GENETIC ALGORITHM AND PARTICLE SWARM OPTIMIZATION TECHNIQUES IN MULTIMODAL FUNCTIONS OPTIMIZATION

A THESIS SUBMITTED TO THE GRADUATE SCHOOL OF APPLIED SCIENCES OF NEAR EAST UNIVERSITY

By
SAYED RAZI ABBAS

In Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

NICOSIA 2017

Sayed Razi ABBAS: GENETIC ALGORITHM AND PARTICLE SWARM OPTIMIZATION TECHNIQUES IN MULTIMODAL FUNCTIONS OPTIMIZATION

Approval of director of Graduate school of Applied Sciences

Prof. Dr. Nadire ÇAVUŞ

We certify this thesis is satisfactory for the award of the degree of Master of Science in Computer Engineering

Examining Committee in Charge:

Prof.Dr. Rahib Abiyev	Committee Chairman, Department of Computer Engineering, NEU
Prof. Dr. Adil Amirjanov	Supervisor, Department of Computer Engineering, NEU
Assist. Prof. Dr. Ali Danandeh Mehr	Department of Civil Engineering, NEU

I hereby state that all material in this document has been gained and conferred in accordance with academic rules and moral conduct. I also announce that, as required by these rules and conduct, I have fully cited and referenced all data and results that are not genuine to this work.

Name, Last name: SAYED RAZI ABBAS

Signature:

Date:

ACKNOWLEDGMENTS

Firstly, I would like to indicate my sincere gratitude to my advisor Prof. Dr. Adil Amirjanov, for the continuous support of my Master study and research, for his patience, motivation, ambition, and massive knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have visualized having a better advisor and mentor for my Master study.

Besides my advisor, I would like to thank the rest of faculty staff and jury members, for their encouragement.

Finally, I take this opportunity to express the profound gratitude from my deep heart to my beloved parents and brothers for their love and continuous support.

ABSTRACT

Evolutionary algorithms (EA's) are well-known mechanisms for optimization problems that involve frequently conflicting objectives. From the year of 1985, there have been some evolutionary attempts which succeeded in solving multimodal optimization problems.

There are two techniques that are already up to the task of optimization - a sequential niche genetic algorithm (SNGA) and a novel adaptive sequential niche technique with particle swarm optimization (ASNPSO) for multimodal function optimization.

SNGA technique is using a simple genetic algorithm (GA) and obtains information in one iteration. It is succeeded by applying a fitness derating function to the raw fitness function, which leads to the fitness values being depressed in the regions of the problem space where solutions have already been found. The effectiveness of the algorithm is tested by some multimodal test functions. The method is at least as fast as fitness sharing methods.

On the other hand, ASNPSO also is used for same purpose of multimodal function optimization. It is using a particle swarm optimization instead of simple GA that is detecting optimal solutions sequentially. In the ASNPSO method, the hilly-valley function is used to determine how to change a fitness of a particle in a sub-swarm run. The algorithm's search ability is strong and adaptive.

The purpose of both algorithms is to optimize multimodal problems with different approaches and I subjected both of them to test with different functions and compared both techniques to obtain knowledge of which one gives more accurate optimal solutions.

Keywords: Genetic algorithm; niche technique; fitness sharing; fitness derating function; multimodal optimization

ÖZET

Evrimsel algoritmalar (EA'lar) sıklıkla çelişen amaçlarla ilgili olan optimizasyon problemleri için bilinen mekanizmalardır. 1985 yılından itibaren, multimodal optimizasyon problemlerini çözmeyi başaran bazı evrimsel girişimler olmuştur.

Optimizasyon görevini yerine getiren iki teknik vardır: ardışık niş genetik algoritması (SNGA) ve çok modlu fonksiyon optimizasyonu için parçacık sürüsü optimizasyonuna (ASNPSO) sahip yeni bir uyarlanabilir sıralı niş tekniği.

SNGA tekniği, basit bir genetik algoritma (GA) kullanmaktadır ve bir yinelemede bilgi almaktadır. Bunun sonucunda, çiğ uygunluk fonksiyonuna bir zindelik düşürme fonksiyonu uygulanır ve sonuç olarak, zihniyet değerlerinin çözümlerin daha önce bulunduğu problem alanının bölgelerinde baskılanmasına yol açar. Algoritmanın etkinliği, bazı çok modlu test fonksiyonları tarafından test edilmiştir. Yöntem en az fitness paylaşım yöntemleri kadar hızlıdır.

Öte yandan, ASNPSO da aynı amaçla multimodal fonksiyon optimizasyonu için kullanılır. Ardışık optimal çözümleri saptayan basit GA yerine parçacık sürüsü optimizasyonunu kullanıyor. ASNPSO yönteminde tepelik-vadiye fonksiyonu, bir alt sürü koşusunda bir parçacık şeklinin nasıl değiştirileceğini belirlemek için kullanılır. Algoritmanın arama becerisi güçlü ve uyarlanabilir.

Her iki algoritmanın amacı, multimodal problemleri farklı yaklaşımlarla optimize etmektir ve her ikisini de farklı fonksiyonlarla test etmeye tabi tuttum ve her iki tekniği karşılaştırarak hangisinin daha doğru en iyi çözümleri sunduğunu öğrentim.

Anahtar Kelimeler: Genetik algoritma; Niş tekniği; Fitness paylaşımı; Fitness indirgeme fonksiyonu; Çok modlu optimizasyon

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

1.1 What is Optimization?

Our life is filled with problems, these problems are the driving force for our inventions and environmental enhancement strategies. In computer science optimization is a computer process based process used to find solution to complex problems. For example, if we want to find the maximum peak for any function, then we need to formulize the precepts for a solution to be recognized as an optimum corresponding to our aim of finding either global optima, or local optima. Nevertheless, we may use constraint to push the algorithms to a feasible peak by suit of constraints, and if we want to make things more difficult we will use mixed constraint types, such as equality and inequality constraints. Finally, optimization can be defined as optimization is to find an algorithm which solves a given class of problem.

In mathematics we can use derivatives or differentiation to find an optima, but not all function are continuous and differentiable. In general, the non-linear programming is to find $\vec{x}$ so as to optimize $f(\vec{x}), \vec{x} = (x_1, x_2, \cdots, x_n) \in \Re^n$, where $\vec{x} \in F \subseteq S$. The objective function $f(\vec{x})$ is defined in search space S, the set $F \in S$ define the feasible region, usually S is defined in n dimensional space from the global space $\Re^n$. Every vector $\vec{x}$ has domain boundaries, where $l(i) \leq x_i \leq u(i), 1 \leq i \leq n$, and the feasible region is defined by a set of constraints.

Inequality constraints, $g_i(x) \leq 0$, and equality constraints $h_j(x) = 0$. Those inequality constraints could be equal to zero then they are called active; however, the equality constraints are always active and equal to zero in the entire of search space. Some researchers were focused in local optima, where the point $\vec{x}$ is local optima $\Leftrightarrow$ there exist ε such that. For all $\vec{x}_0$ in the ε neighborhood of $\vec{x}_0$ in F, $f(\vec{x}) > f(\vec{x}_0)$. Finally, evolutionary algorithms are contrasting the mathematical derivatives to be a global optimizer method with complex objective functions when mathematics fails to give a sensible solution because of the complexity of the hyperspace or function discontinuity elsewhere (Michalewicz & Schoenauery, 1996).

Evolutionary computing is often used to solve such complicated problems, where the boundaries of the feasible region are so strict; whereas, genetic algorithms are an expert optimization method. Its chromosomal representation can be continuous or discreet. Genetic algorithms can be used for complex optimization problem; since, they are not attracted by the shape or the complexity of the objective function. By adding the constraints functions for the infeasible chromosome it can enforce those individuals to be feasible, or it may give them cost to be feasible. On the other hand, the feasible chromosomes have no addition or subtraction from their objective function value. This criterion will enhance the feasible solution and penalize infeasible no matter the shape of the function. Discontinuity is the second problem genetic algorithms can avoid; since, the value of constraints will avoid it.

The field of EA is encircle of genetic algorithms (GAs) and many other methods, here I will describe two methods which is already proposed in my abstract. Before to go in detail of SNGA and ASNPSO I need to explain genetic algorithm and Particle swarm optimization methods.

1.2. Thesis Overview

This thesis is organized in incremental method. Starting with simple and moving to more complicate declaration depending on the issues.

Chapter 2: Discusses Genetic Algorithms framework, structure, and it basic operation.

Chapter 3: Description of different methods in GA and detail of SNGA.

Chapter 4: Discusses particle swarm optimization and it basic operation and ASNPSO.

Chapter 5: Experimental results of 8 functions which are used in SNGA and ASNPSO.

Chapter 6: Analysis and comparison of SNGA and ASNPSO with statistical data.

Chapter 7: Discussion defend on the analysis and comparison.

Chapter 8: Conclusion.

CHAPTER 2

INTRODUCTION TO GENETIC ALGORITHM

Study of mathematical optimization a genetic algorithm is a best and generally used global search method that duplicate the process of natural election. It generates practical solutions to optimization. Genetic algorithms are the large part of evolutionary algorithms (EA), that produce solution to optimization issues using method achieved by natural evolution. (Goldberg, 1989), Genetic algorithm is finding algorithm depend on the logistics of natural election and natural ancestral. They together survival of the fittest among string structures with a structured up to now randomized facts conversion to form a search algorithm with some of the original talent of human search. In every generation, it created new strings. The new string is used for good measure and used for new generation. Genetic algorithm have been developed by (Holland, 1975), the research have been twofold (1) to hypothetical and carefully show the adaptive operation of natural structures, and (2) to create artificial structure software that absorb the important mechanisms of natural and artificial structures science. The research on genetic algorithm has been well made for survival in many different environment. The implications of well made for artificial systems are manifold. If artificial structure can be made better, costly modernize can be decreased or eliminated. If best result found then the actual structures can perform their function longer and batter. Features for self-repair, self-direction, and breeding are the rule in biological system, where they exist in the most knowledgeable artificial systems (Sivanandam & Deepa, 2008). It is also the most popular stochastic optimization methodology used now a day. The basic of GA is Charles Darwin theory of "survival of the fittest", where types must adapt to their territory to survive. Elements with fittest natural traits will have a best capacity and chance to persist. They will also have more preference for reproduction and changing their phenotype and genotype to next generations. GA basic are the genes that carry a set of genes, where in phenotype represent element a single gene. Element have upper and lower bounds that show the minimum and maximum adaptive fitness in phenotype for applicant. Genes can provide solutions or near result for the global problem. Simultaneously, gene length makes scale of participate specific factor set; for example, if gene length is equal to n, then it can represent 2^{n-1}

binary strings (Sivanandam & Deepa, 2008), those can be encoded for (length) $2^{n}-1$ (Reeves & Rowe, 2002). On the other side, every gene has one or more genes, those genes will be stored in a single attitude. The set of all genes will represent a single element. (Holland, 1975) explained genetic algorithm for finding nonlinear problems. GA is problem determined by as there are many limitation for individual representation i.e. binary representation because our aim is to guarantee that GA exactly and absolutely represents all feasible alleles for every point of the search space. These values for genes will represent the genotype that makes direct image for phenotype where we will compute the result as stated to their fitness. In general if can tack values by a decision variable between a , b , and $b > a$, and if it's designed to binary string of length L , then the accuracy will be computed in the next equation (Reeves & Rowe, 2002), where x is best gene width.

Another technique was placed for binary representation of elements, such as gray coding, where the hamming cliff is decrease to one rather than standard binary representation. The chance that at least one gene is presented at each part can be defined in the next equation (Sharda & Voß, 2002).

GA basic runs are initialization, selection, crossover, mutation, termination, population replacement, and fitness evaluation. Figure 1 showed GA flowchart.

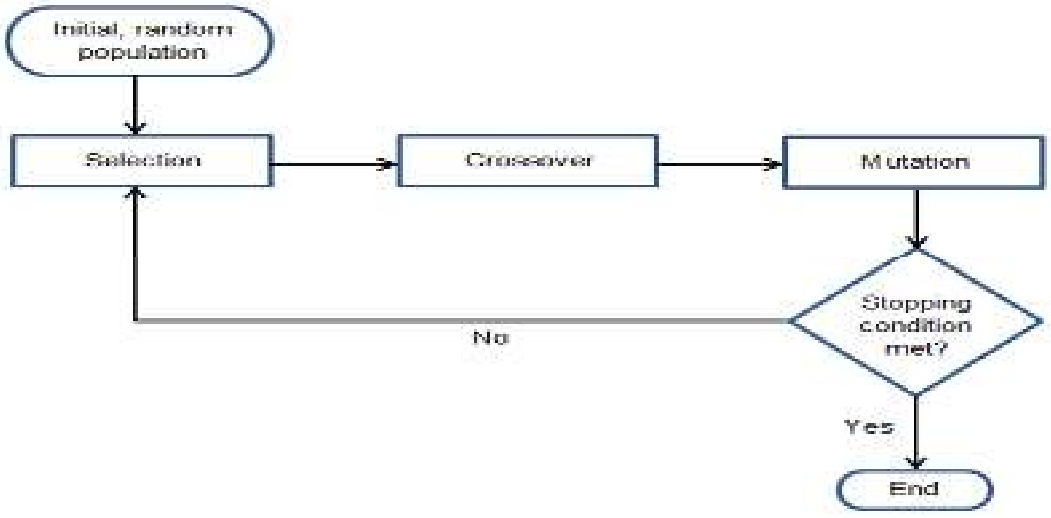

Figure 1: Genetic Algorithms Flowchart

2.1 Binary Representation

The GA process was essentially initiate by (Holland, 1975) was in binary, since as it was the natural chromosomes gene reproduction and its accessibility of applying the GA basic process. For example, to appoint a variable in decimal equal to 15 for a given problem, we start from 0 and start to give discrete values in limit by increment by one. Then we compute to appoint a number from 0 $(0000)_{binary}$ to 15 $(1111)_{binary}$ we need 4 bits. With this method the solution was awful. Basically three issues were highlighted.

A. Number of bit required: that every variable had its own domain D_i , which has lower and upper bound; e.g. problem G_5 let us staple sample of two variables $0 \le x_1 \le 1200$, here we have a problem of planning variable domain in binary level, circulate to flat binary bit matching to every variable. To show x_1, in trivial method we need 11 bit binary string to show it. In another word, $-0.55 \le x_3 \le 0.55$ how could be appointed to it in trivial technique. This planning issue has occur a problem which is defined as Big bound. We need to compute all variables into binary string that store them within boundaries. Sometimes there will be off bits inside, at the time how can we control the domain idea? For example, to represent $200_{decimal}$ in binary $(11001000)_{\text{binary}}$ as it shows, we have many unoccupied alleles, which can in total shift the search space for impractical solution. If we have several solid reconnection operations, either maximization or minimization issues all of them will be *0's or 1's* after a particular number of iterations. And if we need to represent *200* in fewer numbers of bits determine by only full (on) bits, we will have a value less than *200*, which is a loss of useful data points in search space. Either way will be inefficient for an accurate solution, within this bad status of binary reproduction and domain content; still, we can produce a temporary solution. For incase, uniform crossover, where crossover chance is taken separately for every bit substitute of chromosome in some way it's exchange for mutation. Moreover we suggest a methodology for building and retrieving values of variable with less difficulty and more exact results.

Assume we are going to maximize or minimize function, such that $R^i \subseteq R$, each variable x_i , can staple value from domain $D_i = [a_i, b_i]$, where $a_i < b_i$. If we need to optimize $f(x)$ with clarity, every one estate should be constructed by $(a_i - b_i) \times 10^n$, where n is the decimal precisions desired. Let us indicate m_i as the smallest threshold integer then $(b_i - a_i) \times 10^n \leq 2^{m_i} - 1$. For example, $0 \leq D_v \leq 3$, then $D_v = [3,0]$. Assume we need accuracy with degree 2; then $(3-0) \times 10^2 \leq 2^{m_i} - 1$, to represent 300×100 we need 9 bits which indicate the distinction will be as stated to Equation (1). Finally; in order to represent forecast with variable boundaries elsewhere (Goldberg, 1989).

$$v = a_i + decimal(binarystring) \times \frac{b_i - a_i}{2^n - 1} \tag{1}$$

B. To figure a more tricky framework where we have a variable with a big number and another variable with a mini domain. e.g. the same issue G_5 (see page **Error! Bookmark not defined.**) Where $-0.55 \leq x_3 \leq 0.55$, the query is how to represent variable domain that has negative range? Let us forecast scheme, if we use chance to be positive; or negative for exactly set of bit in chromosome, then that will be interest. Still, whenever we design a control matrix for variable negative and positive value that will be begun from the initializing by building variables in domain limit and checkup they token. By inherently of fixed variable rage i.e. stable token contrary of search process running, and forecast it will be the same token), both of them are awful in operation and mathematical evidence. Another issue to study is does all variables within the value of bits, want to be moved? The is answered Yes, I should do, because making proportional number of bits will build the process more tricky mistakes. Another question is how to recover an objective function value from the chromosome? Here we need more than one quality process for recovering variable values and of course we need more tricky designing of bits to ratio or real values. In last, variables are discreet and mainly the same for entire runs and search processes.

