Graduate School Form 30
Updated 1/15/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Joshua Reynolds

Entitled

PARTICLE SWARM OPTIMIZATION APPLIED TO REAL-TIME ASSET ALLOCATION

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

Lauren Christopher
Chair

Russell Eberhart

Paul Salama

Brian King

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Lauren Christopher

Approved by: Brian King 4/27/2015

Head of the Departmental Graduate Program Date

PARTICLE SWARM OPTIMIZATION APPLIED TO REAL-TIME ASSET ALLOCATION

A Thesis

Submitted to the Faculty

of

Purdue University

by

Joshua Reynolds

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2015

Purdue University

Indianapolis, Indiana

"Whether therefore you eat, or drink, or whatsoever you do,
do all to the glory of God." 1 Cor. 10:31
This thesis is dedicated to the glory of God.

ACKNOWLEDGMENTS

I would like to thank my major professor, Dr. Lauren Christopher, for her support and guidance in writing this thesis. I also want to thank the rest of my thesis committee, Dr. Brian King, Dr. Paul Salama, and Dr. Russell Eberhart. Special thanks to Brandon Shafer and Dr. Eberhart and Phoenix Data Corporation for their expertise in particle swarm optimization and Patrick Shaffer at NAVSEA for his feedback and suggestions for development of this research.

TABLE OF CONTENTS

Page

LIST OF FIGURES vi

LIST OF TABLES vii

ABSTRACT viii

1 INTRODUCTION 1
1.1 Overview and Problem Statement 1
1.2 Research Starting Point 2
1.3 New Approach 3

2 PSO AND FITNESS FUNCTION 5
2.1 Fitness Function Design 5
2.1.1 Fitness Priority Component 5
2.1.2 Fitness Power Component 6
2.1.3 Fitness Spread Component 7
2.1.4 Fitness Distance Component 8
2.1.5 Keep Away Penalty 9
2.2 PSO Parameters 9

3 IMPLEMENTATION 12
3.1 PSO Implementation 12
3.2 GUI Implementation 16
3.3 Platform Considerations 17

4 ANALYSIS 18
4.1 Analysis of Fourth Fitness Component. 18
4.2 Selection of Fitness Component Weights 20
4.3 Repeatability Analysis 20
4.4 Run-time Analysis 23

5 CONCLUSIONS 25
5.1 Summary 25
5.2 Future work 25

REFERENCES 27

APPENDIX 28
A.1 GUI Manual 28
A.2 Build Instructions 32

Page

A.2.1 PSO Back-end 32
A.2.2 GUI Front-end 33

VITA 34

LIST OF FIGURES

Figure		Page
1.1	Previous Research	3
3.1	Doxygen Call Graph for Fitness Function	13
3.2	Top Level Fitness Function	15
3.3	New GUI	16
4.1	Sample PSO Run #1	19
4.2	Sample PSO Run #2	19
4.3	Repeatability Test Case	22
A.1	GUI Showing Mouse Interaction	28
A.2	Additional GUI Settings	30

LIST OF TABLES

Table Page

2.1 Test PSO Parameters 10

2.2 Performance PSO Parameters 10

4.1 Fitness Means and Variances 21

4.2 Run-time Results 24

A.1 Initialization Options 31

ABSTRACT

Reynolds, Joshua M.S.E.C.E., Purdue University, May 2015. Particle Swarm Optimization Applied to Real-time Asset Allocation. Major Professor: Lauren Christopher.

Particle Swam Optimization (PSO) is especially useful for rapid optimization of problems involving multiple objectives and constraints in dynamic environments. It regularly and substantially outperforms other algorithms in benchmark tests. This paper describes research leading to the application of PSO to the autonomous asset management problem in electronic warfare. The PSO speed provides fast optimization of frequency allocations for receivers and jammers in highly complex and dynamic environments. The key contribution is the simultaneous optimization of the frequency allocations, signal priority, signal strength, and the spatial locations of the assets. The fitness function takes into account the assets' locations in 2 dimensions, maximizing their spatial distribution while maintaining allocations based on signal priority and power. The fast speed of the optimization enables rapid responses to changing conditions in these complex signal environments, which can have real-time battlefield impact. Results optimizing receiver frequencies and locations in 2 dimensions have been successful. Current run-times are between 450ms (3 receivers, 30 transmitters) and 1100ms (7 receivers, 50 transmitters) on a single-threaded x86 based PC. Run-times can be substantially decreased by an order of magnitude when smaller swarm populations and smart swarm termination methods are used, however a trade off exists between run-time and repeatability of solutions. The results of the research on the PSO parameters and fitness function for this problem are demonstrated.

1. INTRODUCTION

1.1 Overview and Problem Statement

Particle Swarm Optimization (PSO) is an exciting computational tool for optimization applications in scheduling and logistics, hardware development, artificial neural networks, and many other areas. Examples include optimization of mission planning, optimization of allocation of electronic warfare resources, medical analysis and diagnosis, electric utility system load stabilization, and product mix optimization. PSO is exciting because of the ease and speed with applications can be developed (often weeks or months instead of years) and the performance of the solutions, which is often better and orders of magnitude faster than traditional solutions for complex or computationally-intensive problems.

PSO is an evolutionary computation technique developed in 1995 by James Kennedy and Russell Eberhart [1] [2]; with a text on the subject by Eberhart, Simpson, and Dobbins in 1996 [3] and by Eberhart and Shi in 1998 [4]. PSO methods were included in a formal textbook by Eberhart in 2007 [5]. At the time of the writing of this paper, PSO has been around for two decades; it is being researched and utilized in over 30 countries.

PSO has already been applied to some problems in real-time allocation. For weapons allocation for defensive purposes as seen in [6], PSO was shown successful for small scale problems. In this thesis, the application of PSO to real-time asset allocation in the area of electronic warfare (EW) is explored. This is follow-on work to a project done for the Expeditionary Electronic Warfare Division, Spectrum Warfare Systems Department, at the Naval Surface Warfare Center (NSWC) Crane [7]. PSO was used in that project to allocate electronic warfare resources in the frequency spectrum in a rapidly changing environment on a near-real-time basis.

The goal of this research was to extend the previous work to optimize the resources simultaneously in 3D space and across the frequency spectrum. This differs from the previous work in [7] which assumed that all of the assets were co-located at a single point in space. Each RF receiver has a certain programmable bandwidth and maximum allowable input power. The receivers must be allocated to a number of transmitters, where each transmitter has a priority, power, and location. In addition, the transmitters are placed in a defined area, simulating the EW battlefield. It is desirable to receive signals with the highest priority and power while not overloading the RF front end of any receiver. Furthermore, it is advantageous for the receivers to be spread out in 3D space. A viable solution should not compromise frequency allocations that were obtainable with the previous work while at the same time achieving spatial optimization.

In an operational scenario involving the allocation of multiple receiver resources against a suite of dozens of signals with varying powers and priorities in the past required a state-of-the-art system (available to the Navy). This system took nearly two hours to calculate the optimal receiver center frequencies. Of course, this is clearly not useful in an operational environment. The reported PSO solution optimally allocates resources in under one second.

