Optimal Design of Systems that Evolve Over Time Using Neural Networks

by

Michael K. Nolan

M.S. Electrical Engineering (1991)

New Mexico State University

B.S. Management Industrial Engineering (1982)

Rensselaer Polytechnic Institute

Submitted to the System Design and Management Program
in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management
at the
Massachusetts Institute of Technology

June 2005

Signature of Author_____
Michael K. Nolan
System Design and Management Program
June 2005

Certified by _____
David R. Wallace
Esther and Harold E. Edgerton Associate
Professor of Mechanical Engineering
Thesis Supervisor

Certified by _____
Olivier de Weck
Assistant Professor of Aeronautics
and Astronautics and Engineering Systems
Thesis Supervisor

Optimal Design of Systems that Evolve Over Time Using Neural Networks

by

Michael K. Nolan

Submitted to the System Design and Management Program on
Aug 8, 2005 in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Engineering and Management

Abstract

Computational design optimization is challenging when the number of variables becomes large. One method of addressing this problem is to use pattern recognition to decrease the solution space in which the optimizer searches. Human "common sense" is used by designers to narrow the scope of search to a confined area defined by patterns conforming to likely solution candidates. However, computer-based optimization generally does not apply similar heuristics. In this thesis, a system is presented that recognizes patterns and adjusts its search for optimal solutions based on performance associations with these patterns. A design problem was selected that requires the optimization algorithm to assess designs that evolve over time. A small sensor network design is evolved into a larger sensor network design. Optimal design solutions for the small network do not necessarily lead to optimal design solutions for the larger network. Systems that are well-positioned to evolve have characteristics that distinguish themselves from systems that are not well-positioned to evolve. In this study, a neural network was able to recognize a pattern whereby flexible sensor networks evolved more successfully than less flexible networks. The optimizing algorithm used this pattern to select candidate systems

that showed promise for successful evolution. In this limited exploratory study, a genetic algorithm assisted by a neural network achieved better performance than an unassisted genetic algorithm did. In a Pareto front analysis, the assisted genetic algorithm yielded three times the number of optimal "non-dominated" solutions as the unassisted genetic algorithm did. It realized these results in one quarter the CPU time. This thesis uses a sensor network example to establish the merit of neural network use in multi-objective system design optimization and to lay a basis for future study.

Table of Contents

List of Figures

Acknowledgements

I thank Dave Wallace for his patient support and great attitude. His special approach toward leading research was of great help to all of us. He was able to provide a fertile research environment despite the turbulent funding situation and surprising issues involved with DOME technology licensing. I am grateful for the help of Oli de Weck for his work in MSDO and his support of my work. Thanks go to all the professors and classmates who provided an exciting and provoking atmosphere. I am very grateful for the support of my fellow CADLAB students and the staff as well. Finally, I thank my wife Kim and my family for their patient understanding.

Motivation

1.1 Genetic algorithms and other optimization techniques are achieving good results on an increasing number of design problems [1] [2]. One area that still can be improved is the ability to recognize patterns when searching for optimal design solutions. Human "common sense" is used by designers to narrow the scope of search to a confined area defined by patterns conforming to likely solution candidates. However, computer-based optimization generally does not apply similar heuristics.. In this thesis, a system is presented that recognizes patterns and adjusts its search for optimal solutions based on an understanding of these patterns.

1.2 To demonstrate how pattern recognition can improve optimization performance, a design problem was selected that requires the optimization system to observe designs that evolve over time. In this design problem, a small sensor network design is evolved into a larger sensor network design. Optimal design solutions for the small network do not necessarily evolve into optimal design solutions for the larger network. The sensor network chosen for this demonstration is a very basic one in which sensors are placed in layers and required to detect points in a field of interest. The initial design is a two layer network with one node on the first layer and three nodes on the second layer. This is depicted in figure 1.

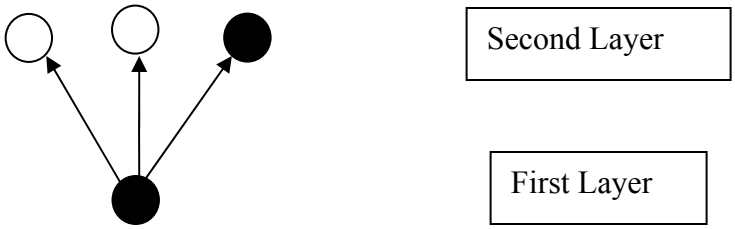


Figure 1 - Two Layer Network

The design problem is then subsequently increased to require the design to be extended to detect points on a third layer. This is shown in figure 2.

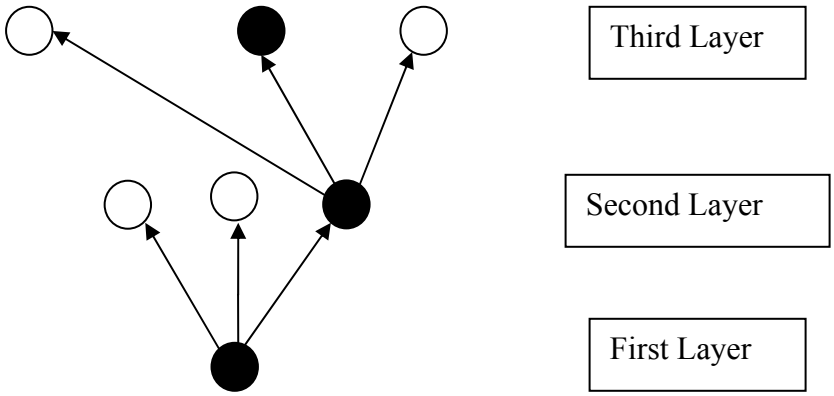


Figure 2 - Three Layer Network

Background

1.3 There has been significant progress in the field of optimization over the past 10 years. Advances have been made in algorithms, processing speed, approaches to

optimization, and integration of computational optimization capabilities within the human design process and organization [3][4] One area of rapidly advancing capability is the field of Multidisciplinary System Design Optimization (MSDO). The field considers not only the computational tools used to perform numerical optimization, but also how the tools are deployed within the design organization as it iterates through a process in search of a design that performs well relative to its objectives. In MSDO, designers consider a number of objectives as they search for acceptable solutions using computational optimization tools [5]. In this thesis, the competing objectives of cost and performance are studied. The problem formulation could easily be extended to include a larger number of objectives.

1.4 Design for Flexibility has also been a research topic of great interest [6]. In order to compete in today's competitive markets, companies must efficiently deploy not only the next product, but a system that can generate families of products. A significant fraction of the work done on one product must efficiently apply to other products. Doing this well is very challenging since it requires considering a much larger number of objectives, a longer timeline, and higher levels of uncertainty in both design factors and objectives.

System designers must consider trade-offs between current capabilities and capabilities that may be available in future systems. If system design is done well, attractive opportunities to evolve or extend the current design are likely to become available. Careful consideration must be given to choices in the current design that affect long-term design options. Current designs that have limited flexibility or limited infrastructure may cause the designers significant adverse long term consequences.

Current capabilities for optimal design flexibility are limited. Attempts to design for flexibility are generally more qualitative than computational. Designers qualitatively discuss options to extend current designs, analyze alternative designs, and use qualitative frameworks to assess the potential for future evolution or extension. Quantitative accounting for design flexibility is currently very limited. This thesis will discuss a method of approaching computational design for flexibility.

Closely related to the concept of flexibility is that of robustness. Robust design is a set of engineering methods widely successful in reducing sensitivity to such noise factors as customer use conditions, manufacturing variability, and degradation of a system over time[7]. In the context of this thesis there are multiple layers of robustness. On a basic level, a design is robust if small variations in the design variables do not have significant adverse impacts to the performance or acceptability of the design as a whole. For example, a robust design achieves good performance with part sizes that vary over a range, not just with part sizes that are very close to a design point value. On a higher level, a design is considered to be robust if it functions well despite variations in customer use[8]. Robust designs withstand variations in customer objectives. So in terms of a numerical optimization, one would look for a design that functions well despite variations in the objective function. Consider a design that does an excellent job of meeting current market demand. A robust design would be in high demand despite changes in market demand. In order to consider these future market influences or changes in the environment, probability and statistics are used in calculating forecasts and conducting simulations. Patterns may exist that can be used to predict future behavior, but frequently they are difficult to distinguish. Neural networks may be

capable of discerning a pattern drawn from a relatively small number of widely separated artifacts taken from a relatively large amount of data. Neural networks gain understanding through the interplay of multiple sources of knowledge [9]. This understanding through pattern recognition can allow a design team to bring a richer body of information to bear in the search for robust, "optimal" design solutions.

Systems that are well-positioned to evolve have characteristics that distinguish themselves from systems that are not well-positioned to evolve. Recognizing the characteristics, or the patterns that help predict ability to successfully evolve is straight forward in some cases. For example, an avionics system in an aircraft with limited cooling, power and space provisions would easily be seen as having limited ability to successfully evolve. A design team could easily recognize similar patterns in designs and use judgment in selecting designs exhibiting favorable patterns. A hypothesis of this thesis is that there are other patterns beneath the surface that are more difficult to discern, but that can be revealed using neural networks. It is expected that incorporating knowledge of these patterns in design can yield significant benefits.

1.5 Computational capabilities have led to considerable change in the world of design and promise to continue to make a large impact. The number and capability of tools that handle higher level issues is increasing. Larger and faster computer hardware has enabled the use of methods that were previously impractical due to computational limitations. Although, artificial intelligence and neural networks have a history of underperforming expectations [10], they have also achieved noteworthy success in a number of areas such as medical pattern recognition, voice recognition, and diagnostics.

In neural networks, several processors are connected in a manner similar in many respects to connections between biological neurons [11]. Information is processed in many interconnected elements simultaneously. The network is trained using trial and error; using and adapting to patterns of previous experience. Neural networks are designed to operate on hardware capable of simultaneously processing information, but can also be run on conventional computers mimicking the parallel computations. Biological neural networks are considered to have certain advantages over computers because of their ability to perform tasks requiring the simultaneous consideration of many pieces of information or constraints. Each constraint may be imperfectly specified and ambiguous.

Most neural networks have some sort of "training" rule whereby the weights of connections are adjusted on the basis of presented patterns. In other words, neural networks "learn" from examples, just like young children learn to recognize what a dog is after they are shown a number of dogs or pictures of dogs. After training, the neural network applies its knowledge to something that is similar, but is not necessarily an exact replica of the data with which it was trained.

Objective

1.6 Hypothesis: neural networks might be used in MSDO to increase system design optimization performance. Using neural networks could allow us to be more effective as we consider numerous factors in system design. Point designs can be brittle and are subject to obsolescence if future factors not included in the model turn out to be pivotal or if factors vary markedly from their predicted levels. Choosing designs that are likely to evolve well is a complex challenge requiring consideration of numerous factors.

Models are used to predict results yielded by various design inputs. Neural networks might be used to recognize patterns between inputs and results. In this thesis, a system is presented that recognizes patterns and adjusts its search for optimal solutions based on an understanding of these patterns. Neural networks should allow us to effectively include knowledge of future possible states of nature, of inexact tolerances, of changing requirements and of changing constraints. This thesis uses a sensor network example to establish the merit of neural network use in multi-objective system design optimization and to lay a basis for future study.

The neural network was used specifically to assist a genetic algorithm optimizer in its search for two layer sensor system designs that evolve into three layer designs effectively. The example problem in this study required the optimizing system to choose a number of standard sensor nodes and a number of more flexible sensor nodes and to place each in a layer of a sensor network. The objective for the neural net was to recognize patterns between the use of the flexible nodes on a two layer sensor system and the objective performance of three layer designs that evolved from the two layer systems. Objective performance was measured in terms of cost and a performance function that rewarded designs with minimum distance between sensors and target points in a field of interest.

Genetic Algorithm Structure

1.7 In this thesis, genetic algorithms were used. The genetic algorithm is a method for solving optimization problems that is based on natural selection, the process that drives

biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions [12,13 8]. For a detailed description of basic genetic algorithm operators, the reader is referred to Holland [14], Goldberg [15], Baeck [16], and Zalzala and Fleming [17]. The algorithms for this study allowed the designer to set a range and interval across which the cross over fraction was run. Table 1 shows the values of genetic algorithm parameters used.

Table 1 - Genetic Algorithm Parameters

Genetic Algorithm Parameter	Value
Cross over fraction	Adjustable from 0 to 1.0
Number of Generations	50
Migration Fraction	0.2
Generations	50
Creation Function	Random population with uniform probability distribution
Mutation Function	Gaussian
Fitness Scaling function	Rank (scales the raw scores based on the rank of each individual instead of its score)
Selection Function	Stochastic Uniform, Tournament

Matlab and the Matlab GA toolbox were used in this work. The Matlab GA had two notable limitations that impacting the study. Both limitations decreased performance, but neither jeopardized the overall viability of the project. The Matlab GA was not built to handle constraints. Design vector values could not be limited to stay within bounds using the GA algorithm. The implementation of the GA therefore had to check for out-of-bounds design variable values and to adjust them. The second limitation of the Matlab GA is that it wouldn't explicitly work with Boolean design variables. Again, the implementation of the GA was adjusted to modify output in order to yield the correct form. Both implementations successfully handled the limitations.

Significantly better results could almost certainly be obtained with other genetic or evolutionary algorithm structures. Further research is recommended using the queueing multi-objective optimizer (QMOO) algorithm [18] and the Struggle algorithm [19].

Neural Network Structure

1.8 The structure of a neural network is depicted in Figure 3. Starting in the bottom of the figure, inputs to a neuron are multiplied by weighting factors and combined using a threshold function. The output is directed to other neurons in a higher layer, combined with outputs of other neurons, weighted and processed through that neuron's threshold function.

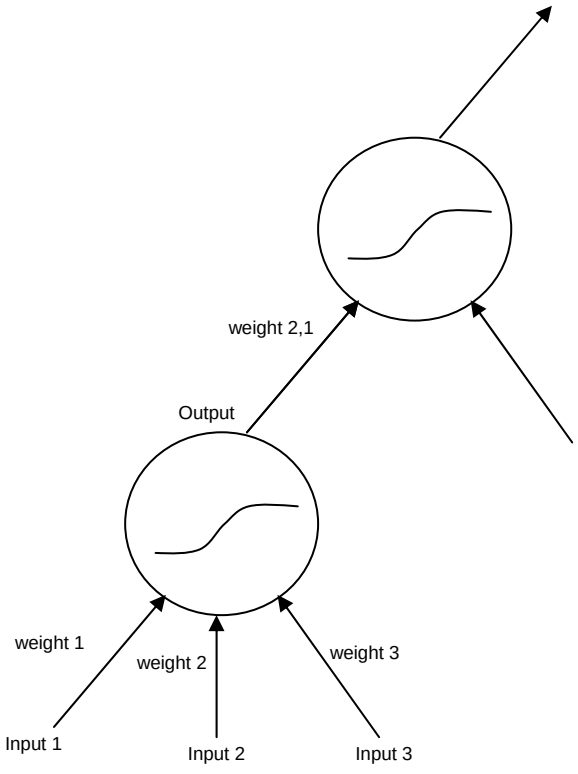


Figure 3 - Neural Network Basic Structure

Figure 4 show details of a typical neuron. The inputs are each multiplied by different weights, a bias Θ is subtracted and the threshold function (in this case a sigmoid function) is applied.

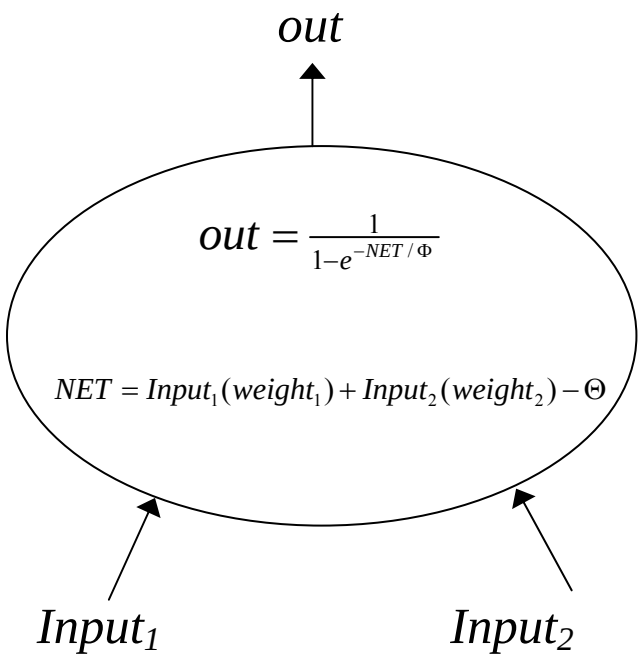


Figure 4 – Neural Network Neuron

The network is trained to recognize patterns through a routine that adjusts the weights. In general, the weights are initialized at random values and the net is trained by comparing its output with known training target values. The network makes adjustments to its weights to model the target pattern more closely. If the output of the net differs significantly from the target, large adjustments in weights are made. If the output differs only slightly, smaller adjustments are made. The network changes weights in the output layer first, then backpropagates them down through each lower layer. There are a number of variations and modifications on the training algorithm, but a basic backpropagation training formula is shown in equation 1:

$$w_{new} = w_{old} + \varsigma \Delta Input + \alpha(w - w_{t-1})$$ **Equation 1**

ζ is the training rate. A low value of ζ will make smaller adjustments in weights than a high ζ. Δ contains the difference between actual target and computed output:

$$\Delta = out(1 - out)(TARGET - out)$$ **Equation 2**

out is the output. α is the momentum coefficient. The momentum coefficient governs how training is affected by trends. A high momentum coefficient causes the weights to continue changing on their current trend while a low α causes the weights to be primarily affected by the most recent input and not by the trend.

The net trains through a number of training cycles, gaining experience and adapting to match the patterns it has been exposed to in the training data. The net stops when Δ decreases to an error goal value. This value can be adjusted depending on the needs of the user.

1.9 One of the problems faced when using neural networks is the difficulty in finding structure and parameters that work well for a particular problem. A lot of fine-tuning and trial and error of parameters is frequently required. The system used in this study to address this problem employed an algorithm that ran the network using parameters over a range of values. The system automatically centered in on parameters yielding good

results and faster learning. A detailed explanation of this architecture and the Matlab code are in Appendix A.

Neural Networks in Matlab

1.10 A three layer backpropagation network was used in this study. Refer to the Matlab help files for a thorough explanation of Matlab's implementation of neural networks [20]. The input layer was constructed of four nodes, the middle (or "hidden") layer was run using 8 nodes and the output layer consisted of a single node. The threshold functions were adjustable in the structure used. All runs reported on in the results section used either a log-sigmoid threshold function or a tan-sigmoid threshold function. The network was run for a maximum of 170 epochs. A range of values was used for the neural network's error goal. The momentum values used in this study were 0.8 and 0.9. Most of the training algorithms used in this study employed an adaptive learning rate.

The system was also designed to use a number of backpropagation learning functions. The functions are summarized in table 2.

Table 2 – Neural Network Training Functions

Matlab Training Function Name	Training Function[21]
traingdx	Gradient descent with momentum and adaptive learning rate backpropagation
Trainlm	Levenberg-Marquardt
Trainrp	Resilient Backpropagation
trainscg	Scaled Conjugate Gradient

Figures 5 - 8 depict the Matlab neural network used in this study.

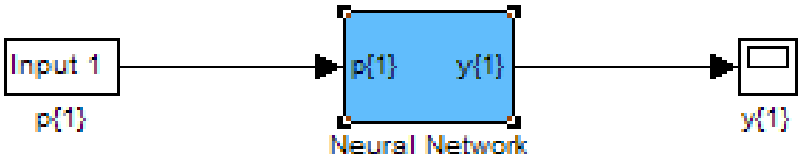


Figure 5 – Top Level Network Structure

The block diagram showing the input and output vectors.

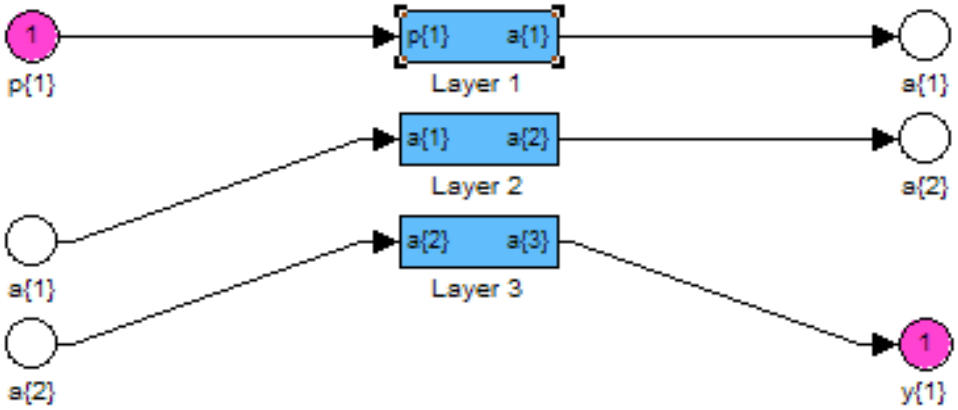


Figure 6 – Mid Level Network Structure

Looking into the block on from the previous diagram, one can see the three layers of the network.

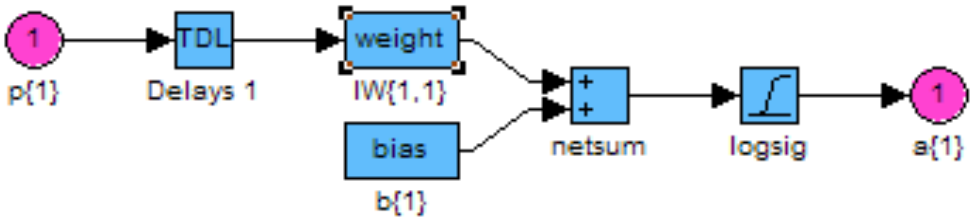


Figure 7 - Network Bias, Summation and Threshold Functions

Looking into the first layer from Figure 7, one can see the bias, summation and threshold functions used by the neuron in computing its output. Note that in this application, the Delays were zero.

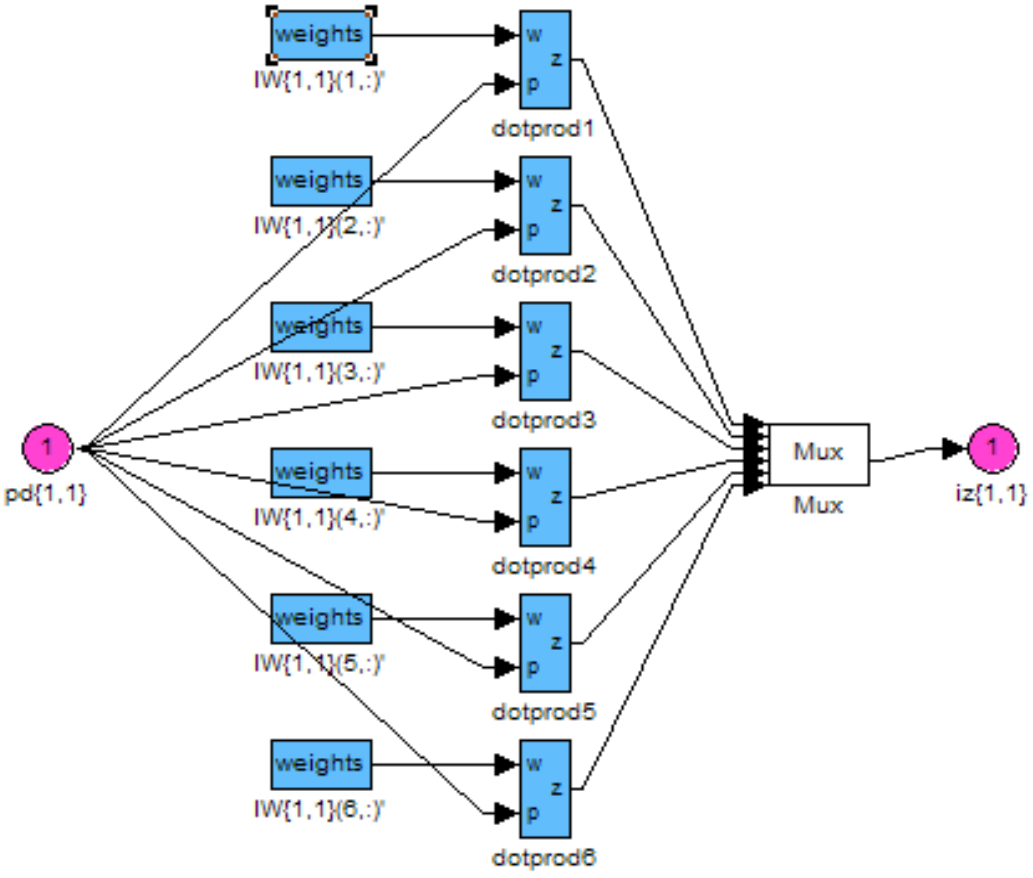


Figure 8 – Detailed Level Network Structure

Looking into the weights block from Figure 8 above, one can see the application of individual weights across the neuron's input vector.

Training, validation and test data sets

1.11 The data were subdivided into training, validation and test subsets. The training data set was used to train the network. The validation set was used as a check during training. When the validation error increases for a specified number of iterations, the training was stopped, and the weights and biases at the minimum of the validation error were returned. [22] The test set was run as a separate check to verify the network design on data other than training data. The performance for one of the training runs is plotted in Figure 9 below. The training performance is plotted in blue, validation in green and test in red. Performance is plotted as the difference between the network's output and the training set target value. In this run, the training set met the goal of 0.005 at the 61st epoch and the run terminated.

