A REPORT

ON

APPLICATION OF NATURE-INSPIRED INTELLIGENT ALGORITHMS IN OPTIMIZATION STUDIES RELATED TO NUCLEAR REACTOR

BY

AMRIT KOCHAR 2014B4A7821P

AT

INDIRA GANDHI CENTER FOR ATOMIC RESEARCH, KALPAKKAM, TAMIL NADU

A Practice School-I station of

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

JUNE, 2016

A REPORT

ON

APPLICATION OF NATURE-INSPIRED INTELLIGENT ALGORITHMS IN OPTIMIZATION STUDIES RELATED TO NUCLEAR REACTOR

BY

AMRIT KOCHAR 2014B4A7821P M.Sc. (Hons.) Mathematics B.E. (Hons.) Computer Science

Prepared in partial fulfilment of the

Practice School-I Course

AT

INDIRA GANDHI CENTER FOR ATOMIC RESEARCH, KALPAKKAM, TAMIL NADU

A Practice School-I station of

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

JUNE, 2016

ACKNOWLEDGEMENT

I thank the management of BITS-Pilani and **Dr. A K Bhaduri**, Director, IGCAR for giving me an opportunity to undergo my Practice School-I program at such an esteemed Institution. It has been an honour and a privilege to be able to undergo training at such a prestigious institution.

I would like to express my gratitude to the Practice School-I program coordinator at IGCAR, **Dr. M Saibaba**, Associate Director, Resource Management Group, IGCAR.

I am thankful to my guide, **Dr. M L Jayalal,** for giving me an opportunity to study and do my project under him. The environment created by working under him was extremely conducive to the completion of my project.

I am grateful to **Dr. Michael Alphonse** (Faculty in Charge, Practice School-I), for his continuous guidance and assistance in academic as well as non-academic related matters.

I would like to thank **Dr. Kabali** for his support and coordination of all the Practice School-I related activities.

I would like to thank all my **friends** for providing their invaluable support and help during the course of my stay in Kalpakkam.

Birla Institute of Technology and Science Pilani, Pilani Campus
Practice School Division

Station: Indira Gandhi Center for Atomic Research

Centre: Kalpakkam

Duration: 23 May, 2016 - 16 July, 2016 **Date of Start:** 25 May 2016

Date of Submission: 12 June 2016

Title of the Project: Application of Nature Inspired Intelligent Algorithms in Optimization Studies related to Nuclear Reactor

ID No: 2014B4A7821P

Name: Amrit Kochar

Discipline: B.E. (Hons.) Computer Science, M.Sc. (Hons.) Mathematics

Name of the Guide: Dr. M L Jayalal

Designation of the Guide: Head, Computing Systems Section, Computer Division

Name of the PS Faculty: Dr. Michael Alphonse

Keywords: Optimization, Nature inspired, fuel bundle burnup, Ackley's function, Genetic Algorithms, Simulated Annealing, Particle Swarm Optimization

Project Areas: Optimization, Fuel Bundle Burnup Optimization Problem, Computational Intelligence

ABSTRACT

With the rising complexity of problems, there is an increased demand for better algorithms to squeeze out the best possible performance out of the limited resources available for a problem. Researchers have long argued, primitive beings do some specific tasks in a highly efficient manner. Nature inspired algorithms aim to learn from their mannerisms to try and get better results. This project comprises of a comparative study between three modern metaheuristic techniques, namely, Genetic Algorithms, Simulated Annealing and Particle Swarm Optimization and on a multi-modal function, the Ackley's function. In the second half of the project, a comparative study was carried out between Genetic Algorithms and Particle Swarm Optimization on the Fuel Bundle Burnup Optimization Problem of the Pressurized Heavy Water Reactor (PHWR). This study sets a strong base on future work in the field of computational intelligence and optimization. This also gives future researchers a head start to try and combine two or more algorithms in hopes to get a better result than when executed individually.

Signature of Student	Signature of PS Faculty
(Amrit Kochar)	(Dr. Michael Alphonse)
Date:	Date:

TABLE OF CONTENTS

1. INTRODUCTION

In the modern age of computers and information, our problems are becoming more and more complex. We need better ways to tackle them. The need calls for more efficient and accurate algorithms which are fast to execute. And as Plato had said, ‘necessity is the mother of invention.’ We see a plethora of new age algorithms being developed and improved upon by thousands of researchers for variety of purposes ranging from construction to poker.

At IGCAR, I did a comparative study of three metaheuristic techniques of optimization, namely**, Genetic Algorithm, Simulated Annealing and Particle Swarm Optimization.** We then applied all three algorithms to the Fuel Bundle Burnup Optimization Problem for Pressurized Heavy Water Reactor (PHWR). A research using GA’s had already been done by the officers of IGCAR and hence I applied the other two techniques in hope to get better results.

In computer science and mathematical optimization, a **metaheuristic** is a higher-level procedure or heuristic designed to find, generate, or select a heuristic (partial search algorithm) that may provide a sufficiently good solution to an optimization problem, especially with incomplete or imperfect information or limited computation capacity. The main focus is on combining exploration and exploitation of the search space effectively. This topic has a few classifications under its umbrella.

All three techniques are nature inspired. Researchers have studied a lot many natural processes to conclude that primitive beings have some highly optimized way to do a few tasks and we can learn from them. The three methods that we use all are nature based and stochastic in essence. Basically, they use random numbers to convergence on the best possible solution.

Three terms which are crucial when we talk about optimization algorithms are **Search Space, Exploration and Exploitation**. **Search Space** is the nth dimension space which is made from the constraints on the input variables. **Exploration** is the tendency to search all areas of the search space. **Exploitation** is the tendency to search a particularly small area of the space intensively.

We can classify optimization techniques on two parameters. **Number of solutions at a given point of time** and its **niche**. Each of these have two diversions. First one has **single solution and population based**. Second one has **Global search and local search.** Single solution would have one solution at a given time and the other will have a set of solutions. Global search would be better at getting to global optima and local one at getting to a local optima.

Genetic Algorithms (GA) was developed by Dr. John Holland and his students in the early 1970's is a **global search, population based** metaheuristic technique. It is inspired by human evolution by natural selection.

Simulated Annealing (SA) was independently described by Scott Kirkpatrick, C. Daniel Gelatt and Mario P. Vecchi in 1983 and by Vlado Černý in 1985. It is a **local search, single solution** based technique, inspired by slow cooling.

Particle Swarm Optimization (PSO) was initially formalized by Kennedy, Eberhart and Shi. It is a **global search, population based** method. It has been modelled on the movement of birds, fishes and locust in groups.

In this report, we start with a detailed introduction to all three algorithms and then move on to their comparative study on a multi modal test function, Ackley's. Then we study their application on the Fuel Bundle Burnup Optimization Problem.

2. THE THREE ALGORITHMS

2.1 GENETIC ALGORITHM

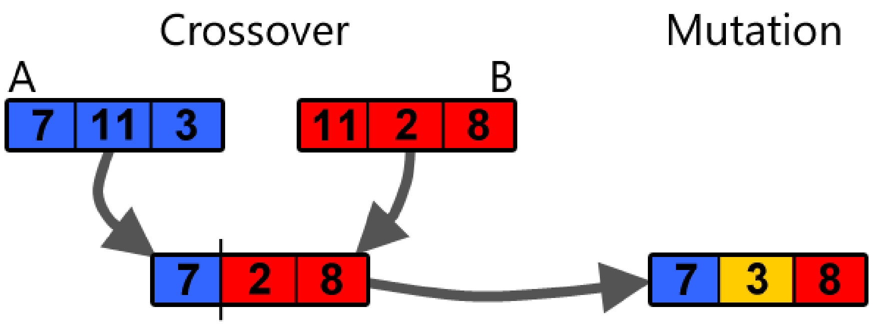
These are based on genetics and natural selection. This class of algorithms represent an intelligent exploitation of a random search used to solve optimization problems. Although randomised, Gas are by no means random, instead they exploit historical information to direct the search into the region of better performance within the search space. The basic techniques of the Gas are designed to simulate processes in natural systems necessary for evolution, especially those follow the principles first laid down by Charles Darwin of "survival of the fittest." Since in nature, competition among individuals for scanty resources results in the fittest individuals dominating over the weaker ones. Its main advantage is that it doesn't break upon change in input as well as it handles noise well.

2.1.1 Basic Algorithm:

```
Generate initial population randomly of size N;
Select Initial parent generation;
Do
    Crossover();
    Mutation();
    Selection next generation;
Until terminal criterion met
Final solution set
```

2.1.2 Explanation:

We randomly generate the starting population to kick off with. The function that we need to minimize is called fitness function. Then we select the fittest ones (we calculate fitness function value of each element in population) with corresponding probability distribution as per Darwin's theory of 'Survival of the fittest.' Two operations, namely, crossover and mutation are modelled on the basis of human genetic evolution. Crossover represents mating among individuals and mutation signifies random modifications. We apply them for maxgens iterations and after each select the fittest ones in a probabilistic manner to get convergence on the fittest solution. Crossover helps to introduce new solutions and help the population search the complete space. Mutation on the other hand, makes sure we do not suffer from premature convergence and introduces random changes to include searches among 'neglected spaces'.



This picture depicts the process of crossover and mutation on a 3 gene chromosome.

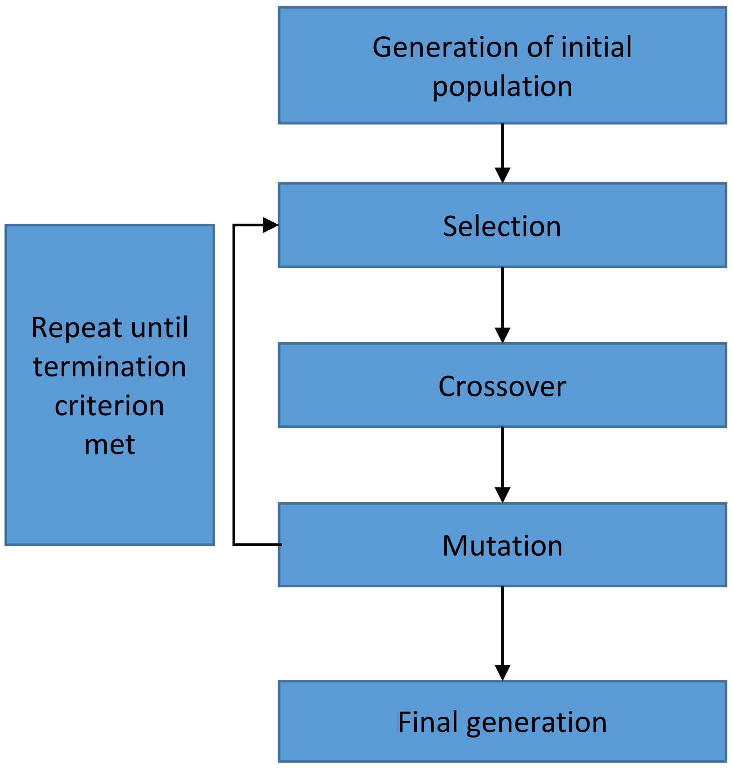
<u>The corresponding terminologies in natural evolution and GA</u>

Natural	**Genetic Algorithm**
Chromosome	String
Gene	Feature
Allele	Feature value
Locus	String position
Genotype	Structure
Phenotype	Parameter set
Epistasis	Nonlinearity

2.1.3 Limitations:

- Repeated fitness function evaluation is costly.
- Do not scale well with complexity due to exposure of lots of elements to mutation.
- No way to choose appropriate stop criterion.

Flow Chart-I: Genetic Algorithms


Generation of initial population
Selection
Crossover
Mutation
Final generation
Repeat until termination criterion met

2.2 SIMULATED ANNEALING

Simulated annealing (SA) is a metaheuristic to approximate global optimization in a large search space. It interprets slow cooling as a slow decrease in the probability of accepting worse solutions as it explores the solution space. Accepting worse solutions is a fundamental property of metaheuristics because it allows for a more extensive search for the optimal solution. Main advantages are that, it functions well with a discrete search space as well as with a large one. Another reason for its popularity is it very useful when it is less important to find a global optima and we can make do with a local one.