The changes made to the optimizer are described in Section 2. Section 3 describes the implementation details. Section 4 details our analysis and Section 5 summarizes and outlines potential future work.

1.2 Research Starting Point

Phoenix Data Corp. provided two pieces of software source code that were used as a starting point for this research. First, we received C++ implementation of generic PSO algorithm minus the fitness function. Secondly, we were provided with a C# to implementation of a PSO optimizer finding the center frequency allocation of assets (receivers). The input to this optimizer consisted of a collection of narrow

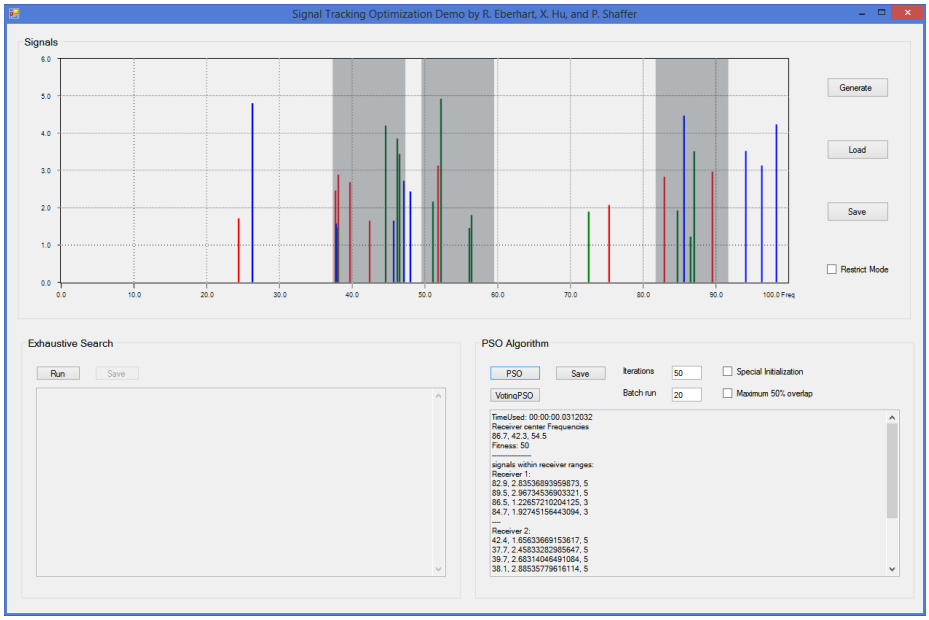

Figure 1.1. Previous Research

band signals, their frequencies, priorities, and powers. Frequencies ranged on an arbitrary scale of 0 to 100 units with a resolution of 0.1, giving a total of 1000 frequencies. Generic integers were assigned to each signal to indicate its power and priority. The optimizer used PSO to assign center frequencies for 3 receivers to a spectrum of 30 signals. Run times averaged 38 ms to run 100 generations with a swarm population size of 30. A graphical user interface (GUI) was developed that allowed basic initialization and running of the PSO. Results were shown in textual format and graphical format in the form of a spectrum plot. The GUI is shown in Figure 1.1. Both of these software contributions were used and expanded upon in this research.

1.3 New Approach

To achieve the optimization goals outlined in the introduction, the fitness function from the previous work was expanded to take into account the spatial location of the assets. A new GUI was developed to support the visualization of spatial location of

assets. All of the new implementation in this research was performed in C++. This language was chosen because of its generally high performance for computationally intensive applications. The C++ PSO implementation provided by Phoenix provided a solid foundation upon which to build. The fitness function from the previous C# project was converted to C++ and used as an initial starting point. The new GUI was also implemented in C++ using the Qt software framework and is described further in Section 3.2.

2. PSO AND FITNESS FUNCTION

2.1 Fitness Function Design

The new fitness function is based on the fitness function found in the previous work. Elements from the old fitness function pertaining to power and priority were modularized so that weights could be independently assigned to each component. Additional fitness components were added to achieve the goal of optimizing in space as well as frequency. The fitness function returns a single floating point value that represents the overall desirability of the input solution. This fitness value is composed of a weighted sum of several fitness components. Each of these fitness components addresses one aspect of the problem. These components are as follows: (1) priority, (2) power, (3) spread, and (4) distance. Each component is described in the following subsections. The overall fitness is calculated according to Equation 2.1.

$$Overall\ Fitness = \sum_{i \in Fitness\ \ Components} Fitness\ Component_{(i)} * Component\ Weight_{(i)} \tag{2.1}$$

2.1.1 Fitness Priority Component

Each signal in the spectrum has been assigned a priority that remains constant through the duration of the problem. As was the case with the previous work, a uniformly distributed random variable is used to assign priorities to signals from the set {1, 3, 5}, where a higher number represents a higher priority. Priority assignment is done upon initialization of the problem. The fitness function calculates the priority fitness component as the sum of the all of the priorities of the received signals according to Equation 2.2. As in the previous work, it is possible for two or more

receivers to overlap in frequency such that they are both receiving the same signal. In this case, the fitness function only counts the priority once for each unique received signal.

$$Fitness\ Priority\ Component = \sum_{i \in Received\ Signals} Priority\ of\ Signal_{(i)} \tag{2.2}$$

2.1.2 Fitness Power Component

In a manner similar to the way in which the priority component is calculated, the power component is found by summing the powers of the received signals. As with the priorities, each transmitter is given a random transmit power upon initialization. This random initialization simulates a varied RF environment that may be encountered in the field. When summing the signal powers, the redesigned fitness function accounts for the distance between the receiver and signal source. After calculating the distance, the free space path loss of the signal is calculated according to $dBLoss = 20{*}log_{10}(d) + 20 * log_{10}(f) + 32.45$ where d is in kilometers and f is in MHz. Thus, the power of each signal is calculated from the perspective of each asset. The power of each received signal is then summed in magnitude form. As with the priority component, the fitness function does not count twice any signal that is received by two or more receivers. The total sum of the received power is converted to dB scale and used in the fitness component. A problem arises when negative dB values are encountered. If the conversion to dB scale results in a negative value, the returned fitness component would subtract from the overall fitness even though it may be beneficial to receive the signals. To overcome this, we add an appropriate offset to the final dB value such that the returned value is guaranteed to be a positive value. This method of calculation is summarized in Equation 2.3. In this equation, $Power\ of\ Signal_{(i)}$ represents the magnitude of the signal as seen by the asset that is receiving it.

$$Fitness\ Power\ Component = 10*\log_{10}\left[\sum_{i \in Received\ Signals} Power\ of\ Signal_{(i)}\right] + Offset \tag{2.3}$$

2.1.3 Fitness Spread Component

One of the main requirements of this research is to ensure that the optimizer produces a solution where the assets are spatially dispersed as much as possible. This spatial dispersion is useful for optimization of problems like battlefield resource distribution of mobile assets, cell phone tower locations, distribution hubs for order fulfillment, etc. Achieving spatial dispersion was accomplished by adding a spread component to the fitness function. Two methods of calculating the spread component were investigated. First, the spread component was calculated by finding the sum of the Euclidean distances between all of the receivers. Our testing showed that this did not effectively spread out the assets. In many cases when optimizing for three assets, the optimizer would place two of the assets near each other and place the third a far away at the edge of the solution space. Through testing, we found that it was possible to effectively counter this behavior by calculating the spread component as the distance between the two closest assets. Calculating the spread component in this manner forced the optimizer to more evenly spread the assets around the solution space.

