A Comparison Between the Firefly Algorithm and Particle Swarm Optimization

Submitted by
Michael F. Lohrer

Computer Engineering, Applied Mathematics

To
The Honors College
Oakland University

In partial fulfillment of the
requirement to graduate from
The Honors College

Mentor: Darrin M. Hanna, Associate Professor of Engineering
Department of Electrical and Computer Engineering
Oakland University

03/01/2013

Abstract

When a problem is large or difficult to solve, computers are often used to find the solution. But when the problem becomes too large, traditional methods of finding the answer may not be enough. It is in turning to nature that inspiration can be found to solve these difficult problems. Artificial intelligence seeks to emulate creatures and processes found in nature, and turn their techniques for solving a problem into an algorithm. Many such metaheuristic algorithms have been developed, but there is a continuous search for better, faster algorithms. The recently developed Firefly Algorithm has been shown to outperform the longstanding Particle Swarm Optimization, and this work aims to verify those results and improve upon them by comparing the two algorithms with a large scale application. A direct hardware implementation of the Firefly Algorithm is also proposed, to speed up performance in embedded systems applications.

Table of Contents

1. Introduction

Many problems today have been adapted to and solved with computers. Computers provide a means to perform calculations more quickly and accurately than by hand, and even make many problems that would otherwise be impossible to solve, solvable. However, there are still classes of problems that, although technically solvable, would take extreme periods of time to solve. For some problems taking a second to calculate the result may be a long time, but for others getting the result in a day may be quick; how reasonable the time taken to calculate the best answer depends completely on the problem at hand. Other problems may not have a solution where it is possible to find the best answer in a reasonable amount of time. For example, trying to calculate the optimal routing for planes for an airline company is an immense problem. There are obvious economic benefits if planes could be flown with minimal empty seats and minimal time waiting at an airport grounded. Since there are many factors that go into this problem including thousands of planes and thousands of airports, with consumer demands changing daily, this problem cannot be solved in a reasonable amount of time; a computer cannot solve this problem and produce results for the airline industry fast enough.

For these large scale problems, finding the exact answer is impossible within a reasonable amount of time. Finding an approximation, however, is possible. In order to do this, a class of programs called artificial intelligence was created, which aims at optimizing any problem as quickly as possible. The downside is that they do not guarantee the best result. But if the best possible solution cannot be obtained in reasonable time, it may not be useful anymore. In this situation artificial intelligence becomes useful. In the plane routing problem, if artificial intelligence can be used to find a close-to-optimal result in a week or less, then airplane routes can be generated by a computer that are almost optimal. If someone were trying to find the

source of a chemical leak, another difficult problem to solve, then getting a good result in a week would not be good enough. Finding the source of the chemical emissions would need to be determined in seconds or a few minutes in order to be the most useful. If standard approaches do not meet these time criteria, then artificial intelligence is a good alternative.

Artificial intelligence is largely inspired by nature. An early class of intelligent programs was based on the idea of evolution and survival of the fittest. Others get their inspiration from colonies of ants, bees, the human immune system, metallurgy, and many other natural processes. In particular, a type of artificial intelligence that has been developed is modeled after swarms of flies and fireflies resulting in an algorithm, or program, called particle swarm optimization and another called the firefly algorithm. The goal of this work will be to compare these two algorithms. There are test problems that can be used to compare them, some of which have lots of local best points, or areas that seem good, but are not the best. To ensure particle swarm optimization and the firefly algorithm scale up well, experiments will be presented applying these algorithms to a very large application. The specific application will be that of finding the source of an emission, such as a chemical or biological hazard spreading through the air from data collected by sensors placed around a large area such as New York City. Sensors that would be used in the gathering of data for this problem will be simulated, and will have various amounts of noise, or randomness, added to them since sensors in real life have noise and errors. On the test problems, the firefly algorithm outperforms particle swarm optimization in every way. For smaller scales of the source localization problem, the firefly algorithm performs better with very noisy sensors. For large source localization configurations however, the main benefit to the firefly algorithm is its speed, as the results obtained are close, but not any better.

Finally, a direct hardware implementation will be introduced as future work. This represents a possible reduction in time, size, and power required to run the firefly algorithm. A general design is given, and the results of implementation so far.

2. Global Optimization

2.1 – Problems and Strategies

Optimization, in a general sense, has the goal of obtaining the best possible result given a range of choices. These choices can be represented with variables in a function, and the result represented by the function evaluation. Thus optimizing a given function is to seek the parameters which lead to the largest, or smallest, possible outcome. Whether the largest or smallest value is desired depends on the specific application, but for either case, the problem can simply be flipped to produce the other. This allows all problems to be treated as minimization problems, which will be the case for the course of this paper.

For a real world problem, the first challenge is to determine a function that models it. A company may seek to minimize material used in a product in order to cut costs. If a cereal company optimizes the design of its cereal boxes, it could reduce the amount of material used per box. The surface area of the box can easily be modeled into a function of the width, height, and length, while the volume contained inside the box can be used to constrain the problem. The output of the function would be the surface area, or material, required, and the parameters would be the dimensions of the box. Now an optimization technique must be used to determine the ideal parameters.

There are two categories of optimization techniques: exact and heuristic. Exact strategies guarantee the optimal solution will be found, and work well for many problems. However for complex problems or ones with a very large number of parameters, exact strategies may require

very high computational costs. A large amount of real-world problems fall in this category of complex problems, and in order to solve them in a reasonable amount of time a different approach is needed. For these problems, artificial intelligence using heuristic strategies is beneficial. A heuristic is a search technique that provides advice on the next move to make, that may or may not be the best way to move. Heuristics are the key to the speed of artificial intelligence, but also why there is no guarantee that the optimal solution will be found. They can come very close to the best solution though, depending on the technique used, and most heuristics will significantly outperform exact strategies for high complexity problems. If an approximate solution is satisfactory, if the heuristic solution is likely to be close to the optimal solution, or if the exact solution requires more time so find than is reasonable, then for these complex problems a heuristic approach is appropriate.

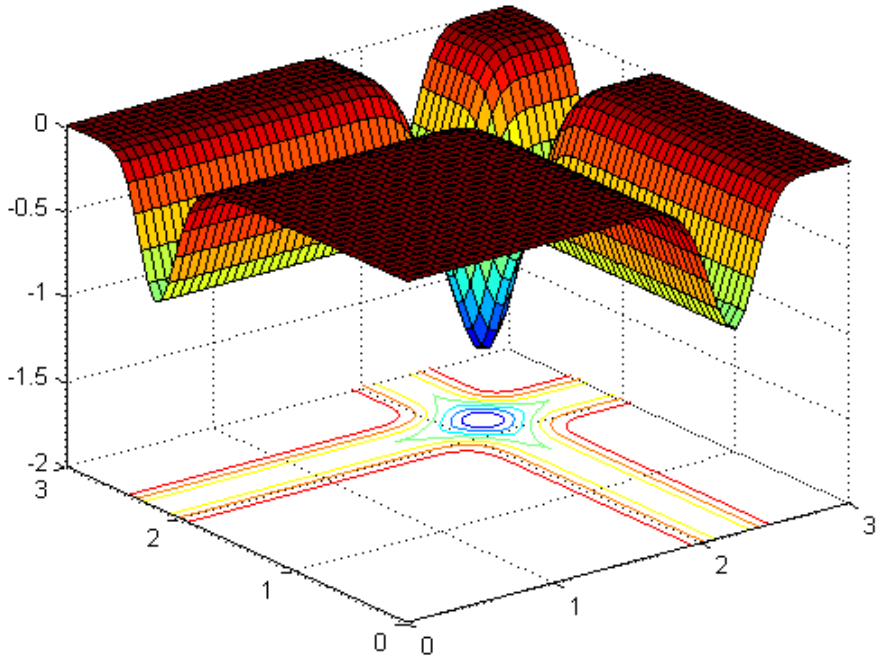


Figure 2.1. Michalewicz's function of two dimensions

The function shown above is Michalewicz's function, with two parameters. Each parameter is represented on the graph as a dimension, with the third dimension being the output

of the function. Functions used for the objective function can have any number of dimensions, but for ease of visualization, a 2-D version is easier to visualize than some D-dimensional problem with large D. Each parameter is also bounded, in this case in the range of zero to three for each parameter. For the optimization problems considered here, every parameter has a bound for the range of possible values it can take on. For real-world problems, these bounds would be determined by physical limitations or other knowledge of the problem that can lead to eliminating impossible or improbable solution spaces. These ranges limit the space required to search to find the optimum, while making sure to not exclude it, and are important for heuristic strategies. The smaller the search space, the less time required and the more accurate the result when using a heuristic method.

It can also be seen from Michalewicz's function that local minimum are possible. The global minimum has the true optimal value, but local minima can prove to be a roadblock to that goal. Local minima can provide what appears to be a good result, when in fact there is a better solution. If not used carefully, many heuristic approaches can get stuck in one or more of these local minima, never finding the desired global minimum. Thus if a problem's objective function potentially has many local minima, a major consideration for which approach to use in solving it has to be how likely it is to not move away from a local minimum.

For problems with many dimensions, exact solution techniques and simple heuristics can be ruled out. To go through all the remaining possibilities would be beyond the scope of this work, but many modern heuristic algorithms are detailed in "The Handbook of Global Optimization," Volume 2 [1] and "Essentials of Metaheuristics" [2]. Metaheuristics can be applied to a broad range of problems; the algorithm providing the solution sees the problem as a

black box. Metaheuristics are some of the most efficient and accurate algorithms for solving complex problems, when an exact approach is not reasonable.

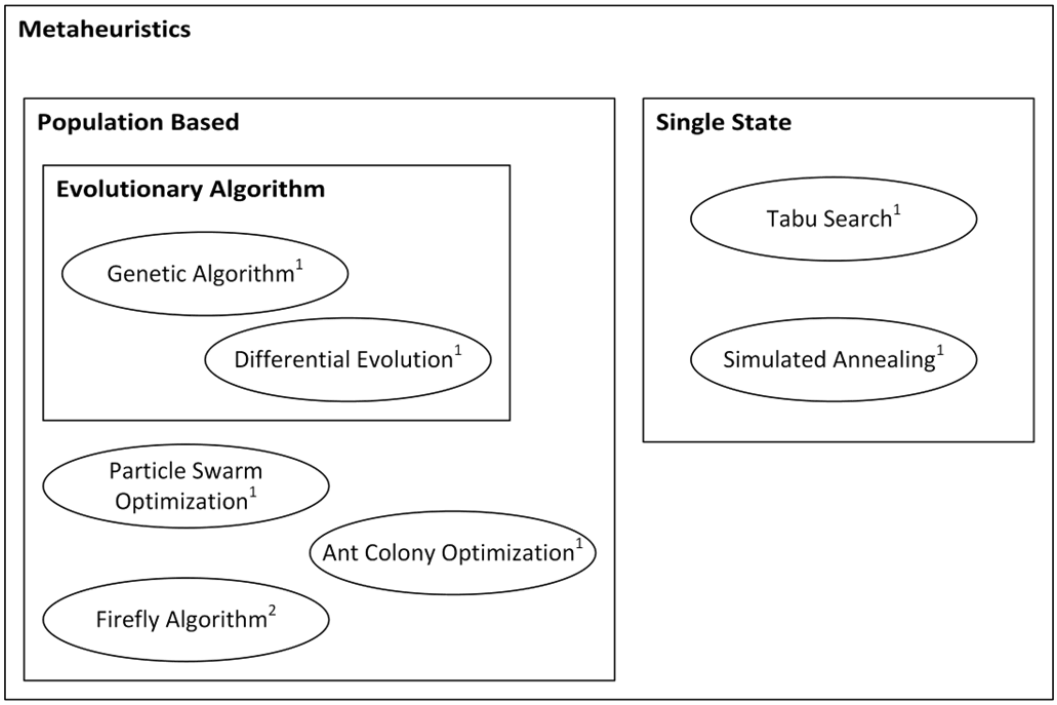


Figure 2.2. Classification of heuristics [2] [3]

The figure above shows the classification of several popular metaheuristics. The single state algorithms, tabu search and simulated annealing, have a single candidate solution that is stored and used to compare against possible new solutions. Population based algorithms store many candidate solutions, and compare them against each other. For exploring large, continuous space regions, single state algorithms are not very efficient because of the lack of comparisons between candidate solutions, and are generally not as good at dealing with escaping local optima. Population-based metaheuristics can be much faster. Some commonly used population-based algorithms are the genetic algorithm and particle swarm optimization, and a recently developed algorithm called the firefly algorithm shows a lot of potential.

2.2 – Genetic Algorithm

The genetic algorithm (GA) is the archetypical evolutionary algorithm. It was invented in 1975 and is modeled after the biological concept of natural evolution of genomes [4]. The GA encodes the parameters of the objective function into a chromosome, which corresponds to a single candidate solution. Multiple chromosomes make up the genome, or population. For an initial population, random chromosomes are generated and evaluated using the fitness function of the problem being optimized. Then the algorithm simulates a "survival of the fittest" type scenario, where each generation of the algorithm attempts to improve upon the preceding generation.