C. Re-construct binary string: following retrieving variables values and computing objective function, we want to apply GA process and penalty. Here one question is how to recover certain variable value from penalty function? Which methodology will use to construct binary string from its matching variables value? We have sketched more than one solution, but all inappropriate. Mainly the left most binary string values about zeroes; in contrast, mathematic optimization method debated before position is to deal only with positive binary strings. So then we can find the inverse of the given penalty function. And we already trying it in simple method that is maximized $x \in [0,31]$. The made objective function loses too many points out of the native function. The result is to adapt the value of penalized element, matching to the same Equation (1).

2.2 Selection

From population to be choosing a two parent chromosomes for crossing is called in GA a selection method. How many individuals will be selected? The set of questions needed to be acknowledged. Will they make a better quality! In case will be a better solution or not after a crossing point. From random population to be choose that called basic selection fully random method, exchanged, where separate objective functions are to be the chance of selection, this method is the soul of roulette wheel selection. Basically it's easy method but some more problem to be answered, e.g. the two parents from populations mating pool, how many new copies these two selected elements will copy themselves for the next population. This is the basic problem with selection method. There is a lot of solutions, that can easily solves these problems. Such as elements rank, fitness pressure balancing and scaling of fitness be based on nature of the issue. Such as, rank selection and steady-state selection many other methods were created to solve these problems. Selection and other assign of selection algorithm are diverting the basic part of convergence of the algorithm. There are two main types of selection, proportional selection, where fitness part is ratio apprehend from comprehensive population part, such as, selection of roulette wheel. The other type of selection is ordinal based selection, where fitness based on ranking (position) of part in population, and the first place taken for the worst individual. We have used binary tournament selection, since as its regularity and ability to give chance for worst individuals to take a part in selection method where their preference is very low. On the way, for limitation of optimization we can assess stochastic ranking as a selection process. Although it can't be identify as a complete one, because of its shortness to select part from the crossing pool. Another appreciative technique require that can make the agreement of selection.

2.2.1 Tournament Selection

Tournament selection works as follows, choose some number t of individuals randomly from the population and copy the best individual from this group into the intermediate population and repeat N times Often tournaments are held only between two individuals (binary tournament) but a generalization is possible to an arbitrary group size t called tournament size. Figure 4 shows the basic tournament selection pseudo code (Blickle & Thiele, 1997).

```
Algorithm: Tournament Selection
Input: the population $P(\tau)$ the tournament size $t \in \{1,2,\cdots\cdots,N\}$
Output: population after Selection
Tournament$(t, J_1, \cdots\cdots, J_N)$
        for $i \leftarrow 1$ to N do
                $J'_j \leftarrow$ best individual out of τ randomly picked individuals
        from
        od
Return $\{J'_1, \cdots\cdots, J'_N\}$
```

Figure 2: Tournament selection pseudo codes (Blickle & Thiele, 1997)

2.2.2 Roulette Wheel Selection

Roulette Wheel is the most familiar selection method used in GA. It starts by selecting elements linearly from the mating pool. However, cumulative element objective function is summed, and the average of fitness is calculated. By using the sum of fitness, individual's fitness is divided sequentially with the total probability of one. Individuals are captured in the roulette space proportional to their fitness. Meanwhile, the number of times individual can be selected is proportional to the average of fitness. When comparing with other method this method has disadvantage. It's hardly dependent on individual objective function, which allows the best individuals to manipulate the mating pool. On the other hand, it will encourage the algorithm for less exploration for infeasible search space from the beginning. However, those individuals may have a solution. Finally, scaling of fitness and other techniques are used to make less impact of fittest individuals for the search process. Figure 2.3.1.1 declares RWS algorithm pseudo code (Goldberg, 1989).

```
Algorithm: roulette wheel selection
Input: the population $P$
Output: population after selection $P^*$
X= random ]0,1[
$i \leftarrow 1$
while $i \neq n$ do
    If i<m & $x < \frac{f(b_i,t)}{\sum_{i=1}^{n} f(b_i,t)}$ then
        $i \leftarrow i+1$
        select $b_i,t$
    fi
od
Return $P^*$
```

Figure 3: Roulette Wheel Selection Algorithms

In contrast with proportion selection, linear ranking selection is based in position, where individual are sorted with respect to problem in hand. Meanwhile the first position will be reserved for the worst element. The positions of the population will have constant probability to be selected with respect to the Equation (2) (Blickle & Thiele, 1997), where linear function will be constructed. The probability of worst individual to be selected will be $\frac{\eta^-}{N}$ (Blickle & Thiele, 1997), and the best will be $\frac{\eta^+}{N}$ (Blickle & Thiele, 1997). The value of η^- must be in between [0,1]; on the other hand, the value of η^+ will be calculated by $\eta^{+} = 2 - \eta^{-}$ (Blickle & Thiele, 1997), where η^{-} value will determine the probability of worst individual to participate in selection process and N is population size and i is the index of element. Figure 2.3.2.1 is illustrates the pseudo code for linear ranking selection algorithm (Blickle & Thiele, 1997).

$$p_i = \frac{1}{N}\left(\eta^{+} + \left(\eta^{+} - \eta^{-}\right)\frac{i-1}{N-1}\right) \tag{2}$$

```
Input: the population P (τ) and the production rate of worst individual $\eta^{-} \in [0,1[$
Output: the population after selection $P(\tau)'$
Linear ranking $(\eta^{-}, J_1, \cdots\cdots, J_n)$
    $\bar{J}$ ← sorted population according to fitness with worst individual at the first position
    $S_0 \leftarrow 0$
    For $i \leftarrow 1$ to N do
        $S_i \leftarrow S_{i-1} + P_i$ ,where $P_i$ value is calculated in equation(2)
    od
    For $i \leftarrow 1$ to N do
        $r \leftarrow random \ [0, S_n]$
        $J_i^{'} \leftarrow \bar{J}$, such that $S_{i-1} \le r \le S_i$
    Od
    Return $\{J_1^{'}, \cdots\cdots, J_N^{'}\}$
```

Figure 4: Linear ranking selection pseudo code (Blickle & Thiele, 1997)

2.3 Crossover

Crossover is the production method that uses exploitation to shift the search process for better region of the search space. It can positively make new elements that are improved their parents by posting their quality into new baby's. It can only reconstruction ancestor's attribute externally any construction of new attribute. For every element GA are flowing to allow incidental for crossover, lean on every element, those individual will be post into the crossing pool. The typical chance of crossover P_c will be even for the whole of the method and = to (0.5-1.0) (Goldberg, 1989), where a homogeneous random generator will carry producing random values; at all, for selected new element selected the value will be correlated with P_c, in order to post element into the crossing pool. Many nature of crossover happen, such as multipoint crossover, homogeneous crossover, single point crossover, and three parent crossovers. In this thesis we used single point crossover, where two parent's capacity are crossing giving to random choose point. One disadvantage is in single point crossover that the first number of parents posts same as in parents, where they are disconnected but they may enclose solution for the issue.

In comparison, one homogeneous random generated crossover points less used than multiple points' crossover, where they can separate the parent and move their values into new baby. This shares priority over single point crossover. Homogeneous crossover uses a single point chance for every non-identical gene, which make a higher chance for condition values to be changed. For example, for binary representation, if condition value is 1 the first element capacity are post to second, and vice versa if zero is found. Figure 5 shows the single crossover pseudo code (Goldberg, 1989).

```
Algorithm: Single Point Crossover
Input: two individuals randomly picked from mating pool
Output: new explored offspring's
Position= random $\{1, \cdots, N\}$
        For $i \leftarrow 1$ to position do
                Child 1[i] = parent 1[i]
                Child 2[i] = parent 2[i]
        end
        For $i \leftarrow$ position $+ 1$ to N do
                Child 1[i] = parent 2[i]
        Child 2[i] = parent 1[i]        end
```

Figure 2: Single crossover pseudo code (Goldberg, 1989)

2.4 Mutation

Mutation is frame work of GA. Mutation avoid algorithms from nature stuck with local minimum, because it scout the complete search space. Such as; if we want to maximize function $f(x) = x$ in constrained interval [0, 7]. Then the starting population might not be the top. By mutating condition we may shift some genes into value close to $(111)_{binary}$, by iteration. Chance of mutation is tried for every gene, which is dispute crossover; so, it infrequently occur because of its small value. There are several kind of mutation which straightly based on description. For example, if we use actual record or integer then mutation criteria will be separate with binary representation. If record are discreet and elements are represented in binary base, then mutation will be a bitwise by exchanging 0 to 1, and vice versa. Finally, chance of mutation, $P_m = \frac{1}{n}$ (Sharda & Voß, 2002), where n is chromosome length. Irregularly, P_m may be fixed, but the representative P_m (0.05-0.5) (Goldberg, 1989), in our structure we use the identical values.

2.5 Population Replacement

There are many options for population replacement, I am going to describe two of them:

1. Succeeding a GA primary process choose only the top elements with some previous methods, the whole parents and baby's are sharing the same chance to be chooses.
2. Choose only from the new generated baby's and kill the whole parents, especially substitution method, where baby's assume their parent.

2.6 Search Termination

There are many criteria which have been constructed for search termination. Because of the stochastic nature of GA it can run for infinity, but it needs to be stopped at any given time because evaluation of the solution is needed. We can classify stopping condition into three types, time dependent, iteration dependent and fitness dependent.

1. Maximum number of generations: We need to stop the algorithm, if we pass the maximum number of iteration. Occasionally we need to forecast the certain number of needed determined by iterations on the difficulty of the issues. e.g. our maximum function evaluation scheme (FES) is equal to 500000. The initial stopping condition will be a number of iterations is the relevant and generally used part.

2. Elapse time: sometime starting time to end time can be adoption as a secondary stopping scheme. Issues are heterogeneous in complexity; occasionally, we can forecast interval for stopping algorithm runs. Concurrently it must stop, if the maximum generation number is reached.

3. Minimal diversity: Compute difference between attribute and fitness inside is a critical running. Since as attributes are maintained, and the process will continue its value even after reconnection process. Algorithms need to be stopped. Occasionally, this standard process more priority over number of iterations.

4. Best Individuals: Whenever the minimum of fitness in the population collapse under the intersect value. This will lead the search process to faster accomplishment that assurance at least one good solution (Sivanandam & Deepa, 2008).

5. Worst Individuals: minimum fitness value for the worst individual perhaps less than intersect scale. In this case intersect value may not be achieved (Sivanandam & Deepa, 2008).

6. Sum of Fitness: the observant treated to have contentment intersected when the sum of fitness in the whole data is less than or equal to the intersect value on the population data. This assurance that logically all elements are in the same area (Sivanandam & Deepa, 2008)

CHAPTER 3
THE METHODS IN GA FOR MULTIMODAL FUNCTION OPTIMIZATION

3.1 Method of Iteration

Iteration is the easiest method which can be used with any optimization method to create multiple solutions. If the optimization method absorb stochastic conduct, then there is a chance that successive runs will come up with different solutions. But this is rather a slow proceed towards. However many times the algorithm is iterated, there is no guarantee that it will locate all maxima of interest.

Applying iteration to any single solution method, the very best we can hope for is that if there are p maxima, we will have to iterate p times. Using blind iteration, to remunerate for the duplicate solutions which are likely to arise, we can expect to have to increase the number of iterations by a factor, which we shall call the redundancy factor, R If all maxima have an equal likelihood of being found, it can be shown that R has a value given by:

$$\mathrm{R} = \sum_{m=i}^{p} 1/\mathrm{m} \tag{3}$$

This can be approximated (Jolley, 1961) for p>3, by:

$$R \approx r + \log p$$

where $r \approx 0.577$(Euler's constant).

This value is relatively small even for p =1000, R is only 7.5. If all maxima are not probably to be found, the value of R will be much higher.

3.2 Parallel Sub-population

Another method which can produce multiple solutions is a GA. Where the population is divided into multiple sub-populations which evolve in parallel. If there is no transmission between sub-populations, this is directly equivalent to iterating a single smaller population many times. Likewise there is no guarantee that different sub-populations will converge on different maxima. Most execution exploit some degree of communication between the sub-populations, to allow good genes to spread. However, the effect of this is to reduce the diversity of solutions, and the whole population will finally converge on a single solution (Grosso, 1985). Local mating schemas experience similar problem (Davidor, 1991).

3.3 Fitness Sharing

(Goldberg and Richardson, 1987) described the design of fitness sharing in a GA as a way of assist solid sub-populations, or species. The aim of sharing comes from an analogy with nature. In natural, there are many different ways in which animals may survive (feed, hunting, on the ground, in trees etc) and different species develop to fill each character. Each character is referred to as an ecological niche. For each niche the physical resources are limited and must be shared among the population of that niche. This provides a obstacle for all creatures to populate a single niche. Alternative creatures evolve to form sub-populations in different niches. The size of each sub-population will send back the availability of resources in the niche. The analogy in function optimization is that the location of each maximum represents a niche, and by acceptably sharing the fitness related with each niche. we can encourage the formation of stable sub-populations at each maximum. Sharing is carried out on the basis that the fitness "payout" within each niche is finite, and must be shared between all individuals in that niche. Therefore, the fitness awarded to an individual will be inversely proportional to the number of other individuals in the same niche. The total payout for a niche is set equal to the height of the maximum so larger maxima can support proportionally larger sub-populations. However, a fundamental problem is where are the niches. How do we decide if two individuals are in the same niche, and should therefore have their fitness values shared. To do this accurately, we need to know where each niche is, and how big it is. Clearly we cannot know where the niches are in advance, otherwise we would not have a problem to solve! (Goldberg and Richardson, 1987) approach this by assuming that if two individuals are close together, within a distance known as the niche radius, then their fitness values must be shared Closeness, in this instance, is measured by the distance in a single dimension decoded parameter space. A method for choosing the niche radius is presented in (Deb & Goldberg,

1989) and this is discussed later. Although their technique is successful, its disadvantages have been pointed out by Smith, (Forrest and Perelson, 1992). To compute accurately the fitness of an individual involves calculating its distance from every other member of the population. The total time complexity will depend on the time taken for the basic GA, plus the additional time taken to perform the fitness sharing calculations. This additional complexity is $O(N^2)$ where *N* is the population size. This is a disadvantage if *N* is large, and *N* must be large if we hope to find many optima. (Goldberg, 1989) recommends that for a multimodal GA expected to find p different solutions, we need a population p times larger than is necessary for a unimodal GA. This is because a certain number of individuals are necessary to accurately locate and explore each maximum. If a GA is to support stable sub-populations at many maxima, the total population size must be proportional to the number of such maxima. The above argument assumes that the population is spread equally among all maxima of interest. However, this may not be the case for two reasons Firstly, under the standard fitness sharing scheme, high fitness peaks will contain proportionally more individuals than low peaks. Secondly, there may be peaks in the fitness function which we are not interested in, but which capture a proportion of the population. This was a significant problem for (Goldberg, Deb and Horn, 1992). This uneven distribution requires that *N* is increased, to ensure that even the smallest peak of interest contains a sufficiently large sub-population. The factor by which N must be increased we call the peak ratio Ø. If p is the proportion of peaks which are of interest, and f_{min} is the fitness of the smallest peak of interest, then Ø is determined as

$$\emptyset = \frac{\text{mean fitness of all peaks}}{f_{min}} \quad (4)$$

So the population size required is N= *npØ* where n is the population size needed to find one solution. Oei, Goldberg and Chang(1991) suggest several improvements to the original fitness sharing algorithm. They say that the time complexity of the fitness sharing calculations may be reduced from ON^2 to *O(N p)* if we sample the population, instead of computing the distance to every other member. They also describe how, in conjunction with tournament selection, a niche size threshold parameter, may be used to limit the maximum number of individuals in each niche. This should prevent highly-fit maxima from gaining significantly more individuals than less-fit maxima. So the effective value of will be between 1 if the uninteresting peaks have low fitness relative to f_{min} and if the uninteresting peaks have fitnesses close to f_{min}. Assuming that these suggestions are viable, we can compute the time complexity as follows :

The time taken for the basic GA to find a single peak is proportional to the number of function evaluations performed before convergence this is approximately (*a N gc*) where α is the time for one function evaluation, performed before convergence; the additional time for the fitness sharing is approximately (*βANpgc),* where β is the time to compute the distance from one individual to another, and A is the number of individuals we must sample in each niche. Using A=5(Oei et al, 1991), and setting N=nØp, the time complexity, C*shair, is*

$$C{shair\ e}=O(n\O pgc(\alpha+5\beta p)) \tag{5}$$

3.4 The Sequential Niche Genetic Algorithm Technique (SNGA)

3.4.1 Basic principles

(Ackley, 1987) compared a number of function optimization techniques, including GAs and hill climbers. When an optimization technique found a sub-optimum peak, he iterated the technique until the (known) global maximum had been found. He examine that to solve difficult problems, (with local maxima), we must search till that we find one of the maxima, and then try again. But blind iteration does not learn lessons. When we iterate a conventional GA, everything learned in the previous run is forgotten. The sequential niche technique reported here is defined on the idea of bringing over knowledge learned in one run to each subsequent run.