By design, the spread component and power component of the fitness will fight each other. It is not possible to maximize both at the same time, since a high-spread fitness solution will place the receivers far away from the signals and thus cause the power fitness component to have a lower score. A challenge arises from the fact that RF loss is input to the system in dB, and follows a log function as distance increases. On the other hand, the spread component is linearly proportional to distance. Two fitness functions need to balance each other for proper operation, so a log of the

distance between receivers is the better choice both theoretically and experimentally. The calculation of the fitness spread component is according to Equation 2.4, in which $Distance_{(ij)}$ represents the euclidean distance between $receiver_{(i)}$ and $receiver_{(j)}$.

$$Fitness\ Spread\ Component = \log_{10}\left[min\left(Distance_{(ij)},\ i{\neq}\mathrm{j}\right)\right] \tag{2.4}$$

2.1.4 Fitness Distance Component

While the fitness spread component successfully disperses the receivers in space, it does not provide any means to distribute the assets near the receivers. It is true that the power fitness component tends to place the assets near the receivers in order to achieve a higher overall power. However, in our testing this sometimes produced unsatisfactory results due to the way in which the spread component and power component tend to fight each other. Prior to adding this fitness component, we observed cases where one asset that had relatively few signals assigned to it would be placed an infinite distance (if the boundaries were removed) from the signals. In these cases, the optimizer "sacrificed" one of the assets by causing its power contribution to become almost non-existent in order to gain an increase in the fitness spread component. Attempts to counter this behavior by adjusting weights on the fitness components were not very successful. Increasing the weight on the power component or decreasing the weight on the spread component had the effect of causing the assets to congregate too close to the receivers. Thus it was difficult to achieve a good middle ground. The addition of the fitness distance component gave more stability to the solutions obtained. This component is calculated by taking the mean of the distances between each asset and the center of mass of the transmitters that it is receiving as shown in Equation 2.5. In this equation, $D_{(i)}$ represents the distance between $receiver_{(i)}$ and the center of mass of the transmitters that $receiver_{(i)}$ is receiving. This distance, $D_{(ij)}$ is subtracted from a constant $Max\ Distance$ to so

that a higher score is given to smaller distances and so that a positive value is always returned. Section 4.1 describes a further analysis that led to the addition of this fitness component.

$$Fitness\ Distance\ Component = \frac{1}{N}\sum_{i=1}^{N} Max\ Distance - D_{(i)} \tag{2.5}$$

2.1.5 Keep Away Penalty

A keep-away penalty has been added to enforce spatial separation between the assets and signal sources. A sharp penalty is added to the overall fitness when any receiver enters a pre-defined boundary around the signal sources. This boundary takes on one of two shapes, depending on the selected problem layout. Either a circular boundary is used as seen in Figure 4.1 or a straight line boundary as seen in Figure 3.3. For every asset that is past this boundary, overall fitness is multiplied by 0.5. Prior to adding this boundary, at least one of the receivers ended up on top of the transmitter signals in order to achieve a high power score. In a real-world scenario battlefield application, it is desirable to keep assets spatially separate from defined target areas. The keep away-penalty addresses this concern by adding a harsh penalty when any asset crosses into the boundary. After the addition of this penalty, the optimizer did not place any assets inside the keep-away boundary for all configurations that were tested.

2.2 PSO Parameters

PSO parameters are provided to the optimizer in the form of an Extensible Markup Language (XML) file that is read upon start up of the optimizer program. Selection of the PSO settings was done according to which of the following two tasks was taking place: (1) development of the fitness function or (2) performance testing. Fitness

Table 2.1.
Test PSO Parameters

Parameter	Value
Inertia Type	Noisy uniform between 0.2 and 0.9
C1	1.49
C2	1.49
Population Size	200
Neighborhood Size	10
Generations	1000 Fixed

Table 2.2.
Performance PSO Parameters

Parameter	Value
Inertia Type	Noisy uniform between 0.3 and 0.8
C1	1.49
C2	1.49
Population Size	50
Neighborhood Size	1
Generations	Variable

parameters for testing were chosen to ensure that sufficiently optimized solutions were found without regard to run-time or performance. The testing swarm parameters are shown in Table 2.1.

As the final application of this research will be for real-time asset allocation, it was necessary to test the real-time feasibility of our work. A new set of PSO parameters was chosen that would substantially decrease run-time while still providing adequate solutions. Table 2.2 shows the parameters used in the run-time test as described in Section 4.4. Section 4 describes comparison between these two sets of parameters.

In Table 2.2, the number of generations is listed as variable. This is because the number of actual generations depends on the convergence of the swarm. After each generation, the change in global best fitness value is examined. This delta-fitness value is averaged over a window size of 50 generations. When the average delta-fitness decreases below 0.01, the swarm will terminate. The swarm will also terminate after 1000 generations regardless of average delta-fitness value.

3. IMPLEMENTATION

3.1 PSO Implementation

As stated in the introduction, C++ was chosen as the language for implementation. C++ has several advantages in that it is a high-level language that still provides speed for computationally intensive problems. The C++ PSO implementation provided by Phoenix Data Corporation gave a foundation upon which to build. Most of the required modifications were to the fitness function. Using the fitness function from the previous project, a new fitness function was developed according to 2.1. The code was modularized by creating subroutines to calculate each of the fitness components. Figure 3.1, generated using Doxygen[1], shows a calling graph for the calculation of the overall fitness value. The function names follow the C++ convention of namespace::class::function-name. Figure 3.2 shows a code snippet of the top level C++ fitness function.

The code in Figure 3.2 shows the function taking a vector of double precision values. This vector represents one point the solution space. The fitness of this point is returned as a double precision value. The fitness function first checks to see that the solution point has the correct number of dimensions. This is mostly a sanity check, because the swarm should have been set up with the correct number of dimensions. The double-nested for-loop following this check is used to arrange the input values into a meaningful structure. For each receiver, N+1 dimensions are required, where N is the number of dimensions the receivers are allowed to move. The +1 dimension specifies the center frequency of the receiver.