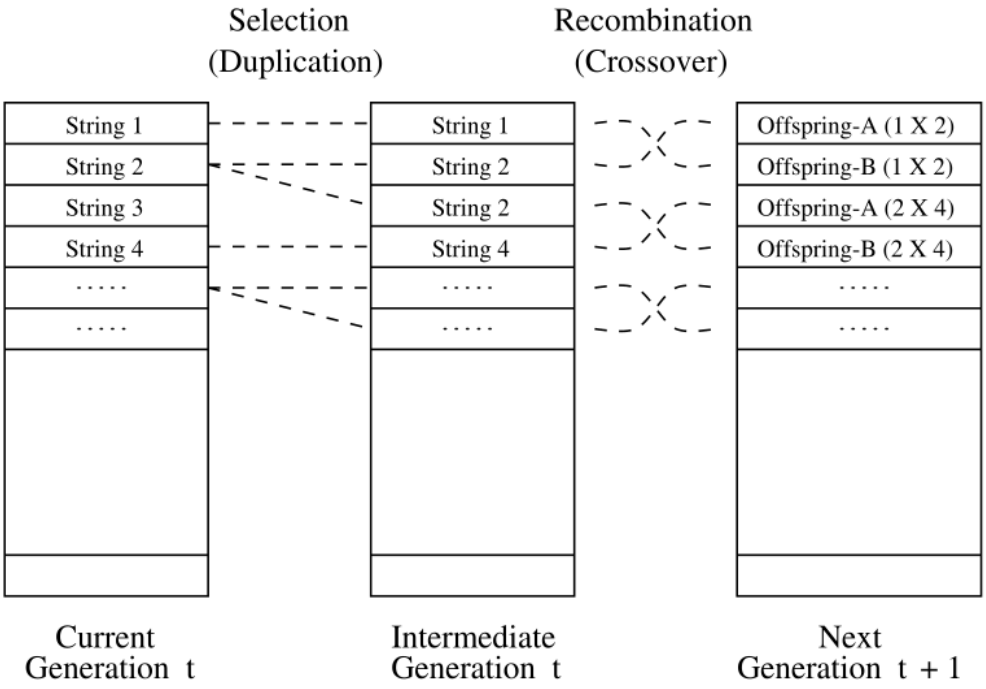


Figure 2.3. GA selection and crossover [5]

For each generation of the GA, three steps are performed: selection, crossover, and mutation. First, the chromosomes from the preceding generation become the parents of the new generation, through a process called selection. The probability of being selected, or even

duplicated, for the list of parents is based on the fitness function evaluation. The better the fitness of the chromosome, the more likely it will survive to be a parent. The selection process is illustrated in figure 2.3, along with the crossover process. Once the parents are selected, crossover begins. Each parent is paired with another, and some of their genes are swapped with each other. After selection and crossover, the parent chromosomes are replaced with their children, and mutation occurs. With a certain probability that is picked by the user of the GA, each gene is modified. The probability selected is usually very low, otherwise the GA will approach a random search as the mutation probability increases.

The GA encodes its chromosomes with binary strings of 0 or 1, and so they perform well for many discrete problems. For problems dealing with continuous space, the GA has to be adapted or combined with another algorithm in order to work. Although this has been done, there are other algorithms that already operate in continuous space, such as the more recently developed particle swarm optimization and firefly algorithm. Additionally, Particle swarm optimization has been shown to perform better than a GA in [6] and [7] and perform quicker in [8]. From the results in literature, it appears that the particle swarm optimization algorithm is a superior choice than the GA, since the two main goals of artificial intelligence algorithms is accurate results and high speed.

2.3 – Particle Swarm Optimization

Particle Swarm Optimization (PSO) models its behavior after the swarming or flocking patterns of animals. Instead of chromosomes, PSO has particles that make up its population, called a swarm. Unlike the GA, there is no "survival of the fittest" selection process for determining the particles that survive to the next generation, but rather just mutation. Each particle is simply moved from one location to another. This mutation is performed in a directed

manner, such that each particle is moved from its previous location to a new, hopefully better, location. The location update process is drawn with vectors in figure 2.5 below.

Each particle knows its position, velocity, and personal best location found so far, and the global best. The update operation for a particle then occurs according to the following function [9]:

$$v_{ij}(t) = v_{ij}(t-1) + \varphi_1 * r_1 * (p_{ij} - x_{ij}(t-1)) + \varphi_2 * r_2 * (p_{gj} - x_{ij}(t-1)) \qquad 2.4$$

where i = 1, 2, …, N for N particles, and j = 1, 2, …, D for D dimensions, v_{ij} is the velocity of the particle, t is the generation or time-step, r_1 and r_2 are random numbers in the range (0, 1), p_{ij} is the personal best location found so far, p_{gj} is the global best, x_{ij} is the location of the particle, and φ_1, φ_2 are called learning rates.

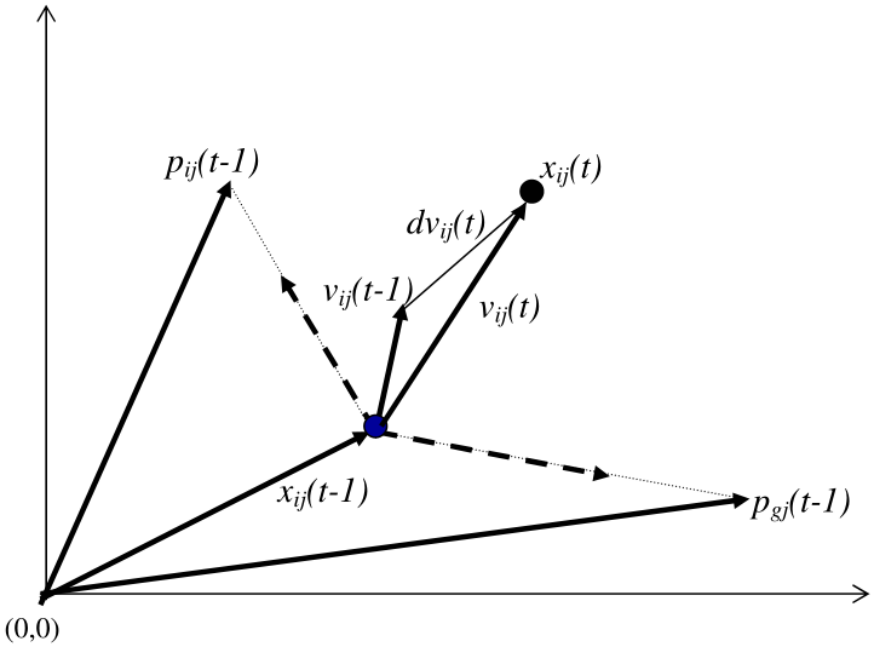


Figure 2.5. PSO update visualized in 2-D space [9]

3. The Firefly Algorithm

The firefly algorithm (FA) is another swarm intelligence algorithm, developed by Xin-She Yang [3]. This new intelligent algorithm is intriguing because its author has shown that it is not only faster than Particle Swarm Optimization, but also that it provides better, more consistent results. It is also interesting that, given the right parameters, the FA essentially becomes PSO. This means that the FA is a more general form of PSO, and can be adapted better to a given problem.

3.1 – The Concept behind the Firefly Algorithm

The Firefly Algorithm was inspired by the flashing of fireflies in nature. There are over 2000 species of fireflies, most of which produce a bioluminescence from their abdomen [10]. Each species of firefly produces its own pattern of flashes, and although the complete function of these flashes is not known, the main purpose for their flashing is to attract a mate. For several species the male is attracted to a sedentary female. In other species, the female can copy the signal of a different species, so that the males of that species are lured in. The female then preys on these males. The flashing can also be used to send information between fireflies. The idea of this attractiveness and information passing is what leads to the inspiration for the FA.

The FA idealizes several aspects of firefly in nature. First, real fireflies flash in discrete patterns, whereas the modeled fireflies will be treated as always glowing. Then, three rules can be made to govern the algorithm, and create a modeled firefly's behavior [6].

1. The fireflies are unisex, and so therefore potentially attracted to any of the other fireflies.
2. Attractiveness is determined by brightness, a less bright firefly will move towards a brighter firefly.
3. The brightness of a firefly is proportional to the value of the function being maximized.

When comparing the brightness of any two fireflies, the locations of the fireflies must be considered. In the real world, if a firefly is searching for another, it can only see so far. The farther another firefly is from it, the less bright it will be to the vision of the first firefly. This is due to the light intensity decreasing under the inverse square law. That is, the light intensity of a firefly a distance r away from the observer will be reduced by a factor of $1/r^2$. The air will also absorb part of the light as it travels, further reducing the perceived intensity. This consideration is also factored into the FA.

3.2 – The Algorithm

To start the algorithm, the fireflies are placed in random locations. The location of a firefly corresponds to the values of the parameters for the objective function to be solved. For a function of three variables, the firefly's position would be easy to visualize in 3-D space, whereas with a function of ten variables it would be a much more difficult task to find a visualization. Then from each firefly's newly acquired position, the objective function is evaluated, and the firefly's light intensity is set as the inverse evaluation. The inverse is used since the goal is to minimize the objective function. Thus a lower function evaluation will result in a higher light intensity. For a demonstration of initialization, shown in figure 3.2 is the initial location of twelve fireflies, in the 2-dimensional sphere function problem. The global minimum for this problem is 0 at the location (0,0), the origin. The contour lines on the graph show that the further away from the origin, the higher the function value. The sphere function is discussed in greater detail in section 4.

After initialization, each firefly is compared against all the rest, and will move towards every brighter firefly encountered. Once a firefly has determined it needs to move towards a brighter firefly, several things have to happen. First, the distance between the fireflies, r, has to

be calculated. Any form of distance calculation that makes sense for the given problem can be used, but for the general case the Cartesian distance is appropriate. The Cartesian distance between two fireflies in D-dimensional space is given as:

$$r_{ij} = \sqrt{\sum_{d=1}^{D} \left(\boldsymbol{x}_{id} - \boldsymbol{x}_{jd}\right)^2} \tag{3.1}$$

where $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are the position vectors for firefly i and j, respectively, with $\boldsymbol{x}_i(n)$ representing the position value for the *n*th dimension.

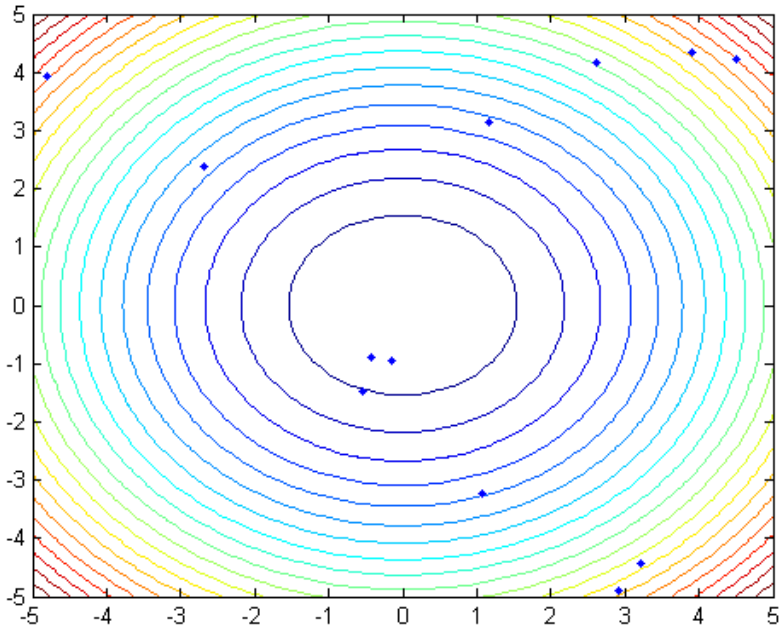


Figure 3.2. Initial locations for 12 fireflies

With the distance between two fireflies known, the attractiveness β can be determined. Since the attractiveness is based on the light intensity, two terms go into the calculation of β. First, from the inverse square law $\beta_1(r) = I_0/r^2$, where I_0 is the intensity of the firefly being moved towards. Then due to the absorption through the air, $\beta_2(r) = I_0 e^{-\gamma r}$, where γ is the absorption coefficient, and $\gamma \in [0, \infty)$. A γ of 0 yields no absorption, while a large γ relates to the fireflies flying in a heavy fog. Then to produce an approximation of the combined terms, and

to avoid the undefined result of $\beta_1(0)$, we obtain $\beta(r) = I_0 e^{-\gamma r^2}$. However, calculating an exponential is expensive, and $I_0 e^{-\gamma r^2}$ can be approximated by $I_0/(1+\gamma r^2)$, which is not as difficult to calculate. Since the attractiveness is directly related to the light intensity, we can take $\beta_0 = I_0$, where β_0 is the attractiveness at $r = 0$, and typically $\beta_0 \in (0, 1]$. Combining the last two remarks results in the expression $\beta(r) = \beta_0/(1+\gamma r^2)$. Changing β_0 changes how attracted fireflies are to others, so lowering β_0 lowers the desire for a firefly to move towards brighter fireflies.

