Minimization Problem

PARTICLE SWARM OPTIMIZATION

Konstantinos Tzagkarakis MP141, Eleftherios Trivizakis MP143 | Computational Intelligence | 7 November 2016

Abstract

The project is an attempt to implement the Particle Swarm Optimization algorithm for finding the closet to zero value of f(z, y), minimization, with the most suitable variable pair of z and y, in a given number of calculations.

We examine the effectiveness of this technic under different circumstances, like changing the number of particles in a swarm, changing the number of iterations or even raising the complexity of the given formula.

Showcasing derived data, provide evidence that supports the validity of theoretical models around the ability of the algorithm to give solutions. The best solution of each iteration drives the movement of the swarm to the optimal solution.

We demonstrate the basic weakness of PSO convergence approach, the personal position of the particles can trap the swarm in their personal optimal solution for a number of iterations.

Table of Contents

Introduction

Before explaining the implementation, we will take a brief look into the basic concepts of Particle Swarm Optimization (PSO) algorithm.

The inspiration for this algorithm came from the social psychology research and the observation of nature, as flock of fish or flying birds move around [1].

The basic variant works by having a population of particles, it's called swarm. Each particle is a candidate solution and is moved in a search space according to best global solution [2], also holds a value for current location, velocity and a fitness value. Every iteration leads to a better global solution [3], which attracts the particles of the entire swarm by calculating an updated velocity and then their new position.

This process is repeated until a satisfactory solution will be discovered or by reaching the last permitted iteration. The PSO convergence [4] variation of the algorithm is used for this implementation. Among with regular calculations, it takes into account the personal best position p (double pbest[]) and the swarm's best known position g (double gbest[]), which approaches a local optimal solution, regardless of how swarm behaves at the time.

It's important to highlight that the initialization of the first particles occurs in a limited scope [5] for x and y variable, so they could give us a swarm formation. There is, also, a need to define an endpoint or/and a finite number for calculations to avoid infinite loops.

Implementation and Experimental Methods

Core Implementation

The basic tool that is used to create the application was eclipse IDE with the JDK 1.8 installed and also the libraries JEval [6], for evaluating the given formula(String) with each particle's position and JFreeCharts [7] for plotting the graphical representation of the best fitness value of each iteration.

The implemented algorithm starts with the initialization of the swarm with random particle position and velocity in a limited scope. Swarm size is also user defined. Each particle holds value for its current position, velocity and fitness value

```
public void initializeSwarm() {
    Particle p;
    for(int i=0; i<swarmy; i++) {
        p = new Particle();

        // randomize location inside a space defined in Problem Set
        double[] loc = new double[PROBLEM_DIMENSION];
        loc[0] = ProblemSet.LOC_X_LOW + generator.nextDouble() * (ProblemSet.LOC_X_HIGH - ProblemSet.LOC_X_LOW);
        loc[1] = ProblemSet.LOC_Y_LOW + generator.nextDouble() * (ProblemSet.LOC_Y_HIGH - ProblemSet.LOC_Y_LOW);
        Location location = new Location(loc);

        // randomize velocity in the range defined in Problem Set
        double[] vel = new double[PROBLEM_DIMENSION];
        vel[0] = ProblemSet.VEL_LOW + generator.nextDouble() * (ProblemSet.VEL_HIGH - ProblemSet.VEL_LOW);
        vel[1] = ProblemSet.VEL_LOW + generator.nextDouble() * (ProblemSet.VEL_HIGH - ProblemSet.VEL_LOW);
        Velocity velocity = new Velocity(vel);

        p.setLocation(location);
        p.setVelocity(velocity);
        swarm.add(p);
    }
}
```

Source Code 1 – initialization of swarm

The fitness value is the last step for initialization, where the location of each particle is tested against the given mathematical formula f(x,y). Java does not support evaluation using from string value, so we use JEval advanced library for adding functional expression parsing and evaluation to the application. It's worth to mention that the main problem with this library is that it does not support scientific notation, which is the standard

```
private static double getEvalFuncDouble(double x, double y, String func) {
    double result=0;
    Evaluator evalEngine = new Evaluator();

    evalEngine.putVariable("z", String.valueOf(new BigDecimal(x)));
    evalEngine.putVariable("y", String.valueOf(new BigDecimal(y)));
    try {
        result = evalEngine.getNumberResult(func);
    } catch (NumberFormatException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (EvaluationException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    return result;
```

Source Code 2 – evaluation (using JEval engine) of the user defined formula representation of very small double values for Java. The solution which is introduced, is to replace double with BigDecimal value type for the x, y with minor lose of precision but unscaled notation.

