b}(n)z^{-1}}. \tag{4.19}$$

This is a bi-modal case, with minima of 0.976 and 0.277. If we choose the initial coefficient values near or equal to the local minima, the algorithm can converge to the global minima after a certain number of adaptations. The coefficients of the adaptive filter obtained by Fan's algorithm are $\overline{a}(n)$ = 0.899225 and $\overline{b}(n)$ = -0.314626.

In the third experiment, the plant (unknown system) transfer function is chosen as

$$H(z) = \frac{1.0}{1.0 - 1.4z^{-1} + 0.98z^{-2}}. \tag{4.20}$$

The adaptive filter transfer function is same as equation (4.17). This plant has two poles at 0.7±j0.7 (modulus = 0.99), which are very close to the unit circle. We found that Fan's algorithms always 'blows up' no matter what value τ has.

Fan's algorithms are based on direct form IIR filter structures, but have difficulty solving high order IIR filter problems and exhibit numerical implementation problems [10].

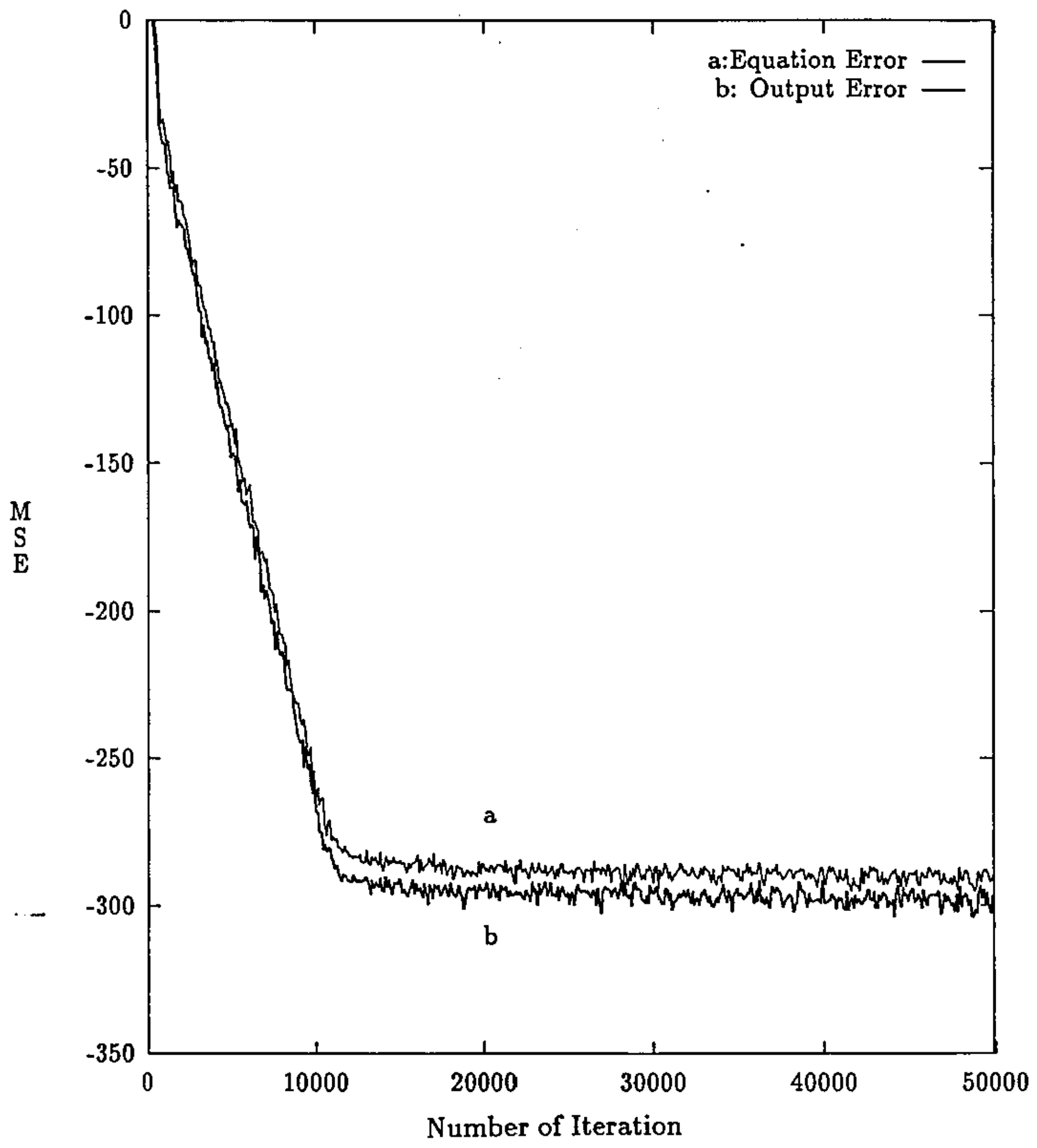

Figure 4.5: The unimodal case simulation of Fan's algorithm, MSE (dB) *vs* number of iteration. τ is set to 0.002, the initial coefficients are set to zero. The plot of mean square errors is obtained by averaging 20 independent square errors.

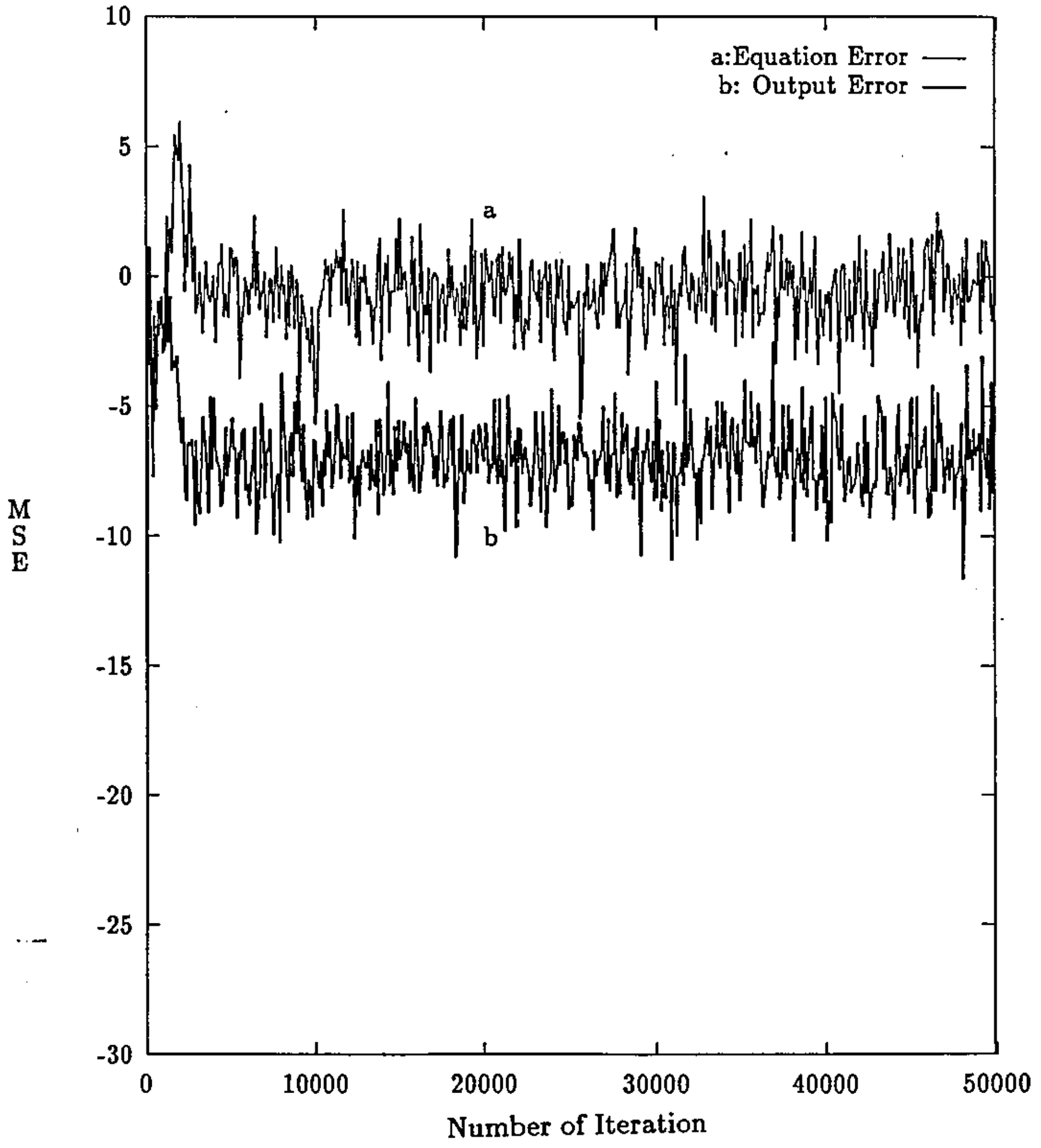

Figure 4.6: The bi-modal case simulation of Fan's algorithm, MSE (dB) *vs* number of iteration. τ is set to 0.001, the initial coefficients are set (-0.519, 0.114), which are the local minima. The plot of mean square errors is obtained by averaging 20 independent square errors.

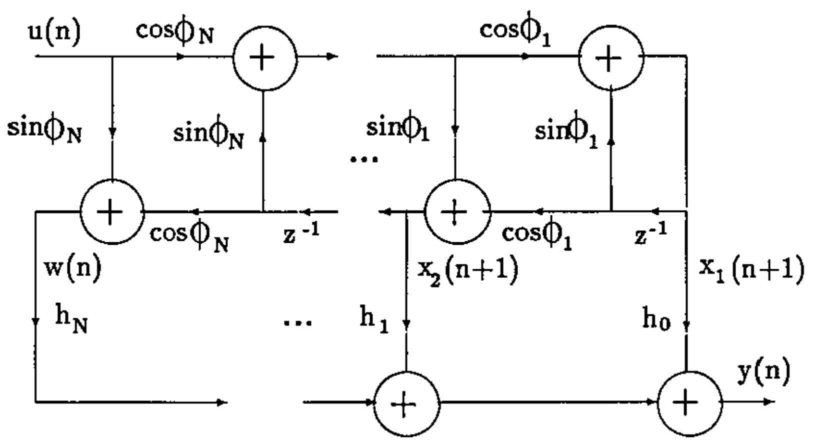

Figure 4.7: Normalized lattice filter.

4.4 Regalia's Algorithms

Phillip Regalia has proposed two adaptive IIR algorithms in [10]. Both of them use the normalized lattice filter structure (Figure 4.7) and the QR method to update the coefficients. The first algorithm is a reinterpretation of the Steiglitz-McBride method (Figure 4.8), while the second one is a variation on the output error method. State space models are employed in the algorithm derivations. We summarize the first QR algorithm in table 4.2.

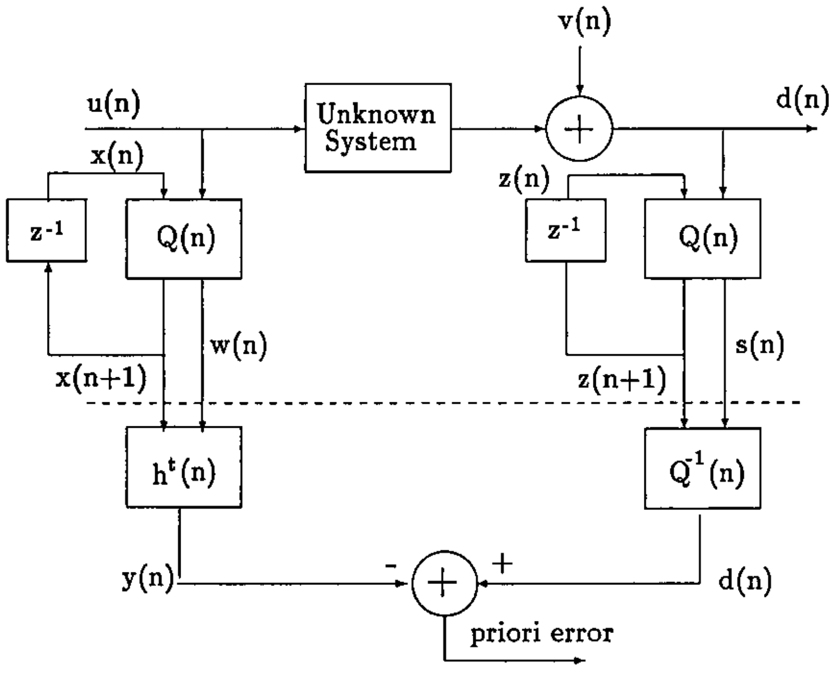

Figure 4.8: Regalia's algorithms model.

Table 4.2: Regalia's QR lattice algorithm.

$$y(n) = \mathbf{h}^t(n) \begin{bmatrix} \mathbf{x}(n+1) \\ w(n) \end{bmatrix} \tag{4.21}$$

$$\begin{bmatrix} \mathbf{x}(n+1) \\ w(n) \end{bmatrix} = \mathbf{Q}(n) \begin{bmatrix} \mathbf{x}(n) \\ u(n) \end{bmatrix} \tag{4.22}$$

$$\begin{bmatrix} \mathbf{z}(n+1) \\ s(n) \end{bmatrix} = \mathbf{Q}(n) \begin{bmatrix} \mathbf{z}(n) \\ d(n) \end{bmatrix} \tag{4.23}$$

$$\mathbf{Q}(n) = \mathbf{U}_1\mathbf{U}_2 \cdots \mathbf{U}_N \tag{4.24}$$

$$\mathbf{U}_k = \begin{bmatrix} \mathbf{I}_{k-1} & & & \\ & -sin\phi_k & cos\phi_k & \\ & cos\phi_k & sin\phi_k & \\ & & & \mathbf{I}_{N-k} \end{bmatrix} \tag{4.25}$$

$$\begin{aligned} \alpha(n) &= [0 \cdots 01]\, \mathbf{Q}^t(n) \begin{bmatrix} \mathbf{z}(n+1) \\ s(n) \end{bmatrix} - \mathbf{h}^t(n) \begin{bmatrix} \mathbf{x}(n+1) \\ w(n) \end{bmatrix} \\ &= \mathbf{q}^t_{N+1} \begin{bmatrix} \mathbf{z}(n+1) \\ s(n) \end{bmatrix} - \mathbf{h}^t(n) \begin{bmatrix} \mathbf{x}(n+1) \\ w(n) \end{bmatrix} \end{aligned} \tag{4.26}$$

$$-\frac{\partial \alpha(n)}{\partial h_i(n)} = \begin{cases} x_{i+1}(n+1), & i = 0, \cdots, N-1 \\ w(n), & i = N \end{cases} \tag{4.27}$$

$$-\frac{\partial\alpha(n)}{\partial\phi_i(n)} = \begin{cases} -z_i(n)\prod_{k=i+1}^{N} cos\phi_k(n), & i = 1,\cdots,N-1 \\ -z_N(n), & i = N \end{cases} \tag{4.28}$$

$$\hat{\mathbf{Q}}(n)\begin{bmatrix} \psi_t(n) \\ \lambda^{1/2}\mathbf{R}(n-1) \end{bmatrix} = \begin{bmatrix} 0 \\ \mathbf{R}(n) \end{bmatrix} \tag{4.29}$$

$$\mathbf{R}(0) = \kappa\mathbf{I} \tag{4.30}$$

$$\hat{\mathbf{Q}}(n) = \hat{\mathbf{Q}}_M \cdots \hat{\mathbf{Q}}_1 \tag{4.31}$$

$$\hat{\mathbf{Q}}_k = \begin{bmatrix} cos\varphi_k & & -sin\varphi_k & \\ & \mathbf{I}_{k-1} & & \\ sin\varphi_k & & cos\varphi_k & \\ & & & \mathbf{I}_{M-k} \end{bmatrix} \tag{4.32}$$

$$\hat{\mathbf{q}}_1(n) = \begin{bmatrix} \prod_{k=1}^{M} cos\varphi_k \\ \mathbf{g} \end{bmatrix} \tag{4.33}$$

$$\alpha(n)\mathbf{g} = \mathbf{R}(n)\Delta\theta(n) \tag{4.34}$$

$$g_k = sin\varphi_k \prod_{i=1}^{k-1} cos\varphi_i, \quad k = 1,2,\cdots M. \tag{4.35}$$

x(n) = [x_1(n) x_2(n) $\cdots$ x_N(n)] is the state vector, *w(n)* and *s(n)* are intermediate signals, Q(n) is an (N+1)×(N+1) matrix, $\mathbf{q}_{N+1}^t$ is the bottom row of $\mathbf{Q}^t$(n), ψ^t(n) is the gradient vector given by equations 4.27 and 4.28, $\hat{\mathbf{Q}}$(n) is the Givens rotation matrix [32], [35], and $\hat{\mathbf{q}}_1$ is the first column of $\hat{\mathbf{Q}}$(n). $\Delta\theta$(n) = θ(n+1) - θ(n) is the current coefficient value minus the previous coefficient value; κ is a small constant.

We performed experiments on two cases using Regalia's QR and LMS [1] algorithms. In the first case, the plant is

$$H_p(z) = \frac{0.5 - 0.4z^{-1} + 0.89z^{-2}}{1.0 - 0.89z^{-1} + 0.25z^{-2}}, \tag{4.36}$$

and the adaptive filter is normalized lattice (Figures 4.7 and 4.8). The mean square error plotted against the number of adaptations is shown in Figure 4.9. The converged coefficient result is given in Table 4.3. The normalized lattice (NL) plant coefficients are calculated according to Gray's formulations in [65], [66].

Table 4.3: The coefficients of the second order filter using Regalia's algorithm.

Coefficient	Plant (NL)	Adaptive Filter
$h_0(n)$	0.818782	0.818527
$h_1(n)$	0.404959	0.404920
$h_2(n)$	0.890000	0.889887
$\phi_1(n)$	-0.792342	-0.711977
$\phi_2(n)$	0.252680	0.250070

[1]Replaces the matrix gain term with the stepsize scalar in the Gauss-Newton standard form coefficient update formula (3.12).

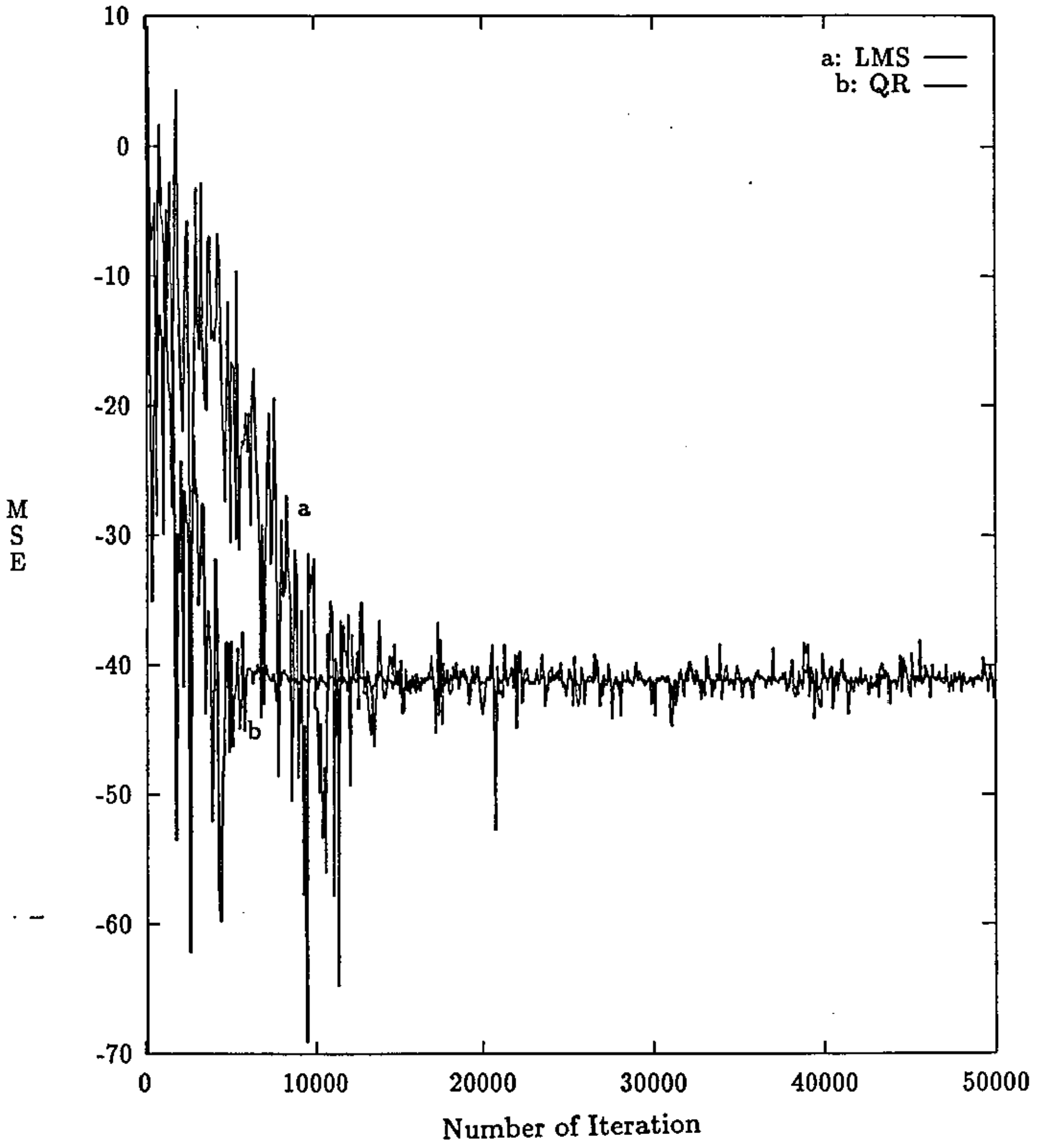

Figure 4.9: Regalia's normalized lattice algorithm. Second order case, MSE (dB) *vs* number of iteration. In the QR algorithm, κ is set to 0.15, $\lambda = 0.9988$. In the LMS algorithm $\mu = 0.00075$.

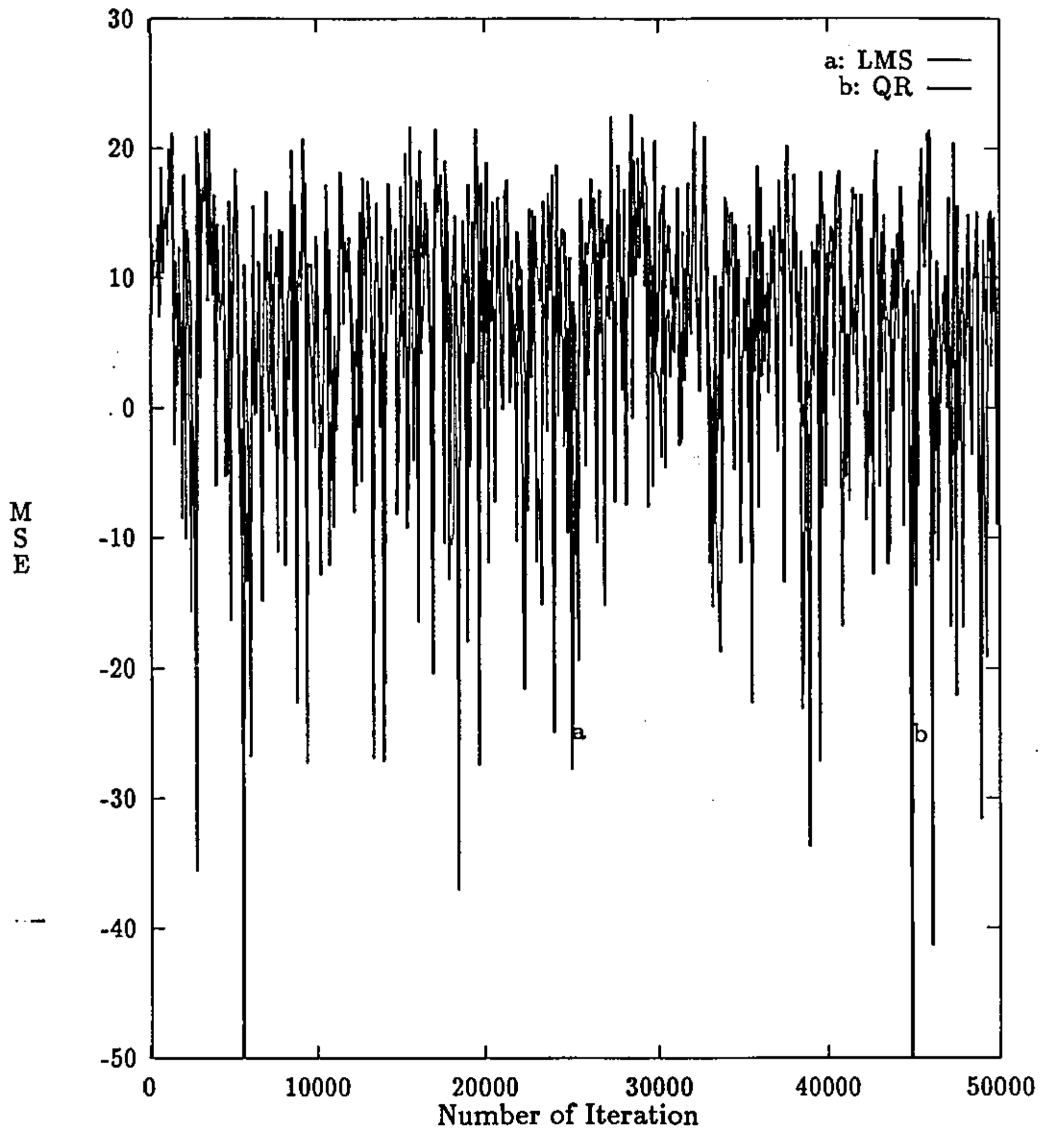

Figure 4.10: Regalia's normalized lattice algorithm. Second order case, MSE (dB) *vs* number of iteration. In the QR algorithm, the κ is set to 0.15, $\lambda = 0.9988$. In the LMS algorithm, μ is set to 0.00075

In the second experiment, the plant transfer function is

$$H_p(z) = \frac{0.5 - 0.4z^{-1} + 0.89z^{-2}}{1.0 - 1.4z^{-1} + 0.98z^{-2}}, \tag{4.37}$$

which has the same poles as transfer function (4.17), and the mean square error is given by Figure 4.10. The mean square error does not converge and the converged coefficients are not obtained.