2.2.1 Basic Algorithm:

```
Set temperature, epsilon, alpha
Generate a random solution
Calculate its cost using fitness function
Do
    Generate a random neighbouring solution
    Calculate the new solution's cost, $c_{new}$
    Delta = $c_{new} – c_{old}$
    Compare them:
        If $c_{new} < c_{old}$ : move to the new solution
        If $c_{new} > c_{old}$ : maybe move to the new solution. (Maybe here is – selection
    probability is exp(-delta/temperature) )
    temperature = temperature * alpha
•Until (temperature > epsilon)
•Final solution
```

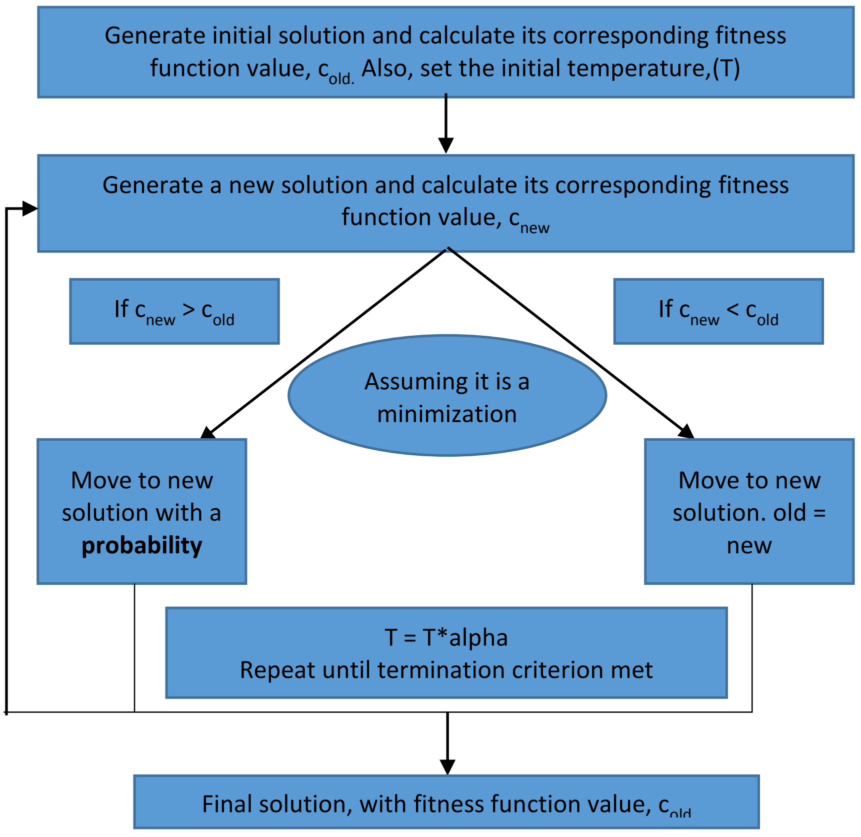
2.2.2 Explanation:

The central theme of this algorithm is as we decrease temperature, the probability of accepting a bad solution decreases. We accept bad solutions in the first place because it helps in better exploration of the search space. We will generate an initial solution and then in each iteration generate a new random solution and compare it with the previous one. If it is better then we accept it or else we accept it with a probability which diminishes as more and more iterations happen.

2.2.3 Limitations:

- It is difficult to tell whether we have an optimal solution at the end of the algorithm's run.
- For functions with a few local optima, SA uses unnecessary computational power.
- It is a slow process.
- When extra information is provided about the system, then SA is at a loss as compared to some other methods.

Flow Chart-II: Simulated Annealing


Generate initial solution and calculate its corresponding fitness function value, $c_{old.}$ Also, set the initial temperature,(T)
Generate a new solution and calculate its corresponding fitness function value, c_{new}
If $c_{new} > c_{old}$
If $c_{new} < c_{old}$
Assuming it is a minimization
Move to new solution with a **probability**
Move to new solution. old = new
T = T*alpha
Repeat until termination criterion met
Final solution, with fitness function value, c_{old}

2.3 PARTICLE SWARM OPTIMIZATION

PSO is a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. It solves a problem by having a population of candidate solutions, here dubbed particles, and moving these particles around in the search-space according to simple mathematical formulae over the particle's position and velocity. Each particle's movement is influenced by its local best known position but, is also guided toward the best known positions in the search-space, which are updated as better positions are found by other particles. This is expected to move the swarm toward the best solutions.

2.3.1 Basic Algorithm:

```
For each particle I = 1 to S do:
    Initialize the particle's position randomly
    Initialize the particle's best known position to its initial position: $p_i \leftarrow x_i$
    If ($f(p_i) < f(g)$) update the swarm's best known position: $g \leftarrow p_i$
    Initialize the particle's velocity: $v_i = 0$
Until termination criterion is met repeat:
    For each particle I = 1 to S do:
        For each dimension d = 1 to n do:
            Update the particle's velocity: $v_i[d] \leftarrow C0*v_i[d] + C1*r*(p_i[d]-x_i[d])$
        +          $C2*s*(g[d]-xi[d])$
        Update the particle's position: $x_i \leftarrow x_i + v_i$
        If ($f(x_i) < f(p_i)$) do:
            Update the particle's best known position: $p_i \leftarrow x_i$
            If ($f(p_i) < f(g)$) update the swarm's best known position: $g \leftarrow p_i$
Now g holds the best found solution.
```

2.3.2 Explanation:

Here, x denotes current position, p – particle's best position, f() – fitness function, g – swarm's best position, v – particle's velocity, CO, C1, C2 – constant weights to vary dependency on previous velocity, particle's best position and swarm's best position respectively, r, s – random numbers between 0 and 1.

The idea is to direct every particle towards better position by communicating details about swarm's best position to everyone, by keeping track of each particle's best position and its previous velocity.

2.3.3 Limitation:

- Performance highly depends on the values of C0, C1, and C2 and how they change with each iteration.
- It depends on a lot of random number generation. For some problems, it might deteriorate the accuracy of the method.
- Cannot work with problems with non-coordinate systems.
- A problem with velocity clamping is, if it reaches the upper limit it will continue its search within a hypercube and will not converge.

2.3.4 Modifications

We tried a lot of things with Simple PSO to improve its performance. Three significant alterations where incorporated.

2.3.4.1 Velocity Clamping

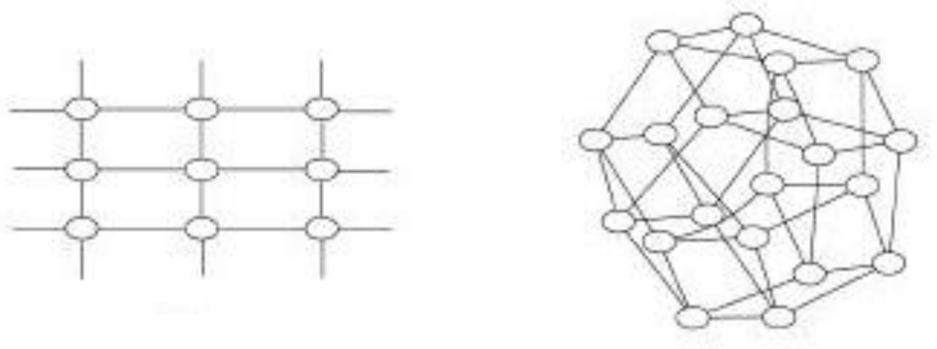
We have an upper limit on velocity so that it does not turn the algorithm into a random search algorithm. If we allow it to increase without bounds, it might make each particle to jump around the solution space without proper exploitation.

2.3.4.2 Better Parameters

There are three parameters that needs to be fine-tuned to obtain good solutions. These three dictate dependency on the following while updating velocity, previous velocity, personal best till now, and global best till now. We decrease the dependency on previous velocity linearly with increasing iterations so as to direct it more and more towards personal and global best.

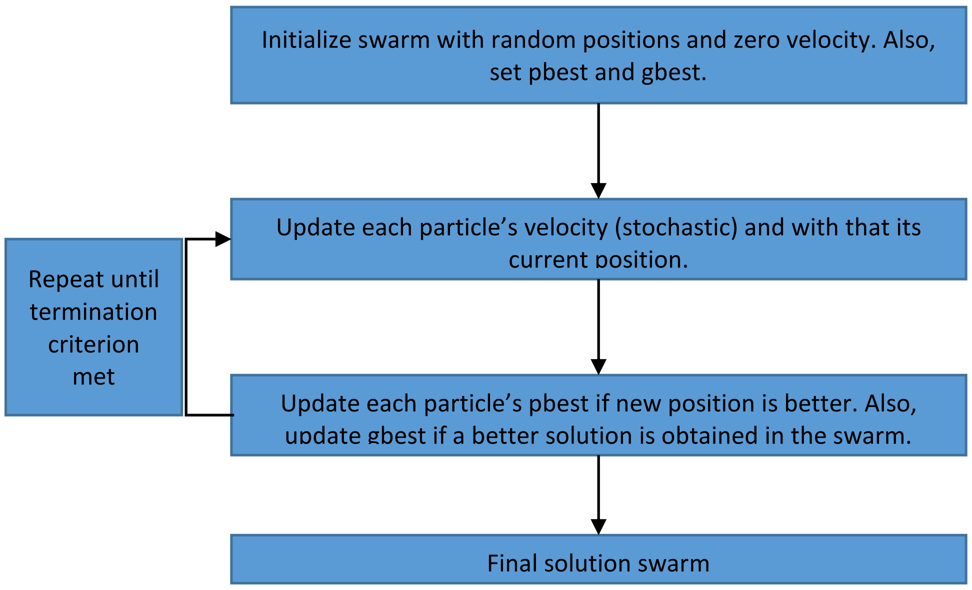
2.3.4.3 Efficient Topologies

The problem with gbest method of the Simple PSO is that it considers all particles together to calculate the swarm’s best which is used as gbest. This very factor results in premature convergence due to under exploitation. We tried two topologies to overcome this shortcoming, namely, **Von Neumann Topology and K Means Clustering Algorithm.** In the first one, each particle is connected to the nearest four points to it, one in each direction. So for that particular particle, gbest is calculated comparing the values of these five particles only and not the whole swarm. K Means Algorithm is a very popular technique in machine learning to cluster points. We use this to cluster our particles into sets. So, for any particle, its corresponding gbest is calculated by comparing the particles belonging to the same set.



A flattened and grid like representation of Von Neumann Topology.

Flow Chart-III: Particle Swarm Optimization


Initialize swarm with random positions and zero velocity. Also, set pbest and gbest.
Update each particle’s velocity (stochastic) and with that its current position.
Repeat until termination criterion met
Update each particle’s pbest if new position is better. Also, update gbest if a better solution is obtained in the swarm.
Final solution swarm

3. STUDY ON ACKLEY'S FUNCTION

3.1 THE FUNCTION

The Ackley function is widely used for testing optimization algorithms. It is characterized by a nearly flat outer region, and a large hole at the centre. It poses a risk for optimization algorithms, to be trapped in one of its many local minima. (Multi-modal). It is a one output function.