[1]Doxygen is a tool for generating source code documentation. It can be obtained from http://www.doxygen.org Last Date Accessed: 04/20/2015

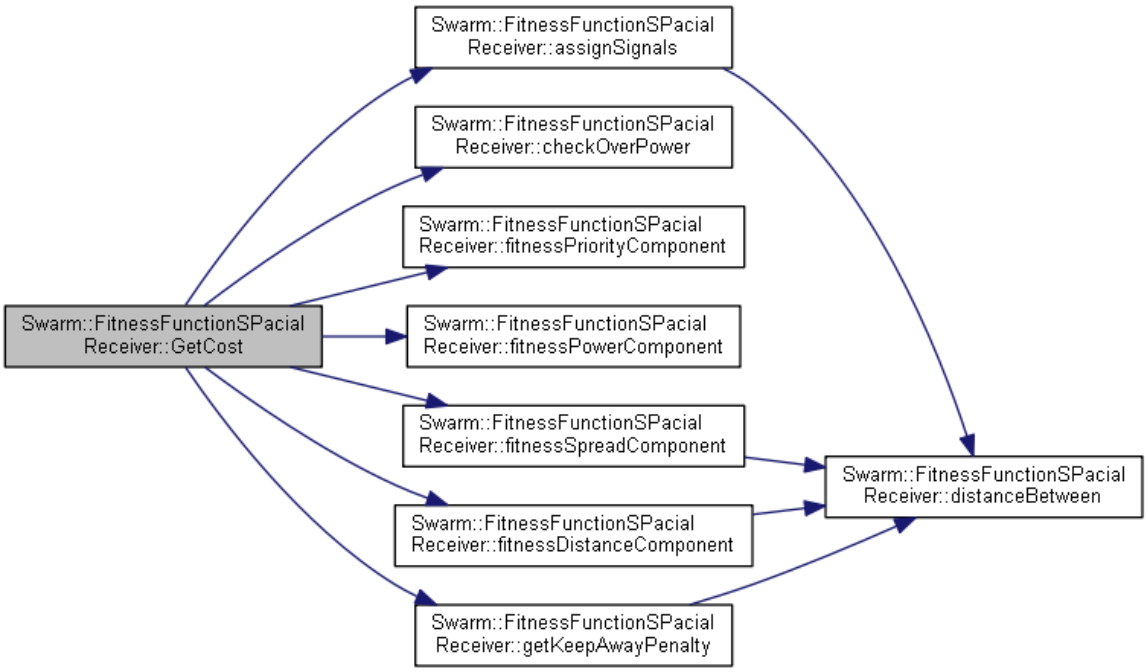

Figure 3.1. Doxygen Call Graph for Fitness Function

The function assignSignals looks at the center frequency of each receiver and checks to see which signals in the spectrum fall within the bandwidth of that receiver. Signals are assigned to receivers so that these can be used later to calculate the fitness. If two receivers have an overlapping bandwidth, any signal in the overlapping portion of the spectrum is only counted once.

The checkOverPower function checks to make sure the RF front-end of each receiver is not overloaded. This is done by summing the magnitudes of all the signals within the bandwidth of a receiver. If this sum exceeds a defined limit, this receiver is said to be saturated. In this case, the receiver is unable to provide any useful intelligence and will not contribute to the overall fitness. The next several lines of code get values for each of the four fitness components and find their weighted sum. The weights are class members which have been set by functions in the GUI. Lastly, the keep-away penalty is applied as described in Section 2.1.5 of this document.

The C++ PSO code together with the fitness function have been compiled as a dynamically linked library (DLL). The GUI application loads this DLL at run time and passes settings and commands to the PSO layer. This architecture allows the PSO code to be independent of the GUI. In subsequent sections of this paper, the PSO layer is referred to as the PSO back-end and the GUI layer is referred to as the GUI front-end.

The PSO back-end depends on the Boost C++ libraries. Boost is a well-respected set of C++ classes that are used in the PSO code to provide random number generation, high precision time keeping, XML parsing, and other small computational tasks. The Boost libraries can be built for the common desktop platforms in use today (Windows, Linux, and Mac). By default, Boost libraries are set up for static linking, meaning that they do not have to be installed on the target machine.

```
double FitnessFunctionSPacialReceiver::GetCost(const std::vector<double>& potentialSolution)
{
    // First, check if our solution point has enough dimensions.
    // For each receiver, we need its coordinates and center frequency.
    unsigned int neededVariables = parameters.numReceivers * (1 + NUM_DIMENSIONS_SPREAD);
    if(potentialSolution.size() < neededVariables)
    {
        std::cerr << "Fitness function not given enough variables.  Needs " << neededVariables << std::endl;
        return 0;
    }

    int dim=0; // Variable to loop through the input dimensions.
    // Pull the multi-dimensional solution point into a meaningful structure.
    for(unsigned int i=0; i < parameters.numReceivers; ++i)
    {
        for(unsigned int j=0; j < NUM_DIMENSIONS_SPREAD; ++j)
            receivers[i].location[j] = potentialSolution[dim++];
            //receivers[i].location[j] = 0;
        receivers[i].BW_lower = potentialSolution[dim] - parameters.halfReceiverBandwidth;
        receivers[i].BW_upper = potentialSolution[dim++] + parameters.halfReceiverBandwidth;
        receivers[i].totalReceiverPower = 0;
        receivers[i].assignedSigs.clear();
    }

    // Assign signals to receivers based on which signals fall into the bandwidth of each receiver.
    assignSignals();
    // Check each receiver to make sure the RF front end is not overloaded.
    checkOverPower();

    // Get each component of the fitness.
    double priorityComponent = fitnessPriorityComponent();
    double powerComponent = fitnessPowerComponent();
    double spreadComponent = fitnessSpreadComponent();
    double distanceComponent = fitnessDistanceComponent();

    // Apply the weights and save them for later if someone wants them.
    lastPriorityComponent = priorityWeight * priorityComponent;
    lastPowerComponent = powerWeight * powerComponent;
    lastSpreadComponent = spreadWeight * spreadComponent;
    lastDistanceComponent = distanceComponent * distanceWeight;
    // Sum the weighted components.
    lastOverallFitness = lastPriorityComponent + lastPowerComponent + lastSpreadComponent + lastDistanceComponent;

    // Apply a penalty if the transmitters are within the keep-away fence.
    lastOverallFitness *= getKeepAwayPenalty();

    return lastOverallFitness;
}
```