The movement of a firefly towards a brighter firefly is determined by $\beta(r)$ and a random component. The random component is key for all metaheuristic algorithms; it allows the algorithm to escape from local optimums. A simple way to create a random distance to move is with a uniform distribution in the range of $[-0.5,0.5]$. Since both positive and negative values possible, the movement can be either forwards or backwards. Another important factor is the scale of the problem. If two parameters of the objective function have different ranges of possible values, a fixed range of random numbers would cause different relative randomness for each dimension. To solve this problem, the random numbers generated can be multiplied with the scale of the dimension to produce a vector of scaling values, S. With all the terms together, the position update equation for a firefly i being attracted to firefly j becomes [6]:

$$\boldsymbol{x}_i = \boldsymbol{x}_i + \beta_0/\left(1+\gamma r_{ij}^2\right)\left(\boldsymbol{x}_j - \boldsymbol{x}_i\right) + \alpha S(R - 0.5) \qquad 3.3$$

where R is a set of uniformly distributed random numbers in the range of $[0,1]$, and α is a parameter controlling the amount of randomness. The randomness parameter α is typically in the range $[0,1]$, where 0 corresponds to no randomness and 1 corresponds to being highly random.

Figure 3.4 shows the position update of the fireflies from their initial locations as shown in figure 3.1. Each firefly has the corresponding function evaluation written next to it. As explained previously, the lower the value, the brighter the firefly. The black vectors correspond to the amount of attraction between firefly i at time $t-1$ and all the fireflies brighter than it, without the random component added. The move to the new position, $x_i(t)$, is shown with the green vector.

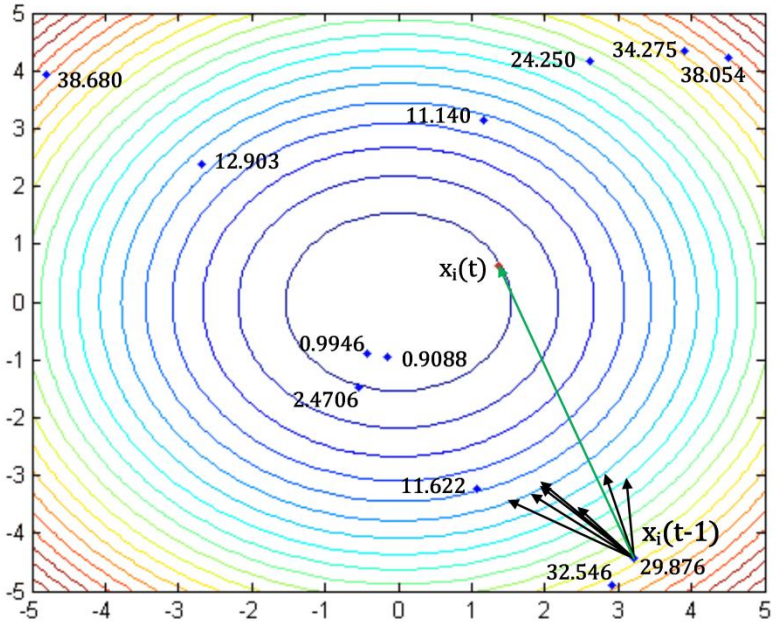


Figure 3.4. Firefly position update shown with vectors

After a firefly has been moved towards all brighter fireflies, its brightness is updated by evaluating the objective function in the new position. The new evaluation is also compared against the best found so far. If it is better, that position becomes the new best. In this way, if a firefly passes through a location better than any other found, but for whatever reason it or any other firefly does not end up there by the last iteration of the algorithm, that best location is still recorded. After re-evaluation, the given firefly is done with its update, and the algorithm moves on to the next firefly. This continues for every firefly, and then one generation of the algorithm

is done. Figure 3.5 shows the sphere function example after one complete generation. Notice after one generation there are no longer fireflies near the edge of the search space, and after five generations, shown in figure 3.6, the fireflies are tightly clustered around the origin.

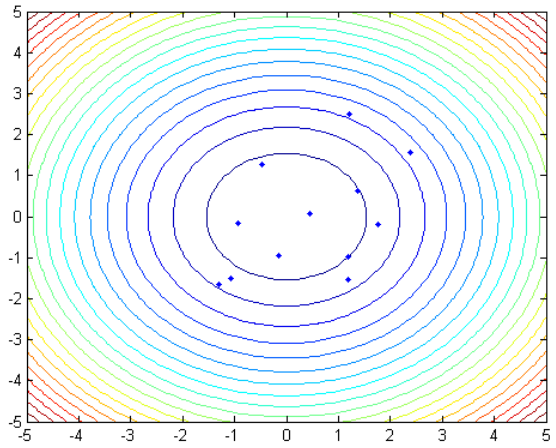


Figure 3.5. After one iteration

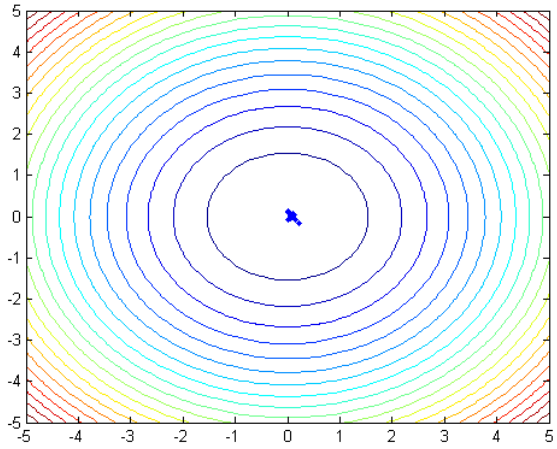


Figure 3.6. After five iterations

After each generation, it must be determined if the algorithm is complete or not. The simplest method for deciding when to stop is to run for a set number of generations. In our sphere function example, it would appear five to ten generations is enough, so for subsequent trials we could just have the FA run for ten generations, and see what we get. A larger problem with more dimensions could take much longer, and thus several hundred generations may be required. This method is commonly used for its simplicity, but it can be inefficient. Another method is to check how much better the results are getting. Since the best firefly is stored, the previous best can also be stored. If the difference between the best and previous best is very small, then likely the fireflies have converged on the optimal location. This method is more complicated, however, since the value cannot be too large, as the global optimum may not have been reached yet, but it cannot be too small, because even when the fireflies converge, there are still small movements being made, and thus the algorithm would never end. A third approach to ending the algorithm is to set a value that is "good enough," so if the best firefly is as good or

better than a given threshold value, we do not need to proceed any farther. This method is also easy to use, but it requires the knowledge of what is a good solution for the given problem. These end conditions can be improved on by using any combination of them. For example, it is always a good idea to have a maximum number of generations to run set, in case another end condition never is met. Then at least the algorithm will not run forever. Now that the entire algorithm has been covered, a summary in pseudocode is given in figure 3.7 below.

Firefly Algorithm

Given:
Objective function f(**x**), where $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ for d dimensions.
Light intensity I_i at $\mathbf{x}_i$ is determined by f($\mathbf{x}_i$)
Define $\alpha, \beta, \gamma, \delta$

```
Give random locations for N fireflies in D dimensions
Initial evaluation of all N fireflies
while (End condition not met)
    Increment t
    for i = 1 to NumFireflies
        for j = 1 to NumFireflies
            if (Ij > Ii)
                Move firefly i towards j in d-dimension
            end if
        end for j
        Evaluate new solution and update light intensity
        Check if the best position found so far
    end for i
    Reduce alpha
    Sort the fireflies
end while
Find the best firefly
```

Figure 3.7. FA Pseudocode

3.3 – Improvements

On top of the base algorithm, several improvements can be made. The first, suggested by Yang [11], has a major effect. It combines the FA with concepts from simulated annealing (SA), mentioned previously in figure 2.2. SA models the annealing process of metals, in which there is

a given temperature schedule. The temperature starts high, and is decreased as time goes on. The temperature is analogous to the amount of randomness, and the generations of the algorithm simulate time passing. This idea of reducing the randomness can be applied to the FA as well. Initially, high randomness is a good thing, but as the fireflies find the better locations, less randomness is beneficial. Thus adopting a randomness reduction method greatly benefits the FA. The best reduction schedule to use varies with the problem at hand, but for the purposes of the experiments undertaken in the next section, the randomness reduction will be determined by the following equation [11]:

$$\alpha(t) = \alpha_0 \delta^t \tag{3.8}$$

where δ is the randomness reduction parameter, and $\delta \in (0, 1]$. A δ of 1 corresponds to no reduction, and lowering δ results in a quicker reduction.

A second improvement can help in the case of large differences in the scale of parameters. The random component is already scaled with the scaling vector S, but the distance between fireflies r is not. Thus the attractiveness β is not scaled, since it relies on r. To fix this, the distance calculation can be modified from the standard Cartesian distance equation to

$$r_{ij} = \sqrt{\sum_{d=1}^{D} \left[\left(\boldsymbol{x}_{id} - \boldsymbol{x}_{jd} \right) / S \right]^2} \tag{3.9}$$

with the familiar scaling values of S. With this implemented, the entire update equation is scaled to the individual parameters of the objective function.

4. Preliminary Examination of the Firefly Algorithm

4.1 – Benchmark Functions

In order to test and verify the FA, there are several benchmark functions that can be used. These functions allow comparisons to be made between this implementation of the FA and others, as well as between the FA and other algorithms. With benchmark functions both speed and accuracy can be measured and compared. In figures 4.2-4.6, five common benchmark functions are shown in their two-dimensional form. All but Schaffer's *f6* function are usually defined with 30 parameters. Extrapolating from the two-parameter form, it is not difficult to imagine how complex solving in 30 dimensions becomes. Schaffer's *f6* is not generalized like the rest, so it remains a function of two parameters. Even with only two parameters, it can be seen that it is fairly complex.

All of these functions have a global minima of $f(\boldsymbol{x}) = 0$. The ending criteria will be set to a maximum number of generations of 500 or when it finds a value at or below the error threshold. The thresholds for each function are given in figure 4.1. Any execution of the FA that results in finding a value at or below the threshold will be considered to have found the global minimum. The range of possible values for each parameter is also given in figure 4.1. These values

Function	Dimension	Range	Error Threshold (≤)
Sphere	30	[-100,100]	0.01
Rosenbrock	30	[-30,30]	100
Rastrigin	30	[-5.12,5.12]	100
Griewank	30	[-600,600]	0.1
Schaffer's f6	2	[-100, 100]	0.00001