Once one maximum has been found, there is no need to search for it again. So our approach is to eliminate that peak from the fitness function, using a method akin to sharing. As mentioned above, sharing is complicated to perform if the locations of the niches are unknown. But after one run of the GA, the location of one of the niches is known. On the second run of the GA, we assume that this niche is conceptually already filled, and few further rewards are available to any individuals which might stray into the area. Instead, individuals are forced to converge on an unoccupied niche, which is in turn also conceptually assumed filled in the third run. This process can continue until we decide (using some criterion, such as the number of maxima we expect in the function) that all maxima have been located.

This algorithm is similar to sharing in its approach. The sharing algorithm essentially works by dynamically modifying the fitness landscape according to the location, in each generation, of individuals in the population. This is done in such a way as to encourage the dispersion of individuals throughout the landscape, which, hopefully, leads to better exploration of the search space, and the identification of all maxima. Conversely, our sequential niche algorithm works by modifying the fitness landscape according to the location of solutions found in previous iterations (unlike the sharing algorithm, the fitness landscape remains static during each run). This is done in such a way as to discourage individuals from re-exploring areas where solutions have been found, thereby encouraging them to locate a maximum elsewhere.

In the sharing algorithm, once the population begins to converge on multiple peaks, cross breeding between different peaks is likely to be unproductive. (Deb, 1989) showed that a mating restriction scheme was able to improve the sharing algorithm for this reason. If mating between individuals on different peak is to be prohibited, performance would be improved by instead running a number of separate, smaller population GAs, one exploring each peak. It might be possible to achieve this with a parallel GA, where the sub-populations on each processor evolve separately after an initial period in which they evolve together, using some algorithm which ensures that no two sub-populations are converging on the same peak. However, we do not adopt this approach because it appears unnecessarily complicated.

3.4.2 Derating functions

The single-peak derating function, G, are possible; two were tested: power law, Eqn. (6), and exponential, Eqn. (7). Here, r is the niche radius (see below), d_{xs} is the distance between x and s, as determined by the distance metric. In Eqn. (6), α is the power factor, which determines how concave (α > 1) or convex (α < 1) the derating curve is, with α = 1 giving a linear function. When dxs = 0, Gp(x; s) = 0. In Eqn. (7), m is the derating minimum value, which defines G when d_{xs} = 0 (such that G(x; s) = m). m must be greater than 0, since log(0) = -∞. m also determines the concavity of the derating curve, with smaller values of m producing more concavity. For both forms, when $d_{xs} \geq$ r, we set G(x; s) = 1.

$$G_p(x,s) = \begin{cases} (d_{xs}/r)^{\alpha} \text{ if } d_{xs} < r \\ 1 \; otherwise \end{cases} \tag{6}$$

$$G_e(x,s) = \begin{cases} exp(logm * (r - d_{xs})/r) & \text{if } d_{xs} < r \\ 1 & otherwise \end{cases} \tag{7}$$

The sharing function described by Goldberg and Richardson (1987), Deb (1989) and Deb and Goldberg

(1989) performs the task of reducing the fitness of an individual depending on its distance from each other individual in the population. In a similar way, our fitness derating function performs the task of reducing the fitness of an individual depending on its distance from each best individual found in previous runs.

To determine the value of niche radius, *r*, we use the same method as Deb (1989), which is as follows. If we imagine that each of the p maxima in the function are surrounded by a hyper sphere of radius r, and that these hyper spheres do not overlap, and that they completely fill the k-dimensional problem space, then if the parameter range for each dimension is normalized to be 0 to 1, *r* is given by:

$$r = \sqrt{k}/2 * \sqrt[k]{p} \tag{8}$$

Deb's technique is simple, but requires that we know (or estimate) the number of maxima in the function, and assumes that all maxima are equally spread throughout the search space. Clearly, there will be functions where these restrictions are not satisfied. For our present experimental investigations, we adopt the simple approach to niche radius given above.

3.4.3 Explanation of algorithm

Any problem to be solved by a GA needs a fitness function. For the sequential niche algorithm we also need a distance metric. This is simply a function which, given two chromosomes, returns a value related to how "close" they are to each other. The most trivial measure of distance is the Hamming distance, although as (Deb and Goldberg, 1989) showed, sharing works better if we use a distance measure based on the decoded parameter space. One reason for this is that for binary-coded chromosomes at least large differences between chromosomes may correspond to small different in parameter space at a Hmming cliff, for example. By using the parameter space to determine the distance between two chromosomes, we avoid any topological distortion introduced by the coding scheme. Typically, a chromosome (the genotype) represents a number of parameters (the phenotype), and they are mapped to a fitness value. If there are k parameters, then each individual can be thought of as occupying a point in k-dimensional space. The distance between two individuals can be taken as the Euclidian distance between them. (As suggested by (Deb, 1989), if the parameter is different dimensions are widely different, it may be desirable to normalize each parameter to cover the range 0 to 1 before the Euclidian distance is determined.) There are other ways to define a distance metric, but we shall use the Euclidian distance. Our algorithm does not rely on any particular distance metric.

Having defined a fitness function and distance metric, the algorithm works as follows (terms in italics are explained below):

1. Initialize: equate the modified fitness function with the raw fitness function.

2. Run the GA (or other search technique), using the modified fitness function, keeping a record of the best individual found in the run.

3. Update the modified fitness function to give a depression in the region near the best individual, producing a new modified fitness function.

4. If the raw fitness of the best individual exceeds the solution threshold, display this as a solution.

5. If not all solutions have been found, return to step (2).

A single application of this overall algorithm we refer to as a sequence, since it consists of a sequence of several GA runs. Knowledge of niche locations (maxima) is propagated to subsequent runs in the same sequence.

A solution is a set of k parameters which define the position of a maximum of interest. The solution threshold represents the lower fitness limit for maxima of interest. (It is assumed that there are a known number, p, of \interesting" maxima, all with fitness greater than the solution threshold.) Its value is set individually for each problem. If no information is available about the likely fitness values of the maxima of interest, the solution threshold may be set to zero. In this case, the algorithm is terminated after the first p peaks have been found

The modified fitness function, M(x), for an individual, x, is computed from the raw fitness function, F (x), multiplied by a number of single-peak de-rating functions . Initially we set M0(x) = F (x). At the end of each run, the best individual, s_n, found in that run is used to determine a single-peak de-rating function, G(x; s_n). The modified fitness function is then updated according to:

$$M_n+1(x) = M_n(x) * G(x; s_n) \quad (9)$$

3.4.4 Termination conditions

Deciding when to halt a GA run and start the next iteration is not a trivial task. Towards the end of a run of any traditional GA, the search efficiency falls off since the population becomes more uniform, causing exploration to rely increasingly on mutation (Goldberg, 1989b). For our algorithm, we need to decide at what point in the run we are unlikely to improve on the best individual found so far, and then terminate the run. The technique we adopt is to record the population average fitness over a halting window of h generations, and terminate the run if the fitness of any generation is not greater than that of the population h generations earlier. Values of h between 5 and 20 gave good results with the test functions we tried. More complex functions with longer convergence times might need larger values. A run is also terminated if an individual is found with a known maximum fitness target, or a maximum number of generations has been reached.

CHAPTER 4

METHOD OF PARTICLE SWARM OPTIMIZATION

Particle swarm optimization (PSO) is basically a stochastic optimization technique developed by (Eberhart and Kennedy, 1995), provoked by social behavior of bird flocking or fish schooling.

Genetic Algorithms(GA) and PSO has many similarities with evolutionary computation techniques. The system is determined with a population of random solutions and examine for optima by updating generations. However, unlike PSO,GA has no evolution operators such as crossover and mutation. In PSO, the potential solutions, called particles, fly through the problem space by tracing the current optimum particles.

Each particle follows its coordinates in the problem space which are associated with the best solution (fitness) it has gain so far. This value is called *pbest*. Another "best" value that is chased after by the particle swarm optimizer is the best value, obtained so far by any particle in the neighbors of the particle. This location is called *lbest*. when a particle considers all the population as its topological neighbors, the best value is a global best and is called *gbest*.

The particle swarm optimization concept contains of, at each time step, accelerating each particle toward its p*best* and *lbest* locations (local version of PSO). Acceleration is measured by a random term, with separate random numbers being generated for acceleration toward *pbest* and *lbest* locations.

In past several years, PSO has been applied as a powerful tool in many researches and application areas. It is found that PSO gets better results in a faster, cheaper way compared with other methods.

Another reason that PSO is successful is that there are few parameters to adjust. One version, with slight changes, works well in a wide range of applications. Particle swarm optimization has been applied for approaches that can be used across a wide range of applications, as well as for specific focused applications.

The focus of this demonstration is on the second topic. Actually, there are a lots of computational techniques provoked by biological systems, such as artificial neural network is a simplified model of human brain; genetic algorithm is inspired by the human evolution.

Now we discuss another type of biological system - social system, more specifically, the collective behaviors of simple individuals, interaction between their environment and each other. Someone of it is known as swarm intelligence. All of the simulations take advantage of local processes, such as those modeled by cellular automata, and might underlie the unpredictable group dynamics of social behavior.

Some famous examples are floys and boids. Both of the simulations were created to elucidate the movement of organisms in a bird flock or fish school. These simulations are frequently used in computer animation or computer aided design.

There are two famous swarm inspired methods in computational intelligence areas: particle swarm optimization (PSO) and Ant colony optimization (ACO). ACO was inspired by the behaviors of ants and has been successfully applied in discrete optimization problems. The concept of particle swarm is originated as a simulation of simplified social system. The original purpose was to graphically simulate the choreography of bird of a bird block or fish school. However, it was discovered that particle swarm model can be used as an optimizer.

4.1. Algorithm

As discussed before, PSO simulates the behaviors of bird flocking. Let's consider the following scenario: a group of birds are randomly searching food in an area. There is only one piece of food in the searching area. All the birds do not know where can they find the food. But they have idea how far the food is in each iteration. So what's the best plan to find the food? The best one is to follow the bird which is nearest to the food.

PSO educate from the scenario and used it to solve the optimization problems. In PSO, each single solution is a "bird" in the searching area. We consider it "particle". All of particles have fitness values which are measured by the fitness function to be optimized, and have velocities which direct the flying of the particles. The particles fly through the problem space by tracing the current optimum particles.

PSO is determined with a group of random particles (solutions) and then searches for optima by updating generations. In each and every iteration, each particle is updated by following two "best" values. The first one is the best solution (fitness) it has gained so far. (The fitness value is also stored.) This value is called *pbest*. Another "best" value that is followed by the particle swarm optimizer is the best value, obtained so far by any particle in the population. This best value is a global best and is called *gbest*. When a particle participates of the population as its topological neighbors, the best value is a local best and is called I-best.

After finding the two best values, the particle updates its velocity and positions with following equation (a) and (b).

$$v[] = v[] + c1 * rand() * (pbest[] - present[]) + c2 * rand() * ([] - present[]) \quad (a)$$

$$present[] = persent[] + v[] \quad (b)$$

v[] is the particle velocity*, persent[]* is the current particle (solution). *pbest[]* and *gbest[]* are defined as discused before. *rand ()* is a random number between (0,1*). c1, c2* are learning factors. usually *c1 = c2 = 2.*

The pseudo code of the procedure is as follows

```
For each particle
    Initialize particle
END
Do
```

```
  For each particle
     Calculate fitness value
     If the fitness value is better than the best fitness value (pBest) in history
        set current value as the new pBest
  End
  Choose the particle with the best fitness value of all the particles as the gBest
  For each particle
     Calculate particle velocity according equation (a)
     Update particle position according equation (b)
  End
While maximum iterations or minimum error criteria is not achieved
```

Particles' velocities on each dimension are hoid to a maximum velocity Vmax. If the sum of accelerations would cause the velocity on that dimension to pass Vmax, which is a parameter specified by the user. Then the velocity on that dimension is bound to Vmax.

Comparisons between Genetic Algorithm and PSO

Most of evolutionary techniques have the following steps:

1. Random generation of an initial population

2. Reckoning of a fitness value for each subject. It will directly depend on the distance to the optimum.

3. Reproduction of the population based on fitness values.

4. If requirements are met, then stop. Otherwise go back to 2.

From the procedure, we can result that PSO shares many common points with GA. Both algorithms start with a group of a randomly generated population, both have fitness values to appraise the population. By random techniques both update the population and search for the optimium. Both systems do not assure success.

However, PSO does not have genetic operators (crossover and mutation). Particles modify themselves with the internal velocity. They also have memory, which is significant to the algorithm.

The information sharing mechanism in PSO is significantly different as compared to genetic algorithms (GAs). In GAs, chromosomes share information among themselves. So the whole population behaves like one group towards an optimal area. In PSO, only *gBest* (or *lBest*) gives out the information to others. It is a single-way information sharing mechanism. The evolution only cares for the best solution. Compared with GA, all the particles incline to converge to the best solution quickly even in the local version in most cases.

4.2 PSO Parameter Control

From the above case, we can learn that there are two key steps when applying PSO to optimization problems: the representation of the solution and the fitness function. One of the advantages of PSO is that PSO take real numbers as particles. It is not like GA, which needs to change to binary encoding, or special genetic operators have to be used. For example, we try to find the solution for f(x) = x1^2 + x2^2+x3^2, the particle can be set as (x1, x2, x3), and fitness function is f(x). Then we can use the standard procedure to find the optimum. The searching is a repeat process, and the stop criteria are that the maximum iteration number is reached or the minimum error condition is satisfied.

There are not many parameter need to be tuned in PSO. Here is a list of the parameters and their typical values.

The number of particles: the typical range is 20 - 40. Actually for most of the problems 10 particles is large enough to get good results. For some difficult or special problems, one can try 100 or 200 particles as well.

Dimension of particles: It is determined by the problem to be optimized,

Range of particles: It is also determined by the problem to be optimized, you can specify different ranges for different dimension of particles.

Vmax: it determines the maximum change one particle can take during one iteration. Usually we set the range of the particle as the Vmax for example, the particle (x1, x2, x3)

X1 belongs [-10, 10], then Vmax = 20

Learning factors: c1 and c2 usually equal to 2. However, other settings were also used in different papers. But usually c1 equals to c2 and ranges from [0, 4]

The stop condition: the maximum number of iterations the PSO execute and the minimum error requirement. We can set the minimum error requirement is one mis-classified pattern. the maximum number of iterations is set to 2000. this stop condition depends on the problem to be optimized.

Global version vs. local version: we introduced two versions of PSO. global and local version. global version is faster but might converge to local optimum for some problems. local version is a little bit slower but not easy to be trapped into local optimum. One can use global version to get quick result and use local version to refine the search.

Another factor is inertia weight, which is introduced by (Shi and Eberhart). If you are interested in it, please refer to their paper in 1998. (Title: A modified particle swarm optimizer).

4.3 The ASNPSO Algorithm

4.3.1 The basic principle

The adaptive sequential niche technique is essentially an add-on technique, which can be used together with any stochastic search algorithm. Hereby, we choose the PSO algorithm to implement it. ASNPSO consists of several sub-swarms. Each sub-swarm can detect one optimal solution. Because of the algorithm using multi-sub-swarms to detect different solutions sequentially, in order to avoid all sub-swarms converging to one or several certain optimal solutions, the algorithm must be able to modify the fitness function of the particles of the sub-swarm run later. Hereby we introduce the penalty function by means of a constrained optimization problem. Assume that a simple death penalty function is used in our algorithm. Then it can make the particle of the moving sub-swarm locate in the same hill of the optimal solutions found before losing influence. In this paper, the modified fitness function of the sub-swarm currently running is to satisfy the following equation:

$$(eval(x_n^i) = \begin{cases} f(x_n^i) & if\ hill_{vally}(x_n^i, xk) = 1, \\ f(x_n^i) - p(x_n^i) & otherwise. \end{cases} \qquad (10)$$

where x_n^i is the *ith* particle of the nth sub-swarm; n is the number of sub-swarm run presently; f(xi n) is a raw fitness function, *xk* is the top point found by the sub-swarm begun before, *k* can be selected from *1* to *n1; p(xi n)* can be a very large value for real world problem and makes the corrected fitness very small than all local optima.

4.3.2 Termination conditions for sub-swarm

The termination condition for each sub-swarm is an important task for sequential niche technique. In practical applications, if the termination condition is satisfied, we must halt the running PSO and switch to run a new sub-swarm. In this paper (Beasley's, 1993) halt window technique is adopted as a termination condition. The technique is to record the population average fitness over a halting window of h generations, and terminate the run if the fitness of any generation is not greater than that of the population h generations earlier. In our experiments, assume that the halt window is set to 20.

```
Repeat
        Run a new sub-swarm
        N=N+1;
        If hill_vally ($x^i, x_k$)==0
            Change the fitness of xi
        Else
            Keep the original fitness
        End
        Next
train the sub-swarm until the halt window of sub-swarm is larger than a given value
Until N is greater than a given value or reach as a maximum iteration number
```

Figure 3: Pseudo code of ASNPSO algorithm.

This algorithm checking escorted by a hill valley function to match the region of a iterating particle and the top intersect region of the swarm floated before belong to one hill or not. The number of sub-swarms used for the algorithm based on the number of optimal results what we need.