4.5 Summary

In this chapter, we presented various adaptive IIR filter simulation results using conventional gradient algorithms - Fan's and Regalia's algorithms. Fan's algorithms do not have guaranteed stability. When the poles are close to the unit circle, for the extreme case used in our experiments, the algorithms do not work at all. Since the algorithm model structure remains direct form, it is difficult to solve high order IIR filter problems. Regalia's algorithms have numerical advantages over other conventional gradient algorithms and have the ability to solve high order IIR filter problem, but also may not converge when the poles are extremely close to the unit circle. In the next two chapters, we will introduce an alternative to the IIR filtering problem - applying genetic algorithms to the adaptation of IIR filters.

Chapter 5

Applying the Simple Genetic Algorithm to IIR Filters

5.1 Introduction

Applying genetic algorithms to the adaptive filtering problem was first studied by D.H. Etter, M.J. Hicks and K.H. Cho [3], who used the genetic algorithms to design adaptive IIR filters. R. Nambiar and P. Mars [5] - [9] applied genetic algorithms to system identification problems, in which the plant is modeled by IIR filters. In their studies, the Simple Genetic Algorithm was used. The Simple Genetic Algorithm (SGA) was named by Goldberg [13], and uses the basic genetic operators (for example roulette wheel selection, one-point crossover, and mutation) in genetic algorithm programming. Based on the Simple Genetic Algorithm, researchers have developed many other genetic algorithms, which are more powerful than the SGA.

R. Nambiar and P. Mars have experimented with cascade, parallel and lattice structures for IIR filters, and have shown the positive gain of applying GAs to IIR filtering. In this chapter, we apply the SGA to the adaptation of IIR filters. We also use the cascade, parallel and lattice IIR structures, but here we use the more general transfer functions of these structures, and improved results are obtained. In the next section, system modeling is introduced; section 3 gives a rough idea of how IIR filter coefficients are coded and decoded in genetic algorithms; the computer simulation results are given in section 4; section 5 discusses the simulation results; and a summary appears in section 6.

5.2 Modeling

Throughout this thesis, the system identification configuration is used in the computer simulations (Figure 3.7). The unknown system can be an adaptive FIR filter or IIR filter. In our studies, we chose IIR filter as the unknown system. The defining relationship between the input and output variables for the IIR (order M) filter is given by

$$y(n) = \sum_{k=0}^{M} a_k(n)x(n-k) + \sum_{k=1}^{M} b_k(n)y(n-k), \tag{5.1}$$

or by the transfer function of the IIR filter

$$H(z) = \frac{a_0(n) + a_1(n)z^{-1} + \ldots + a_M(n)z^{-M}}{1.0 + b_1(n)z^{-1} + \ldots + b_M(n)z^{-M}} = \frac{A_m(z)}{B_m(z)}, \tag{5.2}$$

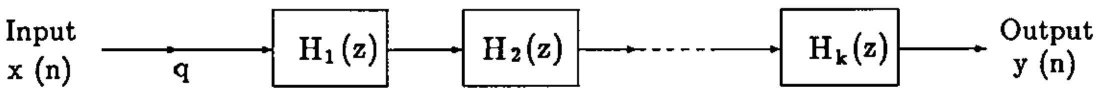

Figure 5.1: The cascade form IIR filter structure.

where $A_m(z)$ and $B_m(z)$ are the z transforms of the output and input signal respectively. In system identification, the unknown system can be identified by direct, cascade, parallel, and lattice form adaptive filter structures. We now introduce these structures.

5.2.1 Direct Form

The direct form is often used in conventional adaptive IIR filter studies (Figure 3.3). Due to the stability problem of the direct form, other IIR filter realizations have been studied.

5.2.2 Cascade Form

The filter (5.1) or (5.2) can be implemented by the cascade form structure, which is given in Figure 5.1. The equivalent cascade-form representation of *H(z)* is

$$H_c(z) = q\prod_{k=1}^{W} \frac{1.0 - a_{1k}(n)z^{-1} - a_{2k}(n)z^{-2}}{1.0 - b_{1k}(n)z^{-1} - b_{2k}(n)z^{-2}}, \tag{5.3}$$

where W = (M+1)/2, if M is odd, or W = M/2, if M is even, and p is a constant.

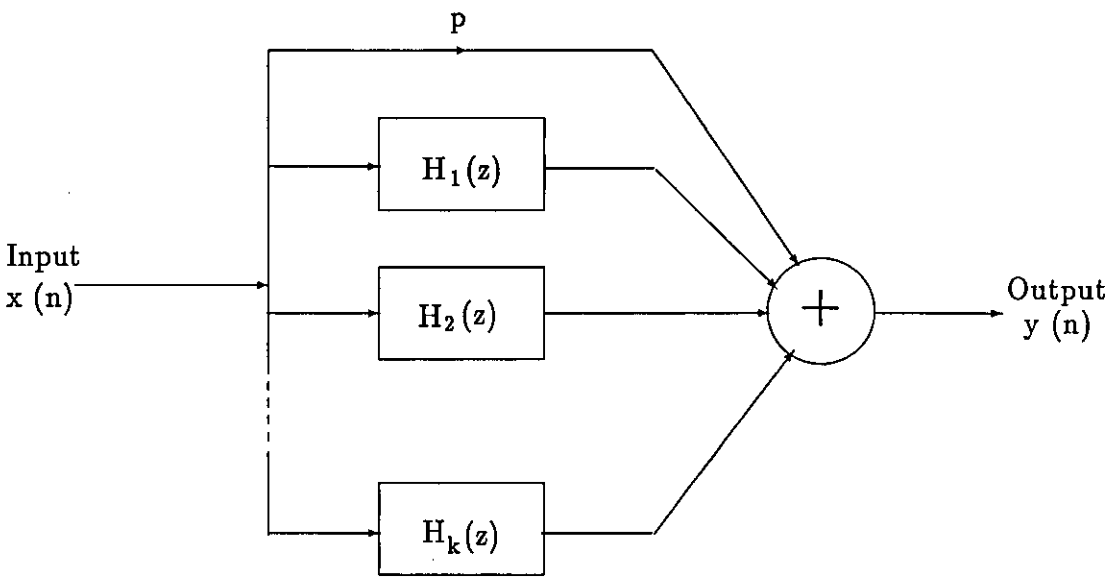

Figure 5.2: The parallel form IIR filter structure

5.2.3 Parallel Form

The filter (5.1) or (5.2) can also be implemented by the parallel form, which is given in Figure 5.2. The equivalent parallel-form representation of H(z) is

$$H_p(z) = p + \sum_{k=1}^{W} \frac{a_{0k}(n) - a_{1k}(n)z^{-1}}{1.0 - b_{1k}(n)z^{-1} - b_{2k}(n)z^{-2}}, \tag{5.4}$$

where W = (M+1)/2, if M is odd, or W = M/2 if M is even, and p is a constant.

For the cascade and parallel structures, the stability of the filters during adaptation is guaranteed by constraining the filter coefficients b_{1k}(n) and b_{2k}(n) to lie within the stability triangle.

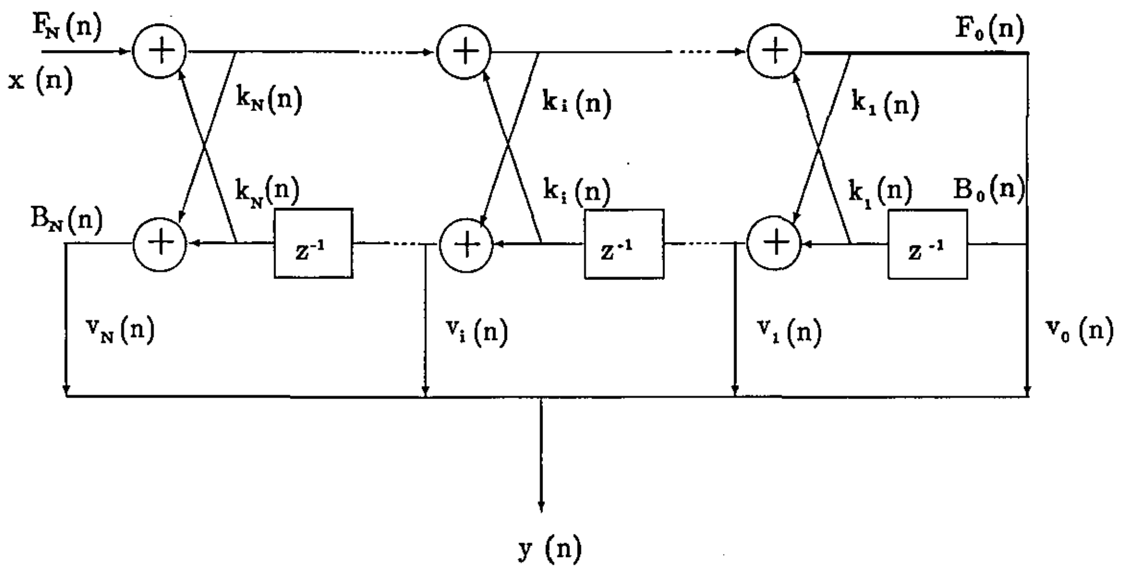

Figure 5.3: Lattice form IIR filter structure

5.2.4 Lattice Form

The filter (5.1) or (5.2) can also be implemented in the form of a lattice with different weights v_i(n) and k_i(n) (Figure 5.3). The lattice form is stable if the lattice coefficients k_i(n) are all less than 1.

The input-output of the lattice filter at time n can be expressed as

$$y(n) = \sum_{i=0}^{M} v_i(n) B_i(n), \tag{5.5}$$

where

$$B_i(n) = B_{i-1}(n) + k_i(n) F_{i-1}(n); \qquad i = M, ..., 1 \tag{5.6}$$

$$F_i(n) = F_{i+1}(n) - k_i(n) B_i(n-1); \qquad i = M-1, ..., 0 \tag{5.7}$$

$$F_M(n) = x(n) \tag{5.8}$$

and x(n) is the input signal, and

$$B_0(n) \quad = \quad F_0(n). \tag{5.9}$$

We will use the above four models to identify the high order unknown system in our simulation experiments.

5.2.5 Direct Form to Lattice Form Coefficient Conversion

Given the coefficients $a_i(n)$ and $b_i(n)$ of a direct form system, one can calculate the corresponding lattice form coefficients $k_i(n)$ and $v_i(n)$ [65]. The lattice coefficients are recursively obtained starting from $A_M(z)$ and $B_M(z)$ (5.2), as follows

$$zC_m(z) = B_m(1/z)z^{-m} \tag{5.10}$$

$$k_{m-1} = b_m \tag{5.11}$$

$$B_{m-1}(z) = \frac{B_m(z) - k_{m-1}zC_m(z)}{1 - k_{m-1}^2} \tag{5.12}$$

$$v_m = a_m \tag{5.13}$$

$$A_{m-1}(z) = A_m(z) - zC_m(z)v_m \tag{5.14}$$

for m = *M, M-1, ..., 1* with $v_0 = a_0$. We will use these conversions later in our computer simulations.

5.3 Parameter Coding and Decoding

Applying Genetic Algorithms to the adaptive filtering problem, one has to code the adaptive filter coefficients into a form which genetic algorithms can deal with, for example, binary strings. In Figure 5.4, a set of second order lattice filter coefficients are coded into a binary string. Each lattice filter coefficient has length 4 (this length is chosen as an example, rather than as a practical length). When we decode this binary coefficient string, first we decode the binary string to several decimal values, for instance, k_0, 0101 to 5; k_1, 0100 to 4, etc, then we map these decimal values to a certain range (min, max) to obtain the real coefficient values. For binary string, the widely used mapping formula is given by:

$$v_c = min + v_d * \frac{max - min}{X} \tag{5.15}$$

where v_c is the coefficient value, v_d is the decoded parameter value, (min, max) is the mapping range, and X is the value when every bit is equal to 1 in the parameter string. For example, the range for k parameters is (-1, 1), then in Figure 5.4, k_1 = -1 + 4*(1+1)/(2^4-1) = -0.466667.

This range (min, max) is very important in our simulations, as different min and max values can produce different simulation results. For the lattice, k_i will lie in the range (-1.0, 1.0) according to the stability condition; the range for v_i will be chosen from experiment.

0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1
v_2	v_1	v_0	k_1	k_0

Figure 5.4: Example of a second order lattice parameter coding, individual string length L = 20, parameter length is 4.

5.4 Computer Simulations

All the simulations in this chapter run 20 independent experiments of the SGA, which uses stochastic remainder selection, standard crossover and mutation. The fitness values are the inverse of the averaged squared error over the 20 experiments

$$fitness = \frac{1.0}{e^2(n)}, \tag{5.16}$$

so we maximize the fitness values in order to minimize the mean square error. Sigma scaling has been used in all the simulations to regulate the individuals in the population, so as to avoid some extraordinary poor individuals that take over a significant proportion of the population, a leading cause of premature convergence. The formula for sigma scaling [13] is

$$f_s = f_i - (\overline{f} - 2.0 * \sigma), \tag{5.17}$$

where the σ is the standard deviation of the population fitness and

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(f_i - \overline{f})^2}{N}}. \tag{5.18}$$

f_s is the scaled fitness (if $f_s < 0$, we set $f_s = 0$), f_i is the fitnesses of individual, and $\overline{f}$ is the population average fitness. The high order unknown system transfer functions are chosen from [67].

5.4.1 IIR filters with Bi-modal Error Surface

We use the example [37] which we have previously used in the last chapter. The unknown system

$$H(z) = \frac{0.05 - 0.4z^{-1}}{1.0 - 1.1314z^{-1} + 0.25z^{-2}} \tag{5.19}$$

is identified by a first order adaptive system

$$H(z) = \frac{a(n)}{1.0 - b(n)z^{-1}}. \tag{5.20}$$

According to the paper [37], this example's mean-square-error (MSE) surface is bi-modal. The global minimum $\xi = 0.277$, the local minimum $\xi = 0.976$, and the corresponding coefficient values are (a, b) = (-0.311, 0.906) and (a, b) = (0.114, -0.519) respectively.

In this experiment, where each coefficient is represented by a 10-bit binary string, the crossover probability $p_c = 0.85$, the mutation probability $p_m = 0.003$, population size is 50, we obtained (a, b) = (-0.314, 0.906). The plot of mean square error (in dB) against generations is given in Figure 5.5.

This example was examined using the genetic algorithms in [9] and [68] as well. We reexamine this case in this and the next chapter to show that genetic algorithms have the ability to tackle multi-modal error surface IIR filter problems, and present the performance improvement we have obtained.

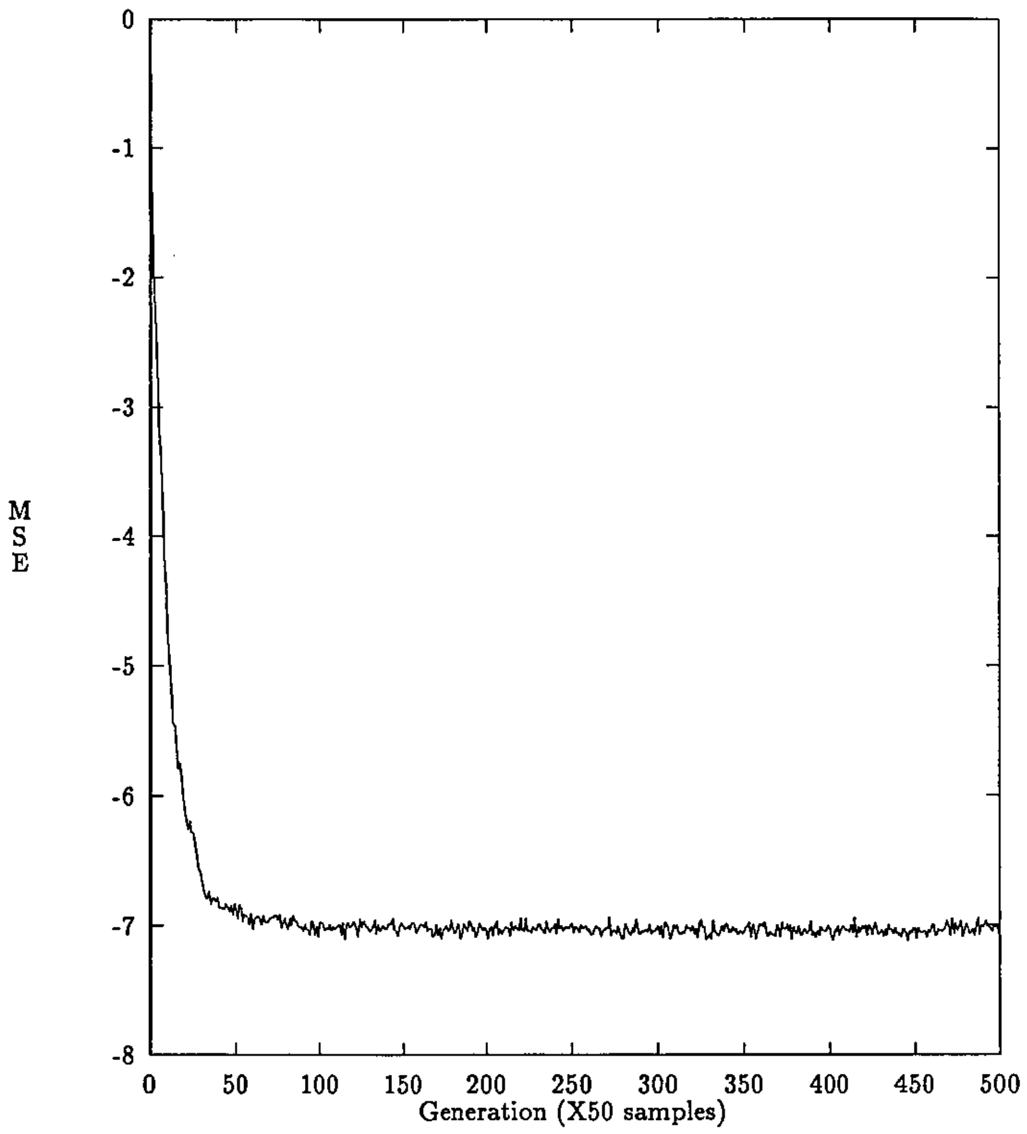

Figure 5.5: Bimodal example, mean squared error (MSE in dB) *vs* generations (averaging 20 independent run). The population size N = 50, string length L = 20, crossover probability $p_c = 0.85$, mutation probability $p_m = 0.003$. One point crossover has been used in the simulation.

5.4.2 IIR Filters with Poles Close to the Unit Circle

The second order filter

$$H(z) = \frac{0.5}{1.0 - 1.4z^{-1} + 0.98z^{-2}}, \tag{5.21}$$

is identified by the filter

$$H(z) = \frac{a_0(n)}{1.0 - b_1(n)z^{-1} - b_2(n)z^{-2}} \tag{5.22}$$

through the SGA. This system has three coefficients to be identified, and has poles at 0.7±j0.7 (modulus = 0.99), which are very close to the unit circle. Many gradient algorithms failed to identify this special case, for example, the algorithms in [10], [39] (see Chapter 4). The genetic algorithm gives the results illustrated in Figure 5.6, which shows the advantage of GAs over gradient algorithms when the poles are extremely close to the unit circle. The unknown system is a second order system, so we chose direct form adaptive IIR filters for this experiment. The direct form gives the coefficients $a_0(n)$ = 0.501466, $b_1(n)$ = 1.399673, $b_2(n)$ = -0.98045.

Another example is also a second order filter

$$H(z) = \frac{0.5 - 0.4z^{-1} + 0.89z^{-2}}{1.0 - 1.4z^{-1} + 0.98z^{-2}}, \tag{5.23}$$

and is identified by the direct adaptive system

$$H(z) = \frac{a_0(n) - a_1(n)z^{-1} - a_2(n)z^{-2}}{1.0 - b_1(n)z^{-1} - b_2(n)z^{-2}}, \tag{5.24}$$

which has five coefficients to be identified, and we use direct and lattice form structures in our computer simulations. The mean squared error (MSE in dB)

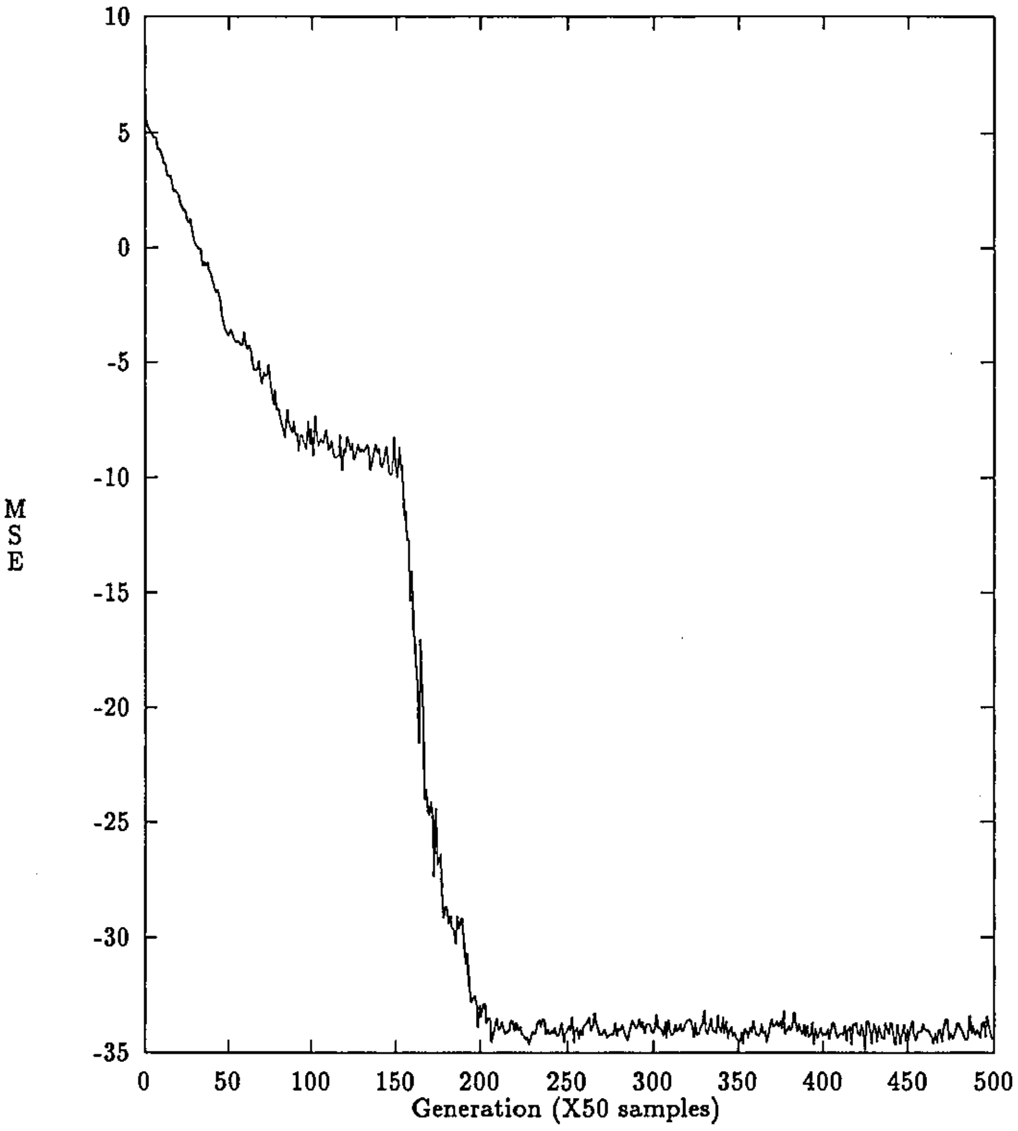

Figure 5.6: Poles close to the unit circle (three coefficients), MSE (dB) *vs* generations. The population size N = 80, string length L = 30, probability of crossover $p_c = 85$, probability of mutation $p_m = 0.0075$. One-point crossover has been used.

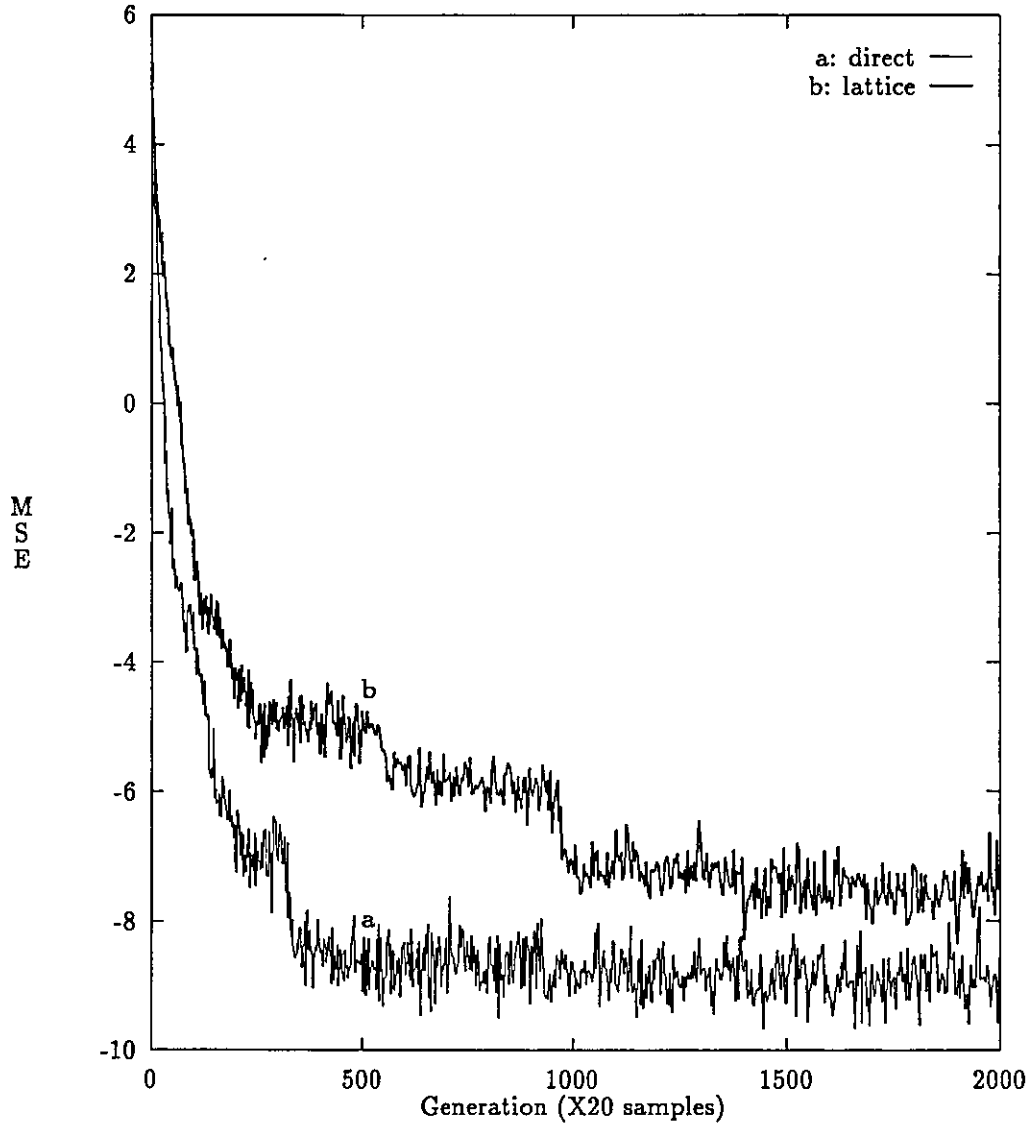

Figure 5.7: Poles close to the unit circle (five coefficients), MSE (dB) *vs* generations. The population size N = 50, string length L = 50 (10-bits for each coefficient), the probability of crossover $p_c = 1.0$, the probability of mutation $p_m = 0.003$. One-point crossover has been used. a) direct structure; b) lattice structure.