The method getMinPos(double[]) return the best fitness value of the swarm, which helps us determine the gBestLocation. That's the best global particle position, in our case determined by the lowest fitness value, and is used for calculating new velocities. The new velocities "moves" the swarm to better solutions.

```
int bestParticleIndex = PSOUtility.getMinPos(fitnessValueList);
if(t == 0 || fitnessValueList[bestParticleIndex] < gBest) {
    gBest = fitnessValueList[bestParticleIndex];
    gBestLocation = swarm.get(bestParticleIndex).getLocation();
}
```

Source Code 3 – finding the best particle in the swarm

Also the particle's best location is need, so we compare its current location with old Vector<Location> pBestLocation.

```
for(int i=0; i<swarmy; i++) {
    if(fitnessValueList[i] < pBest[i]) {
        pBest[i] = fitnessValueList[i];
        pBestLocation.set(i, swarm.get(i).getLocation());
    }
}
```

Source Code 4 – finding the best personal (for each particle) location

Inertia weight [8] is introduced for linearly decrease over generation to adjust the local and global search ability. It is calculated in each iteration.

```
w = W_UPPERBOUND - (((double) t) / iterationsMax) * (W_UPPERBOUND - W_LOWERBOUND);
```

Source Code 5 – inertia weight

The ultimate goal for all the above, essentially, is to combine them together into velocity along with current location of the particle and a random generated quantity. Velocity, as

```
newVel[0] = (w * p.getVelocity().getPos()[0]) +
            (r1 * C1) * (pBestLocation.get(i).getLoc()[0] - p.getLocation().getLoc()[0]) +
            (r2 * C2) * (gBestLocation.getLoc()[0] - p.getLocation().getLoc()[0]);
newVel[1] = (w * p.getVelocity().getPos()[1]) +
            (r1 * C1) * (pBestLocation.get(i).getLoc()[1] - p.getLocation().getLoc()[1]) +
            (r2 * C2) * (gBestLocation.getLoc()[1] - p.getLocation().getLoc()[1]);
```

Source Code 6 – calculating new velocity

the name suggests, is the force that drives swarms into optimal solutions. The new current position is the sum of velocity and (previous) position.

```
newLoc[0] = p.getLocation().getLoc()[0] + newVel[0];
newLoc[1] = p.getLocation().getLoc()[1] + newVel[1];
```