4.3.3 Hill valley algorithm

The determination of the niche radius is generally a hard work existing in most niche methods. However, if we have a method that can determine whether or not two points of search space belong to a peak of the multimodal function, then the niche radius is not needed in this situation. Ursem's hill valley function is the first method, where *ip* and *iq* are any two points in search space. In fact, it can be easily extended to the case including arbitrary dimensions.

Generally speaking, the function returns *0* if the fitness of all the interior points is greater than the minimal fitness of *ip* and *iq*, otherwise it returns *1*. With this function, the algorithm is able to determine whether *ip* and *iq* belong to one hill or not. The sample array is generally used to calculate the interior points where the hill valley function computes the fitness of these samples. The points *i* interior can be calculated as

$$i \text{ interior} = ip + (iq + ip) \ \text{samples}\, j,$$

where *j* is *jth* entry in the array. The upper boundary of *j* is the length of the samples, which is very important for the hill valley function. We refer to the length as sample rate (SR). PSO is a population-based stochastic search algorithm.

$$V\,id = W\,Vid + C1\,rand1(*)(Pid - Xid)$$

$$+\, C2\,rand2(*)(Pgd - Xid),$$

$$Xid = Xid + Vid\,,$$

where i = 1,2,y, N, and W is called as inertia weight. *C1* and *C2* are positive constants, referred to as cognitive and social parameters, and *rand1* (*) and *rand2* (*) are random numbers, respectively, uniformly distributed in [0..1]. The *ith* particle of the swarm can be represented by the D dimensional vector *Xid*, and the best particle in the swarm denoted by the index *g*, the best previous position of the *ith* particle is recorded and represented as *Pid* and the velocity of the *ith* particle is as *Vid*.

```
Hill_valley ( $i_p, i_q$ , $samples$ )
minfit=min (fitness($i_p$), fitness($i_q$))
        for j=1 to samples, length
Calcuate point $i_{interior}$ on the line between the point $i_p$ and $i_q$
        if (minfit > finess($i_{interior}$))
                return 1
        end if
end for
return 0
```

Figure 4: Hill valley function Pseudo code

CHAPTER 5

EXPERIMENTAL RESULTS

In this section, there are eight functions to test in sequential niche genetic algorithm (SNGA) and adaptive sequential niche with particle swam optimization (ASNPSO) which are graphically represented.

5.1 Equal Maxima (F1):

This function has 5 equally area maxima equal height according NSGA:

$$F1(x) = sin^{6}(5\pi x)$$

The results of our algorithm total maximum values are (0.500199, 0.69949, 0.299329, 0.899088, and 0.098102).

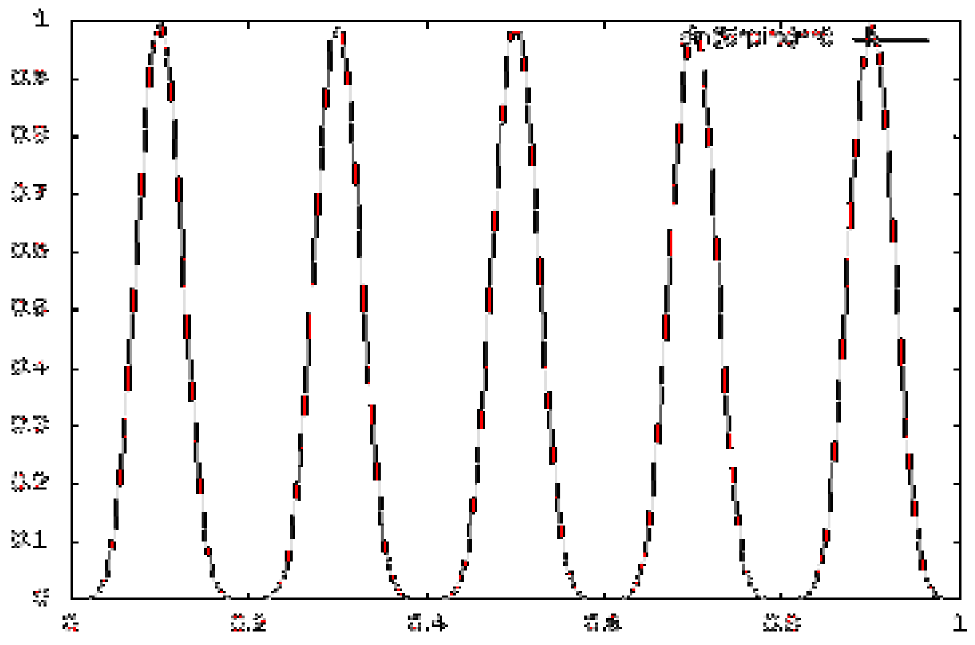

Figure 5: F1 equal maxima.

5.2 Decreasing Maxima (F2):

The second function is same as F1, However the maxima decrease in height exponentially:

$$F2(x) = exp\left(-2\log(2) * (\frac{x-0.1}{0.8}\right)*sin^{6}(5\pi x)$$

The maxima are at the same x values as for F1, but the peak height vary from 1.0 to0.25, (0.098102, 0.299329, 0.498301, 0.6978, and 0.897504)

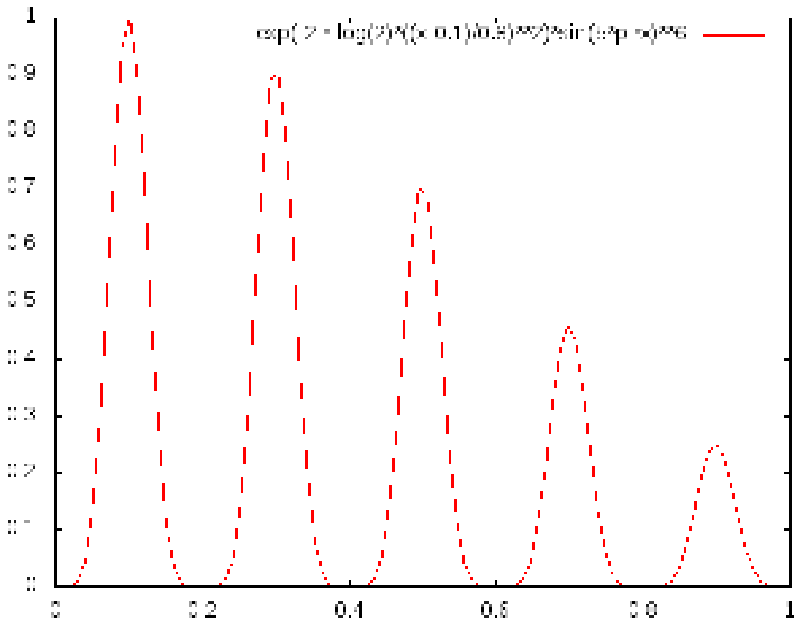

Figure 6: F2 Decrease maxima.

5.3 Uneven Maxima (F3):

The third function is same as F1, However the maxima are unevenly spaced:

$$F3(X) = \exp\left(-2\log(2) * \left(x - \frac{0.1}{0.8}\right)\right) * sin^{6}(5\pi\ (x\text{-}0.05))$$

The maxima are at x values of approximately (0.933302, 0.680825, 0.245847, 0.449539, 0.0775531) all with a height of 1.0.

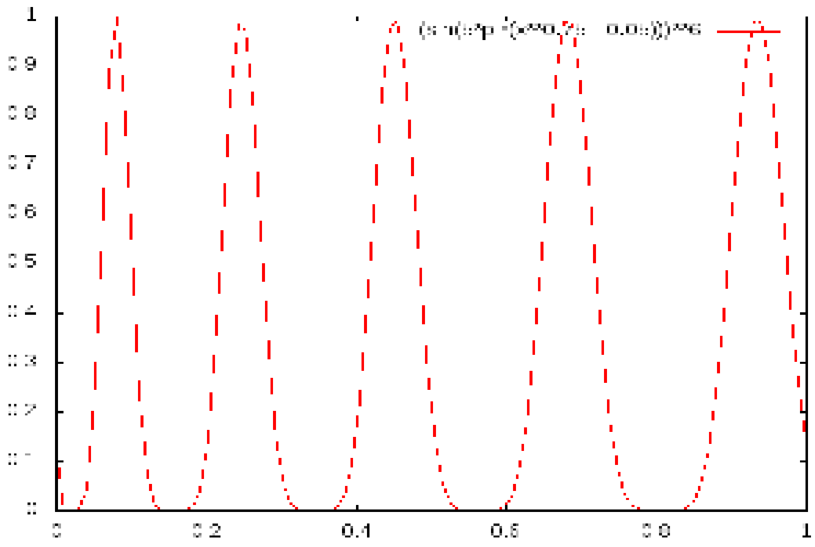

Figure 7: F3 Uneven maxima.

5.4 Uneven Decreasing Maxima (F4):

The fourth function is same as F3, However the maxima decrease in height exponentially.

$$F4(x) = exp\left(-2\log(2) * (\frac{x-0.8}{0.854}\right) *sin^6(5\pi\ (x^{3/4}\text{-}0.05))$$

The maxima are at the same x value as F3, (0.0775531, 0.245847, 0.449539, 0.679078, 0.929407), with the same height as F2

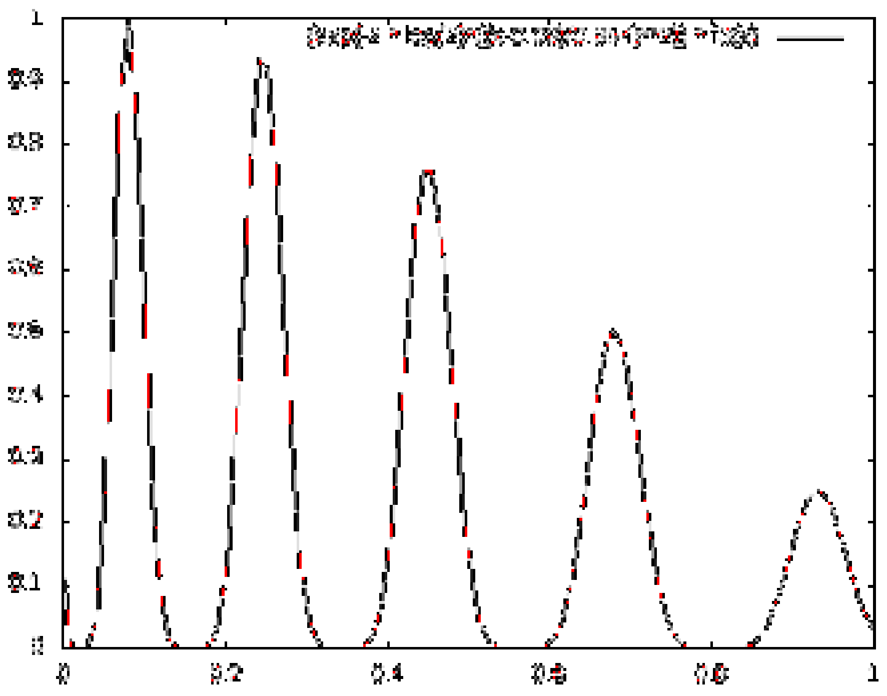
Figure 8: F4 Uneven Decreasing maxima.

5.5 Himmelblau's Function (F5):

This is a function of two variable, (x,y), that Deb's modified to be a maximization issue given by:

$$F5(x\ ,y)=200-(x^2+y-11)^2-x+y^2-7)^2$$

This has four maxima (2.96486, 1.97121), (-2.75633, 3.09146), (3.51842, -1.71378), (-3.63167, -3.27702)

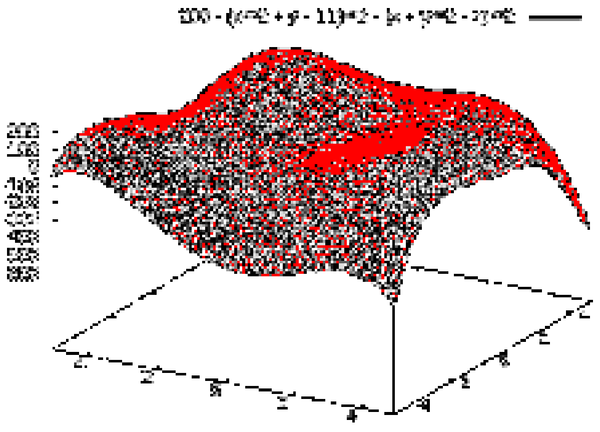

Figure 9: F5 modified Himmelblau's function.

5.6 Shubert Function (F6):

$$f1(X)=\sum_{i=0}^{5} \mathrm{i}\cos(i+1)x+1$$

This function has total 19 maxima, 3 global maxima and 16 local maxima, to get all 19 results of function in our experiment the number of sub-swarms run sequentially is set to 19.

F6 Coordinates values :(-4.58947, 6.2832, -6.28347, -5.10824, 1.17474, 5.10864, -7.45792, 7.45791, -1.17473, 4.1208, -2.16239, -4.12083, 8.44555, 2.16217, -8.44549, 9.42481, -3.1414, -9.42479, and 3.14151).

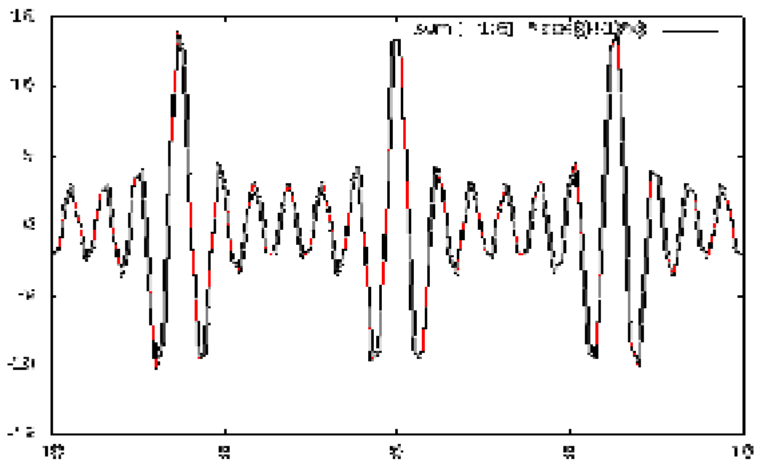

Figure 10: F6 Shubert one dimensional function.

5.7 Scekel's Foxhole Function (F7):

Its two dimensional function with 25 peaks of different heights. Let a(i)=16[(i mod5)-2] and b(i)=16([i/5]-2), the function can be defined as

$$f2(x,y)=500-\frac{1}{0.002+\sum_{i=0}^{24} 1/(1+i+(x-a(i))+(y-b(i))^6)}$$

Where -65.536≤x,y≤65.536.

F7 Coordinates values : : (-32.0038, -31.9927), (-16.003, -32.0083), (-31.9602, -15.9648),(0.0231036, -31.9693), (31.9827, -31.9348),(-16.0087, -15.9782),(16.1685, -31.9159), (-0.0909802, -15.9271),(15.9787, -15.9971),(-31.9427, 0.000643744),(-16.0182, 0.0661799),(-31.9291, 15.9688),(-0.278472, -0.10173), -(-15.7687, 15.9084), (31.9419, -16.0447),(15.9651, -0.0250031),(31.9208, -0.0149629),(0.00604302, 16.052), (16.0172, 15.9236), (-31.9564, 31.9717),(31.9201, 16.018),(-15.9495, 31.9225),(-0.0203267, 31.9487), (15.9776, 31.9405),(31.971, 31.9202).

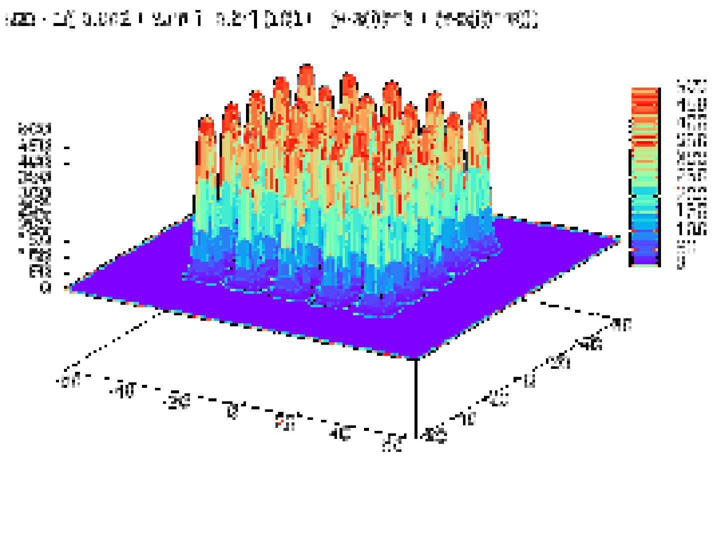

Figure 11: F7 Scekel's Foxhole function where 25 sub=swarm were run sequentially.

5.8 Shubert Function (two dimensional) (F8):

The function is defined as

$$f3\ ((x1,\ x2) = \prod_{i=1}^{2} \sum_{j=1}^{5} j \cos[(j+1)xi + j]$$

where -10≤xi≤10 for i=1,2. it has 760 local minima, 18 of which are global minima with function value of -186.73.In this experiments we converted the minima to maxima through multiplying by a negative sign the raw fitness function.