Table 5.1: The coefficients of the direct structure using the SGA.

coefficients	plant (direct)	adaptive filter
$b_1(n)$	1.40	1.399637
$b_2(n)$	-0.98	-0.980450
$a_0(n)$	0.50	0.519062
$a_1(n)$	0.40	0.438905
$a_2(n)$	-0.89	-0.917889

Table 5.2: The coefficients of the lattice structure using the SGA.

coefficients	plant (lattice)	adaptive filter
$k_0(n)$	-0.707071	-0.708700
$k_1(n)$	0.980000	0.972630
$v_0(n)$	0.225982	0.251222
$v_1(n)$	0.846000	0.876833
$v_2(n)$	0.890000	0.896383

is given in Figure 5.7. The coefficients results (best in the population) are given in Tables 5.1 and 5.2. The corresponding lattice coefficients of equation (5.23) can be calculated according to the formulation given in section 5.2.5.

This experiment shows that both direct and lattice structures work with the SGA. Due to the difficulty of judging high order direct structure coefficients, so as guarantee the stability, identifying an IIR filter with order greater than two normally does not employ the direct structure in the simulation. From here we use cascade, parallel and lattice structures in our computer simulation.

5.4.3 High Order IIR Filters

An order three filter

$$H(z) = \frac{0.0154 + 0.0462z^{-1} + 0.0462z^{-2} + 0.0154z^{-3}}{1.0000 - 1.9900z^{-1} + 1.5720z^{-2} - 0.4583z^{-3}} \tag{5.25}$$

is identified by lattice, cascade and parallel adaptive structures. The cascade and parallel forms are constructed using first or second order filters (in direct form, see section 5.2). The mean square errors are shown in Figure 5.8. We also find that only a few of the coefficients groups in the 20 runs are quite close to the ideal coefficients combination. The best coefficients group we found in our simulation is shown in Table 5.3.

Another example of order five

$$H(z) = \frac{0.0073 - 0.0184z^{-1} + 0.0115z^{-2} + 0.0115z^{-3} - 0.0184z^{-4} + 0.0073z^{-5}}{1.0000 - 4.5064z^{-1} + 8.2615z^{-2} - 7.6908z^{-3} + 3.6326z^{-4} - 0.6961z^{-5}} \tag{5.26}$$

is identified by lattice, cascade and parallel structure adaptive IIR filters. The mean square error is given in Figure 5.9. It gives the similar simulation results to

Table 5.3: The coefficients of the lattice structure using the SGA.

coefficients	plant (lattice)	adaptive filter
$k_0(n)$	-0.875587	-0.851417
$k_1(n)$	0.835463	0.734115
$k_2(n)$	-0.458300	-0.485826
$v_0(n)$	0.085646	0.093646
$v_1(n)$	0.145491	0.186315
$v_2(n)$	0.076846	0.150147
$v_3(n)$	0.015400	0.056696

the previous example, that is the lattice gives the best performance, and cascade is better than parallel. The coefficient result remains unidentified.

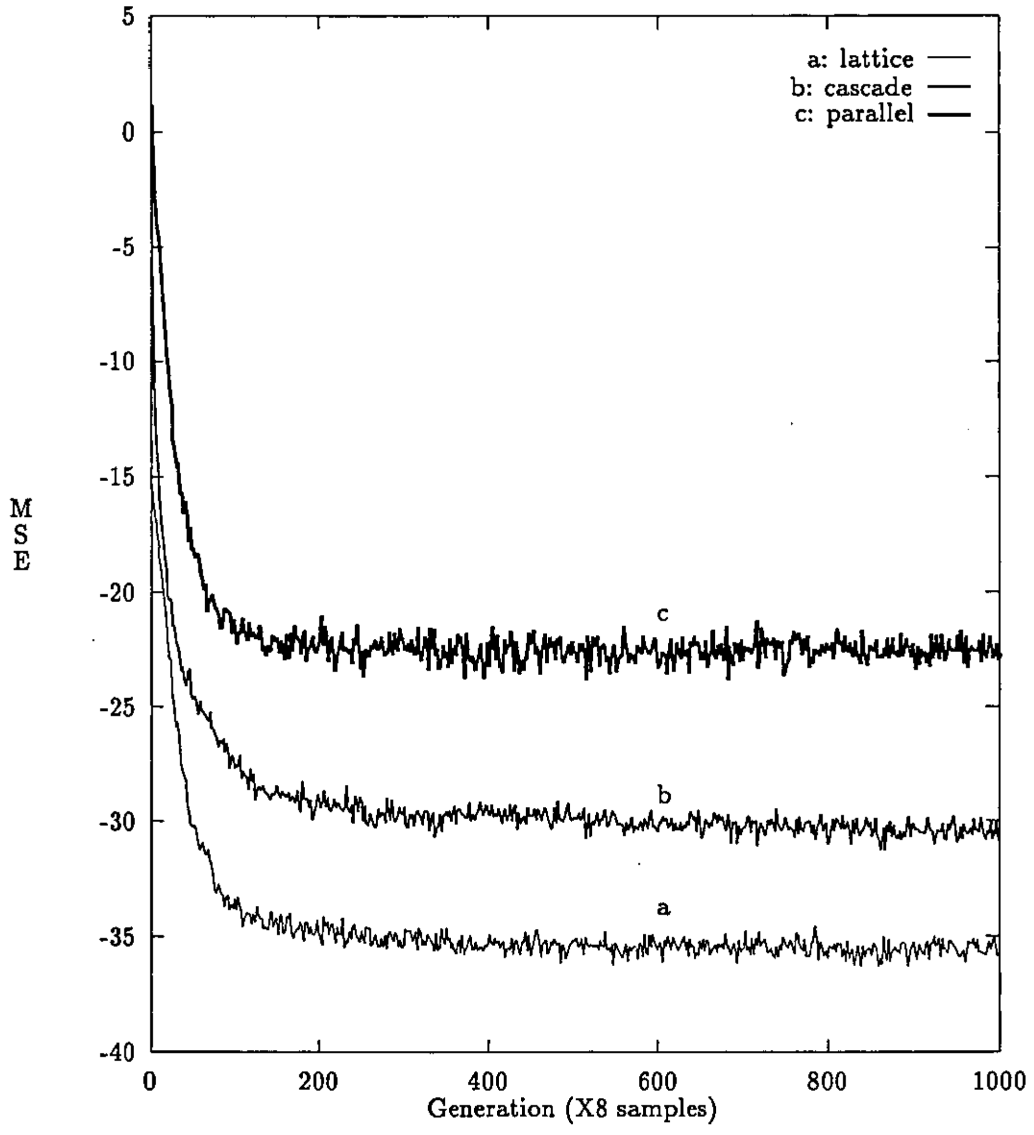

Figure 5.8: The third order filter, MSE (dB) *vs* generations. The population size N = 80, the string length L = 70 (10-bits for each coefficient), the probability of one-point crossover $p_c = 1.0$, the probability of mutation $p_m = 0.003$. a) lattice structure; b) cascade structure; c) parallel structure.

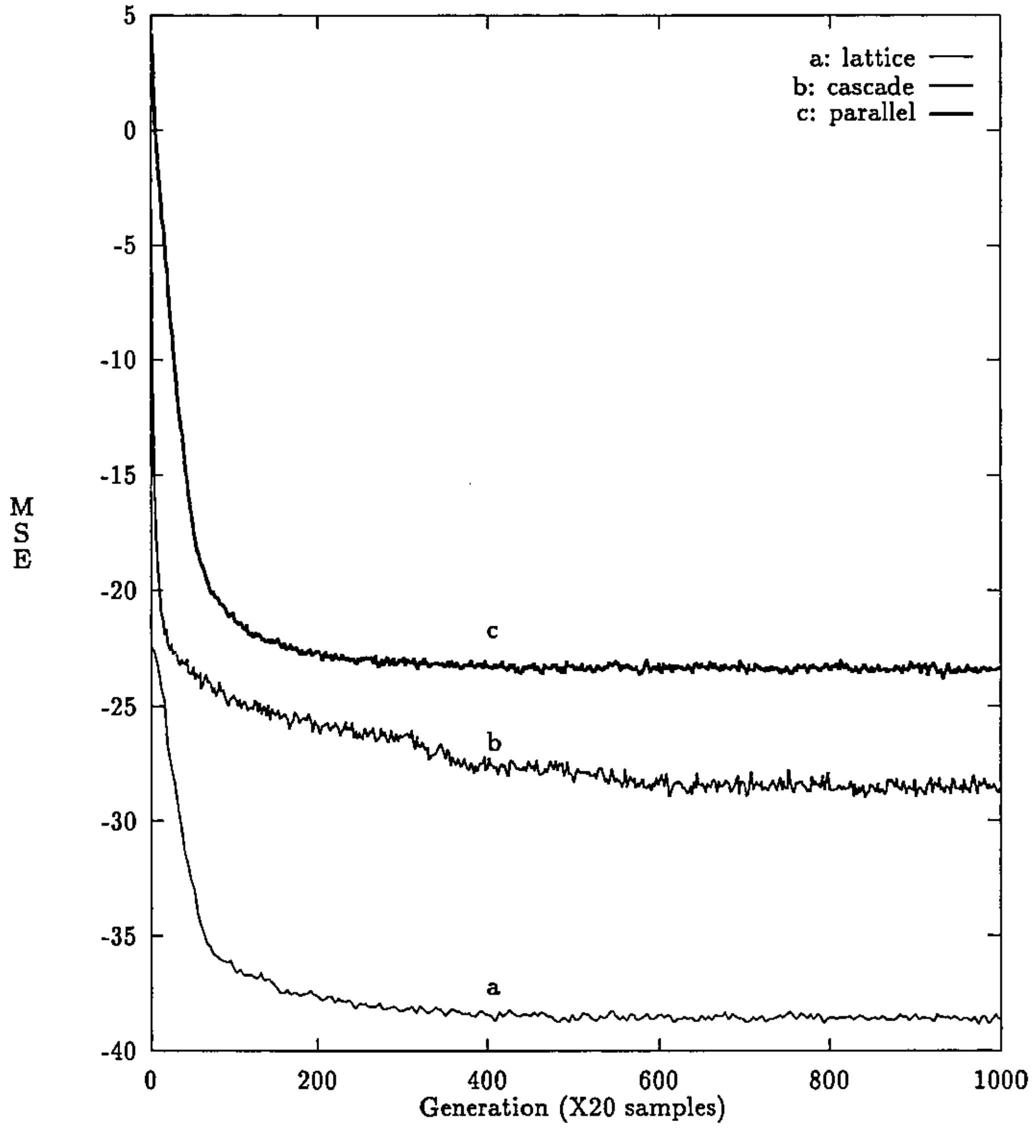

Figure 5.9: The fifth order filter, MSE (in dB) *vs* generations. The population size N = 400, the string length L = 88 (8-bits for each coefficient), the probability of one-point crossover $p_c = 1.0$, the probability of mutation $p_m = 0.003$. a) lattice structure; b) cascade structure; c) parallel structure.

5.4.4 SGA Parameters

All parameters in the previous experiments were chosen on the basis of repeated computer simulations. We give several examples here.

The first example is differences in population size. In Figure 5.10, of the various population sizes used in our simulations, a population size of 400 gives the best MSE performance. In this experiment, the fifth order filter (5.26) and lattice structure are used.

The second example is changes in the string lengths. The different string length for each coefficient used in our simulations are 8-bits, 10-bits, and 15-bits, and the MSE performances are given in Figure 5.11. The 8-bits and the 15-bits are almost the same and give better performance. The filter (5.26) and lattice structure are used in our simulations.

The third example is concerned with the number of crossover points. We experiment with various numbers of crossover points and with uniform crossover, and the performance results are given in Figure 5.12. The filter (5.26) and lattice structure are used in our simulation, and show the choice of one-point crossover is the best.

We also tried many different crossover rates and mutation rates, among which, $p_c = 1.0$, and $p_m = 0.003$ are the best in our later simulations.

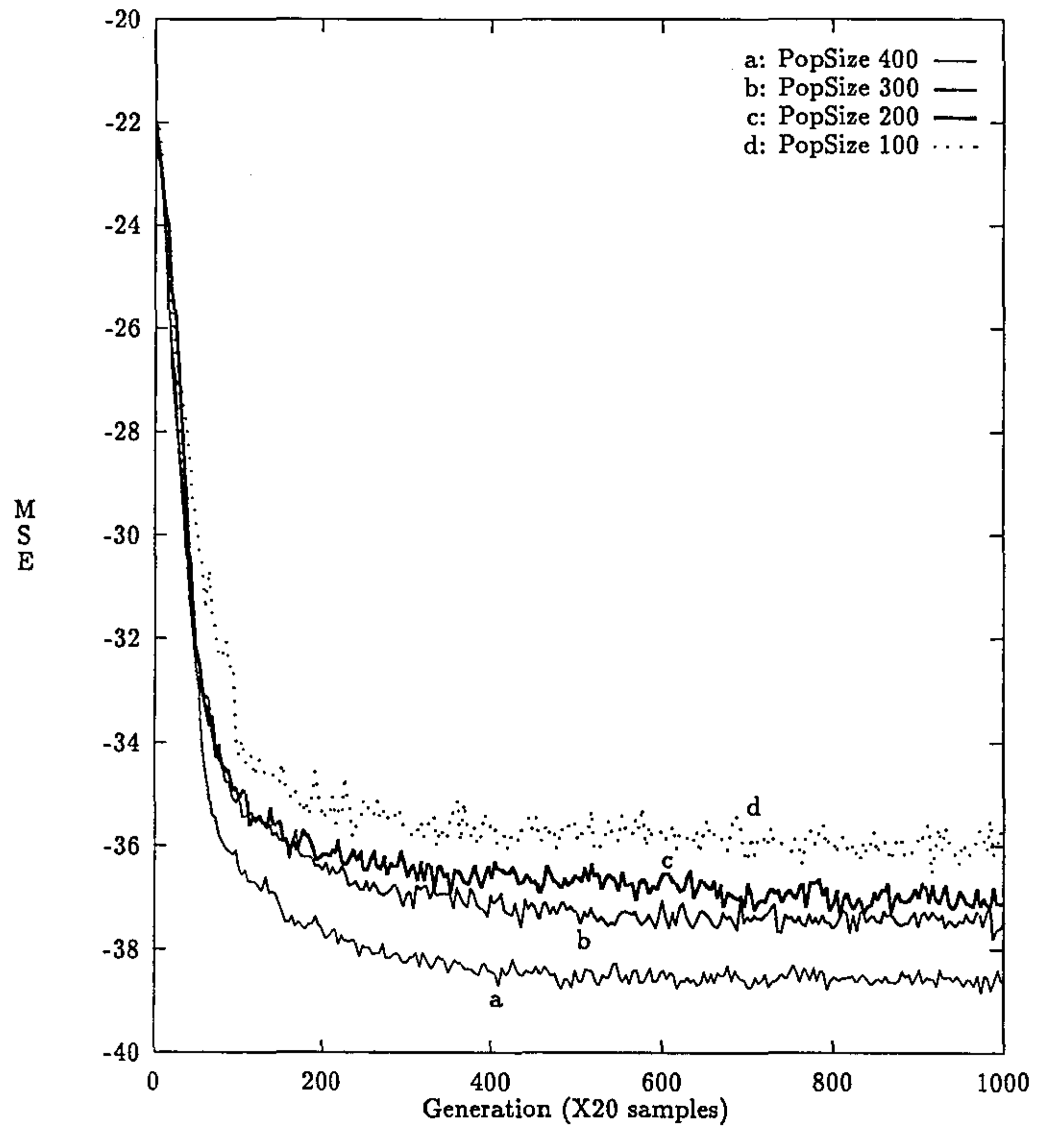

Figure 5.10: The population comparison, MSE (in dB) *vs* generations (averaging 20 independent runs). The string length L = 88 (8-bits for each coefficient), p_c = 1.0, the mutation rate p_m = 0.003, one-point crossover. a) N = 400; b) N = 300; c) N = 200; d) N = 100.

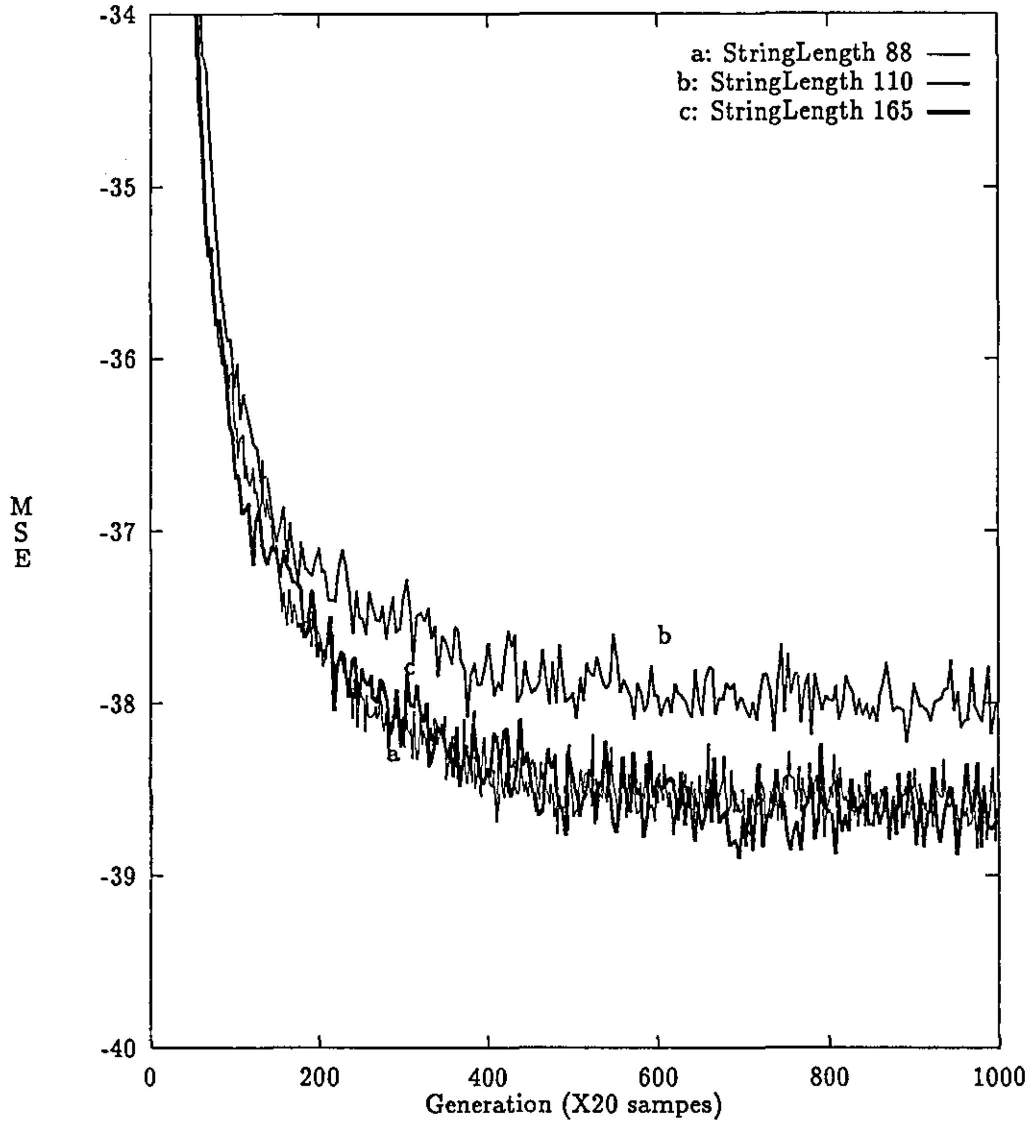

Figure 5.11: The string length comparison, MSE (in dB) *vs* generations (averaging 20 independent runs). The population size N = 400, the crossover rate $p_c = 1.0$, the mutation rate $p_m = 0.003$, one-point crossover. a) string length 88; b) string length 165; and c) string length 110.

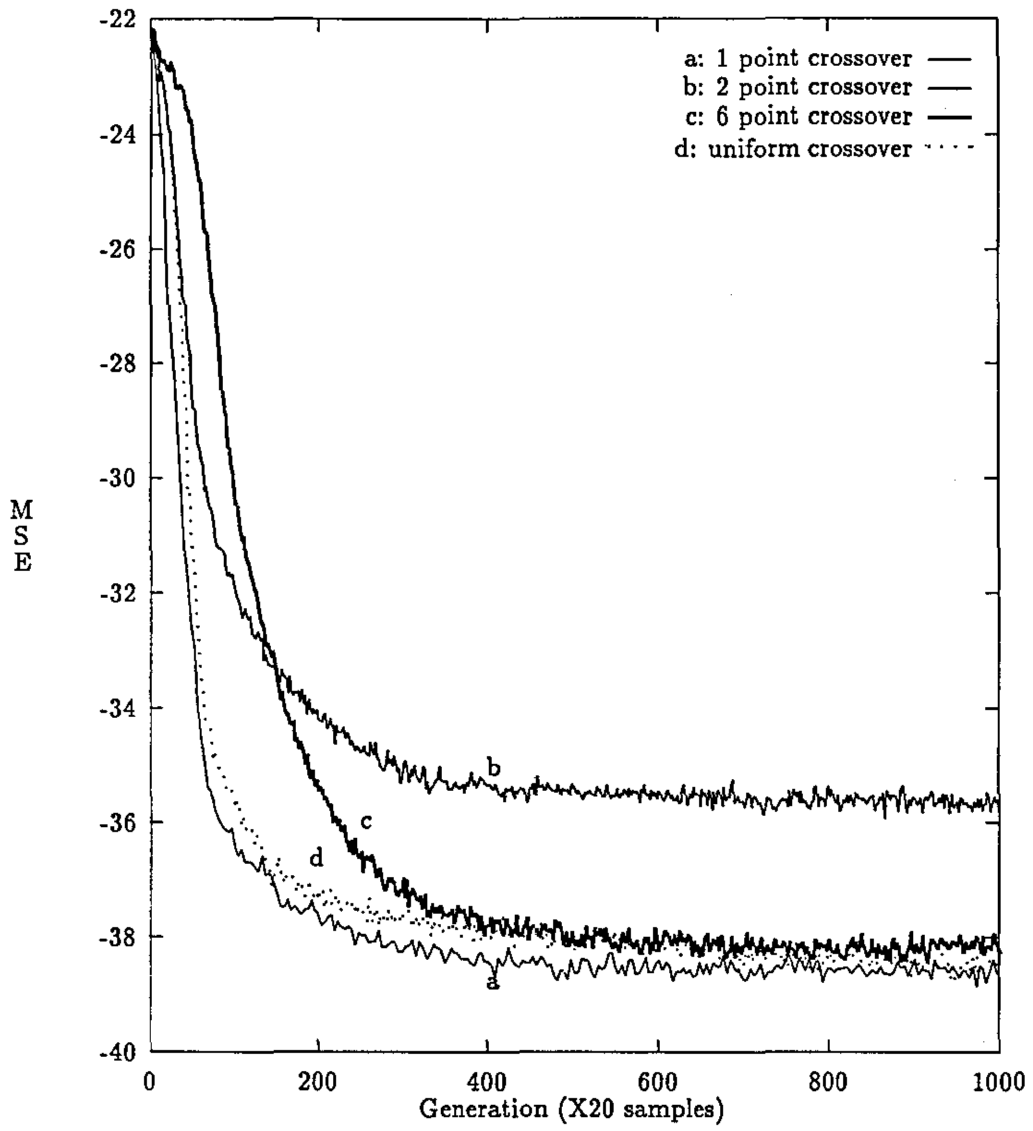

Figure 5.12: The different crossover scheme, MSE (in dB) *vs* generations (averaging 20 independent runs). The population size N = 400, the crossover rate p_c = 1.0, the mutation rate p_m = 0.003, string length L = 88. a) 1-point crossover ; b: 2-point crossover; c) 6-point crossover; d) uniform crossover.

5.5 Discussion

5.5.1 Simulation Performance

A convergence proof for the standard genetic algorithm has been given in [69]. In our simulations, it has been shown that the SGA is able to solve low order IIR filter problems, even with poles close to the unit circle or multi-modal. However when the order becomes high, the coefficient results are not very satisfactory. The reason is that the SGA faces premature convergence caused by the super-individual in the population, dragging the mean square error to converge to some level in the early stage of training.

For the case when the poles are close to the unit circle, the SGA shows an improved solution over many conventional algorithms. Because applying genetic algorithms to the adaptation of IIR filters does not have a stability problem, the first or the second order denominator coefficients are either bounded within (-1.0, 1.0) or within the stability triangle.

5.5.2 Coefficient Decoding Range

We have mentioned that the coefficient decoding range is very important in our simulations. Here we give a detailed discussion on how we deal with this problem in our simulations.

In the direct form structure, we restrict the denominator coefficients to lie within the stability triangle. For the example of the second order filter (5.24), we let $-1 < b_2(n) < 1$ and $-1 - b_2(n) < b_1(n) < 1 + b_2(n)$. The numerator coefficient range can be decided according to the plant numerator, for example, if there is one numerator coefficient which is 1.2, we can set the range (1.0, 1.5) in our simulation, which covers the numerator coefficient 1.2. If there are five coefficients, we should chose a range which covers these five coefficients.