Figure 4.1. Benchmark function dimensions, ranges, and error thresholds

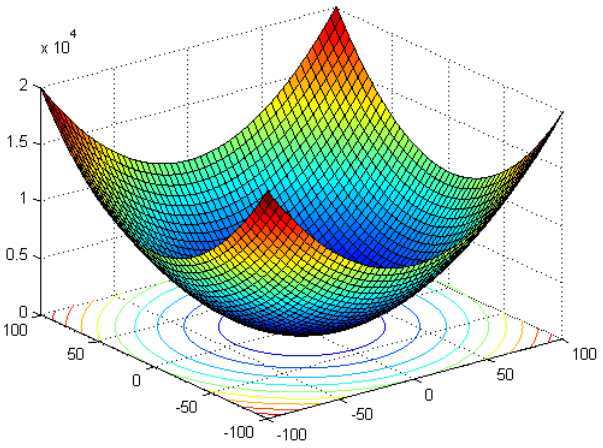


Figure 4.2. Sphere function

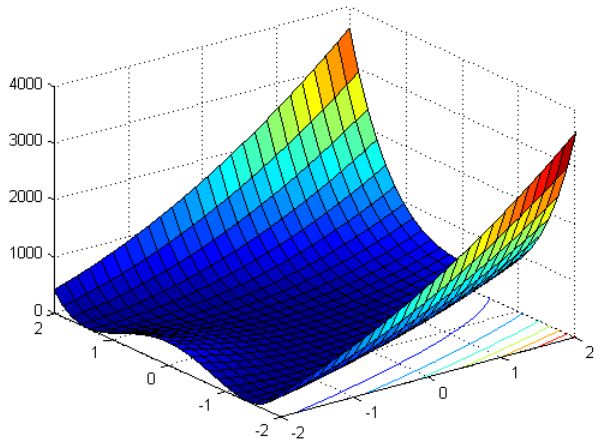


Figure 4.3. Rosenbrock function

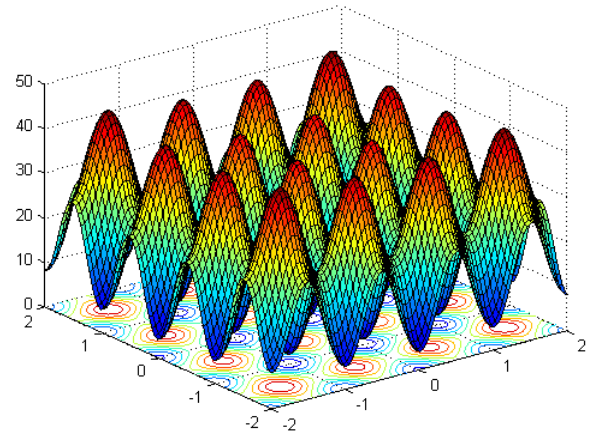


Figure 4.4. Rastrigin function

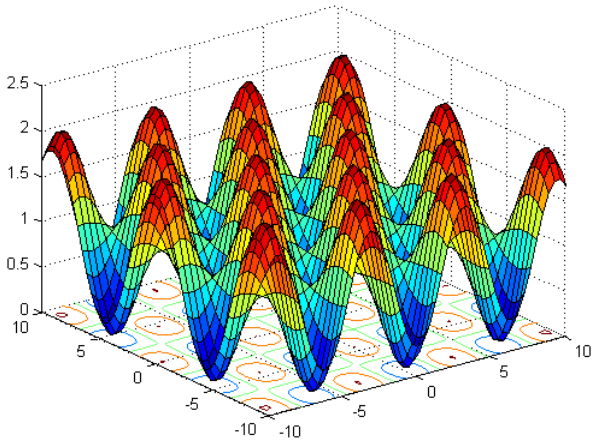


Figure 4.5. Griewank function

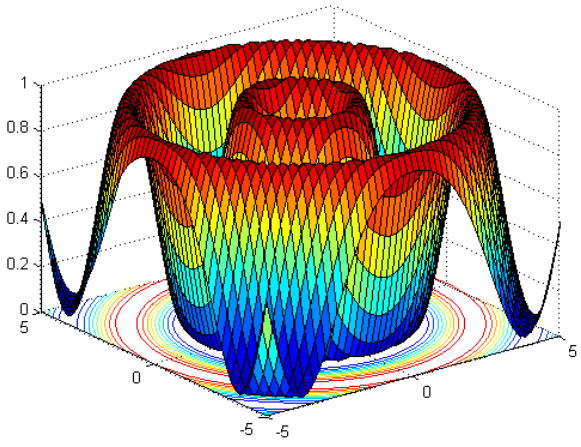


Figure 4.6. Schaffer's *f6* function

represent the maximum and minimum values for any parameter, and are the same for each dimension. All of the function criteria are the same as commonly used in literature, in order to make comparisons easier and more meaningful [9].

4.2 – Matlab results and comparisons

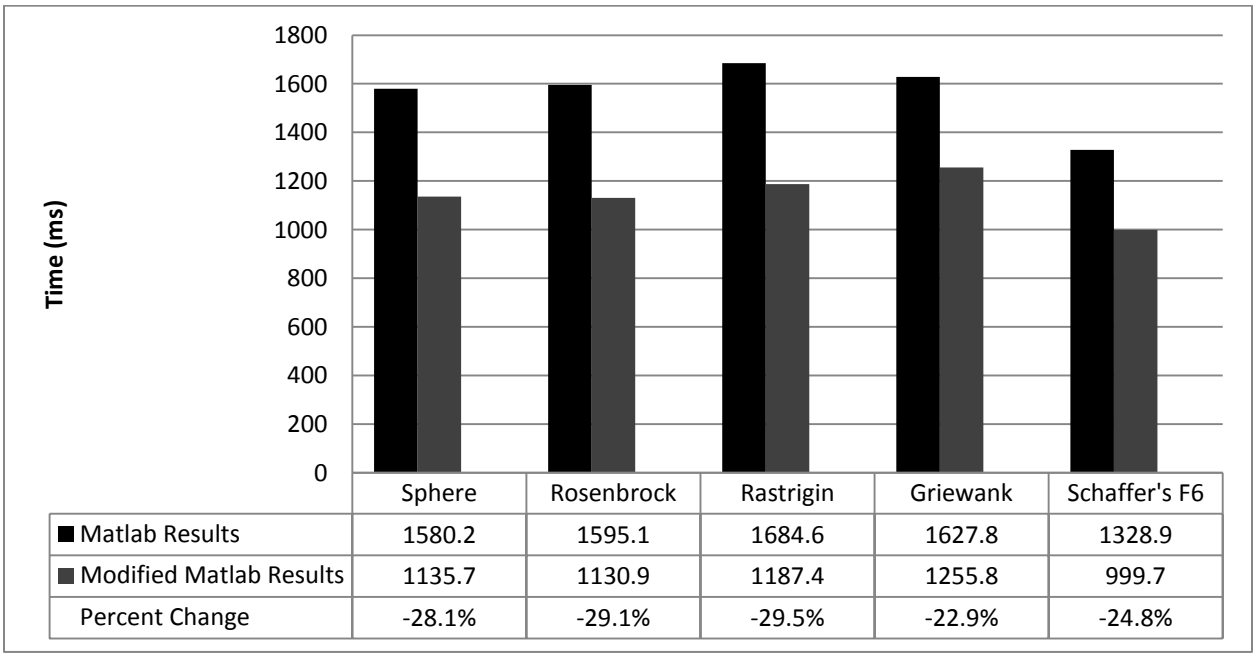
The first testing was done in Matlab. The FA has been implemented by Yang and made freely available online [12] or in his book *Engineering Optimization: An Introduction with Metaheuristic Applications* [11]. For these tests, the number of fireflies used was 30, and the parameters were set such that $\alpha = 0.2$, $\beta = 0.2$, $\gamma = 0.8$, and $\delta = 0.982$. A laptop with an Intel Core i7 740QM processor was used to run the algorithm. The algorithm was run for 500 generations, without the threshold stopping condition since that was not implemented in the available Matlab code. After 500 generations, the time taken was recorded, as well as the best value found, and are shown in figures 4.8 and 4.9, respectively. Since the FA is stochastic in nature, having randomness involved, the results shown are the average of 20 runs, to provide statistically valid data.

After running and analyzing the provided Matlab FA software, a couple possible software improvements became clear. First, and with the largest effect, the calculation of the distance between fireflies was set up to be calculated between every possible pair of fireflies. This does not need to happen, the distance only need to be calculated when the firefly is going to move towards another. This reduces the number of distance calculations per generation, and hence improves the speed of the algorithm. Next, substituting equation 3.2 into equation 3.3 results in

$$\boldsymbol{x}_i = \boldsymbol{x}_i + \frac{\beta_0}{\left(1 + \gamma \sum_{d=1}^{D} \left(\boldsymbol{x}_{id} - \boldsymbol{x}_{jd}\right)^2\right)} \left(\boldsymbol{x}_j - \boldsymbol{x}_i\right) + \alpha S(R - 0.5) \qquad 4.7$$

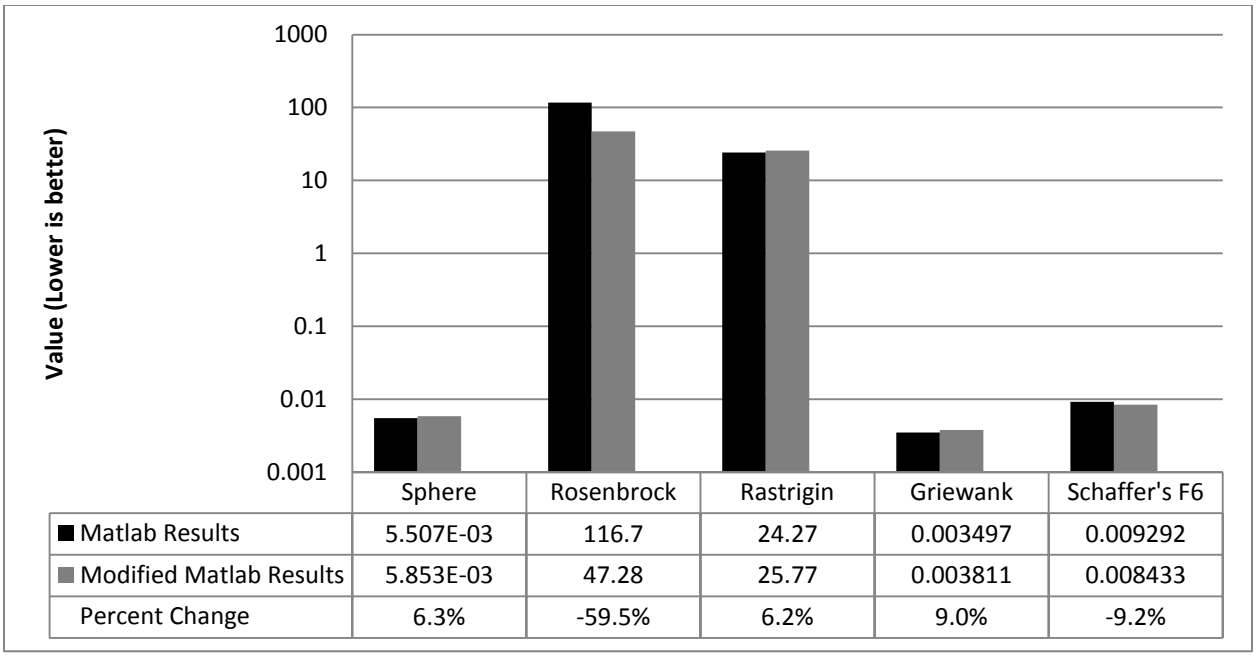
which shows that the distance r_{ij} does not need to be calculated, but simply r_{ij}^2. This saves taking the square root, an expensive computation. Another expensive computation already removed is the exponential in the β calculation. The Matlab FA provided was changed to use the equation in figure 4.7, which also improved the calculation time. The reduced time taken after implementing these changes is shown in figure 4.8 alongside the original time taken.

To verify that the results were still similar, the modified Matlab FA was run tested again with the same parameters as before, and the results are shown in figure 4.9. The percent changes of both time and the results are also shown in figures 4.8 and 4.9, respectively. The time improvement was fairly uniform at 23-30% improvement, while the results of the modified Matlab FA were within 10% of the original except in the case of the Rosenbrock function. For that function, there was an almost 60% improvement on average.



	Sphere	Rosenbrock	Rastrigin	Griewank	Schaffer's F6
Matlab Results	1580.2	1595.1	1684.6	1627.8	1328.9
Modified Matlab Results	1135.7	1130.9	1187.4	1255.8	999.7
Percent Change	-28.1%	-29.1%	-29.5%	-22.9%	-24.8%

Figure 4.8. Time taken for the Matlab FA on the five benchmark functions



	Sphere	Rosenbrock	Rastrigin	Griewank	Schaffer's F6
Matlab Results	5.507E-03	116.7	24.27	0.003497	0.009292
Modified Matlab Results	5.853E-03	47.28	25.77	0.003811	0.008433
Percent Change	6.3%	-59.5%	6.2%	9.0%	-9.2%

Figure 4.9. Best value found with the Matlab FA on the five benchmark functions

4.3 – Java results and comparisons

In order to better understand the FA implementation and code details, the FA was also implemented in Java. After testing out various configurations and possible changes, the implementation remained the same as the one in Matlab. This Java version of the FA was used to compare again the PSO algorithm results contained in literature [9]. The FA was run with generic parameters of $\alpha = 0.2$, $\beta = 0.2$, $\gamma = 0.8$, and $\delta = 0.97$ for the first testing, and compared against an optimized constriction factor PSO with $V_{max} = X_{max}$ and generic parameters of $\chi = 0.729$ and $\varphi_1 = \varphi_2 = 2.05$. The number of both fireflies and particles used was 30. Further details and full results are available in *Performance enhancement and hardware implementation of particle swarm optimization*, [9]. Error thresholds used to determine when a global optima has been reached are listed in figure 4.6. Figure 4.11 at the end of the section shows the comparison between PSO and the FA, using the number of generations as a metric. It does not make sense to use time as a metric for comparison in this case, as the systems the tests

were run on are different. A difference between the two algorithms is that the maximum number of generations to run was set at 5000 for the PSO instead of the 500 for the FA, but that does not affect the results, as any run that does not find the global minimum will not be counted towards the average number of iterations taken. Figure 4.11 shows that in every case, the FA outperforms the PSO, when each is used with generic parameters. The standard deviations are shown as error bars for all but the generic parameter PSO results, since the standard deviation for the PSO was extremely high in most cases. Again the standard deviation is only calculated on results that actually found the global minimum. The fact that the standard deviation is significantly lower for the FA shows its reliability in finding the global optimum. When dealing with metaheuristics, reliability is a big issue, since they generally cannot guarantee that they will find the global optimum within any amount of time.

The generic parameters have been shown to work well, but as every problem is different, it makes sense to adjust them to suit the particular problem at hand. Figure 4.11 also shows the results of manually tuning the parameters, for both the PSO and the FA. The optimal parameters that were used for the FA, and which were found through trial and error, are shown in figure 4.10. The results show that in every case, the FA again outperforms the PSO on the benchmark functions. Improvements range from 79-98% fewer generations required to reach the desired result with the tuned algorithms. The biggest improvement was seen in Schaffer's *f6* function, which was also the only function with two parameters. So for all tested functions, and particularly the low-dimensional one, the FA is much more efficient than PSO.