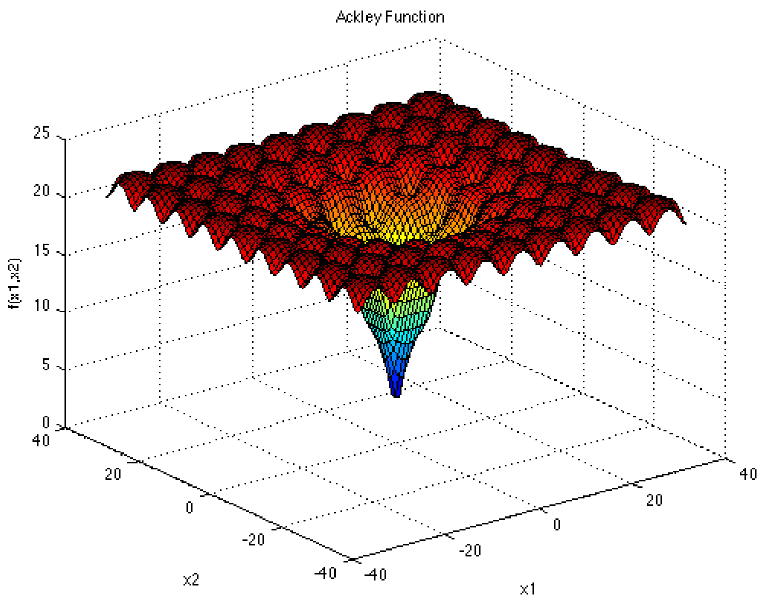
$$f(\mathbf{x}) = -a \exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

Values of constants:

a = 20, b = 0.2, c = 2π, d = 2 (number of dimensions)

Search space: -5 <= x1, x2 <= 5

Global Minimum: It occurs at x1 = 0 = x2 with the value, f(x1, x2) = 0



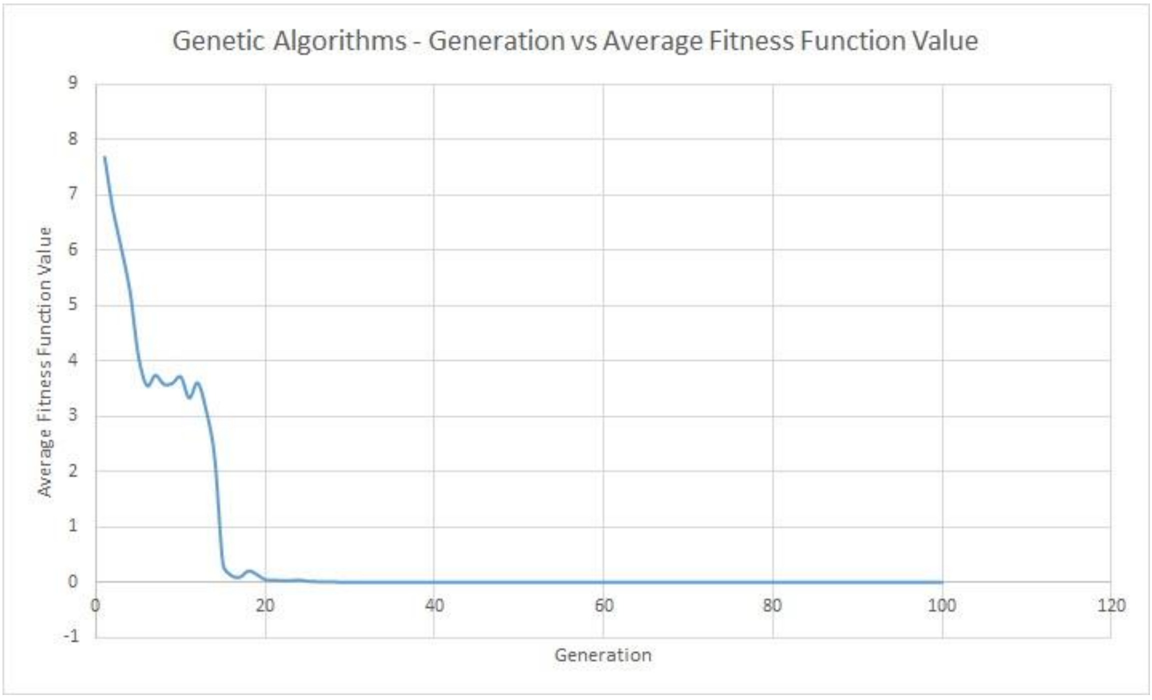
3.2 PLOTS

We ran all three algorithms on Ackley's function and compared their performance.

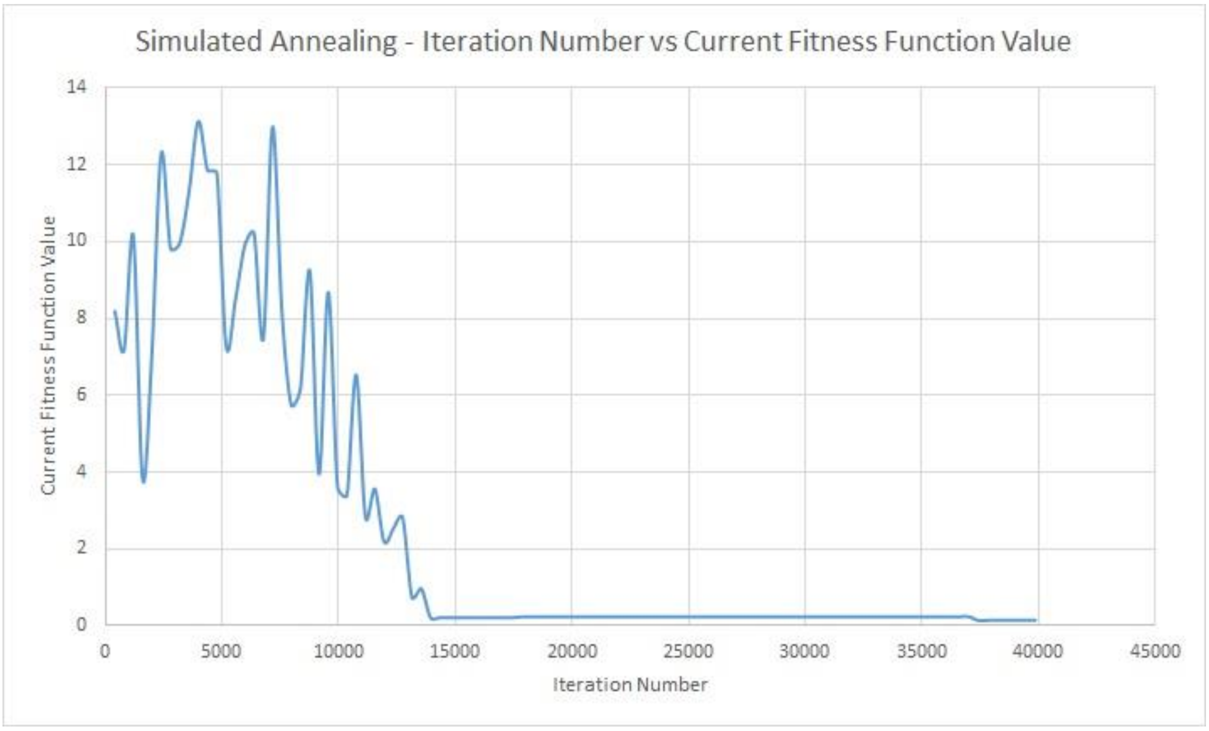
	GA	**SA**	**PSO**
Time to obtain a feasible solution (in milliseconds)	6.9784	50.0688	.06466
Best solution obtained (when run for equal time duration)	0.000153	0.144194	0.010372

Table I: Results of Study of Ackley's Function

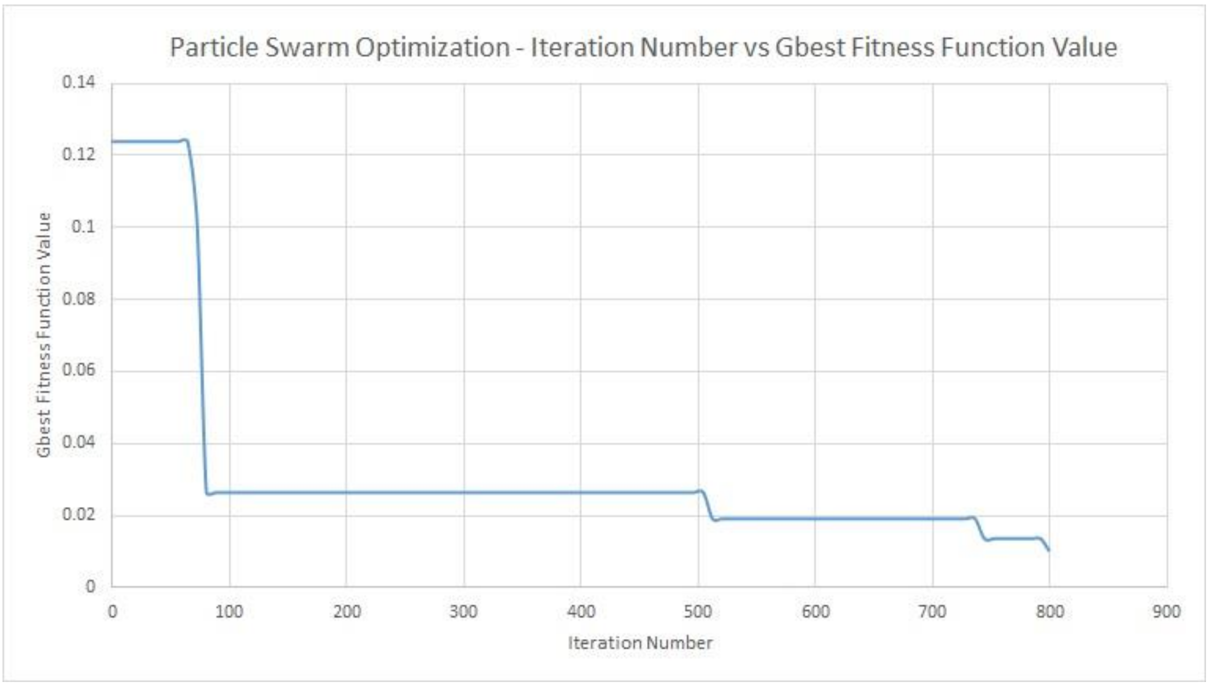
When all three were executed for nearly the same time duration, these plots were generated.



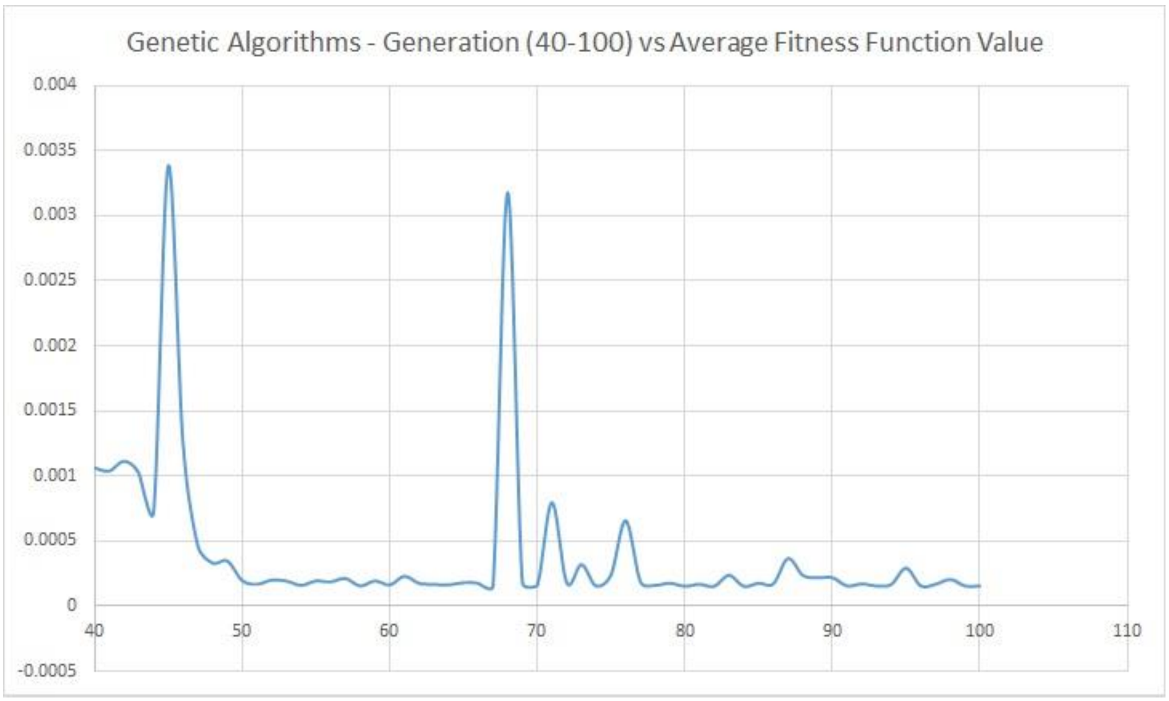
Plot I: Genetic Algorithms on Ackley's (Generation vs Average Fitness Function Values)



Plot II: Simulated Annealing on Ackley's (Iteration Number vs Current Fitness Function Value)