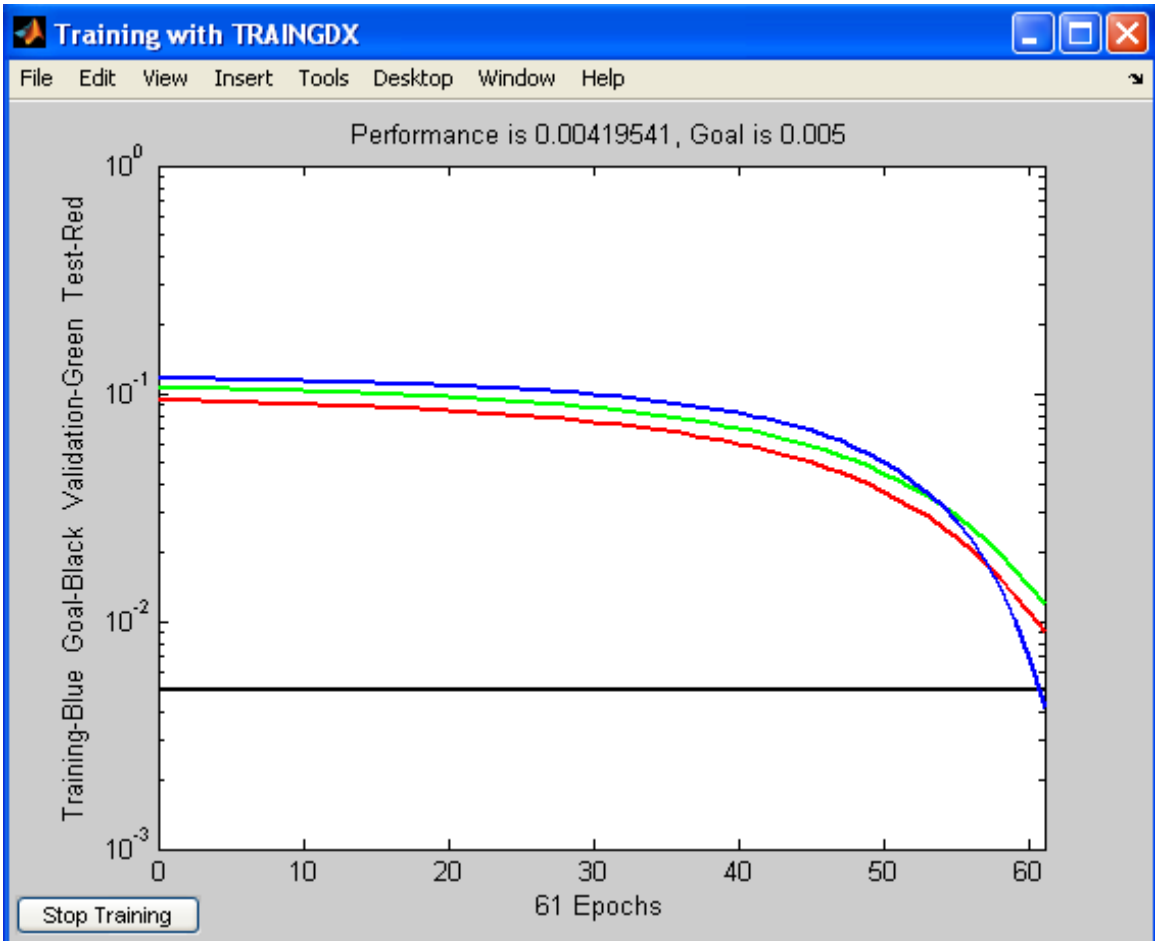


Figure 9 – Use of Training, Validation and Test Subsets

Detailed Problem Description

1.12 The sensor network briefly described in section one offers the opportunity to study the effect flexibility has on the design of systems that evolve over time. Optimization of sensor networks has been addressed in a number of recent papers [Bounova, Cohanim]. Bounova and Cohanim researched the optimization of telescope arrays. Individual radio telescopes were separated by a number of miles and connected with cable. Olivier de Weck and Damien Jourdan described a model of a network designed to warn of intrusion over an area surrounding an asset of value in their paper on multi-objective optimization [23]. In this thesis, a similar, but simpler modeling problem is posed. In this problem, a sensor network may be expanded or improved over time, and design tradeoffs must be considered that balance current performance with future performance. Designs that provide highly efficient and effective initial solutions are not necessarily the ones that provide the most efficient and effective end state as the design is extended. There are a number of cases that match this general structure. A first order model of a satellite constellation is one example. Some satellites may be designed with network capabilities that support the entire constellation while others have more limited communication and processing capability for the constellation, but still have the capability to perform their sensor missions. The sensor network is shown in Figure 10**.**

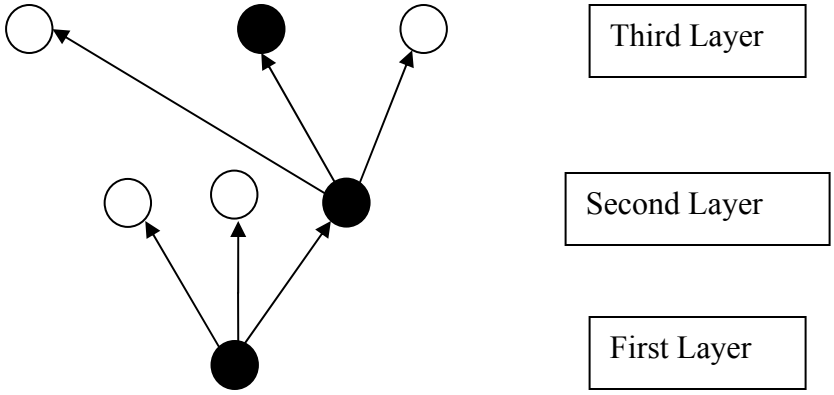


Figure 10 – Sensor Network

In this system, all the nodes are sensors. The black nodes are able to support network functions; they cost more and can connect infrastructure to other nodes. The network is designed and built one layer at a time. As each layer is designed, consideration must be made for subsequent layers that will be designed and built in the future. A flexible design will have adequate provision for favorable installation of additional layers. The best design for today's requirements uses few flexible nodes since they are more costly than inflexible nodes. But the best design for a flexible architecture allowing growth for tomorrow's extension of the network includes a significant number of strategically placed flexible nodes.

Modeling the network

The network is first designed with 2 layers. When the next sensor layer is built, each node on that layer must be connected and supported by a flexible node on the layer beneath it. In this model, there is a physical connection from the lower layer to the upper

layer as in the radio telescope network. Alterations could be made to the model to simulate wireless connections. The cost of the physical connections is modeled by the length of the connection, or the distance between the flexible node and the node on the upper layer it supports.

Figure 11 shows the node position results of one optimization run. For this 2 layer sensor network, the GA placed the sensors in the positions marked as blue squares. There are 2 sensors on the first layer (1 on the y axis) and 4 sensors on the second layer (3 on the y-axis). The network is assessed for sensor performance by inserting "points to detect" in the sensor field. These points are depicted as red triangles. A number of measures could be used to assess sensor performance. For this study, the performance was modeled after an active sensor that looks right and left. A more complicated sensor model is left for future study. For an active sensor, the performance is inversely proportional to the distance to the target raised to the fourth power. The model measured "performance degradation" instead of performance. So the two objectives, cost and performance degradation were to be minimized in this optimization problem. Except where otherwise noted, this paper will use the terms "cost and performance" instead of "cost and performance degradation."

The model used some target points that were fixed and known beforehand and some points that were randomly placed and not known beforehand. This tested the robustness of the design and penalized candidates that were tailored only to perform well relative to known existing conditions. Similar instances may be found in sensor applications, where some areas of interest are known to require careful monitoring and other areas turn out to be important, but are not known beforehand.

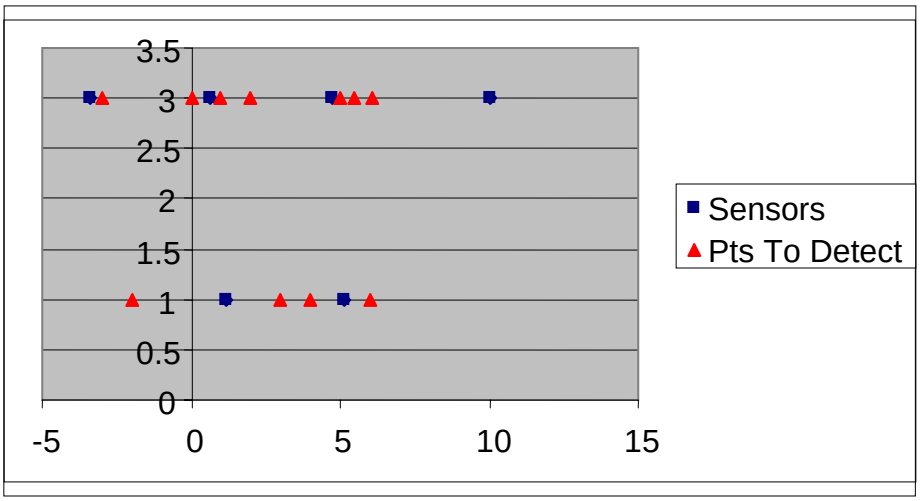


Figure 11 – Points To Detect

Description of the Neural Network / Genetic Algorithm Optimization System

1.13 A top level system description is provided here, while the code and detailed description of each function is contained in Appendix A. The system consists of 59 modules of Matlab code. As discussed in section 1, the sensor network system was modeled as a 2 layer sensor network that was subsequently extended to 3 layers. The objectives were to maximize sensor performance[1] and to minimize cost. Performance was modeled as a function of distance to selected points in the area of interest. These points simulated objects for the sensors to detect. The sensors sensed objects to their right and left[2]. The network model would accept any number of points to detect on any level. Cost was modeled by collecting fixed and variable components. The fixed cost of sensors and infrastructure was added to the variable cost of operation. Both fixed and variable costs went up as the length of connections between sensors increased.

[1] Again, this was coded as minimizing performance degradation as opposed to maximizing performance
[2] A more advanced simulation might model more capable sensors.

A genetic algorithm was used to find optimal solutions to this 2-layer sensor problem.

Figure **12** illustrates the neural network assisted GA algorithm.

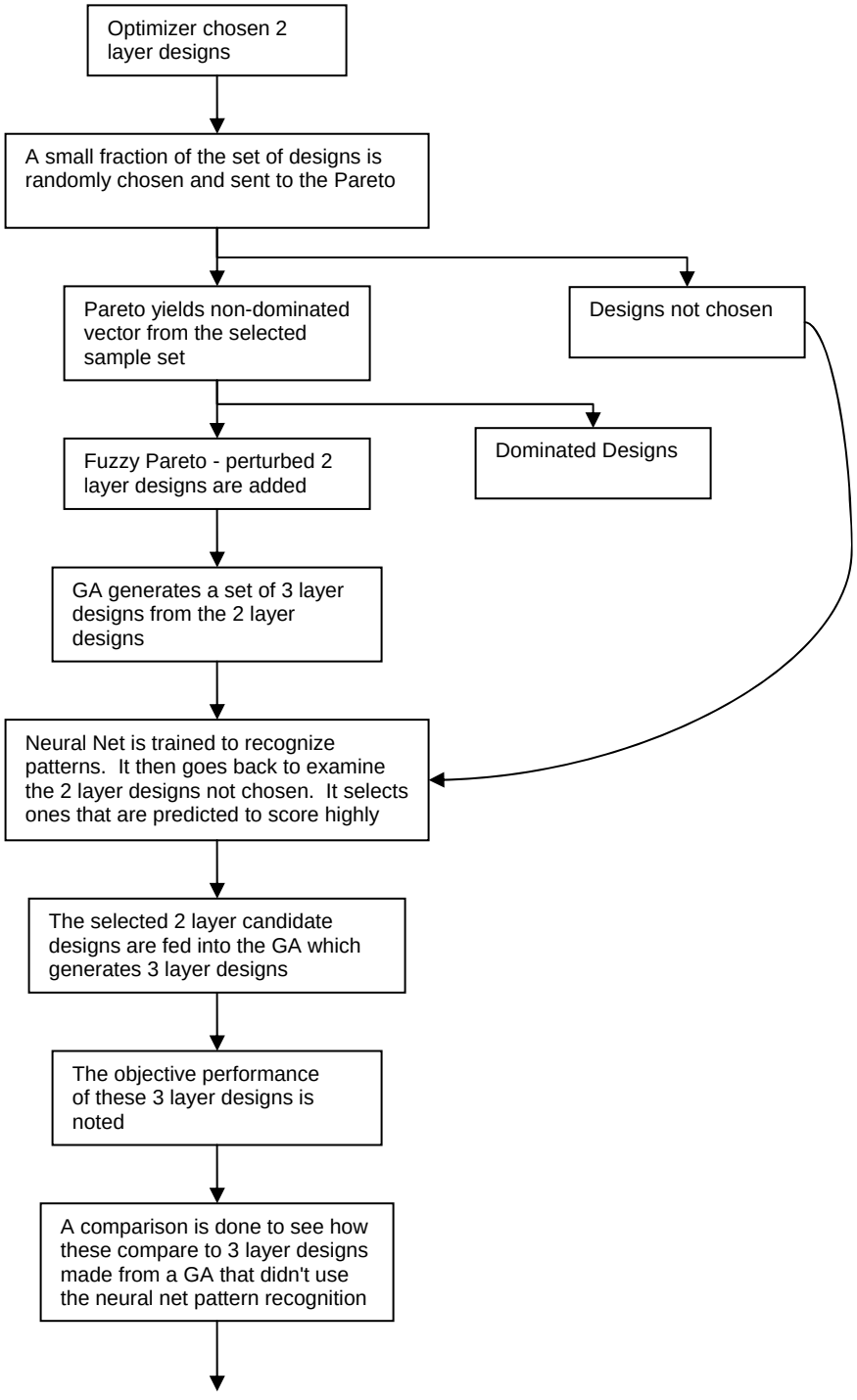


Figure 12 – Neural Network Assisted GA System Structure

Starting from the top, the GA optimizer chose a number of 2 layer designs that performed well relative to the 2 layer objectives. The GA optimizer ran a number of GA parameters as shown in Table 1 and a number of different weightings between cost and performance objectives. The model ran a number of times and the random "points to detect" changed their position on each run. The performance of each design was based on an average of the performances with each set of random points. A small fraction of the GA optimized two layer designs was randomly chosen and sent to the Pareto Front algorithm. The remainder of the designs was set aside as noted in the "Designs not chosen" branch.

The system then used a Pareto analysis to "optimize" the design relative to cost and performance. Design using Pareto analysis has been addressed by Mesac, and Mattson [24] and Smaling and de Weck [25]. In Pareto analysis, instead of finding one "best solution," a range of solutions with attractive values for all objective criteria is examined. If there are 2 objectives, the analysis can be viewed on a plot showing the Pareto frontier; the locus of points that have the best values relative to both objectives. If there are more than 2 objectives, the Pareto frontier is not as easy to visualize, but the principle is the same. Figure 13 shows a Pareto frontier and a ranking scheme.

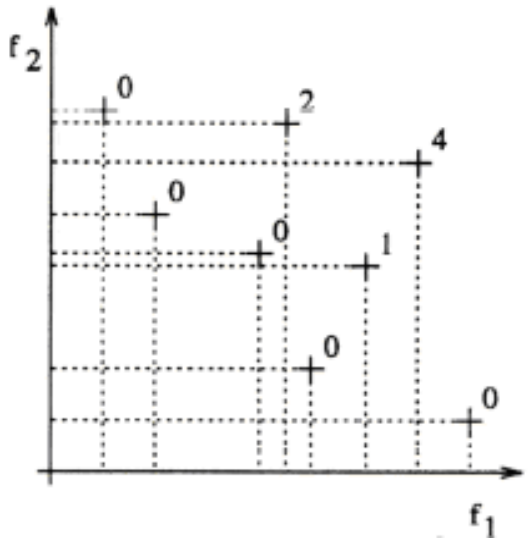


Figure 13 – Pareto Ranking for a Minimization Problem [26]

In this figure, f1 and f2 are the 2 objectives and they are to be minimized. The points labeled "0" are on the Pareto frontier, while the other points are not. Note that points not on the frontier are "dominated," that is, there exists at least one other design point that is better relative to one objective and at least equal relative to the other objective. The algorithm used to incorporate this mechanism was adapted from code shown in Appendix A[27]. The algorithm finds design solutions that are not dominated. Figure 14 shows the original Pareto data. Several of the points are dominated.

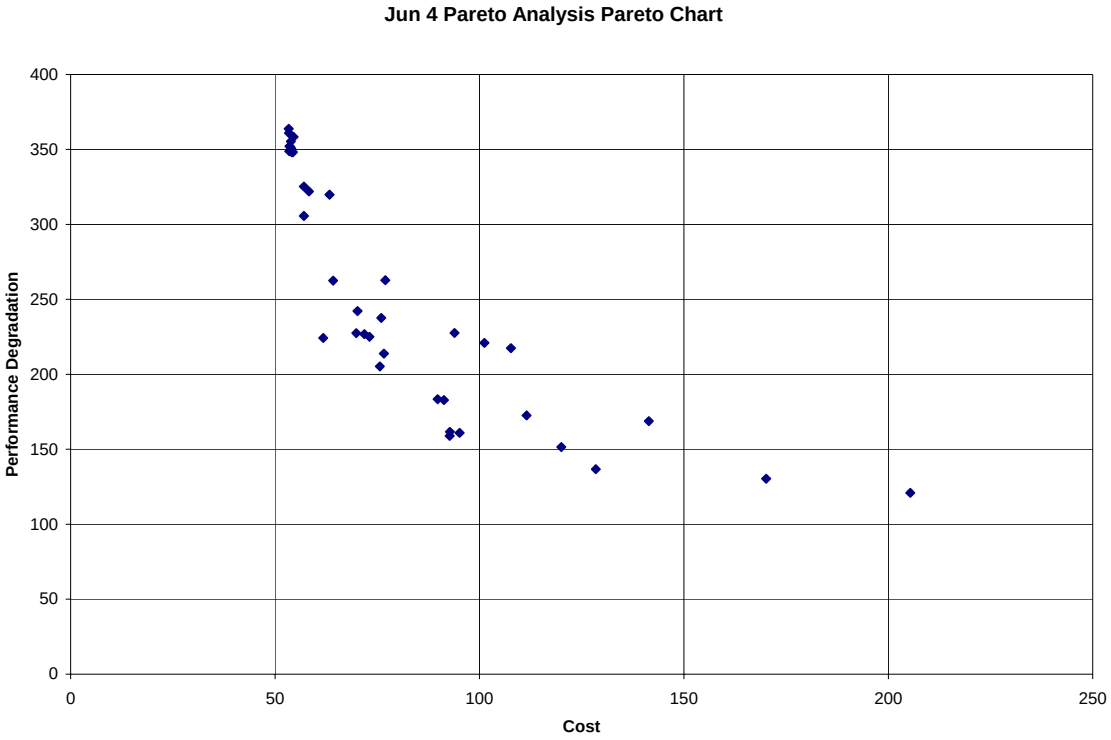


Figure 14 – Original Pareto Data

After Pareto processing, the points remaining are shown in the figure below.

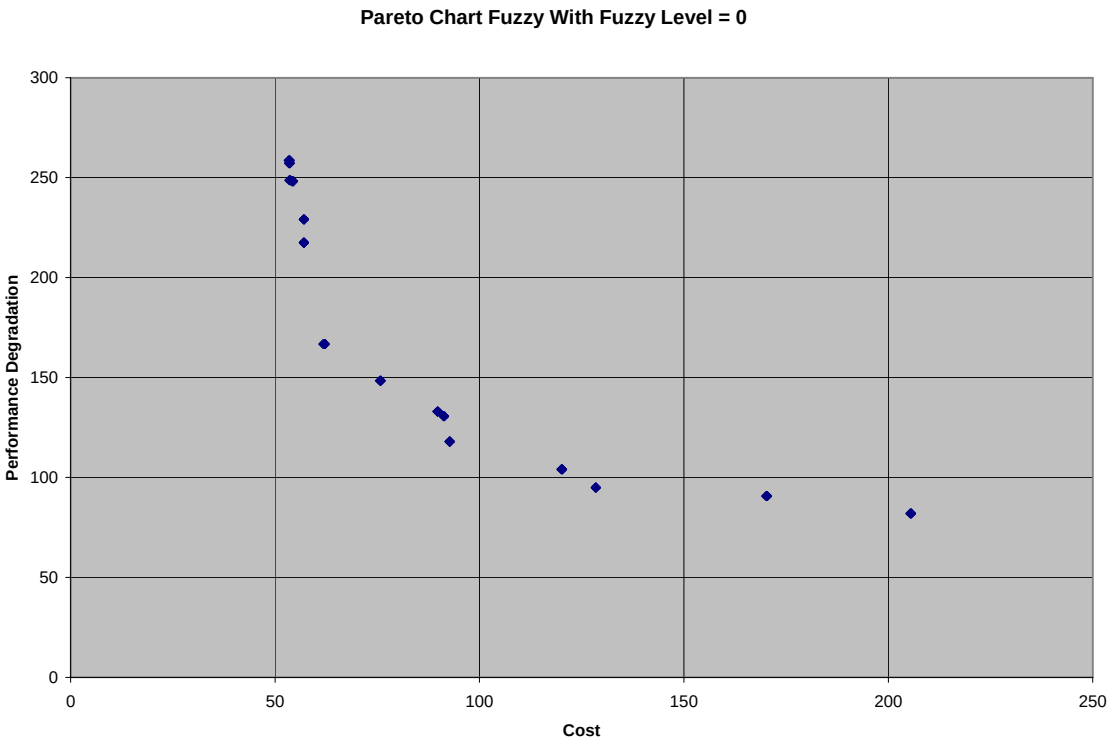


Figure 15 – Post Processing Pareto Data

In Smaling and de Weck's paper [28], a "Fuzzy Pareto Front" was established and used in optimization. The concept is that models constructed early in the design process are not perfect representations of exact information and that in the end, the best solutions generally turn out not to be ones that did not appear exactly on the Pareto front early in design. Smaling therefore considers designs close to, but not necessarily on, the Pareto front. This concept was used in this thesis as well. A modification to Smaling's algorithm was made in that the "Fuzzy Pareto Front" was made by perturbing design solutions on the Pareto front. This "quick fuzzy Pareto front" increases the diversity of

solutions considered while conserving computation time. It should also be noted however that diversity could be increased to a larger degree, albeit at a computational cost, if design solutions within the "fuzzy Pareto front" were chosen rather than perturbing points on the Pareto front. The figure below shows a plot of the (x, y) coordinate positions of an optimization run using the fuzzy Pareto front.

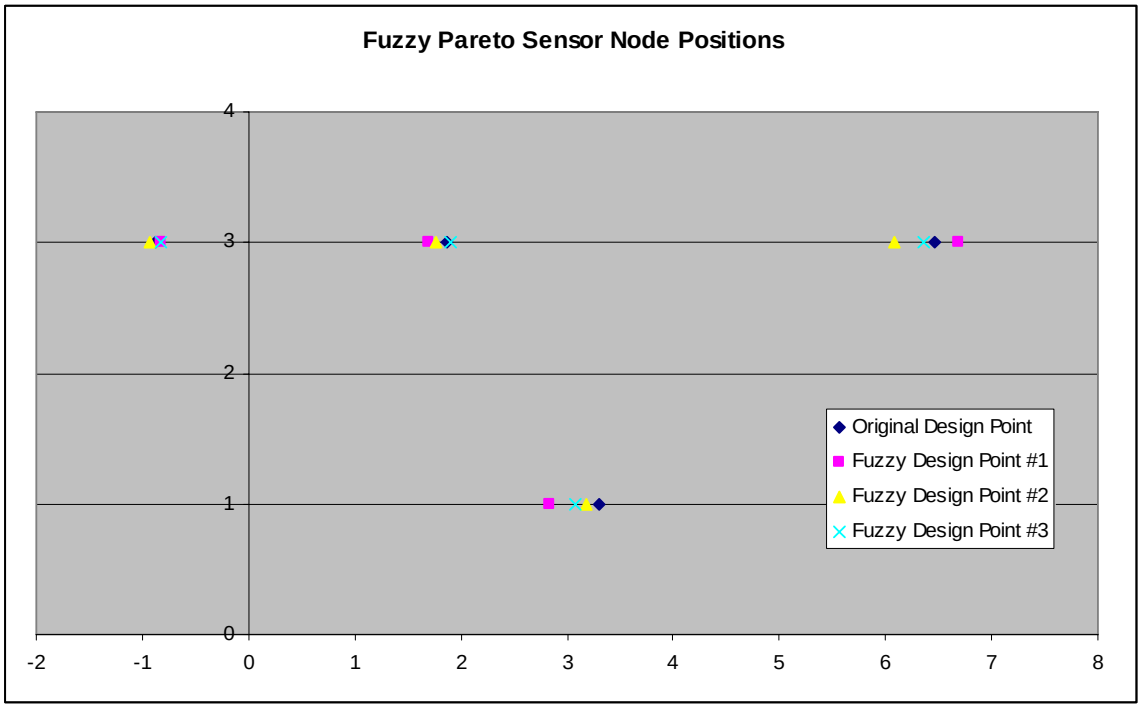


Figure 16 – Sensor Node Positions

The figure below depicts the objective performance corresponding to each of these design solutions.

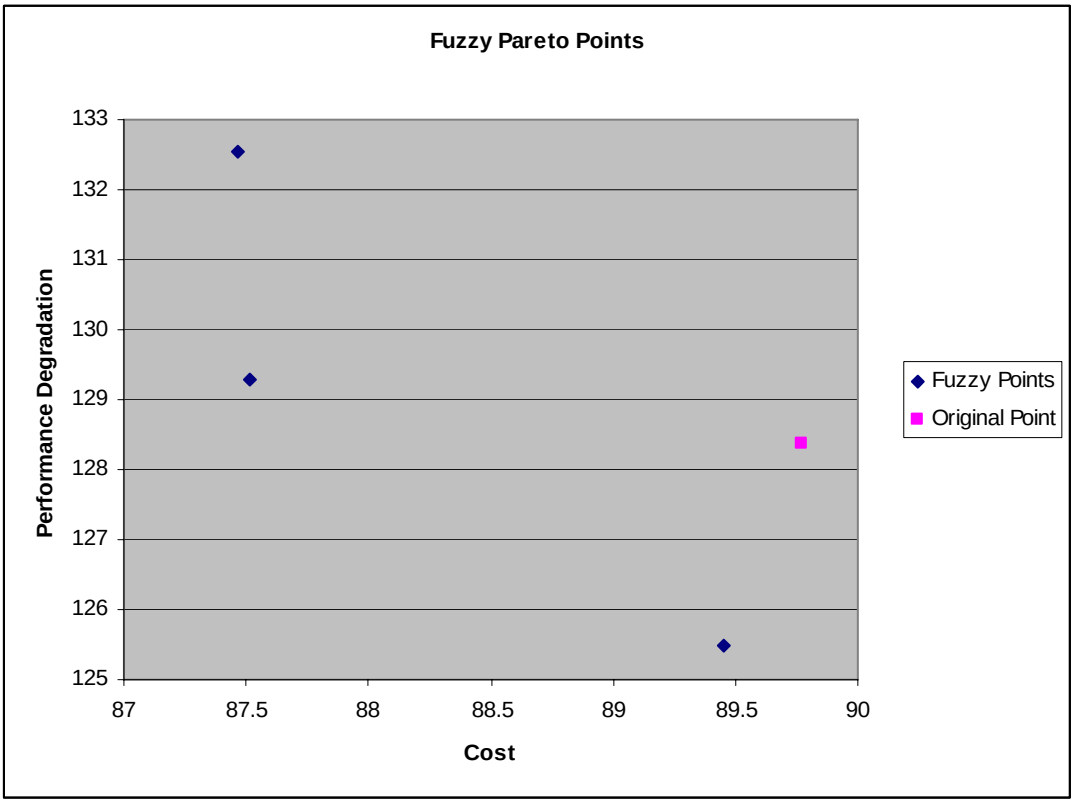


Figure 17 – Fuzzy Node Performance Example

Recall that points left and lower are superior. It is interesting to note that in this case, one of the perturbed points happened to score better relative to the performance objective than the original point and all of them scored better in terms of the cost objective.