Figure 3.2. Top Level Fitness Function

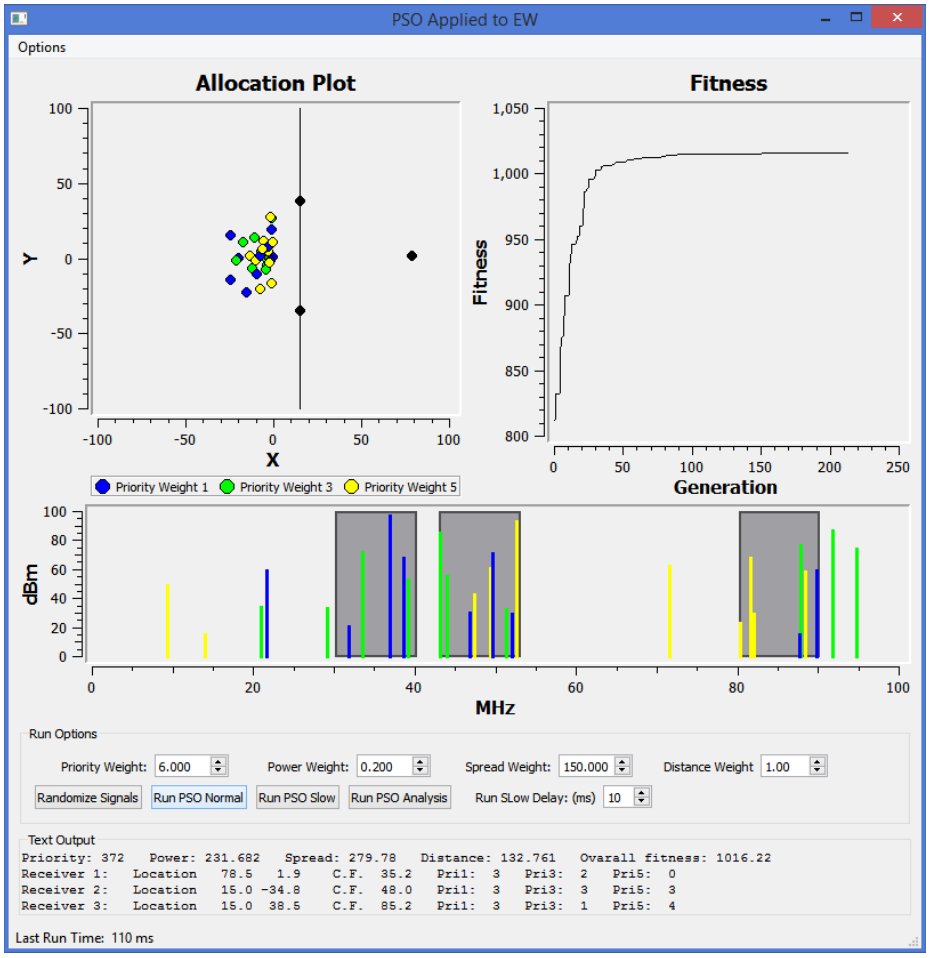

Figure 3.3. New GUI

3.2 GUI Implementation

A significant portion of this research was the design of a GUI that facilitates easy analysis and testing. The Qt software framework was chosen to develop the new GUI. Qt is an open-source, cross-platform framework for developing interactive applications. Its C++ interface makes it easy to integrate into the PSO code, which was also written in C++. In addition to the Qt libraries, the Qwt library was used for all of the plotting functionality. A screenshot of the GUI is shown in Figure 3.3.

The Allocation Plot on the top left shows the spatial location of the assets and signal sources. A black line represents a hard boundary between the assets and signal sources. The signal sources, which are color coded according to their priority, are randomly distributed in the center. Beneath the allocation plot is the spectrum view. This shows the signals sources according to their power and priority, using the same color scheme as the allocation plot. The spectrum view shows the bandwidth of each asset as a grey colored box. To the right of the allocation plot is a graph that shows the global best fitness value for each generation from the last run of the PSO optimizer. Figure 3.3 shows a run with 30 signals and 3 assets.

3.3 Platform Considerations

Research and experimentation was performed in a Windows environment. Microsoft Visual Studio 2013 was used to create and compile the PSO back-end. Qt creator, an IDE developed for Qt, was used to create the GUI front-end. All of the libraries and frameworks used are cross platform. As such, all of the software work from this research can be ported to run under Linux or Mac OS.

4. ANALYSIS

4.1 Analysis of Fourth Fitness Component.

The original design included three fitness components: priority, power, and spread. These three components would were designed to provide a balanced solution satisfying the requirements for each of the fitness components. Initial tests showed that the optimizer did not always produce consistent solutions. Figures 4.1 and 4.2 show the output from consecutive runs of the optimizer. Figure 4.1 shows the assets concentrated near the receivers while Figure 4.2 shows that two of the assets have been pushed out to the edge of the boundary. With only the three fitness components, it was not possible to adjust the fitness component weights to achieve consistent and useful solutions between runs from the same initial conditions.

Therefore, another fitness component was added as described in Section 2.1.4. Qualitative analysis showed that the solutions obtained were indeed more consistent between each run of the optimizer. A quantitative analysis was done to determine the effect that the addition of this component had on the priority score. The optimizer was run 15 times each with and without the distance fitness component using the same initialization of the signals. The priority score for each run was saved. The mean priority scores without and with the distance component were 390.0 and 404.4 respectively. A Mann-Whitney U test was done to determine the significance of this statistic. The Mann-Whitney U test has been successfully used for evaluation of PSO optimizer configurations as seen in [8]. In our case, we obtained a U value of 45, giving a level of significance near 0.005, meaning that there is a 0.5% chance that increase in priority score is due to chance. Both quantitative and qualitative tests show that the addition of this fourth fitness component is beneficial.

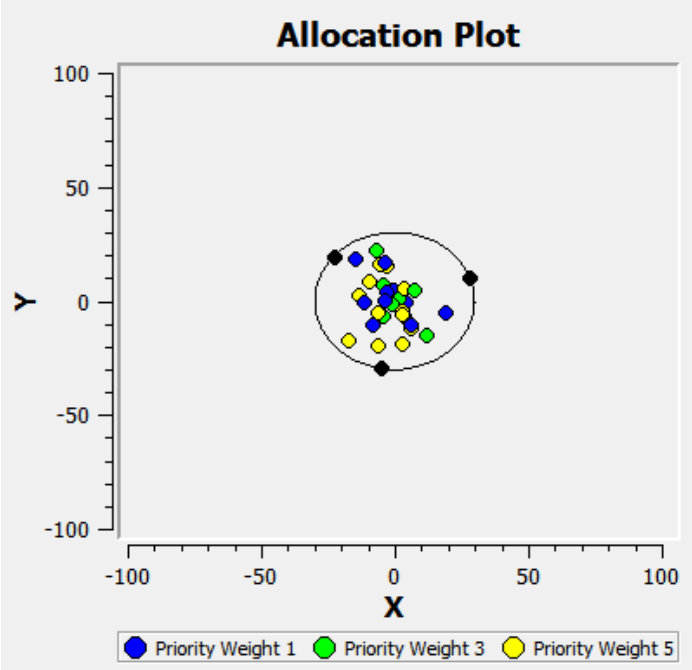

Figure 4.1. Sample PSO Run #1

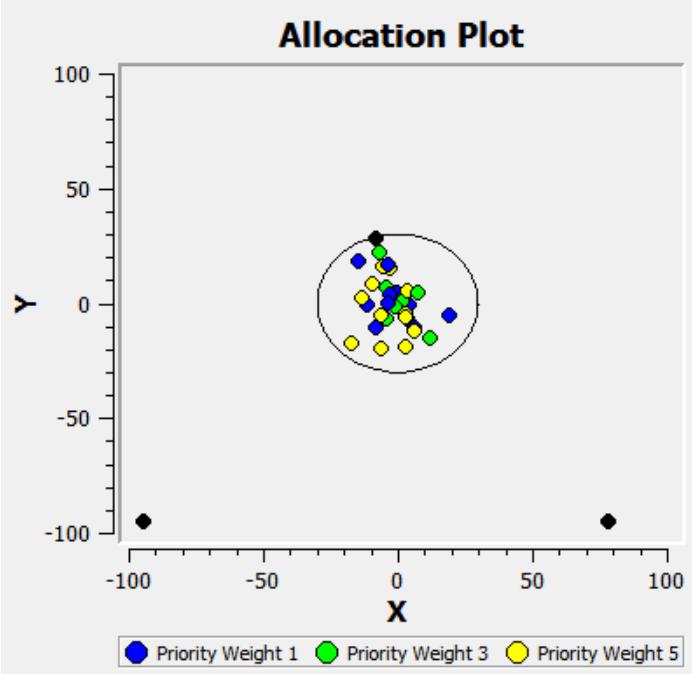

Figure 4.2. Sample PSO Run #2

4.2 Selection of Fitness Component Weights

The weights for the fitness components were determined experimentally. Initially, the weights were adjusted so that the dynamic range of each of the fitness components was roughly similar. Because the priority assignments is the most important aspect of the problem, the priority fitness component weight was increased substantially relative to the other weights.