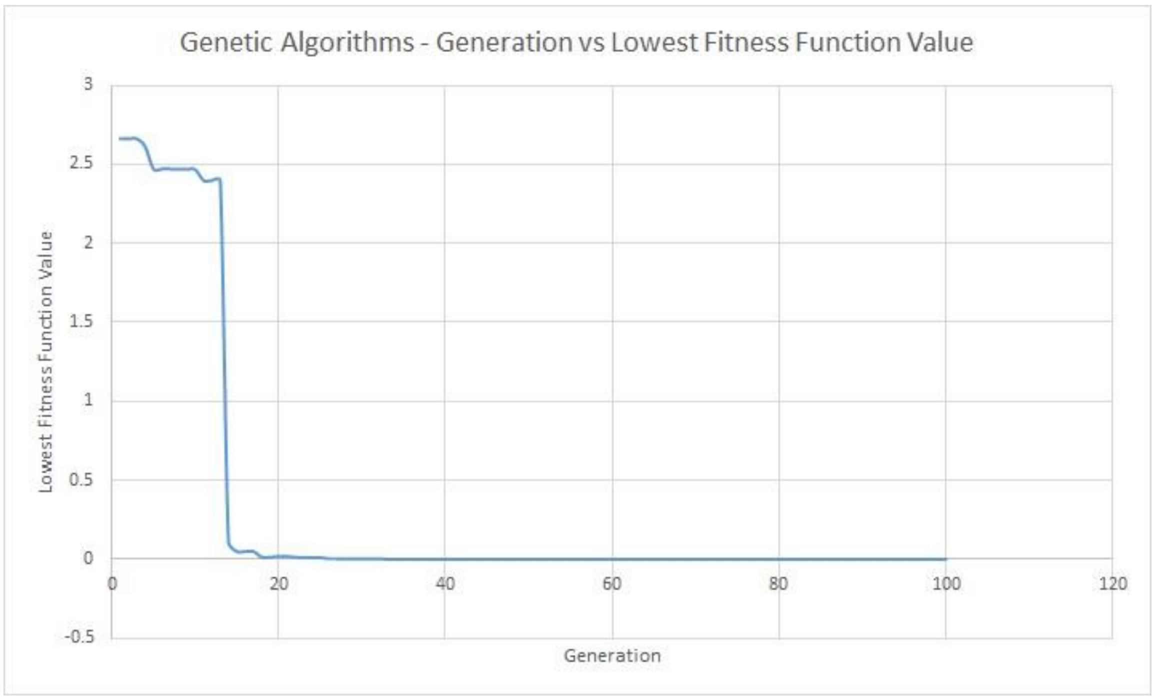


Plot III: Particle Swarm Optimization on Ackley's (Iteration Number vs Gbest Fitness Function Value)

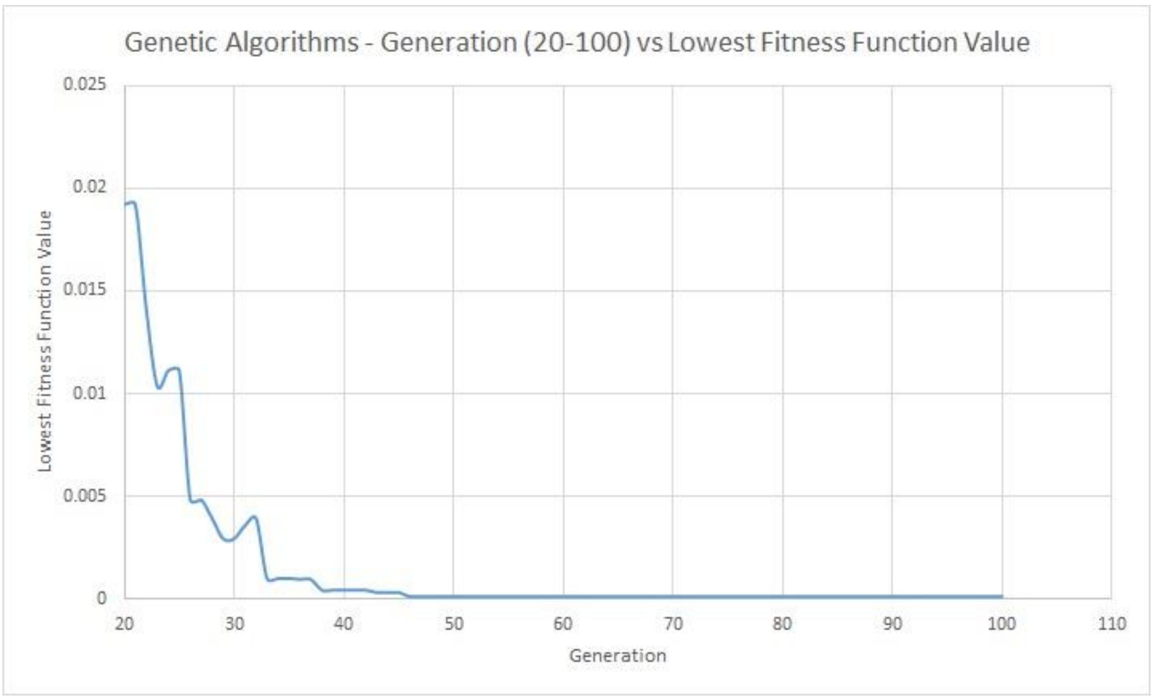


Plot IV: Genetic Algorithms on Ackley's (Generation vs Average Fitness Function Values)

(A zoomed view of generations 40 to 100)

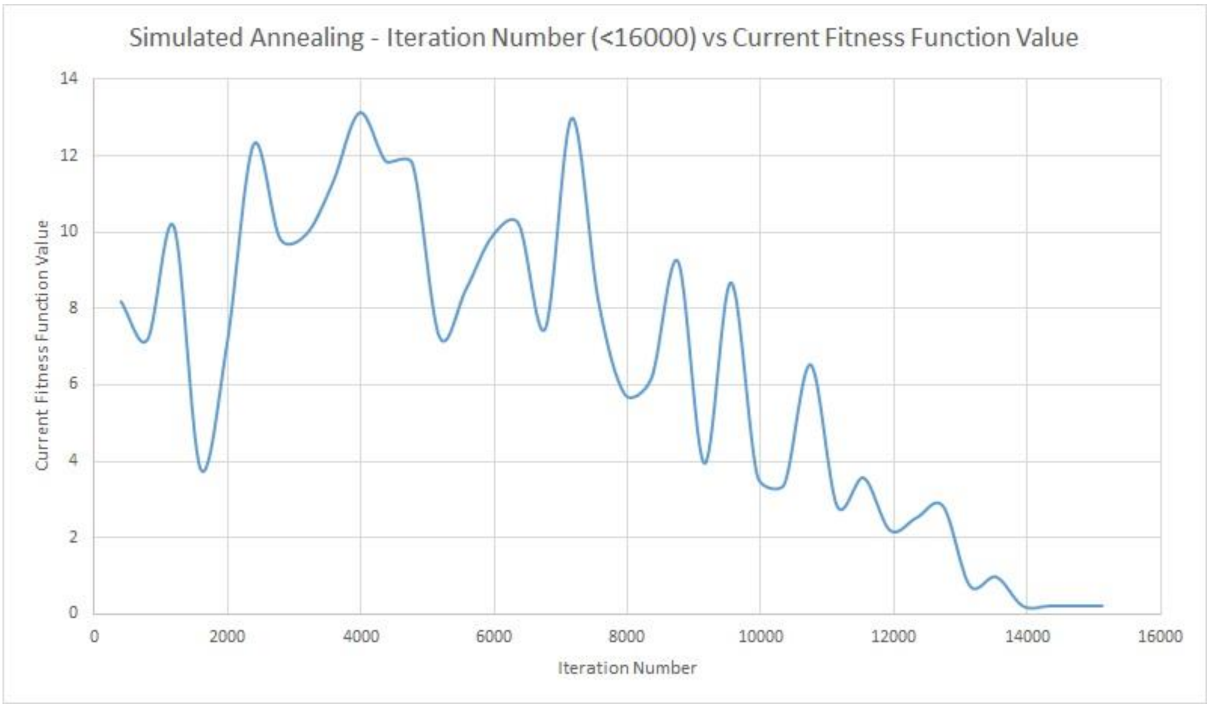


Plot V: Genetic Algorithms on Ackley's (Generation vs Lowest Fitness Function Values)



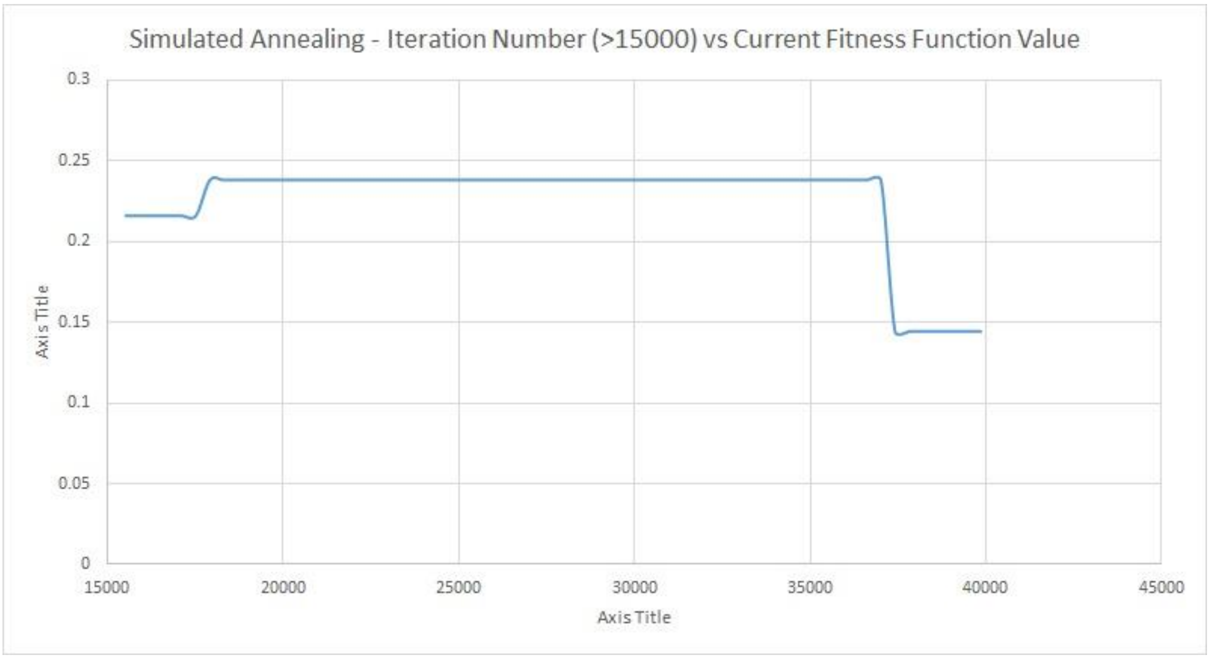
Plot VI: Genetic Algorithms on Ackley's (Generation vs Lowest Fitness Function Values)

(A zoomed view of generations 20 to 100)



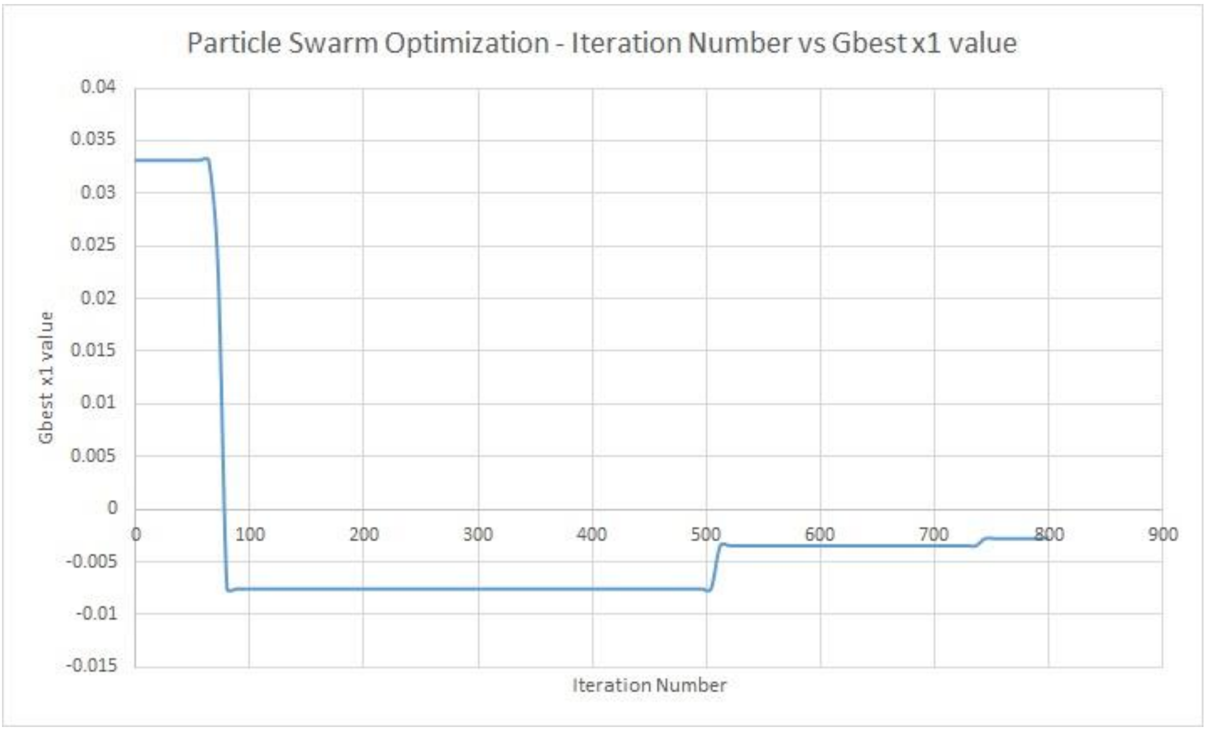
Plot VII: Simulated Annealing on Ackley's (Iteration Number vs Current Fitness Function Value)