Figure 18 shows the resulting fuzzy Pareto Front. The algorithm used a parameter allowing the user to adjust the amount of "fuzziness" applied to the perturbed design parameters.

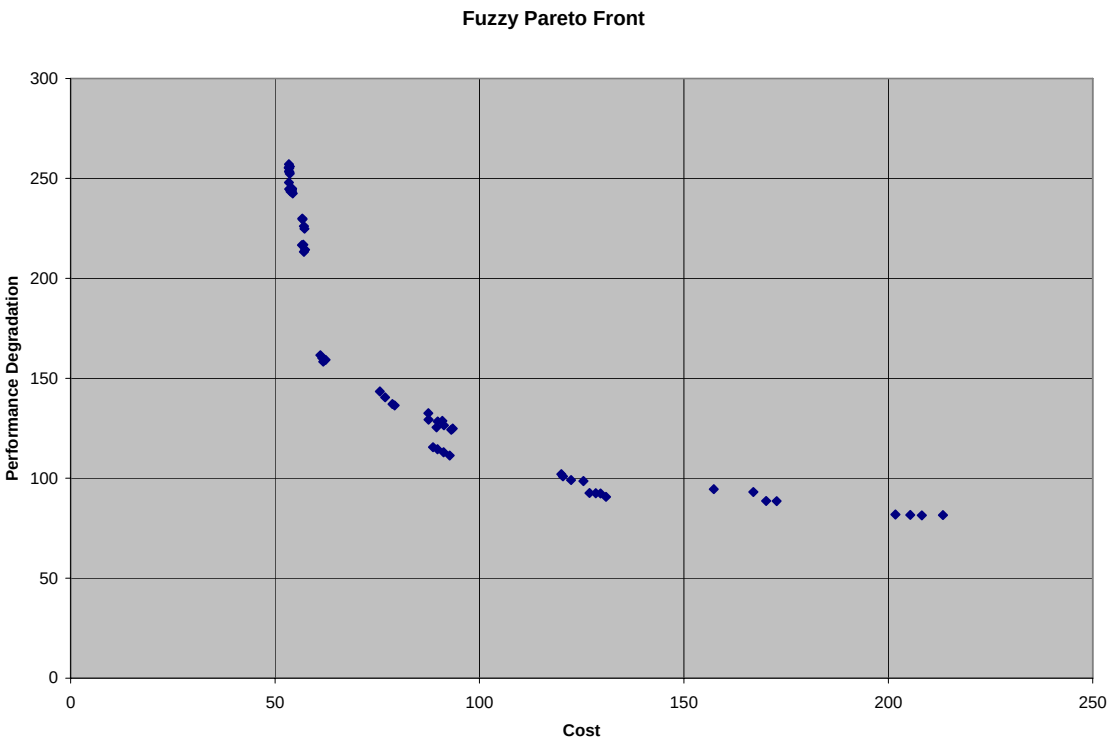


Figure 18 – Fuzzy Pareto Front

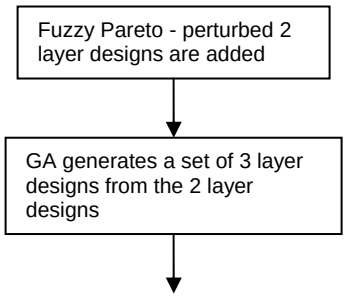


Figure 19 – Pareto Portion of System Diagram

Referring back to the Figure 12 flow chart, the task for the optimizing system at this point was to develop three layer designs by extending the 2 layer designs that emerged from the Pareto computation. The performance of a design system using GA-only

optimization was tested against a system that used a neural network to predict which two layer designs would be likely to perform well once extended to three layers. The neural network analyzed the results of a number of optimizations to discern patterns that could predict which 2 layer designs held promise for extension and which 2 layer designs were more likely to be stuck at a dead end.

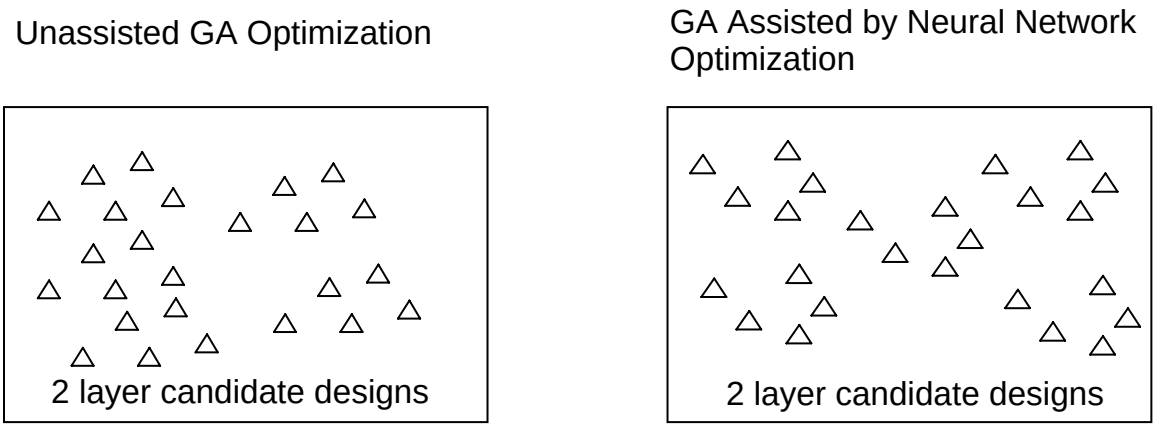


Figure 20 – Optimized 2 Layer Design Space

Conceptually, the process depicted begins with the optimized 2 layer designs, shown here as triangles in the design space. In Figure 21 below, each of the candidates of the unassisted GA must be extended to a 3 layer design in order to select the best. In the GA Assisted by Neural Network, the neural network has identified patterns indicating high potential for successful extension and the optimization focuses on only those candidates with characteristics matching that pattern.

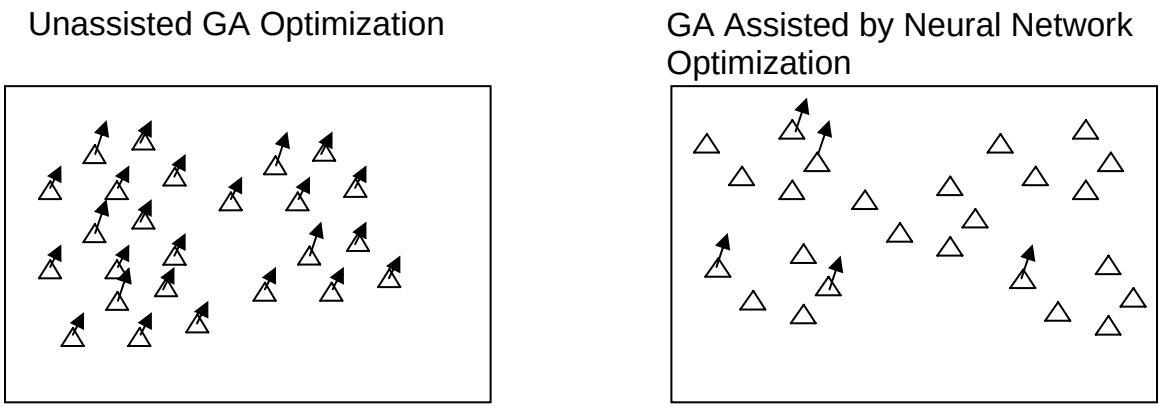


Figure 21 – Notional effect of Neural Network Pattern-Recognition Assistance

In order to identify the pattern, the neural net must select a number of candidates and train itself to recognize characteristics that indicate which candidates are likely to extend to become successful 3 layer designs. So the scale of the problem determines whether or not assisting the GA with a neural network will be worth the time spent learning the patterns. In large design spaces with large requirements for future design extensions, the advantages of pattern recognition are likely to be significant. In small design spaces and those where future design extensions are limited, it is expected that the advantages of applying pattern recognition are unlikely to be worth the cost of finding the patterns. Figure 21 depicts both the assisted and the unassisted optimizers searching the same design space and the GA Assisted Neural Network having only to examine a fraction of the candidate designs and therefore potentially doing a more efficient job. One could also consider both optimizers searching for the same amount of time, but the assisted optimizer finding a superior solution.

Figure 22 shows how the neural network processed each 2 layer design candidate as it was trained to recognize patterns. The neural network trained itself by comparing the 2 layer design vectors "input" to the 3 layer objective vectors "output". It attempted to

discern patterns which would indicate when a 2 layer design was a promising candidate and which patterns would indicate a 2 layer design was not likely to extend into a promising 3 layer network.

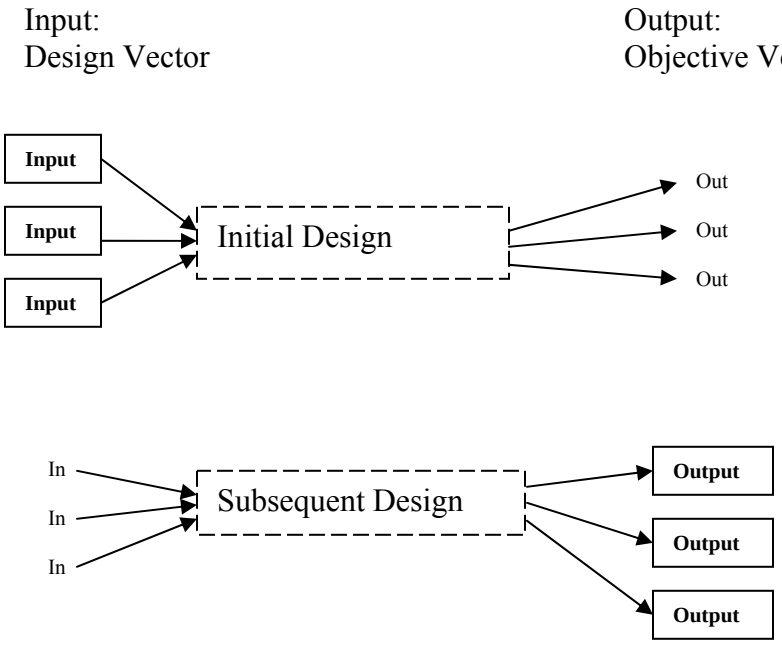


Figure 22 Pattern Recognition Problem

Once the neural network had trained on this smaller set of 2 layer candidates, it went back to the large population of "designs not chosen" and selected promising 2 layer candidates to develop into 3 layer designs. This is shown by the large arrow on the right side of the Figure 12 flow diagram.

Figure 23 shows the cost and performance objective results for a run with a small number of design points.

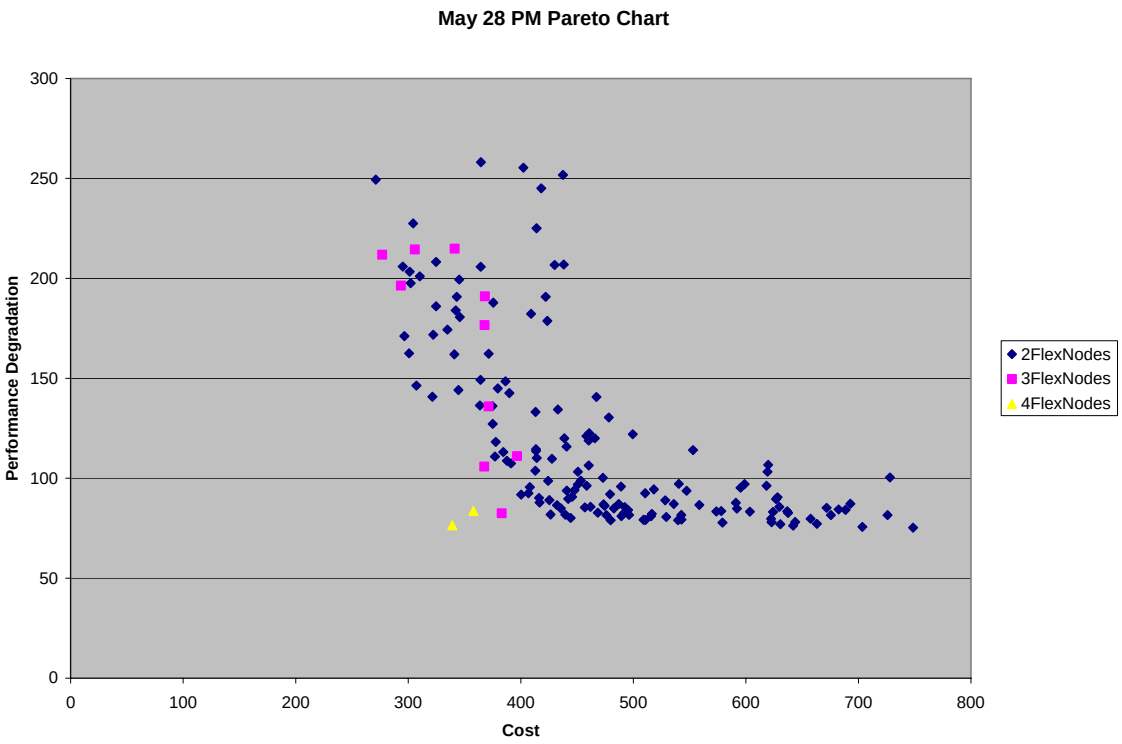


Figure 23 – Cost and Performance Results

The design points were inspected after the run and separated according to the number of flexible sensor nodes each contained.[1] From the figure, one can see that the best performing points both had 4 flexible nodes and that in general the worst performing points had only 2. So the model showed that for this set of conditions, the disadvantage of paying more for flexible nodes was overcome by the advantages offered by higher flexibility. The system selected a fraction of 2 layer design points whose 3 layer results looked promising. Among that group were the 2 layer designs that yielded the objective performance denoted by the yellow triangles in Figure 23. These points had 4 flexible nodes. The neural network recognized the pattern indicating a larger number of flexible nodes yielded better results. The "TestedResults" points on Figure 24 show the 3 layer

results of those selected 2 layer designs. These results clearly indicate the additional benefit of pattern recognition in an optimization process.

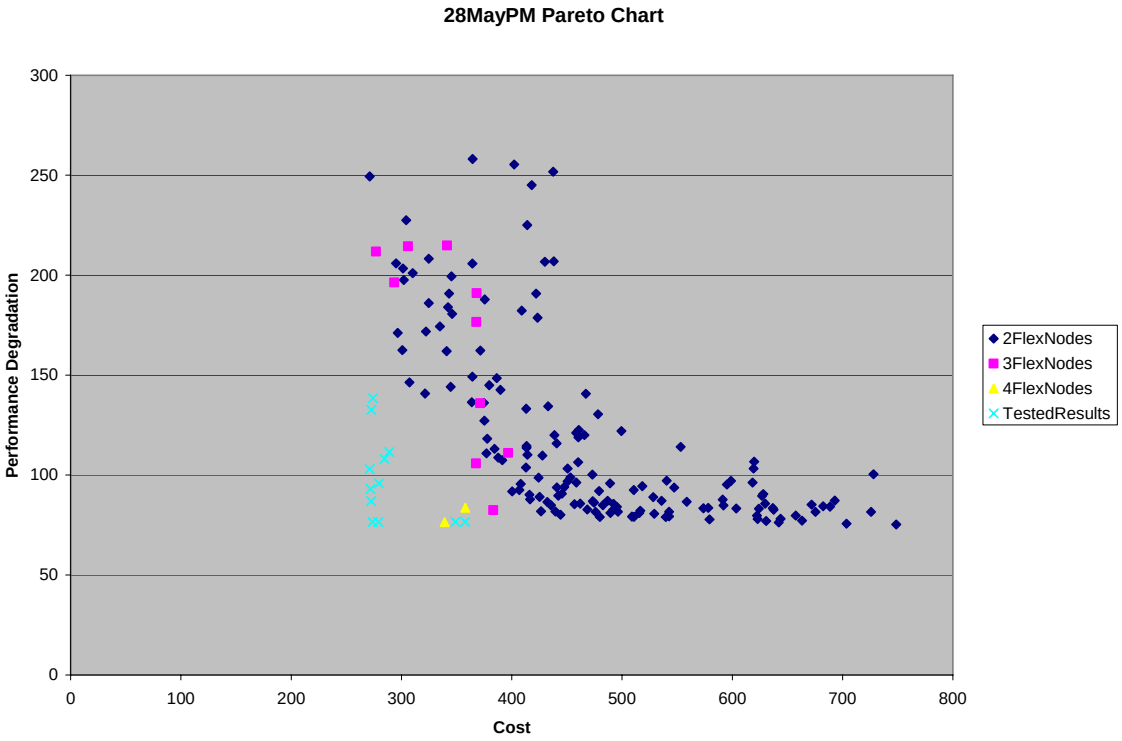


Figure 24 – Effect of Pattern Recognition on Cost and Performance Objectives

The results were obtained early in the research and subsequent tests showed the benefit for investment in pattern recognition is not always so large. Figure 25 depicts a run where a large number of points in the design space were evaluated before the neural net was applied.

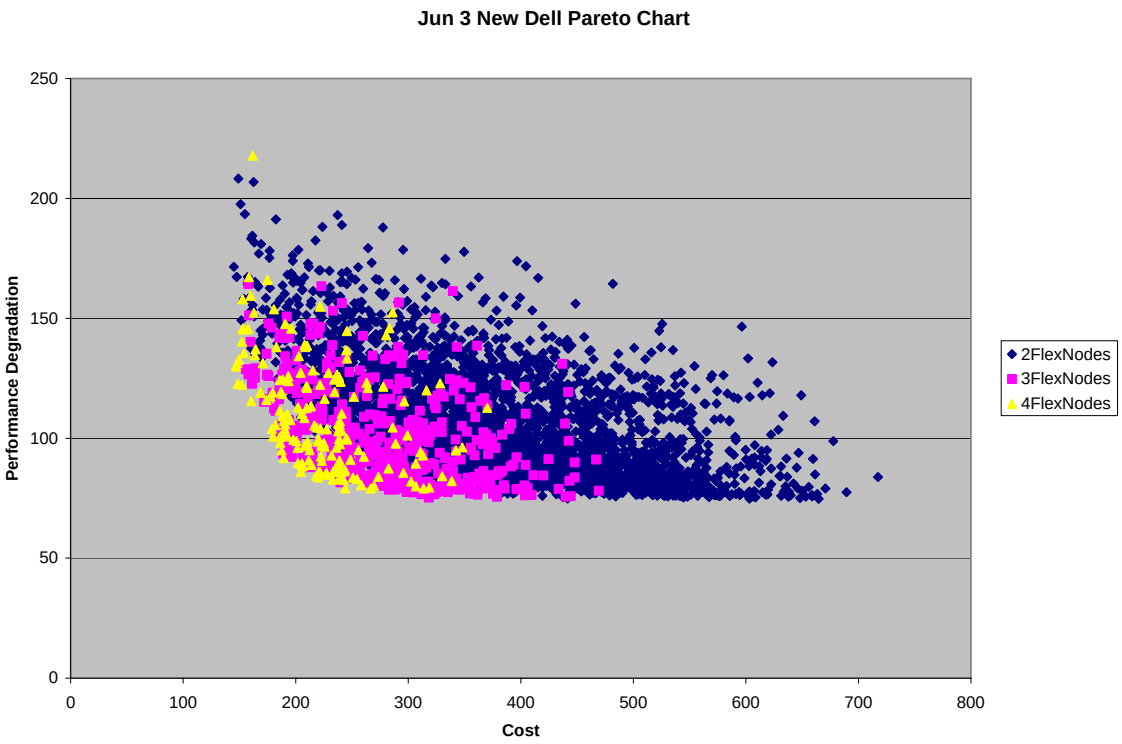


Figure 25 – Large Design Space Population Example

The advantage of using a larger number of flexible nodes is evident. Figure 26 shows the additional benefit gained after the system applied the neural net.

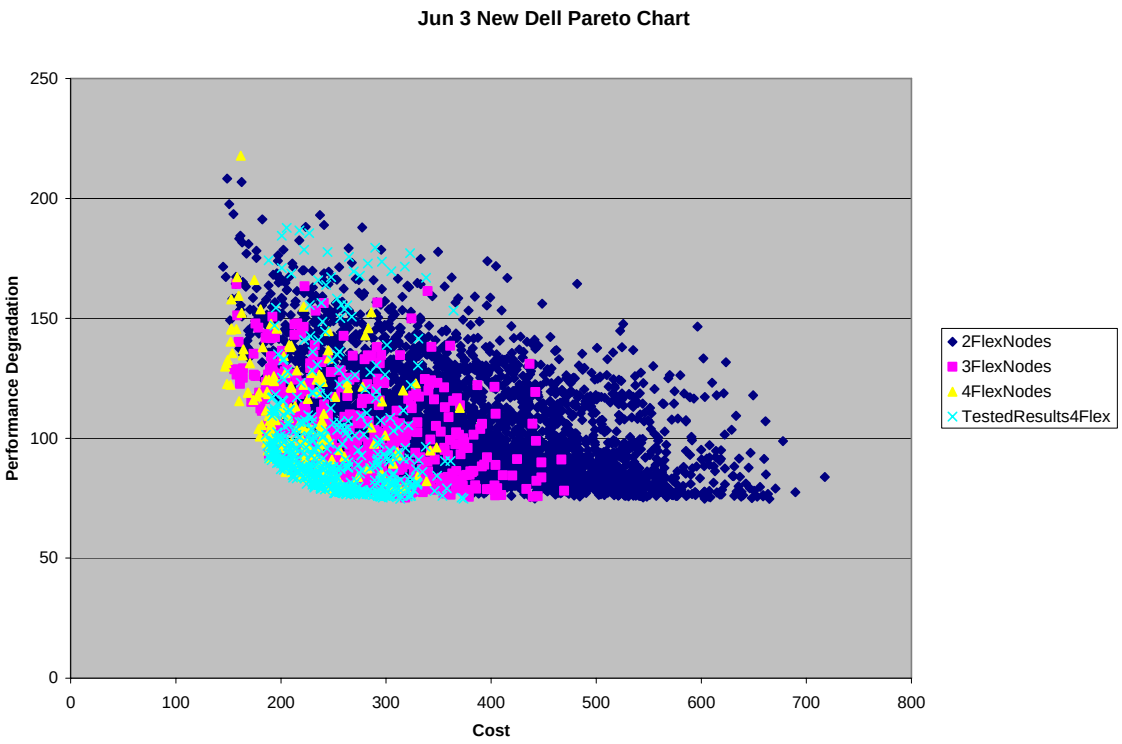


Figure 26 - Large Population Example Neural Net Results

The additional benefit in this case was small since the design space had already been fairly thoroughly searched before the neural net was applied.

Comparison to unassisted GA

The cost and performance of the 3 layer systems developed by the neural network assisted GA were compared to that of an unassisted GA. This was done as a first order rough estimate. A more rigorous description of the costs and benefits of using neural nets in this area is left for follow-on research. Returning to the process outlined earlier, the remainder of the flow diagram is copied below in Figure 27.

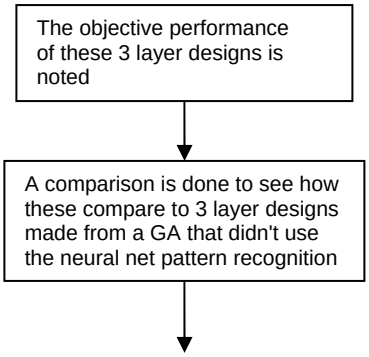


Figure 27 – Comparison Portion of Flow Diagram

The CPU time was recorded as an optimization was performed using both the neural network assisted GA and the unassisted GA. The objective results are shown in **Figure 28** and in Table 3.

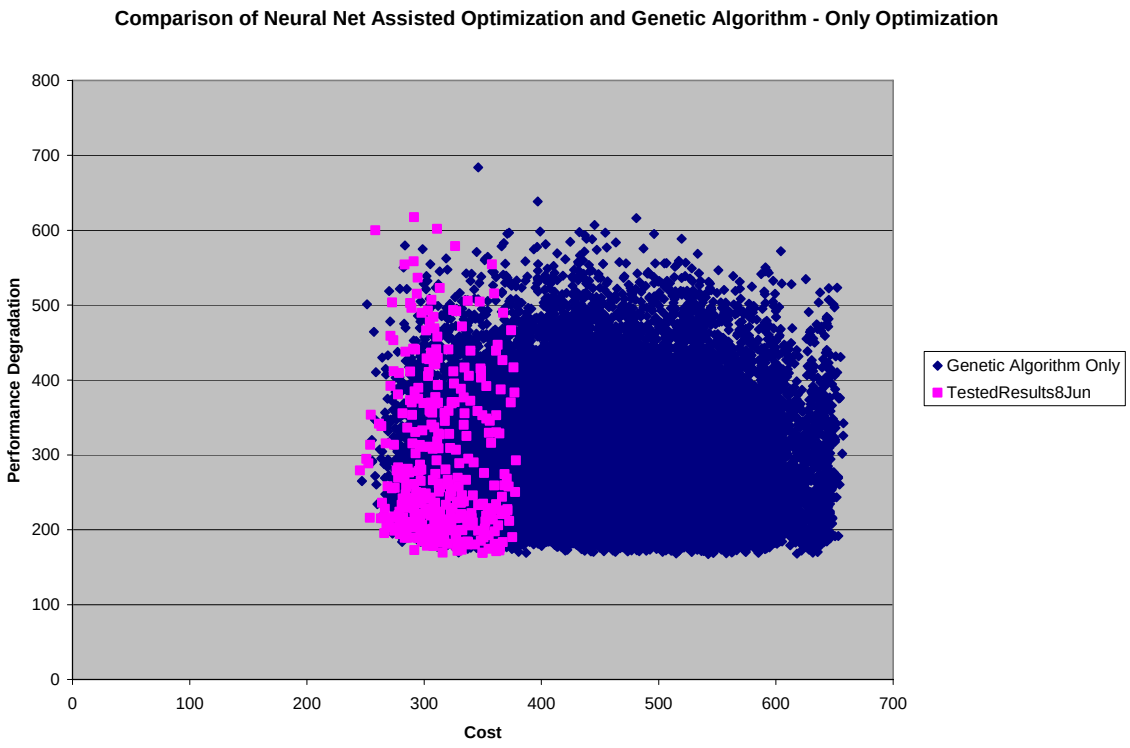


Figure 28 – GA / Assisted GA Comparison

Table 3 - GA / Assisted GA Comparison

	CPU time Hours	Number of Points on Combined Pareto Front, Figure 28	Lowest Cost	Lowest Performance Degradation
Unassisted GA	41	2	246.88	167.69
Neural Net Assisted GA	10	6	245.23	169.08

The points denoted with pink squares are the results from the neural net assisted genetic algorithm system. The points denoted by blue diamonds are the results from the unassisted genetic algorithm. While the best results from the GA-only algorithm are close to those of the neural net system, the neural net system nearly dominates the Pareto front; the lower left frontier. As shown in Table 3, the assisted genetic algorithm yielded three times the number of non-dominated solutions as the unassisted genetic algorithm

did. The GA-only algorithm ran for 41 hours of CPU time as measured by Matlab while the neural net system ran for 10 hours.[3] The genetic algorithm ran a series of optimizations, varying a number of parameters as shown in Appendix B. Each point is the result of one GA parameter set. As mentioned earlier, this was far from an exhaustive examination of genetic algorithms. It therefore does not prove that the optimization system that takes advantage of pattern recognition always outperforms all genetic algorithms and other optimization techniques. It does however strongly indicate that pattern recognition very well may provide superior results. These results are likely to be observed for optimization in general and for design optimization of systems that evolve over time in particular.