F8 Coordinates values: (-0.801366, 4.8633), (-0.804395, -1.423),(5.47962, 5.50401),(-7.69397, -0.799232), (-7.70792, -0.791834),(-1.4232, -7.08),(-1.42678, 5.48026),(-1.43239, -7.08545),(4.86021, -1.42291),(-7.08281, -7.70829),(-7.08522,(-7.71024, -0.799115),(5.48296, 4.85725),(-0.193166, -0.803246),(4.85887, 5.48242),(-7.08209, -7.70786),(-0.79735, 4.85732),(6.08714, -0.799545),(-7.7067, 5.48759),(5.48399, -0.195406),(4.85672, -7.08356),(-0.1943, -7.08131),(-0.196127, 5.48154),(-6.4779, 5.48367), (-6.47834, -7.08442),(-6.48081, -0.799172),(-0.795585, 6.08839),(6.08789, -7.08364),(-7.08227, -0.194554),(-0.801204,-0.1944).

Graphical result of above ASPANO three functions:

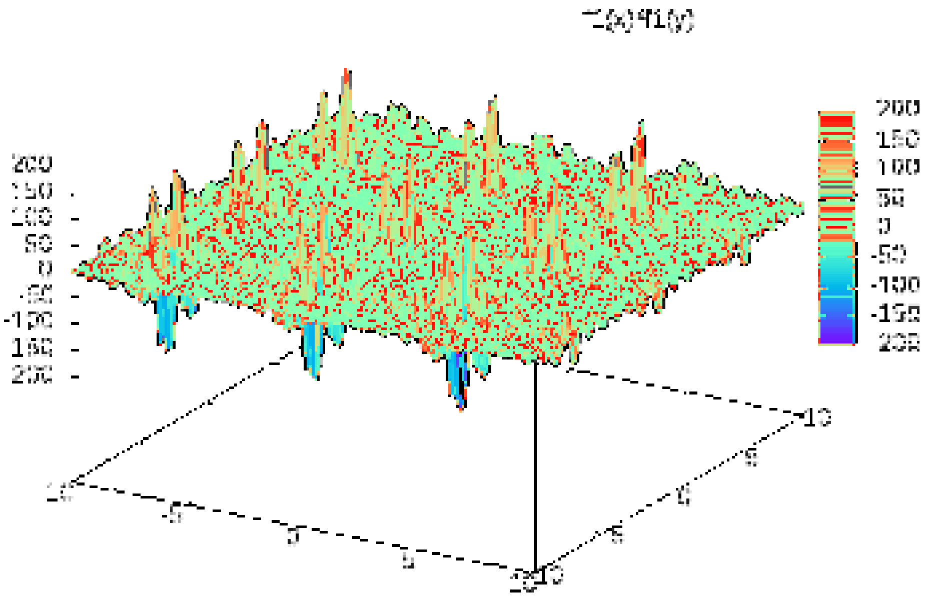

Figure 12: F8 Shubert function where 23 Sub-swarms were run sequentially.

CHAPTER 6

ANALYSIS AND COMPARISON BETWEEN ASNPSO AND SNGA

Both ASNPSO and SNGA are used for multi-modal function optimization. Presently, most niche methods need some more parameter, where the major important parameter is niche radius, which is used in SNGA. Niche radius, improper radius will make a niche algorithm showing worse. The resolution of the niche radius depends less or more on some knowledge from a particular issue. So one problem in Beasley (SNGA) technique is niche radius. In ASNPSO technique most extra parameter included niche radius are not needed. PSO implement in ASNPSO and GA implement in SNGA. In GA each of the three main classes of operation (selection, crossover, and mutation) can be executed in a number of ways. PSO does not label its operations in the same way as GA, but analogies do. Effects of GA (crossover) which normally vary remarkably during in iterate. In the beginning of GA the population member are usually randomly generating. So the crossover can have significant effects moving a chromosome relatively large distance in problem space. Crossover operation is not exist in PSO, in PSO the idea of crossover is shown in such a way that each particle is stochastically accelerated toward its own previous top position, as well as around the global top position.

For further comparison I tried to compare the functions results. The program was written by using C++ programming language.

6.1. Results of Functions:

I did 30 runs of both algorithm SNGA and ASNPSO, and collected the result of 8 functions which are 5 already used by (Beasley, 1993) and 3 of them used by (Zhang et al, 2006). We took the average of coordinates and f(x) values and compared with actual values of functions. (STD) mean is simple standard deviation calculated for both algorithms. The standard deviation of an entire population is known as **σ** (sigma) and is calculated using:

$$\sigma = \sqrt{\frac{\sum (x-\mu)^2}{N}} \qquad (11)$$

Where **x** represents each value in the population, **μ** is the mean value of the population and **N** is the number of values in the population. S-rate is success rate for each of coordinates, to calculate the s-rate divide the goal by the actual value.

6.1.1 F1 equal maxima:

If you see the result values in Table 1 and Table 2 which is for equal maxima function, the result shown that both of the algorithms is quite good in success rate, the SNGA average success rate is 99.92 % and ASNPSO is 100% achieved. The difference in the results, 100 - 99.94 = 0.06 which is not so big. Both of them perform well, but this is the result of only one function, I want to check for further results of functions in following tables.

Table 1: Result values of F1 obtained by SNGA algorithm.

Optimal		**Coordinates**		**F(x) Values**		
F(x) values	**Coordinates**	**Average**	**STD**	**Average**	**STD**	**%S-rate**
1	0.1	0.098102	4.235E-17	0.9973366	0	99.73%
1	0.3	0.299329	1.129E-16	0.9996668	3.388E-16	99.97%
1	0.5	0.500199	2.258E-16	0.9999707	4.517E-16	100.00%
1	0.7	0.69949	5.646E-16	0.9998075	0	99.98%
1	0.9	0.899088	2.258E-16	0.9993845	5.646E-16	99.94%
Average of success rate						99.92%

Table 2: Result values of F1 obtained by ASNPSO algorithm.

Optimal		**Coordinates**		**F(x) Values**		
F(x) values	**Coordinates**	**Average**	**STD**	**Average**	**STD**	**%S-rate**
1	0.1	0.099996327	9.161E-06	0.99999993	1.3053E-07	100.00%
1	0.3	0.2999983	2.374E-05	0.99999959	1.1103E-06	100.00%
1	0.5	0.500001433	9.246E-06	0.99999994	1.0921E-07	100.00%
1	0.7	0.7000034	1.903E-05	0.99999973	7.3999E-07	100.00%
1	0.9	0.879999033	0.1095477	0.99999993	1.6973E-07	100.00%
Average of success rate						100.00%

6.1.2 F2 Decreasing maxima:

In Table 3 and table 4 the results of decreasing maxima function, both algorithm achieved the 100% success rate, 0% difference in success rate so both algorithm perform good for this function.

Table 3: Result values of F2 obtained by SNGA algorithm.

Optimal		**Coordinates**		**F(x) Values**		
F(x) values	**Coordinates**	**Average**	**STD**	**Average**	**STD**	**%S-rate**
1	0.1	0.098102	4.235E-17	0.9973288	0	99.73%
0.917	0.3	0.299329	1.129E-16	0.9172307	0	100.00%
0.70711	0.5	0.498301	2.258E-16	0.7076733	3.388E-16	100.00%
0.4585	0.7	0.6978	5.646E-16	0.4594772	1.694E-16	100.00%
0.25	0.9	0.897504	4.517E-16	0.2510081	1.129E-16	100.00%
Average of success rate						99.946%

Table 4: Result values of F2 obtained by ASNPSO algorithm.

Optimal		**Coordinates**		**F(x) Values**		
F(x) values	**Coordinates**	**Average**	**STD**	**Average**	**STD**	**%S-rate**
1	0.1	0.099999807	6.899E-06	0.99999997	6.0405E-08	100.00%
0.917	0.3	0.299415033	1.026E-05	0.91723582	1.7481E-07	100.00%
0.70711	0.5	0.4988368	1.047E-05	0.70782207	1.2301E-07	100.00%
0.4585	0.7	0.6982493	8.078E-06	0.45954625	4.8687E-08	100.00%
0.25	0.9	0.897669233	9.054E-06	0.25101301	3.2499E-08	100.00%
Average of success rate						100.00%

6.1.3 F3 Uneven maxima:

Table 5 and Table 6 are the results of Uneven maxima function, both algorithm found all five optimal, SNGA found all optima with 99% success rate and ASNPSO found the result with 100% rate, the overall results of all optima for SNGA is 99.85% and ASNPSO with

100%, the difference between them 99.85-100=0.15% which is not so high. So both the algorithm working good for uneven maxima function.

Table 5: Result values of F3 obtained by SNGA algorithm.

Optimal		**Coordinates**		**F(x) Values**		
F(x) values	**Coordinates**	**Average**	**STD**	**Average**	**STD**	**%S-rate**
0.99987	0.08	0.0775531	0	0.9931803	4.517E-16	99.33%
0.99964	0.246	0.245847	1.129E-16	0.9994517	4.517E-16	99.98%
0.99976	0.45	0.449539	2.258E-16	0.999266	4.517E-16	99.95%
0.99991	0.681	0.680825	0	0.9998213	4.517E-16	99.99%
1	0.934	0.933302	4.517E-16	0.9998484	0	99.99%
Average of success rate						99.85%

Table 6: Result values of F3 obtained by ASNPSO algorithm.

Optimal		**Coordinates**		**F(x) Values**		
F(x) values	**Coordinates**	**Average**	**STD**	**Average**	**STD**	**%S-rate**
0.99987	0.08	0.07969848	0.0330261	0.99999979	3.9467E-07	100.00%
0.99964	0.246	0.2466502	1.839E-05	0.9999997	1.0486E-06	100.00%
0.99976	0.45	0.4506288	1.108E-05	0.99999992	1.2245E-07	100.00%
0.99991	0.681	0.6814189	2.021E-05	0.9999998	7.1026E-07	100.00%
1	0.934	0.933894	1.879E-05	0.99999985	4.8475E-07	100.00%
Average of success rate						100.00%

6.1.4 F4 Uneven decreasing maxima:

The Table 7 and Table 8 represent the result of Uneven decreasing maxima function, both algorithms achieved with 100% success rate for all five optimal only for two coordinates the SNGA success rate 99% which is not quite big difference but if we see the overall results for above all four functions ASNPSO received 100 % success rate, which means the

ASNPSO is batter for all above four functions. With these four functions we can't conclude that which one is batter, for further result I need to check some more functions. I tested four more functions to achieve the conclusions, for further test we check the given tables.

Table 7: Result values of F4 obtained by SNGA algorithm.

Optimal		**Coordinates**		**F(x) Values**		
F(x) values	**Coordinates**	**Average**	**STD**	**Average**	**STD**	**%S-rate**
0.99987	0.08	0.0775531	0	0.993169	5.646E-16	99.33%
0.94863	0.246	0.245847	1.129E-16	0.9485405	4.517E-16	99.99%
0.77069	0.45	0.449539	2.258E-16	0.7708143	1.129E-16	100.00%
0.50325	0.681	0.679078	2.258E-16	0.5041095	4.517E-16	100.00%
0.25	0.934	0.929407	4.517E-16	0.2515492	1.129E-16	100.00%
Average of success rate						99.864%

Table 8: Result values of F4 obtained by ASNPSO algorithm.

Optimal		**Coordinates**		**F(x) Values**		
F(x) values	**Coordinates**	**Average**	**STD**	**Average**	**STD**	**%S-rate**
0.99987	0.08	0.079700157	1.163E-05	0.99999964	4.5527E-07	100.00%
0.94863	0.246	0.246276267	1.419E-05	0.94868915	2.7058E-07	100.00%
0.77069	0.45	0.4494941	7.73E-06	0.77081521	4.0855E-08	100.00%
0.50325	0.681	0.679165767	1.024E-05	0.50411148	4.2843E-08	100.00%
0.25	0.934	0.930150367	9.981E-06	0.25161007	1.2405E-08	100.00%
Average of success rate						100.00%

6.1.5 F5 Himmelblau's function:

The Table 9 and 10 are the results of Himmelblau's function which is two dimensional function, this has four maxima with equal height (200), and parameter range is set for both algorithm $-6 \leq x, y \leq 6$. Both algorithm achieved 99% success rate for all maxima, resulted values is good enough for both algorithms.

Table 9: Result values of F5 obtained by SNGA algorithm.

Optimal			Coordinates		F(x) Values		
F(x, y) values	**Coordinates**		**Average**		**Average**	**%S-rate**	**STD**
	X	**Y**	**X**	**Y**	**F(x, y)**		
199.997	3.58	-1.86	3.51842	-1.71378	199.59159	99.80%	5.78152E-14
200	3	2	2.96486	1.97121	199.92083	99.96%	5.78152E-14
199.995	-2.815	3.125	-2.75633	3.09146	199.86348	99.93%	1.44538E-13
199.999	-3.78	-3.28	-3.63167	-3.27702	198.80478	99.40%	0
199.997	3.58	-1.86	3.51842	-1.71378	199.59159	99.80%	5.78152E-14
Average of success rate						99.77%	

Table 10: Result values of F5 obtained by ASNPSO algorithm.

Optimal			Coordinates		F(x) Values		
F(x, y) values	**Coordinates**		**Average**		**Average**	**STD**	**%S-rate**
	X	**Y**	**X**	**Y**	**F(x, y)**		
199.997	3.58	-1.86	3.5849053	-1.848966	199.999627	0.00033659	100.00%
200	3	2	3.5849053	2.033505	199.131572	4.75501049	99.57%
199.995	-2.815	3.125	-2.80565	3.1305933	199.999206	0.00141548	100.00%
199.999	-3.78	-3.28	-3.7791243	-3.315798	199.16361	3.18058422	99.58%
199.997	3.58	-1.86	3.5849053	-1.848966	199.999627	0.00033659	100.00%
Average of success rate							99.79%

6.1.6 F6 One dimensional Shubert function:

The results in Table 11 and Table 12 are for Shubert function, the Shubert function is very popular in a lot of research paper, one dimensional Shubert function is calculated by ASNPSO but check with SNGA. In our thesis research we calculated with both algorithms for test result. In table 6 I just shown the result of global maxima, according of ASNPSO and previous more researches said that this function has total 19 maxima, in which three of them is global and 16 local maxima. I run the program for both algorithm 30 times and estimate the results that the ASNPSO achieved all solutions, but the SNGA can't perform very well. ASNPSO found all 19 solutions but the SNGA just found the global maxima. If

I conclude on the basis of global maxima then still ASNPSO found all three solution with 100% success rate which is better than SNGA.

Table 11: Result values of F6 obtained by SNGA algorithm.

Optimal		**Coordinates**		**F(x) Values**		
F(x, y) values	**Coordinates**	**Average**	**STD**	**Average F(x, y)**	**STD**	**%S-rate**
14.508	-7.0835	-7.08309	9.034E-16	14.507978	7.227E-15	100.00%
14.508	-0.8003	-0.801064	1.129E-16	14.507913	7.227E-15	100.00%
14.508	5.4828	5.48886	9.034E-16	14.501842	9.034E-15	99.96%
Average of success rate						99.99%

Table 12: Result values of F6 obtained by ASNPSO algorithm.

Optimal		**Coordinates**		**F(x) Values**		
F(x) values	**Coordinates**	**Average**	**STD**	**Average F(x)**	**STD**	**%S-rate**
14.508	-7.0835	-7.08354	0	14.5080077	1.084E-14	100.00%
14.508	-0.8003	-0.800276	1.129E-16	14.5080076	5.4202E-15	100.00%
14.508	5.4828	5.48364	9.034E-16	14.5079046	1.084E-14	100.00%
Average of success rate						100.00%

6.1.7 F7 Shekel foxhole De Jong two dimensional function:

The Table 13 and Table 14 are shekel foxhole function results. This is very complicated function. This function has total 25 peaks with different heights, where the range $-65 \leq x, y \leq 65$. I achieved 25 peaks with both algorithms with different heights, the result is shown in table7 and 7.1, but in the above function I convert the local minima to global maximum which is our requirement. The coordinate of both algorithms is quite different but we compared with mathematic value, then the ASNPSO is better than SNGA.

Table 13: Result values of F7 obtained by SNGA algorithm.