(A zoomed view of iterations below 16000)

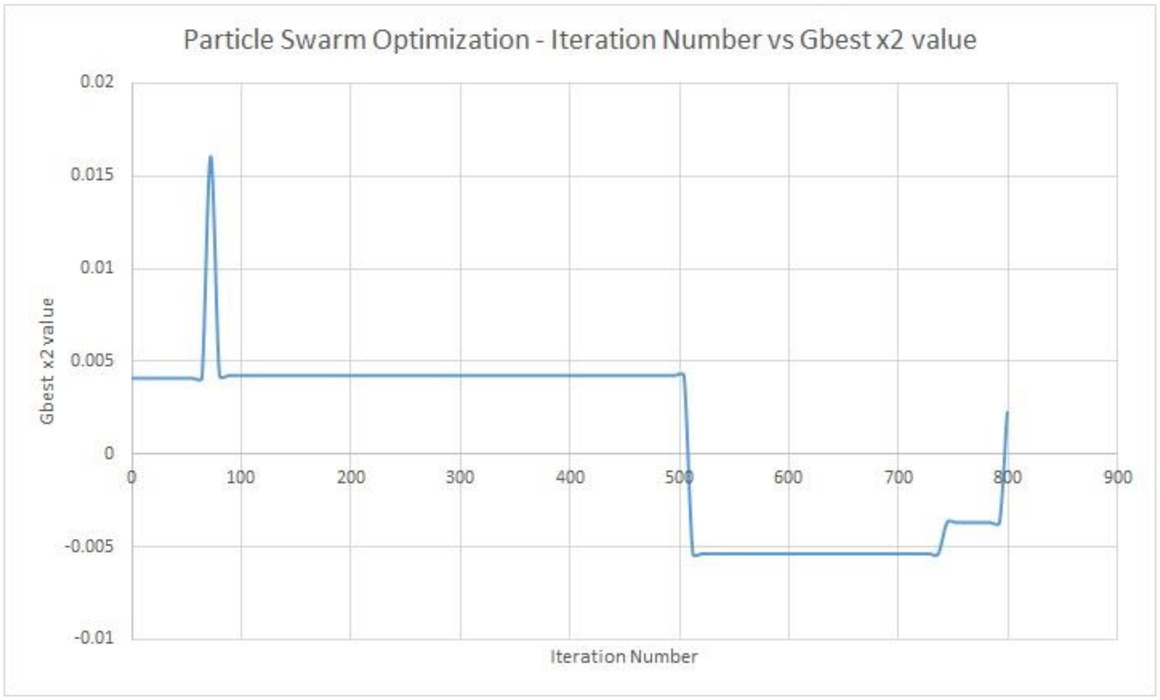


Plot VIII: Simulated Annealing on Ackley's (Iteration Number vs Current Fitness Function Value)

(A zoomed view of iterations above 15000)



Plot IX: Particle Swarm Optimization on Ackley's (Iteration Number vs Gbest x1 value)



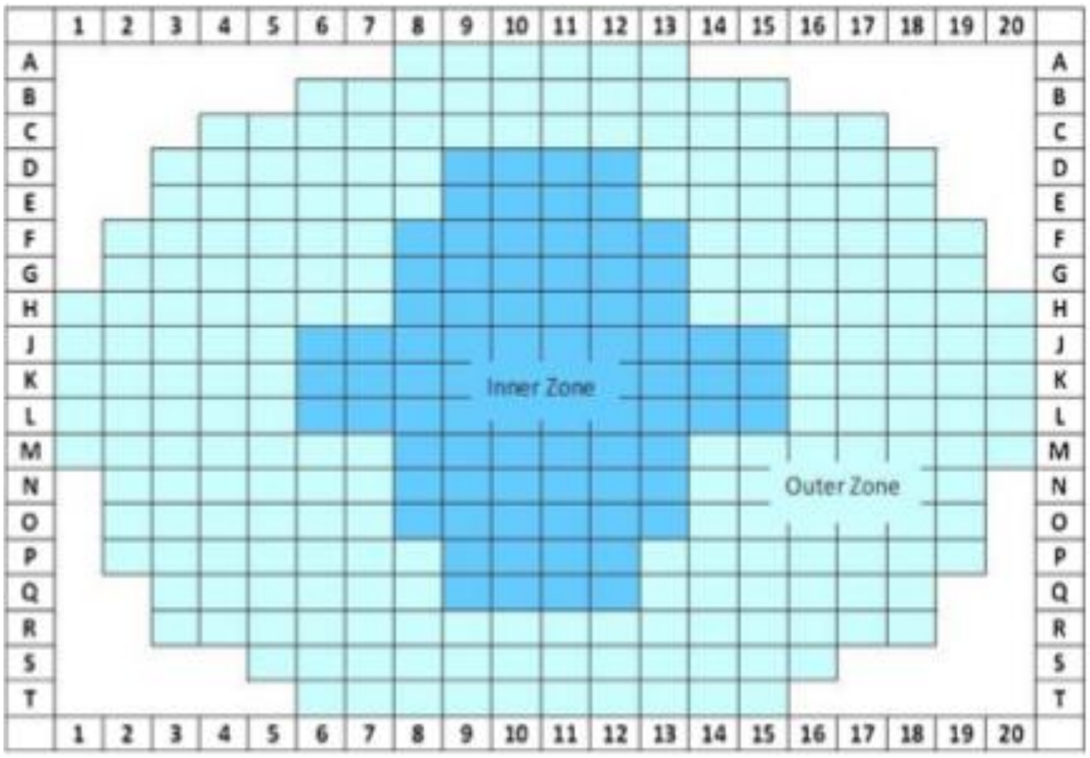
Plot X: Particle Swarm Optimization on Ackley's (Iteration Number vs Gbest x1 value)

3.3 CONCLUSION

We can observe the flow of each algorithm with these plots. SA is not a very effective technique because it is a local search method and we are applying it to find the global minima of Ackley's. PSO gives feasible solution faster cause of being more direct in its approach but GA is better in the long run because it converges slower than PSO but combines exploration and exploitation in an effective manner.

4. THE FUEL BUNDLE BURNUP OPTIMIZATION PROBLEM

4.1 THE PROBLEM



The two burnup zones for the reactor core used for the study

The division of the core into two burnup zones is important for the PHWR. It is seen that without the burnup zones division, the radial neutron flux shape is peaked at the central region of the core and the bundle power at that region will exceed the operating limit at full power operation. The inner zone contains 78 fuel channels and the outer zone contains 228 fuel channels. The present study considers the discharge burnup of inner zone is in the range between 8500 and 11,000 MWd/t and that of outer zone is in the range between 4000 and 6500 MWd/t. The typical discharge burnups for a 220 MWe PHWR are around 10,200 MWd/t and 5500 MWd/t for inner and outer zones respectively. By keeping the discharge burnup of inner zone being higher, the fissile inventory is kept relatively lower at the inner zone as compared to the outer zone channels. This leads to flattening of the neutron flux at the inner region and with that the maximum bundle power (MBP) is within operating bundle power Limit. The aim of the optimization carried out in the present study is to find optimum reference discharge burnup values for the inner and outer zones, in order to obtain maximum average discharge burnup for the total core (BU_{ave}) while satisfying four constraints to ensure safety even while the reactor operates at maximum level.

The reactor physics based simulation code used for this study is called TAQUIL which is an in-house developed FORTRAN programming language based program, particularly suitable for PHWR equilibrium core simulation studies.

In the past, Genetic Algorithms has been used to optimize this process at IGCAR. This project does a comparison with the previously obtained results and results obtained from using PSO with Von Neumann Topology.

4.2 RESULTS

	GA	**PSO**
Average Burnup Value (MWd/t) (taken over 20 runs)	6816.165	6761.632

4.3 CONCLUSION

Genetic Algorithms give us better results than modified Particle Swarm Optimization cause of better exploration and slower convergence.

5. CONCLUSION

- Simulated Annealing is a good technique when searching a small search space as it is a local search based algorithm.

- Particle Swarm Optimization is a more direct method as compared to Genetic Algorithms. It directs its particles straight towards the best solution. This often leads to under exploration.

- Genetic Algorithms move slower than Particle Swarm Optimization but explore the space much better, in turn producing better results in the long run.

- Modified Particle Swarm Optimization is drastically better than the simple version.

- We need to look into ways to combine two or more techniques (especially a global search and a local search one) to try and get better results than when executing them individually.

6. FUTURE SCOPE

This report will help guide future researches into studying the flow of the three algorithms, specifically when applied on a multi-modal function. It also sets a good base for further studies in this subject. An interesting way to take this forward could be to explore combining two or more algorithms together to obtain a better result than what is obtained individually.

