Improved Nelder Mead's Simplex Method and Applications

by

Nam Dinh Pham

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 7, 2012

Keywords: Nelder Mead's simplex method, quasi gradient method,
lossy filter, Error Back Propagation, Lavenberg Marquardt, neural networks

Approved by

Bogdan Wilamowski, Chair, Professor of Electrical and Computer Engineering
Lloyd Stephen Riggs, Professor of Electrical and Computer Engineering
Thaddeus Roppel, Associate Professor of Electrical and Computer Engineering
Michael Baginski, Associate Professor of Electrical and Computer Engineering
Wei-Shinn Ku, Assistant Professor of Computer Science and Software Engineering

Abstract

Derivative free optimization algorithms are often used when it is difficult to find function derivatives, or if finding such derivatives are time consuming. The Nelder Mead's simplex method is one of the most popular derivative free optimization algorithms in the fields of engineering, statistics, and sciences. This algorithm is favored and widely used because of its fast convergence and simplicity. The simplex method converges really well with small scale problems of some variables. However, it does not have much success with large scale problems of multiple variables. This factor has reduced its popularity in optimization sciences significantly. Two solutions of quasi gradients are introduced to improve it in terms of the convergence rate and the convergence speed. The improved algorithm with higher success rate and faster convergence which still maintains the simplicity is the key feature of this paper. This algorithm will be compared on several benchmark functions with the original simplex method and other popular optimization algorithms such as the genetic algorithm, the differential evolution algorithm, and the particle swarm algorithm. Then the comparing results will be reported and discussed.

Acknowledgments

I would like to express my deepest appreciation to my advisor, Professor B. M. Wilamowski. Without his patience and guidance I would not be in the position I am today. From him I have learned a multitude of things, of which, engineering is only the tip of the iceberg.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

SIM: Nelder Mead's Simplex Method

EBP: Error Back Propagation

LM: Lavernberg Marquardt

LMS: Least Mean Square

MLP: Multilayer Perceptron

BMLP: Bridged Multilayer Perceptron

FCC: Fully Connected Cascade

RBF: Radial Basis Function

LVQ: Learning Vector Quantization

Chapter 1

Introduction

The desire for optimality is the inherent nature of humans such as a manufacturer wants to produce its products with the lowest cost, or a delivery company wants to deliver its products to all distributers with the shortest distance to save gasoline, time, etc. These are the typical examples which optimization theories can be applied to give optimal solutions. From the appearance of computers, mathematical theories of optimization have been developed and applied widely. The computer with its computing power has the ability to implement optimization theories very efficiently in the manner of time and cost. The goal of the optimization theories is the creation of a reliable method to optimize models by an intelligent process. Applications of these theories play more important roles for modern engineering and planning, etc.

In real life scientists, engineers, and managers often collect a lot of data and usually fall into difficult situations how to select different factors to obtain desired results. Optimization is a process of how to trade off these factors to find the best solution by evaluating their combinations. Many engineering problems can be defined as optimization problems such as process design, logistics, process synthesis & analysis, telecommunication network, finding of an optimal trajectory for a robot arm, the optimal thickness of steel in pressure vessels, etc [1]. In

practice, optimization algorithms are able to solve these problems but to find the best solution for these problems is often not very easy and straightforward because they include in large search spaces. It will be more challenging particularly in real life systems, which require optimal solutions in an acceptable amount of time.

Optimization is a useful and important tool in the decision science and the analysis of physical systems. In order to use this tool, an objective function has to be defined. This objective function can be the cost, profit, time, etc. Normally, an objective function is modeled by unknown variables to describe its characteristics. And optimization algorithms define values of these variables to meet the requirements of this objective function. If the model is so simplistic, the solution will not reflect useful insights into practical systems. If the model is so complex, optimization algorithms may not give solutions. Therefore, models and optimization algorithms usually have to be complex enough to be handled by the computer. There are numerous optimization algorithms. Each is developed to solve a particular set of problems, and each has its own strength and weakness. Users usually have to evaluate a model and decide which algorithm is suited for [2].

Discrete and continuous optimization: discrete optimization problems are known as integer programming problems. In discrete optimization problems, solutions make sense if and only if variables are integers. To meet this constraint, a good strategy is to solve problems with real variables and then round them up to the closest integers. This type of work is by no means guaranteed to give optimal solutions. In contrast with discrete optimization problems, continuous optimization problems are easier to solve because of the smoothness of continuous functions. Moreover, these problems have an infinite set of solutions with real values; therefore, we can use other information at any point to speculate the function's behavior. However, the same method

cannot be applied to solve discrete optimization problems with a finite set of solutions, where points are close, may have different function values.

Constrained and unconstrained optimization: constrained optimization problems arise from models which have constraints on variables. These can be the constraints of input variables or the constraints to reflect relationships among variables, etc. Unconstrained optimization problems can be considered as particular cases of constrained optimization problems in which constraints of variables can be ignored without effect on the solution. Or these constraints can be counted as penalization terms in the objective functions of unconstrained problems.

Global and local optimization: local optimization algorithms converge much faster than global optimization algorithms. However, its solution is just a local one which is the minimum in the vicinity and it is not guaranteed to be the global solution which is the best of all minima.

Stochastic and deterministic optimization: in some optimization problems, the model cannot be fully defined because it depends on quantities that are unknown at the time of formulation. Normally, a modeler can predict unknown quantities with some degree of confidence. Stochastic optimization algorithms will use these quantifications of the uncertainty to produce solutions that optimize the expected performance of the model. Vice versus with stochastic optimization algorithms, deterministic optimization algorithms assume that the model is fully specified.

Each optimization algorithm has different techniques to converge iteratively to optimal solutions. Some use first derivatives, second derivatives, or function values, etc. to converge. Some accumulate information from previous iterations to predict its sequential convergence to target values. The optimization technique is a key to differentiate one algorithm from another. A good optimization algorithm should possess some following properties:

- *Robustness*: the algorithm has the ability to converge a wide range of problems in its category
- *Efficiency*: the algorithm can converge without too expensive computing cost. This cost can be understood as computing time and storage cost
- *Accuracy*: the algorithm can give solutions with precision. It is not very sensitive with errors when being implemented on computers.

The Nelder Mead's simplex method [3] is a popular derivative free optimization algorithm and is a method of choice for many practitioners. It is the prime choice algorithm in Matlab optimization toolbox. It converges relatively fast and can be implemented relatively easily compared with other classical algorithms relying on gradients or evolutionary computations, etc. Unlike gradient methods [4], [5], the simplex method can optimize a function without calculating its derivatives, which usually require a lot of computing power and are expensive in high dimensional problems as well. This property makes it more advantageous than others.

Although the simplex method is simple and robust in small scale optimization, it easily fails with large scale optimization. In order to become a reliable optimization tool, it has to overcome this shortcoming by improving its convergence rate and convergence speed. This literature will give some new insights on why the simplex method may become inefficient in high dimensional optimization because of its lack of gradient information. This approach explains the low convergence rate without concerning its descent property when the objective function is uniformly convex presented in other literature [6], [7]. The dissertation will particularly present how to improve the simplex method by combining with two different quasi

gradient methods. The improved algorithm without complex mathematic computations can optimize multi-dimensional problems with higher success rate and faster convergence speed.

The genetic algorithm [8], the differential evolution algorithm [9], [10] and the particle swarm algorithm [11], etc. are the other popular derivative free optimization tools which are widely applied and familiar by researchers and practitioners [12]-[14]. These algorithms can perform well in both a global and local search and have the ability to find the optimum solution without getting trapped in local minima. This capability is mostly lacked by local search algorithms such as the calculus-based algorithms or the simplex method. The big issue of global search algorithms is the computational cost which often makes their convergence speed much slower than local search algorithms. The particle swarm optimization is a kind of global search technique. It is a probabilistic technique which is different from the deterministic and stochastic techniques. Compared with the genetic algorithm and the differential evolution algorithm, the particle swarm optimization is simpler in term of computations because its crossover and mutation operation are done simultaneously while the crossover and mutation operation of the genetic algorithm and the differential evolution algorithm are done between each pair in the whole population. With improvements contributed in this paper, the simplex method can be considered as another optional optimization algorithm which can work much more efficiently than other well-known derivative free optimization algorithms in many different fields of engineering or sciences.

1.1 Genetic Algorithm

The genetic algorithm (GA) is invented to mimic the natural behavior of evolution according to the Darwin principle of survival and reproduction [15]. Unlike calculus-based

methods, GA does not require derivatives, and it also has the ability to do a parallel search in the solution space simultaneously. Therefore, it is less likely to get trapped in local minima. Like the particle swarm algorithm and the differential evolution algorithm, GA starts by its initial population, and each individual is called a chromosome to represent a solution. During each generation, chromosomes will be evaluated according to their fitness values and evolved to create new chromosomes for the next generation. New childish chromosomes can be produced in two different ways either by emerging from two parental chromosomes in current generation with the crossover operator or by modifying chromosomes with the mutation operator. In order to maintain the population size, all chromosomes have to go through the natural selecting process. The chromosomes with better genes or better fitness will have higher probability to go to the next generation and other ones with worse genes is more likely to be rejected. This procedure is repeated until the best chromosome close to the optimum solution can be obtained. Another big advantage of GA is that it can be applied in different domains, not just only in optimization problems. However, it still has the limitation of premature convergence and low local convergence speed. Therefore, GA is usually improved by research scholars [16], [17].

1.2 Differential Evolution Algorithm

The differential evolution algorithm (DE) was introduced by R. Storn and K. Price in 1997 [9], [10]. Today it becomes one of the most robust function minimizers with relatively simple self-adapting mutation and is able to solve a wide range of optimization problems. The whole idea of DE is generating a new scheme to compute trial parameter vectors. These new parameter vectors are computed by adding the weighted difference between two population members to a third one. If the resulting vector has a lower objective function value than a

predefined population member, the newly generated vector will replace the vector with which it was compared. Through time, this algorithm has been adapted to increase its efficiency. In 2007, a new concept of multiple trial vectors [18] was introduced into this algorithm. This approach aims to make DE able to converge for a broader range of problems because one scheme of calculating trial vectors may work well with certain type of problems but may not work with other ones. Another approach was proposed where the choice of learning strategies and the two control parameters F (weighing factor) and CR (crossover constant) are dynamically adjusted and also made a significant improvement [19]. Recently, an adaptive differential evolution algorithm with multiple trial vectors can train artificial neural networks successfully and shows its competitive results with the error back propagation algorithm and the Lavenberg Marquardt algorithm [20].

1.3 Particle Swarm Optimization

The particle swarm optimization (PSO) is a concept that simulates the social swarm behavior of a flock of birds or a school of fish in searching for food [21]. The main concept is to utilize the inter-communication between each individual swarm with the best one to update its position and velocity. This algorithm operates on a randomly created population of potential solutions and searches for the optimum value by creating the successive population of solutions. PSO sounds similar to the differential evolution algorithm or the genetic algorithm in term of its selecting strategy of the best child (or the best swarm), but it is really different. In this algorithm, the potential solutions so called swarm particles are moving to the actual (dynamically changing) optimum in the solution space. Each swarm has its own location, best location, velocity, and fitness. In each generation, each swarm will contact with the best swarm and follow him to

update its own information. During its motion, if some swarms find better positions by comparing with their own fitness, they will automatically update themselves. In case there is a swarm finding the new best position, that swarm will be considered immediately as the current best. Because of its global search ability and fast convergence speed compared with other global search algorithms, PSO is applied widespread in optimization.

1.4 Nelder Mead's Simplex Algorithm

The simplex method [3] is a direct downhill search method. It is a simple algorithm to search for local minima and applicable for multidimensional optimization applications. Unlike classical gradient methods, this algorithm does not have to calculate derivatives. Instead it just creates a geometric simplex and uses this simplex's movement to guide its convergence. A simplex is defined as a geometrical figure which is formed by ($n+1$) vertices. Where n is the number of variables of an optimization function, and vertices are points selected to form a simplex. In each iteration, the simplex method will calculate a reflected vertex of the worst vertex through a centroid vertex. According to the function value at this new vertex, the algorithm will do all kinds of operations as reflection or extension, contraction, or shrink to form a new simplex. In other words, the function values at each vertex will be evaluated iteratively, and the worst vertex with the highest function value will be replaced by a new vertex which has just been found. Otherwise, a simplex will be shrunk around the best vertex, and this process will be continued until a desired minimum is met. Moreover, the convergence speed of this algorithm can also be influenced by three parameters α, β, γ (α is the reflection coefficient to define how far a reflected point should be from a centroid point; β is the contraction coefficient to define how far a contracted point should be when it is contracted from the worst point and the reflected point

in case the function value of the reflected point is smaller than the function value of the worst point; γ is the expansion coefficient to define how far to expand from the reflected point in case a simplex moves on the right direction). Depending on these coefficients α, β, γ, the volume of a simplex will be changed by the operations of reflection, contraction, or expansion respectively. The Nelder Mead's simplex method can be summarized as following and more details can be found in the original paper [3].

- Step 1: get α, β, γ, select an initial simplex with random vertices x_0, $x_1, \ldots, x_{n-1}$ and calculate their function values.
- Step 2: sort the vertices $x_0, x_1, \ldots, x_{n-1}$ of the current simplex so that $f_0, f_1, \ldots, f_{n-1}$ in the ascending order.
- Step 3: calculate the reflected point x_r, f_r
- Step 4: if $f_r < f_0$:

  (a) calculate the extended point x_e, f_e

  (b) if $f_e < f_0$, replace the worst point by the extended point $x_n = x_e$, $f_{n-1} = f_e$

  (c) if $f_e > f_0$, replace the worst point by the reflected point $x_n = x_r$, $f_{n-1} = f_r$
- Step 5: if $f_r > f_0$:

  (a) if $f_r < f_i$, replace the worst point by the reflected point $x_n = x_r$, $f_{n-1} = f_r$

  (b) if $f_r > f_i$:

  (b_1) if $f_r > f_{n-1}$: calculate the contracted point x_c, f_c

  (c_1) if $f_c > f_{n-1}$ then shrink the simplex

  (c_2) if $f_c < f_{n-1}$ then replace the worst point by the contracted point $x_n = x_c$, $f_{n-1} = f_c$

(b_2) if $f_r < f_{n-1}$: replace the worst point by the reflected point $x_{n-1} = x_r$, $f_{n-1} = f_r$

- Step 6: if the stopping conditions are not satisfied, the algorithm will return step 2

To describe in details how the Nelder Mead's simplex method works, let f: $\mathbb{R}^n \rightarrow \mathbb{R}$, $x \in \mathbb{R}^n$ be the objective function that should be minimized. For a two dimensional case n= 2, a simplex is a triangle formed by three vertices $B=(x_1, y_1)$, $G=(x_2, y_2)$ and $W(x_3, y_3)$. The function values are evaluated at these vertices $z_i = f(x_i,y_i)$, $i = 1:3$. The subscripts are reordered in the way $z_1 \leq z_2 \leq z_3$. So B is the best vertex, G is the good vertex and W is the worst vertex. Assume α= 1, β= 0.5, γ= 2. This algorithm performs in the following steps:

- Step 1: removal of the worst vertex W and calculate a centroid of rest vertices. In 2-d case shown in Fig. 1.1 the centroid would be average of B&G.

$$M = \frac{B+G}{n} = \frac{B+G}{2} = \left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}\right) \tag{1.1}$$

- Step 2: the function will decrease if we move from W to B and from W to G. So it is possible the function will have a smaller value at R, where R is the reflected point of W through the centroid (Fig. 1.1).

$$R = (1+\alpha)M - \alpha W = 2M - W \tag{1.2}$$

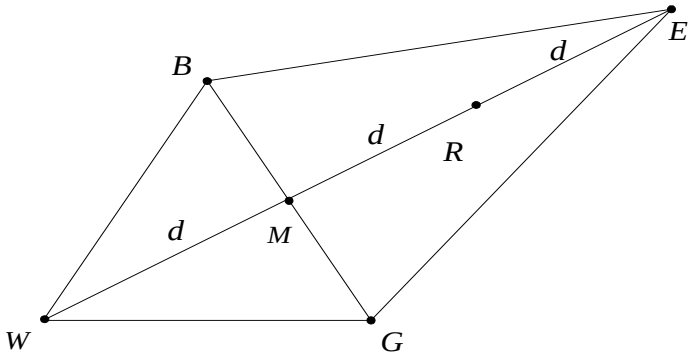

Fig. 1.1: The triangular simplex ΔBGW with midpoint M, reflected point R and extended point E

- Step 3: if the function value at *R* is smaller than the function value at *W*, it means that the simplex is moving in the right direction and the new better simplex ΔBGR is created. At this stage, we can extend the line segment from *M* through *R* to *E* (Fig. 1.1). If the function value at *E* is smaller than the function value at *R*, the new simplex ΔBGE is selected.

$$E = \gamma R + (1-\gamma)M = 2R - M \tag{1.3}$$

- Step 4: if the function value at *R* is greater than the function value at *W*, another point must be tested. The contracted points C_1, C_2 which are the midpoints of the line segments $\overline{WM}$ and $\overline{MR}$ can be considered in this case. The new simplex ΔBGC (Fig. 1.2) will be formed if the function value at *C* is smaller than the function value at *W* (*C*: better point between C_1 and C_2).

$$C_1 = \beta W + (1-\beta)M = \frac{W+M}{2} \text{ or } C_2 = \frac{M+R}{2} \tag{1.4}$$

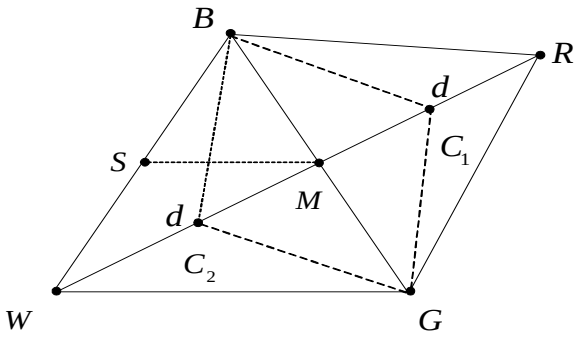

Fig. 1.2: Contracted points C_1 and C_2, shrinking points *S* and *M* toward *B*

- Step 5: if the function value at *C* is not less than the function value at *W*, the points *G* and *W* will be shrunk toward *B*. *G* will move to *M* and *W* will move to *S* which are the midpoints of the line segments $\overline{BG}$ and $\overline{BW}$ respectively (Fig. 1.2).

- Step 6: a new vertex is found to replace the worst vertex *W* iteratively. The algorithm will repeat from step 1 until a desired minimum is found.

Compared with gradient methods, the simplex method is simpler in term of mathematic computation, which is normally more complicated to calculate derivatives and requires more computing cost as well. Unlike the genetic algorithm or the differential evolution algorithm, there is no operation of mutation or crossover in this algorithm. In each iteration, only one new vertex is computed; therefore, it converges much faster. These advantages are key features which motivate authors of this paper to improve the simplex algorithm and make it become a useful optimization tool for engineers, scientists, etc. in many different types of applications [22], [23].

The simplex method is a direct search algorithm. Its computational process is simple and does not require the calculation of derivatives [24]. However, the simplex method without gradients may lead its convergence process in wrong directions. This makes it become unreliable in optimization. This scenario usually happens and remarkably reduces the efficiency of the simplex method in solving high dimensional problems. This dissertation will give some new insights on why the simplex method becomes inefficient in high dimensions if not using the gradient information [25]-[27]. To improve its convergence speed and success rate, the simplex method can be incorporated with other techniques. This dissertation will particularly present how to improve it by combining with quasi gradient methods to define a new way to search for its moving directions reliably. The improved algorithm which does not require complex mathematic computations can optimize multidimensional problems with higher success rate and faster convergence speed.

Chapter 2

Improved Simplex Method with Quasi Gradient Methods

2.1 Deficiency of Nelder Mead's Simplex Method

Although the simplex method was proposed a long time ago (1965) [3] and has not had much success in optimizing large scale problems [28], it is still a method of choice because of its simplicity. As a matter of fact, its necessary improvement of convergence speed and convergence rate is still an attractive research topic in the area of computing and optimization. For this purpose, many authors have proposed different ideas to address this issue. Fuchang Gao and Lixing Han [29] proposed an implementation of the simplex method in which the expansion, contraction, and shrinking parameters depend on the dimension of optimization problems. Another author - Torczon [30], suggested that this poor convergence may be due to the search as direction becomes increasingly orthogonal to the steepest descent direction, etc. Without any satisfactory convergence theory, there is no doubt that the effect of dimensionality should be extended and investigated more. Clearly, this is one of the main reasons restricting its convergence capability. This dissertation is another effort to improve the simplex algorithm with two simple solutions, which are different from other explanations in the literature. Furthermore, the simplicity is also the main goal of authors to keep this algorithm robust and different from other optimization algorithms.

As presented shortly in the overview, the simplex algorithm converges based on the

formation of the geometric simplex and its movement to find local minima. During optimization, this algorithm assumes that the direction to local minima can be found by the operations of reflection, contraction, and expansion without caring about the gradient. In other words, the dynamic change of a geometric simplex through these operations is utilized to approximate better vertices along the gradient direction. However, this assumption is not always true in reality, and that explains why the simplex algorithm fails easily with high dimensional optimization problems. Instead of calculating the reflected vertex as the original algorithm proposed, a new way is presented in the next two paragraphs by combining it with two different quasi gradient methods respectively. These two quasi gradient methods can be assumed as two approaches to approximate gradients by using numerical analysis rather than analytical analysis. With this modification, the improved algorithm converges much faster and more reliably than the original one.

To illustrate this reasoning we can consider two extreme cases where the simplex method may not converge to local minima (using 2-d cases for easy illustration). These two cases with the locations of B (best), G (good), W (worst) points have significantly different gradient directions. In the case (a) Fig. 2.1a the function values at W and G are similar while in the case (b) Fig. 2.1b the function values at B and G are similar. In both cases, the gradient heads to different directions from the simplex method. According to this algorithm, the simplex ΔBGW will start to reflect and search in MR direction first. Once it cannot find a better vertex in this direction, this simplex starts to shrink and a new simplex ΔBSM is created and continues searching in the same direction M_1R_1, which maybe not the right direction to minima. This illustration clearly shows why the simplex method does not have enough the capability to search for its moving directions just by using its simple geometrical movement. This also explains why

this algorithm is not stable in optimizing multi-dimensional problems and mostly fails to optimize large scale problems, or converges very slowly. In order to improve its speed and convergence rate, it needs to rely on the gradient. With a new way to calculate the reflected point according to quasi gradient methods, a new simplex *ΔBGR'* is created instead of *ΔBGR*.