Optimal			**Coordinates**			
F(x, y) values	**Coordinates**		**Average**		**Average**	**STD**
	X	**Y**	**X**	**Y**	**F(x, y)**	
499.002	-32	-32	-31.43694	-32.003383	498.97026	2.31261E-13
0.03557	-16	-32	-15.5212	-31.4076	0.0372633	0
3.61705	0	-32	0.0356936	-31.8486	4.1704076	0
0.02814	16	-32	16.1594	-31.8962	0.0280066	0
0.05816	32	-32	31.693	-31.7388	0.0657182	0
0.05167	-32	-16	-31.8128	-16.2359	0.0527487	0
158.173	-16	-16	-15.839	-15.8974	179.86747	8.67228E-14
0.35879	0	-16	-0.126055	-16.1233	0.3844009	0
0.35275	16	-16	15.7276	-15.8988	0.3738079	0
158.362	32	-16	31.2606	-16.2416	212.66884	1.44538E-13
489.237	-32	0	-31.8348	0.376185	489.23409	2.89076E-13
495.063	-16	0	-16.592	-0.51292	474.11157	1.73446E-13
3.67699	0	0	0.164065	-0.197827	3.0782173	0
4.92253	16	0	16.3273	0.0260224	4.8627562	0
0.09381	32	0	31.8704	0.0713644	0.0971547	0
438.597	-32	16	-32.0864	16.3063	380.81915	1.73446E-13
157.205	-16	16	-15.9381	16.5443	288.09775	1.1563E-13
32.5357	0	16	0.225295	15.9617	31.015352	0
1.09315	16	16	15.8599	16.0134	1.0892633	0
157.447	32	16	31.279	16.1681	126.70272	8.67228E-14
479.847	-32	32	-31.8983	32.0083	479.84656	0
485.45	-16	32	-17.4608	33.1681	479.18692	0
3.77473	0	32	-0.126055	32.2368	4.6701335	0
4.94948	16	32	16.4404	31.9418	5.09374	1.73446E-13
12.4168	32	32	31.4031	31.7357	16.79233	0

Table 14: Result values of F7 obtained by ASNPSO algorithm.

Optimal			**Coordinates**		**F(x) Values**	
F(x, y) values	**Coordinates**		**Average**		**Average**	**STD**
	X	**Y**	**X**	**Y**	**F(x, y)**	
499.002	-32	-32	-31.43694	-32.003383	498.987563	0.07888835
0.03557	-16	-32	-15.947677	-32.017747	0.03562384	0.0003839
3.61705	0	-32	0.0076073	-32.023123	3.56154429	0.35764016
0.02814	16	-32	16.004147	-31.97726	0.02817954	0.00016419
0.05816	32	-32	32.02483	-32.006013	0.05813502	0.00343264
0.05167	-32	-16	-31.94972	-16.003823	0.05205151	0.00140092
158.173	-16	-16	-15.96251	-15.9858	163.062002	31.4551882
0.35879	0	-16	-0.0083122	-16.018917	0.36365907	0.02510766
0.35275	16	-16	16.01629	-16.012507	0.35169366	0.02536448
158.362	32	-16	31.966223	-16.0099	161.097546	18.6676248
489.237	-32	0	-31.956637	0.00655067	489.236821	6.4311E-07
495.063	-16	0	-15.988367	-0.0658067	493.396191	3.92909649
3.67699	0	0	0.0005743	0.00754252	3.71221601	0.26220069
4.92253	16	0	15.965523	0.02998102	4.87271173	0.16818463
0.09381	32	0	31.959147	0.00257594	0.09435996	0.00152292
438.597	-32	16	-31.95868	15.99566	438.80943	8.35346889
157.205	-16	16	-16.035597	16.02421	163.359341	27.557715
32.5357	0	16	0.0033588	16.00237	33.0043473	5.88211416
1.09315	16	16	15.971423	15.9838533	1.0988495	0.01647798
157.447	32	16	31.931333	16.00745	156.324058	11.5258447
479.847	-32	32	-31.955567	31.9707433	479.846561	1.6975E-06
485.45	-16	32	-15.979133	31.9360333	484.285377	1.1244379
3.77473	0	32	0.0087883	31.9391633	3.58582514	0.18792157
4.94948	16	32	15.997653	31.9373933	5.10929324	0.10856763
12.4168	32	32	31.937523	31.95986	13.0182758	0.82066718

6.1.8 F8 Two dimensional Shubert function:

The two dimensional Shubert function range is -10 ≤ x, y ≤ 10 and f(x, y)= -186.73. This function has total 18 global minima and 760 local minima for successfully finding in all global optimal, I set the termination condition in my both algorithm to 30 optimal. But the ASNPSO average successful rate is 100% instead of SNGA, once again ASNPSO proved that the ASNPSO is better than SNGA.

Table 15: Result values of F8 obtained by SNGA algorithm.

Optimal			**Coordinates**				
F(x, y) values	**Coordinates**		**Average**		**Average**	**STD**	**%S-rate**
	X	**Y**	**X**	**Y**	**F(x, y)**		
-185.97	-7.7	-7.1	-7.73352	-7.02997	-179.07018	5.78152E-14	96.29%
-186.57	-7.7	-0.8	-7.68006	-0.780445	-183.99712	0	98.62%
-185.92	-7.7	5.5	-7.70312	5.54254	-178.93391	8.67228E-14	96.24%
-184.61	-1.4	5.5	-1.39241	5.52945	-179.54219	8.67228E-14	97.26%
-185.25	-1.4	-0.8	-1.42127	-0.739953	-178.78281	2.89076E-14	96.51%
-184.65	-1.4	-7.1	-1.4251	-7.02254	-178.6605	2.89076E-14	96.75%
-181.96	4.9	5.5	4.85735	5.48363	-186.72845	8.67228E-14	100.00%
-182.6	4.9	-0.8	4.84014	-0.740897	-178.34555	5.78152E-14	97.67%
-182.01	4.9	-7.1	4.85735	-7.02254	-178.65938	8.67228E-14	98.16%
-185.97	-7.1	-7.7	-7.03782	-7.68106	-180.47156	1.1563E-13	97.04%
-184.65	-7.1	-1.4	-7.08908	-1.40407	-185.62357	5.78152E-14	100.00%
-182.01	-7.1	4.9	-7.06658	4.84448	-185.67353	0	100.00%
-182.6	-0.8	4.9	-0.788925	4.85927	-186.44105	8.67228E-14	100.00%
-185.25	-0.8	-1.4	-0.788925	-1.42366	-186.43945	8.67228E-14	100.00%
-186.57	-0.8	-7.7	-0.788925	-7.70889	-186.4437	0	99.93%
-181.96	5.5	4.9	5.48153	4.8497	-186.56465	8.67228E-14	100.00%
-184.61	5.5	-1.4	5.47314	-1.42069	-186.47578	8.67228E-14	100.00%
-185.92	5.5	-7.7	5.48561	-7.70201	-186.62148	0	100.00%
Average of success rate							98.58%

Table 16: Result values of F8 obtained by ASNPSO algorithm.

Optimal			Coordinates		F(x) Values		
F(x, y) values	**Coordinates**		**Average**		**Average**	**STD**	**%S-rate**
	X	**Y**	**X**	**Y**	**F(x, y)**		
-185.97	-7.7	-7.1	-7.7071997	-7.0832807	-184.85728	8.8493591	99.40%
-186.57	-7.7	-0.8	-7.7081637	-0.7477401	-178.86445	42.8056665	95.87%
-185.92	-7.7	5.5	-7.7080363	5.48209833	-186.6583	0.11363563	100.00%
-184.61	-1.4	5.5	-1.4246577	5.48383433	-186.56395	0.51774658	100.00%
-185.25	-1.4	-0.8	-1.4293163	-0.7985891	-185.23196	7.53362049	99.99%
-184.65	-1.4	-7.1	-1.424194	-7.08392	-186.68385	0.04575921	100.00%
-181.96	4.9	5.5	4.8597923	5.48438633	-186.62007	0.16554527	100.00%
-182.6	4.9	-0.8	4.8577923	-0.7985317	-186.62301	0.20361186	100.00%
-182.01	4.9	-7.1	4.8583357	-7.0807557	-186.62925	0.21391411	100.00%
-185.97	-7.1	-7.7	-7.0837853	-7.7132237	-186.65218	0.13194428	100.00%
-184.65	-7.1	-1.4	-7.083398	-1.4243343	-186.64977	0.12726198	100.00%
-182.01	-7.1	4.9	-7.0845227	4.85810433	-186.67806	0.13856716	100.00%
-182.6	-0.8	4.9	-0.7994361	4.858109	-186.65223	0.10343117	100.00%
-185.25	-0.8	-1.4	-0.802723	-1.426539	-186.53414	0.33489331	100.00%
-186.57	-0.8	-7.7	-0.7999102	-7.7087883	-186.68834	0.06392505	100.00%
-181.96	5.5	4.9	5.4831913	4.858068	-186.67012	0.10549358	100.00%
-184.61	5.5	-1.4	5.483517	-1.4253313	-186.66531	0.08766721	100.00%
-185.92	5.5	-7.7	5.483855	-7.7070773	-186.65642	0.10396365	100.00%
Average of success rate							99.74%

CHAPTER 7

DISCUSSION AND CONCLUSION

7.1 Discussion

After a function-statistics analysis of the features of SNGA and ASNPSO, the results of my research have shown that the success ratio of ASNPSO is higher than SNGA in many functions. In the one-dimensional Shubert function, the SNGA method couldn't find the local optimal, and to find it with SNGA, it is used a method where we even changed the parameter values and resulted with the same. The Shubert function has a total of 19 optimal, of which 3 are global, and 16 are local, whose result is 14.50 mathematically are global and less than 14.50 are local optimal. Using different termination conditions in SNGA to find the local optimal solution for the one-dimensional Shubert function, setting the function to 19 solutions to terminate the program and show the result values. Although it worked, but the values were same to global maxima, and only the coordinates were changed in some digits. This function has already been proved with different methods, even with math which have local optimals. However, SNGA failed.

In the previous discussion, I have come to the conclusion that the GA uses fitness sharing method and the PSO uses a different sharing method than GA, somewhat like birds in groups like a metaphor. In that case, the particles are unsystematically computed and openly move across the dimensional search space where all of them during their movement update their own velocity and location depending on the capital observation of their own and of the whole group. The reforming strategy runs the particle swarm to act against the area with the highest objective function value, and at the end, the total particles will collect around the point with the highest objective value, which is the moment when we are able to say that this is a sharing method.

As noticeable, SNGA uses a niche radius which prevents the algorithm of doing well, unlike ASNPSO, which for its niche radius uses the hill-valley method and avoids the niche radius which is discussed in the sections above.

Also, SNGA used distance matrix to find how the given two chromosomes return values that are closer to each other, on the other hand, in ASNPSO the death penalty function is used which makes the particle of the active sub-swarm to detect in the same hill of the superlative solution which is discovered before losing impact.

7.2 Conclusion

After the thorough differential of ASNPSO and SNGA for their multi-modal function optimization involving statistical data analysis, the result is that ASNPSO achieved superior success rate than SNGA.

Methods used in SNGA and ASNPSO like PSO and GA are motivated by EA and showed themselves to be successful in optimization problems. If these approaches achieved their required purposes it means that there are clearly robustness, in the meta-heuristics, there are control parameters involved and suitable setting, the parameters are key points in achieving the best solution. Basically, some form of examination and probation is needed for each specific example of the advanced problem.

The problem with the SNGA method is the niche radius while the ASNPSO method doesn't need. However, the ASNPSO method uses hill-valley function to regulate if the location of a moving particle and the best concentrating position of the group set afloat before belonging to one hill or not.

The ASNPSO method has had its search focus to optimal solutions by directing particles while the SNGA is a population-based method, and its search focus is done by moving population on average to the optimal solution. To avoid the problem with niche radius in SNGA, the hill-valley technique needs to be implemented in GA, which will be done in next research.

REFERENCES

Ackley, D. H. (1987). An empirical study of bit vector function optimization. *Genetic Algorithms and Simulated Annealing*, 170-204.

Ackley, D. H., & Jgretensetette, J. (1985). A connectionist algorithm for genetic search. *Grefenstette*, *876*, 121-135.

Beasley, D., Bull, D. R., & Martin, R. R. (1993). A sequential niche technique for multimodal function optimization. *Evolutionary Computation*, *1*(2), 101-125.

Davidor, Y. (1991, July). A Naturally Occurring Niche and Species Phenomenon: The Model and First Results. In *ICGA* (pp. 257-263).

De Jong, K. A. (1975). Analysis of the behavior of a class of genetic adaptive systems.

Deb, K. (1989). *Genetic Algorithms in Multimodal Function Optimization*. Clearinghouse for Genetic Algorithms, Department of Engineering Mechanics, University of Alabama.

Deb, K., & Goldberg, D. E. (1989, June). An investigation of niche and species formation in genetic function optimization. In *Proceedings of the 3rd International Conference on Genetic Algorithms* (pp. 42-50). Morgan Kaufmann Publishers Inc..

Eberhart, R. C., & Hu, X. (1999). Human tremor analysis using particle swarm optimization. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on* (Vol. 3, pp. 1927-1930). IEEE.

Eberhart, R. C., & Shi, Y. (2000). Comparing inertia weights and constriction factors in particle swarm optimization. In *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on* (Vol. 1, pp. 84-88). IEEE.

Guo, L., Huang, D. S., & Zhao, W. (2003). Combining genetic optimisation with hybrid learning algorithm for radial basis function neural networks. *Electronics Letters*, *39*(22), 1600-1601.

Kennedy, J., & Eberhart, R. (1942). Particle swarm optimization Proceedings of the International Conference on Neural Networks. *Australia IEEE*, *1948*.

Larrañaga, P., & Lozano, J. A. (Eds.). (2001). *Estimation of Distribution Algorithms: A new tool for Evolutionary Computation* (Vol. 2). Springer Science & Business Media.

Mahfoud, S. W. (1995). A comparison of parallel and sequential niching methods. In *Conference on Genetic Algorithms* (Vol. 136, p. 143).

Mahfoud, S. W. (1995). A comparison of parallel and sequential niching methods. In *Conference on Genetic Algorithms* (Vol. 136, p. 143).

Michalewicz, Z., Dasgupta, D., Le Riche, R. G., & Schoenauer, M. (1996). Evolutionary algorithms for constrained engineering problems. *Computers & Industrial Engineering*, *30*(4), 851-870.

Price, K. V. (1996, June). Differential evolution: a fast and simple numerical optimizer. In *Fuzzy Information Processing Society, 1996. NAFIPS., 1996 Biennial Conference of the North American* (pp. 524-527). IEEE.

Shi, Y., & Eberhart, R. (1998, May). A modified particle swarm optimizer. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on* (pp. 69-73). IEEE.

Tan, K. C., Lee, T. H., & Khor, E. F. (2001). Evolutionary algorithms with dynamic population size and local exploration for multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, *5*(6), 565-588.

Yeniay, Ö. (2005). Penalty function methods for constrained optimization with genetic algorithms. *Mathematical and Computational Applications*, *10*(1), 45-56.

Zaharie, D. (2004, September). Extensions of differential evolution algorithms for multimodal optimization. In *Proceedings of SYNASC* (Vol. 4, pp. 523-534).

Zhang, J., Huang, D. S., Lok, T. M., & Lyu, M. R. (2006). A novel adaptive sequential niche technique for multimodal function optimization. *Neurocomputing*, *69*(16), 2396-2401.