Source Code 7- update current position

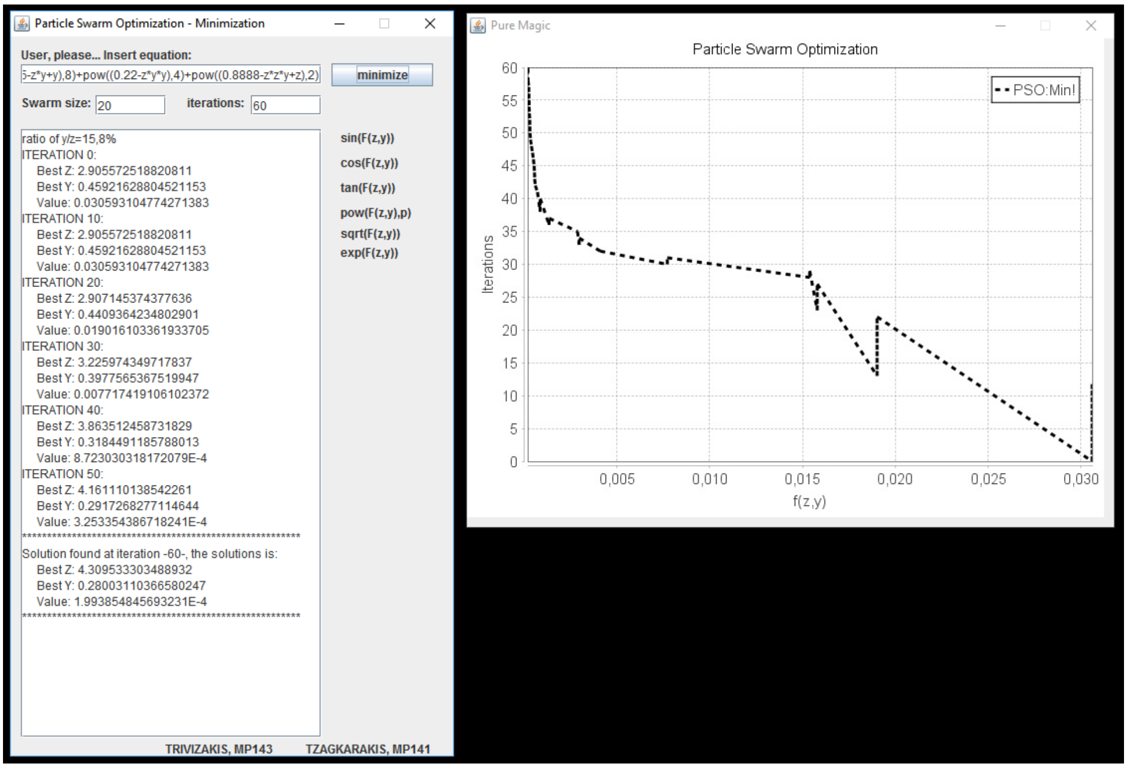
As a final touch for the user interface, using open source code like the JFreeChart library, the application can plot the fitness table of f(x,y), which consists from the best solutions of each iterations.

```
private void plotValues() {
    double[] listFit = new double[finalValues.size()];
    double[] listIt = new double[iterationsNumber.size()];

    for (int i = 0; i < listFit.length; i++) {
        listFit[i] = finalValues.get(i).doubleValue();
        listIt[i] = iterationsNumber.get(i).doubleValue();
    }

    finalValues.clear();
    iterationsNumber.clear();

    MatlabChart fig = new MatlabChart();
    fig.plot(listFit, listIt , ":k", 3.0f, "PSO:Min!");
    fig.RenderPlot();
    fig.title("Particle Swarm Optimization");
    double xlim1 = listFit[0];
    double xlim2 = listFit[listFit.length-1];
    fig.xlim(xlim2, xlim1);
    fig.ylim(0, iterationsMax);
    fig.xlabel("f(z,y)");
    fig.ylabel("Iterations");
    fig.grid("on","on");
    fig.legend("northeast");
    fig.font("Helvetica",15);
    fig.saveas("MyPlot.jpeg",640,480);
    plotFrame();
}
```

Source Code 8 – plotting the fitness values of each iteration for f(x,y)

Also, the main user interface [Capture 1] consists from a text field for the formula, a text field for swarm size and a text field for the number of iterations, as inputs. The button starts the calculation process of the PSO algorithm.

Two components show the output. A non-editable text area prints the numeric data of every 10 iterations, the z, y and the best value of f(z,y) and finally the end the best found solution. The last component if a window frame that projects the plotted values of every iteration, like the matlab's plot would do it.



Capture 1 – left: main gui with user input/output, *right:* a plot for the best fitness of each iteration

Experimental part

In order to verify if that the implementation meets the requirements of actually solving any minimization problem we gather the data from the output of the application.

As a first stage of the experiment, we run the application with three specifications: swarm size, number of iterations and the ratio of y/x. The last specification was chosen mainly because the initial and the next particles of every attempt are always random (in a bounded initial space) but there has to be a common relation among all x and y pairs, so the result can be comparable to its other.

In each and every next stage we keep two of these specification constant, always the ratio and either the swarm size or the number of iterations. As we can see in the result section of this report, we group the results in a way that a conclusion can be drawn.

The last stage will be the comparison among the data and their graphical representation.

The main focus for observation will be how the different swarm sizes can impact at the iterations, what's the amount of iterations that are relevant for a viable or suitable solution, also the possibility of discovering any sort of weakness for this process.

Results

We examined two mathematical formulae:

i. f(z,y) = pow((0.9455-z*y+y),8)+pow((0.22-z*y*y),4)+pow((0.8888-z*z*y+z),2)
ii. f(z,y) = pow((0.9455-z*y+y),8)+pow((1.22-z*y*y),4)+pow((0.8888-z*y+exp(z)),2)

The data are shown briefly in this section and a full review will follow next. The first formula is an easy case for PSO algorithm. There are no complicated calculations, in contrary with the second one in which the exponential (exp(z)) variable makes it near impossible to reach anywhere close to 0. It is crucial thought, to understand the trend of values from first iteration to the final solution and not the results isolated.