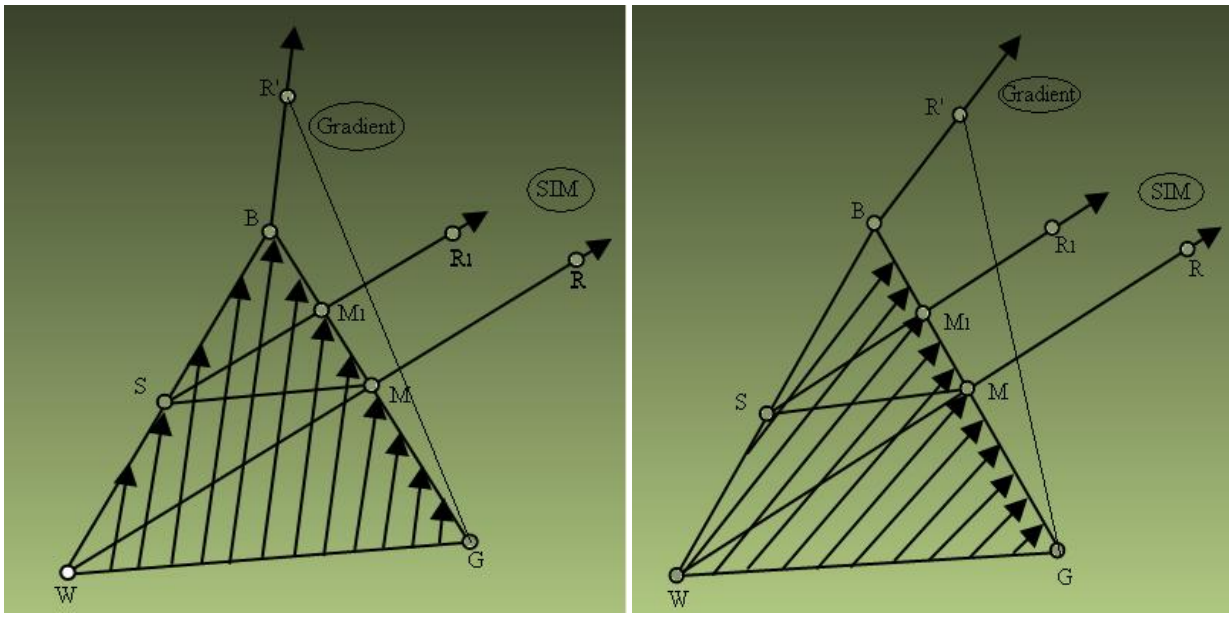

Fig. 2.1: The triangular simplex *ΔBGW* with similar function values at *W* and *G* (case (a)) and the triangular simplex *ΔBGW* with similar function values at *B* and *G* (case (b))

2.2 Quasi Gradient Methods

To maintain the simplicity, two quasi gradient methods are presented to approximate gradients [31]. The first method uses an extra vertex in a simplex. Its accuracy depends on the linearity of a function in the vicinity of a simplex. However, its computing cost does not increase significantly when the size of optimization problems becomes larger. The second method uses a hyper plane equation formed from a simplex. This method can estimate gradients more accurately; therefore, it converges faster. However, its high computing cost of inverse matrixes does not have much advantage with the large size of optimization problems.

2.2.1 Quasi Gradient Method Using an Extra Vertex

This method approximates gradients of a (*n+1*) dimensional plane created from a geometrical simplex. First, it selects an extra vertex composed from (*n+1*) vertices in a simplex and then combines this vertex with other n selected vertices in the same simplex to estimate gradients. Its steps are presented as following:

Assume an optimized function f: $\mathbb{R}^n \rightarrow \mathbb{R}$, $x \in \mathbb{R}^n$

- Step1: initialize a simplex with *(n+1)* random vertices $x_1, x_2, \ldots, x_n$
- Step 2: select an extra vertex x_E with its coordinates composed from *n* vertices in the simplex. In other words, coordinates of the selected vertex are a diagonal of the matrix *X* from *n* vertices in the simplex.

$$x_E = diag\begin{bmatrix} x_{1,1}\, x_{1,2} \ldots x_{1,n-1}\, x_{1,n} \\ x_{2,1}\, x_{2,2} \ldots x_{2,n-1}\, x_{2,n} \\ \ldots\ldots\ldots\ldots\ldots\ldots \\ x_{n,1}\, x_{n,2} \ldots x_{n,n-1}\, x_{n,n} \end{bmatrix}_{nxn} \tag{2.1}$$

$$or\ x_E = \left[x_{1,1}, x_{2,2}, \ldots, x_{n,n}\right] \tag{2.2}$$

- Step 3: approximate gradients based on the extra vertex *E* with other *n* vertices in the selected simplex.

For i =1: n,
  If mod (i, 2) = 0
    $g_i = \dfrac{\partial f}{\partial x_i} = \dfrac{f(i-1) - f(x_E)}{x_{i-1,i} - x_{E,i}}$
  Else
    $g_i = \dfrac{\partial f}{\partial x_i} = \dfrac{f(i+1) - f(x_E)}{x_{i+1,i} - x_{E,i}}$
  End
End

(2.3)

To illustrate how this method works, a 2-d case with f: $\mathbb{R}^2 \rightarrow \mathbb{R}$ and $x, y \in \mathbb{R}^2$ which has a triangular simplex ΔBGW with B (best), G (good), W (worst) vertices is shown in Fig. 2. 2. E is the extra vertex which has its coordinates formed from B and G. Then the approximate gradient of this plane will be:

$$g_1 = \frac{\Delta f}{\Delta x} = \frac{f_B - f_E}{x_2 - x_1};\ g_2 = \frac{\Delta f}{\Delta y} = \frac{f_G - f_E}{y_1 - y_2} \tag{2.4}$$

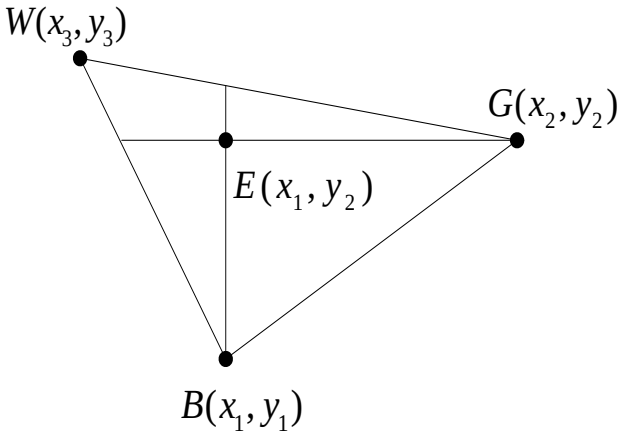

Fig. 2.2: The simplex ΔBGW with extra vertex E

- Step 4: calculate the new reflected vertex R' based on the best vertex B and the approximate gradients Fig. 2.1. Parameter σ is the learning constant or step size.

$$x_{R'} = x_B - \sigma * G \tag{2.5}$$

- Step 5: if the function value at R' is smaller than the function value at B, it means that BR' is the right direction of the gradient. Then R' can be expanded to E'.

$$x_{E'} = (1-\gamma)x_B + \gamma x_{R'} \tag{2.6}$$

2.2.2 Quasi Gradient Method Using a Hyper Plane Equation

This quasi gradient method forms a *(n+1)*-dimensional plane from *(n+1)* vertices in a simplex and then uses matrix calculations to approximate gradients. This method can be described as follows:

Assume an optimized function f: $\mathbb{R}^n \rightarrow \mathbb{R}$, $x \in \mathbb{R}^n$

- Step 1: initialize a simplex with *(n+1)* random vertices $x_1, x_2, \ldots, x_{n+1}$
- Step 2: a *(n+1)*-dimensional hyper plane formed from this simplex is assumed to have the approximate equation

$$V = a_0 + a_1 x_1 + a_2 x_2 + \ldots + a_{n-1} x_{n-1} + a_n x_n \tag{2.7}$$

- Step 3: substitute each vertex into the hyper plane equation eq. 2.7, so there will be *(n+1)* equations:

$$\begin{aligned} V_1 &= a_0 + a_1 x_{1,1} + a_2 x_{1,2} + \ldots + a_{n-1} x_{1,n-1} + a_n x_{1,n} \\ V_2 &= a_0 + a_1 x_{2,1} + a_2 x_{2,2} + \ldots + a_{n-1} x_{2,n-1} + a_n x_{2,n} \\ &\ldots \\ V_{n+1} &= a_0 + a_1 x_{n+1,1} + a_2 x_{n+1,2} + \ldots + a_{n-1} x_{n+1,n-1} + a_n x_{n+1,n} \end{aligned} \tag{2.8}$$

where $V_i = f(x_{i,1}, x_{i,2}, \ldots, x_{i,n})$, $i = 1:n+1$

- Step 4: calculate the approximate gradient matrix by writing the above multi-equations in the matrix form $G = X^{-1} * V$.

$$X^{-1} = \begin{bmatrix} 1 & 1 & \ldots & 1 \\ x_{1,1} & x_{2,1} & \ldots & x_{n+1,1} \\ x_{1,2} & x_{2,2} & \ldots & x_{n+1,2} \\ \ldots & & & \\ x_{1,n} & x_{2,n} & \ldots & x_{n+1,n} \end{bmatrix}^{-1}_{n+1 \times n+1} \tag{2.9}$$

$$V = \begin{bmatrix} V_1 \\ V_2 \\ \dots \\ V_{n+1} \end{bmatrix}_{n+1\times 1} \tag{2.10}$$

$$G = \begin{bmatrix} a_0 \; a_1 \; a_2 \dots a_{n-1} \; a_n \end{bmatrix}_{1\times n+1}, \frac{\partial V}{\partial x_i} = a_i \tag{2.11}$$

- Step 5: calculate the new reflected vertex *R'* according to eq. 2.5.
- Step 6: calculate the new expanded vertex *E'* according to eq. 2.6.

Two quasi gradient methods without using derivatives presented above are much simpler than analytical gradient methods. These two methods do not require the calculation of derivatives. Instead they just approximate gradients without concerning the shape of a function. The first quasi gradient method using an extra vertex does not require high computing cost but its accuracy depends on the linearity of optimized functions in the vicinity of a simplex. Therefore, it converges slower than the second method using a hyper plane equation (comparisons of two methods are presented in the next section and summarized in Tables 2.4-2.7). The second method's computing power is reduced significantly when the scale of optimization problems becomes larger because its computing time grows proportional to the square of the problem size n^2 while the first method's computing cost is proportional to n.

The improved algorithm with quasi gradient search is similar to the original simplex method except that it has to approximate gradients to search for its reflected vertex. In other words, its convergence will rely on the gradient direction through the new reflected vertex *R'* rather than the reflected vertex *R* calculated through the centroid vertex proposed by the original simplex algorithm. The improved simplex method with quasi gradient search can be applied successfully in synthesizing lossy filters, and training artificial neural networks, etc [32].

2.3 Testing Functions

All algorithms are tested on a set of functions with different levels of complexity. These functions are well-known unconstrained optimization functions in literature. A large number of problems are relatively adequate to prove the reliability and robustness of these algorithms. It also warrants that the improved algorithm is much better than the original algorithm overall, not just better than a small set of problems.

Algorithms are tested on a wide range not close to solutions to address on their convergence rate. Therefore, it is much more satisfactory when starting points are generated randomly. Unlike other publications in literature, we do not use the standard starting points for a certain function to test algorithms because it is fairly hard to measure their reliability and robustness, or to differentiate between similar algorithms in this case. The use of initial points farther away from solutions frequently reveals dramatic differences of algorithms as success rate, computing time, etc.

To evaluate the ability to solve problems, we measure these algorithms in terms of their success rate and computing time.

List of benchmark functions:

(1) De Jong function [3], [33]-[35]

$$F_1 = \sum_{i=1}^{n} x_i^2$$

(2) De Jong function with moved axis [36]

$$F_2 = \sum_{i=1}^{n} (x_i - a_i)^2$$

(3) Quadruple function [37]

$$F_3(x) = \sum_{i=1}^{n} \left(\frac{x_i}{4} \right)^4$$

(4) Powell function [3], [34]

$$F_4 = \sum_{i=1}^{n} [\, (x_i + 10x_{i+1})^2 + 5(x_{i+2} - x_{i+3})^2 + (x_{i+1} - 2x_{i+2})^4 + 10(x_i - x_{i+3})^4]$$

(5) Moved axis Parallel hyper-ellipsoid function [36]

$$F_5 = \sum_{i=1}^{n} i x_i^2$$

(6) Zarakov function [38]

$$F_6 = \sum_{i=1}^{n} [x_i^2 + \left(\sum_{i=1}^{n} 0.5 i x_i \right)^2 + \left(\sum_{i=1}^{n} 0.5 i x_i \right)^4]$$

(7) Schwefel function [38]

$$F_7 = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j \right)^2$$

(8) Sum of different power function [36]

$$F_8 = \sum_{i=1}^{n} |x_i|^{i+1}$$

(9) Step function [33]

$$F_9 = \sum_{i=1}^{n} |x_i + 0.5|^2$$

(10) Box function [36]

$$F_{10} = \sum_{i=1}^{n} \left[e^{-a x_i} - e^{-a x_{i+1}} - x_{i+2} (e^{-a} - e^{-10a}) \right]^2$$

where $a = t*i$, t is a constant

(11) Rosenbrock function [1], [33], [34], [39]

$$F_{11} = \sum_{i=1}^{n} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2 \right]$$

(12) Biggs Exp6 function [6], [34]

$$F_{12} = \sum_{i=1}^{n} [\, x_{i+2} e^{t_i x_i} - x_{i+3} e^{-t_i x_{i+1}} + x_{i+5} e^{t_i x_{i+4}} - y_i \,]^2$$

where $t_i = 0.5i$ and $y_i = e^{-t_i} - 5e^{-10t_i} + 3e^{-4t_i}$

(13) Kowalik and Osborne function [33], [34], [39]

$$F_{13} = \sum_{i=1}^{n} \left[a_i - \frac{x_i \left(b_i^2 + b_i x_{i+1} \right)}{b_i^2 + b_i x_{i+2} + x_{i+3}} \right]^2$$

where a, b are vectors in Table 2.1

i	a_i	b_i
1	0.1975	4.0000
2	0.1947	2.0000
3	0.1735	1.0000
4	0.1600	0.5000
5	0.0844	0.2500
6	0.0627	0.1670
7	0.0456	0.1250
8	0.0342	0.1000
9	0.0323	0.0833
10	0.0235	0.0714
11	0.0246	0.0625

Table 2.1: Coefficients of Kowalik and Osborne function

(14) Colville function [33]

$$F_{14} = \sum_{i=1}^{n} [\, 100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2 + (x_{i+2} - 1)^2 + 10.1((x_{i+1} - 1)^2 + (x_{i+3} - 1)^2) + 90(x_{i+2}^2 - x_{i+3})^2 + 19.8(x_{i+1} - 1)(x_{i+3} - 1)]$$

(15) Wood function [6], [34], [39]

$$F_{15} = \sum_{i=1}^{n} [\, 100\left(x_{i+1} - x_i^2\right)^2 + \left(1 - x_i\right)^2 + 90(x_{i+3} - x_{i+2}^2)^2 + (1 - x_{i+2})^2 + 10.1((1 - x_{i+1})^2 + (1 - x_{i+3})^2) + 19.8(1 - x_{i+1})(1 - x_{i+3})]$$

(16) Bard function [6], [33]

$$F_{16} = \sum_{i=1}^{n} \left[y_i - x_i + \frac{u_i}{v_i x_{i+1} + w_i x_{i+2}} \right]^2$$

where y, u, v, w are vectors in Table 2.2

i	y_i	u_i	v_i	w_i
1	0.14	1	15	1
2	0.18	2	14	2
3	0.22	3	13	3
4	0.25	4	12	4
5	0.29	5	11	5
6	0.32	6	10	6
7	0.35	7	9	7
8	0.39	8	8	8
9	0.37	9	7	7
10	0.58	10	6	6
11	0.73	11	5	5
12	0.96	12	4	4
13	1.34	13	3	3
14	2.10	14	2	2
15	4.39	15	1	1

Table 2.2: Coefficients of Bard Functions

2.4 Experimental Results

With this significant improvement, the improved simplex method can be a useful optimization tool to replace for other popular algorithms as the genetic algorithm or the particle swarm algorithm, etc. All evaluated algorithms are written in Matlab, and all experiments are tested on a PC with Intel Quad. In order to compare performances of these algorithms, some assumptions are set: algorithms start with a random initial variables in the range of [-100, 100]; dimension of all benchmark problems is 20; maximum iteration is equal to 100,000; desired error predefined to terminate algorithms is equal to 0.001; coefficients of the simplex method $\alpha = 1$, $\beta = 0.5$, $\gamma = 2$, learning constant $\sigma = 1$. In addition, the genetic algorithm, the differential evolution algorithm, and the particle swarm optimization each has 20 members in its population. These algorithms use the same default values as the ones written in the standard Matlab toolboxes. Because of timing cost to test the genetic algorithm and the differential evolution algorithm, all results in Table 2.3 are the average values calculated over 25 random running times.

	GA		**DE**		**PSO**		**SIM1**	
Optimization Function	Success rate	Comp. time (s)	Success rate	Comp. time (s)	Success rate	Comp. time (s)	Success rate	Comp. time (s)
De Jong	44%	153.435	100%	111.45	96%	5.5137	100%	0.3450
De Jong with moved axis	60%	152.184	100%	122.30	Failure		100%	0.4450
Quadruple	100%	46.6543	80%	117.85	100%	0.0840	100%	0.2541
Powell	Failure		10%	173.76	100%	4.5735	100%	1.1455
Moved axis Parallel hyper-ellipsoid	Failure		85%	123.50	96%	9.7553	100%	0.7254
Zarakov	Failure		Failure		96%	5.2152	100%	1.1444
Schwefel	Failure		Failure		32%	89.9445	100%	0.9587
Sum of different power	100%	141.422	Failure		100%	0.06425	100%	1.7978
Step	35%	166.335	95%	121.09	80%	23.1161	100%	0.3992
Rosenbrock	Failure		Failure		Failure		54%	10.107
Biggs Exp6	Failure		10%	160.05	4%	126.276	27%	9.0644
Colville	Failure		Failure		Failure		40%	5.6746
Wood	Failure		Failure		Failure		44%	7.7333

Table 2.3: Evaluation of success rate and computing time of 20-dimensional function

In this simulation, it is unnecessary to verify how the dimensionality affects the simplex algorithm. Therefore, one experiment with 20 dimensions is conducted. And only the algorithm using an extra vertex (SIM1) is selected to compare because of its simple computations relative to other derivative free optimization algorithms.

- *The Genetic Algorithm*: There are only two cases, the genetic algorithm can obtain 100% success rates. Although the genetic algorithm cannot obtain high success rate as the improved simplex method, but it still shows its convergence ability in 5 out of 13 problems. However, this algorithm cannot converge as fast as the improved simplex method or the particle swarm optimization does. There are 8 out of 13 problems, the genetic algorithm cannot converge. Even the data is not displayed in Table 2.3, but the genetic algorithm shows its convergence trend in this experiment if a number of generations is increased. Among four derivative free optimization algorithms discussed here, the genetic algorithm is the slowest one. This can be explained by its complex mutation and crossover operations.
- *The Differential Evolution Algorithm*: This algorithm does not converge really well with these optimization functions. There are five cases, it converges with high success rates. And there are two cases, it converges with low success rates. It cannot converge nearly half of problems. In order to optimize these functions, this algorithm needs more iterations. Therefore, it will take longer to solve the same problems. The differential evolution algorithm converges faster than the genetic algorithm, but it is much slower than the improved simplex algorithm.
- *The Particle Swarm Algorithm*: The particle swarm algorithm converges relatively

fast; however, it does not have enough consistency. There are four cases, it fails to converge with no matter of a number of generations. There is one case that it converges with a really low success rate. Even it converges more than half of cases, but it is still relatively slower and has lower success rate than the improved simplex method. However, the particle swarm algorithm is much faster than the differential evolution algorithm and the genetic algorithm in term of convergence speed.

- *The Improved Simplex Method*: From the experimental results in Table 2.3, we can conclude that the improved simplex method converges much faster and more efficiently than the genetic algorithm, the differential evolution algorithm, and the particle swarm algorithm in local minimum optimization. This is reflected through its higher success rate and less computing time for each testing function. It can get optimum solutions more than 75% of problems with 100% success rate. In other 25% of problems, its success rates are around 50%, but it is still much better than other algorithms. In term of computing time, this algorithm particularly outclasses the others. Its convergence speed is at least from ten to hundred times faster.

Obviously, the improved simplex algorithm with this significant modification can be an alternative tool to replace efficiently for other optimization tools. This algorithm is a direct search method which is free of derivative calculation. Therefore, it can converge much faster and higher success rate than algorithms based on evolutionary computations such as the genetic algorithm, etc.

The next experiments are conducted with the same assumptions as the last one: algorithms start with a random initial simplex in the range of [-100, 100]; dimensions of all benchmark problems are equal to 10, 15, and 20 respectively; maximum iteration is equal to

100,000; target error predefined to terminate algorithms is equal to 0.001; coefficients α= 1, β= 0.5, γ= 2, learning constant σ= 1. All results in Table 2.4-2.7 are average values calculated over 100 random running times. Figures 2.3- 2.10 are error curves of algorithms plotted for 10 dimensions.

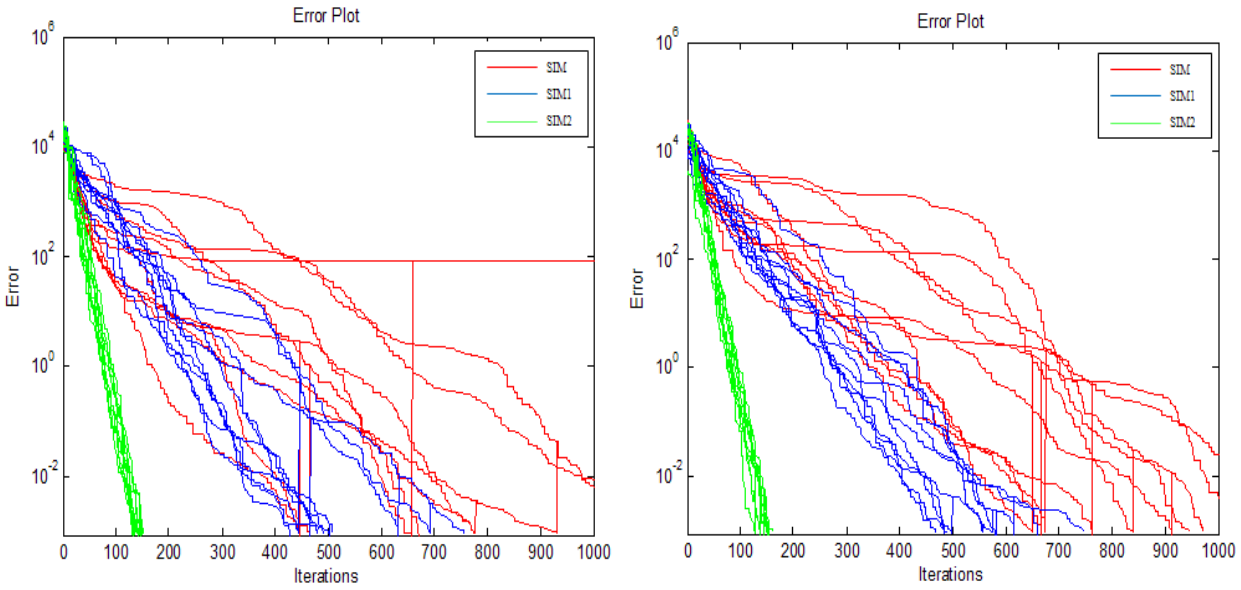

Fig. 2.3: Simulated error curves of De Jong function 1(a) and De Jong function 1 with moved axis (b)

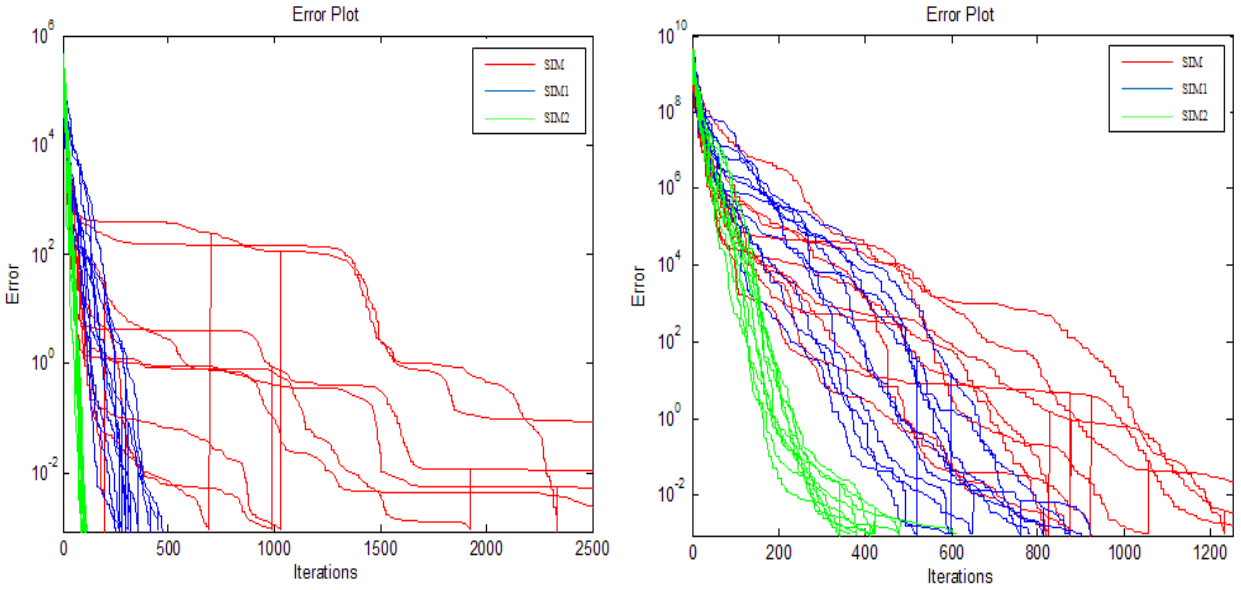

Fig. 2.4: Simulated error curves of Quadruple function(a) and Powell function (b)