The fraction selected to be compared must be carefully chosen. If the fraction is too small, not enough of the design space is surveyed, and the opportunity to identify a promising 2 layer design may be missed. If the fraction is too large, processing time is wasted before the advantages of pattern recognition are applied. Also, a large fraction decreases the number of 2 layer designs left to develop into 3 layer designs. For example, in order to develop designs with 4 Flex nodes, there must be at least one 2 layer with 4 Flex nodes in the storedRecord3 and at least one in the selectedRecord3.

In addition to recognizing the effect flexible nodes have on the performance of the sensor network, the neural network was able to recognize the importance of selecting sensor nodes that were well spaced. Designs that clustered sensors together tended not to be selected by the neural network. Table 4 shows the x coordinate position of second

[3] Both runs were on the same Intel Pentium processor, 1,8 GHz, with 1.00 GB of RAM.

layer nodes. The first three data sets were a representative selection from the designs the neural network chose to evolve. The last three data sets were a representative selection from the designs the neural network chose not to evolve to a three layer design. As one would expect, the chosen designs have a better geometric spread than the designs that were not chosen.

Table 4 – x coordinate Position of Selected Designs

Data Set	1st x-coordinate	2nd x-coordinate	3rd x-coordinate
1	-5.073	2.0269	16.539
2	-5.5615	6.6318	18.808
3	-5.6873	2.343	16.092
4	1.8176	3.3083	9.8717
5	-1.1478	2.9087	10.774
6	-0.051476	2.9926	9.6339

This is further illustrated in Figure 29. The tendency to select designs that spread sensors was then studied by comparing the entire set of designs selected in this run against those not selected by the neural network. The average distance between nodes was found to be higher and the standard deviation was found to be lower for the chosen designs. Results are tabulated in Appendix D.

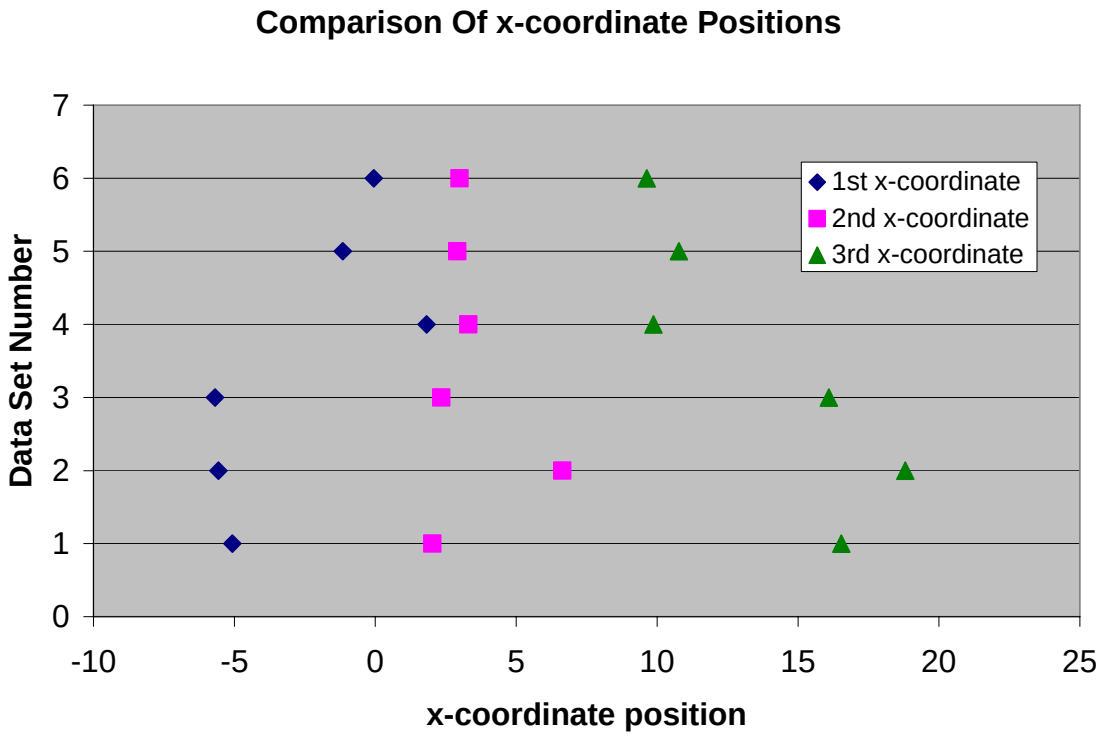


Figure 29 – Geometric Spread of Selected Designs

The genetic algorithm ran over a range of weight values. It was discovered that for optimizations that heavily favored cost, the genetic algorithms developed design solutions that did tend to cluster sensor nodes. Optimizations that weighted performance more heavily yielded more interesting layouts of sensor nodes that better illustrated the design trade offs between the competing objectives of performance and cost. Figure 30 shows the node positions as the weight for cost relative to performance was increased. For low weights, the points are spread nicely in the x dimension.

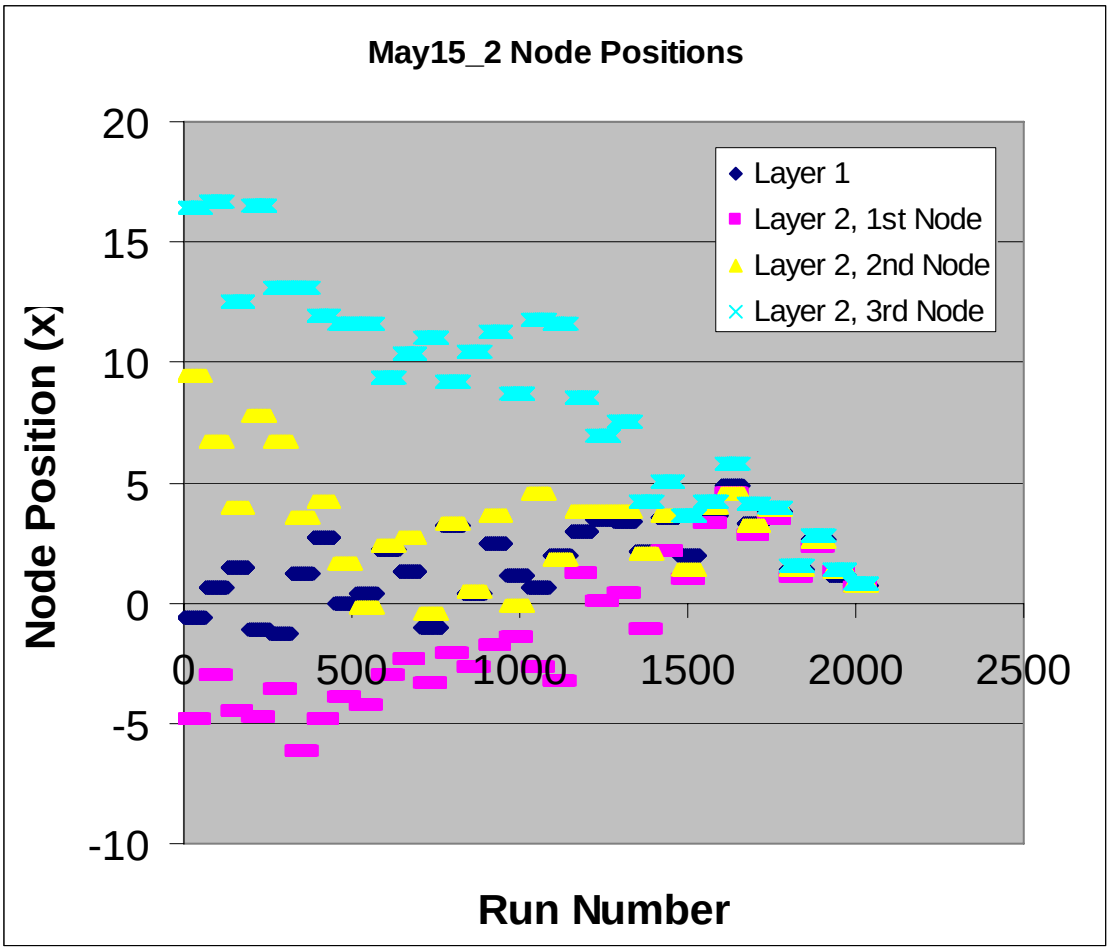


Figure 30 – Node Position Analysis

However, as the cost objective began to carry more weight, the points became clustered together, the distance decreased, allowing short length connections, but the performance suffered badly due to the large area in the field of interest that had no sensor nearby. The node positions were plotted using spreadsheets and the results for various weights are shown in Appendix A.

While linear weighted fitness functions worked satisfactorily, it took a long time to find appropriate weights. The optimal solutions as graphed in Figure 31 were down and to the left, toward the "utopia" point of zero cost and zero performance degradation. For a

given set of weights, design solutions with equal objective value fall along a straight line. The reader is referred to de Weck [29] for a discussion of isoperformance. Changing the weights changes the slope of the line. In this study, better results were achieved using a hyperbolic fitness function:

Cost ^ weight * performanceDegrade ^ (1-weight) **Equation 3**

The hyperbolic fitness function is shown in Figure 31. The affect of weights applied to hyperbolic functions is illustrated with the two fitness plots in the figure.

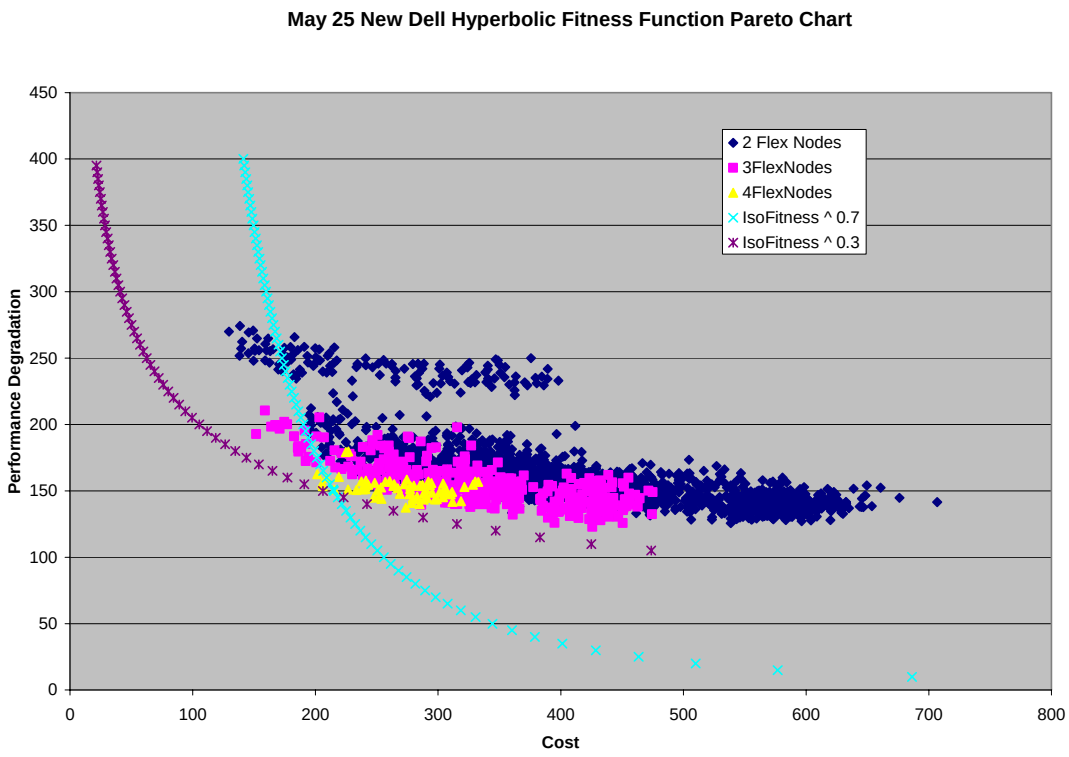


Figure 31 – Hyperbolic Fitness Function

The hyperbolic functions seemed to expose more of the Pareto front since the shape of the hyperbolic function was closer to the shape of the Pareto front. A detailed study of the effects of adjusting fitness function parameters is left for follow-on research. In general, hyperbolic functions seemed to expose larger sections of the Pareto front with fewer tuning adjustments than did linear functions.

The design problem required the sensor network to detect points within the sensor network's field of interest. Some of these points were randomly placed, changing from time to time while others were fixed. An experiment was run to see if changing the ratio of randomly placed nodes to fixed nodes affected the performance of the neural net system. There were 30 total points to detect. The experiment was run with 2, 4, and 8 random nodes. In this limited examination, the neural network had a larger impact on the system with a higher ratio of random nodes. Detailed results are shown in Appendix F. Intuitively the results makes sense; with more variable environments, the value of a neural net to see through the random effects to the underlying driving patterns ought to be high. Further research is required before any firm conclusion may be reached however.

Evolution vs. Extension

2.0 Evolution is defined as a system that changes function, whereas extension is defined as a system that grows, but does not change function. Up to this point, the examples have started with extensible two layer systems. The next test was done on an evolving two layer system. The system was exercised using a problem where the third layer sensors had a passive function. Instead of transmitting and receiving an emission, they only received. Their performance was therefore modeled as proportional to range from the

point to detect squared rather than range raised to the fourth power. This is admittedly a small change in function, but it illustrates an interesting point. The results are shown in Figure 32:

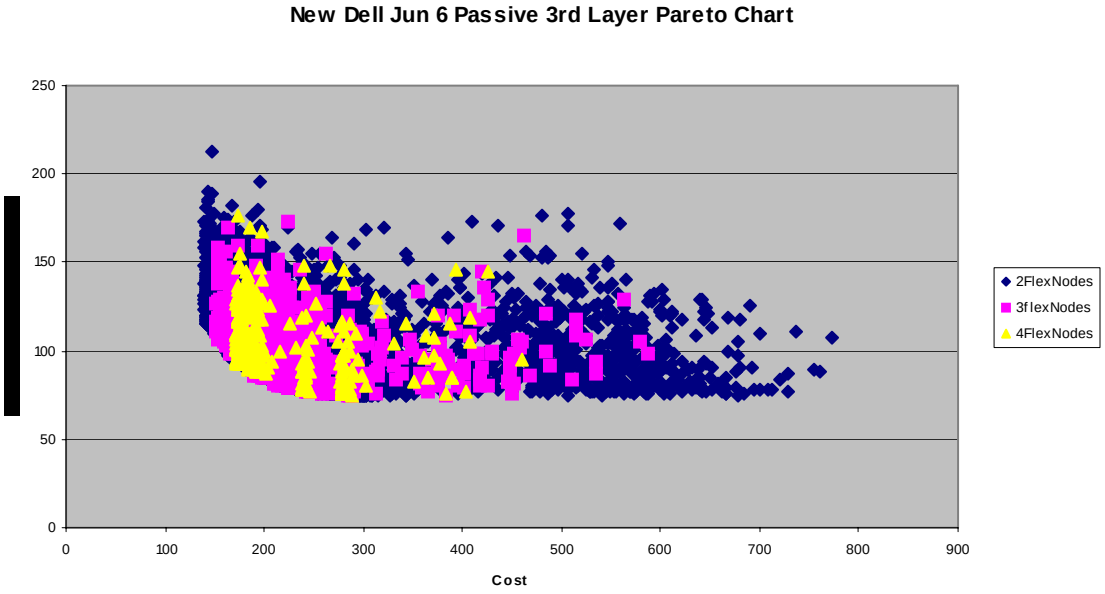


Figure 32 – Passive Layer Results

By making this change in the function, the value of flexible infrastructure was decreased to the point where it was barely perceptible. This makes sense considering the magnitude of the sensor-to-point range terms contributing to the system performance. Since the lower layer sensor to point ranges were raised to a higher power than those of the third layer in the performance computation, the position of the third layer sensors had negligible effect on the performance. Therefore, the flexibility to spread these sensors across the area of interest became unimportant. What the neural net had learned in the two layer case became unimportant when the system evolved as the function changed.

Now another test was run using an altered fitness function. A passive sensor was used, but the importance of the distance between the passive sensor and each point to detect was multiplied by 10,000. This was not based on a physical model, but it is reasonable to assume that the fitness of different sensors may vary widely with range. For example, a passive electro-optical (EO) or infrared (IR) sensor may be more sensitive to range under some conditions than an active RF sensor. The results using this fitness function are shown in Figure 33

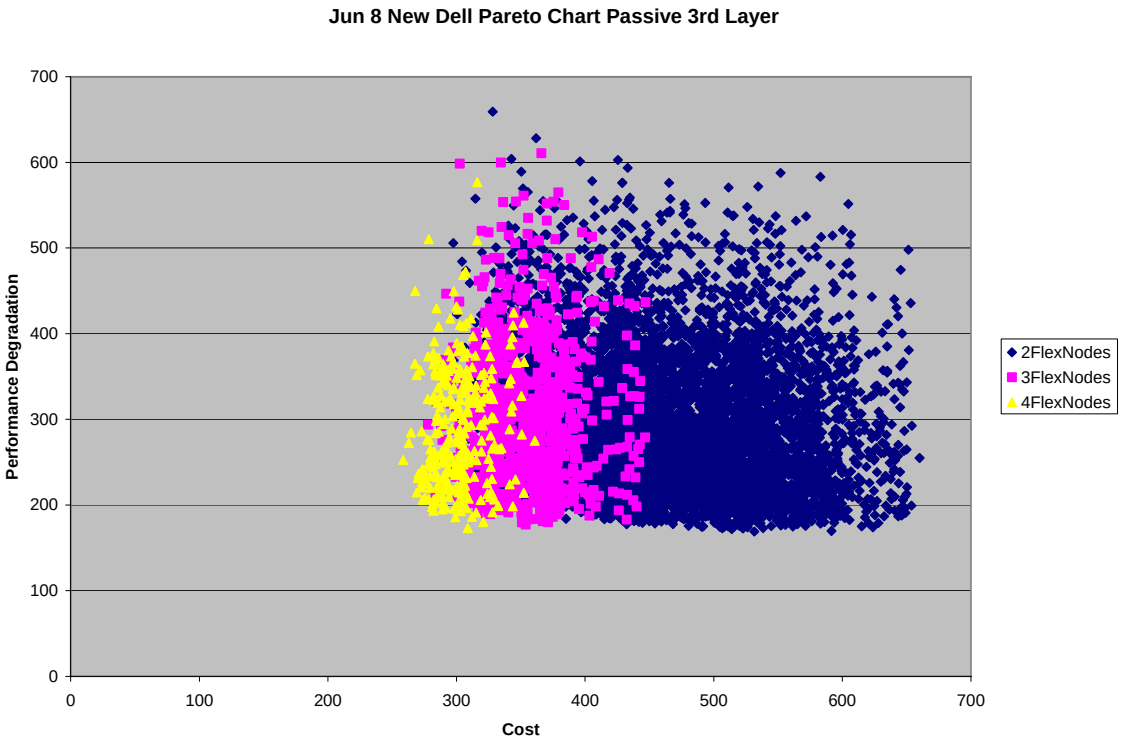


Figure 33 – Scaled Passive Layer Results

The value of the flexible structure is quite evident in this case. The neural network recognized the value of flexibility and focused on designs that used 4 flexible nodes. The system yielded the results shown in figure after a moderate number of runs:

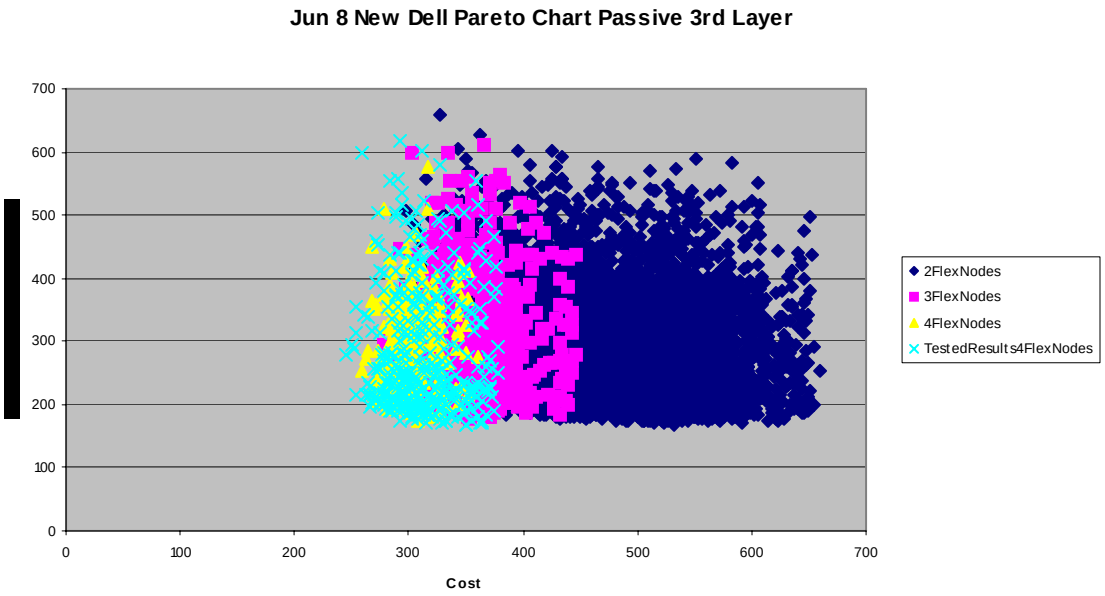


Figure 34 – Scaled Passive Layer Neural Net Assisted Results

Note that the number of design solutions yielded in this "TestedResults" set is relatively small, yet the Pareto front is made entirely of these points. This again demonstrates the value of applying pattern recognition to optimization.

Thus, one can see that sometimes patterns continue as a system is evolved and other times patterns do not. So of course the value of an optimization system that uses pattern recognition is high in some problems and not in others.

Future Work

3.0 This study was a preliminary exploration into an area that appears to offer significant opportunity for fruitful research. Following are recommendations for further study:

3.1 The sensor system was optimized for the best performance of the last configuration. More work could be done to study the best performance for the life of the design, so the current performance should also be considered. The value of money over time should be included in the study, so that value of future costs and benefits are discounted relative to current costs and benefits.

3.2 Tests should be run on higher fidelity models that more closely match the physical world. This was a good starting point to explore pattern recognition, flexibility and design evolution. Building higher fidelity models will further illuminate the degree to which this approach is beneficial and will help direct future research. Using the system on problems with more objectives would beneficial. Recommended higher fidelity models for follow-on study include satellite systems, advanced sensor systems, instrumentation systems in machinery, system platform design, and the effect a design's flexibility is expected to have on meeting future market demand.

3.3 Other optimization techniques should be used in conjunction with this approach. Leyland's queueing multi-objective optimizer (QMOO) [30] is one highly recommended optimizer. This could be done using the Distributed Object-based Modeling Environment (DOME) [31]. Using DOME will not only allow the system to plug into different optimizers like QMOO seamlessly, it will also allow a larger audience to use and work on this technology. DOME has been used with the Struggle GA [32]. Struggle would also probably yield results significantly better than those of the GA used in this paper. Optimization need not be limited to genetic and evolutionary algorithms. Other optimization techniques like Sequential Quadratic Programming (SQP) could be included.

3.4 More experimentation should be done with objective and fitness functions. In particular, the hyperbolic fitness functions should be examined in more detail. Current computer simulation and optimization systems can only accomplish a limited portion of the design workload. While they can accomplish an enormous amount of processing, they require constant adjustment and careful monitoring to ensure they remain headed in the correct direction. Shifting more of the design workload from the designers to the computer system requires better definition of what is desired, of the degree to which different performance values are desired and of how preferences change as factors in the system change.

3.5 More efficient graphical user interfaces (GUIs) can be developed for this system. Increasing the efficiency of the interface between the designers and the computer system appears to be an important area for future research.

3.6 Study can be accomplished that would yield insight into when pattern recognition can best be applied and under what conditions it is likely to yield a significant improvement.

3.7 The system should be made more efficient and a thorough examination of efficiency should be accomplished to determine when its computational cost is worth the benefit gained.

Conclusion

This study of the optimization of systems that evolve over time surfaced a number of interesting questions and yielded several conclusions as well. It was hypothesized that systems that are well-positioned to evolve have characteristics that distinguish themselves from systems that are not well-positioned to evolve. In this particular study, the basic

sensor system illustrated the importance of flexibility to the design of systems that evolve over time. The neural network assisted GA was able to discern patterns between the design inputs and objective outputs of the evolving design. It realized that for this design problem, a more flexible structure yielded better results in the long run despite the higher initial cost of flexible nodes. In the example problem chosen, it was fairly easy to recognize a pattern and to pick designs that would evolve well. The positive results of this study suggest that in general systems design, there are other patterns beneath the surface that are more difficult to discern, but that can be revealed using neural networks.

Both extensible and evolutionary systems were explored. An illustration showed that although some patterns continue as a system is evolved, that is not always the case. In one example, a sensor network design was evolved to one that included passive sensors. The pattern of higher flexibility yielding more optimal solutions did not hold for that particular example.

A fuzzy Pareto algorithm was developed and successfully used to improve the performance of the optimization. Pareto analysis was shown to be a useful way to visualize results for this sensor system design problem.

The genetic algorithm ran over a range of weight values. It was discovered that for optimizations that heavily favored cost, the genetic algorithms developed design solutions that tended to cluster sensor nodes. Optimizations that weighted performance more heavily yielded more interesting layouts of sensor nodes that better illustrated the

design trade offs between the competing objectives of performance and cost. Except for the runs in which the cost objective was heavily weighted, the neural network tended to select designs with larger geometric spreads between nodes.