Not only is the efficiency of the algorithm important, but so is the accuracy. Figure 4.12 shows the percent failure of the two algorithms, with both the generic parameter and tuned parameter results. The only function that the FA did not find the global minimum 100% of the

time was the Rosenbrock function. However, once the parameters were tuned, the FA achieved 100% accuracy on all functions, whereas the PSO failed 10-20% on three of the five functions. This shows that with some careful tuning, the FA can be not only extremely efficient, but also very accurate compared to the PSO. Finding the global optimum reliably is again a huge issue with metaheuristics, and the fact that the FA can experimentally achieve 100% accuracy on all five benchmark functions tested shows the potential for this algorithm.

Function	α	β	γ	δ
Sphere	0.2	0.3	0.8	0.89
Rosenbrock	0.02	0.2	0.9	0.9
Rastrigin	0.1	0.5	0.8	0.8
Griewank	0.2	0.2	0.8	0.89
Schaffer's f6	0.6	0.5	0.8	0.3

Figure 4.10. Parameters used for each of the benchmark functions with the FA

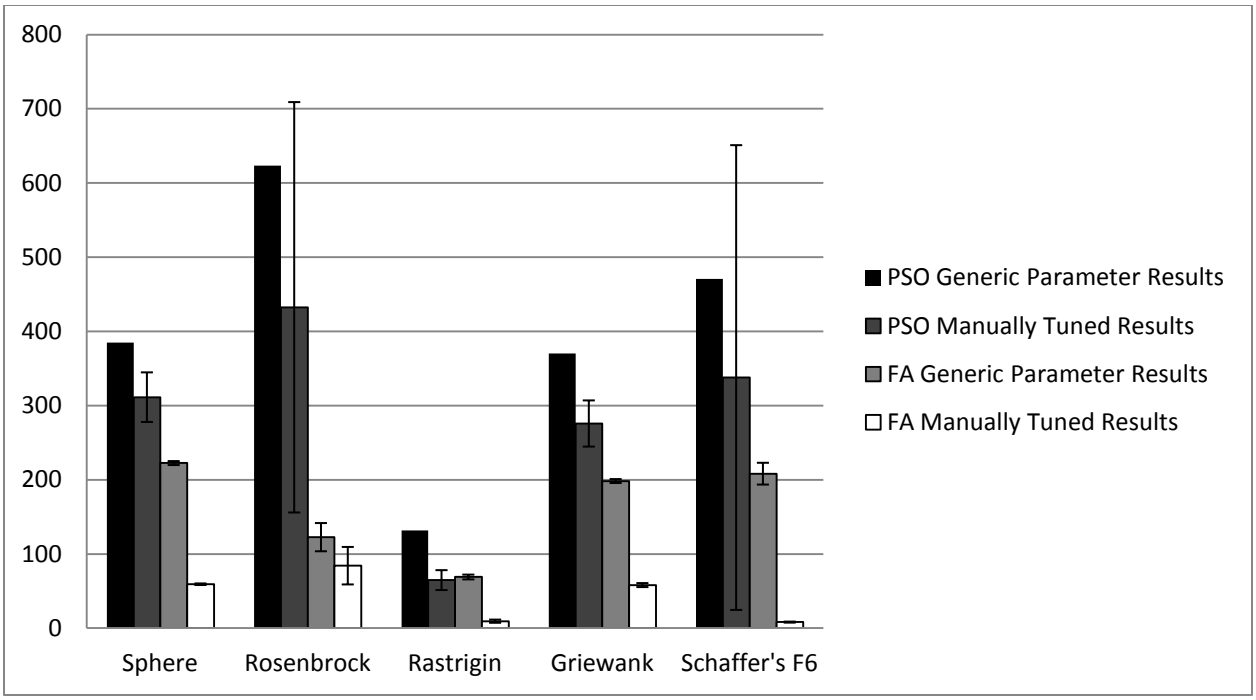


Figure 4.11. Number of generations to find the global minimum of the five benchmark functions

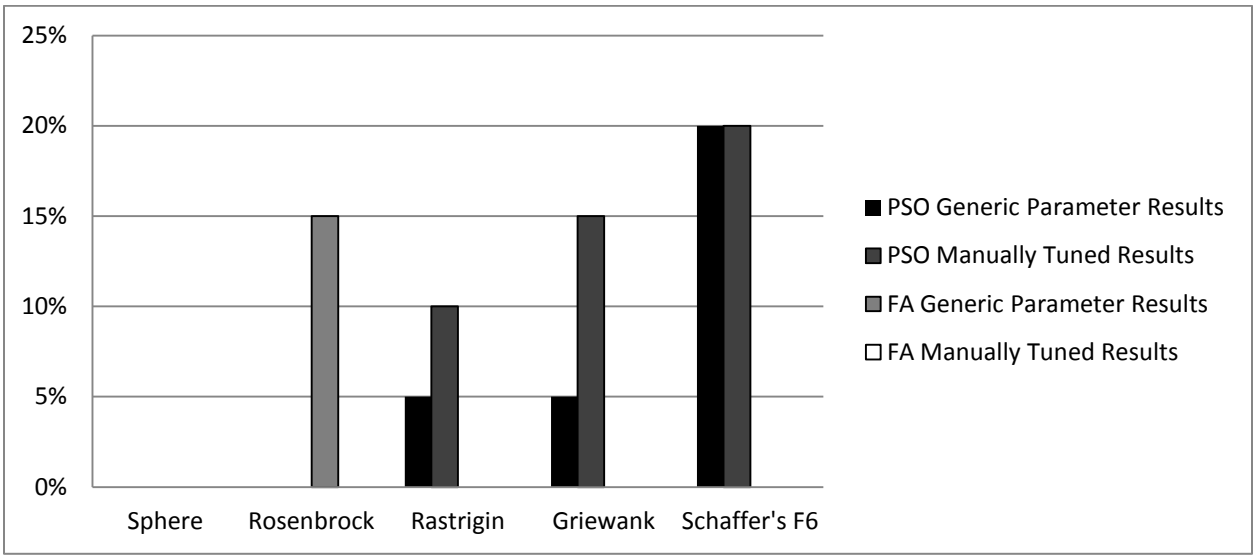


Figure 4.12. Percent of executions that do not find the global minimum

Run #	Sphere	Rosenbrock	Rastrigin	Griewank	Schaffer's F6
1	224	116	68	198	214
2	227	118	63	199	244
3	224	147	77	196	218
4	219	114	65	198	208
5	219	500	68	201	222
6	220	114	67	201	228
7	222	109	67	197	194
8	223	171	69	198	182
9	221	115	69	202	195
10	222	114	70	198	206
11	222	116	65	200	223
12	226	111	71	200	190
13	222	122	70	198	210
14	225	112	70	194	204
15	219	500	72	193	209
16	221	500	75	201	204
17	226	115	68	198	207
18	227	115	67	201	206
19	223	165	70	193	213
20	219	112	69	200	188

Figure 4.13. Results of the FA with Generic Parameters

Run #	Sphere	Rosenbrock	Rastrigin	Griewank	Schaffer's F6
1	59	177	11	63	8
2	60	72	9	58	8
3	59	76	7	56	8
4	59	75	8	55	8
5	59	74	7	62	9
6	61	93	13	54	8
7	59	89	8	61	9
8	59	66	9	58	8
9	60	73	10	59	6
10	61	119	9	56	8
11	56	82	8	62	9
12	58	73	9	54	8
13	60	67	10	57	9
14	59	79	8	62	8
15	59	70	9	57	8
16	59	98	14	58	8
17	60	71	9	57	9
18	59	69	12	59	9
19	60	75	13	57	8
20	59	87	6	57	8

Figure 4.14. Results of the FA with Tunes Parameters

5. Scalability Test: Emission Source Localization

Tracking and localization of an emission's source is an important safety concern. Whether an accident causes hazardous material to leak into the atmosphere, a terrorist strike causes radiation dispersion, or a military encounter requires locating the enemy, finding the source of the emission is critical. The US Department of Energy's Oak Ridge National Laboratory (ORNL) has seen this need and is developing a network called SensorNet for the detection and assessment of chemical, biological, radiological, nuclear, and explosive threats [13]. SensorNet is largely based on a model of many small, distributed sensors for gathering data, and high level nodes for data processing. These data processing nodes could use an algorithm like the FA for finding the location of the source of a chemical leak, a bio-hazard, or even radiation.

5.1 – Modeling the Emission Source Localization Problem

In order to find the source of something spreading through the atmosphere, water, ground, etc, there must be sensors capable of detecting the spread and a model of propagation. Considering the example of a chemical leak through the atmosphere, sensors could be deployed that can detect the level of chemical in their local area. Using more sensors spread around can give a more detailed picture of the situation. For the model of propagation, the intensity at any point away from the source can be assumed to follow the inverse square law, just like the intensity of light from a firefly [14]. This model ignores possible environmental factors such as air currents, but is a good starting point to prove the effectiveness of the FA in a large scale problem.

With this model, data from sensors can be used to find a source. It will be assumed that there is only one source, and that the sensors will be scattered randomly around an area known to contain the source. These sensors will also know their own locations, either from GPS or some

other sensor localization method. The data from these sensors will be sent to a data processing node, which can then determine the location of the source. For the purposes of this examination, the sensors will be simulated, and since sensor readings are inherently noisy, additive Gaussian noise will be added to the sensors' readings. With the added noise, and since the propagation is assumed to follow the inverse square law, a sensor's reading can be calculated by:

$$q_j = \frac{Q_0}{d_j^2}(1 + \sigma G) \tag{5.1}$$

where d_j is the distance to sensor j from the source, Q_0 is the source intensity, σ is the desired standard deviation of the noise, and G is a random number picked from a Gaussian distribution of mean 0 and standard deviation 1. The source is located at (x_0, y_0), which will be determined randomly before the sensors are placed. Figure 5.2 shows the intensity distribution for a source located at (-70,80) and intensity 1000, with no noise added. Figures 5.3-5.5 show examples of the same distribution but with various levels of noise added. These show how rugged the search space can become when noise is added. Within this fixed rectangular search space the sensors will be placed randomly. An example of the placement of sensors and the source location can be seen in figure 5.6.