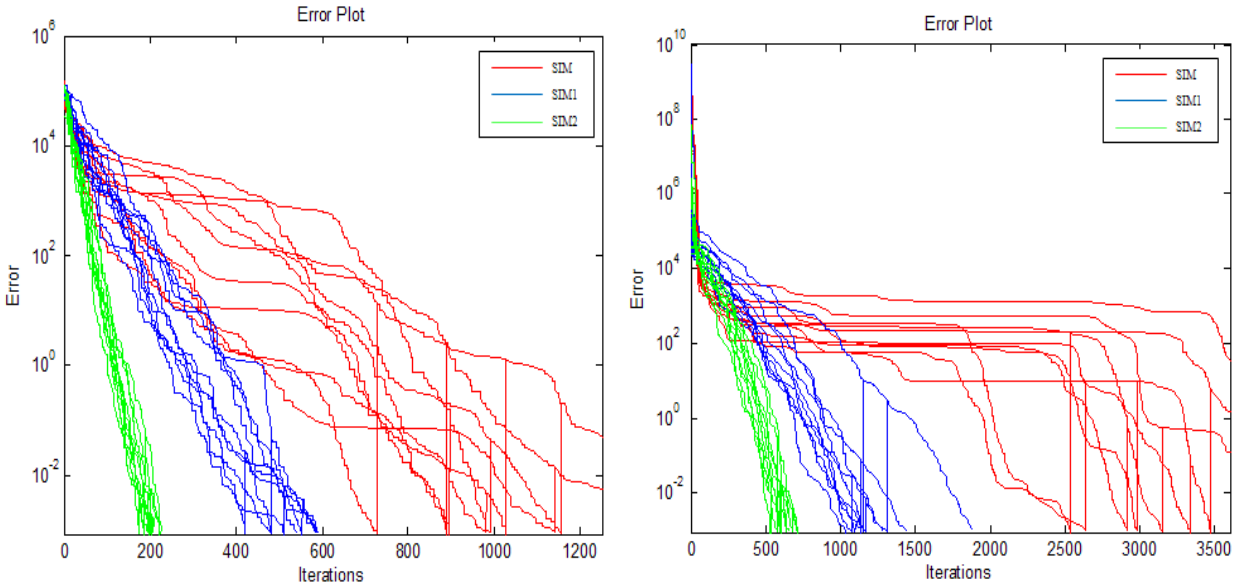

Fig. 2.5: Simulated error curves of Parallel hyperellipsoid function (a) and Zakarov function (b)

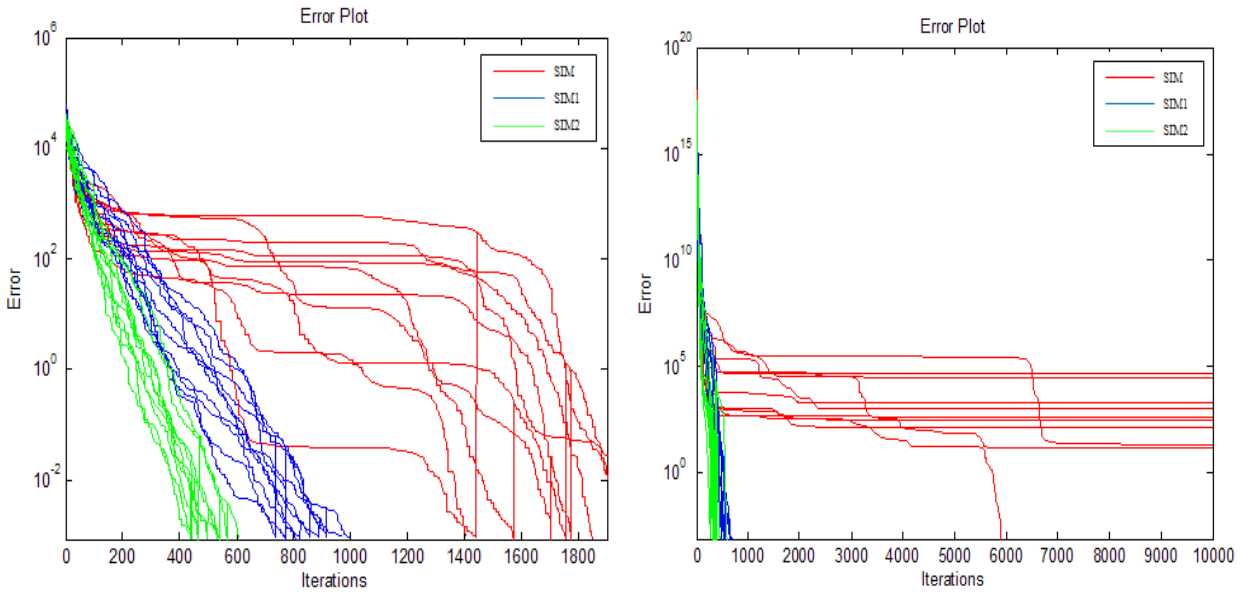

Fig. 2.6: Simulated error curves of Schwefel function (a) and sum of different power function (b)

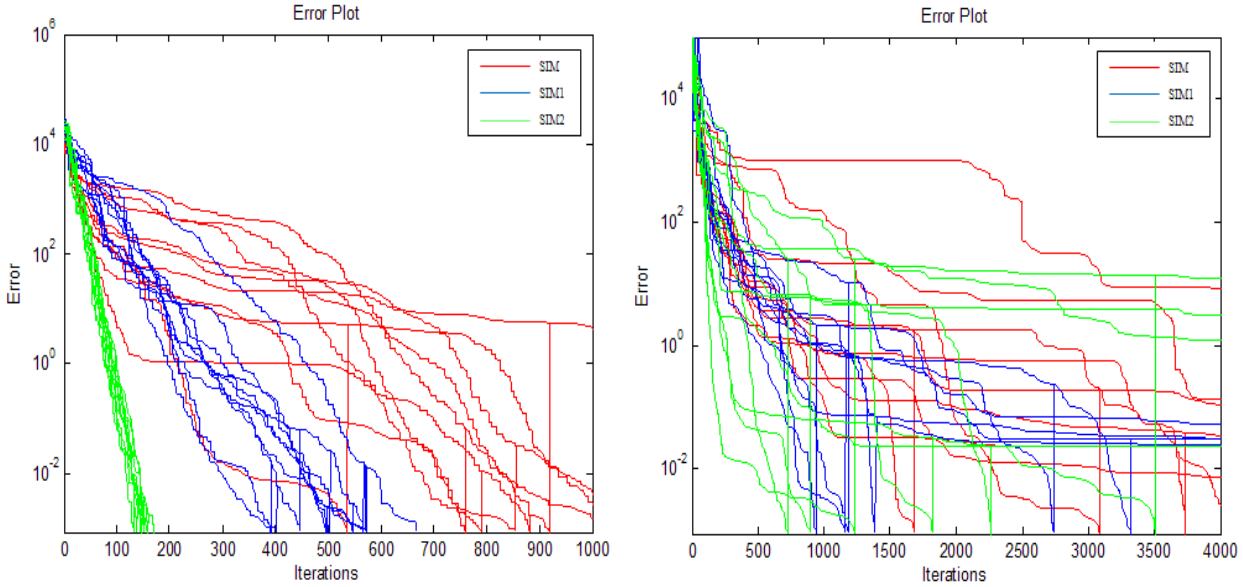

Fig. 2.7: Simulated error curves of Step function (a) and Box function (b)

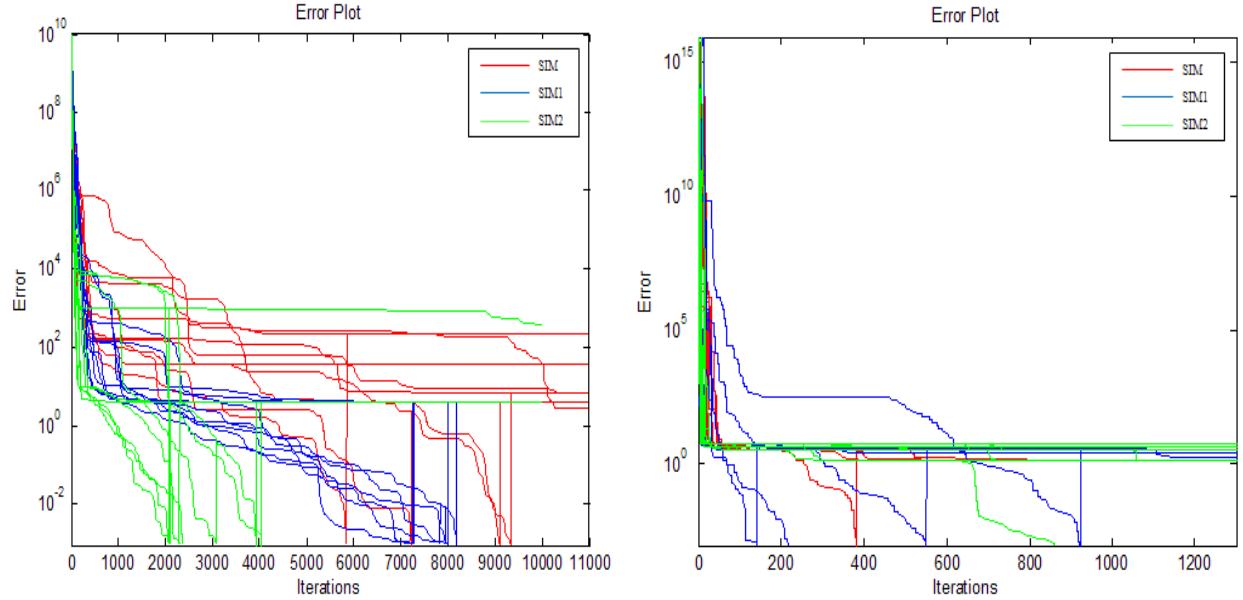

Fig. 2.8: Simulated error curves of Rosenbrock function (a) and Biggs Exp6 function (b)

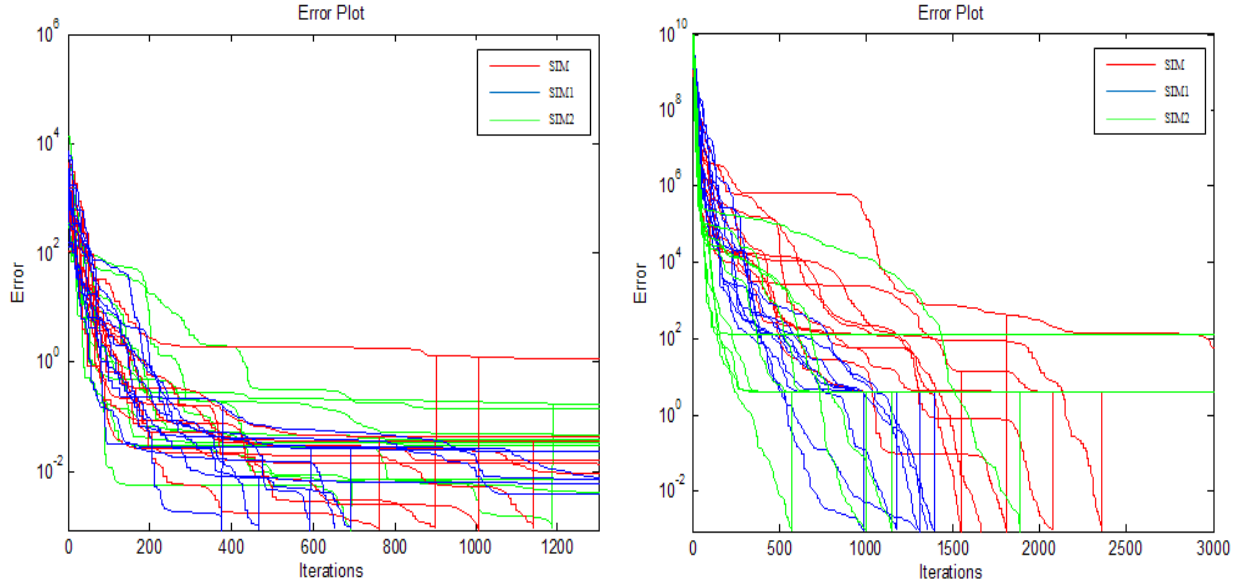

Fig. 2.9: Simulated error curves of Kowalik Osborne function (a) and Colville function (b)

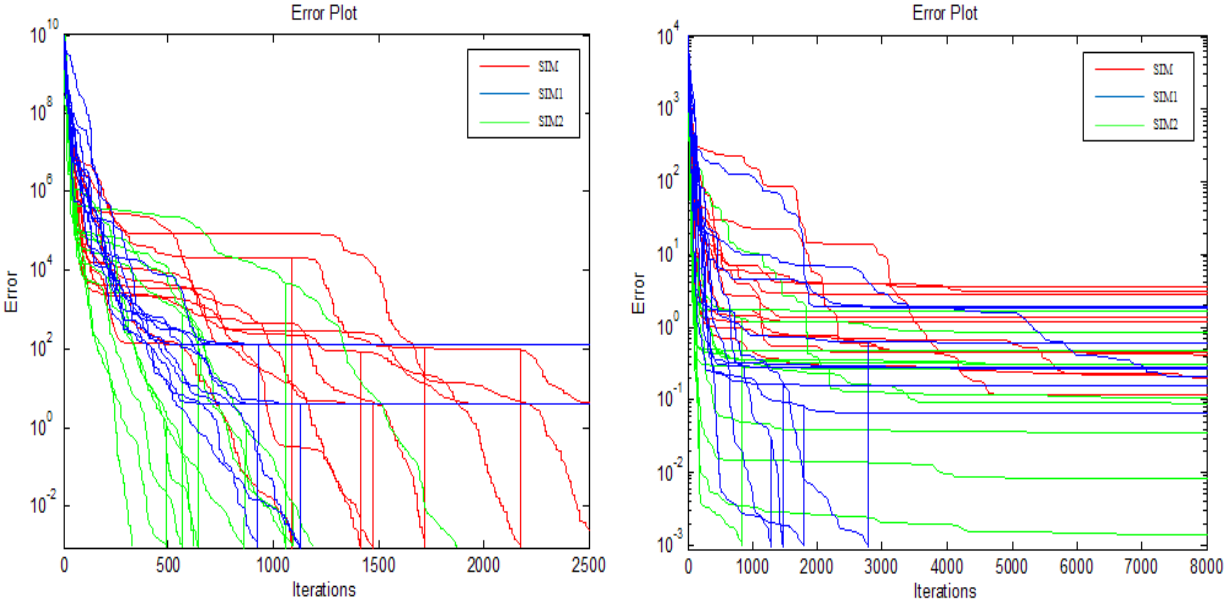

Fig. 2.10: Simulated error curves of Wood function (a) and Bard function (b)

Optimization Function	**Nelder Mead's simplex algorithm**		**Improved simplex algorithm using an extra vertex**		**Improved simplex algorithm using a hyper plane equation**	
	Success rate	Comp. time (s)	Success rate	Comp. time (s)	Success rate	Comp. time (s)
De Jong	100%	0.0907	100%	0.0806	100%	0.0474
De Jong with moved axis	100%	0.0932	100%	0.0824	100%	0.0477
Quadruple	100%	0.2183	100%	0.0539	100%	0.0350
Powell	100%	0.1037	100%	0.1346	100%	0.1444
Moved axis Parallel hyper-ellipsoid	100%	0.1102	100%	0.0841	100%	0.0648
Zarakov	99%	0.3195	100%	0.1879	100%	0.1766
Schwefel	100%	0.1666	100%	0.1400	100%	0.1420
Sum of different power	26%	1.1012	100%	0.1150	100%	0.1232
Step	100%	0.0863	100%	0.0825	100%	0.0475
Box	65%	0.3321	81%	0.3636	81%	0.5761
Rosenbrock	55%	0.8812	76%	1.2094	82%	1.5916
Biggs Exp6	52%	0.1118	60%	0.1244	20%	0.3509
Kowalik and Osborne	48%	0.2828	76%	0.4913	48%	5.0087
Colville	46%	0.2026	50%	0.2081	60%	0.2077
Wood	41%	0.2042	52%	0.2038	58%	0.2155
Bard	16%	0.7240	48%	0.9095	13%	2.2250

Table 2.4: Evaluation of success rate and computing time of 10-dimensional functions

Optimization Function	**Nelder Mead's simplex algorithm**		**Improved simplex algorithm using an extra vertex**		**Improved simplex algorithm using a hyper plane equation**	
	Success rate	Comp. time (s)	Success rate	Comp. time (s)	Success rate	Comp. time (s)
De Jong	9%	0.2539	100%	0.2292	100%	0.0888
De Jong with moved axis	7%	1.0124	100%	0.2263	100%	0.0899
Quadruple	21%	0.9097	100%	0.1778	100%	0.0638
Powell	93%	1.5418	100%	0.4222	100%	0.3303
Moved axis Parallel hyper-ellipsoid	2%	0.6789	100%	0.2358	100%	0.1540
Zarakov	Failure		100%	0.5598	100%	0.5123
Schwefel	2%	1.1187	100%	0.3942	100%	0.3948
Sum of different power	Failure		100%	0.3303	100%	0.3300
Step	13%	0.4213	100%	0.1927	100%	0.0910
Box	Failure		19%	1.8882	4%	4.2749
Rosenbrock	Failure		55%	3.9881	80%	3.3474
Biggs Exp6	4%	0.7417	60%	1.4860	3%	2.5735
Kowalik and Osborne	-		-		-	
Colville	10%	1.9298	53%	0.5028	52%	0.5123
Wood	12%	2.3841	52%	0.5078	61%	0.5003
Bard	Failure	0.7240	11%	3.6462	Failure	2.2250

Table 2.5: Evaluation of success rate and computing time of 15-dimensional functions

Optimization Function	**Nelder Mead's simplex algorithm**		**Improved simplex algorithm using an extra vertex**		**Improved simplex algorithm using a hyper plane equation**	
	Success rate	Comp. time (s)	Success rate	Comp. time (s)	Success rate	Comp. time (s)
De Jong	Failure		100%	0.4529	100%	0.1538
De Jong with moved axis	Failure		100%	0.4188	100%	0.1565
Quadruple	3%	1.7213	100%	0.3212	100%	0.1124
Powell	Failure		100%	0.8969	100%	0.6518
Moved axis Parallel hyper-ellipsoid	Failure		100%	0.4340	100%	0.3289
Zarakov	Failure		100%	1.3628	100%	1.2251
Schwefel	Failure		100%	0.8041	100%	0.9506
Sum of different power	Failure		100%	0.7094	100%	0.6802
Step	Failure		100%	0.3207	100%	0.1541
Box	Failure		5%	2.1718	3%	8.8745
Rosenbrock	Failure		54%	6.0026	75%	6.4713
Biggs Exp6	Failure		27%	3.2283	Failure	
Kowalik and Osborne	-		-		-	
Colville	Failure		40%	1.1572	44%	1.1259
Wood	Failure		44%	1.1708	50%	1.0224
Bard	-		-		-	0.1538

Table 2.6: Evaluation of success rate and computing time of 20-dimensional functions

Optimization Function	**Nelder Mead's simplex algorithm**		**Improved simplex algorithm using an extra vertex**		**Improved simplex algorithm using a hyper plane equation**	
	Success rate	Comp. time (s)	Success rate	Comp. time (s)	Success rate	Comp. time (s)
De Jong	Failure		100%	2.9637	100%	0.8254
De Jong with moved axis	Failure		100%	2.6757	100%	0.9004
Quadruple	Failure		100%	2.0258	100%	0.6318
Powell	Failure		100%	5.0306	100%	4.3891
Moved axis Parallel hyper-ellipsoid	Failure		100%	2.4636	100%	3.3235
Zarakov	Failure		100%	13.072	100%	13.3271
Schwefel	Failure		100%	6.7592	100%	11.8561
Sum of different power	Failure		60%	11.340	100%	6.6748
Step	Failure		100%	2.4082	100%	0.8045
Box	Failure		1%	10.607	5%	20.9250
Rosenbrock	Failure		39%	43.649	72%	78.5312
Biggs Exp6	Failure		5%	36.843	Failure	
Kowalik and Osborne	-		-		-	
Colville	Failure		32%	7.8633	50%	8.2602
Wood	Failure		38%	7.375	49%	7.6051
Bard	-		-		-	

Table 2.7: Evaluation of success rate and computing time of 40-dimensional functions

The comparisons between the simplex algorithm and its improved versions are summarized above. In these simulations all algorithms are still compared in terms of the success rate and computing time. While the success rate is used to describe their reliability, computing time reflects how fast these algorithms can converge. Three different sizes of functions 10, 15, and 20 are also tested respectively for the purpose of comparing their robustness, and it is also used to verify how the simplex method is affected by its dimensionality. From Tables 2.4-2.7, we can draw a conclusion that the improved algorithm shows its better performance than the original simplex method in terms of both success rate and computing time. The experimental results also tell that the improved simplex method, using a hyper plane equation, converges faster than the one using an extra vertex in most cases. Even it requires more computations to approximate the gradient matrix. There are only two cases (Box function and Biggs Exp6 function), the method of a hyper plane equation shows its worse results than the method of an extra vertex. When the problem size increases, the simplex method starts getting worse and is unable to converge. In Table 2.4 of 10 dimensional functions, the simplex algorithm converges relatively well although it does not have a high success rate in several cases. However, these numbers are still good enough and acceptable because of its fast convergence. When the size increases to 15 in Table 2.5, its convergence rate suddenly drops down dramatically, and it totally fails in 20 dimensional problems or higher (Table 2.6-2.7); whereas, the improved algorithm still converges consistently well. It has 100% success in 9 out of 14 problems and over 40% success rate in 3 out of 14 problems. There is only one case of Box function that the improved algorithm cannot obtain a good success rate. Even with 20 dimensional problems, this algorithm is still able to converge very fast when its minimum and maximum computing time is less than 1(s) and 10(s) respectively. Comparing these two algorithms, we can conclude that the improved algorithm

using quasi gradients can define its moving direction more precisely. That is the reason why the improved algorithm converges much better. With the same random choice of initial vertices, the improved simplex method usually gets a higher convergence rate and less computing time than the original simplex method. Even this algorithm is combined with the quasi gradients, it does not face any difficulty to find function derivatives, and particularly its finding such derivatives is not time consuming as classical algorithms based on gradients.

Chapter 3

Synthesize Lossy Ladder Filters with Improved Simplex Method

3.1 Filter Synthesis Algorithms

There are many different types of filters such as the Butterworth filter, the Chebyshev filter, the Inverse Chebyshev filter and the Cauer Elliptic filter, etc. The characteristic responses of these classical filters are different. The Butterworth filter is flat in the stop-band, but it does not have a sharp transition from the pass-band to the stop-band. While the Chebyshev filter has a sharp transition from the pass-band to the stop-band, but it has ripples in the pass-band. Oppositely, the Inverse Chebyshev filter has the same characteristics as the Chebyshev filter, but it has ripples in the stop-band instead of the pass-band. The Cauer filter has ripples in both pass-band and stop-band; however, it has lower order [40], [41]. This section will summarize all steps to design low-pass filters by using these methodologies.

3.1. 1 Butterworth Low- pass Filter

Suppose ω_p, ω_s, α_p, α_s are the pass-band frequency, stop-band frequency, attenuation in pass-band, and attenuation in stop-band of a filter respectively. Depending on which method is used to synthesize, the frequency response of a filter will be different to meet these requirements. For an instance, a Butterworth filter will be designed as followings.