The neural network was able to recognize a pattern whereby flexible sensor networks evolved more successfully than less flexible networks. The optimizing algorithm used this pattern to select candidate systems that showed promise for successful evolution. In this limited exploratory study, a genetic algorithm assisted by a neural network achieved better performance than an unassisted genetic algorithm did. The assisted GA yielded three times the number of optimal design solutions on the Pareto front as the unassisted GA and completed its processing in one quarter the CPU time.

Appendix A

Matlab Code

```
%© Massachusetts Institute of Technology – Michael Nolan
% Portions of Matlab GA tool and neural network code used.
%This runs a Matlab GA, cycling through a number of parameters in order to
%find acceptable performance

inputParameters = xlsread('InputDataFile.xls','sheet2');

w_LowerLimit = inputParameters(1);
w_StepSize = inputParameters(2);
w_UpperLimit = inputParameters(3);
n_LowerLimit = inputParameters(4);
n_StepSize = inputParameters(5);
n_UpperLimit = inputParameters(6);
flexibleSensorCost = inputParameters(7);
inflexibleSensorCost = inputParameters(8);
costPerLength = inputParameters(9);
x_LowerLimit = inputParameters(10);
x_UpperLimit = inputParameters(11);
fractionOne = inputParameters(12);
nrOfRandomRuns = inputParameters(13);
w_LowerLimit3rdLayer = inputParameters(14);
w_StepSize3rdLayer = inputParameters(15);
w_UpperLimit3rdLayer = inputParameters(16);
n_LowerLimit3rdLayer = inputParameters(17);
n_StepSize3rdLayer = inputParameters(18);
n_UpperLimit3rdLayer = inputParameters(19);
nrOfRandomRuns3rdLayer = inputParameters(20);
w_LowerLimitGet3From2 = inputParameters(21);
w_StepSizeGet3From2 = inputParameters(22);
w_UpperLimitGet3From2 = inputParameters(23);
n_LowerLimitGet3From2 = inputParameters(24);
n_StepSizeGet3From2 = inputParameters(25);
n_UpperLimitGet3From2 = inputParameters(26);
nrOfRandomRunsGet3From2 = inputParameters(27);
fuzzyLevel = inputParameters(28);
selectionFraction = inputParameters(29);
fitnessWt = inputParameters(30);

options = gaoptimset('PopInitRange',[[-15 -15 -15 -15 -15 -15 -15]; ...
   [27 27 27 27 27 27 27]]);
rand('state', 71); % These two commands are only included to
randn('state', 59); % make the results reproducible.
```

```
record=[];  record2=[];  record3=[]; x=[];

for w = w_LowerLimit: w_StepSize: w_UpperLimit;  %w is the weighting between
objectives
  %higher w puts more emphasis on cost.
  indexNr=1;
  record2=[];

  wtCostLen2Layer(1) = w;   %objective function weighting
  wtCostLen2Layer(2) = 5;   %time
  wtCostLen2Layer(3) = flexibleSensorCost;
  wtCostLen2Layer(4) = inflexibleSensorCost;
  wtCostLen2Layer(5) = costPerLength;
  wtCostLen2Layer(6) = x_LowerLimit;
  wtCostLen2Layer(7) = x_UpperLimit;
  %_____
  for n = n_LowerLimit: n_StepSize: n_UpperLimit

    for gens = 300:100:300
      options = gaoptimset('Generations',gens);
      for popsize=30:30:30
        options = gaoptimset('PopulationSize',popsize);
        for select = 1:2
          if select == 1
            options = gaoptimset('SelectionFcn', @selectionstochunif);
          else
            options = gaoptimset('SelectionFcn', @selectiontournament);
          end

          [fwriteid] = generateMultObjMainInputFile(wtCostLen2Layer);

          %  To plot the GA as it converves on a solution:
          %   options = gaoptimset(options,'CrossoverFraction', n,'PlotFcns',
@gaplotbestf);
          %  To run without the plot:
          options = gaoptimset(options,'CrossoverFraction', n);
          [x weightedFitness] = ga(@multiObjectiveMain, 7, options);
          %Note that the GA generates real numbers for all design variables,
          %including the flexible variables.  Flexible variables really
          %should be Boolean; either a 1 or a 0.  The main program converts
          %the real flexible values to Booleans at the beginning of the
          %function and calculates the fitness function using Boolean values.
          %This GA function will return the real value it sent to the main
          %function though, not the Boolean 1 or 0 the main function used in
          %its computation.
```

```
            % x is the vector of values returned by the GA.  It is the design
            % vector which includes position and flexible/not flexible
            % information.  Each sensor node therefore has 2 values, an
            % x-coordination

            %May 7 change:  to fix the lack of a variable for layer 1 flexible spot:
            for k=1:4
              x2(k) = x(k);
            end
            x2(5)=1;
            for k=5:7
              x2(k+1)=x(k);
            end

            [x] = makeFlexValues1or0(x2);
            [x] = checkAtLeast1Flex(x);
            [constrained_x] = constrain_x(x);
            [nodesSorted] = sort_a(constrained_x);  %nodesSorted is a matrix of the
            %x y pairs sorted
            sorted_x = nodesSorted(:,1);  %(Taking just the x values of the 2 column
matrix)
            flex_x   = nodesSorted(:,2);

            sorted_x_1Column = [sorted_x; flex_x]; %A column form of the design
            %vector with the sorted xCoordinates on top and the
            %flexible variables on the bottom.
            level=[1 3 3 3 ];
            [dontcare size_x]  = size(x);
            nrOfNodes = size_x/2;
            for i=1:nrOfNodes
              sensorMatrix(i,:)=[nodesSorted(i,1)  level(i)   nodesSorted(i,2)];
            end

            record2(indexNr,:) = sorted_x_1Column';  % note transpose

            indexNr = indexNr+1;
          end
        end
      end
    end
    record3 = cat(1, record3, record2); %Adds record 2 onto the bottom of record3.
                                %Records each run.
end

%Only used for diagnostics:
%Plot the values of weightedFitness against the crossover fraction with the
```

```
%following commands:
%plot(0:.05:1, record);
%xlabel('Crossover Fraction');
%ylabel('Weighted Fitness');

%A fraction of the designs are explored further and the remainder are
%stored.  The neural net will return to the stored patterns to see which
%ones look promising to develop into 3 layer designs.

[selectedRecord3, storedRecord3] = selectFractionOne(record3, fractionOne);

%(this runs 3 or 4 times to allow the random input point to detect
%to show up in different spots in the data used to train the neural net

[JallAverage] = getJallAverage(nrOfRandomRuns, selectedRecord3);
%getCostAndPerf works for 2nd Layer only

% Added for testing only:
tempTest=[JallAverage, selectedRecord3];

minmax = [-1; -1];   %minimize cost, minimize performance degradation

[dmatrix, dvector, ndvector] = paretosort(JallAverage', minmax);

[fuzzyParetoSet] = fuzzyPareto(ndvector, fuzzyLevel, selectedRecord3);

[dontCare, JfuzzyPareto] = getCostAndPerformVals(fuzzyParetoSet);

[fwriteid]=generate3LayerInputFiles(fuzzyParetoSet);

[nrOfFuzzySetsIn, nrDesignVariables] = size(fuzzyParetoSet);

ThreeLayerRecord2L=[];
ThreeLayerRecord3L=[];

indexNr=1;
% The 2 layer fuzzyParetoSet design candidates are now extended to 3 layer
% designs using the genetic algorithm.
%   - runNumber3L is incremented each time the program loops through a
%     fuzzyParetoSet 2 layer design candidate.
%   - w, the weighting of objectives, is increased incrementally
%   - n, the GA crossover fraction, is increased
%   - indexNr increments each time, so there is a unique index number for
%   each of the sets run
%   objective fitness values are not recorded in this section of the
%   program
```

```
% for runNumber3L = 1:2;                %to run only 2 sets for debugging.
for runNumber3L = 1:nrOfFuzzySetsIn;   %commented out to run only 2 sets
  %for debugging.  Each FuzzySetIn is associated with one 2 layer design,
  %(one ThreeLayerRecord2L)
  [fwriteid] = makeFileFor3rdLayerRun(nrOfFuzzySetsIn, nrDesignVariables,...
    runNumber3L);
  % This is an input file for the 3rd Layer Run.
  % The fwriteid variable isn't important outside these functions, but Matlab requires
  % something to be returned.

  x=[];

  w_LowerLimit3rdLayer = inputParameters(14);
  w_StepSize3rdLayer = inputParameters(15);
  w_UpperLimit3rdLayer = inputParameters(16);
  n_LowerLimit3rdLayer = inputParameters(17);
  n_StepSize3rdLayer = inputParameters(18);
  n_UpperLimit3rdLayer = inputParameters(19);
  nrOfRandomRuns3rdLayer = inputParameters(20);
  x_LowerLimit = inputParameters(10);
  x_UpperLimit = inputParameters(11);

  for w = w_LowerLimit3rdLayer: w_StepSize3rdLayer: w_UpperLimit3rdLayer;

    wtCostLen3Layer(1) = w;   %objective function weighting
    wtCostLen3Layer(2) = 5;   %time
    wtCostLen3Layer(3) = flexibleSensorCost;   %flexibleSensorCost   %

    wtCostLen3Layer(4) = inflexibleSensorCost;   %inflexibleSensorCost
    wtCostLen3Layer(5) = costPerLength; %costPerLength
    wtCostLen3Layer(6) = x_LowerLimit;  %len1
    wtCostLen3Layer(7) = x_UpperLimit;   %len2
    [fwriteid]=generateMultObjMainInputFile2(wtCostLen3Layer);

    for n = n_LowerLimit3rdLayer : n_StepSize3rdLayer : n_UpperLimit3rdLayer
      options = gaoptimset(options,'CrossoverFraction', n);
      options = gaoptimset('PopInitRange',...
        [[-10 -10 -10 -10 -10 -10];...
        [ 20  20  20  20  20  20]]);
      [x, weightedFitness] = ga(@multiObjectiveMain3rdLayer, 6, options);

      [x] = makeFlexValues1or0Level3(x);
      [x] = checkAtLeast1FlexLevel3(x);
      [constrained_x]=constrain_x(x);
      [nodesSorted]=sort_a3L(constrained_x);
```

```
        sorted_x = nodesSorted(:,1);     %(Taking just the x values of the 2 column matrix)
        flex_x   = nodesSorted(:,2);
        %Place 3rd Layer x values and flex values in a single column:
        sorted_x_1Column = [sorted_x ;flex_x];
        %                   record2(indexNr,:) = sorted_x_1Column;
        %  each column of ThreeLayerRecord3L is a design variable - position,
        %  flexible or not etc.
        %  each row of ThreeLayerRecord3L is a design vector - the design vector
        %  from run #1, #2 etc.
        % indexNr isn't required to be in the record, but it's
        % easier to write and debug with it there.
        % fuzzyParetoSet(runNumber3L,:) is the 2 Layer Pareto Set design vector
        % x is the 3 layer design vector, so
        % ThreeLayerRecord3L includes the whole design.

        ThreeLayerRecord3LAdd=[indexNr, runNumber3L,
fuzzyParetoSet(runNumber3L,:), sorted_x_1Column'];
        %ThreeLayerRecord3LAdd=[indexNr, runNumber3L,
fuzzyParetoSet(runNumber3L,:), x];

        % Accumulate the ThreeLayerRecord3LAdds in one matrix:
        ThreeLayerRecord3L = cat(1, ThreeLayerRecord3L, ThreeLayerRecord3LAdd);

        indexNr = indexNr+1;
      end
    end
end

[JallAverage, variableCostVector, fixCostVector, totalSensorsCostVector]...
   = getJallAverage3Layer(nrOfRandomRuns3rdLayer, ThreeLayerRecord3L);

% Feed the data into the neural net: the 2 layer net, the corresponding
% 3 layer nets built on top of the 2 layer net, the cost and the
% performance.  Find patterns.

%For this thesis, We'll look at the pattern between the 2 layer net (which is the first
%12 columns of ThreeLayerRecord3L) and JallAverage.

flexible2LMatrix = ThreeLayerRecord3L(:,7:10); %flexible2LMatrix is the 2 layer
%flexible portion of
%3LayerRecord3L (columns 7-14).
costThreeLayerRecord3L = JallAverage(:,1);     %costThreeLayerRecord3L is the
%cost associated with each of the
%3LayerRecord3L records.
%Note that performance is
%not affected by whether the
```

```
%nodes are flexible or not.

perfDegrad3LayerRecord3L = JallAverage(:,2);
fitness3LayerRecord3L = perfDegrad3LayerRecord3L.^(1-fitnessWt) .*   ...
  costThreeLayerRecord3L.^fitnessWt;
%Note that the .^  and .* are element-wise operators (operate on each
%element individually, rather than matrix multiplication.)

%USE PARETO ANALYSIS INSTEAD OF LOOKING FOR 1 BEST SOLUTION:
[allSame]= inputCheck(flexible2LMatrix)
if allSame ==0

  % Instead of looking for lowest cost, the NN will look for best fitness
  %
  [predictedResults]= mainNeuralNet(flexible2LMatrix, fitness3LayerRecord3L,
storedRecord3);

else
  error='All rows flexible2LMatrix the same.'
  predictedResults = costThreeLayerRecord3L(1);
end

%predictedResults are the predicted objective values of the given candidate
%2 Layerdesigns.  They are determined using bestNet, the net that did the
%best job finding a pattern. Each predicted result record includes fields
%for the candidate 2 layer design that led to the three layer objective
%result.

%sortCandidatesAndResults is a sorted list of the predictedResults and
%candidates

[selectedCandidates] = selectCandidates(storedRecord3, predictedResults,
selectionFraction);

%Stored Record3 was the large population of designs that was initially set
%aside.  The neural net searches through this population to find designs
%that match patterns of successful designs and therefore show potential.
%Note that for maximum effectiveness, the best design from the stored
%Record should be compared to the best design found previously in selected
%record, just in case a design from the small initial population is better
%than any of the designs from the large population.  I don't compare them
%however, since I'm just interested in showing the validity of this
%approach and not attempting to find the absolute maximum performing
%optimization scheme.

%testPredictedResults tests the top recommended 2 layer designs by running
```

```
%them back through the sensor model, building a 3 layer design from each 2
%layer design.
%
%Compare these results with the results of optimization schemes that just
%bang out GAs from all the possible starting 2 layer designs instead of
%starting from only those that exhibit a promising pattern for extention.

[testedResults] = Build3LayerFromCandidates(selectedCandidates, inputParameters);
[bestResults] = getBest3Layer(testedResults);




%function [bestnet]=mainNeuralNet(flexible2LMatrix, costThreeLayerRecord3L)
function [predictedResults]=mainNeuralNet(flexible2LMatrix, ...
                        costThreeLayerRecord3L, storedRecord3);

%The program cycles through a range of
%parameters, running the net until it finds parameters that yield adequate
%performance. It trains, then runs the neural net.  The predicted results
%for the stored Record3's are returned.  The prediction is based on the
%pattern the the neural net discerns between flexible2LMatrix and
%costThreeLayerRecord3L.

%This uses either a 2 or 3 layer backprop. There's no provision for 4, 5 etc.
   %S1 is the number of nodes in level 1 of the net etc.
   %If S3 is 0 of course it's a 2 layer net
   %BTF is the backprop training function, default is 'traingdx'
   %it'll use 'learngdm' weight/bias learning function as a default
   %performance function is 'mse'

%These are matrices with first, increment and last values:
S1Values = [4, 1, 4];
S2Values = [8, 1, 8];
S3Values = [1 1 1];        %to make it a 2 layer, set S3=0, TF3 can be anything
TF1Values = {'logsig';'tansig'};
TF2Values = {'logsig';'tansig'};
TF3Values = {'logsig';'tansig'};

%Particular BTF parameter values (list all the values here, then choose to run them by
```

```
%calling their position in the vector)
%These are the parameters that are specific to particular BTFs:
[special_Parameters,SPcount]= specialParameters();

maxEpochsValues = [170, 1, 170];  %default 5000
goalValues = [0.005, .02, 0.045];
momentumValues = [.8 .1 .9];
learnRateValues = [0.01 1 0.01];
learnRateIncreaseValues = [1.05 1 1.05]; %This is the ratio to increase
%learning rate. Matlab default value is 1.05.  Positive real value greater
%or equal to 1.0. For the TF values and the BTF values, a list of values to
%try may be entered. Note that if I only want to try 1 value, set the first
%= last.  The increment must not be set to 0 though.

runNumber = 1;
for S1 = S1Values(1):S1Values(2):S1Values(3)
  for S2 = S2Values(1):S2Values(2):S2Values(3)
    for S3 = S3Values(1):S3Values(2):S3Values(3)
      for i = 1:1
        TF1 = cast(TF1Values(i), 'char');
        for j = 1:1
          TF2 = cast(TF2Values(j), 'char');
          for k = 1:1
            TF3 = cast(TF3Values(k), 'char');
          end

% Cycle through the combinations of BTFs and special BTF parameters:
              for ii=1:SPcount
                theseSpecialParameters=special_Parameters{1,ii};
                 for
maxEpochs=maxEpochsValues(1):maxEpochsValues(2):maxEpochsValues(3)
                    for goal=goalValues(1):goalValues(2):goalValues(3)
                       for
momentum=momentumValues(1):momentumValues(2):momentumValues(3)
                          for learnRate=learnRateValues(1):learnRateValues(2):...
                              learnRateValues(3)
                            for learnRateIncrease=learnRateIncreaseValues(1):...
                                learnRateIncreaseValues(2):learnRateIncreaseValues(3)

BLF = 'learngdm';
PF = 'mse';

%now define, train, test and maybe validate a net using the selected
%net design and architecture parameters:
[net, performanceLog] = defineAndRunNet4(S1,S2,S3, TF1,TF2,TF3
,theseSpecialParameters, ...
```

```
maxEpochs, goal, momentum, learnRate, learnRateIncrease,...
flexible2LMatrix, costThreeLayerRecord3L);

%Note that the defineAndRunNet2 writes the NN parameters and corresponding
%performance to a binary file. resultsLog keeps the data from this entire
%program run in cell arrays. performanceLog contains the data from the last
%function call to defineAndRunNet.

resultsLog(1, runNumber) = {performanceLog};
resultsLog(2, runNumber) = {net};

runNumber = runNumber +1;
                                end
                            end
                        end
                    end
                end
            end

                    end
                end
                end
            end
end

%This section determines the neural net that found the best pattern - the
%pattern with the lowest error
bestPerformanceValue = 100000000;  %default value.
bestPerformanceIndex = -1; %This can be used to check for an error
[dontCare, nrResults] = size(resultsLog) %nrResults is the number of records in
resultsLog
for i=1:nrResults
   for j=1:14
      cell_ij_value = resultsLog{1,i}{1,j};
      newLog{i,j}= cell_ij_value;
   end
   if  bestPerformanceValue > newLog{i,13}
      bestPerformanceValue = newLog{i,13};
      bestPerformanceIndex = i;
   end
end
for j=1:14
bestParmeters{1,j} = newLog{bestPerformanceIndex,j};
end
%Return the net that yielded the best results:
```

```
%This returned an error for some reason:
[bestNet] = resultsLog(2, bestPerformanceIndex); % resultsLog row 2 is "net"
%"one or more output arguments not assigned during call to mainNeuralnet"

%Note that this is not necessarily the fastest net and that other
%attributes may make other nets more attractive depending on the situation.

candidateSensorSets = takeFlexibleVariables(storedRecord3);
[predictedResults]=runNet(bestNet{1,1},candidateSensorSets);
```

```
function [testedResults] = Build3LayerFromCandidates(selectedCandidates,
inputParameters)

[nrCandidates, nrDesignVariables] = size(selectedCandidates);
[JallAverage, ThreeLayerRecord3L] = get3LayerFrom2Layer(nrCandidates, ...
   nrDesignVariables, selectedCandidates, inputParameters);

testedResults = [JallAverage,ThreeLayerRecord3L];
```

```
function[group] = categorizerFlex(xCoordinate, midpoint, nrOnLowerLayerFlex, group);

[dontCare, nrOfMidpoints] = size(midpoint);
nrOnThisLayerFlex = nrOnLowerLayerFlex;
%nrOnThisLayerFlex = xx+1;  %number of sensors on the lower layer of the Flexible
network.

%determine if the group/lower layer point to which the upper layer point is connecting to
%should be incremented:

if group <= nrOfMidpoints
   while midpoint(group) < xCoordinate      %xCoordinate of upper layer point
        group = group +1;
         if group >= nrOnThisLayerFlex    % if already at highest group, don't
                                  % ask for next midpoint
```

```
            break
          end
   end
end
```

```
function [x]=checkAtLeast1Flex(x);

%May 7 change:
%check to ensure at least one flexible node is on each level:
%       if (x(7) < 1 & x(8) < 1)
%          x(7) = 1;
%       end
       if (x(6) < 1 & x(7)<1)
          x(8) = 1;
       end
```

```
function [x]=checkAtLeast1FlexLevel3(x);

% May 7 change:
%for i=11:20
for i=4:6

  if x(i)>=1
       isFlex=1;
       break
   else
       isFlex=0;
   end
end
   if isFlex ~=1
   x(5)=1;
   end
```

```
function [combinedRecord] = combineRecord(resultsLog, nrOfRecords)
% combine the special parameters in the cell array, cell 14 with the rest of
```

```
% the data into a combined record cell array:
combinedRecord = cell(1,23);
for i=1:nrOfRecords
   for k=1:13
      combinedRecord{i,k}=resultsLog{1,i}{1,k}
   end

   for k=1:10
         combinedRecord{i,k+13}=resultsLog{1,i}{1,14}(k)
    end
end
```

```
function [x]=constrain_x(x);
%Apply constraints so the xCoordinates aren't more than 1 from the area of
%interest (which is -5   to   +9)     Todo: make this more efficient by
%combining into loop with checkAtLeastOneFlex

[dontcare size_x]  = size(x);

for i=1:size_x
   if x(i) < -15          %Todo - does this cause problems with c values close to -6?
      x(i) = -15;         %setting the objective function to a bad fitness value didn't seem to
work well.
   end
   if x(i)> 27          %Todo - make this a value in the binary file so it can be
      x(i) = 27;              %adjusted in order to run tests on ranges other than -5 to 9.
   end
end
```

```
function [net, performanceLog] = defineAndRunNet4(S1,S2,S3, TF1,TF2,TF3,
theseSpecialParameters,...
   maxEpochs, goal, momentum, learnRate, learnRateIncrease, ...
   flexible2LMatrix, costThreeLayerRecord3L);

% This function trains, tests and validates a net with its data

if (S3 == 0)              %it's a two layer backprop net
   S_Matrix = [S1 S2];
   TF_Matrix = {TF1 TF2};
```

```
else                    %it's a three layer backprop net
        S_Matrix = [S1 S2 S3];
   TF_Matrix = {TF1 TF2 TF3};
end

BTF = cast(theseSpecialParameters{1,1}, 'char');

   switch BTF

      %define the net:
 %NOTE - when using input files other than alphabet, minmax(alphabet) will
 %have to be changed to reflect the new file.
%todo - do I always want to use learngdm as the BLF?
%BLF -- Backpropagation weight/bias learning function, default = 'learngdm'
%writeMatrix contains the special Parameters and is used to write each in
%its own field in the output file.
%todo do I always want to use PF -- Performance function, default = 'mse'?

      case 'traingdx'
         net = newff(minmax(flexible2LMatrix'), S_Matrix, TF_Matrix, BTF, 'learngdm',
'mse');
         writeMatrix = [0 0 0 0 0 0 0 0 0 0];

      case 'trainlm'
         net = newff(minmax(flexible2LMatrix'), S_Matrix, TF_Matrix, 'trainlm',
'learngdm', 'mse');
         net.trainParam.mu     =  theseSpecialParameters{1,2};  %initialMuValues =
[0.001 0.002];
         net.trainParam.mu_dec =  theseSpecialParameters{1,3};
%muDecreaseFactorValues = [0.1 0.2];
         net.trainParam.mu_inc =  theseSpecialParameters{1,4};
%muIncreaseFactorValues = [10 20];
         net.trainParam.mu_max =  theseSpecialParameters{1,5};  %maximumMuValues
=       [1e10];
         writeMatrix = [net.trainParam.mu net.trainParam.mu_dec ...
            net.trainParam.mu_inc net.trainParam.mu_max 0 0 0 0 0 0];

      case 'trainrp'
         net = newff(minmax(flexible2LMatrix'), S_Matrix, TF_Matrix, 'trainrp',
'learngdm', 'mse');
         net.trainParam.delt_inc = theseSpecialParameters{1,2};   %1.2   Increment to
weight change
         net.trainParam.delt_dec = theseSpecialParameters{1,3};   %0.5    Decrement to
weight change
         net.trainParam.delta0   = theseSpecialParameters{1,4};   %0.07   Initial weight
change
```

```
        net.trainParam.deltamax = theseSpecialParameters{1,5};   %50.0   Maximum
weight change
        writeMatrix = [0 0 0 0 net.trainParam.delt_inc net.trainParam.delt_dec ...
           net.trainParam.delta0 net.trainParam.deltamax 0 0];

      case 'trainscg'
        net = newff(minmax(flexible2LMatrix'), S_Matrix, TF_Matrix, 'trainscg',
'learngdm', 'mse');
        writeMatrix = [0 0 0 0 0 0 0 0 0 0];
     end

% note - the net must be initialized (i.e. "newff...") before changing the
        % net.trainParameters
    net.trainParam.epochs = maxEpochs;
    net.trainParam.goal = goal;
    net.trainParam.lr = learnRate;
    net.trainParam.lr_inc = learnRateIncrease;   % ***************
        %net.trainParam.lr_dec            % Ratio to decrease learning rate
        %net.trainParam.max_fail          % Maximum validation failures
        %net.trainParam.max_perf_inc      % Maximum performance increase
    net.trainParam.mc = momentum;
        %net.trainParam.min_grad          % Minimum performance gradient
%           net.trainParam.show = 1;         % Epochs between showing
%                                    progress - used to run with plot
    net.trainParam.show = NaN;              % NaN is used to run without displaying plot
        %net.trainParam.time              % Maximum time to train in seconds

% train the net

% Sets of data split into training, verification and test sets:

[trainP, trainT, validation, test] = ...
   twoLayerDesignAndJall_intoNN(flexible2LMatrix, costThreeLayerRecord3L);
%          flexible2LMatrix is       x by number of input nodes
%            costThreeLayerRecord3L is  x by 1

[net,tr]=train(net,trainP,trainT,[],[],validation,test);