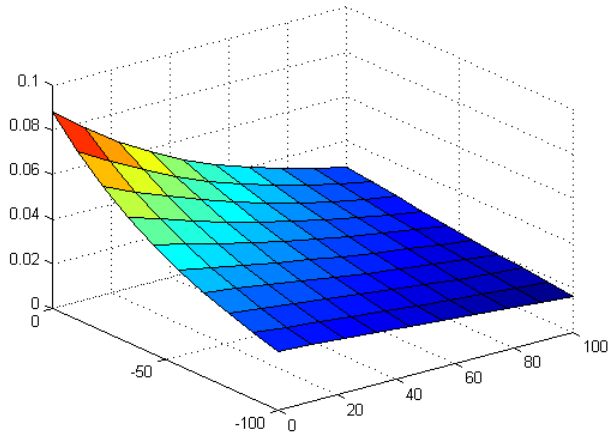


Figure 5.2. 0% Standard Deviation

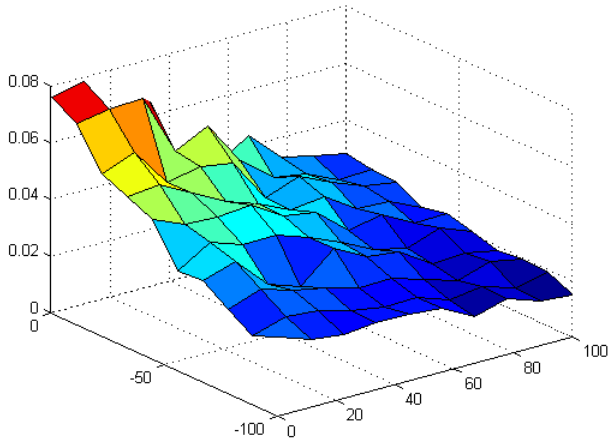


Figure 5.3. 10% Standard Deviation

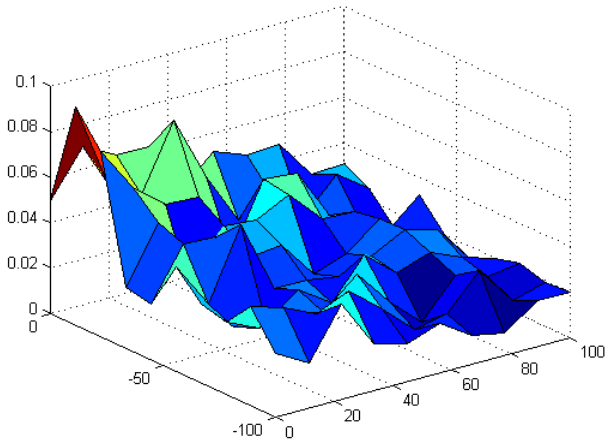


Figure 5.4. 30% Standard Deviation

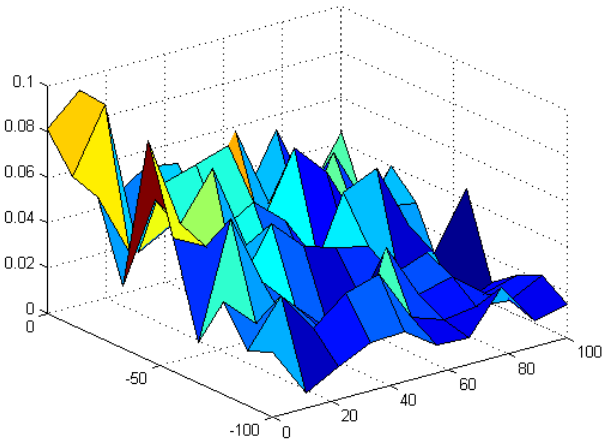


Figure 5.5. 50% Standard Deviation

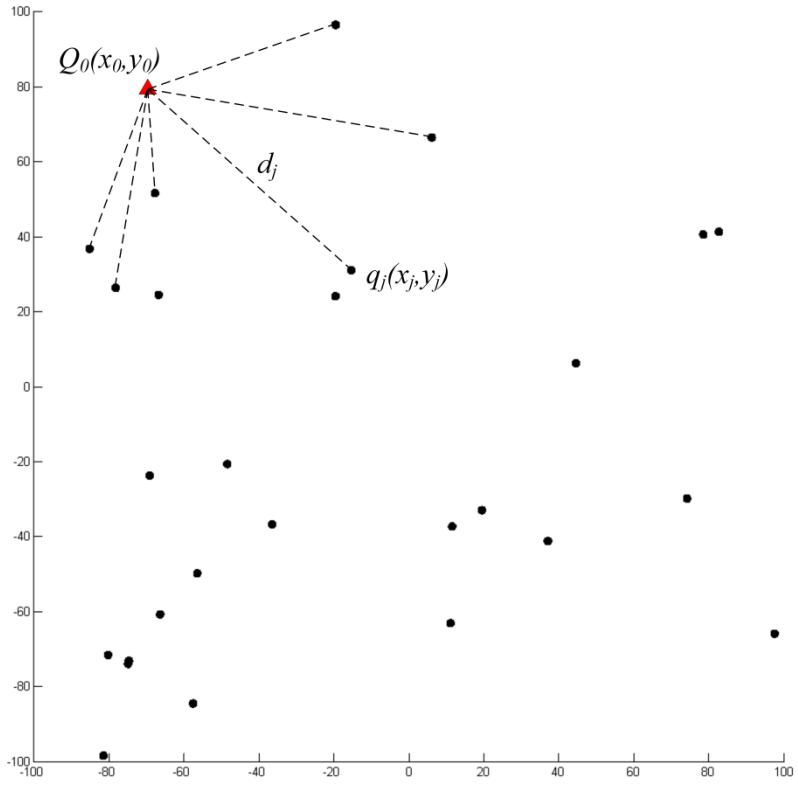


Figure 5.6. Randomly distributed sensors with the source Q_0 located at (x_0, y_0) [9]

Then in order to formulate an objective function, the goal becomes to determine the optimal Q_0 and (x_0, y_0) that will provide the closest match between the propagation model and the actual sensor readings. The error between the two can be taken as

$$dQ_j = Q_0 - \left(q_j d_j^2 + w_j Q_0\right) \tag{5.7}$$

where q_j is the intensity reported by sensor j, d_j is the distance between sensor j and the source location (x_0, y_0) as guessed by a firefly, and w_j is a weighting factor used to correct for the error in the sensor readings. The term $w_j Q_0$ adds a percentage of the source intensity to the sensor's reading, which allows a firefly to pick a value of w_j that will hopefully counter any error in the sensor reading.

Now that the error in an estimate from a single sensor's point of view can be calculated, the least squares method can be used to find the total error of an estimate [14]. The sum of all the squared errors from each sensor will be the objective function, given as [9]:

$$f(Q_0, x_0, y_0, W) = \sum_{j=1}^{N_s} dQ_j^2 = \sum_{j=1}^{N_s} \left[Q_0 - \left(q_j d_j^2 + w_j Q_0\right)\right]^2 \quad 5.8$$

where N_s is the number of sensors, and W is the vector of each w_j for every sensor. So a firefly's position consists of an estimated Q_0, x_0, y_0, and a w_j for every sensor; thus every sensor adds a dimension to the problem.

5.2 – Test Results

The aim of the tests with the FA will be to compare to the PSO results in [9]. Thus, eight different configurations were used to test various scales of the problem, the results of which are given in figures 5.9-5.16. For each configuration, five tests were performed, with the standard deviation of added noise varied from 0% to 50%. The tests were run 20 times each, and the results shown are the averages. Each test also has its own set of FA parameters. The goal of the parameter choice was to find a set that produced the best results in the least amount of time. The biggest parameter affecting time is the number of fireflies, and so is a good place to start when trying to reduce time taken. Reducing the number of fireflies reduces the number of search agents though, so it is often difficult to still produce accurate results. Once the minimal number

of fireflies is determined, the next biggest factor is the number of generations to run. The number of generations also greatly affects the runtime; therefore once again, the minimum amount that still produced accurate results was used. For the larger configurations, changing the β term was necessary. Since β determines the attractiveness of the fireflies, increasing β results in faster movements. In large search spaces this is important, as otherwise it takes significantly longer to reach the same goal. For the smaller configurations, δ was lowered from the standard 0.97, since for a smaller number of generations, the randomness needs to be reduced faster to reach the same level near the end of the run. For α and γ, standard values of 0.2 and 0.8 were used, respectively.

#1	FA				
Noise var. (%)	Location Error (%)	Intensity Error (%)	Execution Time (ms)	Fireflies	10
				Generations	50
0%	1.04	0.47	3.40	Sensors	27
10%	2.63	-0.71	3.45	Range	[-50,50]
20%	3.31	-2.82	3.35	Alpha	0.2
30%	3.60	-5.70	3.40	Beta	0.2
40%	4.77	-11.14	3.65	Delta	0.93
50%	5.88	-14.91	3.35	Gamma	0.8

Figure 5.9. Configuration 1

#2	FA				
Noise var. (%)	Location Error (%)	Intensity Error (%)	Execution Time (ms)	Fireflies	10
				Generations	50
0%	1.04	-2.79	3.15	Sensors	27
10%	1.33	0.65	3.75	Range	[-100,100]
20%	2.34	-2.26	3.55	Alpha	0.2
30%	2.57	-4.94	3.30	Beta	0.2
40%	2.76	-1.07	3.35	Delta	0.93
50%	4.76	-6.75	3.15	Gamma	0.8

Figure 5.10. Configuration 2

#3	FA				
Noise var. (%)	Location Error (%)	Intensity Error (%)	Execution Time (ms)	Fireflies	10
				Generations	50
0%	0.47	-0.71	4.80	Sensors	60
10%	0.79	-0.25	4.95	Range	[-200,200]
20%	1.19	-3.61	4.65	Alpha	0.2
30%	1.30	-2.70	4.80	Beta	0.2
40%	1.70	-5.55	4.75	Delta	0.97
50%	1.80	-4.88	4.55	Gamma	0.8

Figure 5.11. Configuration 3

#4	FA				
Noise var. (%)	Location Error (%)	Intensity Error (%)	Execution Time (ms)	Fireflies	10
				Generations	100
0%	0.10	-0.43	8.05	Sensors	60
10%	0.44	-0.84	8.90	Range	[-400,400]
20%	0.88	-3.13	8.00	Alpha	0.2
30%	1.42	-4.54	8.10	Beta	0.2
40%	1.91	-7.07	8.50	Delta	0.97
50%	2.09	-6.01	8.25	Gamma	0.8

Figure 5.12. Configuration 4

#5	FA				
Noise var. (%)	Location Error (%)	Intensity Error (%)	Execution Time (ms)	Fireflies	15
				Generations	100
0%	0.09	-0.02	28.70	Sensors	90
10%	0.29	-0.78	27.50	Range	[-500,500]
20%	0.73	-3.12	28.15	Alpha	0.2
30%	1.18	-2.36	27.50	Beta	0.2
40%	1.52	-5.23	27.25	Delta	0.97
50%	2.04	-3.69	25.30	Gamma	0.8

Figure 5.13. Configuration 5

#6	FA				
Noise var. (%)	Location Error (%)	Intensity Error (%)	Execution Time (ms)	Fireflies	20
				Generations	100
0%	0.06	0.21	239.8	Sensors	500
10%	0.14	1.18	237.5	Range	[-1500,1500]
20%	0.37	1.41	237.9	Alpha	0.2
30%	0.60	-2.31	240.0	Beta	1.0
40%	0.68	1.21	214.3	Delta	0.97
50%	0.84	2.74	265.3	Gamma	0.8

Figure 5.14. Configuration 6

#7	FA				
Noise var. (%)	Location Error (%)	Intensity Error (%)	Execution Time (ms)	Fireflies	25
				Generations	100
0%	0.05	-0.08	777.9	Sensors	1000
10%	0.10	-0.54	773.0	Range	[-2500,2500]
20%	0.36	-1.52	768.6	Alpha	0.2
30%	0.43	-1.46	778.0	Beta	1.0
40%	0.54	-1.51	776.8	Delta	0.97
50%	0.74	-2.43	689.4	Gamma	0.8

Figure 5.15. Configuration 7

#8	FA				
Noise var. (%)	Location Error (%)	Intensity Error (%)	Execution Time (ms)	Fireflies	30
				Generations	100
0%	0.04	0.05	2325	Sensors	2000
10%	0.06	-0.03	2261	Range	[-5000,5000]
20%	0.15	-0.86	2307	Alpha	0.2
30%	0.48	-1.97	2337	Beta	1.0
40%	0.53	-1.65	2268	Delta	0.97
50%	0.73	-2.30	2088	Gamma	0.8

Figure 5.16. Configuration 8

The results of the PSO applied to emission source localization in [9] are only available in graphs for test configurations 1, 3, and 5, so the exact results are hard to obtain. However, a general idea of how the results between the two algorithms compare can be given. Results are not given for configurations 2 or 4, so no comparison can be made. For configuration 1, the localization error obtained by the FA is worse for 0-20% noise, but better for 30-50% noise. The intensity error is better under all noise conditions, and the time required is about 13 times lower. Note that much of the time speedup is due to being run on a faster processor, but it is still useful to compare relative speedups between configurations.

In configuration 3, the average localization error is higher for 0-30% noise, but lower for 40 and 50%. The average intensity error is about the same for all noise levels, and the execution time is about 180 times faster. With configuration 5, the FA obtains about the same localization and intensity error for 0-20% noise, but for 30-50% noise the FA obtains better results for both. The speed for this configuration was about 50 times faster than the PSO. Thus, in the smaller configurations 1 through 5, for low noise levels, the PSO seems to find better results, but not always as efficiently. For higher noise levels, the FA wins out in terms of accuracy of the results, and except for the smallest configuration is likely much faster.

Configurations 6-8 provide a better comparison, as tabular data is available. The PSO and FA results are compared in figures 5.17-5.20. Configuration 6 and 7 show uniformly worse performance for the FA, but in configuration 8 the intensity error is lower for the FA on all but 30% noise. The time speedups, however, are about 34, 23, and 18 times faster, respectively. These times may or may not actually be faster, given that the FA was run on a faster processor, but the difference between speedups of different configurations shows that the FA runs faster on small to medium scale problems compared to the PSO. For the smaller scale problems, the FA

performs much better under highly noisy conditions, and in fact gets nearly uniform results across all noise levels, compared to the PSO. Since the point of using either the FA or PSO is speed rather than extreme accuracy, the FA seems to perform comparably well to the PSO.