ω_p - pass-band frequency

ω_s - stop-band frequency

α_p - attenuation in pass-band

α_s - attenuation in stop-band

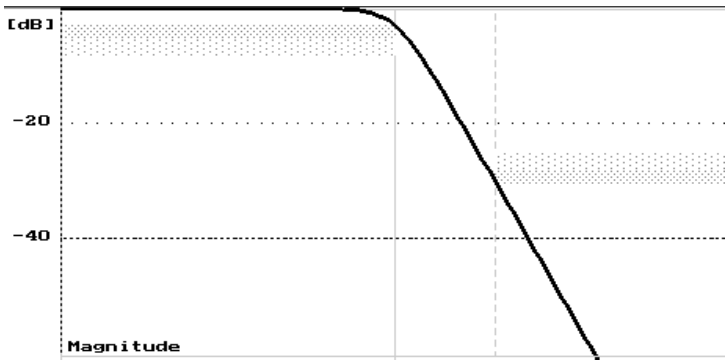

Fig. 3.1: Butterworth filter response

Butterworth filter response: $|T(j\omega)|^2 = \dfrac{1}{1+\dfrac{\omega^{2n}}{\omega_0^{2n}}}$ (3.1)

In order to simplify, we can summarize three basic steps to synthesize any type of low-pass filters. The first step is calculating the order of a low-pass filter. The second step is calculating poles and zeros of a low-pass filter. From hence its transfer function is derived. The third step is designing circuits to meet pole and zero locations; however, this part is another topic of analog filters so it will not be covered in this work [42]-[44].

All steps to design a Butterworth low-pass filter

- Step 1: calculate order of a filter

$$n = \frac{\log[(10^{\alpha_s/10}-1)(10^{\alpha_p/10}-1)]^{1/2}}{\log(\dfrac{\omega_s}{\omega_p})} \quad \text{(n needs to be roundup to integer value)} \quad (3.2)$$

- Step 2: calculate pole and zero locations

Angle if n is odd: $\Omega = \pm\dfrac{k180^0}{n}$; k= 0,1,...(n-1)/2 (3.3)

Angle if n is even: $\Omega = \pm(0.5+\dfrac{k}{n})180^0$; k= 0,1,...(n-2)/2 (3.4)

Normalized pole locations: $a_k = -\cos(\Omega); b_k = \pm\sin(\Omega); (\omega_0 = 1)$ (3.5)

$$\omega_0 = \frac{(\omega_p \omega_s)^{1/2}}{[(10^{\alpha_s/10} - 1)/(10^{\alpha_p/10} - 1)]^{1/(4n)}}; Q_k = \left| \frac{1}{2a_k} \right| \quad (3.6)$$

- Step 3: design circuits to meet pole and zero locations (not covered in this work)

3.1. 2 Chebyshev Low- pass Filter

ω_p - pass-band frequency

ω_s - stop-band frequency

α_p - attenuation in pass-band

α_s - attenuation in stop-band

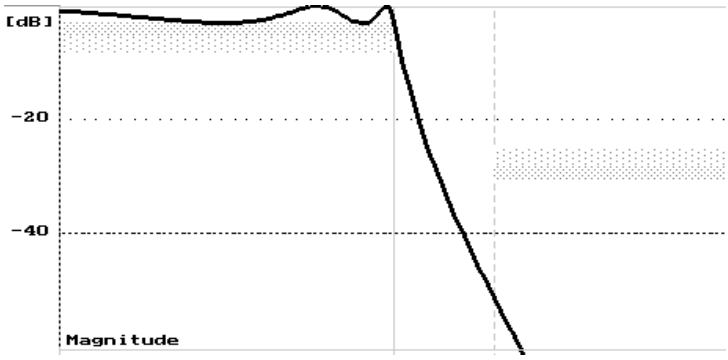

Fig. 3.2: Chebyshev filter response

Chebyshev filter response: $|T(j\omega)|^2 = \frac{1}{1 + \varepsilon^2 C_n^2(\omega)}$ (3.7)

All steps to design a Chebyshev low-pass filter

- Step 1: calculate order of a filter

$$n = \frac{\log[(10^{\alpha_s/10} - 1)(10^{\alpha_p/10} - 1)]^{1/2}}{\log(\frac{\omega_s}{\omega_p})} \text{ (n needs to be roundup to integer value)} \quad (3.8)$$

- Step 2: calculate pole and zero locations

$$\Omega = 90^0 + \frac{90^0}{n} + \frac{(k-1)180^0}{n} \quad (3.9)$$

$$\varepsilon = [10^{\alpha_p/10} - 1]^{1/2}; \gamma = \sinh^{-1}(\frac{1}{\varepsilon})/n \tag{3.10}$$

$$a_k = \sinh(\gamma)\cos(\Omega); b_k = \cosh(\gamma)\sin(\Omega); \omega_k = a_k{}^2 + b_k{}^2; Q_K = |\frac{\omega_k}{2a_k}| \tag{3.11}$$

- Step 3: design circuits to meet pole and zero locations (not covered in this work)

3.1. 3 Inverse Chebyshev Low- pass Filter

ω_p - pass-band frequency

ω_s - stop-band frequency

α_p - attenuation in pass-band

α_s - attenuation in stop-band

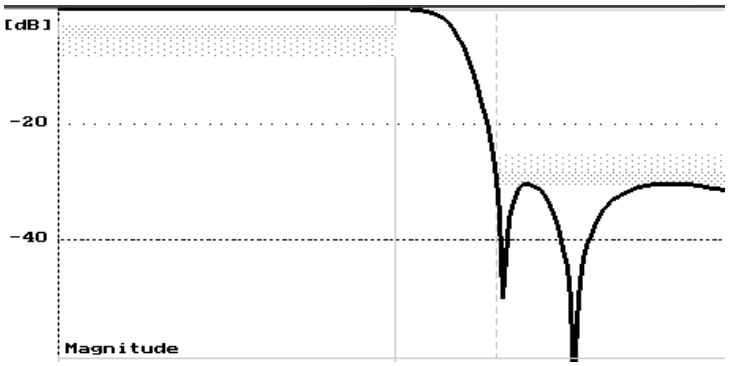

Fig. 3.3: Inverse Chebyshev filter response

Inverse Chebyshev filter response**:** $|T_{IC}(j\omega)|^2 = \dfrac{\varepsilon^2 C_n^2(1/\omega)}{1+\varepsilon^2 C_n^2(1/\omega)}$ (3.12)

The method to design an inverse Chebyshev low-pass filter is almost the same as the one to design a Chebyshev low-pass filter. It is just slightly different on account of the appearance of conjugate poles and zeros.

All steps to design an inverse Chebyshev low-pass filter

- Step 1: calculate order of a filter

$$n = \frac{\ln[4*(10^{\alpha_s/10}-1)/(10^{\alpha_p/10}-1)]^{1/2}}{\log[\frac{\omega_s}{\omega_p}+(\frac{\omega_s^{\,2}}{\omega_p^{\,2}}-1)^{1/2}]} \text{ (n needs to be roundup to integer value)} \quad (3.13)$$

- Step 2: calculate pole and zero locations

$$P_{ic} = \frac{1}{a_k + b_k}, \text{ find zeros } \omega_i = \frac{1}{\cos[\Pi * i/(2n)]}; \text{ i= 2k-1: 1, 3, 5….< np} \quad (3.14)$$

Notes: two conjugate poles on the imaginary axis

- Step 3: design circuits to meet pole and zero locations (not covered in this work)

3.1. 4 Cauer Elliptic Low- pass Filter

ω_p - pass-band frequency

ω_s - stop-band frequency

α_p - attenuation in pass-band

α_s - attenuation in stop-band

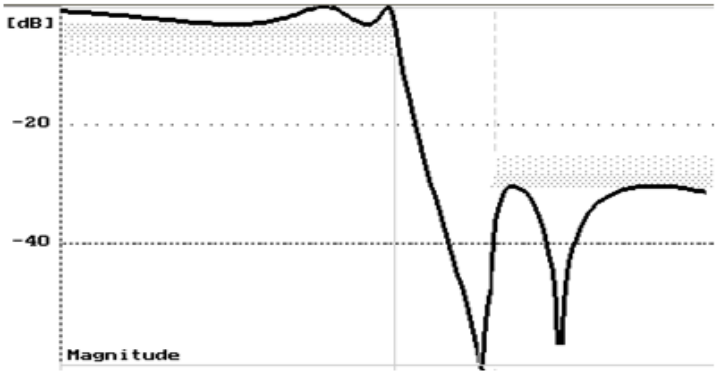

Fig. 3.4: Cauer Elliptic filter response

Cauer Elliptic filter response: $|T(jw)|^2 = \frac{1}{1+\varepsilon^2 R_n^2(w,L)}$ (3.15)

Designing a Cauer Elliptic filter is more complex than designing three previous filters. In order to calculate its transfer function, a mathematic process is summarized as below. Although the low-pass Cauer Elliptic filter has ripples in both stop-band and pass-band, but it has lower

order than the previous filters. In other words, it requires less hardware components for implementation. This is the main advantage of the Cauer Elliptic filter.

$$k = \frac{\omega_p}{\omega_s} \tag{3.16}$$

$$k' = \sqrt{1-k^2} \tag{3.17}$$

$$q_0 = \frac{0.5(1-\sqrt{k'})}{(1+\sqrt{k'})} \tag{3.18}$$

$$q = q_0 + 2q_0^5 + 15q_0^9 + 150q_0^{13} \tag{3.19}$$

$$D = \frac{10^{0.1\alpha_S} - 1}{10^{0.1\alpha_P} - 1} \tag{3.20}$$

$$n \geq \frac{\log(16D)}{\log(1/q)} \tag{3.21}$$

$$\Lambda = \frac{1}{2n} \ln \frac{10^{0.05\alpha_p} + 1}{10^{0.05\alpha_p} - 1} \tag{3.22}$$

$$\sigma_0 = \left| \frac{2q^{1/4} \sum_{m=0}^{\infty} (-1)^m q^{m(m+1)} \sinh[(2m+1)\Lambda]}{1 + 2\sum_{m=1}^{\infty} (-1)^m q^{m^2} \cosh(2m\Lambda)} \right| \tag{3.23}$$

$$\omega = \sqrt{(1+k\sigma_0^2)(1+\frac{\sigma_0^2}{k})} \tag{3.24}$$

$$\Omega_i = \left| \frac{2q^{1/4} \sum_{m=0}^{\infty} (-1)^m q^{m(m+1)} \sinh \frac{(2m+1)\pi\mu}{n}}{1 + 2\sum_{m=1}^{\infty} (-1)^m q^{m^2} \cosh(\frac{2m\pi\mu}{n})} \right| \tag{3.25}$$

$$\mu = \begin{cases} i & \text{for odd } n \\ i - \dfrac{1}{2} & \text{for even } n \end{cases} \quad i = 1,2...,r \tag{3.26}$$

$$V_i = \sqrt{(1 - k\Omega_i^2)(1 - \frac{\Omega_i^2}{k})} \tag{3.27}$$

$$A_{O1} = \frac{1}{\Omega_i^2} \tag{3.28}$$

$$B_{0i} = \frac{(\sigma_0 V_i)^2 + (\Omega_i \omega)^2}{(1 + \sigma_0^2 \Omega_i^2)^2} \tag{3.29}$$

$$B_{1i} = \frac{2\sigma_0 V_i}{1 + \sigma_0^2 \Omega_i^2} \tag{3.30}$$

$$H_0 = \begin{cases} \sigma_0 \prod_{i=1}^{r} \dfrac{B_{0i}}{A_{0i}} & \text{for odd } n \\ 10^{-0.05\alpha_p} \prod_{i=1}^{r} \dfrac{B_{0i}}{A_{0i}} & \text{for even } n \end{cases} \tag{3.31}$$

3.2 Ladder Prototype Synthesis Algorithms

Implementation of analog filters usually falls into two categories: the cascade prototype and the ladder prototype. Implementation of the cascade prototype is much easier and straightforward as the step 3 above. However, the cascade prototype has many drawbacks because it is very sensitive with parameters of circuit elements or the propagation of signal through each cascaded stage can cause errors. Implementation of the ladder prototype is more complex. It requires designers to use tables. Unfortunately, these tables are not easy to use, and they do not cover all possible cases which are the important requirements to design good filters. Furthermore, many high quality filters with support of advanced technologies nowadays can be

implemented on the ladder prototype for examples switched capacitor filters, switched current filters, ladder filters using OPAMP, ladder filters using OTA (operational trans-conductance amplifiers) [45], [46]. That explains why we need simple algorithms to synthesize the ladder prototype conveniently.

3.2. 1 Design Ladder Low-pass Prototype without Zeros

This algorithm deals with low-pass prototypes without zeros; therefore, it follows a classical approach for the Butterworth and Chebyshev filters [47]. Given a transfer function which describes the relationship between an input voltage and an output voltage of a filter circuit, this algorithm will use an auxiliary function to calculate an input impedance $Z_N = R_N + jX_N$ and then continued fractions to find all values of circuit elements [48]. As previous parts, we only describe briefly how the algorithm works without getting into details. Readers can refer to referenced papers to read more.

Assume *T(s)* is a transfer function of a filter and *A(s)* is an auxiliary function. Then its input impedance $Z_N = R_N + jX_N$ can be calculated as following:

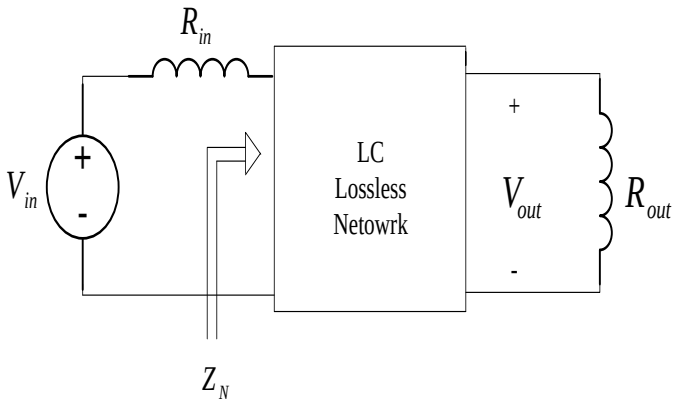

Fig. 3.5: Doubly terminated ladder network without zeros

- Step 1: define transfer function *T(s)*

$$|T(jw)|^2 = \frac{|V_{out}(jw)|^2}{|V_{in}(jw)|^2} \tag{3.32}$$

$$\text{or } T(s)T(-s)\big|_{S=jw} = \frac{R_{out}R_N}{R_{in}^2 + 2R_{in}R_N + R_N^2 + X_N^2} \tag{3.33}$$

where R_{in}: input resistance, R_{out}: output resistance

- Step 2: define auxiliary function *A(s)* respective to *T(s)*

$$|A(jw)|^2 = 1 - \frac{4R_{in}}{R_{out}}|T(jw)|^2 \tag{3.34}$$

$$\text{or } A(s)A(-s)\big|_{S=jw} = \frac{|R_{in} - Z_N|^2}{|R_{in} + Z_N|^2} \tag{3.35}$$

- Step 3: define *A(s)* and *A(-s)* according to equations (3.36) and (3.37). Then calculate *A(s)* from *A(s)A(-s)*

$$A(s) = a_N s^N + a_{N-1}s^{N-1} + \ldots + a_2 s^2 + a_1 s + a_0 \tag{3.36}$$

$$A(s)A(-s) = k_{2N}s^{2N} + k_{2N-2}s^{2N-2} + \ldots + k_2 s^2 + k_0 \tag{3.37}$$

Based on equations (3.38) and (3.39), the *k* terms with respect to the *a* terms can be found by using an iterative procedure

$$k_{2i} = (-1)^{i \bmod 2} a_i^2 + 2\sum_{j=1}^{i} (-1)^{j-1} a_{i-j} a_{i+j}, \quad (0 < i < N/2) \tag{3.38}$$

$$k_{2i} = (-1)^{i \bmod 2} a_i^2 + 2\sum_{j=1}^{N-i} (-1)^{j-1} a_{i-j} a_{i+j}, \quad (N/2 \le i < N) \tag{3.39}$$

- Step 4: once *A(s)* is found, calculate input impedance

$$Z_N(s) = \frac{R_{in}[1 - A(s)]}{1 + A(s)} \tag{3.40}$$

Once the input impedance Z_N is found, we can use the classical continued fraction expansion to refer values of all components. This method is well-documented in [49].

3.2. 2 Design Ladder Low-pass Prototype with Zeros

Ladder filters with zeros as the Cauer Elliptic and the inverse Chebyshev make the synthesis more challenging. In this case, the classical continued fraction expansion cannot be applied straightforwardly as presented in the last part. Instead this method will determine all capacitor and inductor values by removing shunt capacitors and resonant circuits from the impedance equation. This process of removing shunt capacitors and resonant circuits will continue until all resonant circuits are removed. A general type of ladder filters with zeros is depicted in Fig. 3.6. A resonant circuit with an inductor and a capacitor in parallel contributes a pair of zeros.

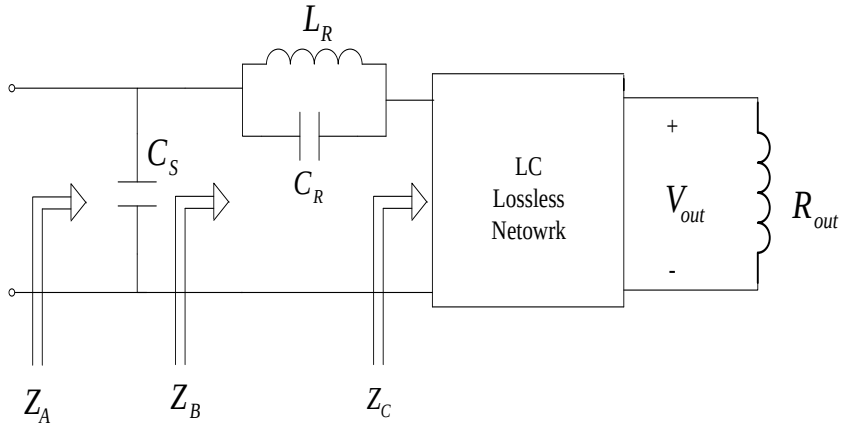

Fig. 3.6: General ladder circuit with presence of zeros

- Step 1: given the input impedance equation calculated from an auxiliary function

$$Z_A(s) = \frac{N_A}{D_A} = \frac{a_{N-1}s^{N-1} + a_{N-2}s^{N-2} + a_{N-3}s^{N-3} + ... + a_2 s^2 + a_1 s + a_0}{b_N s^N + b_{N-1}s^{N-1} + b_{N-2}s^{N-2} + ... + b_2 s^2 + b_1 s + b_0} \quad (3.41)$$

- Step 2: calculate and remove shunt capacitor

$$Z_B(s) = \frac{N_A}{D_A - sC_S N_A} = \frac{a_{N-1}s^{N-1} + a_{N-2}s^{N-2} + a_{N-3}s^{N-3} + ... + a_2 s^2 + a_1 s + a_0}{k_N s^N + k_{N-1}s^{N-1} + k_{N-2}s^{N-2} + ... + k_2 s^2 + k_1 s + k_0} \quad (3.42)$$

where: $k_i = b_i - C_S a_{i-1}$ *for* $(1 \le i \le N), k_0 = b_0$

Shunt capacitor value is a concurrent solution of equations (3.43) and (3.44)

$$C_S = \frac{b_1 z^{N-1} - b_3 z^{N-3} + b_5 z^{N-5} - ...}{a_0 z^{N-1} - a_2 z^{N-3} + a_4 z^{N-5} - ...} \quad (3.43)$$

$$C_S = \frac{-(b_0 z^{N-1} - b_2 z^{N-3} + b_4 z^{N-5} - ...)}{a_1 z^{N-1} - a_3 z^{N-3} + a_5 z^{N-5} - ...} \quad (3.44)$$

where: $z^2 = [L_R C_R]^{-1}$

- Step 3: calculate the resonant inductor value and remove the resonant circuit from the impedance equation

$$Z_B = \frac{N_B}{D_B} = Z_{LC} + Z_C = \frac{L_R s}{z^2 s^2 + 1} + \frac{N_C}{D_C} \quad (3.45)$$

$$D_B = D_C(z^2 s^2 + 1) \quad (3.46)$$

$$L_R = \left.\frac{N_B}{D_C s}\right|_{s \to \frac{j}{z}} = Z_B(s)\left.\frac{(z^2 s^2 + 1)}{s}\right|_{s \to \frac{i}{z}} \quad (3.47)$$

- Step 4: calculate C_R and Z_C
- Step 5: the proess will be repeated until all resonant circuits have been removed. From then, the remaining shunt capacitor, inductor and output resistance are found by using classical continued fraction approach.

3.3 Transformation from Low-pass Filters to Other Type Filters

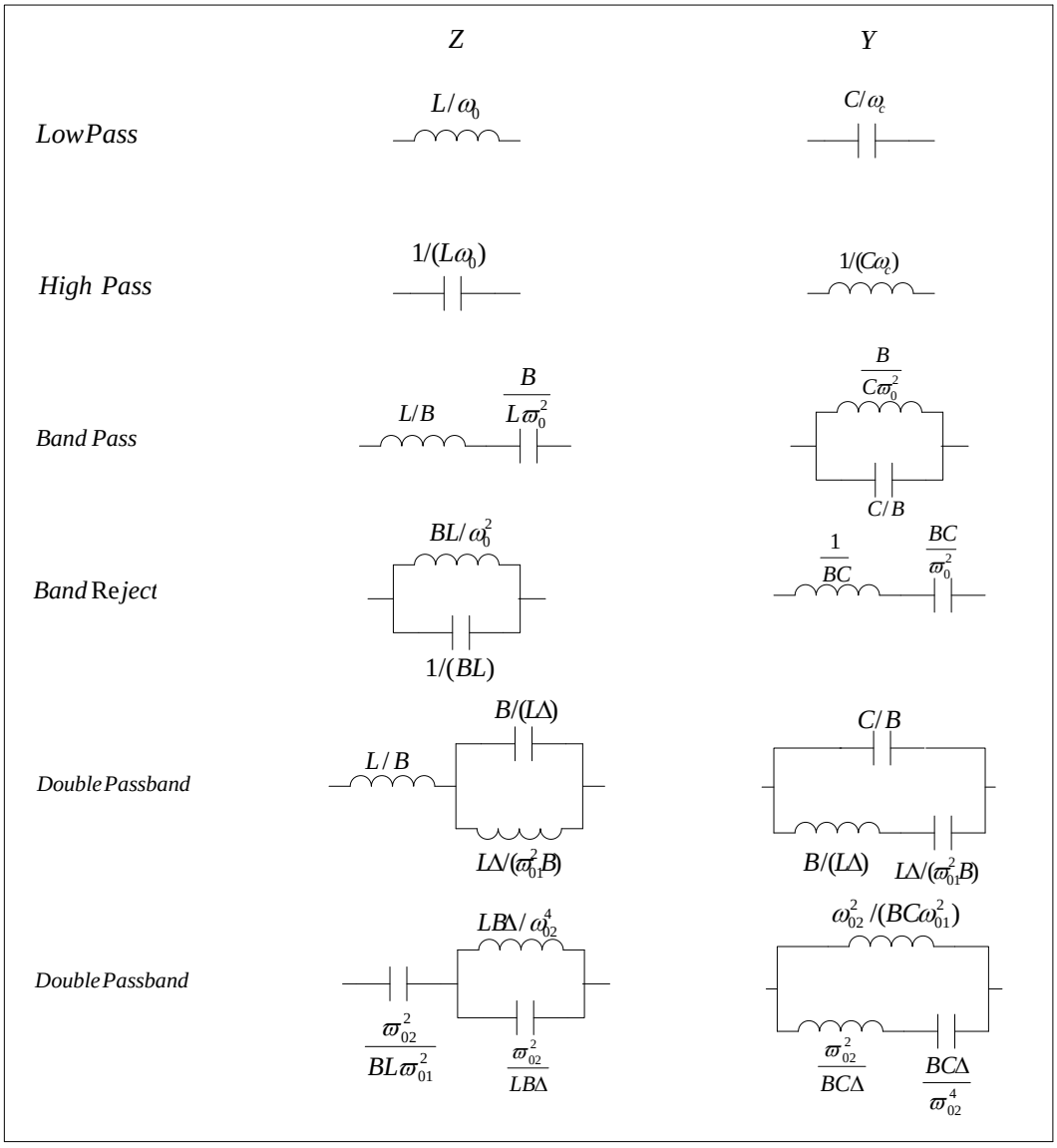

Table 3.1: Transformation of the low pass immittances L and C to ladder arms for high pass, band-pass, band-reject, and multiple pass-band filters

There are existing two different approaches to design high-pass, band-pass and band-stop filters, etc. from low-pass filters. One approach uses the Foster function, and substitutes it into the transfer function of a low-pass filter to get the transfer function of a desired filter. This approach is popularly used in designing active filters. Then the realized circuits can be cascaded

to meet the target function. In contrast with this approach, which requires the transformation from the low-pass function into the target function, another approach can complete the design process of *LC* filters in the low-pass domain *S* without constructing of the transfer function. This approach can be interpreted as transforming the realized circuit from a low-pass filter to desired filters. The operation of this second approach can be summarized in Table 3.1 [47].