7. APPENDIX

1. Genetic Algorithms - Code

```
//header files
#include<stdio.h> //for input/output functionality
#include<math.h> //to use M_Pi constant and some math functions
#include<stdlib.h> //to use rand() and srand() functions for generating random numbers and seeding
#include<time.h> // to calculate time of execution


//initialization of global variables
#define  pop_size 50 //population size
#define maxgen 100 //maximum number of generations
#define pc 0.8 // crossover probability
#define pm 0.02 //mutation probability


double ff(double y1,double y2) //function to evaluate the fitness function value (here it is Ackley's function)
{
        double v1 = exp(pow(((powf(y1,2)+powf(y2,2))/2),0.5)*(-0.2))*(-20);
        double v2 = (-1)*(exp((0.5)*(cos(2*M_PI*y1)+cos(2*M_PI*y2))));
        double v = v1+v2+20+exp(1);
        return (v);
}


struct chr //structure of a chromosome with its x1,x2 values as well as its evaluated fitness function value
{
        int x1[17];
        int x2[17];
        double ffv;

}newpop[pop_size],curpop[pop_size]; // we declare two arrays of chr type to store current and previous generations


//FILE *fp; // declare a file -- will be used for Output of results to an external file


/* We take the required precision to be 4 places after the decimal point.
```

```
  For this we calculate the number of bits m1 and m2 that we need to represent x1,x2 for each chromosome
*/

const int  m1= 17; //number of bits to represent x1,x2 each

double randnum_ab(int a, int b) // function to generate random numbers between a and b passed to the function
{
        double t;
        t = ((double) rand() / (double) RAND_MAX);
        t = (b-a)*t;
        t = t+a;
        return (t);
}

double crv(int y1[]) //to calculate real value between -5 to 5 from binary coding
{
        double fvalue=0;
        int i = 0;
        int pwr = m1-1;
        for(i=0;i<m1;i++)
        {
                fvalue = fvalue + (pow(2,pwr)*y1[i]);
                pwr = pwr-1;
        }
        double rvalue;
        rvalue = -5 + (fvalue*(10/(pow(2,17)-1)));
        return (rvalue);
}

void gen_ini_pop() //function to generate initial population
{
        srand(time(NULL)); //seeding ran() function with current time
        int i,j;
        for(i=0;i<pop_size;i++)
        {
                for(j=0;j<m1;j++)
                {
                        curpop[i].x1[j] = rand()%2; //generating x1 value for the ith solution
                }

                for(j=0;j<m1;j++)
                {
                        curpop[i].x2[j] = rand()%2; //generating x2 value for the ith solution
                }
                double a = crv(curpop[i].x1); //calculating real values of x1 part of the ith solution
                double b = crv(curpop[i].x2); //calculating real values of x2 part of the ith solution
                curpop[i].ffv = ff(a,b); //calculating fitness function value from the real values of x1 and x2
        }

}

double fsumof_ff(struct chr pop[pop_size]) //calculates the sum of fitness function values of the whole population
{
        int i=0;
        double sum=0.0;
        for(;i<pop_size;i++)
        {
                sum += pop[i].ffv;
        }
        return (sum);
}

void selection() // function to select on the basis of 'survival fo the fittest', where a solution with a better fitness function
value will have higher probability to be a part of the next generation
{
        double p[pop_size] = {0}; //empty array to store individual probabilities of each solution
        double q[pop_size] = {0}; // empty array to store cumulative probability till each solution
        double sumof_ff = fsumof_ff(curpop); //sum of fitness function values of all solutions
        double r;
        double sum = 0.0;
        int i,j,l;
        for(i=0;i<pop_size;i++) // loop to calculate individual probabilities of each solution
        {
                p[i] =1/(curpop[i].ffv / sumof_ff) ;
                sum = sum + p[i];
        }

        for(i = 0; i<pop_size; i++) // loop to calculate cumulative probability till each solution
        {
                p[i] = (1/(curpop[i].ffv / sumof_ff))/sum ;
                if(i==0)
                q[0]=p[0];
                else
                q[i] = q[i-1] + p[i] ;
        }
        for(i=0; i<pop_size;i++)
        {
                r = ((double)rand()/(double)RAND_MAX);          // generate random number to compare with
cumulative probabilities so as to decide which solution should be carried to the next generation
                for(j=0;j<pop_size;j++)
                {
                        if(r<q[j])
                        {
                                for(l = 0;l<m1;l++)
                                {
                                        newpop[i].x1[l] = curpop[j].x1[l];
                                        newpop[i].x2[l] = curpop[j].x2[l];
                                }
                                newpop[i].ffv = curpop[j].ffv;
                                break;
                        }
                }
        }
        for(i=0;i<pop_size;i++) // loop to make current generation equal new solution (this is required cause we will
make changes to the new generation in crossover and mutation functions)
        {
                for(j=0;j<m1;j++)
                {
                        curpop[i].x1[j] = newpop[i].x1[j];
                        curpop[i].x2[j] = newpop[i].x2[j];
                }
```

```
                    curpop[i].ffv = newpop[i].ffv;
        }
}


void printpop(struct chr pop[pop_size],int counter) // function to print statistics about a generation after 'counter'
generations
{
        double sum = 0.0;
        double lowest = 100.0;
        int i,j;
        for(i = 0; i<pop_size;i++) // loop to calculate the lowest fitness function value of the 'counter' generation
        {
                sum = sum + pop[i].ffv;
                if(pop[i].ffv < lowest)
                {
                        lowest = pop[i].ffv;
                        j=i;
                }
        }
        printf("The average fitness function value is: %f. \n", (sum/pop_size));
        printf("The least fitness function value of this population is: %f and it is the %d chromosome.\n \n",
lowest,(j+1));
}


//declaration of two variables of type chr to be used in performing crossover
struct chr xover1;
struct chr xover2;


void crossover() //function to perform crossover between two solutions
{
        double r,s;
        int i,t,l,j,k;
        int flag; // variable to keep track of how many solutions have been chosen for crossover
        for(i = 0;i<pop_size;)
        {
                flag = 1;
                r = ((double)rand()/(double)RAND_MAX); //random number generation to choose the first solution
for crossover
```

```
                    if(r<pc)
                    {
                              l=i;
                              for(k=0;k<m1;k++) //loop to save the first solution for crossover in a temporary variable
xover1
                              {
                                        xover1.x1[k] = curpop[i].x1[k];
                                        xover1.x2[k] = curpop[i].x2[k];
                              }
                              xover1.ffv = curpop[i].ffv;


                              for(j=i+1;j<pop_size;j++)
                              {
                                        flag = 0;
                                        s = ((double)rand()/(double)RAND_MAX); //random number generation to
choose the second solution for crossover
                                        if(s<pc)
                                        {
                                                  flag = 1;
                                                  for(k=0;k<m1;k++) //loop to save the second solution for crossover in a
temporary variable xover2
                                   {
                                   xover2.x1[k] = curpop[j].x1[k];
                                        xover2.x2[k] = curpop[j].x2[k];
                                        }
                                        xover2.ffv = curpop[j].ffv;
                                                  t = randnum_ab(1,33); // random number generation between 1 and 33
to choose the point of crossover between the two chosen solutions (single point crossover)
                                                  if(t<17) //check to see if the single point for crossover is in x1 or x2 part
of a solution
                                                  {
                                                            for(k=t;k<m1;k++) //swaping values for crossover
                                                            {
                                                                      xover1.x1[k] = xover1.x1[k] + xover2.x1[k];
                                                                      xover2.x1[k] = xover1.x1[k] - xover2.x1[k];
                                                                      xover1.x1[k] = xover1.x1[k] - xover2.x1[k];
                                                            }
                                                            for(k=0;k<m1;k++) //swaping values for crossover
                                                            {
```

```
                                                       xover1.x2[k] = xover1.x2[k] + xover2.x2[k];
             xover2.x2[k] = xover1.x2[k] - xover2.x2[k];
             xover1.x2[k] = xover1.x2[k] - xover2.x2[k];
                                                 }
                                                 xover1.ffv = ff(crv(xover1.x1),crv(xover1.x2)); //calculate
new fitness function values
             xover2.ffv = ff(crv(xover2.x1),crv(xover2.x2)); //calculate new fitness function values
             }
                                           else //else part of the check
                                           {
                                                 for(k=t-17;k<m1;k++) //swaping values for crossover
                         {
                         xover1.x2[k] = xover1.x2[k] + xover2.x2[k];
                         xover2.x2[k] = xover1.x2[k] - xover2.x2[k];
                      xover1.x2[k] = xover1.x2[k] - xover2.x2[k];
                      }
                                                 xover1.ffv = ff(crv(xover1.x1),crv(xover1.x2)); //calculate
new fitness function values                                          xover2.ffv =
ff(crv(xover2.x1),crv(xover2.x2));
                                           }
                                           for(k=0;k<m1;k++) // store changes in the new population
                                           {
                                                 newpop[i].x1[k]=xover1.x1[k];
                                                 newpop[i].x2[k]=xover1.x2[k];
                                                 newpop[j].x1[k]=xover2.x1[k];
          newpop[j].x2[k]=xover2.x2[k];
                                           }
                                           newpop[i].ffv = xover1.ffv;
                                           newpop[j].ffv = xover2.ffv;


                         }
                         if(flag==1) //checks to see if we do not have a single chosen solution for
crossover left alone
                         {
                                           i=j+1;
                                           break;
                         }
                         else if(flag==0 && j==pop_size-1) //if there is a single solution left for crossover,
we will either keep it or not with equal probability of 0.5
```

```
                                        {
                                                  r = ((double)rand()/(double)RAND_MAX);
                                    if(r>0.5)
                               {
                                        for(k=0;k<m1;k++)
                                        {
                                                  newpop[l].x1[k]=xover1.x1[k];
                                                  newpop[l].x2[k]=xover1.x2[k];
                                        }
                                        newpop[l].ffv = xover1.ffv;
                                                            continue;


                                    }
                                        }
                              }
                    }
                    if(flag==1 && i==j+1) //checks so as to check how many solution underwent crossover and if we had
a solution left alone for crossover
                    {
                              break;
                    }
                    else if (flag==1) //similar checks
                    {
                              i++;
                    }
                    else if(flag == 0 && j==pop_size) //similar checks
                    {
                              break;
                    }
          }
}


void mutation() //function to perform mutation
{
          int i,j;
          double r;
          for(i=0;i<pop_size;i++)
```

```
            {
                        for(j=0;j<m1;j++)
                        {
                                    r = ((double)rand()/(double)RAND_MAX); //random number generation to compare with
mutation probability for x1 part of solution
                                    if(r<pm) //check
                                    {
                                                if(newpop[i].x1[j]==0) //swaping
                                                {
                                                            newpop[i].x1[j]=1;
                                                }
                                                else
                                                {
                                                            newpop[i].x1[j]=0;
                                                }
                                    }
                        }
                        for(j=0;j<m1;j++)
       {
          r = ((double)rand()/(double)RAND_MAX); //random number generation to compare with mutation probability for
x2 part of solution
                                    if(r<pm) //check
          {
            if(newpop[i].x2[j]==0) //swaping
                    {
                                                            newpop[i].x2[j]=1;
                                                }
            else
            {
                                                 newpop[i].x2[j]=0;
                                                }
          }
       }
            }
            for(i=0;i<pop_size;i++) //making current population equal to new population (on which we made changes of
crossover and mutation)
       {
          for(j=0;j<m1;j++)
```

```
        {
          curpop[i].x1[j] = newpop[i].x1[j];
          curpop[i].x2[j] = newpop[i].x2[j];
        }
                          for(j=0;j<m1;j++)
                          {
                                  newpop[i].ffv = ff(crv(newpop[i].x1),crv(newpop[i].x2)); //calculate the new
fitness function value
                  curpop[i].ffv = newpop[i].ffv;
                          }
                  }
}



int main() //main function
{
        int h;
        clock_t start,end; //variables to calculate time of execution
        double time_taken; //variable to store time taken to execute the algorithm
        //fp = fopen("data_gabc_truns.txt","a"); //file for output into an external file
        start = clock(); // start clock
        gen_ini_pop(); //calling function to generate initial population
        selection(); //calling function to perform selection of the first 0th generation
        for(h=1;h<=maxgen;h++) //loop to iterate for maxgen generations
        {
                crossover(); //performing crossover
                mutation(); //performing mutation
                selection(); //performing selection of solutions into the next generation
  }
        end = clock(); //end clock
        time_taken = ((double) end - start); //take difference in time
        time_taken = time_taken / CLOCKS_PER_SEC ;  //divide by a constant to get the time in seconds
        printf("%f\n", time_taken); //print time taken to output screen
        //fprintf(fp,"%f\n",time_taken);
        //fclose(fp); //file close
        return(0); //return of main function
}
```

```
//end of file
```

2. Simulated Annealing - Code

```
//header files
#include<stdio.h> //for input/output functionality
#include<math.h> //to use M_Pi constant and some math functions
#include<stdlib.h> //to use rand() and srand() functions for generating random numbers and seeding
#include<time.h> // to calculate time of execution

#define alpha 0.99
#define ini_temp 500
#define epsilon 0.000001



double ff(double y1,double y2) //function to evaluate the fitness function value (here it is Ackley's function)
{
        double v1 = exp(pow(((powf(y1,2)+powf(y2,2))/2),0.5)*(-0.2))*(-20);
        double v2 = (-1)*(exp((0.5)*(cos(2*M_PI*y1)+cos(2*M_PI*y2))));
        double v = v1+v2+20+exp(1);
        return (v);
}

double randnum_ab(int a, int b) // function to generate random numbers between a and b passed to the function
{
        double t;
        t = ((double) rand() / (double) RAND_MAX);
        t = (b-a)*t;
        t = t+a;
        return (t);
}

//global declaration of variables to store new generated solution in every iteration
```

```
double x1;

double x2;

double ffv;


//FILE *fp; // declare a file -- will be used for Output of results to an external file



void gen_ran_sol() //function to generate random solution in every iteration

{

		x1 = randnum_ab(-5,5);

		x2 = randnum_ab(-5,5);

		ffv = ff(x1,x2);

}


//global declaration of variables to store the accepted solution at a given point of time

double cur_x1;

double cur_x2;

double cur_ffv;


double delta; //variable to store the difference in fitness function value of current and newly generated solution

int iterations = 0; //variable to count the number of iterations


void curr_soln_print(int counter) //function to print the current accepted solution after 'counter' iterations

{

		printf("The solution after %d iteration.",counter);

		printf("\n The value of x1: %f",cur_x1);

		printf("\n The value of x2: %f",cur_x1);

		printf("\n The value of fitness function: %f \n",cur_ffv);

    //fprintf(fp,"%f,%d\n",cur_ffv,counter);


}



int main() //main function

{
```

```
            srand(time(NULL)); //random seed with time
            //fp = fopen("data_sa_truns.txt","a"); //file for output into an external file
            int i;
    double temp = ini_temp; //variable to store initial temperature
    double rand_num; //variable to store random numbers generated during program's run
            clock_t start,end; //variables to calculate time of execution
            double time_taken; //variable to store time taken to execute the algorithm
            start = clock(); // start clock
            //random generation of the first accepted solution
            cur_x1 = randnum_ab(-5,5);
    cur_x2 = randnum_ab(-5,5);
    cur_ffv = ff(cur_x1,cur_x2);
            while( temp > epsilon) //while loop to ensure program runs till temperature is above epsilon
            {
                        iterations++; //to count the number of iterations
                        gen_ran_sol(); //to generate a random temporary solution
                        delta = ffv - cur_ffv; //store the difference in fitness function value
                        if(delta<0) //accept directly if better
                        {
                                    cur_x1 = x1;
                                    cur_x2 = x2;
                                    cur_ffv = ffv;
                        }
                        else
                        {
                                    rand_num = ((double) rand() / (double) RAND_MAX); //generate random number
                                    if(rand_num < exp(-delta/temp)) //accept with probability (which decreases with time)
                                    {
                                                cur_x1 = x1;
            cur_x2 = x2;
            cur_ffv = ffv;
                                    }
                        }
                        temp = temp * alpha; //reduce temperature by alpha factor after each iteration
            }
            end = clock(); //end clock
            time_taken = ((double) end - start); //take difference in time
            time_taken = time_taken / CLOCKS_PER_SEC ;  //divide by a constant to get the time in seconds

            printf("%f\n", time_taken); //print time taken to output screen

            //fclose(fp); //file close

            return(0); //return of main function

}


//end of file
```