Table Data

iterations \ swarm size	20	40	60
20	7,7E-03	4,0E-04	4,0E-05
50	6,0E-04	1,0E-05	1,8E-06
80	5,5E-04	1,4E-05	2,3E-06
120	8,0E-06	1,0E-07	7,4E-09

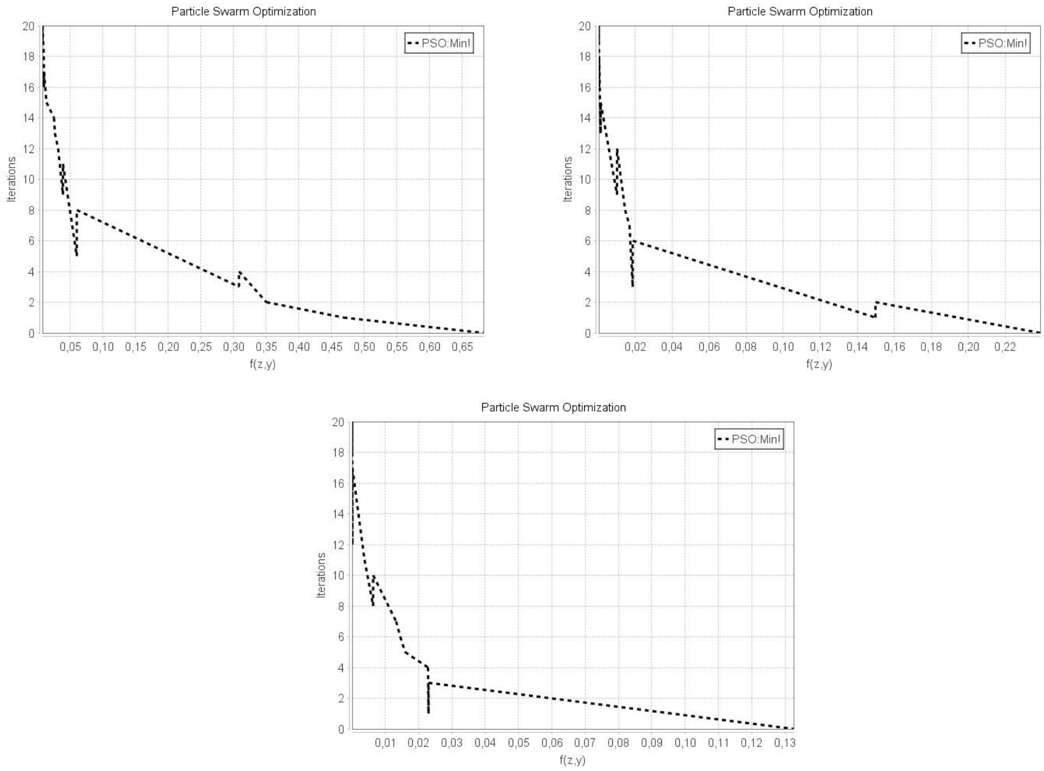
Table 1 – i. data from the execution of application

iterations \ swarm size	20	40	60
20	2,5	2,41	2,37
50	2,36	2,26	2,25
80	2,25	2,227	2,223
120	2,38	2,21	2,217

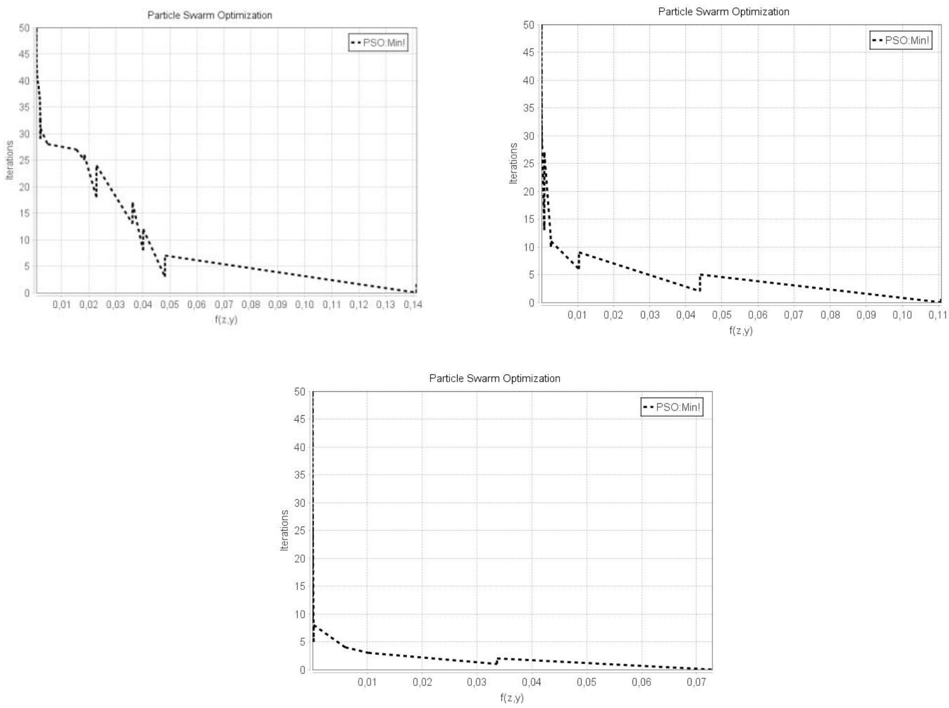
Table 2 – ii. data from the execution of application