3.4 Lossy Filter Synthesis with Improved Simplex Method

Ladder filters are made up of inductors and capacitors and widely used in communication systems. How to design a good filter with a desired frequency response is a challenge because the traditional algorithms as Butterworth, Chebyshev or inverse Chebyshev, etc. just synthesize filters without affects of lossy inductors and capacitors Fig. 3.7. Therefore, the frequency responses of these ideal filters are much different from the ones of real filters. In order to implement a real filter with lossy elements, which has similar characteristics as a prototype filter, we have to shift pole locations of a real filter close to pole locations of a prototype filter. In this part, the improved simplex algorithm can be utilized to replace for analytical solutions or other methods which are very complex and inefficient especially with high order filters.

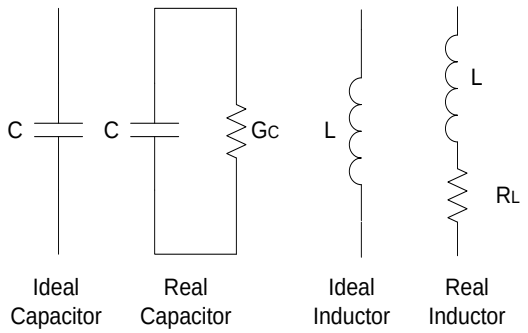

Fig. 3.7: Models of real capacitors and real inductors

Our example is to design a 4th low-pass Chebyshev filter with attenuation in the pass-band α_p= 3dB, attenuation in the stop-band α_s= 30 dB, pass-band frequency ω_p= 1kHz, stop-band frequency ω_s=2kHz. Using the Chebyshev methodology with ideal elements [43], we can find the transfer function of this filter (eq. 3.48) and its circuit with all element values (Fig. 3.8).

$$H(S) = \frac{0.17198}{S^4 + 0.88598S^3 + 1.22091S^2 + 0.64803S + 0.17198} \tag{3.48}$$

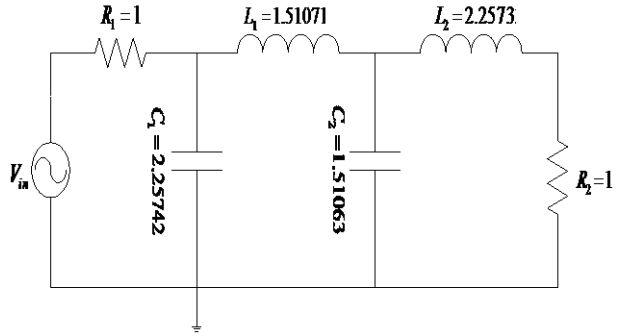

Fig. 3.8: Chebyshev filter circuit with ideal elements

Instead of using ideal elements, we replace them by lossy elements respectively with GC_1= GC_2= RL_1= RL_2= 0.1 (Fig. 3.9). On account of this affect, the synthesized filter has the new transfer function eq. 3.49 with different pole locations Fig. 3.10b. Therefore, the frequency response of the real filter is changed with its shifted cut-off frequency to the left Fig. 3.10a

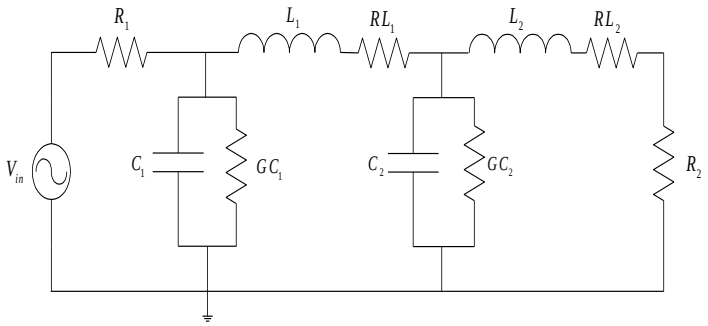

Fig. 3.9: Chebyshev filter circuit with lossy elements

$$H(S) = \frac{1}{a_1 s^4 + a_2 s^3 + a_3 s^2 + a_4 s + a_5} \tag{3.49}$$

where :

$$\begin{cases} a_1 = C_1 C_2 L_1 L_2 R_2 \\ a_2 = C_1 L_1 L_2 + 0.1 C_1 C_2 L_1 R_2 + 0.1 C_1 C_2 L_2 R_2 \\ + 0.1 C_1 L_1 L_2 R_2 + 0.1 C_2 L_1 L_2 R_2 + C_1 C_2 L_2 R_1 R_2 \\ a_3 = 0.1 C_1 L_1 + 0.1 C_1 L_2 + 0.1 L_1 L_2 + C_1 L_2 R_1 \\ + 0.01 C_1 C_2 R_2 + 1.01 C_1 L_1 R_2 + 1.01 C_2 L_1 R_2 \\ + 0.01 C_1 L_2 R_2 + 1.01 C_2 L_2 R_2 + 0.01 L_1 L_2 R_2 \\ + 0.1 C_1 C_2 R_1 R_2 + 0.1 C_1 L_2 R_1 R_2 + 0.1 C_2 L_2 R_1 R_2 \\ a_4 = 0.01 C_1 + 1.01 L_1 + 1.01 L_2 + 0.1 C_1 R_1 + 0.1 L_2 R_1 \\ + 0.101 C_1 R_2 + 0.201 C_2 R_2 + 0.201 L_1 R_2 + 0.101 L_2 R_2 \\ + 1.01 C_1 R_1 R_2 + 1.01 C_2 R_1 R_2 + 0.01 L_2 R_1 R_2 \\ a_5 = 0.201 + 1.01 R_1 + 1.0301 R_2 + 0.201 R_1 R_2 \end{cases} \tag{3.50}$$

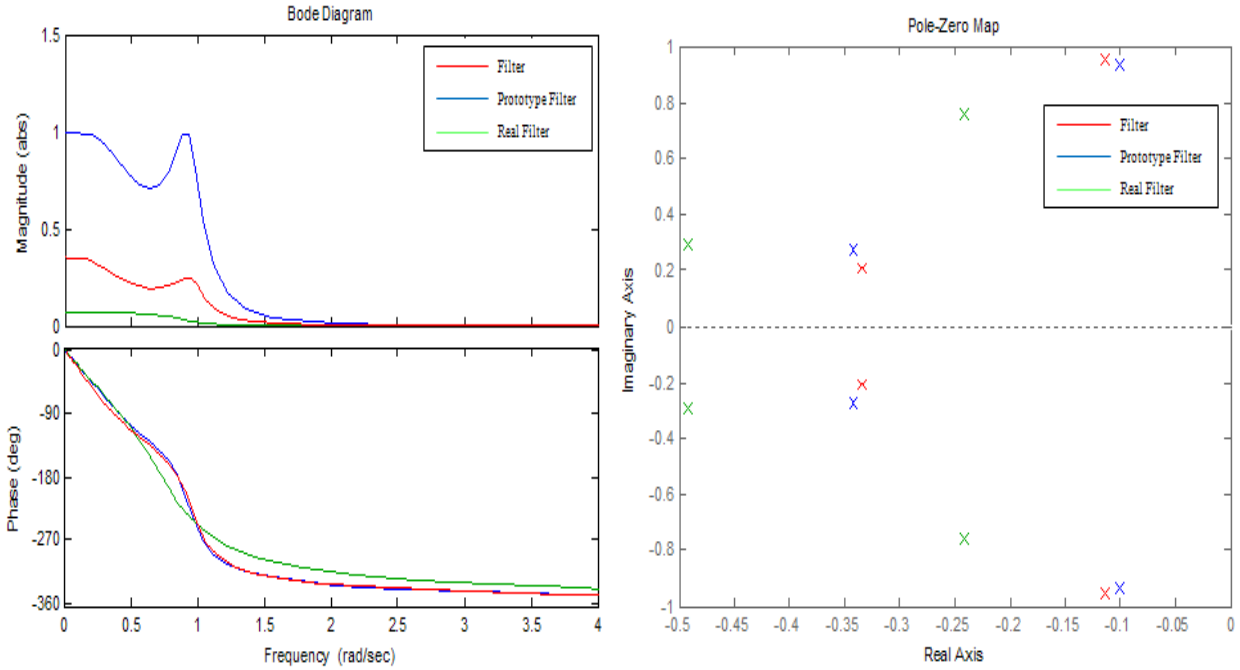

Fig. 3.10: Magnitude and phase responses (a) and pole locations (b) of filters

In order to design a filter having desired characteristics as the ideal filter, its transfer function (eq. 3.49) has to be similar to the transfer function of the prototype filter (eq. 3.48). In other words, its pole locations have to be close to pole locations of the ideal filter (Fig. 3.10b). According to the analytical method presented in [50], they have to solve a nonlinear system (eq. 3.51) of five equations with five unknowns (assume R_2 is known). Clearly, the analytical method is not an effective way to find proper solutions for this filter. Firstly, it is not easy to solve this nonlinear system especially with high order filters. Secondly, its solutions may not be applied in real implementation if one of its component values is a negative number. Therefore, it is necessary to have a simpler method which can synthesize iteratively without complex modifications.

$$\begin{cases} a_1 = 1 \\ a_2 = 0.88598 \\ a_3 = 1.22091 \\ a_4 = 0.64803 \\ a_5 = 0.17198 \end{cases} \tag{3.51}$$

As presented in the previous sections, the improved simplex method has ability to optimize high dimensional problems with very reliable convergence rates. For this particular application, this algorithm can be applied very effectively to synthesize filters. Instead of using the analytical method, we can use the improved simplex method to optimize the error function (eq. 3.52). To guarantee reasonable results with all positive values, we may divide the numerator and denominator of the transfer function of the real filter by the value of C_1 in this case (which does not change characteristics of filters). The desired filter with R_1= 0.07131, R_2= 0.2, C_1= 3.3831, L_1= 0.70676, C_2= 9.7949, L_2= 0.7189 has similar frequency response and pole locations as the ideal filter (Fig. 3.10).

$$Er = [(a_1 - 1)^2 + (a_2 - 0.8859)^2 + (a_3 - 1.2209)^2 + (a_4 - 0.6480)^2 + (a_5 - 0.1719)^2]/5 \quad (3.52)$$

Let us consider another example with a singly terminated 4th order Chebyshev low-pass filter with its ideal transfer function eq. 3.53. To synthesize this filter with lossy elements, we have to replace ideal components in the circuit Fig. 3.11 by real components in the circuit Fig. 3.12 and repeat the same procedure as presented in the previous example.

$$H(s) = \frac{1}{1.20699s^4 + 2.17713s^3 + 3.17051s^2 + 2.44475s + 1} \quad (3.53)$$

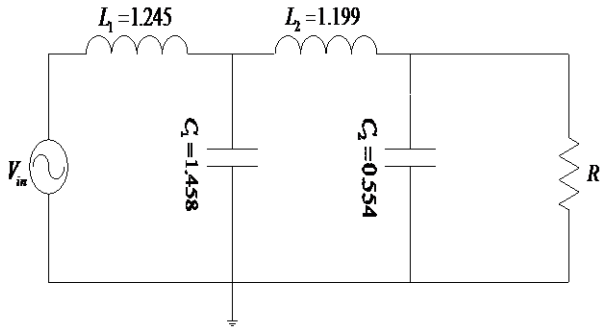

Fig. 3.11: 4th Chebyshev filter circuit with ideal elements

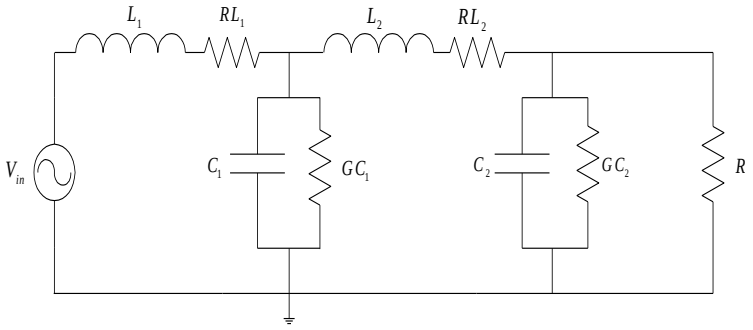

Fig. 3.12: 4th Chebyshev filter circuit with lossy elements

The new transfer function has its coefficients a_1, a_2, a_3, a_4, a_5 as eq. 3.49.

where:

$$\begin{cases} a_1 = C_1C_2L_1L_2R \\ a_2 = C_1L_1L_2 + 0.1C_1C_2L_1R + 0.1C_1C_2L_2R \\ + 0.1C_1L_1L_2R + 0.1C_2L_1L_2R \\ a_3 = 0.1C_1L_1 + 0.1C_1L_2 + 0.1L_1L_2 + 0.01C_1C_2R \\ + 1.01C_1L_1R + 1.01C_2L_1R + 0.01C_1L_2R \\ + 1.01C_2L_2R + 0.01L_1L_2R \\ a_4 = 0.01C_1 + 1.01L_1 + 1.01L_2 + 0.101C_1R \\ + 0.201C_2R + 0.201L_1R + 0.101L_2R \\ a_5 = 0.201 + 1.0301R \end{cases} \tag{3.54}$$

By using the improved simplex method to optimize the error function with different values as eq. 3. 52, the desired filter with R= 2.1727, C_1= 2.0432, L_1= 0.8568, C_2= 0.2497, L_2= 2.1364 has similar frequency response and pole locations as the ideal filter.

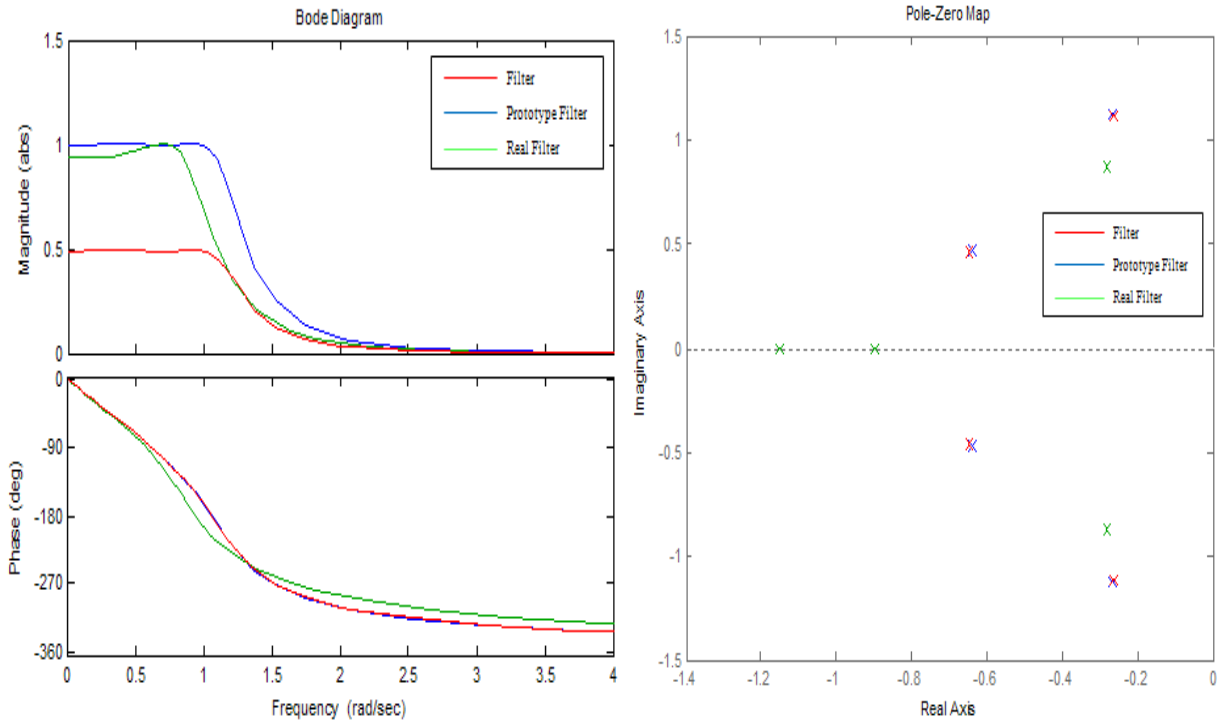

Fig. 3.13: Magnitude and phase responses (a) and pole locations (b) of filters

Chapter 4

Training Neural Networks with Improved Simplex Method

4.1 Artificial Neural Networks

A neuron is normally connected by inputs, weights and a bias weight. Its output is defined by the standard activation function:

$$f = (1+e^{-a})^{-1}, a = \sum_{i}^{n}(w_i x_i - w_{bias}) \tag{4.1}$$

where x_i is input, w_i is weight and w_{bias} is bias weight.

Each neuron can be connected together to form a neural network. A neural network is trained by input patterns and desired outputs. Weights of a network are adjusted to minimize the error function between actual outputs and desired outputs eq. 4.2.

$$E(w) = \frac{1}{2}\sum_{p=1}^{P}\sum_{m=1}^{M}\left(\hat{y}_{m,p} - y_{m,p}\right)^2 \tag{4.2}$$

where: $\hat{y}_{m,p}$ and y_m^p are actual and desired outputs of a network respectively

P: is the total number of patterns

M : is the total number of output neurons.

Although neural networks have shown their potential power for many applications, it is so difficult to train them successfully [51]-[54]. This frustration can be explained from the architectures and the training algorithms. If the size is too small, a neural

network cannot be trained. Inversely, if the size is too large, outputs from a neural network maybe not satiable. It means that a neural network can be an efficient solution once we can select a good architecture and a good algorithm to train it. Many training algorithms have been introduced so far, but each of them has its pros and cons. Some algorithms are good at training these types of neural network architectures, but are not good at training the others. Some algorithms converge very fast but require a lot of computing cost, which limits themselves in many practical applications. Some others do not require high computing cost but are unreliable. As the matter of fact, it is not easy to find a reliable algorithm which has the ability to train all types of neural networks. Until now neural networks are still an interesting area of artificial intelligence. The Error Back Propagation (EBP) is considered as a breakthrough to train neural networks but it is not an effective algorithm because of its slow convergence. The Lavenberg Marquardt is much faster but it is not suitable for large networks.

4.1.1 Neural Network Architectures

As presented above, artificial neural networks are a complex combination of training algorithms and architectures. It is clear that training algorithms are desperately an important key to ANN's success and also the most challenging part. Nevertheless, neural network architectures cannot be contempted, which help us reason many phenomena during training process. In practice, many researchers usually face a problem when some neural networks can be trained very well with training patterns but they perform poorly with verification patterns [55]. This can be explained that the selected neural network architectures are not optimal. Therefore, they do not have enough power to interpolate with the new patterns which are not used during training process. The main goal of neural networks is not to find the exact solutions, but to find the

optimum solutions. In case, the architecture is not optimized, a neural network will lose its generality and will not interpolate well. In order to reduce this side affect, it is better to select the architecture as small number of neurons as possible. A network with more neurons can normally get higher success rate with training algorithms, but its training result is usually far from the desired one. In other words, its success may be misleading. In contrast, a network with fewer neurons can give much better results; however, it is very tough to train this network with small errors and often requires efficient training algorithms. Fewer neurons require more intensive computations which can only work with advanced training algorithms discussed later. There is no proper answer to know how many neurons are optimum for a certain application. The best approach is trial and error, which is usually time-consuming in case an architecture is trained with an inefficient algorithm. However, this work can be easier with powerful architectures using fewer neurons plus more advanced algorithms to train the same problems.

Some well-known architectures as radial basis function (RBF), learning vector quantization (LVQ), etc. which can be trained easily, but they require a large number of neurons equal to the number of patterns or the number of clusters [55]. In order to obtain optimum results, these architectures are not the best solutions compared with others. They not only use more neurons than needed, but also are so expensive in computing cost. This will become more critical in training large networks for complex problems. Because of these reasons, this section will only focus on three basic but fundamental architectures: (1) the multilayer perceptron (MLP), (2) the bridged multilayer perceptron (BMLP), (3) the fully connected cascade (FCC). These three architectures are familiar and widely used in training neural networks.

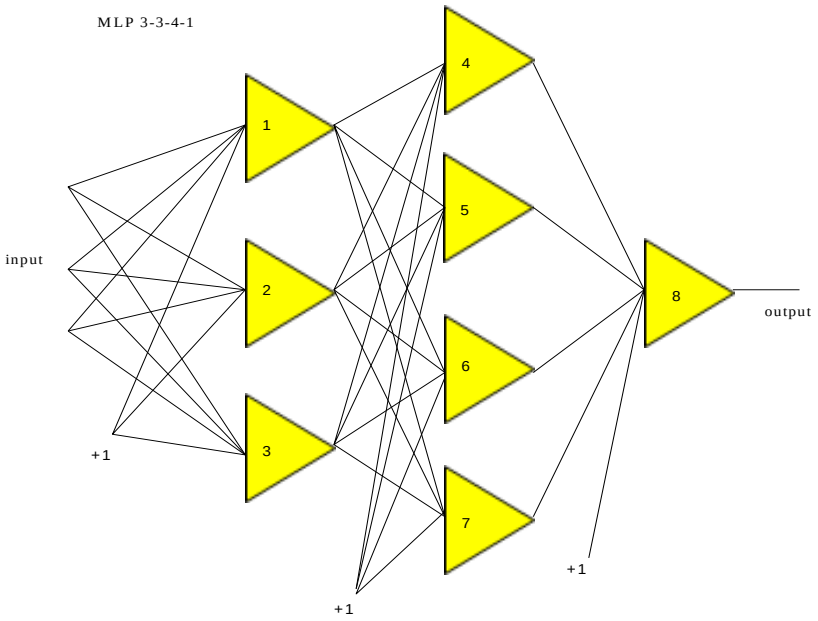

Fig. 4.1: Multilayer perceptron architecture 3-3-4-1 (MLP)

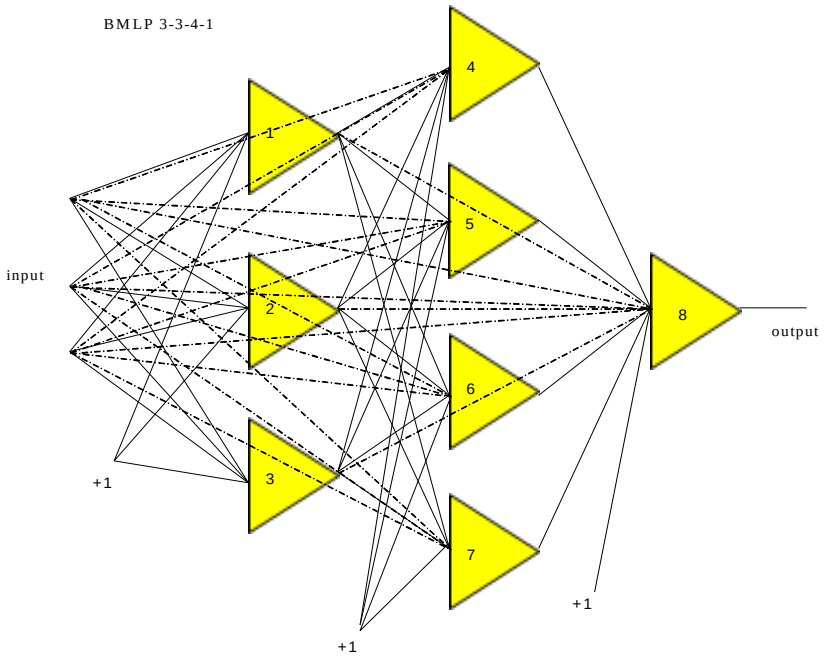

Fig. 4.2: Bridged multilayer perceptron architecture 3-3-4-1 (BMLP)

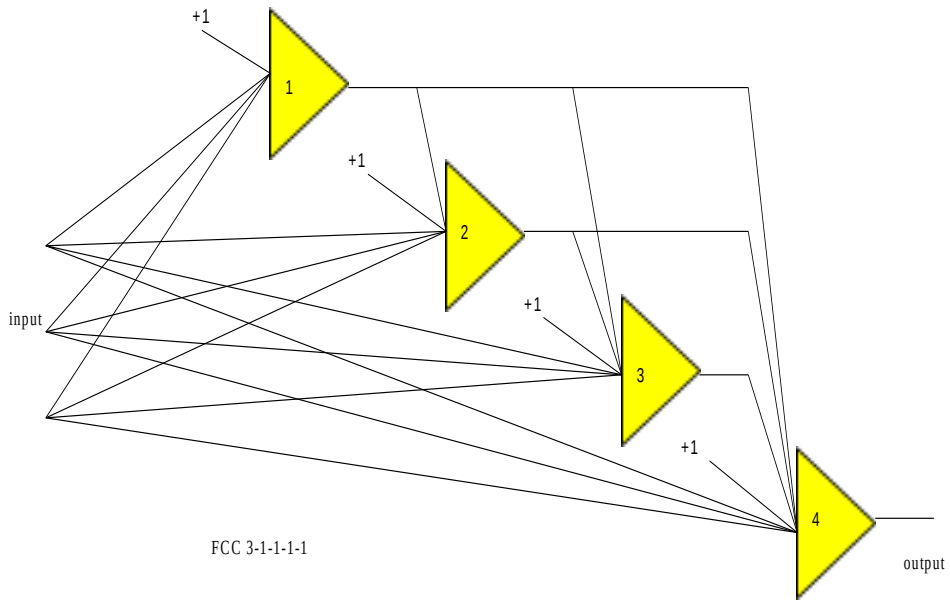

Fig. 4.3: Fully connected cascade architecture 3-1-1-1 (FCC)

The multilayer perceptron without connections cross layers is the oldest architecture. The MLP can have one hidden layer or multiple layers and signals have to propagate through each layer. Disadvantage of this architecture is that it has to use more neurons than other architectures to solve the same problems, and it limits abilities of signal processing. The BMLP and FCC are more powerful than the MLP. These two prototypes allow connections cross layers. With these additional connections, neural networks are more transparent, and hence easier to train. In order to evaluate performances of these three prototypes, an experiment is conducted to test them with parity-N problems which are considered as one of the toughest issues of training neural networks [55]. Then the results are reported in Table 4.1 in term of the number of neurons over weights. From the comparison, the FCC architecture can be the most powerful prototype in most cases. It needs much fewer neurons. It is also easier to find the optimal architecture from the FCC prototype. In contrast, if we use the MLP or BMLP, it will be more difficult. Because there are

more neurons, and from hence there are more possibilities.