3. Particle Swarm Optimization (with good parameters) - Code

```
//header files

#include<stdio.h> //for unput/output functionality

#include<math.h> //to use M_Pi constant and some math functions

#include<stdlib.h> //to use rand() and srand() functions for generating random numbers and seeding

#include<time.h> // to calculate time of execution


#define N 40 //number of particles

#define max_cycles 800 //maximum number of iterations


double C0 = 1; //inertia weight -- dependency in previous velocity

double C1 = 1.8; //factor which accounts for dependency in pbest

double C2 = 1.8; //factor which accounts for dependency in gbest


//FILE *fp; // declare a file -- will be used for Output of results to an external file


double ff(double y1,double y2) //function to evaluate the fitness function value (here it is Ackley's function)

{

            double v1 = exp(pow(((powf(y1,2)+powf(y2,2))/2),0.5)*(-0.2))*(-20);

            double v2 = (-1)*(exp((0.5)*(cos(2*M_PI*y1)+cos(2*M_PI*y2))));

            double v = v1+v2+20+exp(1);

            return (v);

}
```

```
double randnum_ab(int a, int b) // function to generate random numbers between a and b passed to the function
{
        double t;
        t = ((double) rand() / (double) RAND_MAX);
        t = (b-a)*t;
        t = t+a;
        return (t);
}

struct particle //particle structure
{
        double x1; //stores x1 value
        double x2; //stores x2 value
        double v1; //stores v1 value -- velocity corresponding to x1
        double v2; //stores v2 value -- velocity corresponding to x2
        double pbest_x1; //stores its pbest x1 value
        double pbest_x2; //stores its pbest x2 value
        double ffv; //stores its current fitness function value

}swarm[N]; //initialize a swarm of size N

void gen_ini_swarm() //function to generate initial swarm with appropriate starting values
{
        srand(time(NULL)); //random seed with time
        int i;
        for(i=0;i<N;i++)
        {
                swarm[i].x1 = randnum_ab(-5.0,5.0);
                swarm[i].x2 = randnum_ab(-5.0,5.0);
                swarm[i].pbest_x1 = swarm[i].x1;
                swarm[i].pbest_x2 = swarm[i].x2;
                swarm[i].ffv = ff(swarm[i].x1,swarm[i].x2);
                swarm[i].v1 = 0.0;
                swarm[i].v2 = 0.0;
        }
}
```

```
//global variables to store gbest x1 and x2 values for the whole swarm
double gbest_x1;
double gbest_x2;

int best_counter; //variable to store the index of the particle with the best fitness function value

void ini_return_best_index() //function which calculated the index of the particle with the best fitness function value
{
	int i,j;
	double lowest = RAND_MAX;
	for(i=0;i<N;i++) //loop to find the particle with the best x1 and x2 values
	{
		if(swarm[i].ffv < lowest)
		{
			j=i;
			lowest = swarm[i].ffv;
		}
	}
	best_counter = j;
	gbest_x1 = swarm[best_counter].x1; //set gbest x1 as the best x1 obtained from the above loop
	gbest_x2 = swarm[best_counter].x2; //set gbest x2 as the best x2 obtained from the above loop
}

void print_swarm() //function to print the swarm
{
	int i;
	for(i=0;i<N;i++)
	{
		printf("%f %f %f %f %f %f %f\n", swarm[i].x1, swarm[i].x2, swarm[i].v1, swarm[i].v2,
swarm[i].pbest_x1, swarm[i].pbest_x2, swarm[i].ffv);
	}
}

void update_swarm() //function to update each particle of the swarm
{
```

```
            int i;

            double r,s; //variables to store random numbers which will be used to update the velocity of every particle

            for(i=0;i<N;i++)

            {

                        r = ((double) rand() / (double) RAND_MAX); //random number generation

                        s = ((double) rand() / (double) RAND_MAX); //random number generation

                        swarm[i].v1 = (C0*swarm[i].v1) + (C1*r*(swarm[i].pbest_x1 - swarm[i].x1)) +
(C2*s*(swarm[best_counter].pbest_x1 - swarm[i].x1)); //update the particle's v1 velocity corresponding to x1

                        //if(swarm[i].v1 >20) //velocity clamping

                        //swarm[i].v1 = 20;

                        swarm[i].x1 = swarm[i].x1 + swarm[i].v1; //update particle's x1 position

                        r = ((double) rand() / (double) RAND_MAX);

      s = ((double) rand() / (double) RAND_MAX);

      swarm[i].v2 = (C0*swarm[i].v2) + (C1*r*(swarm[i].pbest_x2 - swarm[i].x2)) +
(C2*s*(swarm[best_counter].pbest_x2 - swarm[i].x2)); //update the particle's v2 velocity corresponding to x2

      //if(swarm[i].v2 > 20) //velocity clamping

                        //swarm[i].v2 = 20;

                        swarm[i].x2 = swarm[i].x2 + swarm[i].v2; //update particle's x2 position

                        swarm[i].ffv = ff(swarm[i].x1,swarm[i].x2); //calculate new fitness function value for the ith particle

                        if(swarm[i].ffv < ff(swarm[i].pbest_x1,swarm[i].pbest_x2)) //update particle's pbest if required

                        {

                                    swarm[i].pbest_x1 = swarm[i].x1;

                                    swarm[i].pbest_x2 = swarm[i].x2;

                                    if(swarm[i].ffv < ff(gbest_x1,gbest_x2)) //update swarm's gbest if required

                                    {

                                                gbest_x1 = swarm[i].x1;

                                                gbest_x2 = swarm[i].x2;

                                    }

                        }

            }

}


int main()

{

            //fp = fopen("data_pso_truns.txt","a"); //file for output into an external file

            clock_t start,end; //variables to calculate time of execution

            double time_taken; //variable to store time taken to execute the algorithm

            int i;
```

```
        start = clock(); // start clock
        gen_ini_swarm(); //calling function generate initial swarm
        ini_return_best_index(); //calling function to generate the index of the best particle
        for(i=0;i<max_cycles;i++) //loop to iteration the algorithm for max_cycles time
        {
                update_swarm(); //calling the function to update each particle
                if(C0>0.4) //check to reduce the dependency on previous velocity linearly with time
                {
                        C0 = C0 - ((0.6/(max_cycles-1))*i);
                        C1 = C1 - ((0.5/(max_cycles-1))*i);
                        C2 = C2 - ((0.5/(max_cycles-1))*i);
                }
        }
        end = clock(); //end clock
        time_taken = ((double) end - start); //take difference in time
        time_taken = time_taken / CLOCKS_PER_SEC ;  //divide by a constant to get the time in seconds
        printf("%f\n", time_taken); //print time taken to output screen
        //fclose(fp); //file close
        return(0); //return of main function
}
//end of file
```