In the lattice form structure, we restrict the k coefficients to less than 1 for stability reasons. The v coefficient decoding range is chosen according to the plant v coefficients. For the example of the plant (5.23), the v coefficients vary from -0.707071 to 0.89, and choosing a decoding range (-1, 1) will cover all these five coefficients in our simulation. We can also make this range smaller, depending on whether we have obtained good results, but normally the right range is obtained by conducting many simulations.

5.6 Summary

In this chapter, we applied the SGA to the adaptation of IIR filter problems. We experimented with direct (low order filter), cascade and parallel (high order filter) IIR filter structures in our computer simulations. In low order cases, the direct form's performance is better than the lattice; in high order cases, the lattice structure gives the best results, because of the error propagation of cascade and parallel structures. The SGA can identify low order IIR filter coefficients, but when the order increases, the coefficient results become poor, in the sense that the MSE which is obtained by using SGA might converge to the non-global value. The reason for this is that the SGA faces the premature convergence problem

[19]. In all the simulations, we performed a very large number of experiments to choose the best SGA parameters. We still need to improve the high order IIR filter performance, which should improve the genetic algorithm itself. In the next chapter, we will study another genetic algorithm which can improve the overall performance.

Chapter 6

Applying Genitor to the Adaptation of IIR Filters

6.1 Introduction

In the previous chapter, we applied the Simple Genetic Algorithm to the adaptation of IIR filters. We used direct, cascade and parallel structure adaptive systems to identify the unknown system. The results showed that the lattice structure gave the best performance, and that the cascade structure's performance was better than the parallel's. Even the lattice structures' results are not the ideal solution for high order IIR filters, so we need to look for better genetic algorithms to improve these structure's performance.

In this chapter, we use the same models and coding and decoding methods as we used in the previous chapter, applying the steady state genetic algorithm to the adaptation of IIR filters. The results of the steady state genetic algorithm have improved the SGA's performance. The steady state genetic algorithm we use is Whitley's Genitor [21].

In the next section, we give an introduction to the steady state genetic algorithm. In section 3, computer simulations are conducted through various system models and genetic algorithm parameters. In section 4, a number of discussions based on the simulations are given. Finally in section 5, a summary is given.

6.2 Genitor

Genitor is Whitley's steady state genetic algorithm. It is an acronym for GENetic ImplemenTOR, a genetic search algorithm that differs in three ways from the standard genetic algorithms. First, reproduction produces one offspring at a time. Two parents are selected for reproduction and produce an offspring that is immediately placed back into the population. The second major difference is in how that offspring is placed back into the population. Offspring do not replace parents, but rather the least fit (or some relative less fit) member of the population. In Genitor, the worst individual in the population is replaced. The third difference between Genitor and most other forms of genetic algorithms is that fitness is assigned according to rank rather than by fitness proportionate reproduction. Ranking helps to maintain a more constant selective pressure over the course of search [16], [22]. Goldberg [19] names the selection in Genitor *steady state selection*. We now introduce Genitor in detail.

6.2.1 Reproduction - Steady State Selection

In Genitor, the reproduction is implemented through steady state selection [71]. It begins with sorting the whole population individuals, from best to worst, then uses ranking selection which starts with assigning the number of copies that each individual should receive according to a non-increasing assignment function, and then performs proportionate selection according to that assignment. The assignment function in Genitor is a linear function (probability distribution of x, linear)

$$\beta(x) = b - 2(b-1)x, \qquad x \in [0,1], \tag{6.1}$$

where b is the bias which is defined as a number that specifies the amount of preference to be given to the superior individuals in a genetic population, that is

$$bias = \frac{p_{best}}{p_{mean}}, \tag{6.2}$$

where p_{best} and p_{mean} are the probabilities of the best and mean individuals receiving copies in the next generation. For example, a bias of 2.0 indicates that the best individual has twice the chance of being chosen as the mean individual.

Now we can perform proportionate selection according to this assignment function. The selection formula used in Genitor is

$$index = \frac{N(b - \sqrt{b^2 - 4(b-1)drand48()})}{2(b-1)}, \tag{6.3}$$

where *index* is the index of strings being selected in the population (an integer between 0 and population size *N*), *drand48()* is random number generator which generates random numbers between 0 and 1, and b is the bias. We use this selection scheme to select two parent strings for mating.

original strings	reduced strings
0 0 0 1 1 1 1 0 1 1 0 1 0 0 1 1	- - - - 1 1 - - - 1 - - - - - 1
0 0 0 1 0 0 1 0 1 0 0 1 0 0 1 0	- - - - 0 0 - - - 0 - - - - - 0

Figure 6.1: The reduced surrogates

6.2.2 Recombination

Recombination is implemented through the crossover operator. The crossover used in Genitor is a two point *reduced surrogate* crossover [70], [71]. It is different from that in standard genetic algorithms, in which crossover only occurs in the positions where the parent strings differ. So the first job the reduced surrogate crossover has to do is to identify all the different positions in the two parent strings. Now we consider the two strings and a 'reduced' version of the same strings in Figure 6.1, where the bits the strings share in common have been removed. In reality, only the different bits in the two parents make sense to the recombination, in that the crossover does not change the common bits. Booker refers to strings such as {- - - - 1 1 - - - 1 - - - - - 1} and {- - - - 0 0 - - - 0 - - - - - 0} as the *reduced surrogates* of the original parent chromosomes [70]. In Figure 6.1, the reduced surrogate crossover can happen only between the fifth and the last bit position. The reduced surrogate crossover's main advantage is that the parents are not duplicated in the offsprings. Thus, new sample points in hyperspace are generated. We use this reduced surrogate crossover on the two parents to produce two offsprings. See [71] for details.

6.2.3 Mutation

The mutation operator used in Genitor is the *adaptive mutation* operator [71]. It differs from standard mutation in the way that the mutation is determined to an appropriate level according to the hamming distance between its two parents. The hamming distance is defined as the number of different bits of two strings, with the smaller the difference, the higher the mutation rate. In Genitor, the following formula is used to modify the mutation rate:

$$p_{adaptivemutation} = \frac{p_m}{(h_d/L)100} \tag{6.4}$$

where $p_{adaptivemutation}$ is the adaptive mutation rate, p_m is the mutation rate, h_d is the hamming distance and L is the string length. For example, for the two parents strings

string 1: 1 0 1 0 1 0 0 1

string 2: 1 1 1 1 1 1 1 1,

if the mutation rate is 0.005, the adaptive mutation rate would be

0.005/((4/8)*100) = 0.0001.

This reduces the unnecessary mutation. For example, mutating the allele on locus 4 in string 1 will produce a string which is exactly same as string 2.

After steady state reproduction, reduced surrogate recombination and adaptive mutation, one of the child strings is chosen to be inserted back into the population. This child string is calculated for fitness value (to see its performance) and replaces the least fit string in the population. In this way, one Genitor's generation cycle is completed.

6.3 Computer Simulations

The simulations in this chapter are all conducted through 20 independent runs. The mean square error (MSE) performances are obtained by averaging those 20 independent results. Genitor is a minimum optimization which looks for the smallest (best) value, so we use the squared error as a fitness value directly. The population sizes are chosen relatively large (most of them are 200), because normally Genitor requires large population sizes or multiple populations to combat the premature convergence problem [19]. We run Genitor for 200,000 generations, which seems large, but if compared to the SGA in generation terms, it is almost the same. In the SGA, in each generation, the fitness value for every individual in the whole population has to be calculated, but in Genitor, in each generation, only one individual fitness value is calculated. For example, if every member of the population is calculated for new fitness values in Genitor, it needs 200 (population size) generations. So 200,000 generations in Genitor are comparable with 200,000/200 = 1,000 generations in the SGA.

6.3.1 IIR Filters with Bi-modal Error Surface

We use the same bi-modal example as in the last chapter which uses an order one adaptive system to identify the order two system. The transfer functions are given by equation 5.19 and 5.20 respectively. Genitor parameters are: population size N = 50, string length L = 20, bias = 1.6955, mutation rate p_m = 0.0555. The order one system's coefficients we obtained are (a, b) = (-0.306, 0.912). The mean square error is shown in Figure 6.2.

From this example, we can say that Genitor has the same ability to solve the bi-modal IIR filter problem as the SGA.

6.3.2 IIR Filters with Poles Close to the Unit Circle

The second order filter

$$H(z) = \frac{0.5}{1.0 - 1.4z^{-1} + 0.98z^{-2}} \tag{6.5}$$

is identified by the adaptive filter

$$H(z) = \frac{a_0(n)}{1.0 - b_1(n)z^{-1} - b_2(n)z^{-2}}. \tag{6.6}$$

Using Genitor, the mean square error plot against generations is given in Figure 6.3. The adaptive coefficients are $a_0(n)$ = 0.500978, $b_1(n)$ = 1.399673, $b_2(n)$ = -0.980450. The mean square error performance is an improvement over the SGA, and the coefficients have better values than the SGA's.

In another example, lattice and direct form adaptive structures have been used to identify the unknown system

$$H(z) = \frac{0.5 - 0.4z^{-1} + 0.89z^{-2}}{1.0 - 1.4z^{-1} + 0.98z^{-2}}. \tag{6.7}$$

The direct form transfer function is

$$H(z) = \frac{a_0(n) - a_1(n)z^{-1} - a_2(n)z^{-2}}{1.0 - b_1(n)z^{-1} - b_2(n)z^{-2}}, \tag{6.8}$$

which has five coefficients to be identified. The squared error (in dB) is given in Figure 6.4, and the coefficients obtained from the first run of the 20 from the simulation are shown in Tables 6.1 (direct) and 6.2 (lattice).

This experiment gives the same results as in the previous chapter for the direct and lattice form structures. The results show that using Genitor on an IIR filter problem gives better results than using the SGA. However, as the number of coefficients increases, it become increasingly difficult for Genitor to identify the coefficients. For high order IIR filters, lattice, cascade and parallel structures should be used, rather than the direct form.

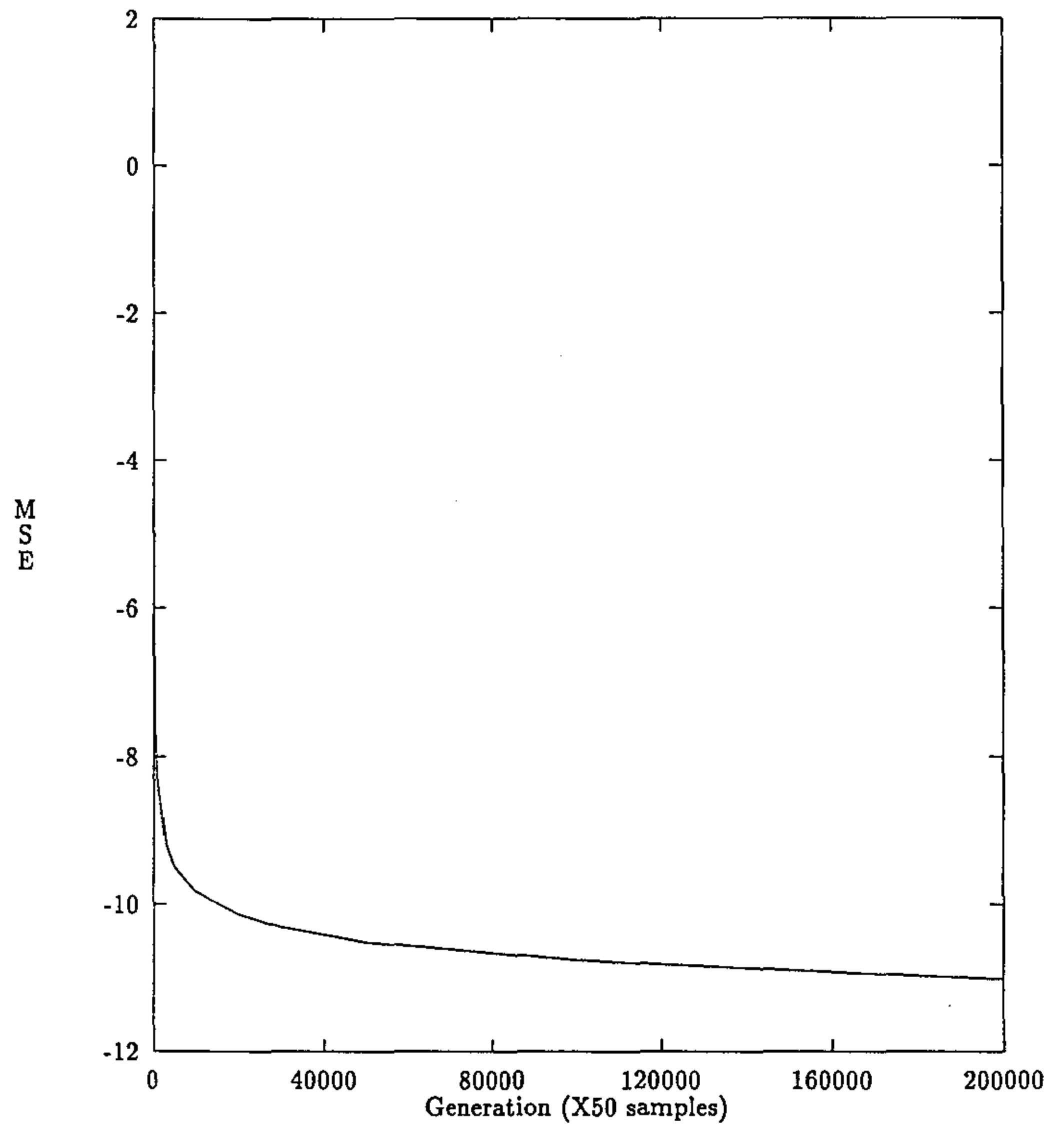

Figure 6.2: Bimodal example, MSE (in dB) *vs* generations. The population size N = 50, string length L = 20 (10-bits for each coefficient), the bias is 1.6955, the mutation rate p_m = 0.0555, the random seed is 12345678.

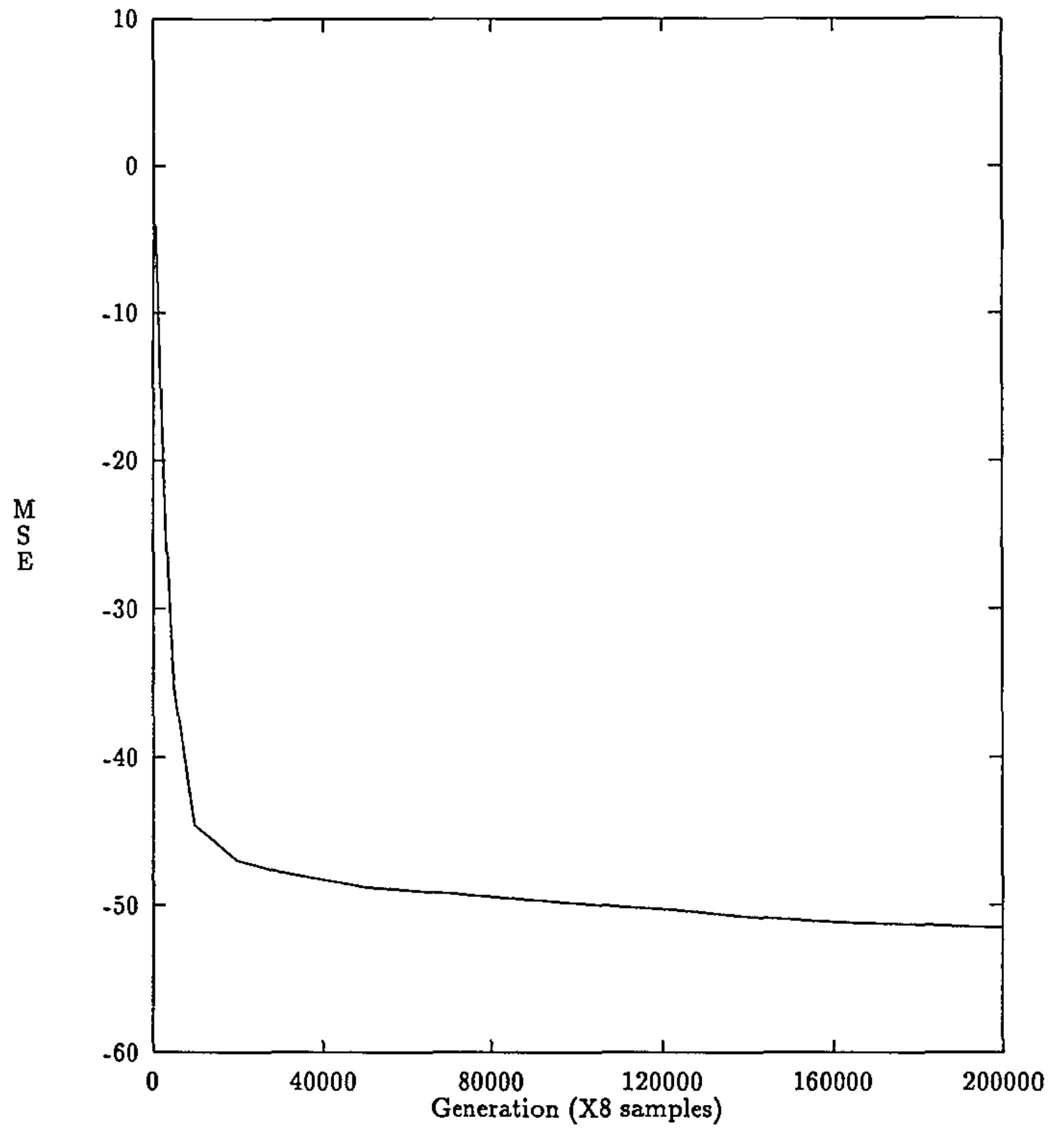

Figure 6.3: Poles close to the unit circle (three coefficients), MSE (in dB) *vs* generations. The population size N = 50, the string length L = 30 (10-bits for each coefficient), the bias is 1.6955, the mutation rate p_m = 0.0555, the random seed is 12345678.

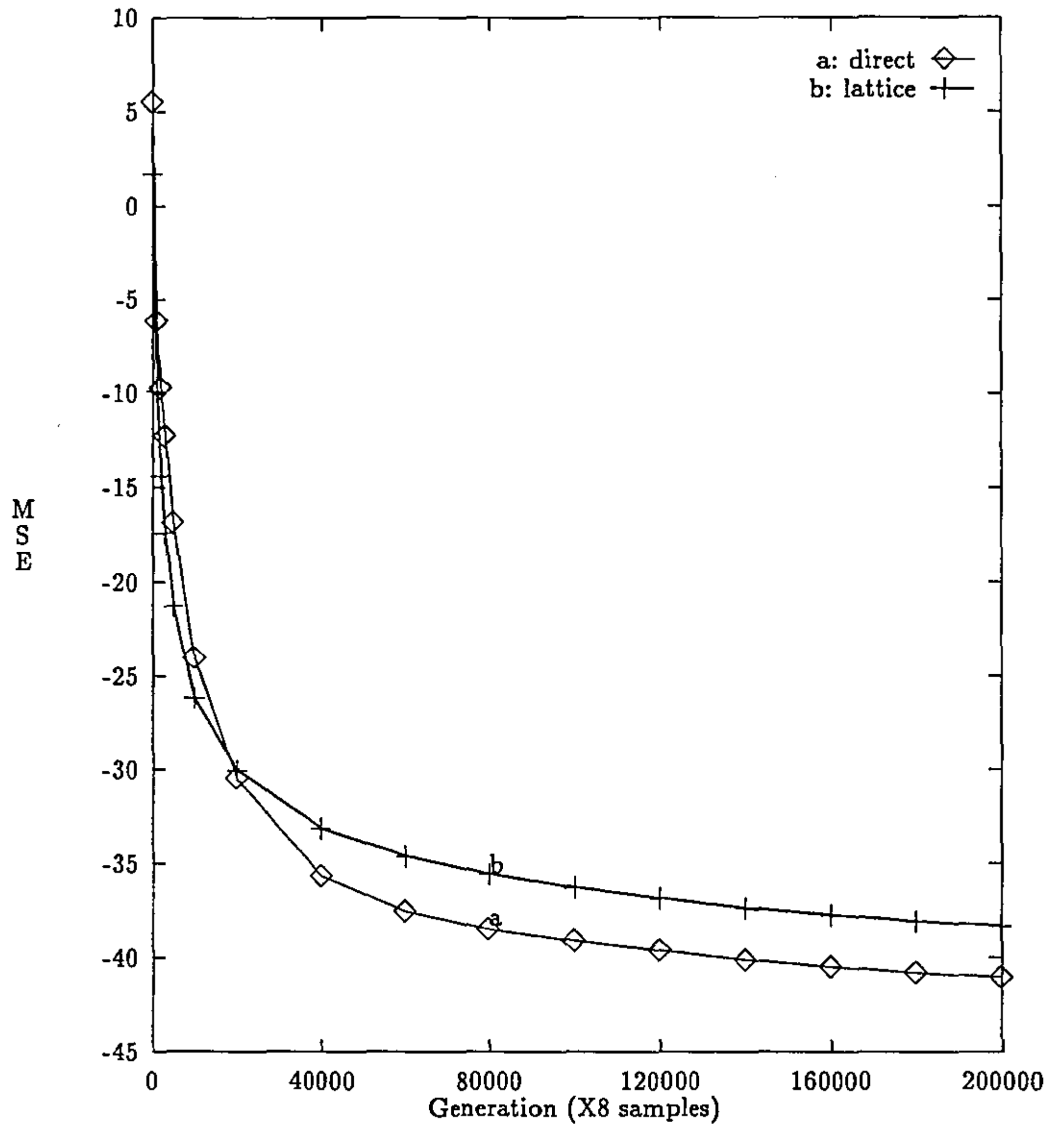

Figure 6.4: Poles close to the unit circle (five coefficients), MSE (in dB) *vs* generations. The population size N = 200, the bias is 1.6955, the probability of mutation $p_m = 0.0555$, the random seed is 12345678. a) direct, string length L = 50 (10-bits for each coefficient); b) lattice, string length L = 75 (15-bits for each coefficient).

Table 6.1: The coefficients of the direct structure using Genitor.

coefficients	plant (direct)	adaptive filter
$b_1(n)$	1.40	1.406027
$b_2(n)$	-0.98	-0.978495
$a_0(n)$	0.50	0.503421
$a_1(n)$	0.40	0.407625
$a_2(n)$	-0.89	-0.874878

Table 6.2: The coefficients of the lattice structure using Genitor.

coefficients	plant (lattice)	adaptive filter
$k_0(n)$	-0.707071	-0.706168
$k_1(n)$	0.980000	0.978149
$v_0(n)$	0.225982	0.224730
$v_1(n)$	0.846000	0.847366
$v_2(n)$	0.890000	0.893928

6.3.3 Higher Order IIR Filters

An order three unknown system

$$H(z) = \frac{0.0154 + 0.0462z^{-1} + 0.0462z^{-2} + 0.0154z^{-3}}{1.0000 - 1.9900z^{-1} + 1.5720z^{-2} - 0.4583z^{-3}}, \tag{6.9}$$

is identified by lattice, cascade and parallel adaptive IIR filter structures. The cascade and parallel forms are constructed using first or second order filters (in direct form, see Chapter 5). The mean squared error performances are given in Figure 6.5. It shows that the lattice form gives the best result, and that the cascade form has a better performance than the parallel form. The convergence speeds for the three forms are similar. The coefficient results (obtained from first run of the 20) for lattice structure, given in Table 6.3, shows that the Genitor is roughly able to identify the seven coefficients after 200000 generations. The overall results are better than the SGA's.

The second example, a fifth order unknown system

$$H(z) = \frac{0.0073 - 0.0184z^{-1} + 0.0115z^{-2} + 0.0115z^{-3} - 0.0184z^{-4} + 0.0073z^{-5}}{1.0000 - 4.5064z^{-1} + 8.2615z^{-2} - 7.6908z^{-3} + 3.6326z^{-4} - 0.6961z^{-5}} \tag{6.10}$$

is identified by the lattice, cascade and parallel structures as well. The mean square error (in dB) performances are given in Figure 6.6, and the coefficients of the lattice structure in Table 6.4. The results show that the lattice form gives the best MSE performance, and that the cascade form is better than the parallel form, which is the same as in the previous example. For the coefficients, Genitor can roughly identify eight out the eleven coefficients, but not each coefficient exactly. The overall results are also better than the SGA's.

Table 6.3: The coefficients of the lattice structure using Genitor.

coefficients	plant (lattice)	adaptive filter
$k_0(n)$	-0.875587	-0.920591
$k_1(n)$	0.835463	0.656362
$k_2(n)$	-0.458300	-0.443831
$v_0(n)$	0.085646	0.115604
$v_1(n)$	0.145491	0.167254
$v_2(n)$	0.076846	0.094461
$v_3(n)$	0.015400	0.015815

The third example, a seventh order unknown system

$$H(z) = \frac{0.0002 + 0.0011z^{-1} + 0.0032z^{-2} + 0.0054z^{-3} + 0.0054z^{-4} + 0.0032z^{-5} + 0.0011z^{-6} + 0.0002z^{-7}}{1.0000 - 3.9190z^{-1} + 7.0109z^{-2} - 7.2790z^{-3} + 4.6934z^{-4} - 1.8690z^{-5} + 0.4236z^{-6} - 0.0420z^{-7}}, \tag{6.11}$$

is identified by the three structures, and the MSE performances are given in Figure 6.7. Genitor can not identify most of the coefficients. This shows that Genitor can not perfectly solve high order IIR filter problems in the coefficient sense, but it does offer some improvement over most of the conventional LMS algorithms in the MSE sense. However, it can be employed in a first stage search, to be followed by other search methods to improve its performance. Still, the lattice structure's performance is better than cascade and parallel structures.