Architecture	Parity-3	Parity-7	Parity-15	Parity-31	Parity-63
MLP	4/16	8/64	16/256	32/1024	64/4096
BMLP	3/14	5/44	9/152	17/560	33/2144
FCC	2/9	3/27	4/70	5/170	6/399

Table 4.1: Number of neurons/weights required for different parity problems using neural network architectures

4.1.2 Error Back Propagation Algorithm

The **s**teepest descent gradient method is a well-known technique in optimization and training neural networks even though it was named differently as the Widrow-Hoff learning rule or the delta learning rule. Before the appearance of the Error Back Propagation (EBP), it had very limited ability to train neural networks, which consist of one input layer and one output layer (Fig 4.4). In order to train this type of neural network, this method has to calculate gradients of an error function of each neural output with respect to each weight. Assume the total error function LMS (Least Mean Square) is defined as:

$$E_j = \frac{1}{2}\sum_{p=1}^{p}\left(o_j^p - d_j^p\right)^2 \tag{4.3}$$

where $o = f(net)$

p: number of patterns

j: number of outputs

Take the derivative of LMS with respect to the weight w_{ij}

$$\frac{\partial E_j}{\partial w_{ij}} = \sum_{p=1}^{p}\left(o_j^p - d_j^p\right)\frac{\partial f(net_j^p)}{net_j^p}x_i \tag{4.4}$$

$$\Delta w_{ij} = \sum_{p=1}^{p}\left(o_j^p - d_j^p\right)\left(f_j^p\right)' x_i \tag{4.5}$$

where x_i is input signal

For each training pattern, each weight will be updated by this formula

$$w_{ij} = w_{ij} - \alpha \Delta w_{ij} \tag{4.6}$$

where α is a learning constant

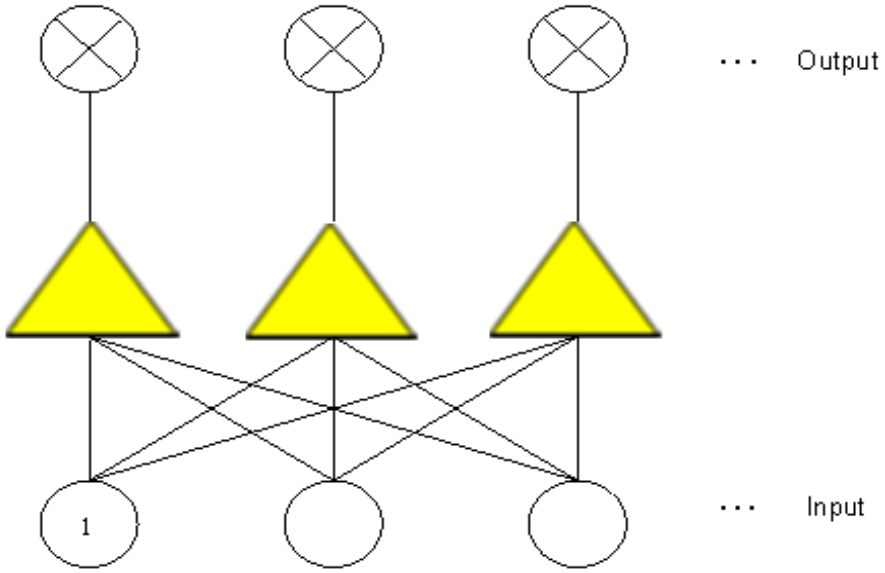

Fig. 4.4: Neural network with one input layer and one output layer

According to this algorithm, the neuron weights will change in proportional to the error values between the desired outputs and the actual outputs, and to the derivative of the activation function, and to the input signal. However, we will face difficulties to train multi-layer neural networks with hidden layers if we try to apply the same approach. The reason is that neural networks do not have target values of each neuron output in hidden layers. Therefore, we do not know how to tell hidden units what to do. In other words, the contribution of weights in hidden layers to outputs is unknown. Actually, this unsolved question used to be a big problem which made neural networks fall out of flavor after an initial period of high popularity in 1950s. It took

30 years before the Error Back Propagation popularized a new way to train hidden neurons. And this algorithm has lead to the new waves of neural network researches and applications.

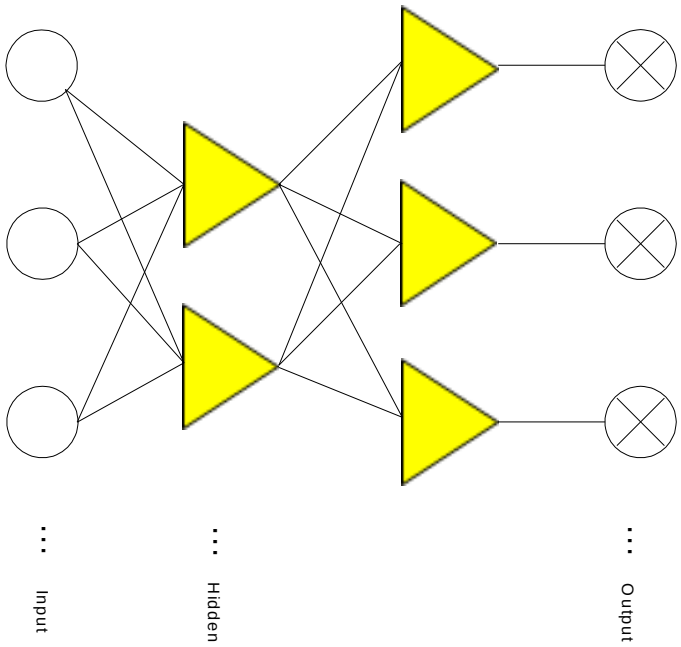

Fig. 4.5: Neural network with one hidden layer

The EBP is considered as a first order gradient method. This algorithm proposed a way to train hidden neurons in multi-layer networks through the back-propagation technique. From hence, it allows us to compute derivatives of error functions with respect to each weight. The EBP is a fundamental algorithm in training neural networks, and understanding the EBP would help us grasp more advanced training algorithms later. Because of this reason, this section will present more details about this algorithm. In order to simplify the presentation, training neural networks with one pattern will be described here.

(1) Definitions

Error signal for unit i: $\delta_i = \partial E / \partial net_i$ (4.7)

The gradient with respect to weight: $\Delta w_{ij} = \partial E / \partial w_{ij}$ (4.8)

The set of nodes anterior to unit *i*: $A_i = \{ j : \exists w_{ij} \}$

The set of nodes posterior to unit *j*: $P_j = \{ i : \exists w_{ij} \}$

(2) Forward propagation, the input units are determined by the external input signal *x*. All other units propagate forward and their outputs are defined as following:

$$y_i = f_i(\sum_{j \in A_i} w_{ij} y_j) \tag{4.9}$$

where *f* is an activation function

(3) Calculate error function between desired values and actual values

$$E = \frac{1}{2} \sum_o (d_o - y_o)^2 \tag{4.10}$$

(4) Calculate gradient at the output layer by the use of chain rule.

$$\Delta w_{ij} = \frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial net_i} \frac{\partial net_i}{\partial w_{ij}} \tag{4.11}$$

The first factor: $\frac{\partial E}{\partial net_i} = \delta_i$ (4.12)

The second factor: $\frac{\partial net_i}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_{k \in A_i} w_{ik} y_k = y_j$ (4.13)

Put these factors together we have $\Delta w_{ij} = \delta_i y_j$ (4.14)

(5) Error back propagation: after calculating errors at output units, it has to be propagated back to calculate errors of hidden units. Once again the chain rule is applied to expand the error function of hidden units in terms of posterior nodes.

$$\delta_j = \sum \frac{\partial E}{\partial net_j} = \sum_{i \in P_j} \frac{\partial E}{\partial net_i} \frac{\partial net_i}{\partial y_j} \frac{\partial y_j}{\partial net_j} \tag{4.15}$$

The first factor: $\frac{\partial E}{\partial net_i} = \delta_i$ (4.16)

The second factor: $\frac{\partial net_i}{\partial y_j} = \frac{\partial}{\partial y_j} \sum_{k \in A_i} w_{ik} y_k = w_{ij}$ (4.17)

The third factor: $\frac{\partial y_j}{\partial net_j} = f_j^{'}(net_j)$ (4.18)

Put these factors together we have $\delta_j = f_j^{'}(net_j) \sum_{i \in P_j} \delta_i w_{ij}$ (4.19)

(6) Error back propagation will continue until the derivative of the error function with respect to each weight is done

$$gradient \quad g = \begin{matrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_n} \end{matrix} \tag{4.20}$$

(7) Update weights for each training pattern in each iteration

$$w_{k+1} = w_k - \alpha g \tag{4.21}$$

As seen above, the EBP algorithm includes in mathematic equations which look complicated. In fact, its process is intuitively very clear. When a training pattern is clamped, it will propagate through a network and produce an actual value at each output neuron. Each actual value is compared with each desired value to calculate the error. In order to train neural networks, training algorithms have to minimize this error value close to zero by adjusting

network weights as gradient methods. However, just by applying this rule straightforwardly, networks cannot update new weights of hidden neurons because these units do not have δ values. The EBP solved this problem by the chain rule which distributes each error value to all hidden neurons that it is connected to and then weighs for each connection. In other words, a hidden neuron will receive a δ value which is equal to a δ value of each output unit multiplied by the weight of connection between those units.

The EBP algorithm is a breakthrough which proposed a new way to train neural networks with hidden layers. It sets up a foundation for many advanced training algorithm later. It also has an advantage over advanced algorithms in term of computing cost, which plays a very important role in training large neural networks for many real applications. However, the EBP also has many disadvantages. It is known as a slow training algorithm which is very difficult to train with small errors. Besides that, this algorithm usually requires larger networks to solve the same problems compared with advanced algorithms. As considered in the section about neural network architectures, unoptimal architectures may result in undesired outputs. On account of these deficiencies, the EBP is not considered as an efficient training algorithm. Many works to improve the EBP in term of convergence speed have been done, but their success is not enough to be more realistic and applicable [56]-[58]. Because of this reason, the advanced training algorithms as Levernberg Marquardt (LM) are used the most. They require more computations, but they are more efficient with faster convergence and smaller error.

4.1.3 Lavenberg Marquardt Algorithm

The Lavenberg Marquardt algorithm is a combination between the gradient descent method and the Gauss-Newton method. It gives exchange between the stability of the first order

method and the speed of the second order method. When the current solution is far from the correct solution, the Lavenberg Marquardt algorithm behaves as the steepest gradient method, which is slow but its convergence is guaranteed. Vice versa, when the current solution is close to the correct solution, it behaves as the Gauss-Newton method, which is much faster. This explains why the Laveberg Marquardt algorithm can train neural networks with smaller errors and faster than the Error Back Propagation.

Like the quasi Newton method, the Lavenberg Marquardt algorithm was designed to approach second order speed without calculating the Hessian matrix with second derivatives. Instead it just approximates by using the Jacobian matrix, which contains first derivatives with respect to weights, biases and network errors. The Jacobian matrix can be calculated through the error back propagation technique as presented in the previous section, which is less complicated than the Hessian matrix.

- *Steepest descent method*

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha\, g \tag{4.22}$$

$$g = \begin{matrix} \dfrac{\partial E}{\partial w_1} \\ \dfrac{\partial E}{\partial w_2} \\ \vdots \\ \dfrac{\partial E}{\partial w_n} \end{matrix} \tag{4.23}$$

- *Newton method*

$$w_{k+1} = w_k - A_k^{-1} g \tag{4.24}$$

where A_k is Hessian matrix

$$A_k = \begin{matrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_2 \partial w_1} & \cdots & \frac{\partial^2 E}{\partial w_n \partial w_1} \\ \frac{\partial^2 E}{\partial w_1 \partial w_2} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_n \partial w_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 E}{\partial w_1 \partial w_n} & \frac{\partial^2 E}{\partial w_2 \partial w_n} & \cdots & \frac{\partial^2 E}{\partial w_n^2} \end{matrix} \tag{4.25}$$

- *Gauss-Newton method*

$$w_{k+1} = w_k - \left(J_k^T J_k\right)^{-1} J_k^T e \tag{4.26}$$

Where J is Jacobian matrix

$$J = \begin{bmatrix} \frac{\partial e_{11}}{\partial w_1} & \frac{\partial e_{11}}{\partial w_2} & \cdots & \frac{\partial e_{11}}{\partial w_N} \\ \frac{\partial e_{21}}{\partial w_1} & \frac{\partial e_{21}}{\partial w_2} & \cdots & \frac{\partial e_{21}}{\partial w_N} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{M1}}{\partial w_1} & \frac{\partial e_{M1}}{\partial w_2} & \cdots & \frac{\partial e_{M1}}{\partial w_N} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{1P}}{\partial w_1} & \frac{\partial e_{1P}}{\partial w_2} & \cdots & \frac{\partial e_{1P}}{\partial w_N} \\ \frac{\partial e_{2P}}{\partial w_1} & \frac{\partial e_{2P}}{\partial w_2} & \cdots & \frac{\partial e_{2P}}{\partial w_N} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{MP}}{\partial w_1} & \frac{\partial e_{MP}}{\partial w_2} & \cdots & \frac{\partial e_{MP}}{\partial w_N} \end{bmatrix} \tag{4.27}$$

N: number of weights

P: number of patterns

M: number of output

Lavenberg-Marquardt method

$$A \approx 2J^T J \tag{4.28}$$

$$g = 2J^T e \tag{4.29}$$

$$w_{k+1} = w_k - \left(J_k^T J_k + \mu I\right)^{-1} J_k^T e \tag{4.30}$$

where g - gradient vector of size N

e - error vector of size $M*P$

Derivatives:

$$g_i = \frac{\partial E}{\partial w_i} \tag{4.31}$$

$$E = \sum_{p=1}^{P}\sum_{m=1}^{M}\left(d_{pm} - o_{pm}\right)^2 = \sum_{p=1}^{P}\sum_{m=1}^{M}\left(e_{pm}\right)^2 \tag{4.32}$$

where $e_{pm} = d_{pm} - o_{pm}$

$$e = \begin{bmatrix} e_{11} \\ e_{21} \\ \vdots \\ e_{M1} \\ \vdots \\ e_{1P} \\ e_{1P} \\ \vdots \\ e_{MP} \end{bmatrix} \tag{4.33}$$

$$g_i = \frac{\partial E}{\partial w_i} = \sum_{p=1}^{P}\sum_{m=1}^{M} 2\frac{\partial e_{pm}}{\partial w_i} e_{pm} = 2\sum_{p=1}^{P}\sum_{m=1}^{M}\frac{\partial e_{pm}}{\partial w_i} e_{pm} \tag{4.34}$$

From equation 4.30, we see that when the scalar factor μ is small, the Lavenberg Marquardt is similar to the Gauss-Newton method. In contrast, when the scalar factor μ is large, it is similar to the steepest decent method with a small step size. Because the Gauss-Newton method converges much faster with higher accuracy, thus the scalar factor μ is decreased after

each successful iteration, and increased in case a tentative iteration tends to increase the performance function. In other words, the scalar is a factor to shift the Lavenberg Marquardt between the Gauss-Newton method and the steepest decent method to reduce the performance function. This factor is normally adjusted automatically by the algorithm.

The Lavenberg Marquardt has more advantages than the Error Back Propagation in terms of convergence speed and smaller error. However, the Lavenberg Marquardt has to compute the Jacobian matrix J and the inversion of J^TJ square matrix, which is not suitable and practical for large neural networks. That is the reason why the training algorithm is still a challenging topic in neural network researches.

4.2 Training Neural Networks with Improved Simplex Method

Both the Error Back Propagation and the Lavenberg Marquardt have pros and cons. The EBP requires less computation but converges extremely slowly with complex networks. In contrast, the LM has more advantages than the EBP in terms of converge speed, success rate, number of neurons, etc. but it is not efficient in training large neural networks. Its computational cost increases approximately proportional to N^2 size of problems. These two well-known algorithms have ability to solve many practical problems, but they are very limited with some reasons have just been presented above.

In order to train neural networks more efficiently, there need to be reliable algorithms having the ability to train neural networks fast enough without expensive computational cost. The Nelder Mead's simplex method is a simple algorithm which has potential to meet these criterions. In every iteration, it just computes an extra vertex and evaluates its function value to converge; whereas, the EBP and the LM require the back propagation to compute the gradient

matrix and the Jacobian matrix. In other words, the Nelder Mead's simplex method can train neural networks only by forward propagation, which is much simpler. Unlike the EBP and the LM algorithms, the simplex method's performance does not depend on the number of training patterns. When the number of training patterns increases, the LM has to compute the bigger size of the Jacobian matrix and the EBP is required to propagate through all patterns iteratively. Unfortunately, the original simplex method is not very successful in training neural networks. As presented in previous sections, this algorithm faces difficulties in optimizing high dimensional problems. Therefore, its training success rate is not very good, and definitely it is not a reliable trainer. By adapting the simplex method with quasi gradient search, the improved simplex method performs much better. Its convergence rate in optimization increases magnificently. This section will present how the improved simplex method with an extra vertex can train neural networks with some parity-N problems. Then its performance is compared with the EBP. Moreover, three different types of architectures will be used in these experiments.

Multilayer Perceptron (MLP) with one hidden layer and one output: number of hidden neurons is equal to number of patterns Fig. 4.6.

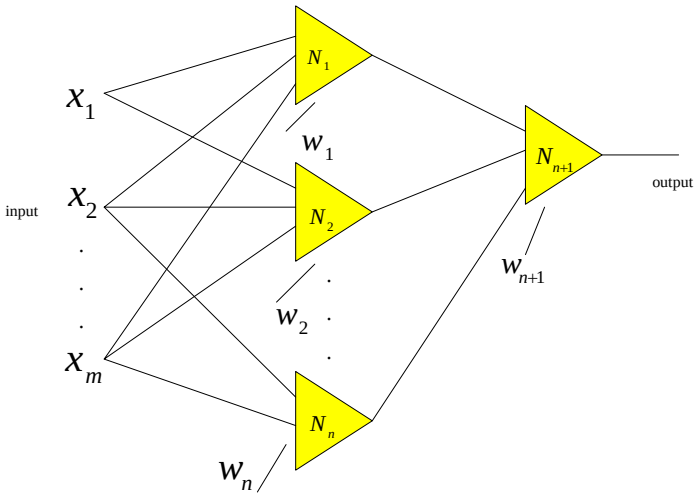

Fig. 4.6: Multilayer perceptron neural network to train parity-N problems

Parity-N	Algorithm	Success rate	Average Iteration	Comp. time (s)
Parity-2	EBP	96%	6226	2.3969
	SIM	39%	190	0.0203
	SIM_Q	100%	293	0.0513
Parity-3	EBP	100%	8423	3.1235
	SIM	30%	437	0.0542
	SIM_Q	100%	739	0.1459
Parity-4	EBP	78%	113902	37.007
	SIM	Failure	-	-
	SIM_Q	75%	21139	4.9239
Parity-5	EBP	58%	9357	5.5821
	SIM	Failure	-	-
	SIM_Q	55%	23595	6.7642

Table 4.2: Comparison of training algorithms with MLP architecture

Bridged Multilayer Perceptron (BMLP) with one hidden layer and one output: number of hidden neurons is equal to number of patterns Fig. 4.7.

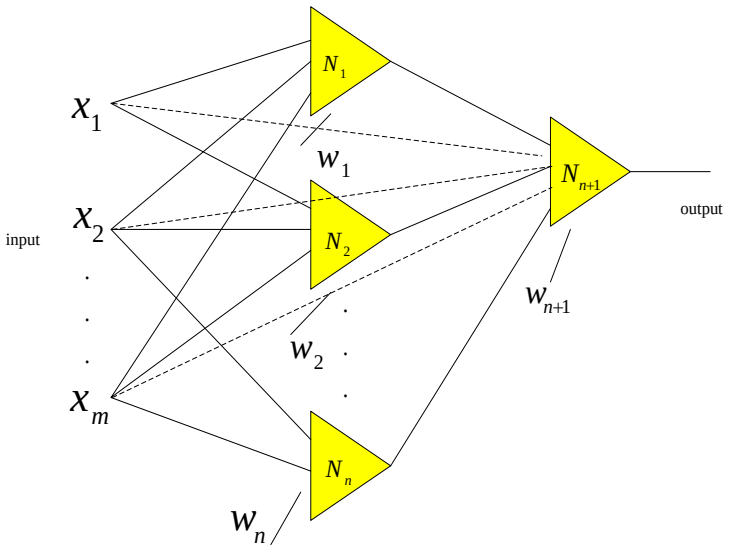

Fig. 4.7: Bridged multilayer perceptron neural network to train parity-N problems

Parity-N	Algorithm	Success rate	Average Iteration	Comp. time (s)
Parity-2	EBP	100%	10137	4.167
	SIM	100%	334	0.0337
	SIM_Q	100%	95	0.0177
Parity-3	EBP	100%	7415	3.651
	SIM	85%	841	0.0909
	SIM_Q	100%	278	0.0582
Parity-4	EBP	85%	164645	69.419
	SIM	Failure	-	-
	SIM_Q	100%	4510	1.068
Parity-5	EBP	2%	21931	5.156
	SIM	Failure	-	-
	SIM_Q	97%	14125	4.1983

Table 4.3: Comparison of training algorithms with BMLP architecture

Fully Connected Cascade (FCC): two neurons are used to train parity- 2, 3 and three neurons are used to train parity- 4, 5 Fig. 4.8.