Graphical representation

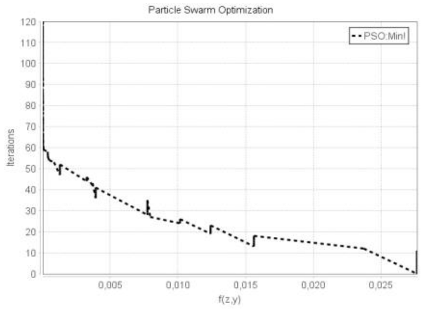
This graphs are very similar to how plotting would have been showed if matlab would plotted them. So it is essential to understand that the scale of each graph defers. To review it properly we have to take into account the values from the above tables and the values from the xx' axon of the graph itself.

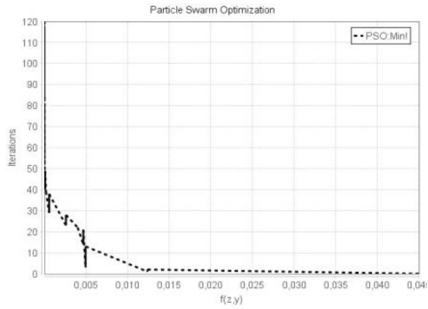


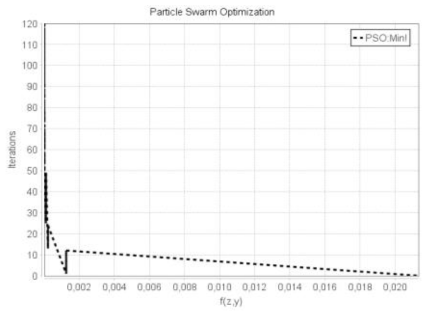
Graphs 1 – i. iterations: 20 / swarm size: a)20, b)40, c)60



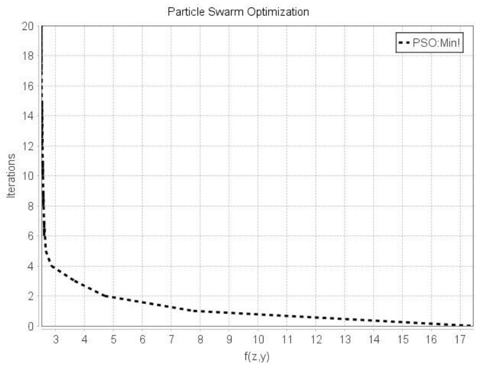
Graphs 2 – i. iterations: 50 / swarm size: a)20, b)40, c)60