4. Particle Swarm Optimization for Fuel Bundle Burnup Optimization Problem (with good parameters and Von Neumann Topology) - Code

```
//header files
#include<stdio.h> //for unput/output functionality
#include<math.h> //to use M_Pi constant and some math functions
#include<stdlib.h> //to use rand() and srand() functions for generating random numbers and seeding
#include<time.h> // to calculate time of execution
#include<string.h> //to read and write into the FORTRAN code(TAQUIL)
#include <float.h> //to use DBL_MAX constant
#include "fileread.h" //to include our custom header file
```

```
#define N 40 //number of particles

#define max_cycles 800 //maximum number of iterations

#define NONEIGH 4 //number of neighbours of each particle (one in each direction)


double C0 = 1; //inertia weight -- dependency in previous velocity

double C1 = 1.8; //factor which accounts for dependency in pbest

double C2 = 1.8; //factor which accounts for dependency in gbest


//FILE *fp; // declare a file -- will be used for Output of results to an external file


struct particle //particle structure

{

        double x1; //stores x1 value

        double x2; //stores x2 value

        double v1; //stores v1 value -- velocity corresponding to x1

        double v2; //stores v2 value -- velocity corresponding to x2

        double pbest_x1; //stores its pbest x1 value

        double pbest_x2; //stores its pbest x2 value

        double ffv; //stores its current fitness function value

        double gbest_x1; //stores gbest x1 value

        double gbest_x2; //stores gbest x2 value

        double mbp; //stores the mbp value

        double mcp; //stores the mcp value

        double keff; //stores the keff value

        double avburnup; //stores the avburnup value

        double pbest_mbp; //stores the pbest mbp value

        double pbest_mcp; //stores the pbest mcp value

        double pbest_keff; //stores the pbest keff value

        double pbest_avburnup; //stores the pbest avburnup value

        int near_coord[NONEIGH]; //stores the coordinates of nearest four particles (one in each direction)


}swarm[N]; //initialize a swarm of size N


void calc_ffv(int i) //function to evaluate the fitness function value

{

        double penalty1, penalty2,penalty3;
```

```
		penalty1=10*swarm[i].mbp-4300; //calculate penalty 1

		penalty2=1000*swarm[i].mcp-3200; //calculate penalty 2

		penalty3=100000 -100000*swarm[i].keff; //calculate penalty 3

		swarm[i].ffv=swarm[i].avburnup-penalty1-penalty2-penalty3; //calculate fitness function

		if((swarm[i].mbp <= 430) && (swarm[i].mcp <= 3.2) && (swarm[i].keff >=1.0005) && (swarm[i].avburnup
>= 6700)) //check to increase the fitness function if the solution is feasible

		{

			swarm[i].ffv=swarm[i].ffv+400;

		}

}


double calc_pbest_ffv(int i) //function to evaluate the fitness function value of pbest

{

		double penalty1, penalty2,penalty3,tmpffv;

		penalty1=10*swarm[i].pbest_mbp-4300; //calculate penalty 1 for pbest mbp

		penalty2=1000*swarm[i].pbest_mcp-3200; //calculate penalty 2 for pbest mcp

		penalty3=100000 -100000*swarm[i].pbest_keff; //calculate penalty 3 for pbest keff

		tmpffv=swarm[i].pbest_avburnup-penalty1-penalty2-penalty3; //calculate the fitness function corresponding to
pbest values

		if((swarm[i].pbest_mbp <= 430) && (swarm[i].pbest_mcp <= 3.2) && (swarm[i].pbest_keff >=1.0005) &&
(swarm[i].pbest_avburnup >= 6700)) //check to increase the fitness function if the solution is feasible

		{

			tmpffv=tmpffv+400;

		}

		return (tmpffv); //return the fitness function value corresponding to pbest values

}


double randnum_ab(double a, double b) // function to generate random numbers between a and b passed to the function

{



		double t;

		t = ((double) rand() / (double) RAND_MAX);

		t = (b-a)*t;

		t = t+a;

		return (t);

}


double calc_dist(double x1,double x2) //function to calculate euclidean distance between x1 and x2 (passed to the
function)
{
        double t = x1-x2;
        return (t*t);
}

void calc_near_east(int i) //function calculate the nearest point to a particle towards its east
{
        int j,lowest_index;
        double cur_dist,temp_dist;
        lowest_index = i;
        cur_dist = DBL_MAX;
        for(j=0;j<N;j++)
        {
                if(swarm[j].x1 < swarm[i].x1)
                {
                        temp_dist = calc_dist(swarm[i].x1,swarm[j].x1);
                        if(temp_dist < cur_dist)
                        {
                                cur_dist = temp_dist;
                                lowest_index = j;
                        }
                }
        }
        //update the corresponding near_coord arrays
        swarm[i].near_coord[3] = lowest_index;
        swarm[lowest_index].near_coord[1] = i;
}

void calc_near_west(int i) //function calculate the nearest point to a particle towards its west
{
        int j,lowest_index;
        double cur_dist,temp_dist;
        lowest_index = i;
        cur_dist = DBL_MAX;
        for(j=0;j<N;j++)
        {
                if(swarm[j].x1 > swarm[i].x1)
                {
                        temp_dist = calc_dist(swarm[i].x1,swarm[j].x1);
                        if(temp_dist < cur_dist)
                        {
                                cur_dist = temp_dist;
                                lowest_index = j;
                        }
                }
        }
        //update the corresponding near_coord arrays
        swarm[i].near_coord[1] = lowest_index;
        swarm[lowest_index].near_coord[3] = i;
}

void calc_near_north(int i) //function calculate the nearest point to a particle towards its north
{
        int j,lowest_index;
        double cur_dist,temp_dist;
        lowest_index = i;
        cur_dist = DBL_MAX;
        for(j=0;j<N;j++)
        {
                if(swarm[j].x2 > swarm[i].x2)
                {
                        temp_dist = calc_dist(swarm[i].x2,swarm[j].x2);
                        if(temp_dist < cur_dist)
                        {
                                cur_dist = temp_dist;
                                lowest_index = j;
                        }
                }
        }
        //update the corresponding near_coord arrays
        swarm[i].near_coord[0] = lowest_index;
        swarm[lowest_index].near_coord[2] = i;
}
```

```
void calc_near_south(int i) //function calculate the nearest point to a particle towards its south
{
        int j,lowest_index;
        double cur_dist,temp_dist;
        lowest_index = i;
        cur_dist = DBL_MAX;
        for(j=0;j<N;j++)
        {
                if(swarm[j].x2 < swarm[i].x2)
                {
                        temp_dist = calc_dist(swarm[i].x2,swarm[j].x2);
                        if(temp_dist < cur_dist)
                        {
                                cur_dist = temp_dist;
                                lowest_index = j;
                        }
                }
        }
        //update the corresponding near_coord arrays
        swarm[i].near_coord[2] = lowest_index;
        swarm[lowest_index].near_coord[0] = i;
}

void calc_near_allpts() //function to calculate nearest four points for all particles
{
        int i;
        for(i=0;i<N;i++) //loops to check ith near_coord array to avoid unnecessary computation of nearest points
        {
                if(swarm[i].near_coord[0] == -1)
                {
                        calc_near_north(i);
                }
                if(swarm[i].near_coord[1] == -1)
                {
                        calc_near_west(i);
                }
                if(swarm[i].near_coord[2] == -1)
                {
                        calc_near_south(i);
                }
                if(swarm[i].near_coord[3] == -1)
                {
                        calc_near_east(i);
                }
        }
}


void gen_fourvar_values(int  i) //function to interact with the FORTRAN coce which performs the reactor phyics calculations and gives us back four important values which are used to calculate fitness function value of the ith particle
{
        double *outputvals;
        char taquilcommand[100];
        strcpy(taquilcommand,TAQUILCMD);
  modifyinputfile(swarm[i].x2,swarm[i].x1); //we give the FORTRAN code (TAQUIL) inner and outer burnup values as input
  if(system(taquilcommand)== -1)
  {
    printf("Error running taquil\n");
    exit(-1);
  }
        outputvals=readoutputfile();
        //we store the four values in corresponding variables of the ith particle
  swarm[i].mbp=*(outputvals);
  swarm[i].mcp=*(outputvals+1);
  swarm[i].keff=*(outputvals+2);
  swarm[i].avburnup=*(outputvals+3);
}


void update_fourvar(int k) //function to update pbest's of each kth particle to the current values of the same four variables
{
        swarm[k].pbest_avburnup = swarm[k].avburnup;
        swarm[k].pbest_mcp = swarm[k].mcp;
        swarm[k].pbest_mbp = swarm[k].mbp;
		swarm[k].pbest_keff = swarm[k].keff;
}

void gen_ini_swarm() //function to generate initial swarm with appropriate starting values
{
		srand(time(NULL)); //random seed with time
		int i,j;
		for(i=0;i<N;i++)
		{
			swarm[i].x1 = randnum_ab(INNERMIN,INNERMAX); //random values
			swarm[i].x2 = randnum_ab(OUTERMIN,OUTERMAX); //random values
			swarm[i].pbest_x1 = swarm[i].x1;
			swarm[i].pbest_x2 = swarm[i].x2;
			swarm[i].gbest_x1 = swarm[i].x1;
			swarm[i].gbest_x2 = swarm[i].x2;
			swarm[i].v1 = 0.0; //zero velocity
			swarm[i].v2 = 0.0; //zero velocity
			gen_fourvar_values(i); //calling function to generate four important values from the FORTRAN code
(TAQUIL)
			update_fourvar(i); //update the four variables
			calc_ffv(i); //calculate fitness function value
			for(j=0;j<NONEIGH;j++)
			{
				swarm[i].near_coord[j] = -1; // put all near_coord values to -1
			}
		}
		calc_near_allpts(); //function to calculate nearest four points for all particles
}

void calc_gbest_allpts() //function to calculate gbest for each particle
{
		int i,j;
		double tmp_x1,tmp_x2;
		for(i=0;i<N;i++)
		{
			tmp_x1 = swarm[i].pbest_x1;
			tmp_x2 = swarm[i].pbest_x2;
```

```
                    for(j=0;j<NONEIGH;j++)
                    {
                              if( calc_pbest_ffv(i) < calc_pbest_ffv(swarm[i].near_coord[j]))
                              {
                                        tmp_x1 = swarm[swarm[i].near_coord[j]].pbest_x1;
                                        tmp_x2 = swarm[swarm[i].near_coord[j]].pbest_x2;
                              }
                    }
                    swarm[i].gbest_x1 = tmp_x1;
                    swarm[i].gbest_x2 = tmp_x2;
          }
}


void print_swarm(int k) //function to print the swarm after kth iteration
{
          int i;
          for(i=0;i<N;i++)
          {
                    if((swarm[i].pbest_avburnup > 6700) && (swarm[i].pbest_keff > 1.0005) && (swarm[i].pbest_mbp <
430) && (swarm[i].pbest_mcp < 3.2))
                    {
                    printf("\n %d Generation \n \n",k);
                    //fprintf(fp,"\n %d %f %f %f %f %f %f %f\n", i, swarm[i].pbest_x1, swarm[i].pbest_x2,
swarm[i].pbest_avburnup,swarm[i].pbest_mbp, swarm[i].pbest_mcp,swarm[i].pbest_keff, calc_pbest_ffv(i));
                    printf("%d %f %f %f %f %f %f %f\n", i, swarm[i].pbest_x1, swarm[i].pbest_x2,
swarm[i].pbest_avburnup,swarm[i].pbest_mbp, swarm[i].pbest_mcp,swarm[i].pbest_keff, calc_pbest_ffv(i));
                    }
          }
}


void update_swarm() //function to update each particle of the swarm
{
          int i;
          double r,s; //variables to store random numbers which will be used to update the velocity of every particle
          for(i=0;i<N;i++)
          {
                    calc_gbest_allpts();
                    r = ((double) rand() / (double) RAND_MAX); //random number generation
```

```
                s = ((double) rand() / (double) RAND_MAX); //random number generation
                swarm[i].v1 = (C0*swarm[i].v1) + (C1*r*(swarm[i].pbest_x1 - swarm[i].x1)) +
(C2*s*(swarm[i].gbest_x1 - swarm[i].x1)); //update the particle's v1 velocity corresponding to x1
                //if(swarm[i].v1 > 10 ) //velocity clamping
                //swarm[i].v1 = 10;
                swarm[i].x1 = swarm[i].x1 + swarm[i].v1; //update particle's x1 position
                r = ((double) rand() / (double) RAND_MAX);
        s = ((double) rand() / (double) RAND_MAX);
        swarm[i].v2 = (C0*swarm[i].v2) + (C1*r*(swarm[i].pbest_x2 - swarm[i].x2)) + (C2*s*(swarm[i].gbest_x2 -
swarm[i].x2)); //update the particle's v2 velocity corresponding to x2
        //if(swarm[i].v2 > 10) //velocity clamping
                //swarm[i].v2 = 10;
                swarm[i].x2 = swarm[i].x2 + swarm[i].v2; //update particle's x2 position
                gen_fourvar_values(i); //function to recalculate four important values for ith particle after updation
                calc_ffv(i); //function to recalculate the fitness function value
                if(swarm[i].ffv > calc_pbest_ffv(i)) //update the pbest of ith particle if required
                {
                        swarm[i].pbest_x1 = swarm[i].x1;
                        swarm[i].pbest_x2 = swarm[i].x2;
                        update_fourvar(i); //update the four variables
                }
        }
}


int main() //main function
{
        //fp = fopen("data_pso_vntopo_taq_truns.txt","a"); //file for output into an external file
        clock_t start,end; //variables to calculate time of execution
        double time_taken; //variable to store time taken to execute the algorithm
        int i;
        start = clock(); // start clock
        gen_ini_swarm(); //calling function generate initial swarm
        for(i=0;i<max_cycles;i++) //loop to iteration the algorithm for max_cycles time
        {
                update_swarm(); //calling the function to update each particle
                if(C0>0.4) //check to reduce the dependency on previous velocity linearly with time
                {
                        C0 = C0 - ((0.6/(max_cycles-1))*i);
                C1 = C1 - ((0.5/(max_cycles-1))*i);

                C2 = C2 - ((0.5/(max_cycles-1))*i);

            }

        }

        print_swarm(--i); //function to print the swarm after max_cycles iterations

        end = clock(); //end clock

        time_taken = ((double) end - start); //take difference in time

        time_taken = time_taken / CLOCKS_PER_SEC ;  //divide by a constant to get the time in seconds

        printf("%f\n", time_taken); //print time taken to output screen

        //fclose(fp); //file close

        return(0); //return of main function

}

//end of file
```

8. REFERENCES

1. Goldberg, David (1989). Genetic Algorithms in Search, Optimization and Machine Learning. Reading, MA: Addison-Wesley Professional.
2. Kirkpatrick, S.; Gelatt Jr, C. D.; Vecchi, M. P. (1983). "Optimization by Simulated Annealing". Science 220 (4598): 671–680
3. Shi, Y.; Eberhart, R.C. (1998). "A modified particle swarm optimizer".
4. Taherkhani, M.; Safabakhsh, R. (2016). "A novel stability-based adaptive inertia weight for particle swarm optimization"
5. Shi, Y.; Eberhart, R.C. (1998). "Parameter selection in particle swarm optimization".
6. Yang, X.S. (2008). Nature-Inspired Metaheuristic Algorithms. Luniver Press.
7. Eberhart, R.C.; Shi, Y. (2000). "Comparing inertia weights and constriction factors in particle swarm optimization".Proceedings of the Congress on Evolutionary Computation. pp. 84–88
8. Van den Bergh, F. (2001). An Analysis of Particle Swarm Optimizers (PhD thesis). University of Pretoria, Faculty of Natural and Agricultural Science.
9. Trelea, I.C. (2003). "The Particle Swarm Optimization Algorithm: convergence analysis and parameter selection". Information Processing Letters 85 (6): 317–325.
10. Pedersen, M.E.H.; Chipperfield, A.J. (2010). "Simplifying particle swarm optimization" (PDF). Applied Soft Computing 10 (2): 618–628
11. Pedersen, M.E.H. (2010). "Good parameters for particle swarm optimization" (PDF). Technical Report HL1001 (Hvass Laboratories).
12. Reynolds, Mark, and Dianhui Wang. AI 2011: Advances in Artificial Intelligence. Springer Berlin Heidelberg, 2011.
13. Zou, Wenping, et al. "Clustering approach based on Von Neumann topology artificial bee colony algorithm." 2011 international conference on data mining (DMIN'11). 2011.
14. Fernandes, Carlos M., et al. "Partially connected topologies for particle swarm." Proceedings of the 15th annual conference companion on Genetic and evolutionary computation. ACM, 2013.
15. Hassan, Rania, et al. "A comparison of particle swarm optimization and the genetic algorithm." Proceedings of the 1st AIAA multidisciplinary design optimization specialist conference. 2005.
16. Davis, Lawrence. "Genetic algorithms and simulated annealing." (1987).