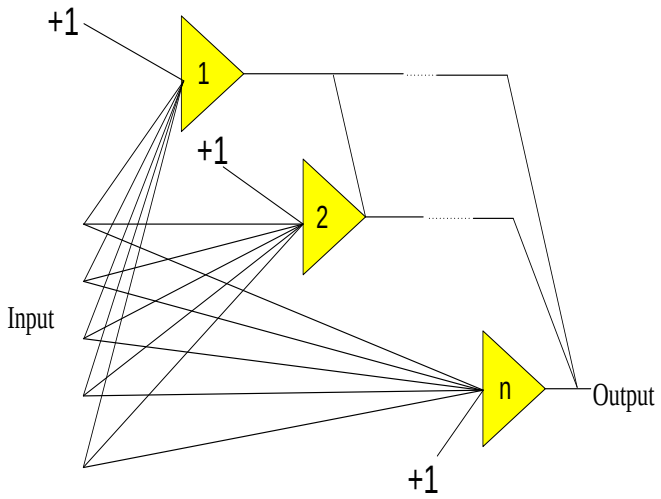

Fig. 4.8: Fully connected cascade neural network to train parity-N problems

Parity-N	Algorithm	Success rate	Average Iteration	Comp. time (s)
Parity-2	EBP	100%	15144	5.567
	SIM	100%	331	0.0345
	SIM_Q	100%	105	0.0206
Parity-3	EBP	100%	14833	4.848
	SIM	43%	539	0.0548
	SIM_Q	100%	240	0.0435
Parity-4	EBP	100%	79703	25.636
	SIM	Failure	-	-
	SIM_Q	95%	14301	2.9833
Parity-5	EBP	7%	204759	77.125
	SIM	Failure	-	-
	SIM_Q	80%	18346	4.4676

Table 4.4: Comparison of training algorithms with fully connected cascade architecture

From the experiments, we can conclude that the improved simplex method is able to train neural networks. It even converges much faster than the Error Back Propagation in most cases. Its success rate is consistently high. Whereas, the original simplex method does not converge really well, and fails to train parity-4, parity-5. Although the improved simplex method shows its ability to train neural networks with lower computing cost, it also has some disadvantages which need to be improved. It is not very stable, and normally requires more neurons than the LM algorithm. In order words, the improved simplex method has difficulty to train neural networks with optimum architectures. This deficiency may come from the accuracy of the quasi gradient methods, which is an interesting topic of future researches. Overall, the improved simplex method still has some advantages than the EBP and LM algorithms and it can be applied in many practical problems. Next section will present applications of the improved simplex method.

4.3 Control Robot Arm Kinematics with Improved Simplex Method

The Error Back Propagation (EBP) is considered as a breakthrough to train neural networks, but it is not an effective algorithm because of its slow convergence. The Lavenberg Marquardt is much faster but it is not suitable for large networks. Although these two algorithms are well-known, they usually face difficulties in many real applications because of their complex computations. Training neural networks to control robot arm kinematics is a typical example.

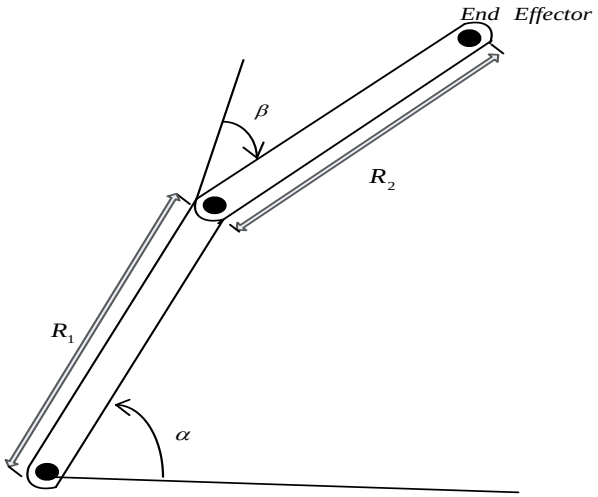

Fig. 4.9: Two-link planar manipulator

Forward kinematics is a practical example which can be resolved by neural networks. Neural networks can be trained to determine the position *x* and *y* of robot arms based on the data α, β read from sensors at the joints. This data set can be calculated from the equations (4.35, 4.36). While R_1, R_2 are the fixed length arms and α, β are the movement angles of robot arms as shown in Fig. 4.9. By sensing its movement angles α, β, the position *x* and *y* of a robot arm can be determined.

$$x = R_1 \cos\alpha + R_2 \cos(\alpha + \beta) \tag{4.35}$$

$$y = R_1 \sin\alpha + R_2 \sin(\alpha + \beta) \tag{4.36}$$

To train a neural network with three neurons fully cascaded in Fig. 4.10, we use 2500 (50x50) training patterns generated from equations (4.35), (4.36) with parameters α, β uniformly distributed in the range of [0, π] and R_1= R_2= 0.5. The desired outputs and the actual outputs from this network are depicted in Fig. 4.11 and Fig. 4.12.

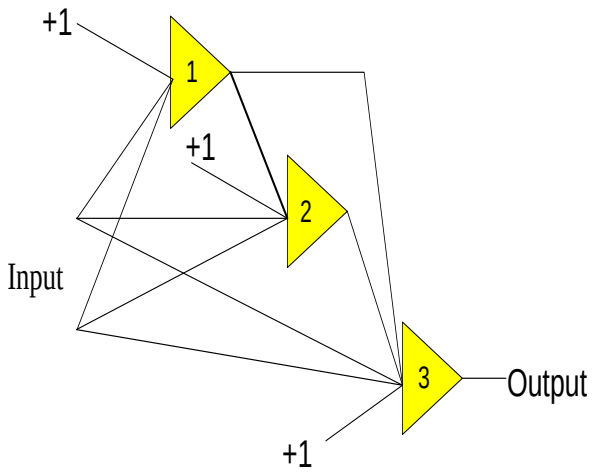

Fig. 4.10: Neural network architecture to control robot arm kinematics

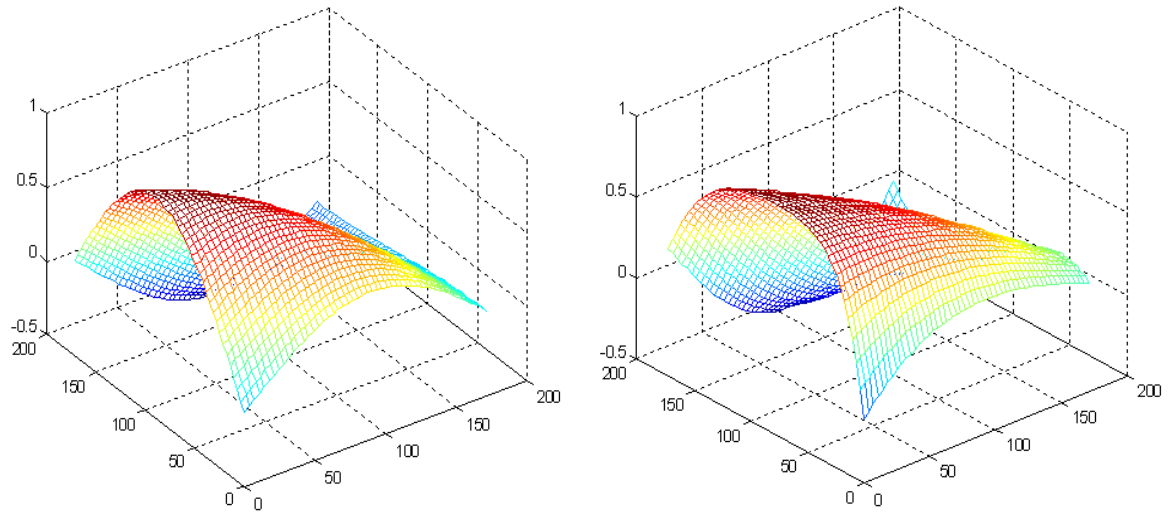

Fig. 4.11: Desired output (a) and actual output (b) from the neural network in *x* direction

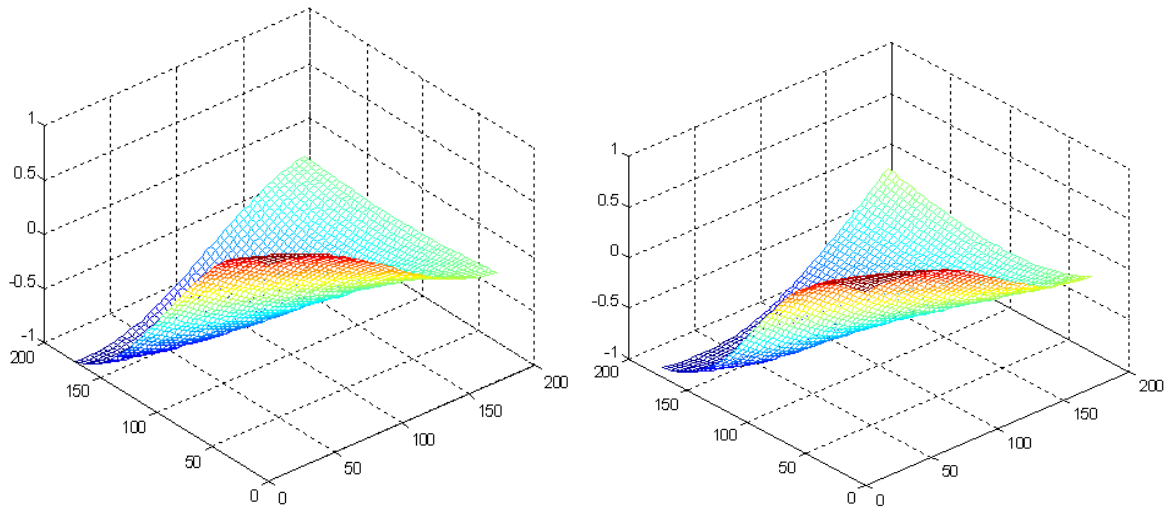

Fig. 4.12: Desired output (a) and actual output (b) from the neural network in *y* direction

As we can see, the desired outputs and actual outputs from the neural network are not so much different with an error about 0.001. The advantage of this algorithm in training neural networks is that its computational cost is proportional to the number of weights not the number of input patterns. For this particular case, all 2500 patterns can be applied iteratively, and the improved simplex method will train neural networks by optimizing a function with seven variables, which are equal to seven weights. In contrast, training neural networks with the Error Back Propagation for 2500 patterns seems impractical because it has to adjust its weights for each training pattern in each iteration. Therefore, its training process is extremely time-consuming and inapplicable for this particular case. The Levenberg Marquardt is known as a fast training algorithm, but its training ability is limited by the number of input patterns *P*, weights *N*, and outputs *M*. In other words, the problem becomes more difficult with the increasing size of a network. In each iteration, this algorithm has to calculate the Jacobian matrix J_{P*MxN} and the inversion of J^TJ square matrix. It is obvious that the LM algorithm cannot train neural networks with seven weights and 2500 input patterns for this robot arm kinematics

because of the huge size of Jacobian matrix J_{2500x7}, which over-limits computing capability of PC computers. In order to train neural networks with the EBP or LM algorithm, the size of training patterns usually has to be reduced. This will ease the computing tension but it will affect the accuracy of neural network outputs significantly. Therefore, the actual outputs may be much different from the desired outputs. In contrast, the increased size of input patterns may affect the convergence rate, but not the training ability of the improved simplex method. This character makes it different from the Error Back Propagation and the Lavenberg Marquardt.

The improved simplex method can be a useful algorithm to train neural networks for many real applications. The improved simplex method can be used to train neural networks in modeling. Curve fitting is a typical example. Neural networks has ability to approximate functions more precisely than the fuzzy system which is well-known. The same neural network architecture as Fig. 4.10 is used to approximate the function f in Fig. 4.13 and its outputs are compared in Fig. 4.14.

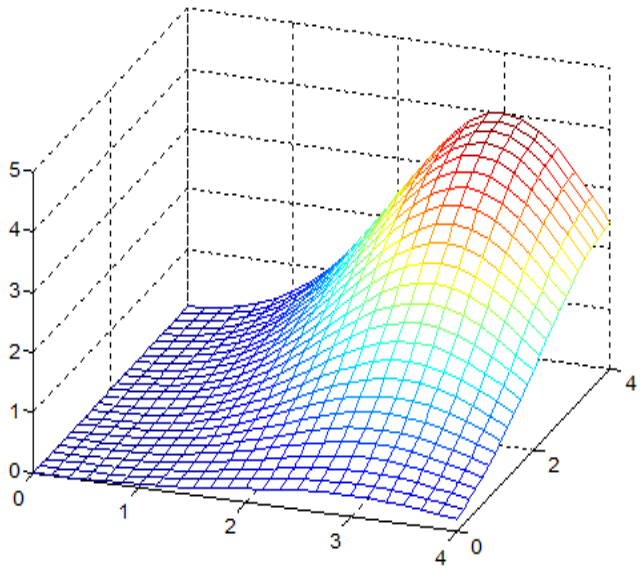

Fig. 4.13: Desired output of a function $f = 4e^{-0.15(x-4)^2-0.5(y-3)^2}$

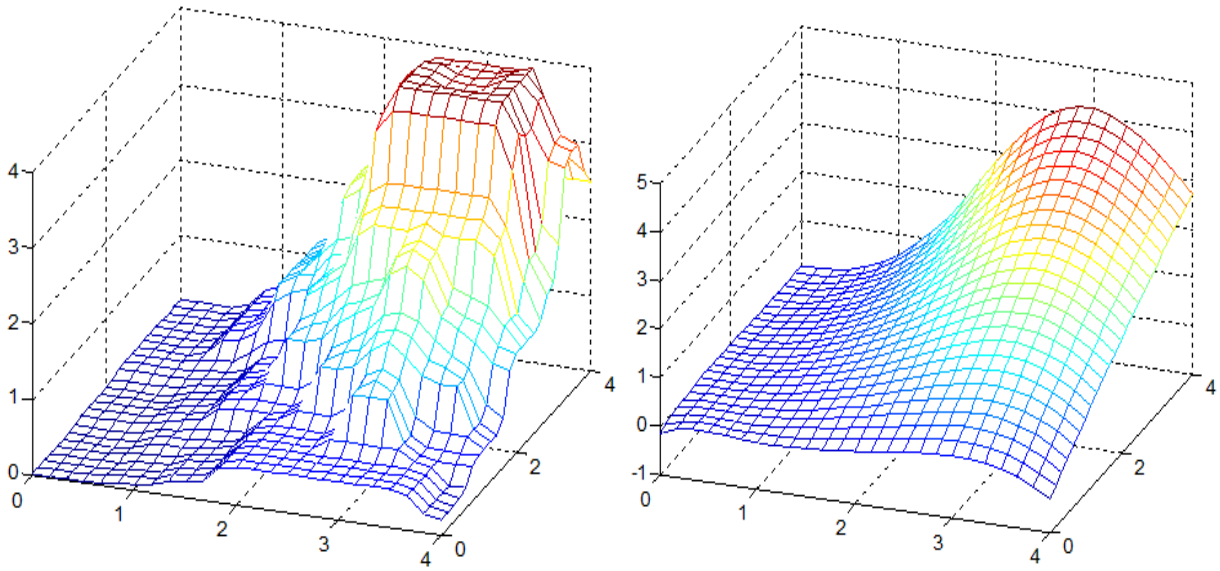

Fig. 4.14: Output from fuzzy system (a), output from neural network (b)

Chapter 5

Conclusions

Upon the comparative study of derivative free optimizations algorithms, it seems like the improved simplex method has more advantages than other algorithms based on evolutionary computations. Their comparison was summarized in Table 2.3-2.7. The improved simplex algorithm with quasi gradient search is presented with details in this paper. It is a derivative free optimization algorithm with two simple approaches of gradient search described. This algorithm can be used when it is difficult to find function derivatives, or if finding such derivatives are time consuming. This algorithm was tested over several benchmark problems of local minimum optimization, and shows its better performance than the original simplex method in terms of both convergence rate and computing time. Therefore, it shows a great deal of large scale optimization problems and has been applied successfully in synthesizing filters and training neural networks. This algorithm also shows very promising results compared with other well-known evolutionary algorithms independent of gradients as the genetic algorithm, the particle swarm algorithm, or the differential evolution algorithm, etc. it outperforms these algorithms with much higher success rate and at least ten to hundred times faster. The experiments tell that this algorithm is an effective alternative for other optimization algorithms. However, the modified algorithm presented in this paper can be improved by using other numerical techniques to calculate more accurate gradients. By using the analytical gradient instead of the quasi

gradient for some optimization problems, the improved simplex method can converge at least ten times faster. These types of improvements can be a good topic of future research.

REFERENCES:

[1] G. C. Onwubolu, B. V. Babu, “New Optimization Techniques in Engineering”, Springer 2004.

[2] J. Nocedal, S. J. Wright, “Numerical Optimization”, Springer, 1999.

[3] J. A. Nelder, R. Mead, “A Simplex Method for Function Minimization,” Computer Journal, vol. 7, pp. 308-313, 1965.

[4] K. Bredies, D. A. Lorenz and P. Mass, “A Generalized Conditional Gradient Method and Its Connection to an Iterative Shrinkage Method,” Comput. Optim. Appl, vol. 42, no. 2, pp. 173–193, 2009.

[5] K. Levenberg, “A Method for the Solution of Certain Problems in Least Squares,” Quart. Appl. Mach., vol. 2, pp. 164–168, 1944.

[6] A. Burmen, J. Puhan and T. Tuma, “Grid Restrained Nelder-Mead Algorithm,” Comput. Optim. Appl, vol. 34, no. 3, pp. 359–375, 2006.

[7] C. T. Kelly, “Detection and Remediation of Stagnation in The Nelder–Mead Algorithm Using a Sufficient Decrease Condition,” SIAM Journal on Optimization, vol. 10, no. , pp. 43–55, 2000.

[8] W. Lenwari, M. Sumner and P. Zanchetta, "The Use of Genetic Algorithms for the Design of Resonant Compensators for Active Filter," IEEE Trans. on Industrial Electronics, vol. 56, no. 8, pp. 2852-2861, August 2009.

[9] R. Storn and K. Price, “Differential Evolution—A Simple and Efficient Heuristic for

Global Optimization over Continuous Spaces," J. Glob. Optim., vol. 11, no. 4, pp. 341–359, Dec. 1997.

[10] K. Price, "An Introduction to Differential Evolution," in New Ideas in Optimization, D. Corne, M. Dorigo, and F. Glover, Eds. London, U.K.: McGraw-Hill, 1999, pp. 79–108.

[11] B. Biswal, P. K. Dash and B. K. Panigrahi, "Power Quality Disturbance Classification Using Fuzzy C-Means Algorithm and Adaptive Particle Swarm Optimization," IEEE Trans. on Industrial Electronics, vol. 56, no. 1, pp. 212-220, Jan 2009..

[12] Sung-Ho Hur, R. Katebi, A. Taylor, "Modeling and Control of a Plastic Film Manufacturing Web Process," IEEE Trans. on Industrial Informatics, vol. 7, no. 2, pp. , May 2011.

[13] C.H. Lo, E.H.K. Fung, Y.K. Wong, "Intelligent Automatic Fault Detection for Actuator Failures in Aircraft," IEEE Trans. on Industrial Informatics, vol. 5, no. 1, pp. , Feb 2009.

[14] F. Tao, D. Zhao, Y. Hu, Z. Zhou, "Resource Service Composition and Its Optimal-Selection Based on Particle Swarm Optimization in Manufacturing Grid System," IEEE Trans. on Industrial Informatics, vol. 4, no. 4, pp. , Nov 2008.

[15] P. Zanchetta, P. W. Wheeler, J. C. Clare, M. Bland, L. Empringham and D. Katsis, "Control Design of a Three-Phase Matrix-Converter-Based AC-AC Mobile Utility Power Supply," IEEE Trans. on Industrial Electronics, vol. 55, no. 1, pp. 209-217, 2008.

[16] K. Hangyu, J. Jing, S. Yong, "Improving Crossover and Mutation for Adaptive Genetic Algorithm," Computer Engineering and Application, vol. 12, pp. 93-96, 2006

[17] L. H. Cheng, W. Y. Ping, "Genetic Algorithm with a Hybrid Crossover Operator and its Convergence," Computer engineering and application, vol. 16, pp. 22-24, 2006.

[18] A. K. Qin, V. L. Huang and P. N. Suganthan, "Differential Evolution Algorithm with

Strategy Adaptation for Global Numerical Optimization," IEEE Transactions on Evolutionary Computation, vol. 13, 2009, pp. 398–417.

[19] E. M. Montes, C. A. Coello, J. V. Ryes, L M. Davila, "Multiple Trial Vectors in Differential Evolution for Engineering Design," Eng. Optim, vol. 39, no. 5, pp. 567-589, July. 2007

[20] A. Slowik, "Application of an Adaptive Differential Evolution Algorithm With Multiple Trial Vectors to Artificial," IEEE Trans. on Industrial Informatics, vol. 58, no. 8, pp. , Nov 2011.

[21] S. Agrawal, Y. Dashora, M. K. Tiwari, Y. J. Son, "Interactive Particle Swarm: A Pareto-Adaptive Metaheuristic to Multiobjective Optimization," IEEE Trans. on System, Man and Cybernetics, vol. 38, no. 2, pp. 258-277, 2008.

[22] S. Wang, J. Watada, W. Pedrycz, "Value-at-Risk-Based Two-Stage Fuzzy Facility Location Problems," IEEE Trans. on Industrial Informatics, vol. 5, no. 4, pp. , Nov 2009.

[23] G. Guo, Y. Shouyi, "Evolutionary Parallel Local Search for Function Optimization," IEEE Trans. on System, Man, and Cybernetics, Vol. 33, no. 6, pp. 864-876, 2003.

[24] C. J. Price, I. D. Coope, and D. Byatt, "A Convergent Variant of the Nelder-Mead Algorithm", Journal of Optimization Theory and Applications, vol. 11, no. 3, pp. 5–19, 2002.

[25] B. Arpad, P. Janez, T. Tadej, "Grid Restrained Nelder-Mead Algorithm", Comput. Optim. Appl, vol. 34, no. 3, pp. 359–375, 2006.

[26] C. T. Kelly, "Detection and Remediation of Stagnation in the Nelder–Mead Algorithm Using a Sufficient Decrease Condition", SIAM Journal on Optimization, vol. 10, no. , pp. 43–55, 2000.

[27] N. Larry, T. Paul, “Gilding the Lily: a Variant of the Nelder-Mead Algorithm Based on Golden-Section Search”, Comput. Optim. Appl, vol. 22, no. 1, pp. 133–144, 2002.

[28] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions," SIAM Journal of Optimization, vol. 9, no. 1, pp. 112-147, 1998.

[29] F. Gao, L. Han, “Implementing the Nelder-Mead Simplex Algorithm with Adaptive Parameters,” Comput. Optim. Appl, vol., no. , pp. , 4th May 2010.

[30] Torczon, “V.:Multi-directional Search: A Direct Search Algorithm for Parallel Machines,” Ph.D. Thesis, Rice University, TX (1989).

[31] M. Manic, B. M. Wilamowski, “Random Weights Search in Compressed Neural Networks Using Overdetermined Pseudoinverse,” IEEE International Symposium on Industrial Electronics 2003, vol. 2, pp. 678-683, 2003.

[32] Nam Pham, B. M. Wilamowski, "Improved Nedler Mead's Simplex Method and Applications," Journal of Computing, vol. 3, issue 3, March 2011, pp. 55- 63, 2011.

[33] C. J. Chung, R. G. Reynolds, “Function Optimization Using Evolutionary Programming with Self-Adaptive Cultural Algorithms,” Proceeding SEAL'96 Selected papers from the First Asia-Pacific Conference on Simulated Evolution and Learning.