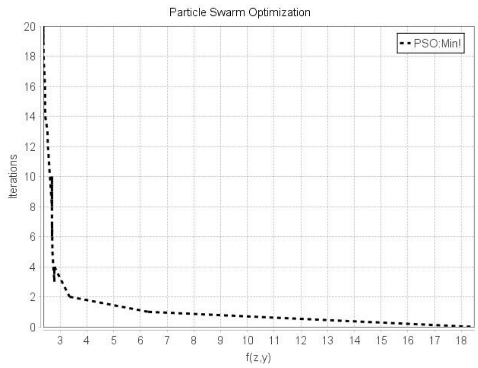


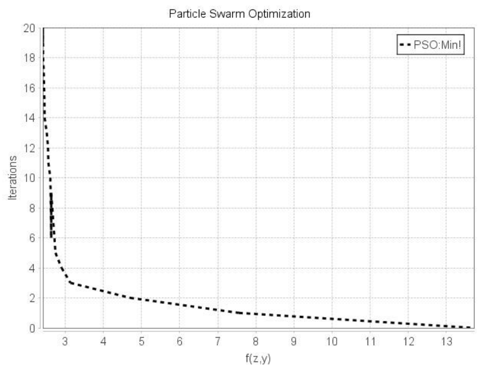




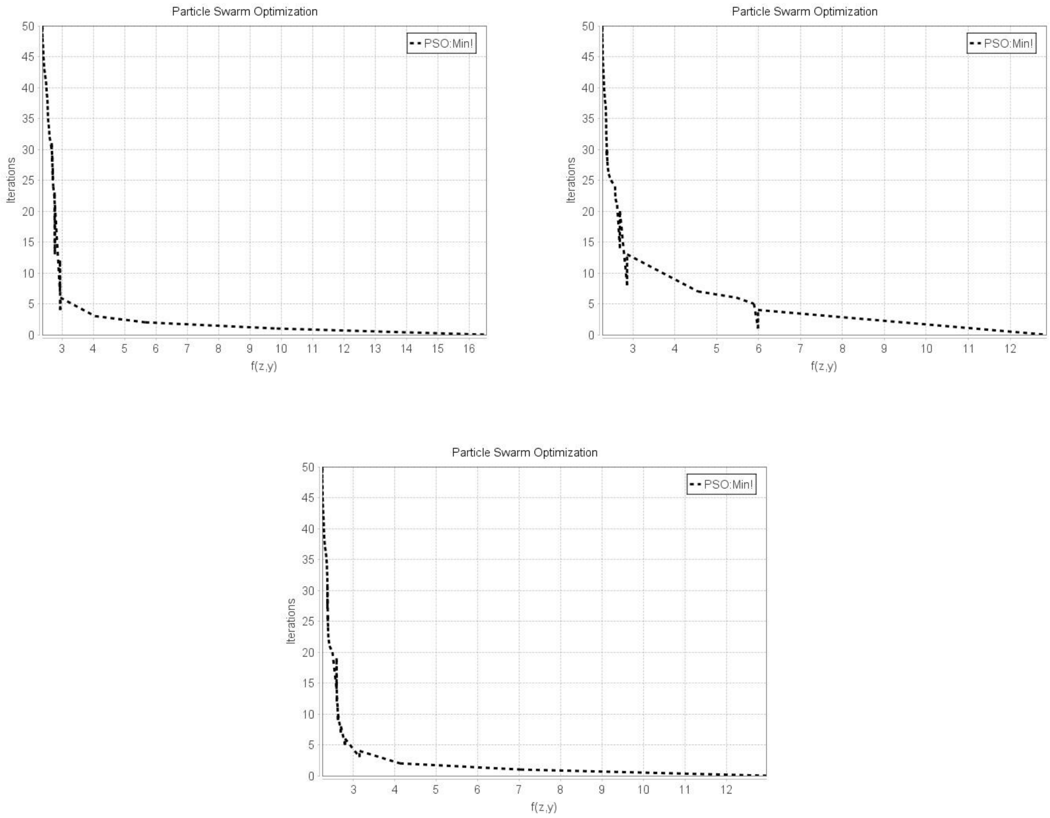
Graphs 4 – i. iterations: 120 / swarm size: a)20, b)40, c)60