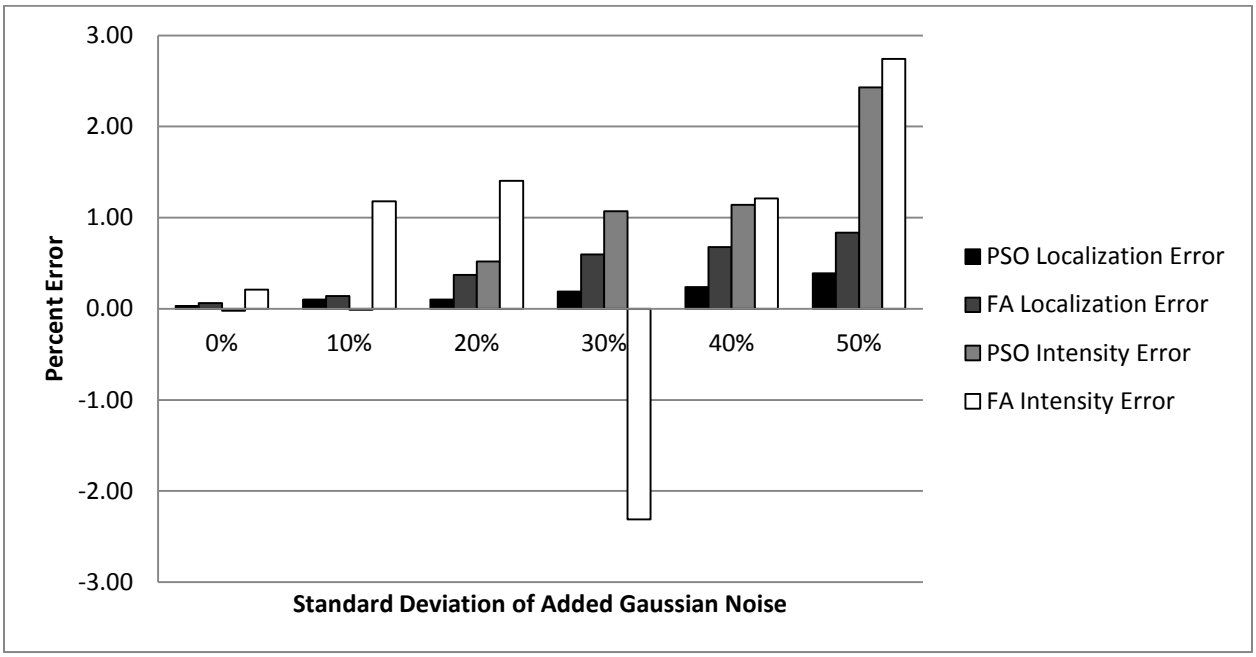


Figure 5.17. PSO and FA on Emission Source Localization Configuration 6

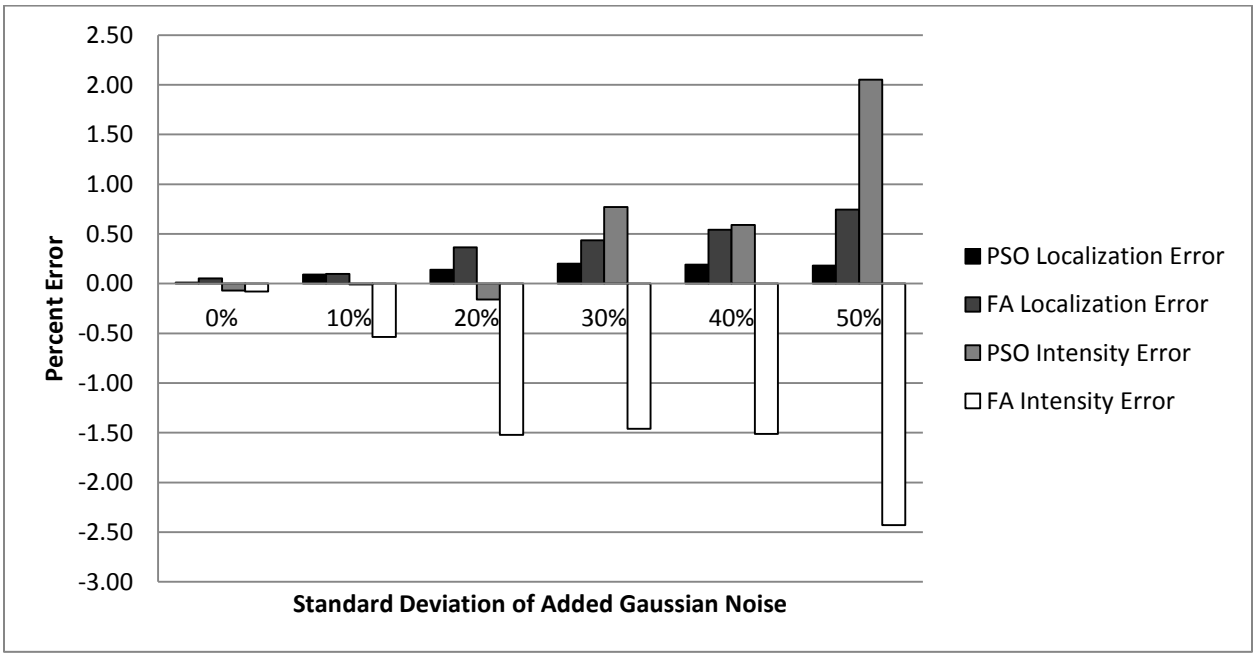


Figure 5.18. PSO and FA on Emission Source Localization Configuration 7

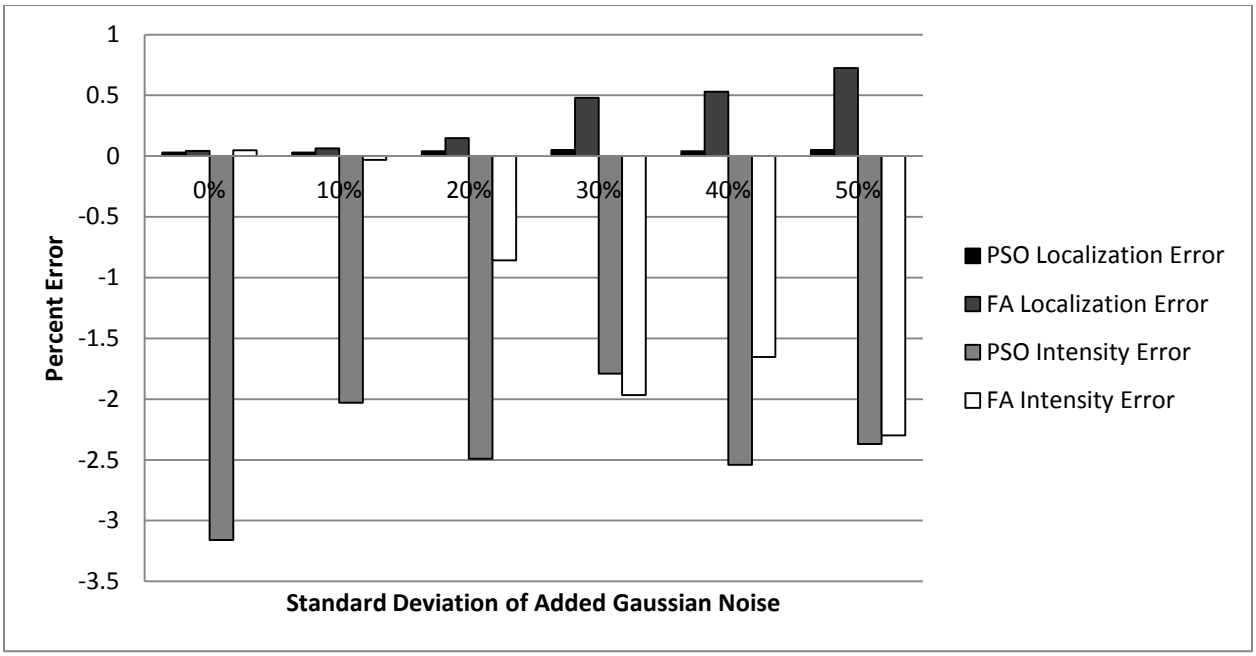


Figure 5.19. PSO and FA on Emission Source Localization Configuration 8

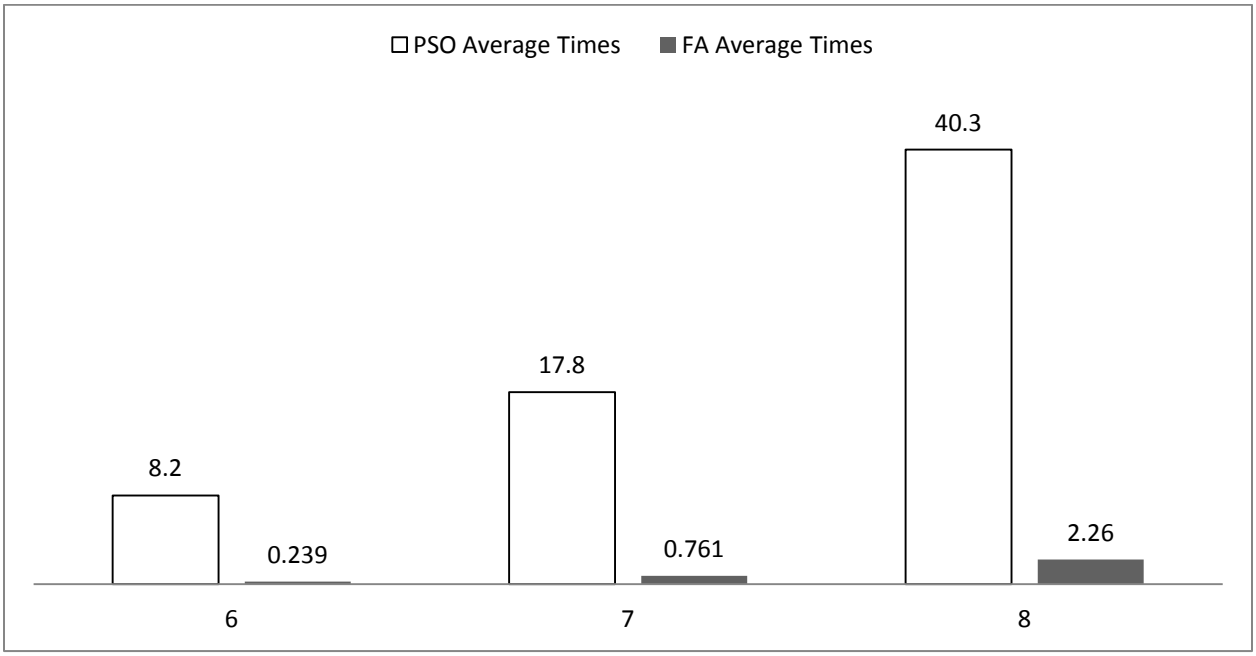


Figure 5.20. PSO and FA Execution Times for Configurations 6-8

6. The Next Step in Speed: A Direct Hardware Implementation

In the last 15 years, major advancements have been made in the realm of hardware design. Miniaturization has allowed more and more to be squeezed onto a single chip. What used to require long manufacturing lead times or massive circuits with numerous chips can now be programmed onto a single chip in a matter of seconds. These microchips, called Field Programmable Gate Arrays (FPGA), are used to implement hardware designs.

6.1 – Advantages of Custom Hardware

In software, there are a great many possibilities, but in the end a programmer is limited by the capabilities of the processor a program is running on. The processor has a limited set of elementary operations that can be performed, that are designed to run any potential program. This makes processors great for running any number of different applications, but not for any one in particular. If a processor were to be designed for a single task, it could be made much more optimal. This is possible on an FPGA, where the functionality can be completely customized. Whereas software designs end up compiled to code that will run on a given processor, a hardware design on an FPGA can be designed to provide any functionality required. This includes running a task in parallel. Processors with multiple cores do allow a certain level of parallelism, but even the elementary operations can be parallelized in hardware, providing an increased level of parallelism.

This parallel operation is important, as modern processors operate at much higher clock speeds than FPGA's. The clock speed determines how fast a system will run, and in a processor is typically a few gigahertz, while an FPGA typically operates with a maximum clock speed of a few hundred megahertz or even much less. The clock speed is not the only important measure of how fast a system runs, however. What matters is how much is accomplished between each

clock cycle. A processor's clock speed has to be slow enough that even its slowest task can be performed before the next clock cycle. An FPGA has the same concern, but if a certain process is way slower than the rest of the design, that process can be split up, into quicker, smaller chunks. Also, a lower clock speed means there will be lower power usage. In a well designed system, even with a lower clock speed, an FPGA can outperform a processor at a specific task. The key for that is being well designed, and hardware design times are generally significantly longer than for software.

Another reason for designing a hardware architecture is for use of solving the emission source localization problem in the field. In some applications, for example military operations abroad, the source of an emission may be required to be found where there are no computers available, but where a small package could be brought. A small, embedded system would typically be used for this scenario, but small the small processors used in embedded systems would suffer from a much increased runtime of the algorithm. But custom hardware would eliminate that problem, as well as using less power, and thus reducing battery requirements.

6.2 – Datapath

The proposed architecture is modular in nature. The first consideration was to be able to make the unit that calculates the fitness function modifiable without changing anything else in the design. This means that the architecture will work to solve any problem desired, with only the need to design hardware to evaluate the fitness function. The next modular aspect is the idea of an update unit. These units store a single firefly's data, and are capable of moving a firefly towards another and updating the intensity of the firefly. In the design shown in figure 6.1, two update units are used. The idea behind the operation of these units is to parallelize the firefly comparisons and movements. In the case of figure 6.1, two fireflies would be loaded into the

two update units from one of the two memories. Then the intensities of those fireflies would be simultaneously compared against the first firefly in the swarm. If either needs to move, its position will be updated, and then the second firefly in the swarm will be compared against. This continues until all fireflies have been compared against and any necessary movements have been made. Then the two fireflies in the update units will have their intensities re-calculated, and will be stored in the opposite memory from the one being read from. The reason for using two memories is to separate the firefly generations from each other, just like is done is software. After the first two fireflies are finished, the next two are loaded into the update units, and so on, until all fireflies have been updated to the new generation.

Since two fireflies can be loaded at once with two update units, the number of loading periods is halved, and thus the time is approximately halved. The only reason it is not fully halved is because of the case where one firefly in an update unit needs to move towards the firefly being compared against but the other does not. If there were only one update unit, the firefly that did not need to move could move on and compare itself against the next firefly in the swarm. Therefore the actual percentage speedup of adding more units would have to be experimentally tested. But depending on hardware resources, more update units could be added to further speed up the algorithm. In a very large chip or with a small number of fireflies, the number of update units could even be as large as the number of fireflies, which would mean completely parallel operation. In this case the algorithm is reduced to a simple loop where all the fireflies are loaded and then compared against every firefly, and all movements happen in parallel.