[34] J. J. More, B. S. Garbow, and K. E. Hillstrom, “Testing Unconstrained Optimization Software,” ACM Trans. on Mathematical Software, vol. 7, no. 1, pp. 136-140, 1981.

[35] J. T. Betts, ”Solving the Nonlinear Least Square Problem: Application of a General Method,” Journal of Optimization Theory and Applications, vol. 18, no. 4, 1976.

[36] K. M. Bryden, D. A. Asklock, S. Corn, S. J. Wilson, “Graph- Based Evolutionary Algorithms,” IEEE Trans. on Evolutionary Computation, vol. 10, no. 5, pp. 550-567,

2006.

[37] J. D. Hewlett, B. M. Wilamowski, G. Dundar, "Optimization Using a Modified Second-Order Approach With Evolutionary Enhancement," IEEE Trans. on Industrial Electronics, vol. 55, no. 9, pp. 3374-3380, Sept 2008.

[38] C.J. Price, I.D. Coope, and D. Byatt, "A Convergent Variant of the Nelder-Mead Algorithm," Journal of Optimization Theory and Applications, vol. 11, no. 3, pp. 5–19, 2002.

[39] L. Nazareth, P. Tseng, "Gilding the Lily: A Variant of the Nelder-Mead Algorithm Based on Golden-Section Search," Comput. Optim. Appl, vol. 22, no. 1, pp. 133–144, 2002.

[40] Steve Winder, "Analog and Digital Filter Design", Newnes 2002.

[41] M. R. Kobe, J. Ramirez-Angulo, and E. Sanchez-Sinencio, "FIESTA-A Filter Educational Synthesis Teaching Aid", IEEE Trans. Education, 32(3), pp. 280-286, August 1989.

[42] B. M. Wilamowski, "A Filter Synthesis Teaching-Aid", Rocky Mountain ASEE Section Meeting, Golden CO, USA, April 6, 1990.

[43] B. M. Wilamowski and R. Gottiparthy, "Active and Passive Filter Design with MATLAB", International Journal on Engineering Educations, vol. 21, No 4, pp. 561-571, 2005.

[44] B. M. Wilamowski, S. F. Legowski, and J. W. Steadman, "Personal Computer Support for Teaching Analog Filter Analysis and Design Courses", IEEE Trans. on Education, vol E-35, no 4, pp. 351-361, 1992.

[45] W. M. Anderson, B. M. Wilamowski, and G. Dundar, "Wide Band Tunable Filter Design Implemented in CMOS", 11th INES 2007 -International Conference on Intelligent Engineering Systems, Budapest, Hungary, pp. 219-223, June 29 2007-July 1 2007.

[46] W. Tangsrirat, T. Dumawipata and S. Unhavanich, “Realization of Low-pass and Band-pass Leapfrog Filters Using OAs and OTAs”, SICE 2003 Annual Conference, vol. 3, pp 4-6, 2003.

[47] Rolf Schaumann, M.E. Van Valkenburg, “Analog Filter Design”, Oxford 2001

[48] R. Koller, B. M. Wilamowski, "Simulation of Analog Filters Using Ladder Prototypes" proceedings of 23-Pittsburgh Conference on Modeling and Simulations, Pittsburgh, USA, April 30 - May 1, vol 23, part 4. pp. 1755-1761, 1992

[49] T. C. Fry, “The Use of Continued Fractions in the Design of Electrical Networks”, American Mathematical Society, pp. 463-498, 1929.

[50] Marcin Jagiela and B.M. Wilamowski “A Methodology of Synthesis of Lossy Ladder Filters” 13-th IEEE Intelligent Engineering Systems Conference, INES 2009, Barbados, April 16-18., 2009, pp. 45-50.

[51] C. Kwan and F. L. Lewis, “Robust Back-stepping Control of Nonlinear Systems Using Neural Networks”, IEEE Trans. System, Man and Cybernetics. A, Syst.,Humans, vol. 30, no. 6, pp. 753–766, Nov. 2000.

[52] H. Miyamoto, K. Kawato, T. Setoyama, and R. Suzuki, “Feedback-error Learning Neural Network for Trajectory Control of a Robotic Manipulator”, IEEE Trans. on Neural Network, vol. 1, no. 3, pp. 251–265, 1988.

[53] Y. Fukuyama, Y. Ueki, “An Application of Neural Networks to Dynamic Dispatch Using Multi Processors”, IEEE Trans. on Power Systems, vol. 9, no. 4, pp. 1759-1765, 1994.

[54] G. Indiveri, E. Chicca, R. Douglas, "A VLSI Array of Low-power Spiking Neurons and Bi-stable Synapses with Spike-timing Dependent Plasticity", IEEE Trans. on Neural Networks, vol. 17, no. 1, pp. 211-221, Jan 2006.

[55] B. M. Wilamowski, “Neural Network Architectures and Learning Algorithms”, IEEE Industrial Electronics Magazine, vol. 3, no. 4, pp.56-63, 2009.

[56] R. A. Jacobs, “Increased Rates of Convergence through Learning Rate Adaption”, IEEE Trans. on Neural Network, vol. 5, no. 1, pp. 295-307, 1988.

[57] T. Tollenaere, “SuperSAB: Fast Adaptive Back Propagation with Good Scaling Properties”, IEEE Trans. on Neural Networks, vol. 3, no. , pp. 561-573, 1990.

[58] R. Salomon, J. L. Van Hemmen, “Accelerating Back Propagation through Dynamic Self-Adaption”, IEEE Trans. on Neural Networks, vol. 9, no. , pp.589-601, 1996.

APPENDIX

APPENDIX 1: Nelder Mead's simplex method

```
function [f_BEST,BEST]=nelder_mead_nd(obj,x0,d_SIM,df_min,ite_max,times)
%   INPUT ARGUMENTS:
%   nelder_mead_nd(@testf1,[100,100],1,1e-4,2e2,100)
%   obj        - Handle of objective function.
%   x0         - Initial starting point.
%   d_SIM   - Size of initial simplex.
%   df_min  - Minimum improvement required for termination.
%   ite_max - Desired number of iterations.

%  OUTPUT ARGUMENTS:
%   BEST       - Location of baest solution.
%   f_BEST     - Best value of the objective found.
%   SIMPLEX - Matrix conatining final simplex.
%   f              - Objective values for each point in the simplex.

tavg_ite=0;
tsecond=0;
second=0;
succ_time=0;
avg_ite=0;
avg_time=0;
avg_error=0;
average_min=0;
%Initialize parameters and create simplex
for itee=1:times,   %training timesa=1;
   tic;
   a=1;
   b=2;
   c=0.5;
   n=length(x0);
   X0=ones(n,1)*x0;
   SIMPLEX=[X0+diag(d_SIM*(rand(1,n)));x0]; % create simplex vertices
   f(n+1)=0;
   f_mid(n)=0;
   mid=zeros(n)
```

```
for init=1:n+1
      f(init)=feval(obj,SIMPLEX(init,:));
   end
   init=0;
   SIMPLEX(:,end+1)=f';
   SIMPLEX=sortrows(SIMPLEX,n+1);   %sort row depending of value of f in ascending order;

   f=SIMPLEX(:,end)';
   SIMPLEX(:,end)=[];

   %% Simplex Code
   for ite=1:ite_max,
      Pb=sum(SIMPLEX(1:n,:))/n;      %calculate the centroid P_ of points with i#h
      Ps=(1+a)*Pb-a*SIMPLEX(end,:);    %calculate reflection point of Ph:Ps
      f_Ps=feval(obj,Ps);

      if f_Ps<f(1)    %f(P*)<f(l)
         Pss=(1-b)*Pb+b*Ps;   %calculate P** by expansion
         f_Pss=feval(obj,Pss);
         if f_Pss<f(1)    %f(P**)<f(l)
            SIMPLEX(end,:)=Pss;      %replace Ph by P**
            f(end)=f_Pss;
         else
           SIMPLEX(end,:)=Ps;        %replace Ph by P*
           f(end)=f_Ps;
         end
      else
         check=0;
         for i=1:n,
            if f_Ps>f(i)      % f_P*>f_i and i#h
               check=1;
               break;
            end
         end

         if check==0
            SIMPLEX(end,:)=Ps;        %replace Ph by P*
            f(end)=f_Ps;
         else
            if f_Ps>f(end)    %f_P*>f_h
               Pss=c*SIMPLEX(end,:)+(1-c)*Pb; %calculate P** by expansion
               f_Pss=feval(obj,Pss);
               if f_Pss>f(end)    %f(P**)>f(h)
                  for i=1:n+1
                     SIMPLEX(i,:)=(SIMPLEX(i,:)+SIMPLEX(1,:))/2; %replace all Pi' by (Pi+Pl)/2
```

```
                f(i)=feval(obj,SIMPLEX(i,:));
              end
            else
              SIMPLEX(end,:)=Pss;      %replace Ph by P**
              f(end)=f_Pss;
            end
          else
            SIMPLEX(end,:)=Ps;       %replace Ph by P*
            f(end)=f_Ps;
          end
        end

    end

    % reorder and display iteration output
    SIMPLEX(:,end+1)=f';
    SIMPLEX=sortrows(SIMPLEX,n+1);

    f=SIMPLEX(:,end)';
    SIMPLEX(:,end)=[];
   %calculate error
    error(ite)=f(1);
    t(ite)=ite;

   % terminate condition3 for neural network training
    if f(1)<df_min,
       succ_time=succ_time+1;
       avg_ite=avg_ite+ite;
       avg_time=avg_time+1;
       avg_error=avg_error+f(1);
       second=second+toc;
       break;
    end
 end;

% display the result
succ_rate=succ_time/times;
BEST=SIMPLEX(1,:);
f_BEST=f(1);
average_min=average_min+f_BEST;
tavg_ite=tavg_ite+ite;
tsecond=tsecond+toc;
disp(' ');
disp(['Minimum value of f = ',num2str(f_BEST),])
disp(['located at x = [',num2str(BEST),'].'])
```

```
    disp(['Success rate = [',num2str(succ_rate),'].'])
    %plot
    semilogy(t,error,'-r');
    xlabel('Iterations')
    ylabel('Error')
    title('Error Plot')
    ax=axis; ax(3)=0; ax(2)=110; axis(ax);
    hold on;
end

    avg_iteration=avg_ite/avg_time;
    avg_errors=avg_error/avg_time;
    avg_second=second/avg_time;
    tavg_iteration=tavg_ite/times;
    avg_minimum=average_min/times;
    avg_tsecond=tsecond/times;
    disp(['Average Iteration = ',num2str(avg_iteration),])
    disp(['Average Error = ',num2str(avg_errors),])
    disp(['Average second = ',num2str(avg_second),])
    disp(['tAverage Iteration = ',num2str(tavg_iteration),])
    disp(['tAverage Minimum = ',num2str(avg_minimum),])
    disp(['tAverage second = ',num2str(avg_tsecond),])
return
```

APPENDIX 2: Improved simplex method with quasi- gradient method using an extra vertex

```
function [f_BEST,BEST]=nelder_mead_ndmd1(obj,x0,d_SIM,df_min,ite_max,times)
%   INPUT ARGUMENTS:
%   nelder_mead_ndmd1(@testf1,[100,100],1,1e-4,2e2,100)
%   obj        - Handle of objective function.
%   x0         - Initial starting point.
%   d_SIM   - Size of initial simplex.
%   df_min  - Minimum improvement required for termination.
%   ite_max - Desired number of iterations.

%  OUTPUT ARGUMENTS:
%   BEST       - Location of baest solution.
%   f_BEST     - Best value of the objective found.
%   SIMPLEX - Matrix conatining final simplex.
%   f              - Objective values for each point in the simplex.

format long;
tavg_ite=0;
tsecond=0;
second=0;
succ_time=0;
avg_ite=0;
avg_time=0;
avg_error=0;
average_min=0;
%% Initialize parameters and create simplex
for itee=1:times,   %training timesa=1;
   tic;
   alpha=1;
   a=1;
   b=2;
   c=0.5;
   n=length(x0);
   mo=zeros(1,n);
   mu=0.1;
   X0=ones(n,1)*x0;
   SIMPLEX=[X0+diag(d_SIM*(rand(1,n)));x0]; % create simplex vertices
   f(n+1)=0;
   f_mid(n)=0;
   mid=zeros(n);

   for init=1:n+1
      f(init)=feval(obj,SIMPLEX(init,:));
   end
```

```
init=0;
SIMPLEX(:,end+1)=f';
SIMPLEX=sortrows(SIMPLEX,n+1);    %sort row depending of value of f in ascending order;

f=SIMPLEX(:,end)';
SIMPLEX(:,end)=[];

%% Simplex Code
for ite=1:ite_max,

    Pb=sum(SIMPLEX(1:n,:))/n;       %calculate the centroid P_ of points with i#h
    Ps=(1+a)*Pb-a*SIMPLEX(end,:);    %calculate reflection point of Ph:Ps
    f_Ps=feval(obj,Ps);
    Pss=(1-b)*Pb+b*Ps;    %calculate P** by expansion
    f_Pss=feval(obj,Pss);

    if f_Ps>f(1)

        %% using composite point
        for i=1:n,
          ord(i)=SIMPLEX(i,i);
        end
        f_ord=feval(obj,ord);
        for i=1:n,
          if mod(i,2)==0
             grad(i)=(f(i-1)-f_ord)/(SIMPLEX(i-1,i)-ord(i));
          else
             grad(i)=(f(i+1)-f_ord)/(SIMPLEX(i+1,i)-ord(i));
          end
        end
        Gs=SIMPLEX(1,:)-alpha*grad/sqrt(sum(grad.^2));

        % Calculate reflected point
        P3=(1+a)*SIMPLEX(1,:)-SIMPLEX(end,:);
        P1=SIMPLEX(1,:);
        P2=Gs;
        PP=(P3-P1).*(P2-P1);
        u=sum(PP)/sum((P2-P1).^2);
        Gs=P1+u*(P2-P1);
        f_Gs=feval(obj,Gs);

        if f_Gs<f_Ps
           Ps=Gs;            %new reflected point
           f_Ps=f_Gs;
           Pb=SIMPLEX(1,:);
```

```
        Pss=(1-b)*SIMPLEX(1,:)+b*Ps;   %calculate P** by expansion
        f_Pss=feval(obj,Pss);
    end
end
if f_Ps<f(1)    %f(P*)<f(l)
     if f_Pss<f(1)    %f(P**)<f(l)
         SIMPLEX(end,:)=Pss;      %replace Ph by P**
         f(end)=f_Pss;
     else
        SIMPLEX(end,:)=Ps;        %replace Ph by P*
        f(end)=f_Ps;
     end
else
    check=0;
    for i=1:n,
        if f_Ps>f(i)      % f_P*>f_i and i#h
            check=1;
            break;
        end
    end
    if check==0
        SIMPLEX(end,:)=Ps;        %replace Ph by P*
        f(end)=f_Ps;
    else
        if f_Ps>f(end)    %f_P*>f_h
            Pss=c*SIMPLEX(end,:)+(1-c)*Pb; %calculate P** by expansion
            f_Pss=feval(obj,Pss);
            if f_Pss>f(end)    %f(P**)>f(h)
                for i=1:n+1
                  SIMPLEX(i,:)=(SIMPLEX(i,:)+SIMPLEX(1,:))/2; %replace all Pi' by (Pi+Pl)/2
                  f(i)=feval(obj,SIMPLEX(i,:));
                end
            else
              SIMPLEX(end,:)=Pss;      %replace Ph by P**
              f(end)=f_Pss;
            end
        else
          SIMPLEX(end,:)=Ps;        %replace Ph by P*
          f(end)=f_Ps;
        end
    end
end

% reorder and display iteration output
SIMPLEX(:,end+1)=f';
```

```
        SIMPLEX=sortrows(SIMPLEX,n+1);
        f=SIMPLEX(:,end)';
        SIMPLEX(:,end)=[];
        error(ite)=f(1);
        t(ite)=ite;
        % terminate condition3 for neural network training
        if f(1)<df_min,
           succ_time=succ_time+1;
           avg_ite=avg_ite+ite;
           avg_time=avg_time+1;
           avg_error=avg_error+f(1);
           second=second+toc;
           break;
        end
     end;
    % display the result
    succ_rate=succ_time/times;
    BEST=SIMPLEX(1,:);
    f_BEST=f(1);
    average_min=average_min+f_BEST;
    tavg_ite=tavg_ite+ite;
    tsecond=tsecond+toc;
    disp(' ');
    disp(['Minimum value of f = ',num2str(f_BEST),])
    disp(['located at x = [',num2str(BEST),'].'])
    disp(['Success rate = [',num2str(succ_rate),'].'])
    %plot
    semilogy(t,error,'b');
    xlabel('Iterations')
    ylabel('Error')
    hold on;
end
    avg_iteration=avg_ite/avg_time;
    avg_errors=avg_error/avg_time;
    avg_second=second/avg_time;
    tavg_iteration=tavg_ite/times;
    avg_minimum=average_min/times;
    avg_tsecond=tsecond/times;
    disp(['Average Iteration = ',num2str(avg_iteration),])
    disp(['Average Error = ',num2str(avg_errors),])
    disp(['Average second = ',num2str(avg_second),])
    disp(['tAverage Iteration = ',num2str(tavg_iteration),])
    disp(['tAverage Minimum = ',num2str(avg_minimum),])
    disp(['tAverage second = ',num2str(avg_tsecond),])
return
```

APPENDIX 3: Improved simplex method with quasi-gradient method using a hyper plane equation

```
function [f_BEST,BEST]=nelder_mead_ndmd2(obj,x0,d_SIM,df_min,ite_max,times)
%   INPUT ARGUMENTS:
%   nelder_mead_ndmd2(@testf1,[100,100],1,1e-4,2e2,100)
%   obj        - Handle of objective function.
%   x0         - Initial starting point.
%   d_SIM   - Size of initial simplex.
%   df_min  - Minimum improvement required for termination.
%   ite_max - Desired number of iterations.

%  OUTPUT ARGUMENTS:
%   BEST    - Location of baest solution.
%   f_BEST  - Best value of the objective found.
%   SIMPLEX - Matrix conatining final simplex.
%   f       - Objective values for each point in the simplex.

format long;
tavg_ite=0;
tsecond=0;
second=0;
succ_time=0;
avg_ite=0;
avg_time=0;
avg_error=0;
average_min=0;
% Initialize parameters and create simplex
for itee=1:times,   %training timesa=1;
   tic;
   alpha=1;
   a=1;
   b=2;
   c=0.5;
   n=length(x0);
   mo=zeros(1,n);
   mu=0.1;
   X0=ones(n,1)*x0;
   SIMPLEX=[X0+diag(d_SIM*(rand(1,n)));x0]; % create simplex vertices
   f(n+1)=0;
   f_mid(n)=0;
   mid=zeros(n);

   for init=1:n+1
```

```
   f(init)=feval(obj,SIMPLEX(init,:));
end
init=0;
SIMPLEX(:,end+1)=f';
SIMPLEX=sortrows(SIMPLEX,n+1);    %sort row depending of value of f in ascending order;

f=SIMPLEX(:,end)';
SIMPLEX(:,end)=[];

% Simplex Code
for ite=1:ite_max,

    Pb=sum(SIMPLEX(1:n,:))/n;       %calculate the centroid P_ of points with i#h
    Ps=(1+a)*Pb-a*SIMPLEX(end,:);    %calculate reflection point of Ph:Ps
    f_Ps=feval(obj,Ps);
    Pss=(1-b)*Pb+b*Ps;    %calculate P** by expansion
    f_Pss=feval(obj,Pss);

    if f_Ps>f(1)
       % using hyper plane equation
       I=SIMPLEX(:,1:n);
       A=ones(1,n+1)';
       A(:,2:n+1)=I;
       B=f';
       P=pinv(A)*B;
       grad=P';
       Gs=SIMPLEX(1,:)-alpha*grad(1,2:n+1);
       % Calculate reflected point
       P3=(1+a)*SIMPLEX(1,:)-SIMPLEX(end,:);
       P1=SIMPLEX(1,:);
       P2=Gs;
       PP=(P3-P1).*(P2-P1);
       u=sum(PP)/sum((P2-P1).^2);
       Gs=P1+u*(P2-P1);
       f_Gs=feval(obj,Gs);

       if f_Gs<f_Ps
          Ps=Gs;            %new reflected point
          f_Ps=f_Gs;
          Pb=SIMPLEX(1,:);
          Pss=(1-b)*SIMPLEX(1,:)+b*Ps;   %calculate P** by expansion
          f_Pss=feval(obj,Pss);
       end

    end
```

```
if f_Ps<f(1)    %f(P*)<f(l)
     if f_Pss<f(1)    %f(P**)<f(l)
        SIMPLEX(end,:)=Pss;      %replace Ph by P**
        f(end)=f_Pss;
     else
       SIMPLEX(end,:)=Ps;       %replace Ph by P*
       f(end)=f_Ps;
     end
else
   check=0;
   for i=1:n,
      if f_Ps>f(i)      % f_P*>f_i and i#h
         check=1;
         break;
      end
   end
   if check==0
      SIMPLEX(end,:)=Ps;       %replace Ph by P*
      f(end)=f_Ps;
   else
      if f_Ps>f(end)    %f_P*>f_h
         Pss=c*SIMPLEX(end,:)+(1-c)*Pb; %calculate P** by expansion
         f_Pss=feval(obj,Pss);
         if f_Pss>f(end)    %f(P**)>f(h)
            for i=1:n+1
              SIMPLEX(i,:)=(SIMPLEX(i,:)+SIMPLEX(1,:))/2; %replace all Pi' by (Pi+Pl)/2
              f(i)=feval(obj,SIMPLEX(i,:));
            end
         else
           SIMPLEX(end,:)=Pss;      %replace Ph by P**
           f(end)=f_Pss;
         end
      else
        SIMPLEX(end,:)=Ps;       %replace Ph by P*
        f(end)=f_Ps;
      end
   end
end

% reorder and display iteration output
SIMPLEX(:,end+1)=f';
SIMPLEX=sortrows(SIMPLEX,n+1);

f=SIMPLEX(:,end)';
```

```
        SIMPLEX(:,end)=[];
        error(ite)=f(1);
        t(ite)=ite;

        % terminate condition3 for neural network training
        if f(1)<df_min,
           succ_time=succ_time+1;
           avg_ite=avg_ite+ite;
           avg_time=avg_time+1;
           avg_error=avg_error+f(1);
           second=second+toc;
           break;
        end
     end;
    % display the result
    succ_rate=succ_time/times;
    BEST=SIMPLEX(1,:);
    f_BEST=f(1);
    average_min=average_min+f_BEST;
    tavg_ite=tavg_ite+ite;
    tsecond=tsecond+toc;
    disp(' ');
    disp(['Minimum value of f = ',num2str(f_BEST),])
    disp(['located at x = [',num2str(BEST),'].'])
    disp(['Success rate = [',num2str(succ_rate),'].'])
    % plot
    semilogy(t,error,'b');
    xlabel('Iterations')
    ylabel('Error')
    hold on;
end
    avg_iteration=avg_ite/avg_time;
    avg_errors=avg_error/avg_time;
    avg_second=second/avg_time;
    tavg_iteration=tavg_ite/times;
    avg_minimum=average_min/times;
    avg_tsecond=tsecond/times;
    disp(['Average Iteration = ',num2str(avg_iteration),])
    disp(['Average Error = ',num2str(avg_errors),])
    disp(['Average second = ',num2str(avg_second),])
    disp(['tAverage Iteration = ',num2str(tavg_iteration),])
    disp(['tAverage Minimum = ',num2str(avg_minimum),])
    disp(['tAverage second = ',num2str(avg_tsecond),])
return
```

APPENDIX 4: Test function

```
function [f]=testf1(x,c)
% Robot arm training
f=0;
p=length(c);
gain=0.5;
for i=1:p
    % three neurons
    f1=tanh(gain*(x(1)*c(i,1)+x(2)*c(i,2)+x(3)));
    f2=tanh(gain*(x(4)*c(i,1)+x(5)*c(i,2)+x(6)*f1+x(7)));
    f=f+(1/p)*(tanh(gain*(x(8)*c(i,1)+x(9)*c(i,2)+x(10)*f1+x(11)*f2+x(12)))-c(i,3))^2;
end
return
```