%[net,tr]=train(net,trainP,trainT,[],[],[],[]);

%log the data
performanceLog = tr;
c=size(tr.epoch);
  e=c(1,2);
i=1;

  performance_log(1,i).learnPerform = tr.perf(e);
```

```
%   tr.epoch(e) - the (e) gives us the value of the last epoch.

%       The complete results record is:
      %resultsRecord = [S1,S2,S3,TF1,TF2,TF3, BTF, maxEpochs, goal, momentum,
learnRate, ...
        %    learnRateIncrease, tr.epoch(e), tr.vperf(e), tr.tperf(e)]

          performanceLog = {S1,S2,S3,TF1,TF2,TF3,BTF, maxEpochs, goal, ...
                            momentum, learnRate, tr.epoch(e), tr.perf(e), writeMatrix};

fid = fopen('tempFileWrite.txt','a');
fprintf(fid,'%3.0f %3.0f %3.0f %s %s %s %s %4.0f %e %e %e %4.0f %e %e %e %e %e
%e %e %e %e %e %e\n',...
   S1,S2,S3,TF1,TF2,TF3,BTF, maxEpochs, goal, momentum, learnRate, tr.epoch(e),
tr.perf(e),...
   writeMatrix);
 fclose(fid);

  DDD=tr.perf(e);
  fwriteid = fopen('temp.bin','a');
  fwrite(fwriteid,DDD,'float32');

    %The binary write was successful.  This checked to ensure it wrote
    %correctly:
    %freadid = fopen('temp.bin','r');
    %D2 = fread(freadid,'float32');
    %disp((D2'))




function [fixCost, totalSensorsCost]=fixedCostFlex(sensorMatrix,infrastructureCost, ...
                          FlexibleSensorCost, InflexibleSensorCost);

%function [fixCost, totalSensorsCost]=fixedCostFlex(sensorMatrix,infrastructureCost, ...
%                          FlexibleSensorCost, InflexibleSensorCost);

sum = 0;

[numberOfSensors, dontCare] = size(sensorMatrix);
```

```
for i=1:numberOfSensors
   sum = sum + sensorMatrix(i,3);
end
NrOfFlexibleSensors = sum;
NrOfInflexibleSensors = numberOfSensors - sum;


totalSensorsCost = NrOfFlexibleSensors * FlexibleSensorCost ...
            + NrOfInflexibleSensors * InflexibleSensorCost;
fixCost = infrastructureCost + totalSensorsCost;




function [flexibleMatrix, nrOnEachLevelFlex]=flexibleMatrixBuilder(sensorMatrix);
% flexibleMatrix is levelMatrix with the inflexible sensors removed.
[nrOnEachLevel, levelMatrix]=sensorArray2(sensorMatrix);
[One, nrOfLayers] = size(nrOnEachLevel);

nrOnEachLevelFlex=[0 0 0 0 0 0 0 0 0 0];

for layerIndex = 1:nrOfLayers
  flexibleRow = 1;
    for pointer = 1:nrOnEachLevel(1,layerIndex)
       if levelMatrix(pointer, 3, layerIndex) == 1
          nrOnEachLevelFlex(1,layerIndex) = nrOnEachLevelFlex(1,layerIndex)+1;
        for column = 1:2
           flexibleMatrix(flexibleRow, column, layerIndex)= ...
                              levelMatrix(pointer, column, layerIndex);
        end
       flexibleRow = flexibleRow + 1;
       end
    end
end




function [fuzzyParetoSet] = fuzzyPareto(ndvector, fuzzyLevel, record3);
%make a quick fuzzy Pareto set near the non-dominated points by adding
%solutions whose variable values are slightly different. (ie perturb the
%design variable values)
fuzzyParetoSet = [];
[nrRecords, nrVariables]=size(record3);
for i=1:nrRecords
```

```
   if ndvector(i) == 1
      fuzzyParetoSet = cat(1,fuzzyParetoSet,record3(i,:));  %If nondominated, add the
design
                                    %vector solution to fuzzyParetoSet.
      temp1(i,:)=record3(i,:);
      temp2(i,:)=record3(i,:);
      temp3(i,:)=record3(i,:);

      for j=1:4
         temp1(i,j)=normrnd(record3(i,j),abs(record3(i,j))/12*fuzzyLevel);
         temp2(i,j)=normrnd(record3(i,j),abs(record3(i,j))/12*fuzzyLevel);
         temp3(i,j)=normrnd(record3(i,j),abs(record3(i,j))/12*fuzzyLevel);

      end
      % Binomial random distribution used to apply "fuzzy Pareto" to the
      % 7th through 12th: temp 1 mostly 0's, temp2 mostly 1's.

      temp1(i,5) = 1;
      temp2(i,5) = 1;
      temp3(i,5) = 1;
       for j=6:8
         temp1(i,j) = binornd(temp1(i,j),fuzzyLevel);
         temp2(i,j) = binornd(temp1(i,j),fuzzyLevel * 0.8);
         temp3(i,j) = binornd(temp1(i,j),fuzzyLevel * 0.6);

            if temp1(i,6)==0 & temp1(i,7)==0
               temp1(i,8) = 1;
            end
            if temp2(i,6)==0 & temp2(i,7)==0
               temp2(i,8) = 1;
            end
            if temp3(i,6)==0 & temp3(i,7)==0
               temp3(i,8) = 1;
            end
      end

     fuzzyParetoSet = cat(1,fuzzyParetoSet,temp1(i,:));  %a bit rough, but should work
     fuzzyParetoSet = cat(1,fuzzyParetoSet,temp2(i,:));
     fuzzyParetoSet = cat(1,fuzzyParetoSet,temp3(i,:));

      end
end

end
```

```
function[fwriteid]=generate3LayerInputFiles(fuzzyParetoSet);

% This function writes the 2 Layer design vector and corresponding objective vector
% data to a binary file that can be read by MultiObjectiveMain3rdLayer.
% The fwriteid variable isn't important outside this function, but Matlab
% requires something to be returned.

fwriteid = fopen('fuzzyParetoSet.bin','w');
fwrite(fwriteid,fuzzyParetoSet,'float32');
fclose(fwriteid);



function[fwriteid]=generateMultObjMainInputFile(writeValue);
%{
This function writes parameters to a file that is read by the
MultiObjectiveMain function. The fwriteid variable isn't important
outside this function, but Matlab requires something to be returned.
multiObjectiveMain is run by the Matlab GA and can only accept one
input variable in the function call, so any additional inputs must
be sent through this file.
%}

fwriteid = fopen('wtCostLen2Layer.bin','w');

fwrite(fwriteid,writeValue,'float32');
fclose(fwriteid);



function[fwriteid]=generateMultObjMainInputFile2(writeValue);
%{
This function writes parameters to a file that is read by the
MultiObjectiveMain function. The fwriteid variable isn't important
outside this function, but Matlab requires something to be returned.
multiObjectiveMain is run by the Matlab GA and can only accept one
```

```
input variable in the function call, so any additional inputs must
be sent through this file.
%}

fwriteid = fopen('wtCostLen3Layer.bin','w');
fwrite(fwriteid,writeValue,'float32');
fclose(fwriteid);




function[JallAverage, ThreeLayerRecord3L]=get3LayerFrom2Layer(nrOfFuzzySetsIn,
nrDesignVariables, fuzzyParetoSet, inputParameters)
% The 2 layer fuzzyParetoSet design candidates are now extended to 3 layer
% designs using the genetic algorithm.

[fwriteid]=generate3LayerInputFiles(fuzzyParetoSet);

ThreeLayerRecord2L=[];
ThreeLayerRecord3L=[];
indexNr=1;

inputParameters = xlsread('InputDataFile.xls','sheet2');

w_LowerLimitGet3From2 = inputParameters(21);
w_StepSizeGet3From2 = inputParameters(22);
w_UpperLimitGet3From2 = inputParameters(23);
n_LowerLimitGet3From2 = inputParameters(24);
n_StepSizeGet3From2 = inputParameters(25);
n_UpperLimitGet3From2 = inputParameters(26);
nrOfRandomRunsGet3From2 = inputParameters(27);
fuzzyLevel = inputParameters(28);

options = gaoptimset('Generations',300);
options = gaoptimset('PopInitRange',[[-10 -10 -10 -10 -10 -10 -10]; ...
                              [20 20 20 20 20 20 20]]);

rand('state', 71); % These two commands are only included to
randn('state', 59); % make the results reproducible.
record=[];  record2=[];  record3=[]; x=[];

%   - runNumber3L is incremented each time the program loops through a
%     fuzzyParetoSet design candidate.
%   - w is increased as the weighting of objectives changes
%   - n is increased as the crossover fraction is changed in the GA.
```

```
%   - indexNr increments each time, so there is a unique index number for
%   each of the sets run
%   objective fitness values are not recorded in this section of the
%   program
%for runNumber3L = 1:2;                %to run only 2 sets for debugging.
   for runNumber3L = 1:nrOfFuzzySetsIn;   %commented out to run only 2 sets
   %for debugging.  Each FuzzySetIn is associated with one 2 layer design,
   %(one ThreeLayerRecord2L)
   [fwriteid] = makeFileFor3rdLayerRun(nrOfFuzzySetsIn, nrDesignVariables,...
                                          runNumber3L);
   % This is the input file for the 3rd Layer Run.
   % The fwriteid variable isn't important outside these functions, but Matlab requires
   % something to be returned.

   x=[];

   for w = w_LowerLimitGet3From2: w_StepSizeGet3From2:
w_UpperLimitGet3From2;          %w is the weighting between objectives - higher w puts
more emphasis on cost

         wtCostLen3Layer(1) = w;   %objective function weighting
         wtCostLen3Layer(2) = 5;   %time
         wtCostLen3Layer(3) = 2;   %flexibleSensorCost
         wtCostLen3Layer(4) = 2.2;   %inflexibleSensorCost
         wtCostLen3Layer(5) = 2.5; %costPerLength
         wtCostLen3Layer(6) = -15;  %len1
         wtCostLen3Layer(7) = 27;   %len2
         [fwriteid]=generateMultObjMainInputFile2(wtCostLen3Layer);

         for n = n_LowerLimitGet3From2 : n_StepSizeGet3From2 :
n_UpperLimitGet3From2  %todo: expose w and n to make them adjustable in the main
program
        %for n=0:0.1:1    %commented out to speed up debugging
            options = gaoptimset(options,'CrossoverFraction', n);

%May 7 Change:
            options = gaoptimset('PopInitRange',...
            [[-10 -10 -10 -10 -10 -10];...
              [ 20  20  20  20  20  20]]);
%May 7 Change - 6 variables:
            [x, weightedFitness] = ga(@multiObjectiveMain3rdLayer, 6, options);

            [x] = makeFlexValues1or0Level3(x);
            [x] = checkAtLeast1FlexLevel3(x);
            [constrained_x]=constrain_x(x);
            [nodesSorted]=sort_a3L(constrained_x);
```

```
            sorted_x = nodesSorted(:,1);     %(Taking just the x values of the 2 column
matrix)
            flex_x   = nodesSorted(:,2);
            sorted_x_1Column = [sorted_x ;flex_x];
            %                   record2(indexNr,:) = sorted_x_1Column;
               %  each column of ThreeLayerRecord3L is a design variable - position,
               %  flexible or not etc.
               %  each row of ThreeLayerRecord3L is a design vector - the design vector
               %  from run #1, #2 etc.
               % indexNr isn't required to be in the record, but it's
               % easier to write and debug with it there.
               % fuzzyParetoSet(runNumber3L,:) is the 2 Layer Pareto Set design vector
               % x is the 3 layer design vector, so
               % ThreeLayerRecord3L includes the whole design.
               ThreeLayerRecord3LAdd=[indexNr, runNumber3L,
fuzzyParetoSet(runNumber3L,:), sorted_x_1Column'];
                % Accumulate the ThreeLayerRecord3LAdds in one matrix:
            ThreeLayerRecord3L = cat(1, ThreeLayerRecord3L,
ThreeLayerRecord3LAdd);

            indexNr = indexNr+1;
          end
       end
end

nrOfRandomRuns=1;

%[JallAverage] = getJallAverage3Layer(nrOfRandomRuns, ThreeLayerRecord3L);
[JallAverage, variableCost, fixCost, totalSensorsCost] = ...
   getJallAverage3Layer(nrOfRandomRuns, ThreeLayerRecord3L);
```

```
function[bestResults] = getBest3Layer(testedResults)
%function[best3LayerNetworks,bestResults] = getBest3Layer(testedResults)
Jall1 = testedResults(:,1);  %Jall1 is set equal to the first column of the objective matrix
Jall2 = testedResults(:,2);  %Jall2 is set equal to the second column of the obj. matrix
[sortedJall1, index1] = sort(Jall1);  %index1 is the rank of each design solution (each
row) relative to objective 1.
```

```
[nrDesigns, dontCare] = size(testedResults);

for i = 1:nrDesigns
   if index1(i)== 1
      bestResults = testedResults(i)
   end
end
```

```
function[Jall, JallTranspose] = getCostAndPerformVals(record2);
%This function takes record2, which has the design vector values, and
%returns Jall, the objective matrix showing how each design did relative to
%the objectives.  Record2 was generated using weighted objectives, or
%ratios between objectives.  The performance relative to the weighted
%objective function is not used further in this algorithm.

totalCost = 0;
variableCost = 0;
fixCost = 0;

freadid = fopen('wtCostLen2Layer.bin','r');
readFileParameters = fread(freadid,'float32');
fclose(freadid);

w                  = readFileParameters(1); % weighting of objectives -TODO - remove?
time               = readFileParameters(2);
flexibleSensorCost   = readFileParameters(3);
inflexibleSensorCost = readFileParameters(4);
costPerLength        = readFileParameters(5);
len1               = readFileParameters(6);
len2               = readFileParameters(7);

pointsToDetect = getPointsToDetect(2);

[nrDesignVectors, nrDesignVariables] = size(record2);

      for designVector = 1: nrDesignVectors;
        for designVariable = 1:nrDesignVariables;
          a(designVariable) = record2(designVector, designVariable);
        end

       if (a(5) < 1 & a(6)<1)
```

```
        a(7) = 1;
      end

    sensorMatrix = [[a(1) 1 1];  %the first layer node is always flexible
                [a(2) 3 a(5)]; [a(3) 3 a(6)]; [a(4) 3 a(7)]];
       [num, levelMatrix] = sensorArray2(sensorMatrix);
       [flexibleMatrix, nrOnEachLevelFlex]=flexibleMatrixBuilder(sensorMatrix);
       [length_] = totalLength4(sensorMatrix,levelMatrix,flexibleMatrix);
       [infrastruct_Cost]= infraCost(sensorMatrix,costPerLength,length_);
       [variableCost] = varCost(sensorMatrix, length_, time);
       [fixCost]=fixedCostFlex(sensorMatrix,infrastruct_Cost, flexibleSensorCost,
inflexibleSensorCost);

       totalCost = variableCost + fixCost;

       [performanceDegrade] = performCalcStocastic(sensorMatrix, len1, len2,
pointsToDetect);

       Jall(1, designVector) = totalCost;
       Jall(2, designVector) = performanceDegrade;
       JallTranspose=Jall';
     end
```

```
function[Jall, JallTranspose, variableCostVector, fixCostVector, ...
   totalSensorsCostVector] = ...
   getCostAndPerformVals3Layer(ThreeLayerRecord3L);

%function[Jall, JallTranspose] = getCostAndPerformVals3Layer(ThreeLayerRecord3L);

%This function takes record2, which has the design vector values, and
%returns Jall, the objective matrix showing how each design did relative to
%the objectives.
totalCost = 0;
variableCost = 0;
fixCost = 0;

freadid = fopen('wtCostLen3Layer.bin','r');
readFileParameters = fread(freadid,'float32');
```

```
fclose(freadid);

w                 = readFileParameters(1); % weighting of objectives
time              = readFileParameters(2);
flexibleSensorCost   = readFileParameters(3);
inflexibleSensorCost = readFileParameters(4);
costPerLength     = readFileParameters(5);
len1              = readFileParameters(6);
len2              = readFileParameters(7);

[nrOf3LayerDesigns ,nrOf3LayerDesignVariables] = size(ThreeLayerRecord3L);

b = ThreeLayerRecord3L;
    for g = 1: nrOf3LayerDesigns;  %cycle through all design vectors
      % note the first two elements of
      % b {b(g,1) and b(g,2)} are the index number and runNumber3L
      % from ThreeLayerRecord3L

%May 7 change:
%         sensorMatrix3Layer= [[b(g,3) 1 b(g,9)]; [b(g,4) 1 b(g,10)]; ...
%               [b(g,5) 3 b(g,11)]; [b(g,6) 3 b(g,12)];...
%               [b(g,7) 3 b(g,13)]; [b(g,8) 3 b(g,14)];...
%               [b(g,15) 5 b(g,25)]; [b(g,16) 5 b(g,26)]; [b(g,17) 5 b(g,27)];...
%               [b(g,18) 5 b(g,28)]; [b(g,19) 5 b(g,29)]; [b(g,20) 5 b(g,30)]; [b(g,21) 5
b(g,31)];...
%               [b(g,22) 5 b(g,32)]; [b(g,23) 5 b(g,33)]; [b(g,24) 5 b(g,34)]];

        sensorMatrix3Layer= [[b(g,3) 1 1]; %the first layer node is always flexible
          [b(g,4) 3 b(g,8)];  [b(g,5) 3 b(g,9)];[b(g,6) 3 b(g,10)];
          [b(g,11) 5 b(g,14)]; [b(g,12) 5 b(g,15)];[b(g,13) 5 b(g,16)]];

[flexibleMatrix, nrOnEachLevelFlex]=flexibleMatrixBuilder(sensorMatrix3Layer);
[num, levelMatrix] = sensorArray2(sensorMatrix3Layer);
[length_] = totalLength4(sensorMatrix3Layer,levelMatrix,flexibleMatrix);
[infrastruct_Cost]= infraCost(sensorMatrix3Layer,costPerLength,length_); %todo -
remove sensorMatrix3Layer
[variableCost] = varCost(sensorMatrix3Layer, length_, time); %todo - remove
sensorMatrix3Layer

%[fixCost]=fixedCostFlex(sensorMatrix,infrastructureCost, ...
%                        flexibleSensorCost, inflexibleSensorCost);

[fixCost, totalSensorsCost]=fixedCostFlex(sensorMatrix3Layer,infrastruct_Cost, ...
                        flexibleSensorCost, inflexibleSensorCost);
        fixCostVector(g)         = fixCost;         %added for debugging
        totalSensorsCostVector(g)  = totalSensorsCost;  %added for debugging
```

```
        variableCostVector(g)      = variableCost;      %added for debugging

totalCost = variableCost + fixCost;

pointsToDetect = getPointsToDetect(3);
[performanceDegrade] = performCalcStocastic(sensorMatrix3Layer, len1, len2,
pointsToDetect);

%Not needed: [weightedFitness] = 2 * performanceDegrade + totalCost;

       Jall(1, g) = totalCost;             %cost for each design vector g
       Jall(2, g) = performanceDegrade;    %performanceDegradation for each design
vector g
      end
       JallTranspose=Jall';
end




function [JallAverage] = getJallAverage(nrOfRandomRuns, record3);
%Run the record3 designs through getCostAndPerformVals to assess
%performance over 3 runs where the random point's position changes.  (Note
%that cost is not affected by the change in the position of the random
%point.) The average over the 3 runs is recorded in JAllAverage:

for i=1:nrOfRandomRuns
   [Jall] = (getCostAndPerformVals(record3))';
   if i==1
      JallSum = Jall;
   else
   JallSum = JallSum+Jall;
   end
end
JallAverage = JallSum / nrOfRandomRuns;




function [JallAverage, variableCostVector, fixCostVector, totalSensorsCostVector] =...
   getJallAverage3Layer(nrOfRandomRuns, ThreeLayerRecord3L);
%Note that the only averaged value is JallAverage, but that the others will
%just be the value from the last of the run of 3
```

```
%Run the record3 designs through getCostAndPerformVals to assess
%performance over 3 runs where the random point's position changes.  (Note
%that cost is not affected by the change in the position of the random
%point.) The average over the 3 runs is recorded in JAllAverage:

for i = 1:nrOfRandomRuns
   [Jall, JallTranspose, variableCostVector, fixCostVector, totalSensorsCostVector]=...
      getCostAndPerformVals3Layer(ThreeLayerRecord3L);
   Jall = Jall';
   variableCostVector = variableCostVector';
   fixCostVector = fixCostVector';
   totalSensorsCostVector = totalSensorsCostVector';

      if i==1
         JallSum = Jall;
      else
      JallSum = JallSum+Jall;
      end
end
JallAverage = JallSum / nrOfRandomRuns;
```

```
function [individualLength]=getLength(lowerLayerSensorPosition,
upperLayerSensorPosition)
  lowerPosit=lowerLayerSensorPosition;
  upperPosit=upperLayerSensorPosition;
  individualLength = ((lowerPosit(:,1) - upperPosit(:,1))^2 + (lowerPosit(:,2)...
     - upperPosit(:,2))^2  )^0.5;
```

```
function [pointsToDetect] = getPointsToDetect(nrOfLayers);
%This function populates the points to detect in the field for the sensor
%network to detect.  It puts 2 random points in the two layer or 4 in the 3
%layer field.

%Note - layers of points must be entered in order - all first layer points,
%then all second layer etc.  The order of the points within each layer
%doesn't matter.
```

```
   R1 = unifrnd(-15,27);
   R2 = unifrnd(-15,27);
   R3 = unifrnd(-15,27);
   R4 = unifrnd(-15,27);

if (nrOfLayers == 3)

  pointsToDetect = [[6 1]; [-1 1]; [3 1]; [-6 1]; ...
                        [-15 1]; [22 1]; [-7 1]; [R1 1];

                [9 3]; [16 3]; ...
                [7 3]; [11 3]; [-7 3]; [0 3]; [17 3]; ...
                  [-3 3]; [12 3]; [-4 3]; [4 3]; [-13 3]; [26 3]; [R2 3]; [R3 3];

                [13 5];[4 5];[8 5];[18 5];[-13 5];[10 5];[R4 5]];

   else
      if (nrOfLayers == 2)
      pointsToDetect = [[6 1]; [-1 1]; [3 1]; [-6 1]; ...
                        [-15 1]; [22 1]; [-7 1]; [R1 1];

                [9 3]; [16 3]; ...
                [7 3]; [11 3]; [-7 3]; [0 3]; [17 3]; ...
                  [-3 3]; [12 3]; [-4 3]; [4 3]; [-13 3]; [26 3]; [R2 3]; [R3 3]];
      else
      printOut = 'error - Number of layers not supported by Points To Detect function'
   end
end




function [infrastruct_Cost]= infraCost(sensorMatrix,costPerLength,length_);
% This determines the cost of the lengths between sensors.
% note - in the one level version, this function determined the lengths.
% That's been moved to the totalLength function.

      infrastruct_Cost = length_*costPerLength;
```

```
function[allSame]= inputCheck(inputVectors)
%This function check to see if all the design vectors are the same.  In the
%unusual case where they are, the neural net has nothing to train on and is
%skipped.  (Trying to determine a pattern between inputs and outputs isn't
%possible if all the inputs are the same).  This returns a 1 if they are
%all the same.
[nrRows, nrColumns] = size(inputVectors);
firstRow = inputVectors(1,:);
i=2;
allSame=1;
while ((i <= nrRows) & (allSame == 1))
   for j=1:nrColumns

      if (firstRow(j) ~= inputVectors(i,j))
         allSame = 0;
      end
   end
   i=i+1;
end
```

```
function [trainP, trainT, validation, test] = inputData();

%not used in the program - illustrative use only
%I copied these out of Matlab and pasted them in, by-passing this function
%just to test the rest of the code.

% P_in is a matrix of vectors.  Each vector is an individual set of data
% and all input data vectors are in the same form.
% T_in is the target vector corresponding to its input vector.

%todo - may want to add a randomizer to take data sets which may be in an
%order and randomize them before splitting into the training, validation
%and test sets
%todo -  add valRatio, testRatio so the proportion of validation, training
%and test sets aren't hard-coded (currently at 1/4, 1/2, 1/4)

P_in = {[0 1 0 1 1 1]' [1 0 0 1 0 0]' [0 1 0 1 1 1]'...
      [1 0 0 1 0 0]' [0 1 0 1 1 1]' [1 0 0 1 0 0]' ...
      [0 1 0 1 1 1]' [1 0 0 1 0 0]'};

T_in = {.44075  .59786   .44075...
      0.55753  .43287   .50819...
```

```
      0.44228  .59512};

%R - number of rows of input (elements in each input vector)
%Q - number of columns (or sets) of input
[R,Q] = size(P_in);

% todo - put a reader in to read in an ASCII file. Not real important
% Remember the data into local variables in functions disappears after the
% function terminates.

% As shown in the Matlab example "Sample Training Session: Backpropogation":
% Divide the data up into training, validation and test subsets.
% We will take one fourth of the data for the validation set,
% one fourth for the test set and
% one half for the training set.
% We pick the sets as equally spaced points throughout the original data.
%iitest contains the indices of the P_in vectors used for test
%iivalidation contains... for validation  etc.

iitest      = 2:4:Q;
iivalidation = 4:4:Q;
iitrain     = [1:4:Q 3:4:Q];

validation.P = P_in(:,iivalidation);  validation.T = T_in(:,iivalidation);
    test.P = P_in(:,iitest);             test.T = T_in(:,iitest);
    trainP = P_in(:,iitrain);            trainT = T_in(:,iitrain);




function [fwriteid]=makeFileFor3rdLayerRun(nrOfFuzzySetsIn, ...
                                nrDesignVariables, runNumber3L);

%This function generates the parameters to pass in to multiObjectiveMain3rd
%since multiObjectiveMain3rd is wrapped in the GA function which won't
%allow it to accept arguments normally.  The parameters are written to the
%binary file parameters3rdLayer.bin.  This file is overwritten each time.

info3rdLayer = [nrOfFuzzySetsIn, nrDesignVariables, runNumber3L];
```

```
fwriteid = fopen('parameters3rdLayer.bin','w');
fwrite(fwriteid, info3rdLayer,'float32');
fclose(fwriteid);
```

```
function [x] = makeFlexValues1or0(x) %make the flexible node values 1's or 0's:
%May 7 change:
%       for i=7:12
        for i=5:8   %layer 1 node is always flexible, but assigned in multObjMain
        if x(i) >=1
              x(i) = 1;
%                sensorMatrix(i-6,3)=1;
           else
              x(i) = 0;
%                    sensorMatrix(i-6,3)=0;
           end
        end
```

```
function [x] = makeFlexValues1or0Level3(x) %make the flexible node values 1's or 0's:

for i=4:6
   if x(i) >= 1
      x(i) = 1;
      %May 7 change:
      %sensorMatrix3Layer(i-4,3)=1;
   else
      x(i) = 0;
         %May 7 change:
         %sensorMatrix3Layer(i-4,3)=0;
   end
end
```

```
function [weightedFitness]=multiObjectiveMain(a1);
%Main program
```

```
%This determines total cost of a sensor set

for k=1:4
   a(k) = a1(k);
end
a(5)=1;
for k=5:7
   a(k+1)=a1(k);
end

[a] = makeFlexValues1or0(a); %make the flexible node values 1's or 0's

[a]=checkAtLeast1Flex(a);  %check to ensure at least one flexible node is on each level

[dontcare size_a]  = size(a);

%Apply constraints so the xCoordinates aren't more than 1 from the area of
%interest (which is -5   to   +9)     Todo: make this more efficient by
%combining into loop below
%todo - is this required, since constrain is used outside the GA and this
%module?
for i=1:size_a
   if a(i) < -15
      a(i) = -15;
   end
   if a(i)> 27
      a(i) = 27;
   end
end

freadid = fopen('wtCostLen2Layer.bin','r');  % Read the Wt Cost Len file.
readFileParameters = fread(freadid,'float32');
fclose(freadid);

w                  = readFileParameters(1); % weighting of objectives
time               = readFileParameters(2);
flexibleSensorCost   = readFileParameters(3);
inflexibleSensorCost = readFileParameters(4);
costPerLength        = readFileParameters(5);
len1               = readFileParameters(6);
len2               = readFileParameters(7);

pointsToDetect = getPointsToDetect(2);

[nodesSorted]=sort_a(a);
```

```
level=[1 3 3 3];
nrOfNodes = (size_a)/2;

for i=1:nrOfNodes
   sensorMatrix(i,:)=[nodesSorted(i,1)  level(i)   nodesSorted(i,2)];
end

        %Note that the GA generates real numbers for all design variables,