APPENDICES

APPENDIX 1

SNGA Algorithms CPP Source Code

```
//# include "stdafx.h"
# include <cstdlib>
# include <iostream>
# include <iomanip>
# include <fstream>
# include <iomanip>
# include <cmath>
# include <ctime>
# include <cstring>
# include "ModifiedFitnessFunction.h"


using namespace std;
//
//  Change any of these parameters to match your needs
//
#define PI 3.14159265
# define POPSIZE 50
# define MAXGENS 100
# define NVARS 1
# define PXOVER 0.25
# define PMUTATION 0.1
//
struct genotype
{
      double gene[NVARS];
      double fitness;
      double upper[NVARS];
      double lower[NVARS];
      double rfitness;
      double cfitness;
};

struct genotype population[POPSIZE + 1];
struct genotype newpopulation[POPSIZE + 1];

void crossover(int &seed);
void elitist();

//evaluate takes in parameter now the function to be optimized
template <class FunctionClass>
void evaluate(const FunctionClass & func);

int i4_uniform_ab(int a, int b, int &seed);
void initialize(string filename, int &seed);
void keep_the_best();
void mutate(int &seed);
double r8_uniform_ab(double a, double b, int &seed);
void report(int generation, ofstream & log_file);
void selector(int &seed);
void timestamp();
void Xover(int one, int two, int &seed);

struct BaseTestFunction {
      virtual const char * name()const = 0;
      virtual double operator()(double param)const = 0;
};

struct F1 : public BaseTestFunction {
      const char * name()const {
            return "pow(sin(5 * PI*param), 6)";
      }

      double operator()(double param)const {
            return pow(sin(5 * PI*param), 6);
      }
};

//new variables related to the paper
int nb_maxima = 5;


//ModifiedFitnessFunction<TestFunction> mff(tf,);

//****************************************************************
*********80

int main()

//****************************************************************
*********80

{
      ofstream log_file;
      log_file.open("log_f1.txt");
      string filename = "simple_ga_input.txt";


      double lbound, ubound;
      {
            ifstream input;
            input.open(filename.c_str());

            if (!input)
            {
                  cerr << "\n";
                  cerr << "INITIALIZE - Fatal error!\n";
                  cerr << "  Cannot open the input file!\n";
                  char rr;
                  std::cin >> rr;
                  exit(1);
            }
            //
            //  Initialize variables within the bounds
```

```
		//
		input >> lbound >> ubound;
	}

	//ModifiedFitnessFunction<TestFunction>
mff(tf,lbound,ubound,nb_maxima);
	//mff.addPeak();


	int generation;
	int i;
	int seed;

	timestamp();
	cout << "\n";
	cout << "SIMPLE_GA:\n";
	cout << "  C++ version\n";
	cout << "  Beasley sequential niche genetic algorithm.\n";

	if (NVARS < 2)
	{
		cout << "\n";
		cout << "  The crossover modification will not be
available,\n";
		cout << "  since it requires 2 <= NVARS.\n";
	}

	///init the four functions in an array

	F1 ff1;

	BaseModifiedFitnessFunction * mff = new
ModifiedFitnessFunction<F1>(ff1, lbound, ubound, nb_maxima);
//GetModifiedFitnessFunction(*(test_functions[i_function]));

	for (int i_max = 0; i_max < nb_maxima; ++i_max) {

		seed = 123456789;

		initialize(filename, seed);

		//ModifiedFitnessFunction<TestFunction> mff(tf, lbound,
ubound, nb_maxima);
		evaluate(*mff);

		keep_the_best();

		for (generation = 0; generation < MAXGENS; generation++)
		{
			selector(seed);
			crossover(seed);
			mutate(seed);
			report(generation, log_file);
			evaluate(*mff);
			elitist();
```

```
            }

            cout << "\n";
            cout << "  Best member after " << MAXGENS << "
generations:\n";
            cout << "\n";

            for (i = 0; i < NVARS; i++)
            {
                  cout << "  var(" << i << ") = " <<
population[POPSIZE].gene[i] << "\n";
                  log_file << "  var(" << i << ") = " <<
population[POPSIZE].gene[i] << "\n";
            }

            mff->addPeak(population[POPSIZE].gene[0]);

            cout << "\n";
            cout << "  Best fitness = " << population[POPSIZE].fitness
<< "\n";
            log_file << "\n";
            log_file << "  Best fitness = " <<
population[POPSIZE].fitness << "\n";


            timestamp();
      }
      log_file.close();
      //
      //  Terminate.
      //
      cout << "\n";
      cout << "SIMPLE_GA:\n";
      cout << "  Normal end of execution.\n";
      cout << "\n";

      delete mff;

      char iputchar;
      std::cout << "Enter a char to quit" << std::endl;
      std::cin >> iputchar;

      return 0;
}
//****************************************************************
*********80

void crossover(int &seed)

//****************************************************************
*********80

{
      const double a = 0.0;
      const double b = 1.0;
```

```
	int mem;
	int one;
	int first = 0;
	double x;

	for (mem = 0; mem < POPSIZE; ++mem)
	{
		x = r8_uniform_ab(a, b, seed);

		if (x < PXOVER)
		{
			++first;

			if (first % 2 == 0)
			{
				Xover(one, mem, seed);
			}
			else
			{
				one = mem;
			}

		}
	}
	return;
}
//*****************************************************************
*********80

void elitist()

//*****************************************************************
*********80

{
	int i;
	double best;
	int best_mem;
	double worst;
	int worst_mem;

	best = population[0].fitness;
	worst = population[0].fitness;

	for (i = 0; i < POPSIZE - 1; ++i)
	{
		if (population[i + 1].fitness < population[i].fitness)
		{

			if (best <= population[i].fitness)
			{
				best = population[i].fitness;
				best_mem = i;
			}
```

```
                  if (population[i + 1].fitness <= worst)
                  {
                        worst = population[i + 1].fitness;
                        worst_mem = i + 1;
                  }

            }
            else
            {

                  if (population[i].fitness <= worst)
                  {
                        worst = population[i].fitness;
                        worst_mem = i;
                  }

                  if (best <= population[i + 1].fitness)
                  {
                        best = population[i + 1].fitness;
                        best_mem = i + 1;
                  }

            }

      }
      //
      if (population[POPSIZE].fitness <= best)
      {
            for (i = 0; i < NVARS; i++)
            {
                  population[POPSIZE].gene[i] =
population[best_mem].gene[i];
            }
            population[POPSIZE].fitness =
population[best_mem].fitness;
      }
      else
      {
            for (i = 0; i < NVARS; i++)
            {
                  population[worst_mem].gene[i] =
population[POPSIZE].gene[i];
            }
            population[worst_mem].fitness =
population[POPSIZE].fitness;
      }

      return;
}
//**********************************************************************
*********80

template <class FunctionClass>
void evaluate(const FunctionClass & func)
```

```
//******************************************************************
*********80

{
      int member;
      int i;
      double x[NVARS + 1];

      for (member = 0; member < POPSIZE; member++)
      {
            for (i = 0; i < NVARS; i++)
            {
                  x[i + 1] = population[member].gene[i];
            }

            population[member].fitness = func(x[1]);//pow(sin(5 *
PI*x[1]), 6);

      }
      return;
}
//******************************************************************
*********80

int i4_uniform_ab(int a, int b, int &seed)

//******************************************************************
*********80

{
      int c;
      const int i4_huge = 2147483647;
      int k;
      float r;
      int value;

      if (seed == 0)
      {
            cerr << "\n";
            cerr << "I4_UNIFORM_AB - Fatal error!\n";
            cerr << "  Input value of SEED = 0.\n";
            exit(1);
      }
      //
      //  Guarantee A <= B.
      //
      if (b < a)
      {
            c = a;
            a = b;
            b = c;
      }

      k = seed / 127773;
      seed = 16807 * (seed - k * 127773) - k * 2836;

      if (seed < 0)
      {
            seed = seed + i4_huge;
      }

      r = (float)(seed) * 4.656612875E-10;
      //
      //  Scale R to lie between A-0.5 and B+0.5.
      //
      r = (1.0 - r) * ((float)a - 0.5)
            + r   * ((float)b + 0.5);
      //
      //  Use rounding to convert R to an integer between A and B.
      //
      value = round(r);
      //
      //  Guarantee A <= VALUE <= B.
      //
      if (value < a)
      {
            value = a;
      }
      if (b < value)
      {
            value = b;
      }

      return value;
}
//*****************************************************************
*********80

void initialize(string filename, int &seed)

//*****************************************************************
*********80

{
      int i;
      ifstream input;
      int j;
      double lbound;
      double ubound;

      input.open(filename.c_str());

      if (!input)
      {
            cerr << "\n";
            cerr << "INITIALIZE - Fatal error!\n";
            cerr << "  Cannot open the input file!\n";
            char rr;
            std::cin >> rr;
            exit(1);
      }
      //
      //  Initialize variables within the bounds
      //
      for (i = 0; i < NVARS; i++)
      {
            input >> lbound >> ubound;

            for (j = 0; j < POPSIZE + 1; j++)
            {
                  population[j].fitness = 0;
                  population[j].rfitness = 0;
                  population[j].cfitness = 0;
                  population[j].lower[i] = lbound;
                  population[j].upper[i] = ubound;
                  population[j].gene[i] = r8_uniform_ab(lbound,
ubound, seed);
            }
      }

      input.close();

      return;
}
//********************************************************************
*********80

void keep_the_best()

//********************************************************************
*********80

{
      int cur_best;
      int mem;
      int i;

      cur_best = 0;

      for (mem = 0; mem < POPSIZE; mem++)
      {
            if (population[POPSIZE].fitness < population[mem].fitness)
            {
                  cur_best = mem;
                  population[POPSIZE].fitness =
population[mem].fitness;
            }
      }
      //
      //  Once the best member in the population is found, copy the
genes.
      //
      for (i = 0; i < NVARS; i++)
      {
            population[POPSIZE].gene[i] =
population[cur_best].gene[i];
      }

      return;
}
//**********************************************************************
*********80

void mutate(int &seed)

//**********************************************************************
*********80

{
      const double a = 0.0;
      const double b = 1.0;
      int i;
      int j;
      double lbound;
      double ubound;
      double x;

      for (i = 0; i < POPSIZE; i++)
      {
            for (j = 0; j < NVARS; j++)
            {
                  x = r8_uniform_ab(a, b, seed);
                  if (x < PMUTATION)
                  {
                        lbound = population[i].lower[j];
                        ubound = population[i].upper[j];
                        population[i].gene[j] = r8_uniform_ab(lbound,
ubound, seed);
                  }
            }
      }

      return;
}
//**********************************************************************
*********80

double r8_uniform_ab(double a, double b, int &seed)

//**********************************************************************
*********80

{
      int i4_huge = 2147483647;
      int k;
      double value;

      if (seed == 0)
```

```
	{
		cerr << "\n";
		cerr << "R8_UNIFORM_AB - Fatal error!\n";
		cerr << "  Input value of SEED = 0.\n";
		exit(1);
	}

	k = seed / 127773;

	seed = 16807 * (seed - k * 127773) - k * 2836;

	if (seed < 0)
	{
		seed = seed + i4_huge;
	}

	value = (double)(seed) * 4.656612875E-10;

	value = a + (b - a) * value;

	return value;
}
//********************************************************************
*********80

void report(int generation, ofstream & log_file)

//********************************************************************
*********80

{
	double avg;
	double best_val;
	int i;
	double square_sum;
	double stddev;
	double sum;
	double sum_square;

	if (generation == 0)
	{
		cout << "\n";
		cout << "  Generation       Best            Average
Standard \n";
		cout << "  number           value           fitness
deviation \n";
		cout << "\n";
		log_file << "\n";
		log_file << "  Generation       Best            Average
Standard \n";
		log_file << "  number           value           fitness
deviation \n";
		log_file << "\n";
	}
```

```
      sum = 0.0;
      sum_square = 0.0;

      for (i = 0; i < POPSIZE; i++)
      {
            sum = sum + population[i].fitness;
            sum_square = sum_square + population[i].fitness *
population[i].fitness;
      }

      avg = sum / (double)POPSIZE;
      square_sum = avg * avg * POPSIZE;
      stddev = sqrt((sum_square - square_sum) / (POPSIZE - 1));
      best_val = population[POPSIZE].fitness;

      cout << "  " << setw(8) << generation
            << "  " << setw(14) << best_val
            << "  " << setw(14) << avg
            << "  " << setw(14) << stddev << "\n";

      log_file << "  " << setw(8) << generation
            << "  " << setw(14) << best_val
            << "  " << setw(14) << avg
            << "  " << setw(14) << stddev << "\n";

      return;
}
//*****************************************************************
*********80

void selector(int &seed)

//*****************************************************************
*********80

{
      const double a = 0.0;
      const double b = 1.0;
      int i;
      int j;
      int mem;
      double p;
      double sum;
      //
      //  Find the total fitness of the population.
      //
      sum = 0.0;
      for (mem = 0; mem < POPSIZE; mem++)
      {
            sum = sum + population[mem].fitness;
      }
      //
      //  Calculate the relative fitness of each member.
      //
      for (mem = 0; mem < POPSIZE; mem++)
```

```
        {
              population[mem].rfitness = population[mem].fitness / sum;
        }
        //
        //  Calculate the cumulative fitness.
        //
        population[0].cfitness = population[0].rfitness;
        for (mem = 1; mem < POPSIZE; mem++)
        {
              population[mem].cfitness = population[mem - 1].cfitness +
                    population[mem].rfitness;
        }
        //
        //  Select survivors using cumulative fitness.
        //
        for (i = 0; i < POPSIZE; i++)
        {
              p = r8_uniform_ab(a, b, seed);
              if (p < population[0].cfitness)
              {
                    newpopulation[i] = population[0];
              }
              else
              {
                    for (j = 0; j < POPSIZE; j++)
                    {
                          if (population[j].cfitness <= p && p <
population[j + 1].cfitness)
                          {
                                newpopulation[i] = population[j + 1];
                          }
                    }
              }
        }
        //
        //  Overwrite the old population with the new one.
        //
        for (i = 0; i < POPSIZE; i++)
        {
              population[i] = newpopulation[i];
        }

        return;
}
//****************************************************************************
*********80

void timestamp()

//****************************************************************************
*********80

{
# define TIME_SIZE 40
```

```
      static char time_buffer[TIME_SIZE];
      const struct tm *tm;
      size_t len;
      time_t now;

      now = time(NULL);
      tm = localtime(&now);

      len = strftime(time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p",
tm);

      cout << time_buffer << "\n";

      return;
# undef TIME_SIZE
}

// void timestamp()

//
//**********************************************************************
*********80

// {
// # define TIME_SIZE 40

//    static char time_buffer[TIME_SIZE];
//    struct tm tm_v;
//    size_t len;
//    time_t now;

//    now = time(NULL);
//    localtime(&tm_v,&now);

//    len = strftime(time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p",
&tm_v);

//    cout << time_buffer << "\n";

//    return;
// # undef TIME_SIZE
// }
//**********************************************************************
*********80

void Xover(int one, int two, int &seed)

//**********************************************************************
*********80
//

{
      int i;
      int point;
      double t;
      //
      //  Select the crossover point.
      //
      point = i4_uniform_ab(0, NVARS - 1, seed);
      //
      //  Swap genes in positions 0 through POINT-1.
      //
      for (i = 0; i < point; i++)
      {
            t = population[one].gene[i];
            population[one].gene[i] = population[two].gene[i];
            population[two].gene[i] = t;
      }

      return;
}
```

For all functions the program is same just only change the function statement.

F 2:

```
struct F2 : public BaseTestFunction {
      const char * name()const {
            return "exp(-2*log(2)*(pow((x - 0.1)/0.8,2))) * pow(sin(5
* PI*x), 6)";
      }

      double operator()(double x)const {
            return exp(-2 * log(2)*(pow((x - 0.1) / 0.8, 2))) *
pow(sin(5 * PI*x), 6);
      }
};
```

F 3:

```
struct F3 : public BaseTestFunction {
      const char * name()const {
            return "pow(sin(5 * PI*(pow(x,0.75) - 0.05)), 6)";
      }
      double operator()(double x)const {
            return pow(sin(5 * PI*(pow(x, 0.75) - 0.05)), 6);
      }
};
```

F 4:

```
struct F4 : public BaseTestFunction {
      const char * name()const {
            return "exp(-2 * log(2)*(pow((x - 0.08) / 0.854, 2))) *
pow(sin(5 * PI*(pow(x, 0.75) - 0.05)), 6)";
      }

      double operator()(double x)const {
            return exp(-2 * log(2)*(pow((x - 0.08) / 0.854, 2))) *
pow(sin(5 * PI*(pow(x, 0.75) - 0.05)), 6);
      }
};
```
F 5:
```
struct F5 {
      const char * name()const {
            return "200.0 - pow(x*x + y - 11.0,2.0) - pow(x + y*y - 
7.0,2.0)";
      }

      double operator()(double x,double y)const {
            return 200.0 - pow(x*x + y - 11.0,2.0) - pow(x + y*y - 
7.0,2.0);
      }
};
```
F 6:
```
struct F6 : public BaseTestFunction {
      const char * name()const {
            return "sum(i * cos((i + 1.0) * x + 1)";
      }

      double operator()(double x)const {
            double ret = 0;
            for (double i = 1; i <= 5; ++i) {
                  ret += i * cos((i + 1.0) * x + i);
            }
            return ret;
      }
};
```

F 7:
```
struct F7 {
      double operator()(double x,double y)const {
            double sum = 0;
            for (double i = 0; i <= 24; ++i) {
                  sum += 1.0 / (1.0 + i + pow(x - function_a(i), 6) + 
pow(y - function_b(i), 6));
            }

            return 500-1 / (0.002 + sum);
      }
};
```

F 8:
```
struct F8 {
      double operator()(double x,double y)const {
            double product = 1.0;
            double sum;

            double params[2];
            params[0] = x;
            params[1] = y;

            for (double i = 1; i <= 2; ++i) {
                  sum = 0;
                  for (double j = 1; j <= 5; ++j) {
                        sum += j * cos((j + 1.0) * params[(int)(i)-1]
+ j);
                  }

                  product *= sum;
            }

            return -product;
      }
};
```

ModifiedFitnessFunction code:

```
#include <vector>
#include <cmath>

class BaseModifiedFitnessFunction {
public:
      virtual void addPeak(double peak) = 0;

      virtual double operator()(double val)const = 0;
};

template <class Function>
class ModifiedFitnessFunction : public BaseModifiedFitnessFunction{
public:
      typedef Function FunctionType;

      ModifiedFitnessFunction(const Function & f,double
low_bound,double up_bound,int nb_maxima) :

      m_function(f),m_nbMaxima(nb_maxima),m_low_bound(low_bound),m_up_
bound(up_bound){
        m_radius = 1.0 / (2.0 * m_nbMaxima);
      }

      void addPeak(double peak);

      double operator()(double val)const;

      static double alpha;
private:
      double m_low_bound;
      double m_up_bound;
      double m_nbMaxima;
      double m_radius;
      Function m_function;
      std::vector<double> m_peaks;
};

template <class Function>
double ModifiedFitnessFunction<Function>::alpha = 2.0;
```

```
template <class Function>
void ModifiedFitnessFunction<Function>::addPeak(double peak) {
      m_peaks.push_back(peak);
}

template <class Function>
double ModifiedFitnessFunction<Function>::operator()(double val)
const{
      double raw_val = m_function(val);
      val = (val - m_low_bound)/(m_up_bound -
m_low_bound);//normalization

      double dist;
    for (int i = 0; i < m_peaks.size(); ++i){
        dist = fabs(val - m_peaks[i]);
        if(dist < m_radius){
                  raw_val *= std::pow(dist/m_radius,alpha);//Gp in the
paper (first one)
        }
    }

      return raw_val;
}

//template <class Function>
//ModifiedFitnessFunction<Function> GetModifiedFitnessFunction(const
Function & function) {
//    ModifiedFitnessFunction<Function> mff;
//    return mff;
//}

//template <class TModifiedFitnessFunction>
//struct GetGet {
//    typedef
//
//};
```