In order to fine tune the fitness component weights, a series of tests were run with varying component weights. The priority component weight was fixed and the remaining three were allowed to vary. To determine what counted as a good combination of weights, the value of the priority score was used. For each combination of fitness component weights, the optimizer was run 30 times and the the priority score was averaged over the 30 runs. There were a number of weight combinations that allowed for a high priority score. Of these, one combination was selected that qualitatively produced a good compromise between power and spread. In any final implementation, the end user would need to make small adjustments to the power and spread component weights to give the desired trade-off between those two components. The weights that we ended up using are as follows: priority: 6.0, power: 0.1 spread: 75, distance: 1.0.

4.3 Repeatability Analysis

In order to determine the repeatability of obtained solutions, we ran a statistical analysis on the fitness values obtained from the optimizer. The PSO optimizer was run using a number of different initialization settings and PSO parameters. For each configuration, the optimizer was run 50 times. Statistical means and variances were obtained for each configuration. Tests were run using the test parameters from Table 2.1 and the performance parameters from Table 2.2. The results are detailed in Table 4.1.

Table 4.1.
Fitness Means and Variances

Configuration	Mean	Standard Deviation
3 Receivers, 30 transmitters, left-right layout, test PSO parameters	804	1.18
4 Receivers, 30 transmitters, left-right layout, test PSO parameters	891	5.57
5 Receivers, 50 transmitters, left-right layout, test PSO parameters	1234	11.6
3 Receivers, 30 transmitters, left-right layout, performance PSO parameters	794	10.7
4 Receivers, 30 transmitters, left-right layout, performance PSO parameters	868	14.4
5 Receivers, 50 transmitters, left-right layout, performance PSO parameters	1197	25.8

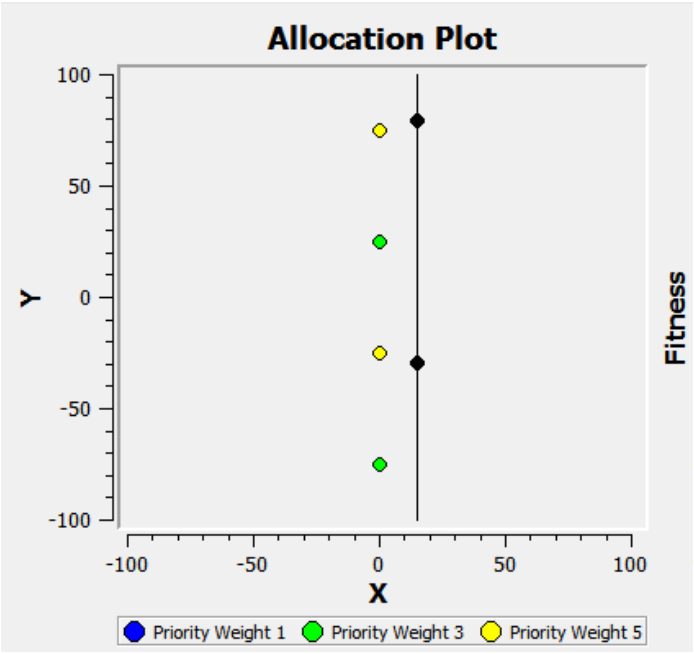

Figure 4.3. Repeatability Test Case

As a further repeatability test, a special test problem was designed with a known global maximum solution. The test problem was designed to contain local maxima in which the PSO might become stuck. A statistical analysis was run using this test setup to determine how well the PSO finds the global best solution without becoming stuck in local maxima. Figure 4.3 shows the test case where 4 transmitters of alternating priority and equal power are uniformly spaced along the battlefield line. Figure 4.3 shows the global best solution for two assets. Any other solution is a local maximum solution. Using the PSO parameters from Table 2.1, it was found that the optimizer found the global best solution with a probability greater than 0.99. Running the swarm in this configuration for 50 configurations gave an mean fitness value of 645.8 and a standard deviation of 0.001. However, when the performance PSO parameters from Table 2.2 were used, the probability of finding the global best solution dropped to 0.77. The mean fitness value in this case was 628.9 and the standard deviation of fitness was 37.8.

An unexpected finding resulted from the repeatability testing described in the previous paragraph. It was noticed that the priority and power fitness components were highly dependent on the spectral density of the signals. The test case as seen in Figure 4.3 required the priority weight and power weight to be increased in order to compensate for the sparse spectrum. Because a dense or sparse spectrum will respectively result in more or less signals being received by each asset, the power and priority fitness components will increase or decrease relative to the spectral density of the operating environment of the optimizer. The fitness component weights found to work with the test case in Figure 4.3 are priority: 20.0, power: 1.0 spread: 75, distance: 1.0. Future work will address this issue.

4.4 Run-time Analysis

Several run-time analyses were performed on the optimizer. Run-times were found for varying problem sizes and varying swarm parameters. For each run-time analysis, the PSO was run 50 times and the resulting run-times were averaged. Tests were run with both sets of PSO parameters as described in Tables 2.1 and 2.2. All run-time tests were performed on an Intel Core i7-4710HQ processor. All tests were run using a single thread of execution. Table 4.2 shows the results from the run-time analysis.

Table 4.2.
Run-time Results

Configuration	Average Run-time
3 Receivers, 30 transmitters, left-right layout, test PSO parameters	452ms
5 Receivers, 30 transmitters, left-right layout, test PSO parameters	667ms
7 Receivers, 50 transmitters, left-right layout, test PSO parameters	1098ms
3 Receivers, 30 transmitters, left-right layout, performance PSO parameters	24ms
5 Receivers, 30 transmitters, left-right layout, performance PSO parameters	45ms
7 Receivers, 50 transmitters, left-right layout, performance PSO parameters	85ms

5. CONCLUSIONS

5.1 Summary

The results of this research show that PSO is applicable to real-time optimization of a complex spatial and frequency asset allocation problems. The main contribution of this research was an extension of the optimizer to include spatial awareness of the assets by the addition of two new parameters in the fitness function. The test results show fast run times which are between 1100ms and 450ms with a 99% repeatability of finding the global best solution in our test problem. Much faster run-times of 24ms 85ms have been achieved with a repeatability of 77% in the same test case. Results are visualized on a newly-developed and user-friendly GUI that is cross-platform.

5.2 Future work

All work in this research assumed that the assets were constrained to move in two dimensions. As real world applications often work in three dimensional space, it will be necessary to research the effectiveness of this optimizer when applied to thee-dimensional space. The move to three dimensions will require support in both the PSO back-end and the GUI front-end. The code in the PSO back-end has been written in a dimensionally generic way so that it should be compatible with three dimensions after a simple re-compile, although this has not been tested yet. Changes to the GUI will need to be made in order to visualize three-dimensional assets.