        %including the flexible variables.  Flexible variables really
        %should be Boolean; either a 1 or a 0.  The main program converts
        %the real flexible values to Booleans at the beginning of the function and
        %calculates the fitness function using Boolean values.  This ga
        %function will return the real value it sent to the main function
        %though, not the Boolean 1 or 0 the main function used in its
        %computation.

totalCost = 0;
variableCost = 0;
fixCost = 0;

[flexibleMatrix, nrOnEachLevelFlex]=flexibleMatrixBuilder(sensorMatrix);

[num, levelMatrix] = sensorArray2(sensorMatrix);
[length_] = totalLength4(sensorMatrix,levelMatrix,flexibleMatrix);
[infrastruct_Cost]= infraCost(sensorMatrix,costPerLength,length_);
[variableCost] = varCost(sensorMatrix, length_, time);
[fixCost]=fixedCostFlex(sensorMatrix,infrastruct_Cost, flexibleSensorCost,
inflexibleSensorCost);

totalCost = variableCost + fixCost;

[performanceDegrade] = performCalcStocastic(sensorMatrix, len1, len2,
pointsToDetect);

[weightedFitness] = performanceDegrade^(1-w) * totalCost^w;




function [weightedFitness]=multiObjectiveMain3rdLayer(c);
%Main program
%This determines total cost of a sensor set

%Check to ensure constraints are met.  There is no way to handle
```

```
%constraints yet using Matlab's GA.

[dontcare size_c]  = size(c);
for i=1:size_c
   if c(i) < -15
      c(i) = -15;
   end
   if c(i)> 27
   c(i) = 27;
   end
end

freadid = fopen('parameters3rdLayer.bin','r');
info3rdLayer = fread(freadid,'float32');
fclose(freadid);
nrOfFuzzySetsIn   = info3rdLayer(1);
nrDesignVariables = info3rdLayer(2);
runNumber3L       = info3rdLayer(3);

freadid = fopen('wtCostLen3Layer.bin','r');
readFileParameters = fread(freadid,'float32');
fclose(freadid);

w                 = readFileParameters(1); % weighting of objectives
time              = readFileParameters(2);
flexibleSensorCost   = readFileParameters(3);
inflexibleSensorCost = readFileParameters(4);
costPerLength        = readFileParameters(5);
len1              = readFileParameters(6);
len2              = readFileParameters(7);

freadid = fopen('fuzzyParetoSet.bin','r');
D2 = fread(freadid,'float32');
k=1;
for i=1:nrDesignVariables
   for j=1:nrOfFuzzySetsIn
      b(j,i) = D2(k);    %b is the fuzzyParetoSet data
      k=k+1;
   end
end
fclose(freadid);

g=runNumber3L;

   sensorMatrix2Layer= [[b(g,1) 1 1]; %the first layer node is always flexible
            [b(g,2) 3 b(g,6)];[b(g,3) 3 b(g,7)]; [b(g,4) 3 b(g,8)]];
```

```
%check to ensure at least one flexible node is on each level:
%(These should already have been checked previously and should be fine.)

if (b(g,2) < 1 & b(g,3)<1)
   b(g,4) = 1;
end

     [c] = makeFlexValues1or0Level3(c);
     [c] = checkAtLeast1FlexLevel3(c);
     [constrained_c]=constrain_x(c);
     [nodesSorted]=sort_a3L(constrained_c);

      sorted_c = nodesSorted(:,1);     %(Taking just the x values of the 2 column matrix)
flex_c   = nodesSorted(:,2);
sorted_c_1Column = [sorted_c ;flex_c];

      c = sorted_c_1Column';   %Note this has been transposed to a row vector

         %Note that the GA generates real numbers for all design variables,
         %including the flexible variables.  Flexible variables really
         %should be Boolean; either a 1 or a 0.  The main program converts
         %the real flexible values to Booleans at the beginning of the function and
         %calculates the fitness function using Boolean values.  This GA
         %function will return the real value it sent to the main function
         %though, not the Boolean 1 or 0 the main function used in its
         %computation.

         sensorMatrix3Layer= [[b(g,1) 1 1]; %the first layer node is always flexible
            [b(g,2) 3 b(g,6)];  [b(g,3) 3 b(g,7)];[b(g,4) 3 b(g,8)];
         [c(1) 5 c(4)]; [c(2) 5 c(5)];[c(3) 5 c(6)]];

totalCost = 0;
variableCost = 0;
fixCost = 0;

[flexibleMatrix, nrOnEachLevelFlex]=flexibleMatrixBuilder(sensorMatrix3Layer);

[num, levelMatrix] = sensorArray2(sensorMatrix3Layer);
[length_] = totalLength4(sensorMatrix3Layer,levelMatrix,flexibleMatrix);
[infrastruct_Cost]= infraCost(sensorMatrix3Layer,costPerLength,length_);
[variableCost] = varCost(sensorMatrix3Layer, length_, time);
[fixCost]=fixedCostFlex(sensorMatrix3Layer,infrastruct_Cost, ...
                  flexibleSensorCost, inflexibleSensorCost);

totalCost = variableCost + fixCost;
```

```
pointsToDetect = getPointsToDetect(3);
[performanceDegrade] = performCalcStocastic(sensorMatrix3Layer, len1, len2,
pointsToDetect);

[weightedFitness] = performanceDegrade^(1-w) * totalCost^w;s
```

```
function [dmatrix, dvector, ndvector] = paretosort(Jall, minmax);
%© Massachusetts Institute of Technology - Prof. de Weck and Prof. Willcox
%Engineering Systems Division and Dept. of Aeronautics and Astronautics.
%Edited by Michael Nolan.

%This creates a domination matrix showing which designs are dominated over
%all the objectives.  The domination matrix is interpreted like this: if
%there is a 1 in row 2, column 3, that means design 2 dominates design 3.

% Each column of Jall contains objective values for a particular design.
% So columns would be labeled design 1, design 2, etc.
% Each row of Jall contains a particular objective.  So rows would be
% labeled cost, speed, etc.

  [nrObjectives, nrDesigns] = size(Jall);  %mine
  z = nrObjectives;
  n = nrDesigns;

  % Put a -1 one for objectives (eg speed, cost etc.) we want
  % minimized and a +1 for objectives we want maximized.

dmatrix = zeros(n,n);      %fill the dmatrix with zeros
dvector = zeros(n);        %fill the dvector with zeros

for ind1=1:(n-1)
   for ind2=(ind1+1):n
      Ja=Jall(:,ind1).*sign(minmax);
      Jb=Jall(:,ind2).*sign(minmax);
      scorea=0;scoreb=0;
      for indz=1:z
         if Ja(indz)>Jb(indz)
            scorea=scorea+1;
```

```
        elseif Ja(indz)<Jb(indz)
            scoreb=scoreb+1;
        else
            % both solutions are equal
        end
    end
    if scoreb==0&scorea~=0
        dmatrix(ind1,ind2)=1; %a dominates b
    elseif scorea==0&scoreb~=0
        dmatrix(ind2,ind1)=1; %b dominates a
    else
        % no domination in this pair
    end
  end
end

%Go through each column to find dominated designs.  If there is a 1
%anywhere in a design's column, that design is dominated and in a perfect
%world we would disregard it.  It's dvector (dominated vector) entry is 1.
%ndvector is the nondominated vector.
%
for column=1:n
  product=1;
  for row=1:n
    product = product * (dmatrix(row,column) - 1);
  end
  if product == 0
    dvector(column) = 1;
    ndvector(column) = 0;
  else
    dvector(column) = 0;
    ndvector(column) = 1;
  end
  ndvector = ndvector';    %Transpose to make it a column vector.
  dvector = dvector(:,1);  %Remove the unnecessary columns.
end
end
```

```
function[performanceDegrade]=performCalc(sensorMatrix, len1, len2)
%We assume the sensorMatrix is in order with the lowest number first
%len1 and len2 form the range of x values for which we are measuring
```

```
%the performance

[nrOnEachLevel, levelMatrix]=sensorArray2(sensorMatrix);
[xx, nrOfLevels] = size(nrOnEachLevel);

%cumDist is the cumulative distance from each point to the closest sensor
      cumDist=0;

for level = 1 : nrOfLevels
   %k is the number of the sensor on the current level that is closest.
   %It will start at 1.
      k=1;

   for x=len1 : 2 : len2;

      %We'll check to see if sensor 2 is closer than sensor 1.  Once we
      %finally reach an x value on the pipe for which 2 is closer,
      %we switch to measure from point 2 etc.
      if k < nrOnEachLevel(level)
         if abs(levelMatrix(k,1,level) - x) > abs(levelMatrix(k+1,1,level) - x);
%           if abs(sensorMatrix(k,1) - x) > abs(sensorMatrix(k+1,1) - x);
            k=k+1;
         end
      end
      cumDist = cumDist + abs(levelMatrix(k,1,level) - x);
   end

end

performanceDegrade=cumDist;




function[performanceDegrade]=performCalcStocastic(sensorMatrix, len1, len2,...
                                          pointsToDetect);
%We assume the sensorMatrix is in order with the lowest number first
%len1 and len2 form the range of x values for which we are measuring
%the performance
%PerformCalc evenly measures coverage over an area.  This program measures
%performance relative to specific points in the pointsToDetect matrix.
%This uses a range ^ 4  calculation, simulating an active sensor sending
%a signal whose strength diminishes with range squared.  It receives the
%signal from the target with signal strength that diminishes with range
%squared.
```

```
[nrOnEachlayer, layerMatrix]=sensorArray2(sensorMatrix);
[xx, nrOfLayers] = size(nrOnEachlayer);

[nrPointsOnEachlayer, pointsMatrix]=sensorArray2(pointsToDetect);
[yy, nrOfLayersOfPts] = size(nrPointsOnEachlayer);

%cumDist is the cumulative distance from each point to the closest sensor
      cumDist=0;

for layer = 1 : nrOfLayers
   %k is the number of the sensor on the current layer that is closest.
   %It will start at 1.

   nrPointsOnThislayer = nrPointsOnEachlayer(1,layer);

   for i = 1 : nrPointsOnThislayer;
         x = pointsMatrix(i, 1, layer);   %Point-to-detect x coordinate
         k=1;
      %Check to see if sensor 2 is closer than sensor 1.  Once we
      %finally reach an x value on the layer for which 2 is closer,
      %we switch to measure from point 2 etc:
         while (k < nrOnEachlayer(layer)&...
           (abs(layerMatrix(k,1,layer) - x) > abs(layerMatrix(k+1,1,layer) - x)))
             k=k+1;
         end

cumDist = cumDist + abs(layerMatrix(k,1,layer) - x)^4;

   end
end
%added  for R^4 to bring perfDegrade into same range as R^1:
scalingFactor = 230/800000;

performanceDegrade=cumDist*scalingFactor;




function [forecastJs]=runNet(net,candidateSensorSets);
%From help: Simulation (sim)The function sim simulates a network. sim takes
%the network input p, and the network object net, and returns the network
%outputs a. Here is how you can use sim to simulate the network we created
%above for a single input vector: p = [1;2]; a = sim(net,p)
forecastJs = sim(net,candidateSensorSets');   %Note transpose.
```

```
function[selectedCandidates] = selectCandidates(storedRecord3, ...
                              predictedResults, selectionFraction);

[sortedResults, index] = sort(predictedResults);
[nrCandidates dontCare] = size(storedRecord3);
nrSelected = selectionFraction * nrCandidates;   % note this is a real number here

if (nrSelected < 1)
   nrSelected =1;
end

selectedCandidates = [];
for i =1:nrSelected
      selectedCandidates = [selectedCandidates; storedRecord3(index(i),:)];
end
```

```
function [selectedRecord3, storedRecord3] ...
                           =selectFractionOne(record3, fractionOne);
clear store; clear recordStored; clear recordSelected; ...
   clear listOfChosenIndices; clear storedRecord3; clear selectedRecord3;
[nrOfRecords, nrColumns] = size(record3);
store = false;   %assume not stored unless proved otherwise.
%This function selects a fraction of the designs to explore.  The designs
%are randomly chosen.

%This will divide record3 into two parts, one to be selected and one to be
%stored.  If fractionOne is less than 0.5, the records to be selected will
%be called the "chosen" records.  If fractionOne is greater than or equal
%to 0.5, the records to be stored will be called "chosen".  This function will
%initially operate on the "chosen" records, then assign the others by
%process of elimination.
%fractionOne=.6;
if fractionOne < 0.5
   fractionChosen = fractionOne;
else
   fractionChosen = 1-fractionOne;
```

```
   store = true;
end
nrOfRecordsToChoose = round(nrOfRecords * fractionChosen);

i=1;
while i <= nrOfRecordsToChoose
chosenIndex = unidrnd(nrOfRecords);  %discrete random # between 1 and # of Records.
   %check to ensure the index was not already selected:
   unique = 1;
   j = 1;
   while (unique == 1 & j < i)
      if chosenIndex == listOfChosenIndices(j);
        unique = 0;
      end
   j=j+1;
   end
   if unique==1
      %add unique chosen index to the list of chosen indices:
      listOfChosenIndices(i)=chosenIndex;
      i=i+1;
   else %don't add since that index is already on the list
      unique=1;
   end
end

k=1; m = 1;
if store == true
   for i=1:nrOfRecords
      recordStored = false;      %Assume record not stored until proven otherwise
      for j=1:nrOfRecordsToChoose
         if i == listOfChosenIndices(j);
            storedRecord3(k,:)=record3(i,:);
            recordStored = true;
            k=k+1;            %todo consider break   here (or while loop)
         end
      end
          if recordStored ~= true
            selectedRecord3(m,:)=record3(i,:);
            m=m+1;
          end
   end
else     % store == false and the chosen records are to be selected instead
   for i=1:nrOfRecords
      recordSelected = false;      %Assume record stored until proven otherwise
      for j=1:nrOfRecordsToChoose
         if i == listOfChosenIndices(j);
```

```
            selectedRecord3(k,:)=record3(i,:);
            recordSelected = true;
            k=k+1;
          end
        end
          if recordSelected ~= true
            storedRecord3(m,:)=record3(i,:);
            m=m+1;
          end
    end
end
```

```
function [nrOnEachLevel, levelMatrix]=sensorArray2(sensorSet)
%the sensorArray is a n by 2 matrix of sensor x, y coordinates (x in the
%first column, y in the second column.)
%THIS FUNCTION ASSUMES THE SENSOR ARRAY IS GROUPED SO ALL
SENSORS ON
%ONE LEVEL ARE NEXT TO EACH OTHER AND ARE IN ORDER
%

[totalNrOfSensors,z]=size(sensorSet);
currentLevel=1;
nrSensorsOnLevel=1;
for i=1:totalNrOfSensors
    yValue = sensorSet(i,2);
    if i == 1
        yValueOfLast = yValue;
    else
        if (yValueOfLast == yValue)
            nrSensorsOnLevel = nrSensorsOnLevel+1;
        else
            nrOnEachLevel(currentLevel)=nrSensorsOnLevel;
            nrSensorsOnLevel=1;
            currentLevel=currentLevel+1;
            yValueOfLast = yValue;
        end
    end
   nrOnEachLevel(currentLevel)=nrSensorsOnLevel;
end

%Separate the sensorSet matrix (which is sensorArray in the main progam)
%into matrices for each level.  (This could be done with vectors for each
```

```
%level since the y value is repeated for each element of the entry for a
%particular matrix.)

%This works, but packs zeros on the end.
%It does know the number of sensors on any particular level, so you can
%figure out how to get around it.
%LevelMatrix elements are:
% 1st element - the index number of the sensor on its layer
% 2nd element - 1 points to x coordinate, 2 points to the y coordinate from sensorSet
% 3rd element - the index number of the layer
levelMatrixIndex=0;
i=1;
   while i <= totalNrOfSensors
   levelMatrixIndex = levelMatrixIndex + 1;

      for j=1 : nrOnEachLevel(levelMatrixIndex)
      levelMatrix(j,:,levelMatrixIndex) = sensorSet(i,:);
      i = i+1;
      end
   end
end
```

```
function [nodesSorted]=sort_a(a);
%This function sorts the a vector from the GA into an ordered matrix with
%two parts, one for each layer for the 2 layer network. The first matrix
%column is the xCoordinate and the second is 1 if node is flexible and 0 if
%not.  The nodes are sorted in rows from smallest xCoordinate value in the
%first row to the largest xCoord value in the last row.

nrOnEachLevel = [1 3];

[dontcare size_a]  = size(a);

nrOfNodes = (size_a)/2;

for i=1:nrOfNodes
   allNodes(i,:) = [a(i) a(i+nrOfNodes)];
end
```

```
%the "a" vector is split into 2 matrices; one for the lower layer and one for
%the upper layer
for i=1:nrOnEachLevel(1)
   lowerLayer(i,:) = allNodes(i,:);
end

for j = nrOnEachLevel(1)+1 : nrOfNodes
   upperLayer(j-nrOnEachLevel(1) , :) = allNodes(j,:);
end

%Sort both layers:
[b,index] = sort(lowerLayer);

if nrOnEachLevel(1)>1
   for i=1:nrOnEachLevel(1)
      nodesSorted(i,:)      = [b(i,1)  lowerLayer(index(i,1) , 2)];
   end
else
  nodesSorted(1,:) = lowerLayer;  %sort messes this up if only 1 entry - it'll sort the
across instead of down.
end

[c,index] = sort(upperLayer);
for i=1:nrOnEachLevel(2)
   nodesSorted(i+nrOnEachLevel(1),:) = [c(i,1)  upperLayer(index(i,1) , 2)];
end




function [nodesSorted]=sort_a3L(a);
%This function sorts the a vector from the GA into an ordered matrix for the 3rd layer.
The first matrix
%column is the xCoordinate and the second is 1 if node is flexible and 0 if
%not.  The nodes are sorted in rows from smallest xCoordinate value in the
%first row to the largest xCoord value in the last row.

nrOnEachLevel = [3 0];

[dontcare size_a]  = size(a);
nrOfNodes = size_a/2;

%for layer=1:nrLayers  %todo - make this able to handle multiple layers
for i=1:nrOfNodes
   allNodes(i,:) = [a(i) a(i+nrOfNodes)];
```

```
end

%the "a" vector is split into 2 matrices; one for the lower layer and one for
%the upper layer
for i=1:nrOnEachLevel(1)
   lowerLayer(i,:) = allNodes(i,:);
end

%for j = nrOnEachLevel(1)+1 : nrOfNodes
%    upperLayer(j-nrOnEachLevel(1) , :) = allNodes(j,:);
%end

%Sort:
[b,index] = sort(lowerLayer);
for i=1:nrOnEachLevel(1)
   nodesSorted(i,:)      = [b(i,1)  lowerLayer(index(i,1) , 2)];
end
```

```
function [specialParameters, SPcountr]= specialParameters();
%This function generates a matrix with all the special parameters.  Each
%combination is an element of the matrix.  The elements are in the form of
%cell arrays.  The matrix is generated once by main and left as a variable in
%main. In order to change the parameters for trainlm or trainrp, change the code
%below (there's no data file or inputGUI)

SPcountr = 1;
   for BTFindex = 1:4
      switch BTFindex

         case 1
         BTF = 'traingdx';
        specialParameters{:, SPcountr} = {BTF};
           SPcountr = SPcountr + 1;

         case 2
            BTF = 'trainlm';
            initialMuValues =       [0.001];
            muDecreaseFactorValues = [0.1 0.2];
            muIncreaseFactorValues = [10 20];
            maximumMuValues =       [1e10];
          for i = 1:size(initialMuValues,2)
```

```
            initialMu = initialMuValues(:,i);
            for j = 1:size(muDecreaseFactorValues,2)
               muDecreaseFactor = muDecreaseFactorValues(:,j);
               for k = 1:size(muIncreaseFactorValues,2)
                  muIncreaseFactor = muIncreaseFactorValues (:,k);
                  for m = 1:size(maximumMuValues,2)
                     maximumMu = maximumMuValues(:,m);
          specialParameters{:, SPcountr} = {BTF initialMu muDecreaseFactor
muIncreaseFactor maximumMu};
          SPcountr = SPcountr + 1;

                  end
               end
            end
         end

      case 3
                     BTF = 'trainrp';
      delt_incValues =       [ 1.2];        %Increment to weight change
      delt_decValues =       [ 0.5  0.6];   %Decrement to weight change
      initialWtChangeValues = [ 0.07];
      maxWtChangeValues =     [ 50.0];

      for i = 1:size(delt_incValues,2)
          delt_inc = delt_incValues(:,i);
          for j = 1:size(delt_decValues,2)
             delt_dec = delt_decValues(:,j);
             for k = 1:size(initialWtChangeValues,2)
                initialWtChange = initialWtChangeValues (:,k);
                for m = 1:size(maxWtChangeValues,2)
                   maxWtChange = maxWtChangeValues(:,m);
          specialParameters{:, SPcountr} = {BTF delt_inc delt_dec initialWtChange
maxWtChange};
          SPcountr = SPcountr + 1;
                end
             end
          end
      end

      case 4
      BTF = 'trainscg';
      sigmaValues  =  [5.0e-1];  %Determines change in weight for 2nd derivative
approximation
      lambdaValues = [5.0e-2];   %Regulates indefiniteness of the Hessian

      for i = 1:size(sigmaValues,2)
```

```
            sigma = sigmaValues(:,i);
            for j = 1:size(lambdaValues,2)
               lambda = lambdaValues(:,j);
            end
         end
         specialParameters{:, SPcountr} = {BTF sigma lambda};
        % SPcountr = SPcountr + 1;     not incremented since it's the last one.
                end
   end
```

```
function [candidateSensorSets] = takeFlexibleVariables(storedRecord3);
%This function takes the flexible variables - (the last 6) from
%storedRecord3 and outputs them to candidateSensorSets.

candidateSensorSets(:,1:4) = storedRecord3(:,5:8);
```

```
function [totalLength] = totalLength4(sensorMatrix,levelMatrix,flexibleMatrix);
%This function determines the total length of connections across the
%network
totalLength = 0;
%cycle through the elements of the nrOnEachLevel matrix
[nrOnEachLevel, levelMatrix] = sensorArray2(sensorMatrix);
[flexibleMatrix, nrOnEachLevelFlex] = flexibleMatrixBuilder(sensorMatrix);

[dontCare, nrOfLayers] = size(nrOnEachLevel);

%This loop computes the lengths to the next higher level.  There is no need
%to run this loop for the last level since nothing is above it:
for layer = 1 : nrOfLayers - 1
   group = 1;
        %make a vector of midpoints - points midway between each lower layer of
   %flexible sensors.  The values in the midpoint vector are the x
   %coordinates of those points.

   %       To handle layers that have only 1 flexible node:
```

```
      if nrOnEachLevelFlex(layer)==1   % if there's only 1 flexible sensor, it is the
midpoint
          midpoint(1) = flexibleMatrix(1, 1,layer);
        else
          for lowerLayerIndex = 1:nrOnEachLevelFlex(layer) - 1
             midpoint(lowerLayerIndex) = (flexibleMatrix(lowerLayerIndex, 1, layer)...
                          +  flexibleMatrix(lowerLayerIndex+1, 1, layer)) / 2;
          end
       end

   for upperLayerIndex = 1:nrOnEachLevel(layer+1);
      xCoordinate = levelMatrix(upperLayerIndex, 1, layer+1);  %xCoordinate of upper
layer point

   if nrOnEachLevelFlex(layer)==1
      %if there's only 1 flexible sensor, it is the midpoint and the
      %categorizerFlex function can't handle it (trouble figuring number
      %on the lower layer of the Flexible network using the midpoint
      %matrix)
      positionLower = [flexibleMatrix(1,1,layer), flexibleMatrix(1,2,layer)];

   else
      nrOnLowerLayerFlex = nrOnEachLevelFlex(layer);
      [group] = categorizerFlex(xCoordinate, midpoint, nrOnLowerLayerFlex, group);

   positionLower = [flexibleMatrix(group,1,layer), flexibleMatrix(1,2,layer)];

   end

   positionUpper = [xCoordinate, levelMatrix(1,2,layer + 1)];

   totalLength = totalLength + getLength(positionLower, positionUpper);

   end

 %   This loop cycles through the midpoints on the lower layer, grouping
   %   points on the upper layer to connect to the sensor on the lower
   %   layer just prior to the midpoint.  The index then moves to the next
   %   midpoint and makes the next group etc.

      %position lower is the position of the node on the lower level to
      %which the connection is attached.
      %position upper is the position of the node on the lower level to
      %which the connection is attached.

            %note that the y coordinates are all the same, so using point 1 works fine:
```

```
            %levelMatrix(1,2,j)

end
```

```
function [trainP, trainT, validation, test] = ...
twoLayerDesignAndJall_intoNN(flexible2LMatrix, costThreeLayerRecord3L);

%validation and test matrices are broken into:
%validation.P, validation.T, test.P, test.T

% P_in is a matrix of vectors.  Each vector is an individual set of data
% and all input data vectors are in the same form.
% T_in is the target vector corresponding to its input vector.

%example input vector:
%P_in = {[1 2], [3 4], [5 6], [7 8], [2 3], [5 4], [3 2] [2 6]};
%T_in = {[4 1] [2 3] [4 5] [2 6] [2 3] [4 5] [9 0] [4 1]};

[nrOfDesigns, nrOfDesignVariables]=size(flexible2LMatrix);
[nrOfDesigns, nrOfObjectives]=size(costThreeLayerRecord3L);

%[nrOfDesigns, dontCare]=size(JallAverage);
    for i=1:nrOfDesigns
    T_in{i} = costThreeLayerRecord3L(i,:)/1000;  %todo - If this has more than one
element, it probably needs to be transposed also
    flexible2LMatrixTrans=flexible2LMatrix';
    P_in{i} = flexible2LMatrixTrans(:,i);

   end

%R - number of rows of input (elements in each input vector)
%Q - number of columns (or sets) of input
[R,Q] = size(P_in);

% As shown in the Matlab example "Sample Training Session: Backpropogation":
% Divide the data up into training, validation and test subsets.
% We will take one fourth of the data for the validation set,
% one fourth for the test set and
% one half for the training set.
% We pick the sets as equally spaced points throughout the original data.
```

```
%iitest contains the indices of the P_in vectors used for test
%iivalidation contains... for validation  etc.

iitest       = 2:4:Q;
iivalidation = 4:4:Q;
iitrain      = [1:4:Q 3:4:Q];

validation.P = P_in(:,iivalidation);  validation.T = T_in(:,iivalidation);
     test.P = P_in(:,iitest);              test.T = T_in(:,iitest);
     trainP = P_in(:,iitrain);             trainT = T_in(:,iitrain);

% By putting all the input data in one vector and its target vector in
% another, we are making the sets one dimensional.  One might consider
% allowing 2 dimensional or 3 dimensional inputs and outputs or clustered
% inputs and outputs.  That might save time or increase performance.
% However, the NN should be able to recognize patterns and associations
% among inputs after training and be able to rebuild any parts that were
% disassembled by putting everything into the 2 vectors.




function [variableCost] = varCost(sensorMatrix, length_, time);
variableCost = time*length_;
```