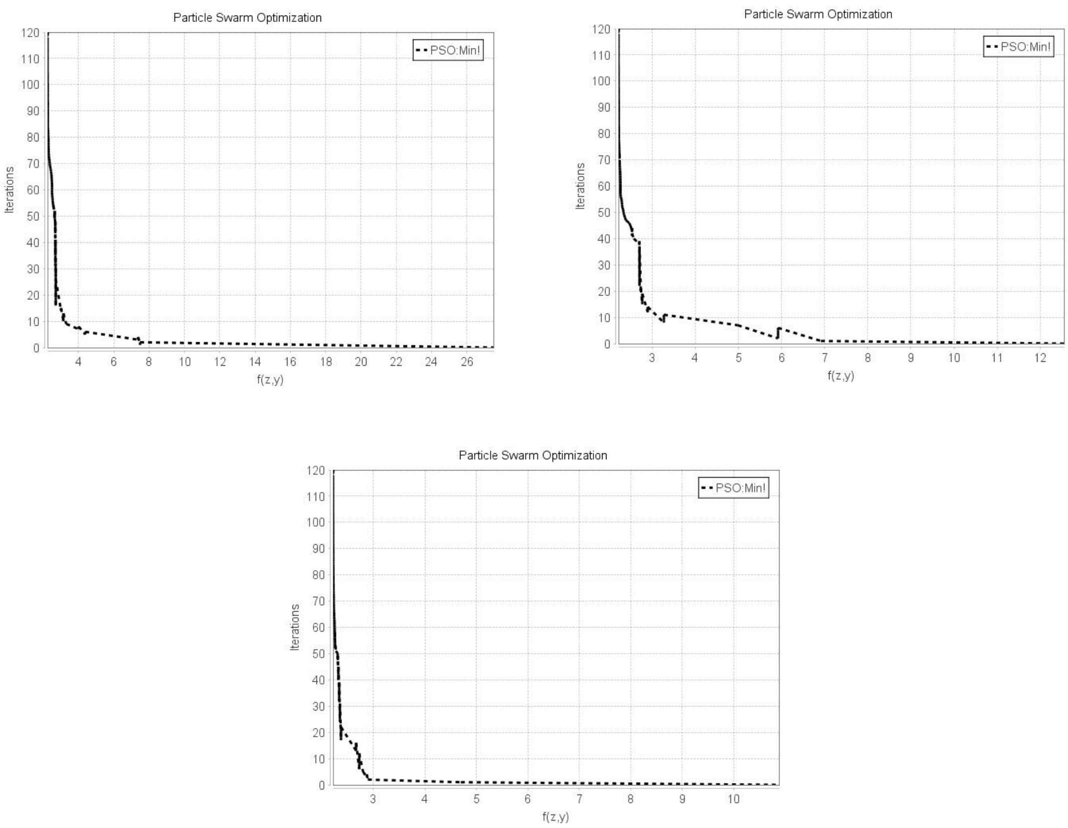




Graphs 3 – ii. iterations: 20 / swarm size: a)20, b)40, c)60



Graphs 6 - ii. iterations: 50 / swarm size: a)20, b)40, c)60



Graphs 5 - iterations: 120 / swarm size: a)20, b)40, c)60

Summary and Conclusion

The deduction of any conclusion will be extracted by comparing the raw data from the tables or by comparing the graphs. We have group the graphs by the number of iterations that take place for each execution of the application. There are two formulae (i, ii) examined, so two different data sets produced by the application.

The first case is a simple polynomial formula. A first look at the [Graphs 1][Graphs 2][Graphs 4] tells us that the amount of particles in a swarm can accelerate the minimization process. It is obvious that it takes a larger amount of iterations for [Graphs 1] to get similar results to [Graphs 4].

When a 60-particle swarm (60psw) is used, the initial value of f(z,y) is 0.13, more than five times lower than a 20-particle swarm (20psw), with 0.68 initial value. That difference is nothing compared to the final solution (20 iterations [Table 1]) of each configuration. For the 60psw is 4.1E-5, 187 times lower than 20psw, 7,7E-3, at 20 iterations.

That difference rises accordingly as the number of iterations increases.

But even if we compare the results from [Table 1] for 60psw only, the first raw gives the solution 4E-5 for 20 iterations and the fourth solution 7,45E-9 for 120, which is 5714 times better than the first one. Just for comparison for 20psw/120-iteration solution is 962 times better.

The question then arises, is it better to use multiple iterations and small swarm or the opposite?

Using *System.currentTimeMillis(),* an object that returns time in milliseconds, is possible to calculate the execution time of the application methods, that are our focus. The measurements showed that 20psw with 120 iterations are slightly faster than using a 60psw with 50 iterations [Table 3]. The advantage of the later configuration is the more stable times and higher accuracy of the solutions.

The only difference between the two formulae, i and ii, is the addition of the exponential factor to the second. That is the reason why the f(z,y) values of ii never reaches near 0, thus minimize [Graphs 4][Graphs 5][Graphs 6]. Also it takes slightly more time for execution, especially with 60psw/50 iteration configuration [Table 3].

In some cases, swarm is trapped [9] in their particles' personal best position for substantial amount of iterations like in here [Capture 2]. Although, a better global position exists, it is not sufficient to move the swarm to the best available solution and further. A solution would be the utilization of an orthogonal learning strategy for an improved use of the information, including faster global convergence, higher solution quality, and stronger robustness [10].

Concluding, overall Particle Swarm Optimization and its convergence variation, works as supposed to. It was possible to minimize any given formula and get a set of solutions for the f(z,y) with random location in a search space. There are also weaknesses, issues with the quality of the solutions or performance slowdowns. All of them already solved [10] by the community with better implementation or use of new technics.

References

[1] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of IEEE International*, Piscataway, NJ, 1995.