Table 6.4: The coefficients of the lattice structure using Genitor.

coefficients	plant (lattice)	adaptive filter
$k_0(n)$	-0.964752	-0.882260
$k_1(n)$	0.991410	-0.174535
$k_2(n)$	-0.980882	-0.911924
$k_3(n)$	0.961684	0.958617
$k_4(n)$	-0.696100	-0.649281
$v_0(n)$	0.000329	0.013298
$v_1(n)$	0.000760	0.008265
$v_2(n)$	0.005577	0.006223
$v_3(n)$	0.006815	0.008498
$v_4(n)$	0.014497	0.014191
$v_5(n)$	0.007300	0.006724

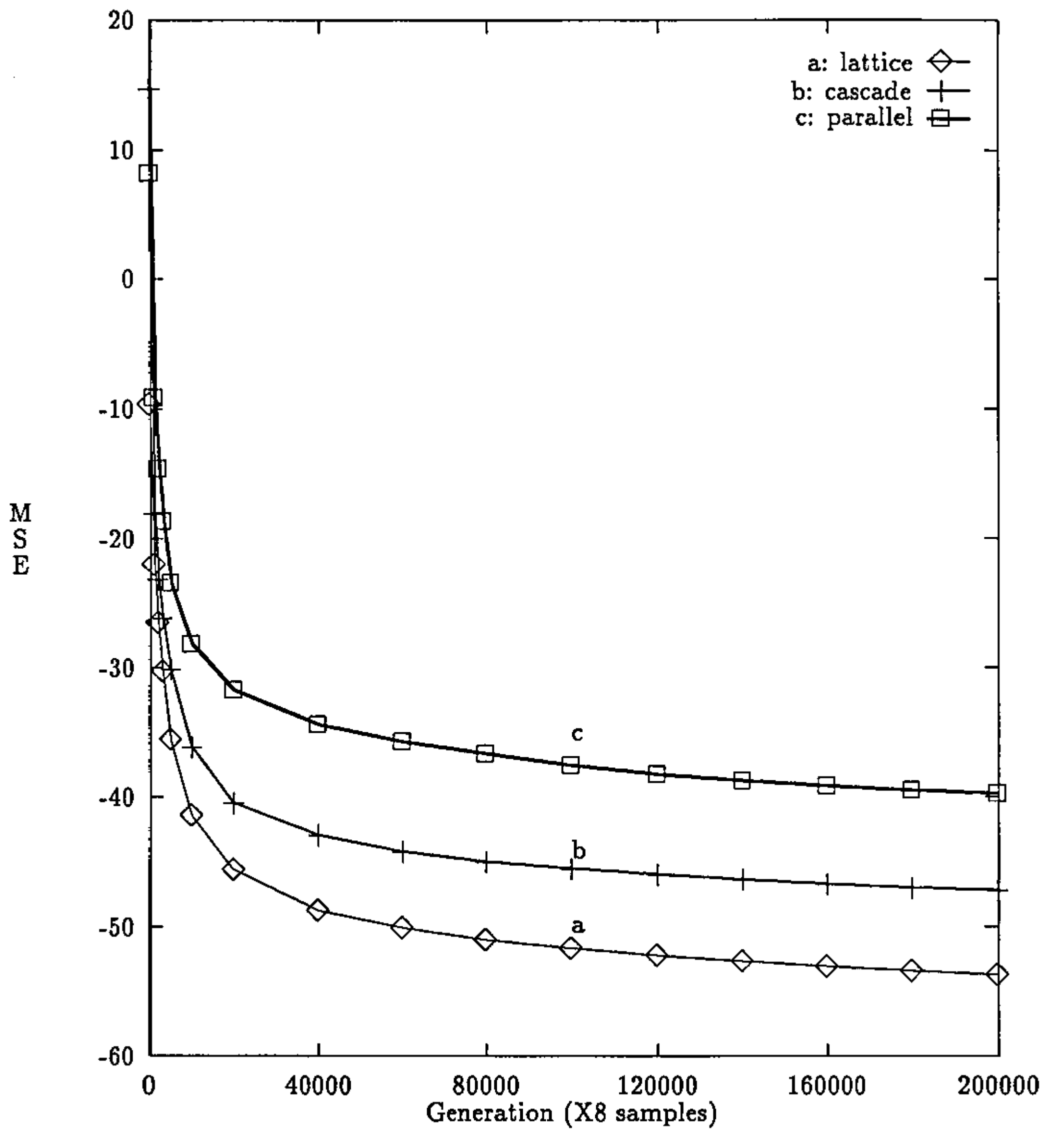

Figure 6.5: The third order filter, MSE (in dB) *vs* generations. The population size N = 200, the string length L = 105 (15-bits for each coefficient), the bias is 1.6955, the probability of mutation p_m = 0.0555, the random seed is 12345678. a) lattice structure; b) cascade structure; c) parallel structure.

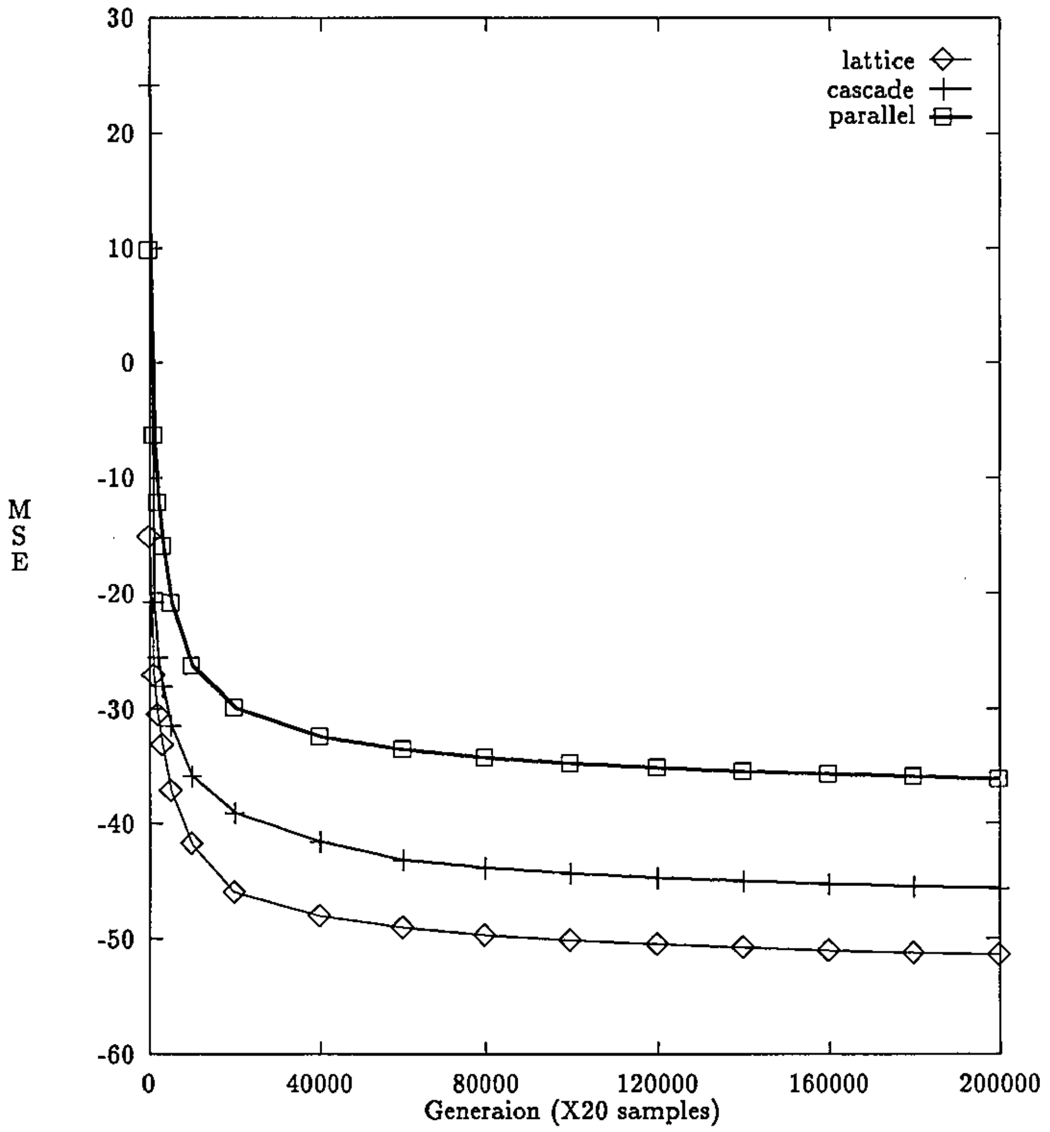

Figure 6.6: The fifth order filter, squared squared error (in dB) *vs* generations (average 20 independent runs). The population size N = 200, string length L = 165 (15-bits for each coefficient), the bias is 1.6955, the mutation rate p_m = 0.0555, the random seed is 12345678.

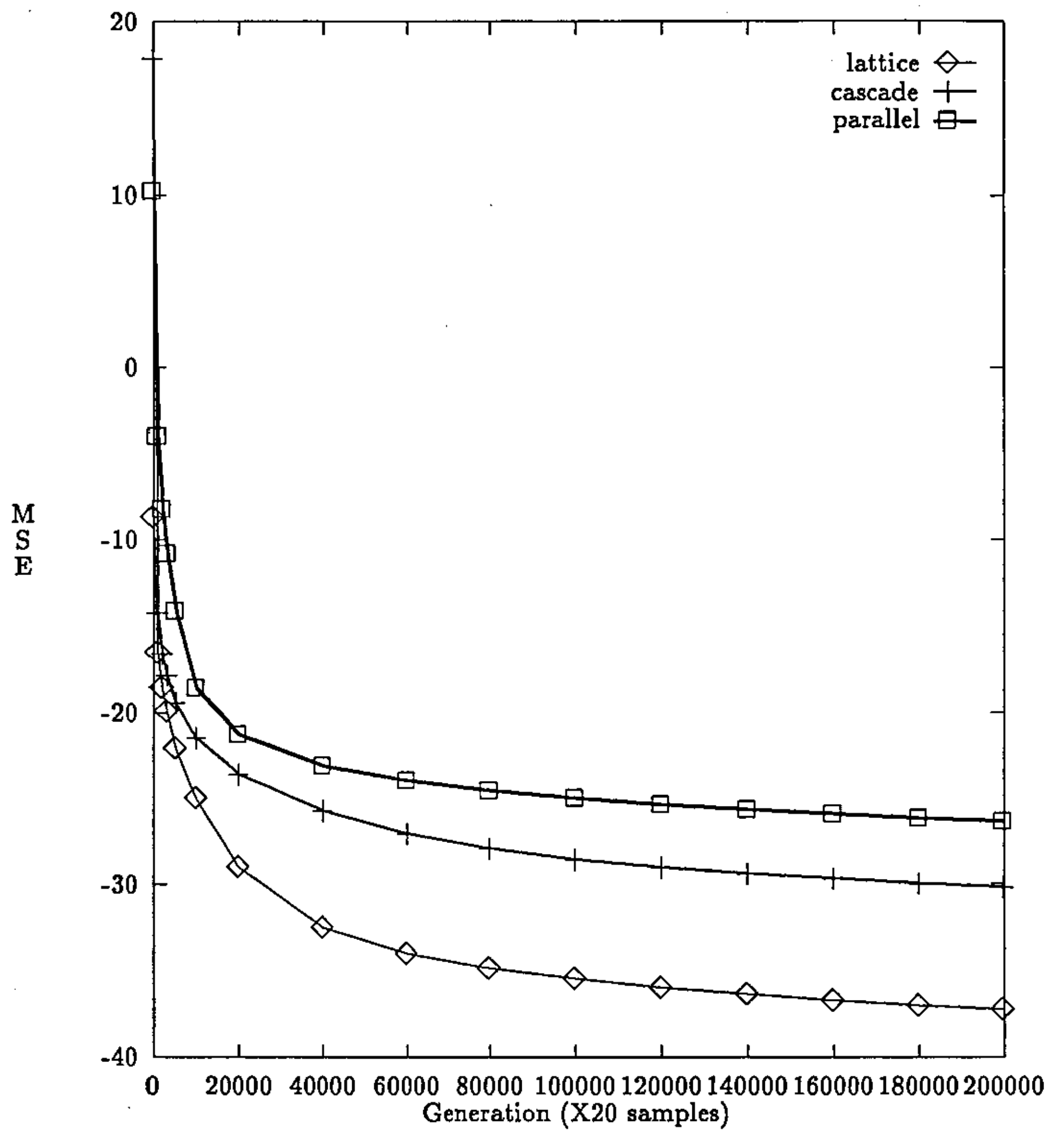

Figure 6.7: The seventh order filter, MSE (in dB) *vs* generations. The population size N = 200, string length L = 225 (15-bits for each coefficient), the bias is 1.6955, the mutation rate p_m = 0.0555, the random seed is 12345678.

6.3.4 Genitor Parameters

The population size. We ran simulations for the various population sizes 50, 100, 200 and 400, and the MSE performances are given in Figure 6.8. Population size 200 gives the best performance, this agrees with the relatively large population size requirement of Genitor. However, the population size is problem dependent: a population with too many members results in long waiting times for significant improvement [72]. The population size of 400 will also double the computation cost in our experiments, so a population size of 400 is not a good choice. The experimental structure used is the lattice, and the unknown system is given by equation (6.9). Similar simulations show that this population size is also the best for filters (6.10) and (6.11)

The results of different string lengths (5, 8, 10, 15, and 20) are observed in Figure 6.9, using the filter which is represented by equation (6.9) and the lattice structure. It shows that the string length L = 105 (15-bits for each coefficient) gives the best result. Similar simulations show that this string length is also the best for filters (6.10) and (6.11).

The bias is bounded between 1.0 and 2.0, and we use many different bias values (1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.6955, 1.7, 1.8, 1.9, and 2.0) in our simulation, which are shown in Figure 6.10. In these simulations, the filter (6.9) and the lattice structure were used. The results show that bias = 1.6955 is the best (for ease of viewing, we have not given all the bias value results), and we have used this value in all the simulations we have performed.

We tried many mutation rates (0.001, 0.005, 0.01, 0.02, 0.03, 0.04, 0.05, 0.0555, 0.06, and 0.075) in our simulations, several of which are shown in Figure 6.11. We found that a mutation rate of 0.0555 is the best, so we again used it in all our simulations. In this experiment the filter (6.9) and the lattice structure were used.

In the Genitor package, the random seed was randomly chosen by D. L. Whitley [71] to be 12,345,678 (it has to be chosen between 1 and 2,147,483,647). We also experimented with another 32 random seeds in our simulation (2^0, 2^1, 2^2, ... , $2^{31} - 1$), and the corresponding MSE performances are given in Figure 6.12.

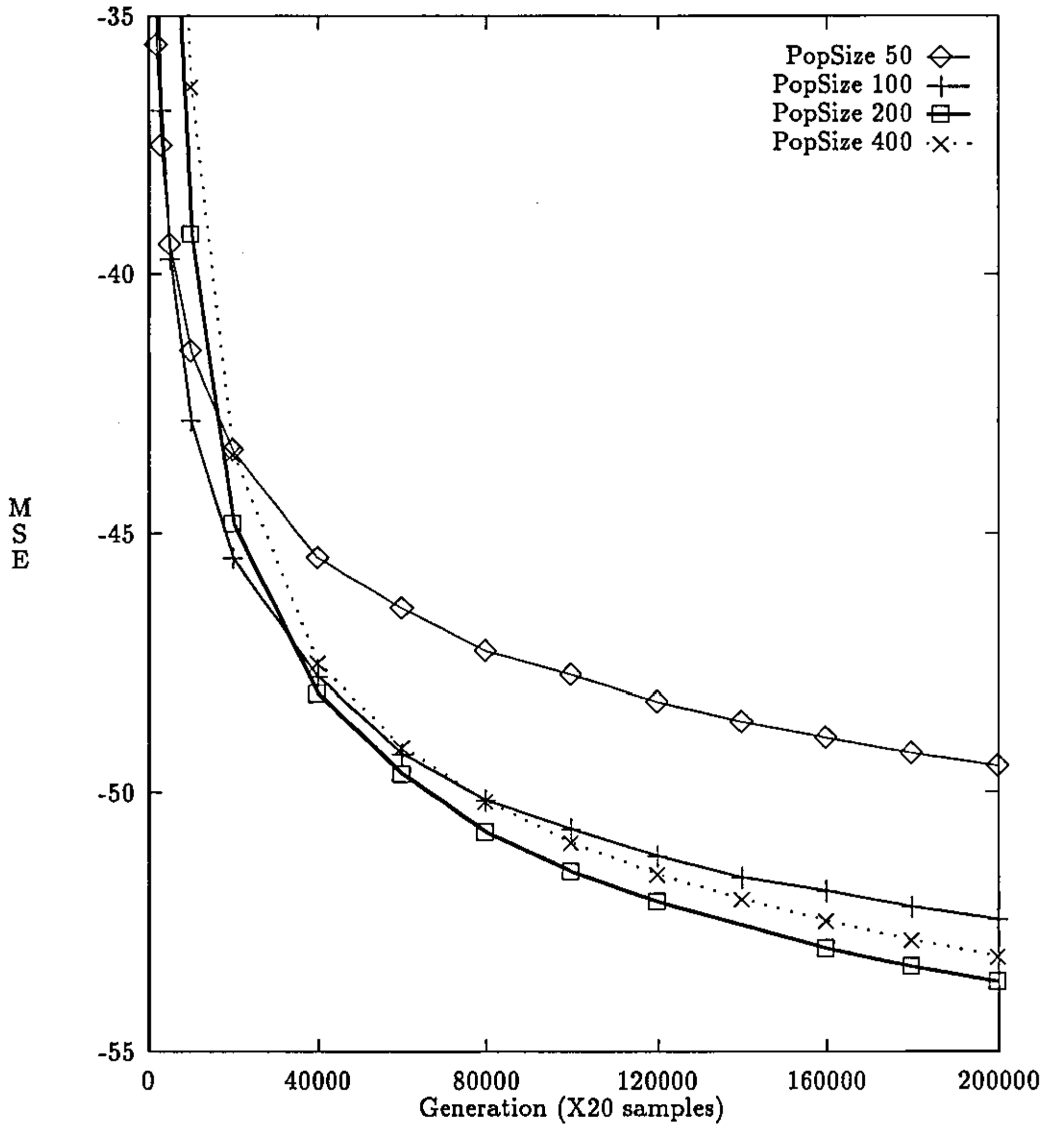

Figure 6.8: The population comparison (using the third order filter and the lattice structure), MSE (in dB) *vs* generations. The string length L = 105 (15-bits for each coefficient), the bias is 1.6955, the mutation rate p_m = 0.0555, and the random seed is 12345678.

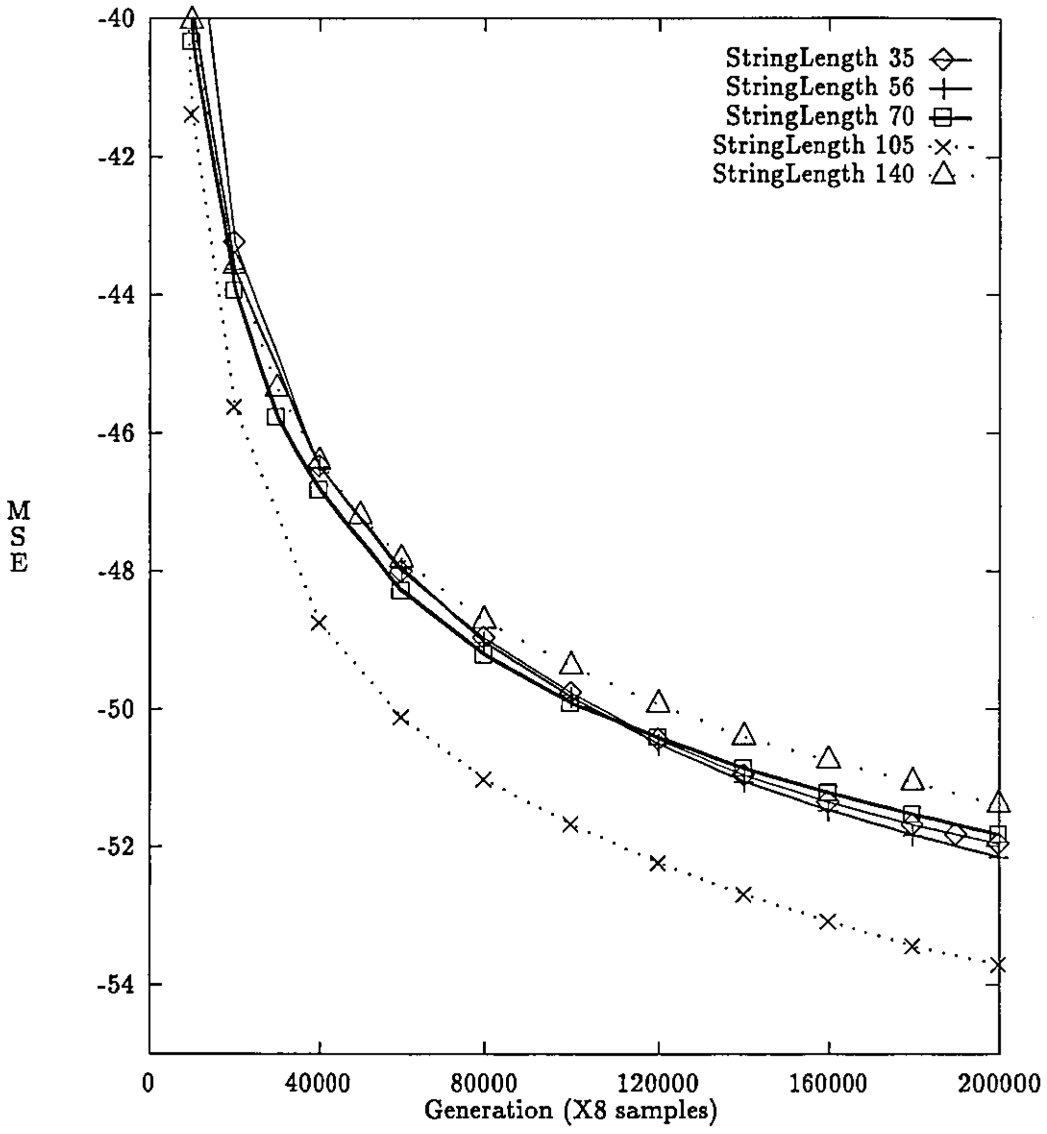

Figure 6.9: String length comparison (using the third order filter and the lattice structure), MSE (in dB) *vs* generations. The population size N = 200, the bias is 1.6955, the mutation rate p_m = 0.0555, and the random seed is 12345678.

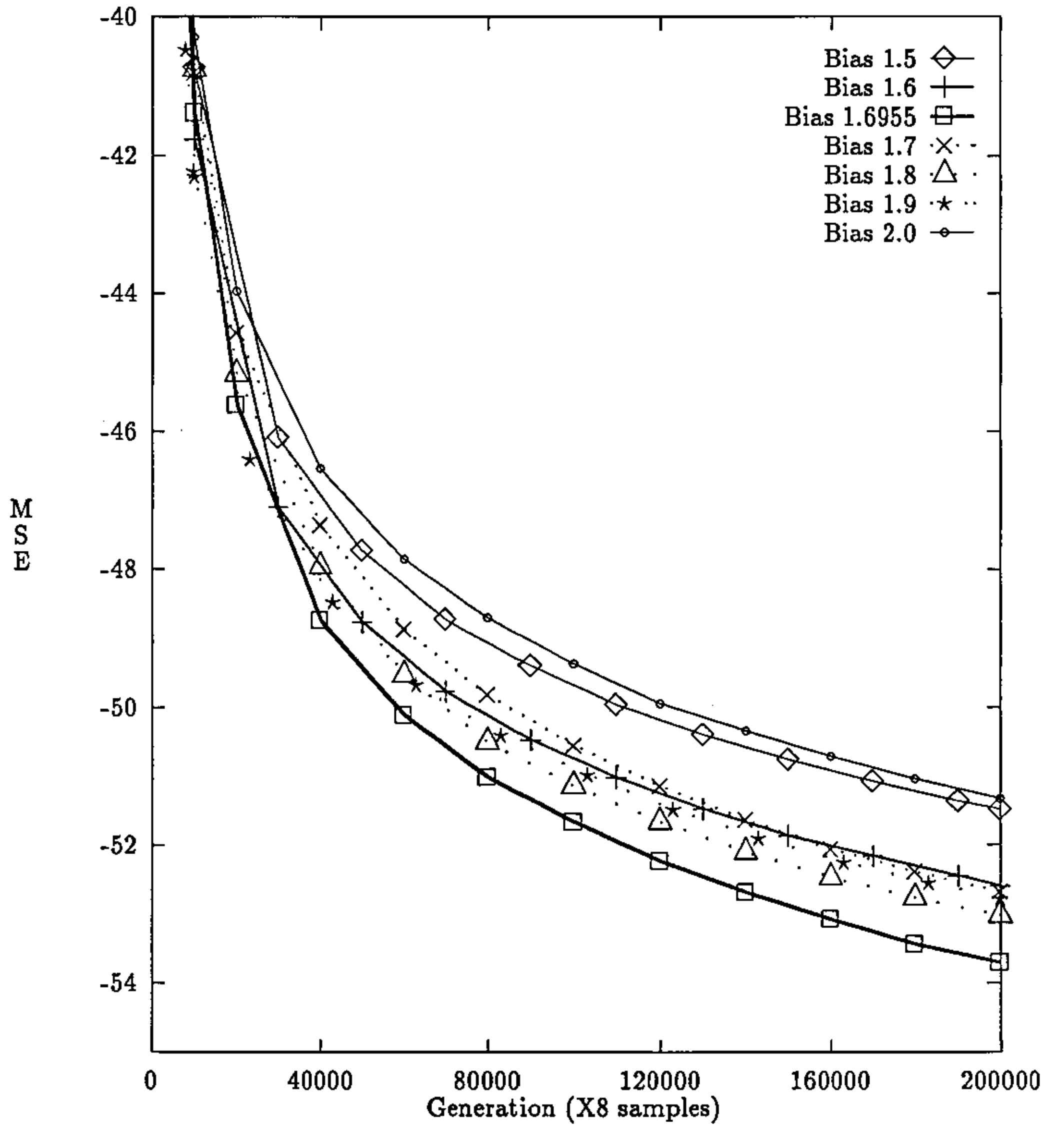

Figure 6.10: The Bias comparison (using the third order filter and the lattice structure), MSE (in dB) *vs* generations. The population size N = 200, string length L= 105, the mutation rate p_m = 0.0555, and the random seed is 12345678.