As mentioned in Section 4.3, there is a slight issue with two of the swarm parameters being sensitive to the spectral density. Future work could experiment with a way to normalize the priority and power fitness component so that a given set of fitness component weights can apply to all operating environments.

Researchers in swarm technology have experimented with humans in the swarm and human-swarm hybrid optimizers as seen in [9]. Human assisted swarms use the input from a human to steer the convergence to a desired solution. A follow-on project to this research will be to determine the applicability of human in the swarm technology to problems presented in this paper.

REFERENCES

REFERENCES

[1] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *1995 IEEE International Conference on Neural Networks, Proceedings*, vol. 4, November 1995, pp. 1942–1948 vol.4.

[2] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, October 1995, pp. 39–43.

[3] R. Eberhart, P. Simpson, and R. Dobbins, *Computational Intelligence PC Tools*. San Diego, CA, USA: Academic Press Professional, Inc., 1996.

[4] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *The 1998 IEEE International Conference on Evolutionary Computation Proceedings, 1998 IEEE World Congress on Computational Intelligence*, May 1998, pp. 69–73.

[5] R. C. Eberhart and Y. Shi, *Computational Intelligence: Concepts to Implementations*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[6] F. Johansson and G. Falkman, "Real-time allocation of defensive resources to rockets, artillery, and mortars," in *2010 13th Conference on Information Fusion (FUSION)*, July 2010, pp. 1–8.

[7] R. Eberhart, J. Keller, J. Kraft, and J. Verdon, "Plenary speakers and invited tutorial speakers," in *2012 IEEE Symposium on Computational Intelligence for Security and Defence Applications (CISDA)*, July 2012, pp. 1–6.

[8] I. L. Schoeman, "Niching in particle swarm optimization," Ph.D. dissertation, University of Pretoria, 2010.

[9] D. Palmer, M. Kirschenbaum, E. Mustee, and J. Dengler, "Human-swarm hybrids outperform both humans and swarms solving digital jigsaw puzzles," in *2014 IEEE Symposium on Swarm Intelligence (SIS)*, December 2014, pp. 1–8.

APPENDIX

APPENDIX

A.1 GUI Manual

The GUI has been designed to allow easy analysis and testing of the optimizer. Most of the GUI is contained within one window as shown in Figure A.1. The Allocation Plot on the top left shows the spatial location of the assets and signal

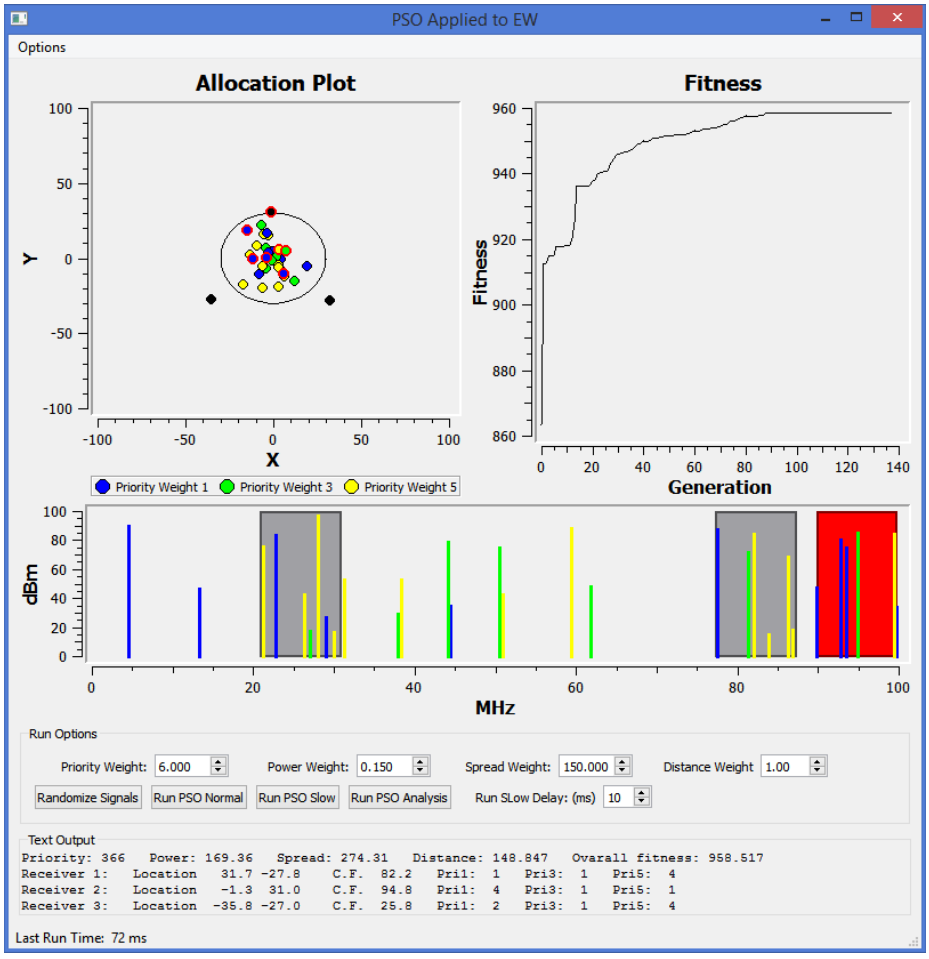

Figure A.1. GUI Showing Mouse Interaction

sources. A black line represents a hard boundary between the assets and signal sources. The signal sources, which are color coded according to their priority, are randomly distributed in the center. Beneath the allocation plot is the spectrum view. This shows the signal sources according to their power and priority, using the same color scheme as the allocation plot. The spectrum view shows the bandwidth of each asset as a grey colored box. To the right of the allocation plot is a graph that shows the global best fitness value for each generation from the last run of the PSO optimizer. Figure A.1 shows a run with 30 signals and 3 assets.

Figure A.1 shows the mouse interaction of the GUI. Hovering the mouse over any receiver or its associated bandwidth will cause both the receiver bandwidth to be highlighted as well as the receiver dot and any signals that are within the bandwidth. This allows for easier analysis. After each run, additional textual information is given that shows exact locations and frequencies of the receivers. Weights for each of the fitness function components are easily adjusted from GUI. Figure A.2 shows the additional settings which can be changed from within the GUI. Table A.1 describes the functions of each of the initialization options.

There exist three ways for the PSO to terminate. The first and most basic method is termination by number of generations. Running for a fixed number of generations is an easy and effective method to use when testing. Most tests were run for a large number of fixed generations to ensure that the swarm had sufficiently converged. A more sophisticated method is to examine the global best fitness value after each generation. When the difference in fitness values between consecutive generations decreases below a set threshold, the swarm will terminate. This method effectively examines the slope of the fitness vs generation plot to determine an appropriate time to terminate. Often, the slope is averaged over a fixed window size in order to account for variations in the swarm. A final method of termination is to run for a fixed time. We incorporated all three of these methods and allowed them to be used simultaneously. The swarm will terminate when any of the three termination conditions is met.