[2] M. Clerc and J. Kennedy, "The particle swarm-explosion, stability and convergence in a multidimensional," in *IEEE Transactions on Evolutionary Computation*, 2002.

[3] Y. Shi and R. C. Eberhart, "A modified particle swarm optimizer," in *Proceedings of IEEE International*, 1998.

[4] F. Van den Bergh, "A convergence proof for the particle swarm optimiser," in *Fundamenta 34. Informaticae*.

[5] A. P. Engelbrecht, Fundamentals of Computational Swarm Intelligence, Chichester, UK: John Wiley & Sons, 2005.

[6] breidecker, "JEval," [Online]. Available: http://jeval.sourceforge.net/. [Accessed 29 10 2016].

[7] mungady and taqua, "JFreeChart," [Online]. Available: https://sourceforge.net/projects/jfreechart. [Accessed 29 10 2016].

[8] Z. Zheng and S. Yuhui, "Inertia Weight Adaption in Particle Swarm Optimization Algorithm," in *Advances in Swarm Intelligence*, Chongqing, Springer Berlin Heidelberg, 2011, pp. 71-79.

[9] M. r. Bonyadi and Z. Michalewicz, "A locally convergent rotationally invariant particle swarm optimization algorithm," in *Swarm intelligence*, 2014, p. 159–198.

[10] Z. H. Zhan, J. Zhang, Y. Li and Y. H. Shi, "Orthogonal Learning Particle Swarm Optimization," in *IEEE Transactions on Evolutionary Computation*, 2011.

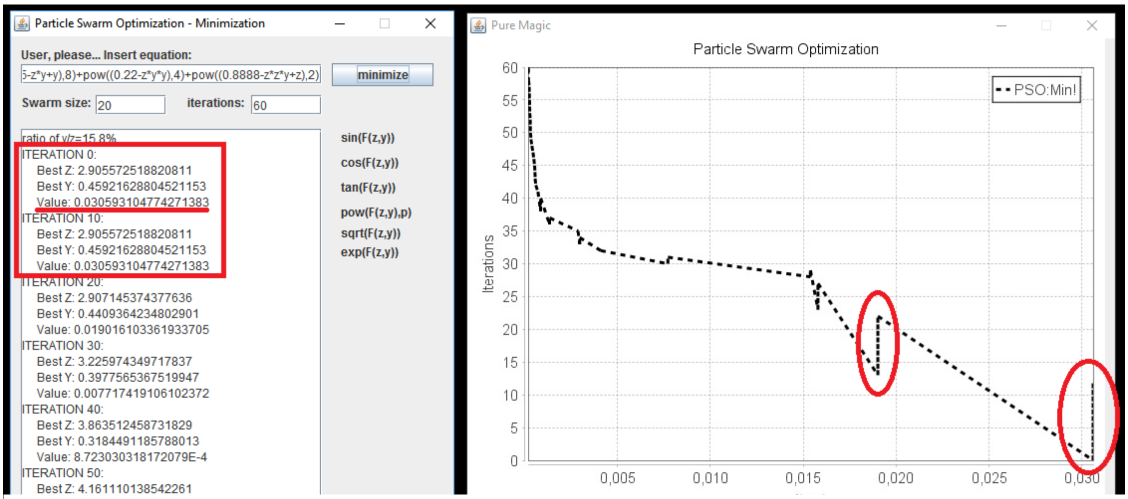
Appendices

1. Benchmark in msec between sw20it120 and sw60it50

	i		ii	
	sw20it120	sw60it50	sw20it120	sw60it50
1	368	3473	4042	3675
2	4384	3462	3084	3609
3	3368	3466	3126	3658
4	3159	3397	3086	3603
5	3111	3467	3420	3497
6	3030	3541	3174	3592
7	2970	3567	3038	3692
8	2512	3855	3076	3604
9	3240	3509	3004	3673
10	3064	3553	3079	3690
11	2909	3456	3915	3761
12	2893	3447	3369	3609
13	3093	3420	3120	3658
14	3344	3494	3150	3611
15	2904	3362	3099	3593
16	3118	3490	3336	3561
17	3080	3466	3114	3630
18	3121	3420	4130	3653
19	4078	3452	3127	3595
20	3117	3479	3057	3308
Σ	**3209,2**	**3488,8**	**3277,3**	**3613,6**

Table 3 – Benchmark

2. Indication of trapped swarm



Capture 2 – same best values of f(z,y) for a few iterations