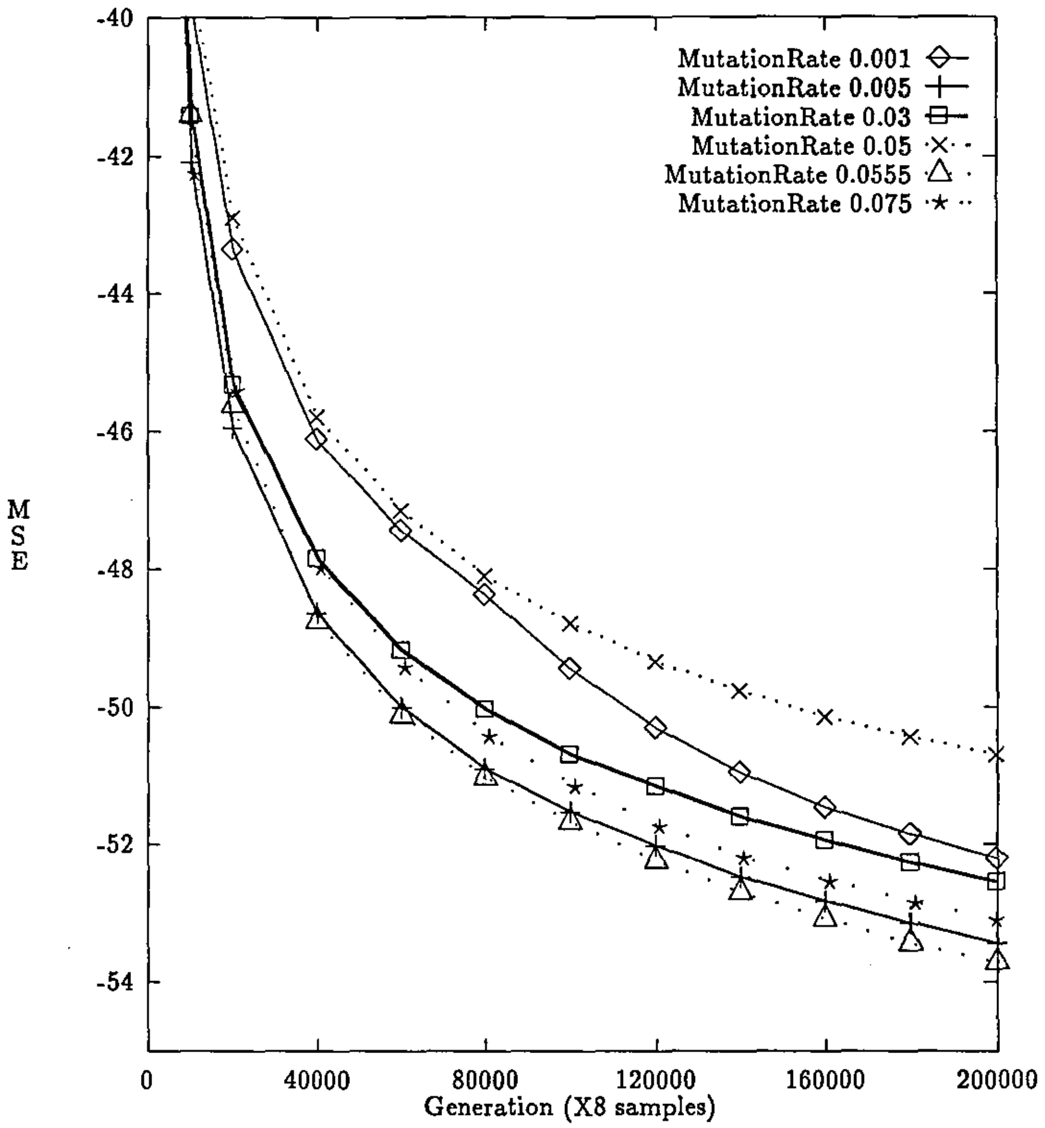

Figure 6.11: The mutation rate comparison (using the third order filter and the lattice structure), MSE (in dB) *vs* generations. The population size N = 200, string length L= 105, the bias is 1.6955, and the random seed is 12345678.

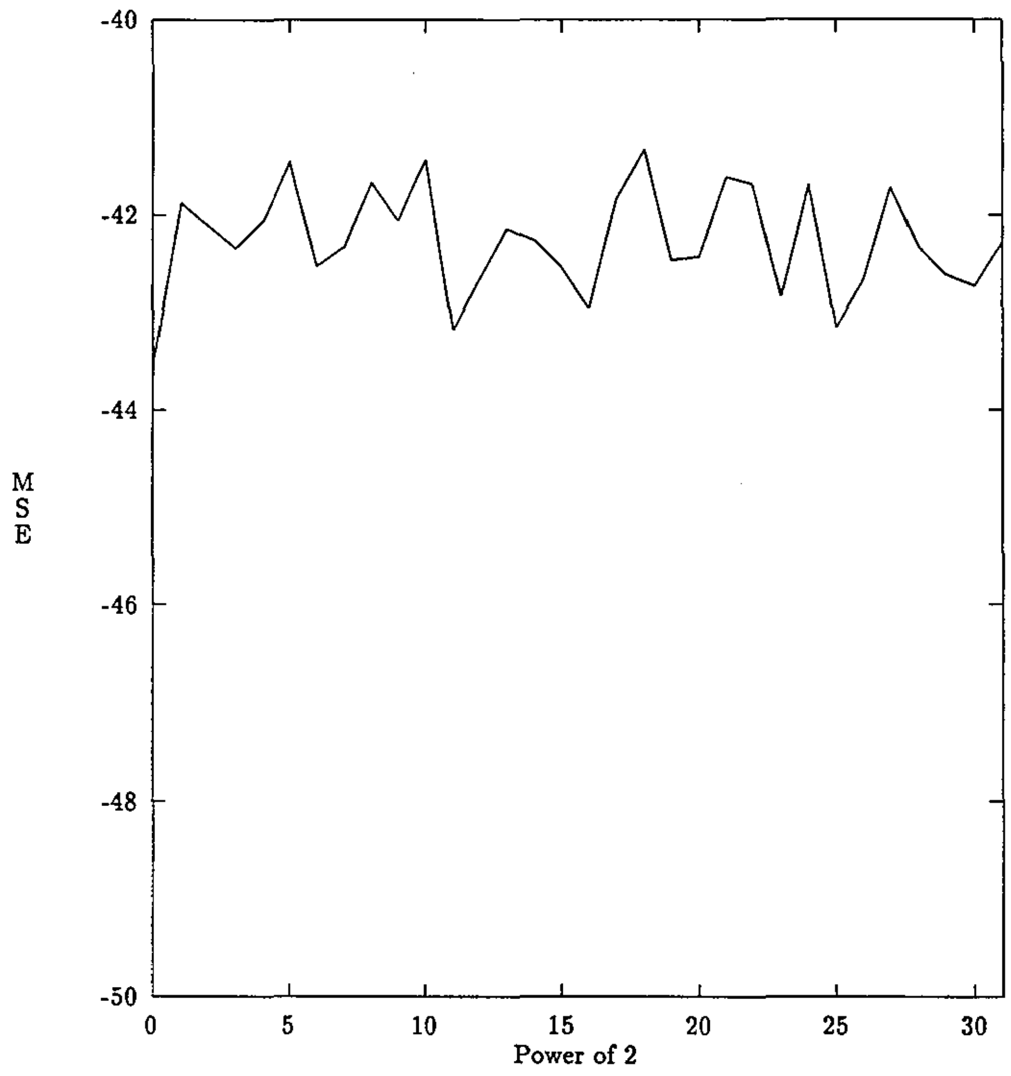

Figure 6.12: The random seed comparison, MSE *vs* random seed (powers of two). The plant $(0.5\text{-}0.4z^{-1}+0.89z^{-2})/(1.0\text{-}0.7z^{-1}+0.1z^{-2})$ and the lattice structure are used. The population size N = 200, string length L = 75 (15-bits for each coefficient), the bias is 1.6955, the data sample window width is 8 samples, and the mutation rate is 0.0555.

6.4 Discussion

6.4.1 MSE Performance

The overall mean square error performances obtained from Genitor are better than those obtained using the SGA. The reason is that in Genitor, the steady state selection helps to maintain the population diversity, so as to avoid the premature convergence that the SGA had faced. In the second order case, the direct structure is slightly better than the lattice, for various reasons. In higher order filter cases, the lattice structure's performance is the best, and the cascade structure's performance is better than the parallel structure's. One reason is that the lattice structure has the biggest population diversity, and the parallel structure has the least. Another reason why the cascade and parallel structures are not as good as the lattice structure is due to the propagation of quantization errors through the filter banks, resulting in an erroneous estimate of error for that particular filter [5].

6.4.2 Coefficient results

The coefficient results show that increasing the number of coefficients or the order would increase the difficulty Genitor has in identifying these coefficients, because Genitor is still not a perfect algorithm for solving IIR filter problems. But Genitor offers a big improvement over the SGA for identifying the coefficients. In previous studies [5], [6], the sixth order and the tenth order filters only have six and ten coefficients respectively, because of the modeling method. In our study, the third order, fifth order and seventh order filters have seven, thirteen and fifteen coefficients respectively.

6.4.3 Comparison with Gradient Algorithms

Comparing the results of Chapters 5 and 6 with Chapter 4, we find that the first advantage genetic algorithms have over gradient algorithms is that they can tackle IIR filtering problems with poles close to the unit circle very well. For the case we have given in our simulations, gradient algorithms [10], [39] failed to produce any converged result.

The second advantage of genetic algorithms is that they have the ability to solve higher order IIR filtering problems. Although the coefficient results are not perfect, they can act as a first stage in IIR filtering search algorithms.

6.4.4 Computational Complexity

Genitor works on a single individual per generation, choosing two individuals for the birth according to linear ranking, and choosing the currently worst individual in the population for replacement by the newly born individual to form one generation. Genitor starts with ranking the population. Once an initial ranking is established, Genitor does not need to completely sort the population again. Each generated individual is simply inserted in its proper place. However, the search for this place requires *O(logN)* steps if a binary search is used. Moreover, the selection of a single individual from the ranked list can also be done in *O(logN)* steps. Since both of these steps must be performed N times to fill an equivalent population (in comparison with the generation-based schemes), the computational complexity of the selection scheme of Genitor is *O(NlogN)* [19]. It is obviously greater than the SGA requires, which is *O(logN)* [19]. The crossover and mutation in Genitor are more complicated than those in SGA, so the overall computational complexity of Genitor is greater than the SGA's.

6.5 Summary

In this chapter, we have applied Genitor to the adaptation of IIR filter problems. We investigated bi-modal, poles close to the unit circle and higher order filter problems, and the performance of direct, lattice, cascade and parallel structures. In the bi-modal case, we demonstrated that the Genitor has the ability to solve the multi-modal adaptive IIR filter problem. For the case when the poles are close to the unit circle, we demonstrated that Genitor has a superior ability to tackle these kind of problems over conventional gradient algorithms. For higher order adaptive IIR filters (three or more), simulation results show that the lattice structure is the best structure for Genitor to employ.

The cascade and parallel structure may have different groups of coefficients to realize the same transfer function. For example, the coefficients of one section in a cascade or parallel system can be interchanged with those of another while still realizing the same overall transfer function. This non-uniqueness introduces additional saddle points into the performance surface, causing the search to become less efficient [10], [41]. This further confirms the preference for the lattice structure.

However, even for the lattice structure, when the filter order increases (in terms of the number of coefficients), it will be more difficult for Genitor to identify all the coefficients. In all the experiments, we performed a very large number of simulations to chose the best Genitor parameter, and use them in all our simulations. Generally, Genitor produce better results than the SGA, but there remain improvements which can be made, which will be discussed in the next chapter.

Chapter 7

Conclusion

7.1 Introduction

The topic of this thesis is the study of the application of genetic algorithms to the adaptation of IIR filtering problems. Two different genetic algorithms, the Simple Genetic Algorithm and Genitor, have been applied to the adaptation of IIR filtering problems. These studies have shown that genetic algorithms have a number of advantages over conventional gradient IIR filter algorithms. In the following section, the conclusions for this thesis are drawn, and in the final section, prospective topics for further study are proposed.

7.2 Conclusions Arising from the Research

On the basis of the work performed in this thesis, the following conclusions may be drawn:

- Genetic algorithms are able to solve IIR filtering problems, and offer stable performance.
- Genetic algorithms can identify adaptive IIR filter coefficients for certain orders.
- Genitor provides improved performance over its SGA counterpart.
- Genetic algorithms offer advantages over gradient algorithms for the adaptation of IIR filters with poles close to the unit circle.
- The lattice form structure of an IIR filter offers the best performance among cascade, parallel and lattice form structures.

Each of these points is considered in greater detail in the following paragraphs.

The principal difficulty with adaptation of an IIR filter is the stability problem due to the poles of the filter. The conventional gradient algorithms do not have guaranteed stability, but require that the stability is monitored. The results obtained in Chapters 5 and 6 demonstrate that genetic algorithms provide stable performance when solving IIR filter problems. The reason is that genetic algorithms code the real decimal coefficients into binary form, and after the results are obtained decode them into decimal. The decoded coefficients are limited to a certain range, for example the stability triangle, so that the poles of the IIR filter never go beyond the unit circle.

In this thesis, we provide some filter coefficient results. The genetic algorithms are able to identify the coefficients of an IIR filter up to a certain order. The results from the SGA show that it can roughly identify the coefficients of order three IIR filters, which have seven coefficients. When the number of coefficients is greater than this, for example the order five filter, the SGA was unable to identify most of the coefficients. Genitor can roughly identify the coefficients of a fifth order IIR filter. So in a coefficient sense, Genitor provides an improved performance over the SGA.

From the results of Chapters 5 and 6, we can conclude that Genitor provides better overall MSE performance compared with its SGA counterpart. The main reason is that Genitor differs from the SGA in that reproduction produces one offspring at a time, this offspring replaces the least fit individual in the population, and ranking selection is used in the selection phase. These prevent the premature convergence of the SGA, and lead the algorithm to a better convergence level.

The results of Chapter 4 show that for an IIR filter with poles extremely close to the unit circle, gradient algorithms [10] and [39] failed to obtain any convergent results. When we conducted the same experiment using the genetic algorithms in Chapters 5 and 6, convergent results were obtained. This shows that genetic algorithms are global, robust searching algorithms and provide stable convergence.

The computer simulations have used three IIR filter structures: cascade, parallel and lattice. Due to the difficulty of judging the coefficient range of higher order direct form IIR filters, the direct structure was not used for filter orders greater than two. The first advantage of the lattice structure is that it is simpler to choose the lattice coefficient limiting range than for the cascade and parallel structures. In our simulations, we limit the k coefficients within the range (-1,1), and for the cascade and parallel structures, we limit the first or the second denominator coefficient to lie within the stability triangle to ensure stable convergence.

The second advantage is that the lattice structure provides a numerical stability advantage over direct and other structures. The third advantage is that the lattice performace surface does not have any saddle points. However, the cascade and the parallel structures are faced with this saddle point problem because several first or second order filter combinations could realize the same transfer function. Together with the simulation results in Chapters 5 and 6, we conclude that the lattice structure is the ideal IIR filter structure in our studies.

7.3 Areas for Further Investigation

To conclude this chapter, several suggestions for further study are presented. From the simulation results, we can see that the level of the convergence floor still needs to be improved, and that the best performance of Genitor can only identify eight out of eleven coefficients of the fifth order filter. For the seventh order IIR filter, Genitor can not give the correct value of any coefficient. This requires further investigation.

Firstly, the advanced operators of genetics can be used in genetic algorithms. These operators include low level operators (diploidy, dominance, inversion, duplication and deletion, etc.), and high level operators (migration, marriage restriction, and sharing functions, etc.) [13]. These operators provide a greater ability for human beings and animals to survive in nature.

Secondly, completely different genetic algorithms can be applied to the adaptation of IIR filter problems, such as CHC (a genetic algorithm) [27], parallel genetic algorithms [28], [73], and the messy genetic algorithms [74] etc..

Thirdly, the similar natural evolution algorithms can be applied to the adaptation of IIR filter problems, such as Genetic Programming [25], evolutionary programming and evolution strategies [75]. These evolution algorithms emerged at the same time as genetic algorithms, but provide different features.

References

[1] Ma, Q. and Cowan, C.F.N., 'Genetic Algorithms Applied to the Adaptation of IIR Filters', *SIGNAL PROCESSING*, in press.

[2] Ma, Q. and Cowan, C.F.N., 'Steady State Genetic Alogrithm Approach to the Adaptation of IIR Filters', Proceeding of International Conference on Digital Signal Processing, vol. 1, pp. 148-153, Cyprus, June 1995.

[3] Etter, D.H., Hicks, M.J., and Cho, K.H., 'Recursive Adaptive Filter Design Using an Adaptive Genetic Algorithm', *Proceeding of International Conference on Acoustics, Speech, and Signal Processing*, pp. 635-638, 1982.

[4] Grefenstette, J.J., 'Optimization of Control Parameters for Genetic Algorithms', *IEEE Transactions on Systems, Man, and Cybernetics*, 16, No. 1, pp. 122-128, 1986.

[5] Nambiar, R., and Mars, P., 'Genetic Algorithms for Adaptive Digital Filtering', IEE Colloquium on Genetic Algorithms for Control Engineering, London, 1992.

[6] Nambiar, R. and P. Mars, 'Genetic and Annealing Approaches to Adaptive Digital Filtering', *Proceeding of IEEE 26th Asimolar Conference on Signals, Systems and Computers*, pp. 871-875, Monterey California, 1992.

[7] Nambiar, R., Tang, C. K. K., and Mars, P., 'Genetic and Learning Automata Algorithms for Adaptive Digital Filters', *Proceeding of IEEE International Conference on Acoustics, Speech and Signal Process.*, San Francisco, March 1992.

[8] Nambiar, R., and Mars, P., 'Adaptive IIR Filtering Using Natural Algorithms' , *IEE Workshop on Natural Algorithms In Signal Processing*, London, November 1993.

[9] Nambiar, R., 'Genetic Algorithms and Adaptive Digital Filtering', Internal Report, School of Engineering and Computer Science, University of Durham, September 1991.

[10] Regalia, P.A., 'Stable and Efficient Lattice Algorithms for Adaptive IIR Filtering', *IEEE Transaction on Signal Processing*, vol. 40, pp. 375-388, 1992.

[11] Darwin, C., *On the Origin of Species by Means of Natural Selection*, Murray, London, 1859.

[12] Smith, J.M., *Evolutionary Genetics*, Oxford University Press, Oxford, 1989.

[13] Goldberg, D.E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Weslley Publishing Company, 1989.

[14] Holland, J., *Adaptation In Natural and Artificial Systems*, University of Michigan Press, 1975.

[15] DeJong, K., *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, PhD Dissertation, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, 1975.

[16] Whitley, D.L, *A Genetic Algorithm Tutorial*, Technical Report CS-93-103, Department of Computer Science, Colorado State University, 1993.

[17] Collins, R.J., *Studies in Artificial Evolution.* PhD Dissertation, Department of Computer Science, University of California at Los Angeles, 1992.

[18] Baker, J.E., 'Reducing Bias and Inefficiency in the Selection Algorithm', *Proceeding of the Second International Conference on Genetic Algorithms and Their applications*, pp. 14-21, 1987.

[19] Goldberg, D.E. and Deb, K., 'A Comparative Analysis of Selection Schemes Used in Genetic Algorithms', *Foundations of Genetic Algorithms*, G. Rawlines, ed., Morgan Kaufmann, pp. 69-93, 1991.

[20] Baker, J.E., 'Adaptive selection methods for genetic algorithms'. *Proceeding of an International Conference on Genetic Algorithms and Their Applications*, pp. 101-111, 1985.

[21] Whitley, D.L. and Kauth, J., 'GENITOR: A Different Genetic Algorithm' *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, pp. 118-130, Denver, 1988.

[22] Whitley, D.L., 'The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best'. *Proceeding of the Third International Conference on Genetic Algorithms*, pp. 116-121, 1987.

[23] Spears, W.M., and DeJong, K.A., 'An Analysis of Multi-Point Crossover', *Foundation of Genetic Algorithms*, G. Rawlins, ed., pp. 301-315, 1991.

[24] Syswerda, G., 'Uniform Crossover in Genetic Algorithms', *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann Publishing, 1989.

[25] Tackett, W.A., *Recombination, Selection, and The Genetic Construction of Computer Programs*, PhD Dissertation, Department of Computer Science University of Southern California, 1994.

[26] Smith, R.E., and Goldberg, D.E., 'Diploidy and Dominance in Artificial Genetic Search', *Complex Systems 6*, pp. 251-285, 1992.

[27] Eshelman, L., 'The CHC Adaptive Search Algorithm', *Foundations of Genetic Algorithms and Classifier Systems*, G. Rawlins, ed., pp. 265-283, Morgan-Kaufmann, 1991.

[28] Pettey, C.B., Leuze, M.R., and Grefenstette, J.J.,'A Parallel Genetic Algorithm', *Proceeding of the 2nd International Conference on Genetic Algorithms*, pp. 155-161, MIT, 1987.

[29] Cowan C.F.N. and Grant, P.M., *Adaptive Filters*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[30] Widrow B. and Stearns S.D., *Adaptive Signal Processing*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[31] Mulgrew B. and Cowan C.F.N., *Adaptive Filters and Equalisers*, Kluwer Academic Publishers, Norweel, Mass., 1988.

[32] Haykin, S, *Adaptive Filter Theory*, 2nd edition, Prentice Hall, 1991.

[33] Cowan, C.F.N., 'Performance Comparisons of Finite Linear Adaptive Filters', *IEE Proceedings, Part F*, vol. 134, no. 3, pp. 211-216, June 1987.

[34] Cioffi, J. M. and Kailath, T., 'Fast, Recursive Least Squares Transversal Filters for Adaptive Filtering', *IEEE Transaction on Acoustics, Speech, Signal Processing*, vol. ASSP-32, No. 2, pp. 304 - 337, 1984.

[35] Yang B. and Böhme J. F, 'Rotation-Based RLS Algorithms: Unified Derivations, Numerical Properties, and Parallel Implementations', *IEEE Transaction on Signal Processing*, vol. 40, No. 5, pp. 1151 - 1166, 1992.

[36] Shynk J. J., 'Adaptive IIR Filtering', *IEEE ASSP Magazine*, pp. 4 -21, April 1989.

[37] Johnson C. R, JR., 'Adaptive IIR Filtering: Current Results and Open Issues', *IEEE Transaction on Information Theory*, vol. IT-30, No. 2, pp. 237 - 250, March 1984.

[38] Claasen, T.A.C.M, Mecklenbrauker, W.F.G, and Peek, J.B.H., 'Effects of Quantization and Overflow in Recursive Digital Filters', *IEEE Transaction*

on Acoustics, Speech, and Signal Processing, vol. ASSP-24, pp. 517-529, 1976.

[39] Fan, H., and Jenkins, W.K., 'A New Adaptive IIR Filter', *IEEE Transaction on Circuits Systems*, vol. CAS-33, pp. 939-947, 1986.

[40] Steiglitz, K. and McBride, L. E., 'A Technique for the Identification of Linear Systems', *IEEE Transaction on Automatic Control*, vol. AC-10, pp. 461-464, October 1965.

[41] Nayeri, M., and Jenkins, W.K., 'Alternate Realizations to Adaptive IIR Filters and Properties of Their Performance Surfaces', *IEEE Transaction on Circuits and Systems*, vol.36, pp. 485-496, 1989.

[42] Parikh, D., Ahmed N., and Stearns, S.D., 'An Adaptive Lattice Algorithms for Recursive Filters', *IEEE Transaction on Acoustics, Speech, and Signal Processing*, vol. ASSP-28, pp. 485-496, 1989.

[43] Ljung, L. and Söderström, T., *Theroy and Practice of Recursive Identification*, MIT Press, Cambridge, 1983.

[44] Tang, C. K. K., and Mars, P., 'Stochastic learning Automata and Adaptive IIR Filters', *IEE PROCEEDINGS-F*, vol. 138, August 1991.

[45] Glover, J.R., 'Adaptive noise cancelling applied to sinusoidal interferences', *IEEE Transaction on Acoustics, Speech, Signal Processing*, vol. ASSP-35, pp. 481-491, December 1977.

[46] Atal, B.S. and Schroeder, M.R., 'Adaptive Predictive Coding of Speech Signals', *Bell System Technical Journal*, vol. 49, pp. 1973-1986, October 1970.

[47] Atal, B.S. and Schroeder, M.R., 'Predictive Coding of Speech and Subjective Error Criteria' *IEEE Transaction on Acoustics, Speech, and Signal Processing*, vol. ASSP -27, pp. 247-254, June 1979.

[48] Makhoul, J., 'Linear Prediction: A Tutorial Review', *Proceeding of the IEEE*, vol. 63, No. 4, pp. 561-580, April 1975.

[49] Proakis, J.G., *Digital Communications*, Second Edition, McGraw-Hill, Singapore, 1989.

[50] Qureshi, S.U.H., 'Adaptive Equalization', *Proceeding of the IEEE*, vol. 73, No. 9, September 1985.

[51] Mulgrew, B, and Cowan, C.F.N., 'An Adaptive Kalman Equalizer: Structure and Performance', *IEEE Transaction on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, No. 12, pp. 1727-1734, December 1987.

[52] Gibson, G.J., Siu, S, and Cowan, C.F.N., 'The Application of Nonlinear Structures to the Reconstruction of Binary Signals', *IEEE Transaction on Signal Processing*, vol. 39, No. 8, pp. 1877-1884, August 1991.

[53] Widrow, B. 'Adaptive Noise Cancelling: Principles and Applications', *Proceeding of IEEE*, vol. 63, pp. 1692-1716, 1975.

[54] Harrison, W. A., Lim, J. S. and Singer, E., 'A New Application of Adaptive Noise Cancellation', *IEEE Transaction Acoustics, Speech, and Signal Process*, vol. ASSP-34, pp. 21-27, January 1986.

[55] Mendel, J. M., *Discrete Techniques of Parameter Estimation: The Equation Error Formulation.* Marcel Dekker, New York, 1973.

[56] Stearns, S. D., 'Error Surfaces of Recursive Adaptive Filters',*IEEE Transaction on Circuits Systems*, vol. CAS-28, pp. 603-606, 1981.

[57] Feintuch, P. L., 'An adaptive recursive LMS filter', *Proceeding of the IEEE*, vol. 64, pp. 1622-1624, November 1976.

[58] Johnson, C.R. Jr., and Larimore, M.G., 'Comment on and Additions to "An Adaptive Recursive LMS Filter"', *Proceeding of the IEEE*, vol. 65, pp. 1399 - 1402, September 1977.