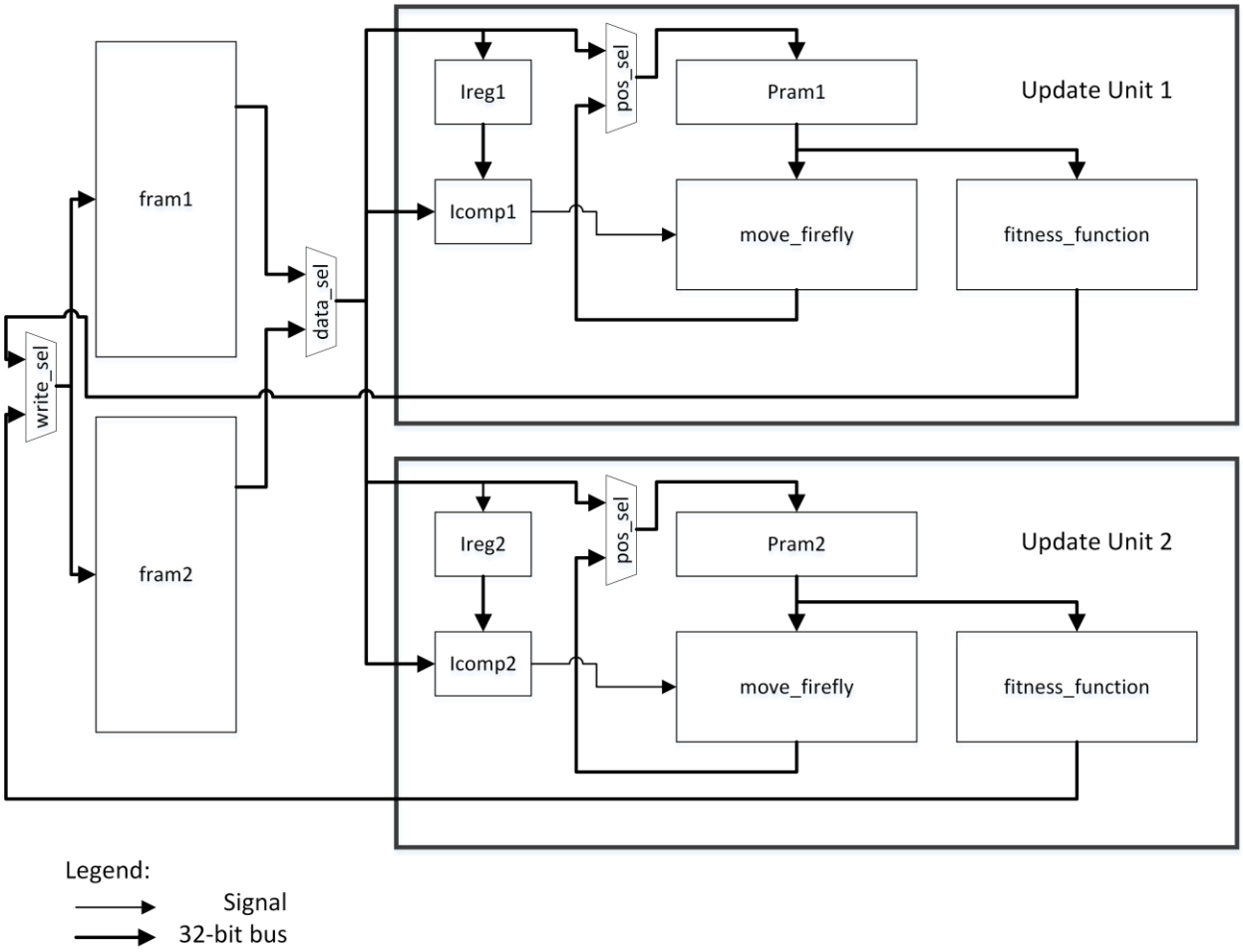


Figure 6.1. Hardware datapath

6.3 – Control Unit

Figure 6.2 below shows the control unit logic for the algorithm as a state diagram. The controller assumes that fram1 is initially loaded with fireflies in random positions. For the initialization state, the ram_sel signal is set to 0, indicating that fireflies will be read out of fram1, and new data stored into fram2. The firefly index n and the generation count gen are reset to 0. Then the load state is entered. In the load state, the intensities for all the fireflies in update units are loaded, and then all of the positions are loaded. The swarm index i is set to 0, and then the compare state is entered. In this state every update unit compares its firefly to the one in memory at index i. Then the controller waits for any position update to finish. Once the swarm

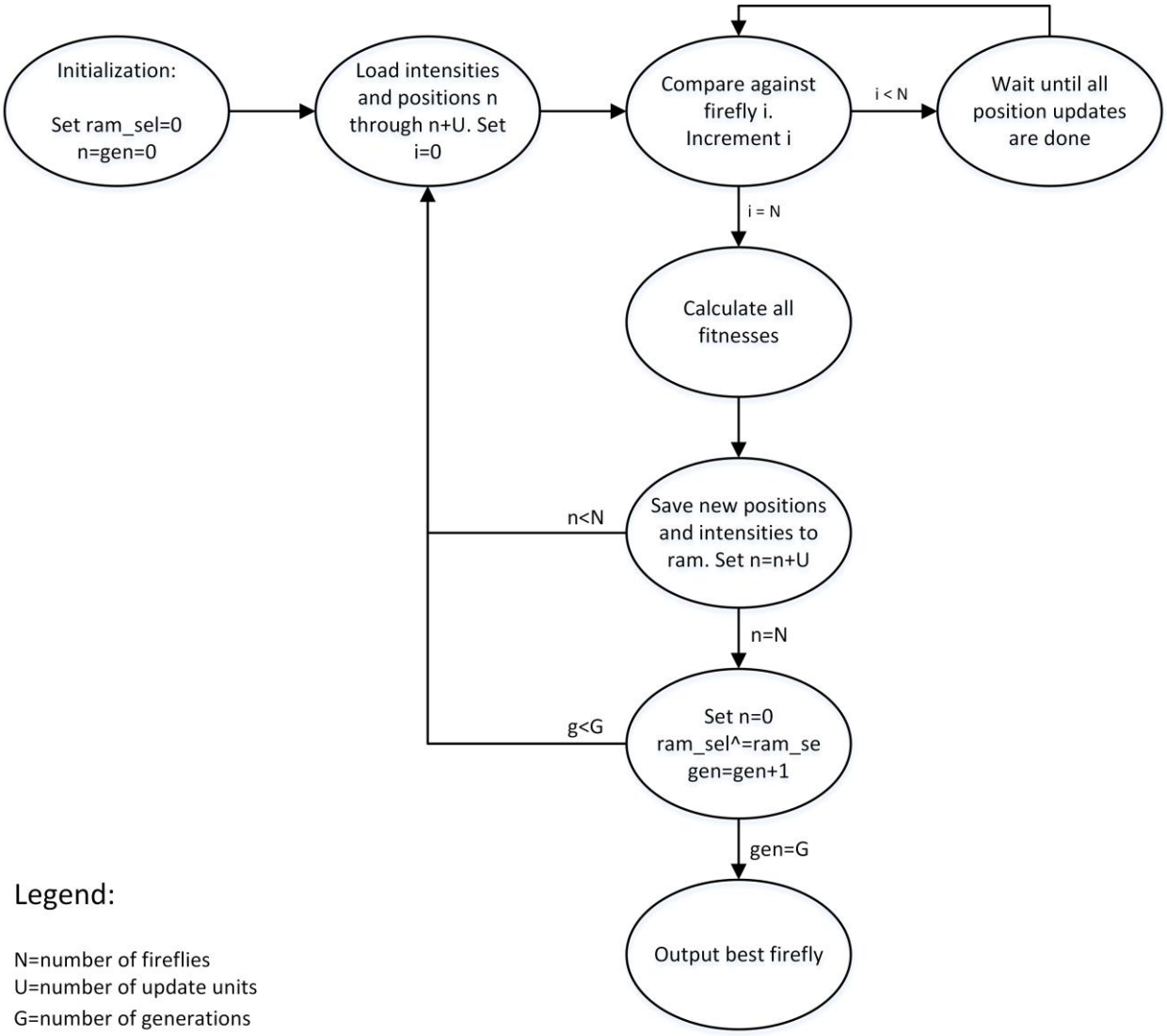


Figure 6.2. Control unit

index i has reached the number of fireflies N, and all position updates have finished, then the fitness update state is entered. Here every update unit re-calculates its firefly's intensity. Once this is done, all the new positions and intensities for each update unit are saved to the opposite ram selected by ram_sel, and n is set to n+U. If the firely index n is still less than the number of fireflies N, then the load state is re-entered, and the previous steps repeated. If n=N, then n is set to 0, ram_sel is flipped such that if it was 0 it is now 1 and if it was 1 it is now 0, and finally the generation counter gen is incremented by 1. If gen is less than the desired number of generations

G, the load state is re-entered and the previous steps are repeated. If gen=G, then the algorithm is done, and the best firefly is output as the result.

7. Conclusion

The firefly algorithm is one of the latest artificial intelligence algorithms developed. Inspired by the flashing of fireflies, it gets its inspiration from nature, like many of the other metaheuristic algorithms. The social aspect of fireflies provides an efficient means of traversing a search space, and avoiding any local optima. The fogginess of the medium, the amount of attraction between fireflies, and the amount of randomness all play a role, and it is important to adjust them properly to the problem at hand. With the addition of randomness reduction and scaling of the distance, the FA can be adapted even further to improve results. All of the results obtained throughout this work show that the FA is a very efficient algorithm. In the benchmark functions, we have seen the FA achieves better results than the PSO, and does so in a small fraction of the time. The emission source localization proves quite the challenge, and the FA actually outperforms the PSO in noisy situations. It appears that the FA is the superior algorithm when it comes to problems with many local optima. And when time is of the essence, the FA returns results extremely fast.

For future research, the FA can be fully implemented in hardware. This will provide for a reduction in time and energy used to run the algorithm. A parallel implementation could even further improve speed. As for the algorithm itself, it would be worth investigating possible improvements, such as hybridization with other algorithms. The addition of the randomness reduction concept from simulated annealing helped improve results, and this could be investigated further, such as increasing the randomness at various stages in order to escape local optima. Other such algorithms likely have concepts worth exploring and adapting to the FA.

For the emission source localization problem, an interesting possibility would be to incorporate sensor data from multiple time periods. The processor node could receive sensor data at one time, and a little later receive updated data. A way to use this extra data intelligently could possibly be developed to further improve results.

References

[1] P. M. Pardalos and H. E. Romeijn, Eds., Handbook of Global Optimization, vol. 2, Kluwer Academic Publishers, 2002.

[2] S. Luke, Essentials of Metaheuristics, Lulu, 2009.

[3] X. S. Yang, Nature-Inspired Metaheuristic Algorithms, Luniver Press, 2008.

[4] J. H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, Ann Arbor: University of Michigan Press, 1975.

[5] D. Whitley, "A Genetic Algorithm Tutorial," *Statistics and Computing*, vol. 4, no. 2, pp. 65-85, 1994.

[6] X. S. Yang, "Firefly Algorithms for Multimodal Optimization," in *Stochastic algorithms: foundations and applications*, 2009.

[7] J. Robinson, "Particle Swarm, Genetic Algorithm, and their Hybrids: Optimization of a Profiled Corrugated Horn Antenna," in *Antennas and Propagation Society International Symposium*, 2002.

[8] R. Hassan, B. Cohanim and O. Weck, "A Comparison of Particle Swarm Optimization and the Genetic Algorithm," in *Proceedings of the 1st AIAA Multidisciplinary Design Optimization Specialist Conference*, 2005.

[9] G. S. Tewolde, *Performance Enhancement and Hardware Implementation of Particle Swarm Optimization*, 2008.

[10] S. M. Lewis and C. K. Cratsley, "Flash Signal Evolution, Mate Choice, and Predation in Fireflies," *Annu. Rev. Entomol.*, vol. 53, pp. 293-321, 2008.

[11] X. S. Yang, Engineering optimization: An introduction with metaheuristic applications, Hoboken, NJ: John Wiley & Sons., 2010.

[12] X. S. Yang, 2011. [Online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/29693-firefly-algorithm. [Accessed February 2013].

[13] ORNL, "SensorNet: A System of Systems to Provide Nationwide Detection and Assesment of Chemical, Biological, Radiological, Nuclear, and Explosive (CBRNE) threats," 2004. [Online]. Available: http://computing.ornl.gov/cse_home/datasystems/sensornet.pdf. [Accessed February 2013].

[14] M. P. Michaelides and C. G. Panayiotou, "Plume Source Position Estimation Using Sensor Networks," in *Proceedings of the 2005 IEEE International Symposium on Intelligent Control, Mediterranean Conference on Control and Automation*, Limassol, Cyprus, 2005.