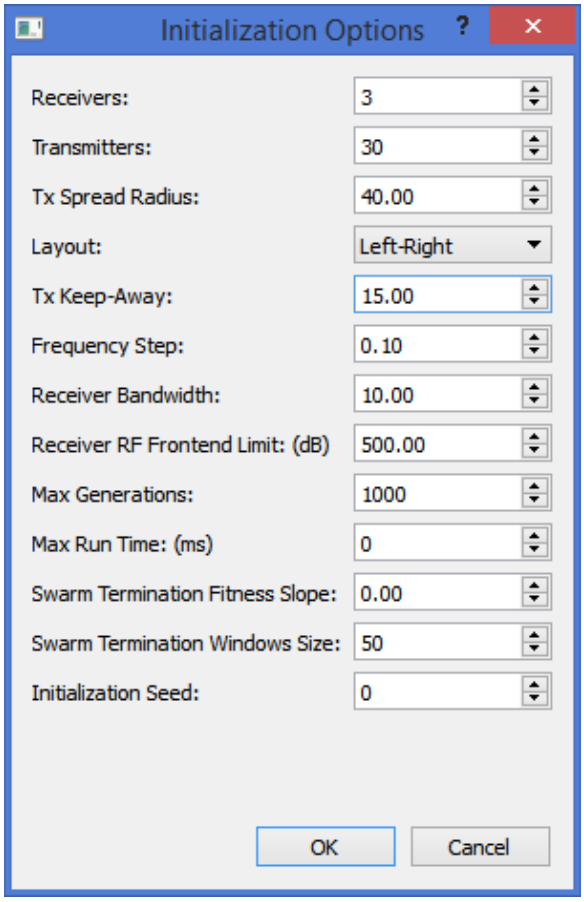

Figure A.2. Additional GUI Settings

Table A.1.
Initialization Options

Option	Description
Receivers:	The number of receivers (assets) that will be optimized.
Transmitters:	The number of transmitters (signal sources) that will exist in the spectrum.
Tx Spread Radius:	Transmitters are uniformly distributed in an area with this radius. If the left-right layout is used, transmitters are uniformly spaced in a semicircle on the left side.
Layout:	Select the battlefield layout: either circular or left-right.
Tx Keep-Away	Specifies the radius from the center that is off limits to assets (for a circular layout) or the location of the divider line (for a left-right layout).
Frequency Step:	The minimum step size for the tuners of the assets.
Receiver Bandwidth:	The bandwidth of each receiver.
Receiver RF Front-end Limit: (dB)	Any receiver whose RF front end exceeds this limit will be considered over powered and unusable.
Max Generations:	The PSO will stop after this number of generations, regardless of other settings.
Max Run Time (ms)	The PSO will stop after this time has elapsed, regardless of other settings. Set to 0 to disable this termination method.
Swarm Termination Fitness Slope:	The PSO will terminate after the slope of the fitness value vs generation is less than this value, regardless of other settings.
Swarm Termination Window Size:	The window size (in generations) over which to average the slope of the fitness value. Set to 0 to disable this termination method.
Initialization Seed:	Seed to use for the random initialization of the signals. Set to 0 to use the current time as the seed.

An option exists to run the PSO in slow time. By clicking the "Run PSO Slow" button on the main GUI, the software will add a delay between each generation. Additionally, all the plots are updated between each generation. This allows the user to visually see the swarm converging to a solution. The amount of delay between each generation is user-adjustable from the main GUI.

A.2 Build Instructions

All of the development and research as outlined in this paper was performed under a Windows environment. The machine used was a standard laptop PC running Windows 8.1. The processor equipped is an Intel Core i7-710HQ with 8 GB of installed memory. As there were no Microsoft Windows specific libraries used, the code created by this research should be portable to other environments such as Linux or Mac OS, however, the instructions that follow will be specific to a Windows environment. The code has been kept in two separate projects: the PSO back-end and the GUI front-end. The PSO back-end compiles to a dynamically linked library. The GUI front-end compiled to a binary executable and is dependent of the PSO back-end library. Build instructions for each project are described below. The instructions are written for someone who is already familiar with C++ compiling and linking.

A.2.1 PSO Back-end

The PSO Back-end was built using Microsoft Visual Studio 2013 Express. Some of the C++ source code requires a compiler with C++ 11 support. As such, Microsoft Visual Studio 2010 and earlier will not work to compile this project. Before proceeding with building the PSO Back-end, the BOOST C++ libraries need to be present on the development machine. Boost[1] is licensed for royalty-free use in closed-source and open-source project. Boost 1.56.0 was used in this research. Follow the instructions

[1]Boost can be obtained from http://www.boost.org Last Date Accessed: 04/20/2015

included in the BOOST download in order to build the BOOST libraries using the Microsoft Visual Studio command prompt. After setting up the PSO back-end project to link into the BOOST libraries, it can be built as a DLL library.

A.2.2 GUI Front-end

The PSO back-end project must be build before compiling or running the GUI front-end. Two additional dependencies must be met before compiling the GUI. First, Qt[2] must be downloaded and installed. Qt is available as a pre-built package, however for the work done in this research, Qt was built manually from source because no pre-built version was available for the Microsoft Visual Studio 2013 compiler. The version of Qt used is 5.3. By default, the Qt libraries build for dynamic linking. This requires the Qt libraries to be installed on any machine that the final software is to be run on. As was done in this project, it is possible to build Qt to support static linking. The second dependency is Qwt[3]. Instructions to build Qwt are included in the download.

When developing applications using Qt, it is very advantageous to use in included IDE, Qt Creator. Qt Creator provides drag-and-drop tools for creating a GUI. Additionally, it takes care of most of the work to link the Qt libraries. Qt Creator will automatically detect the compilers that are installed on the machine. It is important to use the same compiler that was used to create the PSO back-end DLL.

The GUI front-end project should compile to and executable (EXE) file. In order for it to run, the PSO back-end DLL must exist in the same directory from which the EXE is being run. Additionally, the directory must contain an XML configuration file which specifies the PSO parameters.

[2]Qt can be obtained from http://www.qt.io Last Date Accessed: 04/20/2015

[3]Qwt can be obtained from http://qwt.sourceforge.net Last Date Accessed: 04/20/2015

VITA

Joshua Reynolds is a graduate student at the Purdue School of Engineering and Technology at Indianapolis. He is completing the requirements for a Master of Science degree in Electrical and Computer Engineering with a special emphasis on communications and signal processing. He has previously worked at 3dB-Labs Inc. where he implemented algorithms for real-time Pulse-Doppler radar processing as well as demodulation and mapping of Automatic Dependent Surveillance Broadcast (ADSB) data. He has also implemented algorithms for real-time symbol synchronization of M-Ary FSK signals. Joshua Reynolds is also involved in his local community through volunteer projects with the Alpha Lambda Delta and Phi Eta Sigma honor societies as well as ministry in his local church. After graduation, he will be working as a DSP engineer with 3dB-Labs Inc. in West Chester, Ohio.