[59] Widrow, B. and McCool, J. M., 'Comments on "An Adaptive Recursive LMS Filter"', *Proceeding of the IEEE*, vol. 65, pp. 1402-1404, September 1977.

[60] Parikh, D., and Ahmed, N., 'On an Adaptive Algorithms for IIR Filters', *Proceeding of the IEEE*, vol. 66, pp. 585-588, May 1978.

[61] Larimore, M. G., Treichler, J. R., and Johnson, C. R., Jr., 'SHARF: An Algorithm for adapting IIR digital filters', *IEEE Transaction on Acoustics, Speech, and Signal Processing*, vol. ASSP-28, August 1980.

[62] Johnson, C. R. Jr., 'A Convergence Proof for A Hyperstable Adaptive Recursive Filter', *IEEE Transaction Information Theory*, vol. IT-25, Nov. 1978.

[63] Johnson, C.R. Jr., and Taylor, T., 'Failure of A Parallel Adaptive Identifier with Adaptive error Filtering', *IEEE Transaction on Automatic Control*, vol. AC-25, December 1980.

[64] Stoica, P. and Soderstrom, T., 'The Steiglitz-McBride Algorithm Revisited - Convergence Analysis and Accuracy Aspects', *IEEE Transaction on Automatic Control*, vol. AC-26, pp. 712-717, 1981.

[65] Gray, A.H., Jr. and Markel, J.D., 'Digital Lattice and Ladder Filter Synthesis', *IEEE Transaction on Audio and Electroacoustics*, vol. AU-21, pp. 491-500, 1973.

[66] Gray, A. H., JR, and Markel, J. D., 'A Normalized Digital Filter Structure', *IEEE Transaction on Acoust. Speech and Signal Processing*, vol. ASSP-23, June 1975.

[67] Mitra, S.K. and Kaiser, J.F., ed, *Handbook for Digital Signal Processing*, Wiley, New York, 1993.

[68] S. J. Flockton and M. S. White, 'The Application of Genetic Algorithms to Infinite Impulse Response Adaptive Filters', IEE Colloquium on Signal Processing , London 1992.

[69] Yao, L., and Sethares, W.A., 'Nonlinear Parameter Estimation via the Genetic Algorithm', *IEEE Transaction on Signal Processing*, vol. 42, No. 4, pp. 927-935, April 1994.

[70] J. Booker, 'Improving Search in Genetic Algorithms', in *Genetic Algorithms and Simulated Annealing*, L. Davis, ed. Morgan Kaufman, 1987, pp. 61-71.

[71] Whitley, D.L., *The Genitor Package*, ftp site: ftp.cs.colostate.edu, /pub.

[72] Goldberg, D.E., 'Sizing Populations for Serial and Parallel Genetic Algorithms', *Proceeding of the Third International Conference ong Genetic Algorithms*, Morgan Kaufmann Publishers, San Mateo, California, 1989.

[73] Mühlenbein, H, 'Evolution in Time and Space - The Parallel Genetic Algorithm, *Foundations of Genetic Algorithms Classifier Systems*, G. Rawlins, ed., pp. 317-337, 1992.

[74] Deb, K, *Binary and Floating-point Function Optimisation using Messay Genetic Algorithms*, PhD dissertation, Department of Engineering Mechanics, The University of Alabama, 1991.

[75] Bäck, T, Rudolph, G and Schwefe, H, 'Evolutionary Programming and Evolution Strategies: Similarities and Differences', *Proceeding of the Annual Conference on Evolutionary Programming*, San Diego, CA, 1993.

Appendix A

Publications

[1] Qiang Ma and Colin F. N. Cowan, 'Genetic Algorithms Applied to the Adaptation of IIR Filters', *Signal Processing*, vol. 48, issue 2, January 1996.

[2] [1] Qiang Ma and Colin F. N. Cowan, 'Steady State Genetic Algorithms Approach to the Adaptation of IIR Filters', Proceeding of The International Conference on Digital Signal Processing, Vol. 1, pp. 148-153, Cyprus, June 1995.

[1]Appended at back of thesis

Appendix B

Computer Programs

B.1 Source Code for SGA

```
/************************************/
/* File main.c, the main driver function */
/************************************/
#include "header.h"

main (argc, argv)
int argc;
char *argv[];
{
    FILE *fp1, *fp2;

    int i, j, gen, n;
    double t, p, *mse;

    system ("clear");

    /* Initialization */
    init_var ();
    randomize (); /* Give a random seed when drand48() is called */
```

```
mse = (double *) malloc (sizeof(double)*(maxgen+1));
if (mse == 0)
{
    printf ("GA runtime error ... out of memory");
}
fp1 = fopen (argv[1], "w+");
fp2 = fopen (argv[2], "w+")

for (gen = 0; gen ≤ maxgen; gen++) *(mse+gen) = 0.0;

for (n = 1; n ≤ num_run; n++)
{
    gen = 0;

    init_pop ();
    for (i = 1; i ≤ pop_size; i++)
    {
        for (j = 1; j ≤ length_chrom; j++) templ[j] = oldpop[i][j];
        evaluation (i, templ, fold, fitnessold);
    }
    s_scale (sfold, fitnessold);
    statistics (sfold, fitnessold);

    *mse += 1.0/avg;

    /* Generation Loop */
    for (gen = 1; gen ≤ maxgen; gen++)
    {
        generation ();
        statistics (sfnew, fitnessnew);
        *(mse+gen) += 1.0/avg;
        if (gen == maxgen-1 || gen == maxgen)
        {
            report (gen, fitnessnew, fnew, newpop);
            show_cof (fp1);
        }
```

```
            swap_pop_fit (oldpop, newpop, sfold, sfnew);
        }
    }

    for (gen = 0; gen ≤ maxgen; gen++)
    {
        *(mse+gen) = (*(mse+gen))/num_run;
        fprintf (fp2, "%d %lf\n", gen, 10*log10(*(mse+gen)));
    }
    fclose (fp1);
    fclose (fp2);
}

/*********************************/
/* File: init.c, initialization for SGA */
/*********************************/

#include "header.h"
void init_var ()
{
    char y;

    printf ("******** GA Data Entry Initialization ********");
    printf ("\n\nEnter max generations = ");
    scanf ("%d", &maxgen);
    printf ("Enter population size = ");
    scanf ("%d", &pop_size);
    printf ("Enter chromosome length = ");
    scanf ("%d", &length_chrom);
    printf ("Enter parameter length = ");
    scanf ("%d", &length_parm);
    printf ("Enter parameter numbers = ");
    scanf ("%d", &num_parms);
    printf ("Enter crossover probability = ");
    scanf ("%f", &pcross);
    printf ("Enter mutation probability = ");
    scanf ("%f", &pmutation);
    printf ("How many cross points do you want = ");
    scanf ("%d", &num_point);
    printf ("How many run do you want in your program = ");
    scanf ("%d", &num_run);
}

    /* Initialize a population at random */

void init_pop ()
{
    int i, j;
    for (i = 1; i ≤ pop_size; i++)
    {
        for (j = 1; j ≤ length_chrom; j++)
        {
            oldpop[i][j] = flip (0.5);
    }
}

/*******************************************************/
/* File: newgen.c, function for evolution. Function create new */
/*      generation through selection, crossover and mutation. */
/*******************************************************/

#"header.h"

void generation ()
{
    int i, j, n, mate1, mate2;
    int oldchrom1[MAXSTR], oldchrom2[MAXSTR];
    int newchrom1[MAXSTR], newchrom2[MAXSTR], temp2[MAXSTR];
    double t, p, q;

    /* Perform any preselection actions before generation */
    preselect(sfold, &avg_scale);

    for (i = 1; i ≤ pop_size; i = i+2)
    {
        mate1 = select (); /* Pick pair of mates*/
        mate2 = select ();

        for (j = 1; j ≤ length_chrom; j++)
        {
            oldchrom1[j] = oldpop[mate1][j];
            oldchrom2[j] = oldpop[mate2][j];
        }
        npoints_cross (oldchrom1, oldchrom2, newchrom1, newchrom2);
        /* uniform_cross (oldchrom1, oldchrom2, newchrom1, newchrom2); */

        for (j = 1; j ≤ length_chrom; j++)
        {
            newpop[i][j] = newchrom1[j];
            newpop[i+1][j] = newchrom2[j];
        }
        evaluation (i, newchrom1, fnew, fitnessnew);m
        parent1[i] = mate1;
        parent2[i] = mate2;
        /* xsite[i] = jcross; */

        evaluation (i+1, newchrom2, fnew, fitnessnew);
        parent1[i+1] = mate1;
        parent2[i+1] = mate2;
        /* xsite[i+1] = jcross; */
    }
    s_scale (sfnew, fitnessnew);
}

/*****************************************************************/
/* File: eval.c, fitness and coefficients evaluation function, filter order is 3 */
/*       and lattice structure is used here.                                    */
/*****************************************************************/

#include "header.h"
```

```
double cof_k[MAXPOP][3], cof_v[MAXPOP][4];

void evaluation(i, templ, objfunc, fitness)
int i, templ[];
double objfunc[], fitness[];
{
    int n, k, temp2[MAXSTR];
    double t;
    extern double objfun();
    extern double map_parm();

    t = (double) length_parm;
    jposition = 1;

    for (n = 0; n < 3; n++)
    {
        extract_parm(templ, temp2, length_chrom, length_parm);
        cof_k[i][n] = map_parm(decode(length_parm, temp2), 1.0, -1.0, pow(2.0, t)-1.0);
    }
    for (n = 0; n < 4; n++)
    {
        extract_parm(templ, temp2, length_chrom, length_parm);
        cof_v[i][n] = map_parm(decode(length_parm, temp2), 0.2, 0.0, pow(2.0, t)-1.0);
    }
    objfunc[i] = objfun(cof_k[i], cof_v[i]);
    fitness[i] = 1.0/objfunc[i];
}

    /* Swap new population to old population and new fitness to old fitness */

void swap_pop_fit(a1, a2, b1, b2)
int a1[][MAXSTR], a2[][MAXSTR];
double b1[], b2[];
{
    int i, j;

    for (i = 1; i ≤ pop_size; i++)
    {
        for (j = 1; j ≤ length_chrom; j++) a1[i][j] = a2[i][j];
        b1[i] = b2[i];
    }
}

/************************************************/
/* File: objfun.c, evaluate objective function value.      */
/*        filter order is three and lattice structure is used */
/************************************************/

#include "header.h"
extern double gauss ();

double objfun (k, v)
double *k, *v;
{
    int i, n;
    double t;
    double d[4]; /* Plant output */
    double x[4]; /* Input to Plant & Adaptive filter */
    double f[4]; /* Forward stage output */
    double b[4];
    double yout; /* Adaptive filter output */
    double error;

    for(i = 0; i < 4; i++)
    {
        x[i] = 0.0;
        d[i] = 0.0;
        f[i] = 0.0;
        b[i] = 0.0;
    }

    t = 0.0;
    for (n = 0; n != 8; n++)
    {
        for (i = 3; i != 0; i–)
        {
            d[i] = d[i-1];
            x[i] = x[i-1];
        }
        x[0] = gauss();
        d[0] = 0.0154*x[0]+0.0462*x[1]+0.0462*x[2]+0.0154*x[3]
                        +1.9900*d[1]-1.5720*d[2]+0.4583*d[3];

        f[3] = x[0];
        for(i = 3; i != 0; i–)
        {
            f[i-1] = f[i] - k[i-1]*b[i-1];
            b[i] = b[i-1] + k[i-1]*f[i-1];
        }
        b[0] = f[0];

        yout = 0.0;
        for (i= 0; i < 4; i++) yout += v[i]*b[i];
        error = d[0] - yout;
        error = error*error;
        t = t + error;
    }
    t = t/8.0; /* Average 8 independent random gauss noise */
    return (t);
}

/**********************************************/
/* File: coding.c, functions for coding and decoding */
/**********************************************/

#include "header.h"

    /* function for decode binary string to unsigned integr */

unsigned int decode(length, chromosome)
int length, chromosome[];
```

```
{
    int i;
    unsigned int accum, powerof2;

    accum = 0;
    powerof2 = 1;
    for (i = 1; i ≤ length; i++)
    {
        if (chromosome[i]) accum = accum + powerof2;
        powerof2 = 2*powerof2;
    }
    return (accum);
}

    /* Map unsigned integer x to a desired value */

double map_parm(x, maxparm, minparm, fullscale)
unsigned int x;
double maxparm, minparm, fullscale;
{
    double a;
    a = minparm + ((maxparm - minparm)/fullscale)*x;
    return(a);
}

    /* Extract each binary parameter from the whole chromosome */

void extract_parm(chromfrom, chromto, lchrom, lparm)
int chromfrom[], chromto[];
int lchrom, lparm;
{
    int j, jtarget;

    j = 1;
    jtarget = jposition + lparm - 1;
    if (jtarget > lchrom) jtarget = lchrom;
    while(jposition ≤ jtarget)
    {
        chromto[j] = chromfrom[jposition];
        jposition += 1;
        j += 1;
    }
}

    /* Decode binary string to a desired coefficient */

void decode_parms(nparms, lchrom, chrom, parms)
int nparms, lchrom;
int chrom[], parms[];
{
    int j, jposition, lparm;
    int chromtemp[MAXSTR];
    double maxparm, minparm, parameter[MAXSTR];

    j = 1; /* Coefficient counter */
    jposition = 1; /* String position counter */

    for (j = 1; j ≤ nparms; j++)
    {
        extract_parm(chrom, parms, lchrom, lparm);
        parameter[j] = map_parm(decode(lparm, parms), maxparm, minparm,
                                pow(2.0,lparm)-1);
    }
}

    /* Sigma scaling procedure */

void s_scale(scale_fitness, fitness)
double scale_fitness[], fitness[];
{
    int i;

    favg = 0.0;
    sigma = 0.0;
    for (i = 1; i ≤ pop_size; i++) favg += fitness[i];
    favg = favg/pop_size;
    for (i = 1; i ≤ pop_size; i++) sigma += pow((fitness[i]-favg), 2.0);
    sigma = sigma/pop_size;
    sigma = sqrt (sigma);
    for (i = 1; i ≤ pop_size; i++)
    {
        scale_fitness[i] = fitness[i] - (favg - 2.0*sigma);
        if (scale_fitness[i] < 0) scale_fitness[i] = 0.0;
    }
}

/*****************************************/
/* File: report.c, print results on screen or file. */
/*****************************************/

#include "header.h"

void show_string (array)
int array[MAXSTR];
{
    int i;
    for (i = length_chrom; i != 0; i–) printf ("%d", array[i]);
}

show_cof (fp)
FILE *fp;
{
    int n, i, j;
    extern double cof_k[MAXPOP][5], cof_v[MAXPOP][6]; /* lattice */

    for (i = 1; i ≤ pop_size; i++)
    {
        for (j = 0; j != 5; j++) fprintf (fp, "%lf ", cof_k[i][j]);
        for (j = 0; j != 6; j++) fprintf (fp, "%lf ", cof_v[i][j]);
        fprintf (fp, "\n");
    }
```

```
}

report (g, fitness, obj, pop)
int g, pop[MAXPOP][MAXSTR];
double fitness[MAXPOP], obj[MAXPOP];
{
    i, j;

    for (i = 1; i ≤ pop_size; i++)
    {
        printf ("%2d) ", i);
        if (g != 0) printf("(         show_string (pop[i]);
        printf (" %18.6f %lf\n", fitness[i], obj[i]);
    }
    printf("\nsumfitness=%lf max=%lf min=%lf\n", sumfitness, max, min);
    printf ("avg = %lf \n", avg);
}

/************************************************/
/* File statis.c, calculate the statistics of the population */
/************************************************/

#include "header.h"

void statistics (fitness, object)
double fitness[MAXPOP], object[MAXPOP];
{
    int i;
    double sum;

    sumfitness = fitness[1];
    sum = object[1];
    min = object[1];
    max = object[1];

    for (i = 2; i ≤ pop_size; i++)
    {
        sumfitness += fitness[i];
        sum += object[i];
        if (object[i] > max) max = object[i];
        if (object[i] < min) min = object[i];
    {
    avg = sum/pop_size;
    avg_scale = sumfitness/pop_size;
}

/******************************************/
/* File srselect.c, Stochastic remainder selection */
/******************************************/

extern int pop_size;
int choices[1001], nremain;
double fraction[1001];

preselect(x1, x2)
double x1[], *x2;
{
    int j, jassign, k;
    double expected;

    if(*x2 == 0)
        for(j = 1; j ≤ pop_size; j++) choices[j] = j;
    else
    {
        j = 1;
        k = 1;

        do
        {
            expected = x1[j]/(*x2);
            jassign = (int) expected;
            fraction[j] = expected - jassign;
            while(jassign > 0)
            {
                jassign–;
                choices[k] = j;
                k++;
            }
            j++;
        } while (j ≤ pop_size);

        j = 1;
        while(k ≤ pop_size)
        {
            if(j > pop_size) j = 1;
            if(fraction[j] > 0.0)
            {
                if(flip(fraction[j]))
                {
                    choices[k] = j;
                    fraction[j] = fraction[j] - 1.0;
                    k++;
                }
            }
            j++;
        }
    }
    nremain = pop_size;
}

    /* Selection using remainder method */

int select()
{
    int jpick, slect;

    jpick = rnd(1, nremain);
    slect = choices[jpick];
    choices[jpick] = choices[nremain];
    nremain- -;
    return(slect);
```

```
}

/************************************/
/* File crossover,c function for crossover */
/************************************/

#include < stdio.h >
#include < math.h >

extern int pop_size;
extern int length_chrom;
extern int jcross;
extern int num_point;
extern float pcross;
extern float pmutation;

void swap_elements(x, y)
int *x, *y;
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

void swap_bits(array1, array2, x, y)
int array1[], array2[], x, y;
{
    int i, temp;

    for(i = x; i ≤ y; i++)
    {
        temp = array1[i];
        array1[i] = array2[i];
        array2[i] = temp;
    }
}
```

```
void sort_array (n, array)
int n, array[];
{
    register int i, j, min;

    for (i = 1; i < n; i++)
    {
        min = i;
        for (j = i+1; j ≤ n; j++)
            if (array[j] < array[min]) min = j;
        swap_elements (&array[i], &array[min]);
    }
}

    /* N points crossover */
void npoints_cross(parent1, parent2, child1, child2)
int parent1[], parent2[];
int child1[], child2[];
{
    int i, n, x, y;
    static int k;
    point_array[50]; /* Use to store cross site */

    /* Randomly produce cross site */
    for(n = 1; n ≤ num_point; n++) point_array[n] = rnd(1, length_chrom);

    /* If crossover points is odd, make it even by using the length as the last
        crossover point */
    if((num_point % 2) == 1) /* the points have to be pairs */
    {
        k = num_point + 1;
        point_array[k] = length_chrom;
    }
    else
        k = num_point;
    /* Rerrange the cross site from smallest to bigest*/
    sort_array(k, point_array);

    /* According to crossover rate do cross */
    if(flip(pcross))
    {
        for(n = 1; n ≤ k; n = n+2)
        {
            x = point_array[n]+1;
            y = point_array[n+1];
            if(x != y) swap_bits(parent1, parent2, x, y);
        }
    }

    for(i = 1; i ≤ length_chrom; i++)
    {
        child1[i] = mutation(parent1[i], pmutation);
        child2[i] = mutation(parent2[i], pmutation);
    }
}

    /* Uniform crossover */

void uniform_cross(parent1, parent2, child1, child2)
int parent1[], parent2[];
int child1[], child2[];
{
    int i, j, n;

    if(flip(pcross))
        for(n = 1; n ≤ length_chrom; n++)
            if((n % 2) == 0) swap_elements(&parent1[n], &parent2[n]);

    for(i = 1; i ≤ length_chrom; i++)
    {
        child1[i] = mutation(parent1[i], pmutation);
        child2[i] = mutation(parent2[i], pmutation);
    }
}

/*************************************************/
/* File mutation.c, function for mutation, return 0 or 1 */
/*************************************************/

int mutation(a, b)
int a;
double b;
{
    int i;

    if (flip(b))
    {
        if (a == 1) i = 0;
        if (a == 0) i = 1;
        return (i);
    }
    else
        return(a);
}

/*************************************************/
/* File: header.h, this file include all global variables */
/*************************************************/

#include <stdio.h>
#include <math.h>

#define MAXRAND 2147483647.0 /* 32 bits Maximum random number */
#define MAXPOP 1001 /* Maximum population size */
#define MAXSTR 201 /* Maximum length of chromosomes */

extern double drand48 (); /* Generate random number between (0-1) */
double avg, max, min; /* Average,maximum and minmum value of fitness */
double avg_scale;
```

```
float pcross, pmutation; /* Probability of crossover and mutation */
double sumfitness; /* Sum of fitness value */
double fitnessold[MAXPOP], fitnessnew[MAXPOP]; /* The fitness value of
                                        current and new generation */
double fold[MAXPOP], fnew[MAXPOP]; /* Objective function value */
double sfold[MAXPOP], sfnew[MAXPOP]; /* Fitness value after sigma scaling */
double favg, sigma; /* Average fitness value and standard deviation of fitness */
int parent1[MAXPOP], parent2[MAXPOP]; /* The two individual string chosed
                                            as parents */

int pop_size; /* Current population size */
int length_chrom; /* Length of chromosome string */
int length_parm; /* Length of binary parameter */
int num_parms; /* number of parameters */
int oldpop[MAXPOP][MAXSTR], newpop[MAXPOP][MAXSTR]; /* Old and new
                                            population */
int xsite[MAXPOP]; /* Crossover site */
unsigned int xold[MAXPOP], xnew[MAXPOP]; /* Decoded value from binary */
int jposition; /* Bit position in chromosome string */
int jcross; /* Crossover site */
int maxgen;
int num_run; /* Number of individual experements */
int num_point; /* Crossover points number */
```