ModifiedFitnessFunction2D code:

```
#pragma once

#include <vector>
#include <cmath>

template <class Function>
class ModifiedFitnessFunction2D {
public:
      struct Param {
            Param(double x_, double y_) : x(x_), y(y_) {}

            double dist(const Param & other)const {
                  return sqrt(pow(x - other.x, 2) + pow(y - other.y,
2));
```

```
            }

            double x;
            double y;
      };

      typedef Function FunctionType;

      ModifiedFitnessFunction2D(const Function & f, double low_bound,
double up_bound, int nb_maxima) :
            m_function(f), m_nbMaxima(nb_maxima),
m_low_bound(low_bound), m_up_bound(up_bound) {
            m_radius = 1.0 / (2.0 * m_nbMaxima);
      }

      void addPeak(double peak_x,double peak_y);

      void addPeak(const Param & param_);

      double operator()(double val_x, double val_y)const;

      void normalize(Param & param_)const;

      static double alpha;
private:
      double m_low_bound;
      double m_up_bound;
      double m_nbMaxima;
      double m_radius;
      Function m_function;
      std::vector<Param> m_peaks;
};

template <class Function>
double ModifiedFitnessFunction2D<Function>::alpha = 2.0;

template <class Function>
void ModifiedFitnessFunction2D<Function>::normalize(Param &
param_)const {
      param_.x = (param_.x - m_low_bound) / (m_up_bound -
m_low_bound);
      param_.y = (param_.y - m_low_bound) / (m_up_bound -
m_low_bound);
}

template <class Function>
void ModifiedFitnessFunction2D<Function>::addPeak(double peak_x,
double peak_y) {
      Param p(peak_x, peak_y);
      normalize(p);
      m_peaks.push_back(p);
}

template <class Function>
```

```
void ModifiedFitnessFunction2D<Function>::addPeak(const Param &
param_) {
      Param p(param_);
      normalize(p);

      m_peaks.push_back(p);
}

template <class Function>
double ModifiedFitnessFunction2D<Function>::operator()(double val_x,
double val_y) const {
      double raw_val = m_function(val_x,val_y);
      //val_x = (val_x - m_low_bound) / (m_up_bound -
m_low_bound);//normalization
      //val_y = (val_y - m_low_bound) / (m_up_bound -
m_low_bound);//normalization

      Param val_peak(val_x, val_y);
      normalize(val_peak);

      double dist;
      for (int i = 0; i < m_peaks.size(); ++i) {
            //dist = fabs(val - m_peaks[i]);
            dist = val_peak.dist(m_peaks[i]);
            if (dist < m_radius) {
                  raw_val *= std::pow(dist / m_radius, alpha);//Gp in
the paper (first one)
            }
      }

      return raw_val;
}

//template <class Function>
//ModifiedFitnessFunction2D<Function>
GetModifiedFitnessFunction(const Function & function) {
//    ModifiedFitnessFunction2D<Function> mff;
//    return mff;
//}

//template <class TModifiedFitnessFunction>
//struct GetGet {
//    typedef
//
//};
```

APPENDIX 2

ASNPSO Algorithm Source Code

```
#include "Sworm.h"
#include "FitnessFunction.h"
#include <cmath>
#include "ASNPSO.h"
#include "FitnessFunction.h"
#include <fstream>

struct F1 {
      double operator()(const double * x)const {
            double ret = 0;
            for (double i = 1; i <= 5; ++i){
                  ret += i * cos((i + 1.0) * (*x) + i);
            }
            return ret;
      }
};

int main()
{
      std::ofstream log_file;
      log_file.open("log_f1.txt");

      int nb_maxima = 19;
      const int dim = 1;
      int SR = 10;
      int maxItertion = 10000;

      double W = 0.729;
      double C1 = 1.149445;
      double C2 = 1.149445;

      //double W = 0.01;
      //double C1 = 0.5;
      //double C2 = 0.5;

      double pos_lbound[1];
      (*pos_lbound) = -10;
      double pos_ubound[1];
      (*pos_ubound) = 10;

      double vel_lbound[1];
      (*vel_lbound) = -0.005;
      double vel_ubound[1];
      (*vel_ubound) = 0.005;
```

```
	FitnessFunction<dim, F1> ff(SR, 15, pos_lbound, pos_ubound);

	int population = 100;

	ASNPSO<dim, FitnessFunction<dim, F1> > asnpso(W, C1, C2,
pos_lbound, pos_ubound, vel_lbound, vel_ubound, population,
nb_maxima,ff);

	asnpso.run(maxItertion);


	//for (int i = 0; i < 100; ++i){
	//	Agent<dim> best_agent = little_sworm.getBestAgent();
	//	std::cout << "average fitness " <<
little_sworm.getAverageFitness() << " "
	//		<< "best position at " <<
best_agent.getPosition()[0] << " "
	//		<< "with fitness " << best_agent.getFitness() <<
std::endl;
	//	little_sworm.iterate();
	//}

	//std::cout << "Best position " <<
(*little_sworm.getBestAgent().getPosition()) << std::endl;

	asnpso.writeLog(std::cout);
	asnpso.writeLog(log_file);

	log_file.close();


	char inp;
	std::cin >> inp;

	return 0;
}
```

For all functions same as above in GA just change the function place.

Libraries of ASNPSO algorithm :

Agent.h

```
#pragma once

#include "RandomTool.h"
#include <algorithm>

template <int dim>
class Agent {
public:
	Agent<dim>(){}
	template <class FitnessFunction>
	void init(double * pos_lbound, double * pos_ubound, double *
vel_lbound, double * vel_ubound, const FitnessFunction & func);

	template <class FitnessFunction>
	void iterate(double W, double C1, double C2, const double *
bestAgentPos, const FitnessFunction & ff);

	Agent<dim> & operator=(const Agent<dim> & other);

	Agent(const Agent<dim> & other){ (*this) = other; }

	inline const double * getPosition()const { return _position; }

	inline const double * getBestPosition()const { return
_best_position; }

	inline double getFitness()const { return _fit; }

	inline double getBestFitness()const { return _best_fit; }
protected:
	double _fit;
	double _best_fit;
	double _position[dim];
	double _best_position[dim];
	double _velocity[dim];
};

template <int dim>
template <class FitnessFunction>
void Agent<dim>::init(double * pos_lbound, double * pos_ubound,
double * vel_lbound, double * vel_ubound, const FitnessFunction &
func) {
	for (int i = 0; i < dim; ++i) {
		_position[i] = random(pos_lbound[i], pos_ubound[i]);
		_velocity[i] = random(vel_lbound[i], vel_ubound[i]);

		_best_position[i] = _position[i];
	}

	_fit = func(*this);
	_best_fit = _fit;
}

template <int dim>
template <class FitnessFunction>
void Agent<dim>::iterate(double W, double C1, double C2, const double
* bestAgentPos, const FitnessFunction & ff) {
	for (int i = 0; i < dim; ++i) {
		_velocity[i] = W * _velocity[i] + C1 * random(0, 1) *
(_best_position[i] - _position[i]) + C2 * random(0, 1) *
(bestAgentPos[i] - _position[i]);
		_position[i] += _velocity[i];
	}
	_fit = ff(*this);
	if (_fit > _best_fit) {
		for (int i = 0; i < dim; ++i)
			_best_position[i] = _position[i];

		_best_fit = _fit;
	}
}

template <int dim>
Agent<dim> & Agent<dim>::operator=(const Agent<dim> & other) {
	_fit = other._fit;
	_best_fit = other._best_fit;

	for (int i = 0; i < dim; ++i) {
		_velocity[i] = other._velocity[i];
		_position[i] = other._position[i];
		_best_position[i] = other._best_position[i];
	}

	return (*this);
}
```

ASNPSO.cpp

```
#include "Sworm.h"
#include "FitnessFunction.h"
#include <cmath>
#include "ASNPSO.h"

struct Sinus {
	double operator()(const double * x)const {
		return std::sin(*x);
	}
};

int main()
{
	const int dim = 1;
	double W = 0.01;
	double C1 = 0.5;
	double C2 = 0.5;
	double pos_lbound[1];
	(*pos_lbound) = 0;
	double pos_ubound[1];
	(*pos_ubound) = 2 * 3.141;

	double vel_lbound[1];
	(*vel_lbound) = 0;
	double vel_ubound[1];
	(*vel_ubound) = 0.05;

	int population = 100;
```

```
      ASNPSO<1, RawFitnessFunction<1,Sinus> > little_sworm(W, C1, C2,
pos_lbound, pos_ubound, vel_lbound, vel_ubound, population);


      for (int i = 0; i < 100; ++i){
            Agent<dim> best_agent = little_sworm.getBestAgent();
            std::cout << "average fitness " <<
little_sworm.getAverageFitness() << " "
                  << "best position at " <<
best_agent.getPosition()[0] << " "
                  << "with fitness " << best_agent.getFitness() <<
std::endl;
            little_sworm.iterate();
      }

      std::cout << "Best position " <<
(*little_sworm.getBestAgent().getPosition()) << std::endl;


      char inp;
      std::cin >> inp;

    return 0;
}
```

ASNPSO.h

```
#include "Sworm.h"
#include "time.h"
#include <cassert>
#include <iomanip>

template <int dim, class FitnessFunction>
class ASNPSO : public Sworm<dim,FitnessFunction>{
public:
      ASNPSO(double W, double C1, double C2, double * pos_lbound, double
* pos_ubound, double * vel_lbound, double * vel_ubound, int
population,int nbMaxima = 1, FitnessFunction ff = FitnessFunction()) :
            Sworm<dim,FitnessFunction>(W, C1, C2, pos_lbound, pos_ubound,
vel_lbound, vel_ubound, population, ff), _nb_maxima(nbMaxima){
            _do_log = true;
            _window_size = 20;
            _log.str("");
      }

      void reinit();

      void run(int maxIteration);

      template <class stream>
      inline void writeLog(stream & s)const{
            s << _log.str();
      }
private:
      int _rand_seed;
```

```
	void runOneIterationLog()const;

	void runBeginLog()const;

	void runEndIterationLog()const;


	bool _do_log;
	mutable std::ostringstream _log;

	int _nb_maxima;
	int _window_size;
	bool end()const;

	mutable std::vector<double> _average_fitness;
};


template <int dim, class FitnessFunction>
void ASNPSO<dim, FitnessFunction>::run(int maxIteration){
	double average_fitness;
	bool do_reinit;

	//runBeginLog();

	for (int k = 0; k < _nb_maxima; ++k){
		do_reinit = false;

		runBeginLog();
		runOneIterationLog();

		for (int i = 0; i < maxIteration; ++i){
			Sworm<dim,FitnessFunction>::iterate();
			runOneIterationLog();

			if (end()){
				do_reinit = true;
				break;
			}

			average_fitness = Sworm<dim, 
FitnessFunction>::getAverageFitness();
			if (_average_fitness.size() >= _window_size){
				assert(_average_fitness.size() == _window_size);
				_average_fitness.erase(_average_fitness.begin());
			}

			_average_fitness.push_back(average_fitness);
		}

		this->_ff.addBestAgent(this->getBestAgent());
		//std::cout << "best agent raw value " << 
_ff.getRawFunction()(getBestAgent().getBestPosition()) << std::endl;
		if (_do_log)
			_log << std::endl;

		runEndIterationLog();
```

```
            if (_do_log)
                  _log << std::endl <<std::endl;

            _average_fitness.resize(0);

            //if (do_reinit)
            reinit();
      }
}

template <int dim, class FitnessFunction>
bool ASNPSO<dim, FitnessFunction>::end()const{
      if (_average_fitness.size() < _window_size)
            return false;

      double average_fitness = Sworm<dim,
FitnessFunction>::getAverageFitness();

      for (int i = 0; i < _average_fitness.size(); ++i)
            if (average_fitness < _average_fitness[i])
                  return false;

      return true;
}

template <int dim, class FitnessFunction>
void ASNPSO<dim, FitnessFunction>::reinit(){
      for (int i = 0; i < this->_agents.size(); ++i) {
            this->_agents[i].init(this->_pos_lbound, this->_pos_ubound,
this->_vel_lbound, this->_vel_ubound, this->_ff);
      }

      this->_best_agent = this->findBestAgent();
}

template <int dim, class FitnessFunction>
void ASNPSO<dim, FitnessFunction>::runOneIterationLog()const{
      if (!_do_log)
            return;

      const Agent<dim> & ag = this->getBestAgent();
      //_log << std::string("aoerij");
      _log << std::setw(16) << this->getAverageFitness() << std::setw(20)
<< ag.getBestPosition()[0];

      if (dim == 2)
            _log << ", " << ag.getBestPosition()[1];


      _log << std::setw(27) << ag.getBestFitness() << std::endl;
}

template <int dim, class FitnessFunction>
void ASNPSO<dim, FitnessFunction>::runBeginLog()const{
      if (!_do_log)
            return;
```

```
	_log << "average fitness |       best agent position       | best
agent fitness" << std::endl << std::endl;
}

template <int dim, class FitnessFunction>
void ASNPSO<dim, FitnessFunction>::runEndIterationLog()const{
	if (!_do_log)
		return;

	_log <<
"=========================================================================
"<<std::endl;
	_log << "=== maximum info : best agent position | best fitness |
function value : ";
	const Agent<dim> & ag = this->getBestAgent();
	_log << ag.getBestPosition()[0];

	if (dim == 2)
		_log << ", " << ag.getBestPosition()[1];


	_log << " | " << ag.getBestFitness() << " | " << this-
>_ff.getRawFunction()(ag.getBestPosition()) << std::endl;
	_log <<
"=========================================================================
";

}
```

FitnessFunction.h

```
#pragma once

#include "Agent.h"
#include <vector>
#include <algorithm>

template <int dim,class Function>
class FitnessFunction {
public:
	bool hillValley(const Agent<dim> & agent0,const Agent<dim> &
agent1) const;

	FitnessFunction(int SR, double penalty,const double *
lbound,const double * ubound) : _SR(SR), _penalty(penalty) {
		for (int i = 0; i < dim; ++i){
			_ubound[i] = ubound[i];
			_lbound[i] = lbound[i];
		}
	}

	double operator()(Agent<dim> & agent)const;

	inline void addBestAgent(const Agent<dim> & a){
_best_agents.push_back(a); }
```

```
	double bestRawValue(const Agent<dim> & agent)const{ return 
_raw_function(agent.getBestPosition()); }

	const std::vector<Agent<dim> > & getBestAgents()const{ return 
_best_agents; }

	const Function & getRawFunction()const{ return _raw_function; }
private:
	double _ubound[dim];
	double _lbound[dim];
	double _penalty;
	int _SR;
	Function _raw_function;
	std::vector<Agent<dim> > _best_agents;
};

template <int dim, class Function>
bool FitnessFunction<dim, Function>::hillValley(const Agent<dim> & 
agent0, const Agent<dim> & agent1) const{
	double step[dim];
	double position[dim];
	double minfit = 
std::min<double>(_raw_function(agent0.getPosition()), 
_raw_function(agent1.getBestPosition()));

	for (int i = 0; i < dim; ++i){
		step[i] = (agent1.getBestPosition()[i] - 
agent0.getPosition()[i]) / ((double)(_SR + 1));
		position[i] = agent0.getPosition()[i];
	}

	for (int i = 0; i < _SR; ++i){
		for (int j = 0; j < dim; ++j){
			position[j] += step[j];
		}

		if (_raw_function(position) < minfit)
			return true;
	}

	return false;
}

template <int dim, class Function>
double FitnessFunction<dim, Function>::operator()(Agent<dim> & 
agent)const{
	for (int i = 0; i < dim; ++i){
		if ((agent.getPosition()[i] < _lbound[i]) || 
(agent.getPosition()[i] > _ubound[i])){
			return _raw_function(agent.getPosition()) - 
_penalty;
		}
	}

	double fit = _raw_function(agent.getPosition());
	for (int i = 0; i < _best_agents.size(); ++i){
		if (!hillValley(agent, _best_agents[i])){
			fit -= _penalty;
		}
	}

	return fit;
}


template <int dim,class Function>
class RawFitnessFunction {
public:
	double operator()(const Agent<dim> & a)const {
		return _raw_function(a.getPosition());
	}
private:
	Function _raw_function;
};
```

RandomTool.cpp

```
#include "RandomTool.h"

double random(double lbound, double ubound) {
	return ((double)(rand())) / ((double)(RAND_MAX)) * (ubound -
lbound) + lbound;
}
```

RandomTool.h

```
#pragma once

#include <stdlib.h>

double random(double lbound, double ubound);
```