B.2 Source Code for Genitor

The Genitor package can be found from ftp site: ftp.cs.colostate.edu, /pub. We will give the main driver function and the application functions. The functions have been used in these codes but not appeared independently refer to the Genitor package.

```
/*******************************************/
/* File: main.c, Genitor applied to IIR filter problem */
/*******************************************/

#include <stdio.h>
#include <ctype.h>
```

```
#include "ga_random.h"
#include "gene.h"
#include "ga_global.h"
#include "ga_params.h"
#include "ga_pool.h"
#include "ga_selection.h"
#include "ga_status.h"
#include "ga_signals.h"
#include "op_adapt_mutate.h"
#include "op_red_surrog.h"

extern float iir_eval();

int main (argc, argv)
int argc;
char *argv[];
{
    int i, j;
    int numdiffs, num_exp;
    GENEPTR mom, dad, child;
    FILE *fp, *fp1, *fp2, *fp3, *fp4, *fp5, *fp6, *fp7;

    float *k, *v;
    double *mse, *cof;

    /* Set the global parameters according to command line argument */
    argc- -; /* not include executable program itself */
    argv++;
    parse_command_line (argc, argv);

    /* Print parameter values */
    fprintf (stdout, "\n")
    print_params (stdout);
    fprintf (stdout, "\n")

    /* Seed the random number generator */
    srandom (RandomSeed);
```

```
/* Allocate a genetic pool referenced by the global, Pool */
if ( !(Pool = get_pool(PoolSize, StringLength)) )
    fatal_error(NULL);

/* Allocate temporary storage for parents of reproduction */
mom = get_gene (Pool->string_length);
dad = get_gene (Pool->string_length);

if ( !(mse = (double *) malloc (sizeof(double)*NumberTrials)) )
    fatal_error(NULL);
if ( !(cof = (double *) malloc (sizeof(double)*NumberTrials)) )
    fatal_error(NULL);

if ( !(k = (float *) malloc (sizeof(float)*10)) )
    fatal_error (NULL);
if ( !(v = (float *) malloc (sizeof(float)*10)) )
    fatal_error (NULL);

for ( i = 0; i < NumberTrials; i++) *(mse+i) = 0.0;

fp = fopen("mse", "w+");
fp1 = fopen("cof", "w+");

for (num_exp = 0; num_exp < Experiments; num_exp++)
{
    if (num_exp) CurrentGeneration = 0;

    /* Initialize the genetic pool with data */
    init_pool(SeedPool, Pool, 0, Pool->size, iir_eval, k, v);

    /* Sort the initial genetic pool data */
    sort_pool (Pool);

    /* Optimization */
    for (; CurrentGeneration < NumberTrials; CurrentGeneration++)
    {
```

```
/* Choose two genes for reproduction */
get_parents (mom, dad, Pool, linear, SelectionBias);

/* Reproduce */
numdiffs = red_surrogate_cross(mom->string, dad->string, Pool->string_length);

/* Mutation */
if (MutateRate > 0.0)
    adaptive_mutate (mom->string, Pool->string_length, numdiffs,
                          MutateRate);

/* Choose one of the two offspring to insert into
    the genetic pool */
child = ((bitgen() == 0) ? mom : dad);
child->worth = iir_eval(child->string, StringLength, k, v);

/* Insert new gene into population according to its worth */
insert_gene (child, Pool);

/* Mean square error */
*(mse+CurrentGeneration) += (double) avg_pool (Pool);

/* Filter coefficients */
if (num_exp == Experiments - 1)
{
    if (CurrentGeneration == NumberTrials - 1)
    {
        for (i = 0; i < Pool->size; i++)
        {
            Pool->data[i].worth = iir_eval (Pool->data[i].string,
                                          StringLength, k, v);
            fprintf (fp1, "%lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf
                                          %lf %lf %lf \n",
                    v[0], v[1], v[2], v[3], v[4], v[5], v[6], v[7],
                    k[0], k[1], k[2], k[3], k[4], k[5], k[6]);
        }
    }
```

```
            }
        }
    }

    for (i = 0; i < NumberTrials; i++)
    {
        *(mse+i) = *(mse+i)/Experiments;
        if ( i % 100 == 0) fprintf (fp, "%d %lf\n", i, 10*log10(*(mse+i)));
    }
    fclose (fp);
    fclose (fp1);
}

/*********************************************/
/* File: iir_eval.c, fitness and coefficient evaluation, */
/*********************************************/

#include <stdio.h>
#include <math.h>
#include "gene.h"

#define PARMLENGTH 15

int bit_location;

    /* Convert binary data to decimal data */
unsigned int bin_to_dec (string, length)
GENE_DATA string[];
int length;
{
    int i;
    unsigned int decimal, power_of_two;

    decimal = 0;
    power_of_two = 1;

    /* From right to left */
    for (i = length-1; i != -1; i–)
    {
        if (string[i] == '1') decimal += power_of_two;
        power_of_two = 2*power_of_two;
    }
    return (decimal);
}

    /* Bound value between max and min of the range */

float bound_parm (x, maxparm, minparm, fullscale)
unsigned int x;
float maxparm, minparm;
double fullscale;
{
    float value;
    value = minparm + ( (maxparm-minparm)/fullscale )*x;
    return (value);
}

    /* Exact each binary parameter from the whole gen, right to left */

void extract_string (string, parm_string, string_length, length)
GENE_DATA string[], parm_string[];
int string_length, length;
{
    int i = length - 1, target;

    target = bit_location - length + 1;
    if (bit_location < 0) bit_location = 0;
    while (bit_location ≥ target)
    {
        parm_string[i] = string[bit_location];
        bit_location -= 1;
        i -= 1;
    }
}
```

```
    /* Function to evaluate fitness and coefficient values, lattice */

float iir_eval (string, length, k, v)
int length;
float *k, *v;
GENE_DATA string[];
{
    int i, j;
    float object;
    double t;
    GENE_DATAPTR temp; /* cofficient string */

    extern float object_fun();

    if ( !(temp = (GENE_DATAPTR)malloc(sizeof
                                    (GENE_DATA)*(PARMLENGTH+1))))
        fatal_error("temp memory NULL");

    bit_location = length - 1;

    t = (double) PARMLENGTH;
    for (i = 0; i < 3; i++)
    {
        extract_string (string, temp, length, PARMLENGTH);
        k[i] = bound_parm (bin_to_dec(temp, PARMLENGTH), 1.0, -1.0,
                                                        pow(2.0, t)-1.0);
    }
    for ( i = 0; i < 4; i++)
    {
        extract_string (string, temp, length, PARMLENGTH);
        v[i] = bound_parm (bin_to_dec(temp, PARMLENGTH), 0.2, 0.0,
                                                        pow(2.0, t)-1.0);
    }
    free (temp);
    object = object_fun (k, v);
    return (object);
```

```
}

    /* Function to evaluate fitness and coefficient values, parallel or cascade */

float iir_eval (string, length, a, b)
int length;
float *a, *b;
GENE_DATA string[];
{
    int i, j;
    float object;
    double t;
    GENE_DATAPTR temp; /* cofficient string */

    extern float object_fun();

    if ( !(temp = (GENE_DATAPTR)malloc(sizeof
                                    (GENE_DATA)*(PARMLENGTH+1))))
        fatal_error("temp memory NULL");

    bit_location = length - 1;

    t = (double) PARMLENGTH;

    /* first order cofficients */
    extract_string (string, temp, length, PARMLENGTH);
    a[0] = bound_parm (bin_to_dec(temp, PARMLENGTH), 1.0, -1.0,
                                                        pow(2.0, t)-1.0);

    extract_string (string, temp, length, PARMLENGTH);
    a[1] = bound_parm (bin_to_dec(temp, PARMLENGTH), 2.0, -2.0,
                                                        pow(2.0, t)-1.0);

    /* constant cofficient */
    extract_string (string, temp, length, PARMLENGTH);
        a[2] = bound_parm (bin_to_dec(temp, PARMLENGTH), 3.0, 0.0,
                                                        pow(2.0, t)-1.0);
    /* sencond order cofficients */
    extract_string (string, temp, length, PARMLENGTH);
    b[0] = bound_parm (bin_to_dec(temp, PARMLENGTH), 1.0, -1.0,
                                                        pow(2.0, t)-1.0);
    extract_string (string, temp, length, PARMLENGTH);
    b[1] = bound_parm (bin_to_dec(temp, PARMLENGTH), 1.0-b[0],
                                            -1.0+b[0], pow(2.0, t)-1.0);
    for ( i = 0; i < 2; i++)
    {
        extract_string (string, temp, length, PARMLENGTH);
        b[i+2] = bound_parm (bin_to_dec(temp, PARMLENGTH), 1.12, -1.12,
                                                        pow(2.0, t)-1.0);
    }
    free (temp);
    object = object_fun (a, b);
    return (object);
}

/************************************************/
/* File: objfun.c, lattice form see source code for SGA, */
/*        here give the function parallel and cascade        */
/************************************************/

#ifdef PARALLEL
#define FILTER_STRUCT 1
#else
#ifdef CASCADE
#define FILTER_STRUCT 0
#endif
#endif

extern double gauss();

float object_fun(a, b)
float *a, *b;
{
```

```
int i, n;
float t;
float d[4]; /* Plant output */
float x[4]; /* Input to Plant & Adaptive filter */
float y1[3]; /* First order filter output */
float y2[3]; /* Second order filter output */
float yout; /* Adaptive filter output */
float error;

/* Initialization all input and output to zero at time zero */
for (i = 0; i < 4; i++)
{
    x[i] = 0.0;
    d[i] = 0.0;
}
for (i = 0; i < 3; i++)
{
    y1[i] = 0;
    y2[i] = 0;
}

t = 0.0;
for (n = 0; n != 8; n++)
{
    for (i = 3; i != 0; i-)
    {
        d[i] = d[i-1];
        x[i] = x[i-1];
    }
    for (i = 2; i != 0; i-)
    {
        y1[i] = y1[i-1];
        y2[i] = y2[i-1];
    }

    /* Gauss white noise input */
    x[0] = (float) gauss();
```

```
        d[0] = 0.0154*x[0]+0.0462*x[1]+0.0462*x[2]+0.0154*x[3]
                +1.9900*d[1]-1.5720*d[2]+0.4583*d[3];

        /* If cascade structure is being used */
        if (!FILTER_STRUCT)
        {
            y1[0] = a[2]*x[0] - a[2]*a[1]*x[1] + a[0]*y1[1];
            y2[0] = y1[0] - b[3]*y1[1] - b[2]*y1[2] + b[1]*y2[1] + b[0]*y2[2];
            yout = y2[0];
        }
        /* If parallel structure is being used */
        if (FILTER_STRUCT)
        {
            y1[0] = a[1]*x[0] + a[0]*y1[1];
            y2[0] = b[3]*x[0] - b[2]*x[1] +b[1]*y2[1]+ b[0]*y2[2];
            yout = a[2]*x[0] + y1[0] + y2[0];
        }

        /* Error obtained via minus adaptive out from plant output */
        error = d[0] - yout;
        error = error*error;
        t = t + error;
    }
    return (t);
}
```

B.3 Utility and Other Code

```
/*******************************************/
/* File: gen_util.c, utility for genetic algorithms */
/*******************************************/
#include <stdio.h>
#include <math.h>
extern double drand48 ();

int flip (a)
```

```
double a;
{
    int f;

    if (a == 1.0) f = 1;
    else
    {
        if (drand48 () ≤ a ) f = 1;
        else f = 0;
    }
    return (f);
}

    /* Return a random integer between low and high inclusive */

int rnd(low, high)
int low, high;
{
    int i, j;
    double t;

    if (low ≥ high) i = low;
    else
    {
        t = drand48()*(high-low+1)+low;
        i = (int) t;
        if (i > high) i = high;
    }
    return (i);
}

    /* Give a random seed */

void randomize()
{
    double x;
    printf("Enter seed random value = ");
    scanf ("%lf", &x);
    srand48 (x);
}

/*****************************************/
/* File: guass.c, generate gauss random number */
/*****************************************/

#include <stdio.h>
#include <math.h>

extern double drand48 ();

double gauss ()
{
    double i, j;
    double value;
    i = drand48 ();
    j = drand48 ();
    value = sqrt( -2.0*log(i) )*cos( 2*3.141592654*j );
    return (value);
}

/******************************************/
/* This program synthesis direct form IIR filter to */
/* lattice form (coefficient calculation).          */
/******************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 3 /* Filter order, can be changed */

void main()
{
    int n, i, j;
    double a[N]; /* Denominator of direct form */
    double p[N]; /* Numerator of direct form */
    double v[N]; /* Output coefficients */
    double k[N-1]; /* Lattice reflects coefficients */
    double b[N];

    /* Direct form coefficients */
    a[0] = 1.0; a[1] = -1.4;
    a[2] = 0.98;
    p[0] = 0.5; p[1] = -0.4;
    p[2] = 0.89;

    for(n = N-1; n != 0; n-)
    {
        for(i = 0; i < N; i++)
            b[i] = a[n-i];
        k[n-1] = a[n];

        for(i = 0; i < n; i++)
            a[i] = (a[i] - k[n-1]*b[i])/(1.0 - k[n-1]*k[n-1]);
        v[n] = p[n];

        for(i = 0; i < n; i++)
            p[i] = p[i] - b[i]*v[n];
    }
    v[0] = p[0];

    for(n = 0; n < N; n++)
        printf("v[%d] = %lf\n", n, v[n]);

    printf("\n");
    for(n = 0; n < N-1; n++)
        printf("k[%d] = %lf\n", n, k[n]);
}
```

Steady State Genetic Algorithm Approach to the Adaptation of IIR Filters

Qiang Ma and Colin F. N. Cowan

Department of Electronic and Electrical Engineering,
Loughborough University of Technology, Loughborough, Leics. LE11 3TU
Tel: +44 1509 228119, e-mail: q.ma@lut.ac.uk

Abstract

This paper presents a steady state Genetic Algorithm approach to the adaptation of adaptive IIR filters. Conventional adaptive IIR filter algorithms such as LMS and RLS algorithms suffer from potential instability and complexity problems. Applying Genetic Algorithms to IIR filtering problems provides an alternative way to approach the IIR filtering problem. Genetic Algorithms as IIR filter learning algorithms can provide guaranteed filter stability. In this paper three filter structures - cascade, parallel and lattice are studied, the computer simulation results show that the Genetic Algorithm has advantage in the case where poles are close to the unit circle and for high order filter problems.

1 Introduction

Adaptive infinite impulse response (IIR) filters are used in a wide variety of signal processing and control applications due to the superior system modeling abilities afforded by the poles of an IIR filter transfer function. The difficulties for adaptive IIR filters are, first the potential instability, and second, the mean square error surface of an IIR filter can be multi-modal, causing learning algorithms to converge to a local minimum.

Genetic Algorithm approaches to adaptive IIR filtering have been developed recently [1] [2]. Positive results by using the Genetic Algorithm to tackle the adaptive IIR problem has been demonstrated in [2], especially for high order filters. This work in [2] experimented with cascade, parallel and lattice structures for IIR filters. In this paper, we also explore cascade, parallel and lattice structures. We use more general filter models as both the unknown and the adaptive systems in system identification, which are more difficult for the Genetic Algorithm to solve compared to the models in [2].

The adaptive IIR filter and Genetic algorithms is briefly discussed in section 2 and system modeling and steady state Genetic Algorithm in section 3. The results of some computer simulations of steady state Genetic Algorithm for IIR filters are presented in section 4.

2 Adaptive IIR filters and Genetic Algorithms

Adaptive filters can be classified as adaptive finite impulse response (FIR) filters and adaptive infinite impulse response (IIR) filters. Adaptive FIR filters face computational complexity problems, although there are many fast algorithms. For certain real physical systems, adaptive IIR filters can be more economical, in the sense of lower filter order compared to adaptive FIR filter counterparts. Adaptive IIR filter poles can provide a good match to many real systems.

Adaptive IIR filters face instability problems, especially when the poles are close to the unit circle. Adaptive IIR filters' error surface can be multi-modal, making adaptive IIR filter algorithms very difficult in terms of finding the global optimum. The direct form adaptive IIR filter implementation can exhibit high roundoff noise in the presence of finite precision arithmetic, and remains susceptible to quantization limit cycles [3]. If the IIR filter's pole is near the unit circle, for conventional gradient adaptive IIR filter algorithms, the direct form IIR filter's stability is not guaranteed. For example, algorithms in [4] failed to converge for this condition, algorithms in [4] are variants of Steiglitz-McBride technique [5] where the filter structure remains direct form. This has motivated researchers to look for alternative structures. Cascade, parallel and lattice structures have been documented, see [6], [7] and [8], etc.. The solution given in [6] uses the LMS algorithm on parallel and cascade form adaptive IIR filters. It introduces additional saddle points in the performance surface which are unstable solutions in the parameter space [6]. [7] uses the LMS algorithm on the lattice form adaptive IIR filter and maintains computational complexity $O(M^2)$ for gradient calculation. The algorithms in [8] are normalized lattice-based, the first algorithm is a reinterpretation of the Steiglitz-McBride method, while the second is a variation on the output error method, both of them are $O(M)$ complexity. The coefficients are updated by using the QR-based Gauss-Newton algorithm in [8] which needs many matrix computations and for some cases where poles are extremely close to the unit circle, the algorithms also failed to converge. Genetic Algorithms are another alternative solution to the adaptive IIR filtering problem which is very successful in tackling the poles close to the unit circle and high order filter problems.

Genetic algorithms are search algorithms based on the mechanics of natural selection and genetics [9]. These algorithms encode a potential solution to a specific problem on a simple chromosome-like data(normally binary) structure and apply genetic operators to these structures so as to preserve critical information [10].

Genetic algorithms are a population based, robust optimisation method, especially used to tackle high-dimensional, muti-modal search space problems. The Genetic Algorithm operators include mainly selection, recombination(crossover) and mutation. Applying Genetic Algorithms to optimisation problems begins with a population of chromosomes, which are randomly initialized. After that each binary individual of the population is decoded to a set of parameters(coefficients), the parameters are applied to the optimisation function(such as mean square error) to evaluate the function values. In Genetic Algorithms the function values are termed as the fitness values. Among the population, according to the fitness values, a selection operator is used to select the best individual which will possibly survive in the next generation and form the intermediate population. At this stage the recombination operator can be applied. Picking up a pair of strings among the intermediate population, and crossing them into one another with a probability p_c for exchanging genetic information to produce a new pair of strings. This procedure repeats a certain number of times until the full population is filled. After recombination, the mutation operator can be performed. Each bit of every individual in the whole population can mutate with a very low probability p_m. This provides greater ability to ensure that every part of the search space is visited. After these three operators have been performed, the new population can be evaluated. The evaluation, selection, recombination and mutation construct one Simple Genetic Algorithm (SGA) generation cycle [9].

3 System Modeling and Steady State Genetic Algorithm

Applying Genetic Algorithms on the adaptive filtering problem was first studied by Etter [1]. Later Nambiar and Mars [2] applied Genetic Algorithms to system identification problems. In this paper we again explored system identification by using direct, cascade, parallel and lattice structures.

In system identification, the unknown system can be identified by direct, cascade, parallel and lattice form adaptive filter structures. For an Mth order IIR filter (M > 2)

$$H(z) = \frac{a_0(n) + a_1(n)z^{-1} + ... + a_M(n)z^{-M}}{1.0 + b_1(n)z^{-1} + ... + b_M(n)z^{-M}} \tag{1}$$

the equivalent cascade-form representation of H(z) is

$$H_c(z) = q \prod_{k=1}^{W} \frac{1.0 - a_{1k}(n)z^{-1} - a_{2k}(n)z^{-2}}{1.0 - b_{1k}(n)z^{-1} - b_{2k}(n)z^{-2}}, \tag{2}$$

where W = (M+1)/2, if M is odd or W = M/2, if M is even, the equivalent parallel-form representation of H(z) is

$$H_p(z) = p + \sum_{k=1}^{W} \frac{a_{0k}(n) - a_{1k}(n)z^{-1}}{1.0 - b_{1k}(n)z^{-1} - b_{2k}(n)z^{-2}}, \tag{3}$$

where W = (M+1)/2 if M is odd or W = M/2 if M is even, q and p are constants. The stability of filters during adaptation is guaranteed by constraining the filter coefficients b_{1k}(n) and b_{2k}(n) to lie within the stability triangle [2].

The filter (1) can also be implemented in the form of a lattice with different weights v_i(n) and k_i(n), which is stable if the lattice coefficients k_i(n) are all less than 1. The input-output of the lattice filter at time n can be expressed as:

$$y(n) = \sum_{i=0}^{M} v_i(n)B_i(n) \tag{4}$$

where

$$\begin{aligned} B_i(n) &= B_{i-1}(n) + k_i(n)F_{i-1}(n); \qquad i = M, ..., 1 && (5)\\ F_i(n) &= F_{i+1}(n) - k_i(n)B_i(n-1); \qquad i = M-1, ..., 0 && (6)\\ F_M(n) &= x(n) && (7) \end{aligned}$$

and x(n) is the input signal, and

$$B_0(n) = F_0(n). \tag{8}$$

We will use the above three models to identify the high order unknown system in our simulation experiments.

The genetic algorithm we use in this paper is Genitor [10], which can be termed as the steady state Genetic Algorithm. It provides better performance relative to the Simple Genetic Algorithm and has a few different features relative to the Simple Genetic Algorithm (SGA) which has been used in [2]. First, reproduction produces one offspring at a time. Two parents are selected for reproduction and produce an offspring that is immediately placed back in the population. The second major difference is in how that offspring is placed back in the population. Offspring do not replace parents, but rather the least fit (or some relatively less fit) member of the population. In Genitor, the worst individual in the population is replaced. The third difference between Genitor and the Simple Genetic Algorithm is that fitness is assigned according to rank rather than by fitness proportionate reproduction. Ranking helps to maintain a more constant selective pressure over the course of the search[10].

4 Simulation Results

We present here simulations for direct, cascade, parallel and lattice structures. The squared error has been chosen as the fitness value. In all the results, the squared error obtained from each generation is plotted against the generation number, we use different window lengths to average the instantaneous error to form the error for each experiment shown in the plotted results. The results are obtained after averaging 20 independent simulations, the population size chosen was 200, because normally Genitor requires large population sizes or multiple populations to combat the premature convergence problem [11]. We run Genitor for 200,000 generations, which seems large, but if compared to the SGA in computation terms, it is relatively efficient. In the SGA, in each generation, fitness values for every individual in the whole population has to be calculated, but in Genitor, in each generation, only one individual fitness value is calculated. The Genitor (or steady state) selection bias is 1.6955, the crossover operator in Genitor is two point reduced surrogate form crossover [10] the mutation operator is adaptive mutation, the mutation rate is 0.0555, each coefficient has 15 binary bits.

Experiment 1. Lattice and direct form adaptive structures have been used to identify the unknown system:

$$H(z) = \frac{0.5 - 0.4z^{-1} + 0.89z^{-2}}{1.0 - 1.4z^{-1} + 0.98z^{-2}}. \tag{9}$$

This system has poles at 0.7±j0.7, which are close to the unit circle. Many gradient algorithms failed to identify this special case, for example , algorithms in [4] and [8]. The genetic algorithm gives the results illustrated in figure 1 which shows the advantage of GA over the gradient algorithms for poles close to unit circle problem. The unknown system is a second order system, so we chose lattice and direct form adaptive IIR filters for this experiment.

Experiment 2. An order three system:

$$H(z) = \frac{0.0154 + 0.0462z^{-1} + 0.0462z^{-2} + 0.0154z^{-3}}{1.0000 - 1.9900z^{-1} + 1.5720z^{-2} - 0.4583z^{-3}} \tag{10}$$

is identified by lattice, cascade and parallel adaptive structures. The cascade and parallel forms are constructed by using first or second order filters (in direct form). The results are given in figure 2. It shows that the lattice form gives the best result, the cascade form has better performance than the parallel form. The convergence speeds for the three forms are similar.

Experiment 3. A much higher order seventh unknown system:

$$H(z) = \frac{0.0002 + 0.0011z^{-1} + 0.0032z^{-2} + 0.0054z^{-3} + 0.0054z^{-4} + 0.0032z^{-5} + 0.0011z^{-6} + 0.0002z^{-7}}{1.0000 - 3.9190z^{-1} + 7.0109z^{-2} - 7.2790z^{-3} + 4.6934z^{-4} - 1.8690z^{-5} + 0.4236z^{-6} - 0.0420z^{-7}} \tag{11}$$

is identified by the three structures, the results are given in figure 3. This shows that the Genetic Algorithm has the power to solve a high order problem better than the other structures. The results show the same trend as in Experiment 2.

Experiment 4. We run the simulation for various population sizes, see figure 4. Population size 200 gives the best performance, this agrees with the relatively large population size requirement of Genitor. The experimental structure used is the lattice, filter with the unknown system given by equation (10).

5 Conclusion

In this paper, applying Genetic Algorithms to the adaptation of IIR filtering has been studied. The simulation results do show that the Genetic Algorithm approach to the adaptive IIR filtering problem has some advantages. Above all the lattice structure gave the best results. The cascade structure is better than parallel form. We can draw a conclusion: the lattice structure is the ideal structure, it not only gives the best performance, but also it is easy to choose the coefficient decoding region for genetic algorithm adaptation.

References

[1] Etter, D.H., 'Recursive Adaptive Filter Design Using an Adaptive Genetic Algorithm', Proc. of IEEE Conf. on ASSP, pp.635-638, 1982.

[2] Nambiar, R, and Mars, P., 'Genetic and Annealing Approaches to Adaptive Digital Filtering', Proc. IEEE 26th Asimolar Conference on Signals, Systems and Computers, Monterey California, Oct. 1992.

[3] Claasen, T.A.C.M., Mecklenbrauker, W.F.G., and Peek, J.B.H., 'Effects of Quantization and Overflow in Recursive Digital Filters', IEEE Trans. Acoust., Speech, Signal Processing, vol. ASSP-24, pp.517-529, 1976.

[4] Fan, H. and Jenkins, W.K., 'A New Adaptive IIR Filter', IEEE Trans. Circuits Systems, vol. CAS-33, pp.939-947, 1986.

[5] K. Steiglitz and L.E. McBride, 'A technique for the identification fo linear systems', IEEE Trans. Automat. Contr., vol. AC-10, pp375-388, 1965.

[6] Nayeri, M., and Jenkins, W.K., 'Alternate Realizations to Adaptive IIR Filters and Properties of Their Performance Surfaces', IEEE Trans. on Circuits and Systems, vol.36, pp485-496, April 1989.

[7] Parikh, D., Ahmed, N., and Stearns, S.D., 'An Adaptive Lattice Algorithms for Recursive Filters', IEEE Trans. Acoust., Speech, Signal Processing, vol. ASSP-28, pp. 485-496, April 1989.

[8] Regalia, P.A., 'Stable and Efficient Lattice Algorithms for Adaptive IIR Filtering', IEEE Trans. on Signal Processing, vol. 40, pp. 375-388, 1992.

[9] Goldberg, D.E., 'Genetic Algorithms in Search, Optimization, and Machine Learning', Addison-Weslley Publishing Company, 1989.

[10] Whitley, D., 'A Genetic Algorithm Tutorial', Colorado State University Technical Report CS-93-103, November 1993.

[11] Goldberg, D. and Deb, K., 'A Comparative Analysis of Selection Schemes Used in Genetic Algorithms', in 'Foundations of Genetic Algorithms'. G. Rawlins, ed. Morgan Kaufmann, 1991.

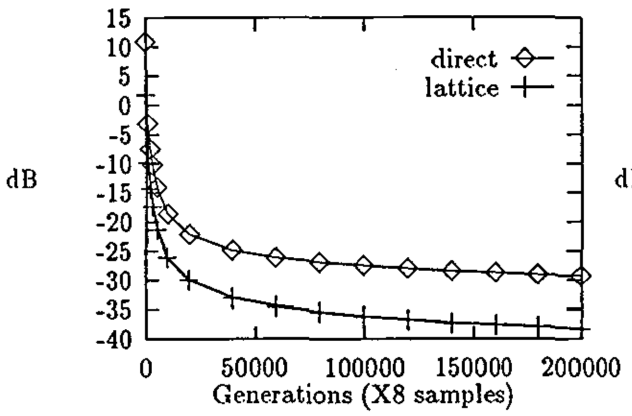

Figure 1: The second order filter, squared error (in dB) vs generations.

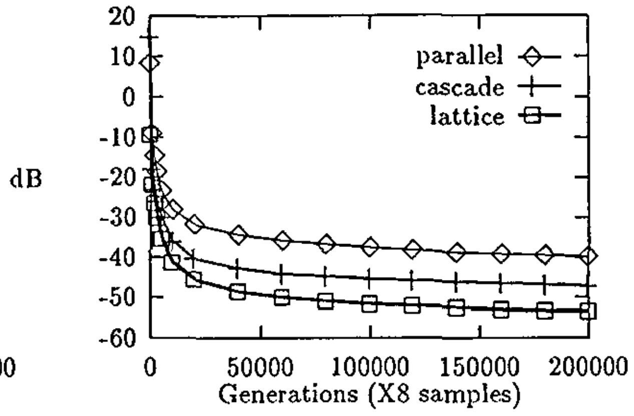

Figure 2: The third order filter, squared error (in dB) vs generations.

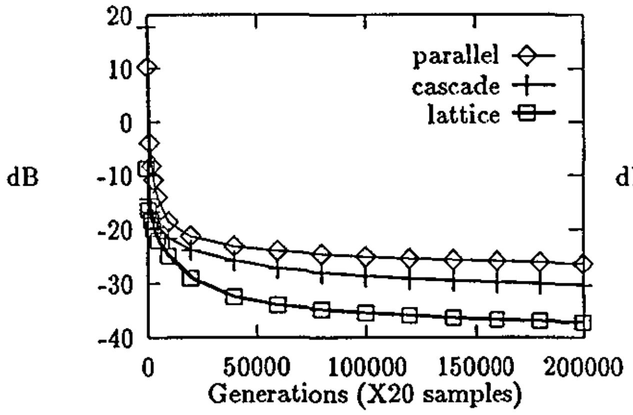

Figure 3: The seventh order filter, squared error (in dB) vs generations.

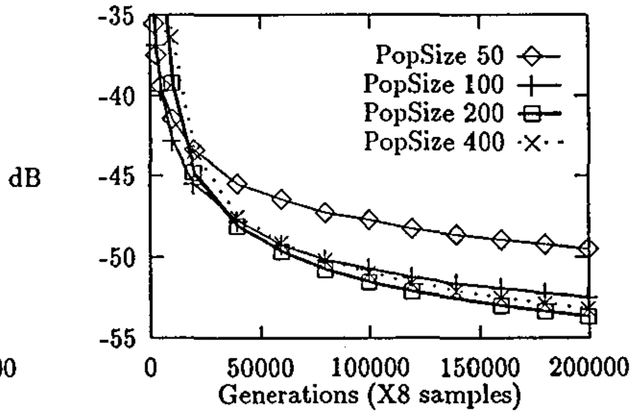

Figure 4: Squared error (in dB) vs generations for various population sizes.