Appendix B

M.S. Excel Input File Parameters

Module: CallMultiObj

2 Layer

	LowerLimit	Step Size	UpperLimit
w	0	0.04	0.4
n	0.1	0.1	0.8

flexibleSensorCost	2
inflexibleSensorCost	2.2
costPerLength	2.5
Lower limit for x	-15
Upper limit for x	27
fractionOne	0.4
nrOfRandomRuns	2
fuzzyLevel	0.3

3 layer

	LowerLimit	Step Size	UpperLimit
w_3rdLayer	0	0.05	0.5
n_3rdLayer	0.1	0.2	0.8
nrOfRandomRuns_3rdLayer	2		

fitnessWt used in fitness3LayerRecord3L computation	0.7

Module: Get3LayerFrom2Layer

Get3LayerFrom2Layer

	LowerLimit	Step Size	UpperLimit
w_Get3From2	0	0.05	0.5
n_Get3From2	0.1	0.2	0.8
nrOfRandomRuns_Get3From2	1		

Appendix C

Algorithm Run Results

May28PM Run:

2 Layer

	LowerLimit	Step Size	UpperLimit
w	0	0.05	0.5
n	0.1	0.4	0.8

flexibleSensorCost	2
inflexibleSensorCost	2.2
costPerLength	2.5
Lower limit for x	-15
Upper limit for x	27
fractionOne	0.9
nrOfRandomRuns	1
fuzzyLevel	0.3

3 layer

	LowerLimit	Step Size	UpperLimit
w_3rdLayer	0	0.05	0.5
n_3rdLayer	0.1	0.4	0.8
nrOfRandomRuns_3rdLayer	2		

fitnessWt used in fitness3LayerRecord3L computation	0.4

Get3LayerFrom2Layer

	LowerLimit	Step Size	UpperLimit
w_Get3From2	0	0.1	0.5
n_Get3From2	0.1	0.4	0.8
nrOfRandomRuns_Get3From2	1		

2FlexNodes		3FlexNodes		4FlexNodes	
Cost	PerformanceDegrade	Cost	PerformanceDegrade		
486.96	87.099	396.61	111.1	339.11	76.43
508.97	79.167	383.19	82.431	357.81	83.714
675.47	81.542	367.56	105.81		
643.72	78.093	371.36	135.88		
630.68	76.941	293.4	196.33		
682.33	84.366	341.26	214.83		
624.07	83.126	305.86	214.36		
671.8	85.221	276.82	211.76		
619.74	106.73	367.76	176.54		
547.35	93.749	368.22	190.96		
657.55	79.721				
637.43	82.61				
622.91	77.984				
591.19	87.726				
629.78	85.718				
642.04	76.311				
603.51	83.267				
591.97	84.754				
728	100.45				
703.55	75.645				
748.68	75.26				
725.83	81.56				
692.8	87.172				
688.58	84.199				
482.71	84.877				
460.72	122.52				
619.31	103.25				
618.27	96.251				
598.8	97.022				
622.39	79.762				
542.74	79.284				
540.39	97.175				
595.19	95.22				
628.14	90.434				
626.89	89.578				
663.14	77.15				
579.22	77.802				
636.7	83.45				
496.25	81.652				
479.87	79.03				
442.07	89.69				
439.66	81.714				
542.58	81.693				
528.37	88.904				
539.77	79.052				
578.16	83.566				
492.21	85.555				

425.31	89.059
426.44	81.851
444.2	80.145
516.43	82.137
573.71	83.391
474.48	86.17
491.96	82.706
529.24	80.648
475.95	81.58
478.19	130.46
489.45	81.001
458.3	121.11
468.52	82.808
499.44	121.99
535.92	87.107
553.09	114.07
515.63	81.024
558.62	86.608
510.75	79.135
510.55	92.534
495.37	84.054
489.07	95.86
456.87	85.446
432.14	86.555
445.47	90.635
465.83	120
473.51	86.881
416.09	90.136
479.19	92.035
408.04	95.529
435.6	84.868
406.78	92.528
458.45	96.269
450.7	103.22
518.27	94.456
460.4	118.87
416.61	87.795
432.98	134.37
462.01	85.655
473.04	100.23
453.29	98.795
440.84	93.769
400.34	91.735
447.74	93.968
424.21	98.694
460.26	106.37
413.51	113.63
364.24	149.21
377.05	110.88
467.18	140.7

450.56	96.877
438.65	119.98
427.71	109.7
375.07	127.23
384.32	113.05
412.84	103.77
440.62	115.78
321.41	140.8
391.32	107.36
414.21	110.18
307.18	146.32
379.56	144.92
386.33	148.52
363.69	136.43
389.78	142.6
374.79	136.13
387.68	108.72
413.1	133.13
413.36	114.49
418.1	245.03
437.48	251.77
364.52	258.13
402.19	255.35
344.6	144.1
377.73	118.13
304.36	227.47
422.11	190.81
295.02	205.9
343.1	190.77
324.76	186
334.85	174.3
271.16	249.41
342.14	183.94
409.04	182.28
364.37	205.74
414.05	225.1
345.34	199.39
302.08	197.56
300.77	162.51
345.79	180.61
423.46	178.75
340.91	162
438.09	206.93
324.76	208.26
310.21	200.97
429.95	206.74
375.56	187.85
301.25	203.31
322.13	171.82
296.55	171.09

Appendix D

x coordinate position test

From Jun3PMNewDell Data Set:

First 3 pages of data shown here, remainder on CD.

Distance Between Points on Second Layer

ThreeLayerRecord3L

1st to 2nd	2nd to 3rd	combine the two columns		
12.8988	14.9991	14.9991	Mean:	8.5719335
12.8988	14.9991	14.9991	Std deviation:	3.7166942
12.8988	14.9991	14.9991		
12.8988	14.9991	14.9991		
12.8988	14.9991	14.9991		
12.8988	14.9991	14.9991		
12.8988	14.9991	14.9991		
12.8988	14.9991	14.9991		
12.8988	14.9991	14.9991		
12.8988	14.9991	14.9991		
12.8988	14.9991	14.9991		
12.8988	14.9991	14.9991		
12.6378	13.954	13.954		
12.6378	13.954	13.954		
12.6378	13.954	13.954		
12.6378	13.954	13.954		
12.6378	13.954	13.954		
12.6378	13.954	13.954		
12.6378	13.954	13.954		
12.6378	13.954	13.954		
12.6378	13.954	13.954		
12.6378	13.954	13.954		

TestedResults (SelectedCandidates, selected by neural net)

1st to 2nd	2nd to 3rd	combine the two columns		
5.0812	12.2719	12.2719	Mean:	10.241304
5.0812	12.2719	12.2719	Std deviation:	3.5511484
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		
5.0812	12.2719	12.2719		

12.6378	13.954	13.954
12.6378	13.954	13.954
12.484	15.1344	15.1344
12.484	15.1344	15.1344
12.484	15.1344	15.1344
12.484	15.1344	15.1344
12.484	15.1344	15.1344
12.484	15.1344	15.1344
12.484	15.1344	15.1344
12.484	15.1344	15.1344
12.484	15.1344	15.1344
12.484	15.1344	15.1344
12.484	15.1344	15.1344
12.484	15.1344	15.1344
13.0851	14.397	14.397
13.0851	14.397	14.397
13.0851	14.397	14.397
13.0851	14.397	14.397
13.0851	14.397	14.397
13.0851	14.397	14.397
13.0851	14.397	14.397
13.0851	14.397	14.397
13.0851	14.397	14.397
13.0851	14.397	14.397
13.0851	14.397	14.397
13.0851	14.397	14.397
12.0589	14.3189	14.3189
12.0589	14.3189	14.3189
12.0589	14.3189	14.3189
12.0589	14.3189	14.3189
12.0589	14.3189	14.3189
12.0589	14.3189	14.3189
12.0589	14.3189	14.3189
12.0589	14.3189	14.3189
12.0589	14.3189	14.3189
12.0589	14.3189	14.3189
12.0589	14.3189	14.3189
12.0589	14.3189	14.3189
12.0233	13.6311	13.6311

5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
5.0812	12.2719	12.2719
8.8428	12.873	12.873
8.8428	12.873	12.873
8.8428	12.873	12.873
8.8428	12.873	12.873
8.8428	12.873	12.873
8.8428	12.873	12.873
8.8428	12.873	12.873
8.8428	12.873	12.873
8.8428	12.873	12.873
8.8428	12.873	12.873

12.0233	13.6311	13.6311	8.8428	12.873	12.873
12.0233	13.6311	13.6311	8.8428	12.873	12.873
12.0233	13.6311	13.6311	8.8428	12.873	12.873
12.0233	13.6311	13.6311	8.8428	12.873	12.873
12.0233	13.6311	13.6311	8.8428	12.873	12.873
12.0233	13.6311	13.6311	8.8428	12.873	12.873
12.0233	13.6311	13.6311	8.8428	12.873	12.873
12.0233	13.6311	13.6311	8.8428	12.873	12.873
12.0233	13.6311	13.6311	8.8428	12.873	12.873
12.0233	13.6311	13.6311	8.8428	12.873	12.873
12.0233	13.6311	13.6311	8.8428	12.873	12.873
12.2307	14.527	14.527	8.8428	12.873	12.873
12.2307	14.527	14.527	8.8428	12.873	12.873
12.2307	14.527	14.527	8.8428	12.873	12.873
12.2307	14.527	14.527	8.8428	12.873	12.873
12.2307	14.527	14.527	8.8428	12.873	12.873
12.2307	14.527	14.527	8.8428	12.873	12.873
12.2307	14.527	14.527	8.8428	12.873	12.873
12.2307	14.527	14.527	8.8428	12.873	12.873
12.2307	14.527	14.527	8.8428	12.873	12.873
12.2307	14.527	14.527	8.8428	12.873	12.873
12.2307	14.527	14.527	8.8428	12.873	12.873
12.2307	14.527	14.527	8.8428	12.873	12.873
11.4447	14.2114	14.2114	8.8428	12.873	12.873
11.4447	14.2114	14.2114	8.8428	12.873	12.873
11.4447	14.2114	14.2114	8.8428	12.873	12.873
11.4447	14.2114	14.2114	8.8428	12.873	12.873
11.4447	14.2114	14.2114	8.8428	12.873	12.873
11.4447	14.2114	14.2114	8.8428	12.873	12.873
11.4447	14.2114	14.2114	8.8428	12.873	12.873
11.4447	14.2114	14.2114	8.8428	12.873	12.873
11.4447	14.2114	14.2114	8.8428	12.873	12.873
11.4447	14.2114	14.2114	8.8428	12.873	12.873
11.4447	14.2114	14.2114	8.8428	12.873	12.873
11.4447	14.2114	14.2114	8.8428	12.873	12.873
12.4468	12.9794	12.9794	8.8428	12.873	12.873
12.4468	12.9794	12.9794	8.8428	12.873	12.873
12.4468	12.9794	12.9794	8.8428	12.873	12.873
12.4468	12.9794	12.9794	8.8428	12.873	12.873

12.4468	12.9794	12.9794	8.8428	12.873	12.873
12.4468	12.9794	12.9794	8.8428	12.873	12.873
12.4468	12.9794	12.9794	6.674	12.5503	12.5503
12.4468	12.9794	12.9794	6.674	12.5503	12.5503
12.4468	12.9794	12.9794	6.674	12.5503	12.5503
12.4468	12.9794	12.9794	6.674	12.5503	12.5503
12.4468	12.9794	12.9794	6.674	12.5503	12.5503
12.4468	12.9794	12.9794	6.674	12.5503	12.5503
13.0109	13.4684	13.4684	6.674	12.5503	12.5503
13.0109	13.4684	13.4684	6.674	12.5503	12.5503
13.0109	13.4684	13.4684	6.674	12.5503	12.5503
13.0109	13.4684	13.4684	6.674	12.5503	12.5503
13.0109	13.4684	13.4684	6.674	12.5503	12.5503
13.0109	13.4684	13.4684	6.674	12.5503	12.5503
13.0109	13.4684	13.4684	6.674	12.5503	12.5503
13.0109	13.4684	13.4684	6.674	12.5503	12.5503
13.0109	13.4684	13.4684	6.674	12.5503	12.5503
13.0109	13.4684	13.4684	6.674	12.5503	12.5503
13.0109	13.4684	13.4684	6.674	12.5503	12.5503
13.0109	13.4684	13.4684	6.674	12.5503	12.5503
12.4867	13.0411	13.0411	6.674	12.5503	12.5503
12.4867	13.0411	13.0411	6.674	12.5503	12.5503
12.4867	13.0411	13.0411	6.674	12.5503	12.5503
12.4867	13.0411	13.0411	6.674	12.5503	12.5503
12.4867	13.0411	13.0411	6.674	12.5503	12.5503
12.4867	13.0411	13.0411	6.674	12.5503	12.5503
12.4867	13.0411	13.0411	6.674	12.5503	12.5503
12.4867	13.0411	13.0411	6.674	12.5503	12.5503
12.4867	13.0411	13.0411	6.674	12.5503	12.5503
12.4867	13.0411	13.0411	6.674	12.5503	12.5503
12.4867	13.0411	13.0411	6.674	12.5503	12.5503
12.4867	13.0411	13.0411	6.674	12.5503	12.5503
12.3398	12.406	12.406	6.674	12.5503	12.5503
12.3398	12.406	12.406	6.674	12.5503	12.5503
12.3398	12.406	12.406	6.674	12.5503	12.5503
12.3398	12.406	12.406	6.674	12.5503	12.5503
12.3398	12.406	12.406	6.674	12.5503	12.5503
12.3398	12.406	12.406	6.674	12.5503	12.5503
12.3398	12.406	12.406	6.674	12.5503	12.5503

12.3398	12.406	12.406
12.3398	12.406	12.406
12.3398	12.406	12.406
12.3398	12.406	12.406
12.3398	12.406	12.406
10.5462	14.1207	14.1207
10.5462	14.1207	14.1207
10.5462	14.1207	14.1207
10.5462	14.1207	14.1207
10.5462	14.1207	14.1207
10.5462	14.1207	14.1207
10.5462	14.1207	14.1207
10.5462	14.1207	14.1207
10.5462	14.1207	14.1207
10.5462	14.1207	14.1207
10.5462	14.1207	14.1207
10.5462	14.1207	14.1207
10.5027	13.9933	13.9933
10.5027	13.9933	13.9933
10.5027	13.9933	13.9933
10.5027	13.9933	13.9933
10.5027	13.9933	13.9933
10.5027	13.9933	13.9933
10.5027	13.9933	13.9933
10.5027	13.9933	13.9933
10.5027	13.9933	13.9933
10.5027	13.9933	13.9933
10.5027	13.9933	13.9933
10.5027	13.9933	13.9933
10.8233	14.6242	14.6242
10.8233	14.6242	14.6242
10.8233	14.6242	14.6242
10.8233	14.6242	14.6242
10.8233	14.6242	14.6242
10.8233	14.6242	14.6242
10.8233	14.6242	14.6242
10.8233	14.6242	14.6242
10.8233	14.6242	14.6242
10.8233	14.6242	14.6242

6.674	12.5503	12.5503
6.674	12.5503	12.5503
6.674	12.5503	12.5503
6.674	12.5503	12.5503
6.674	12.5503	12.5503
6.674	12.5503	12.5503
6.674	12.5503	12.5503
6.674	12.5503	12.5503
6.674	12.5503	12.5503
6.674	12.5503	12.5503
6.674	12.5503	12.5503
6.674	12.5503	12.5503
6.674	12.5503	12.5503
6.674	12.5503	12.5503
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311

10.8233	14.6242	14.6242
10.8233	14.6242	14.6242
10.4821	13.7519	13.7519
10.4821	13.7519	13.7519
10.4821	13.7519	13.7519
10.4821	13.7519	13.7519
10.4821	13.7519	13.7519
10.4821	13.7519	13.7519
10.4821	13.7519	13.7519
10.4821	13.7519	13.7519
10.4821	13.7519	13.7519
10.4821	13.7519	13.7519
10.4821	13.7519	13.7519
10.4821	13.7519	13.7519
8.367	14.7984	14.7984
8.367	14.7984	14.7984
8.367	14.7984	14.7984
8.367	14.7984	14.7984
8.367	14.7984	14.7984
8.367	14.7984	14.7984
8.367	14.7984	14.7984
8.367	14.7984	14.7984
8.367	14.7984	14.7984
8.367	14.7984	14.7984
8.367	14.7984	14.7984
8.367	14.7984	14.7984
8.3459	15.1022	15.1022
8.3459	15.1022	15.1022
8.3459	15.1022	15.1022
8.3459	15.1022	15.1022
8.3459	15.1022	15.1022
8.3459	15.1022	15.1022
8.3459	15.1022	15.1022
8.3459	15.1022	15.1022
8.3459	15.1022	15.1022
8.3459	15.1022	15.1022
8.3459	15.1022	15.1022
8.3459	15.1022	15.1022
8.3922	14.0671	14.0671

6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
6.0614	12.311	12.311
4.5056	11.6698	11.6698
4.5056	11.6698	11.6698
4.5056	11.6698	11.6698
4.5056	11.6698	11.6698
4.5056	11.6698	11.6698
4.5056	11.6698	11.6698
4.5056	11.6698	11.6698
4.5056	11.6698	11.6698
4.5056	11.6698	11.6698
4.5056	11.6698	11.6698
4.5056	11.6698	11.6698
4.5056	11.6698	11.6698
4.5056	11.6698	11.6698

8.3922	14.0671	14.0671	4.5056	11.6698	11.6698
8.3922	14.0671	14.0671	4.5056	11.6698	11.6698
8.3922	14.0671	14.0671	4.5056	11.6698	11.6698
8.3922	14.0671	14.0671	4.5056	11.6698	11.6698
8.3922	14.0671	14.0671	4.5056	11.6698	11.6698
8.3922	14.0671	14.0671	4.5056	11.6698	11.6698
8.3922	14.0671	14.0671	4.5056	11.6698	11.6698
8.3922	14.0671	14.0671	4.5056	11.6698	11.6698
8.3922	14.0671	14.0671	4.5056	11.6698	11.6698
8.3922	14.0671	14.0671	4.5056	11.6698	11.6698
8.3922	14.0671	14.0671	4.5056	11.6698	11.6698
8.6119	14.8168	14.8168	4.5056	11.6698	11.6698
8.6119	14.8168	14.8168	4.5056	11.6698	11.6698
8.6119	14.8168	14.8168	4.5056	11.6698	11.6698
8.6119	14.8168	14.8168	4.5056	11.6698	11.6698
8.6119	14.8168	14.8168	4.5056	11.6698	11.6698
8.6119	14.8168	14.8168	4.5056	11.6698	11.6698
8.6119	14.8168	14.8168	4.5056	11.6698	11.6698
8.6119	14.8168	14.8168	4.5056	11.6698	11.6698
8.6119	14.8168	14.8168	4.5056	11.6698	11.6698
8.6119	14.8168	14.8168	4.5056	11.6698	11.6698
8.6119	14.8168	14.8168	4.5056	11.6698	11.6698

Appendix E

Node Position input weights and Results

x-coordinates:

Layer 1 ,	Layer 2,	Layer 2,	Layer 2	Run Number
-0.57199	-4.8313	9.4389	16.429	1
-0.57199	-4.8313	9.4389	16.429	2
-0.57199	-4.8313	9.4389	16.429	3
-0.57199	-4.8313	9.4389	16.429	4
-0.57199	-4.8313	9.4389	16.429	5
-0.57199	-4.8313	9.4389	16.429	6
-0.57199	-4.8313	9.4389	16.429	7
-0.57199	-4.8313	9.4389	16.429	8
-0.57199	-4.8313	9.4389	16.429	9
-0.57199	-4.8313	9.4389	16.429	10
-0.57199	-4.8313	9.4389	16.429	11
-0.57199	-4.8313	9.4389	16.429	12
-0.57199	-4.8313	9.4389	16.429	13
-0.57199	-4.8313	9.4389	16.429	14
-0.57199	-4.8313	9.4389	16.429	15
-0.57199	-4.8313	9.4389	16.429	16
-0.57199	-4.8313	9.4389	16.429	17
-0.57199	-4.8313	9.4389	16.429	18
-0.57199	-4.8313	9.4389	16.429	19
-0.57199	-4.8313	9.4389	16.429	20
-0.57199	-4.8313	9.4389	16.429	21
-0.57199	-4.8313	9.4389	16.429	22
-0.57199	-4.8313	9.4389	16.429	23
-0.57199	-4.8313	9.4389	16.429	24
-0.57199	-4.8313	9.4389	16.429	25
-0.57199	-4.8313	9.4389	16.429	26
-0.57199	-4.8313	9.4389	16.429	27
-0.57199	-4.8313	9.4389	16.429	28
-0.57199	-4.8313	9.4389	16.429	29
-0.57199	-4.8313	9.4389	16.429	30
-0.57199	-4.8313	9.4389	16.429	31
-0.57199	-4.8313	9.4389	16.429	32
-0.57199	-4.8313	9.4389	16.429	33
-0.57199	-4.8313	9.4389	16.429	34
-0.57199	-4.8313	9.4389	16.429	35
-0.57199	-4.8313	9.4389	16.429	36
-0.57199	-4.8313	9.4389	16.429	37
-0.57199	-4.8313	9.4389	16.429	38
-0.57199	-4.8313	9.4389	16.429	39
-0.57199	-4.8313	9.4389	16.429	40
-0.57199	-4.8313	9.4389	16.429	41

-0.57199	-4.8313	9.4389	16.429	42
-0.57199	-4.8313	9.4389	16.429	43
-0.57199	-4.8313	9.4389	16.429	44
-0.57199	-4.8313	9.4389	16.429	45
-0.57199	-4.8313	9.4389	16.429	46
-0.57199	-4.8313	9.4389	16.429	47
-0.57199	-4.8313	9.4389	16.429	48
-0.57199	-4.8313	9.4389	16.429	49
-0.57199	-4.8313	9.4389	16.429	50
-0.57199	-4.8313	9.4389	16.429	51
-0.57199	-4.8313	9.4389	16.429	52
-0.57199	-4.8313	9.4389	16.429	53
-0.57199	-4.8313	9.4389	16.429	54
-0.57199	-4.8313	9.4389	16.429	55
-0.57199	-4.8313	9.4389	16.429	56
-0.57199	-4.8313	9.4389	16.429	57
-0.57199	-4.8313	9.4389	16.429	58
-0.57199	-4.8313	9.4389	16.429	59
-0.57199	-4.8313	9.4389	16.429	60
-0.57199	-4.8313	9.4389	16.429	61
-0.57199	-4.8313	9.4389	16.429	62
-0.57199	-4.8313	9.4389	16.429	63
-0.57199	-4.8313	9.4389	16.429	64
0.60084	-3.026	6.7042	16.662	65
0.60084	-3.026	6.7042	16.662	66
0.60084	-3.026	6.7042	16.662	67
0.60084	-3.026	6.7042	16.662	68
0.60084	-3.026	6.7042	16.662	69
0.60084	-3.026	6.7042	16.662	70
0.60084	-3.026	6.7042	16.662	71
0.60084	-3.026	6.7042	16.662	72
0.60084	-3.026	6.7042	16.662	73
0.60084	-3.026	6.7042	16.662	74
0.60084	-3.026	6.7042	16.662	75
0.60084	-3.026	6.7042	16.662	76
0.60084	-3.026	6.7042	16.662	77
0.60084	-3.026	6.7042	16.662	78
0.60084	-3.026	6.7042	16.662	79
0.60084	-3.026	6.7042	16.662	80
0.60084	-3.026	6.7042	16.662	81
0.60084	-3.026	6.7042	16.662	82
0.60084	-3.026	6.7042	16.662	83
0.60084	-3.026	6.7042	16.662	84
0.60084	-3.026	6.7042	16.662	85
0.60084	-3.026	6.7042	16.662	86
0.60084	-3.026	6.7042	16.662	87
0.60084	-3.026	6.7042	16.662	88
0.60084	-3.026	6.7042	16.662	89
0.60084	-3.026	6.7042	16.662	90
0.60084	-3.026	6.7042	16.662	91

0.60084	-3.026	6.7042	16.662	92
0.60084	-3.026	6.7042	16.662	93
0.60084	-3.026	6.7042	16.662	94
0.60084	-3.026	6.7042	16.662	95
0.60084	-3.026	6.7042	16.662	96
0.60084	-3.026	6.7042	16.662	97
0.60084	-3.026	6.7042	16.662	98
0.60084	-3.026	6.7042	16.662	99
0.60084	-3.026	6.7042	16.662	100

(Remainder of points on CD).

Input parameters, including weight, w:

	LowerLimit	Step Size	UpperLimit
w	0.03	0.05	0.2
n	0.2	0.1	0.8

flexibleSensorCost	2
inflexibleSensorCost	2.2
costPerLength	2.5
Lower limit for x	-15
Upper limit for x	27
fractionOne	0.9
nrOfRandomRuns	1
fuzzyLevel	0

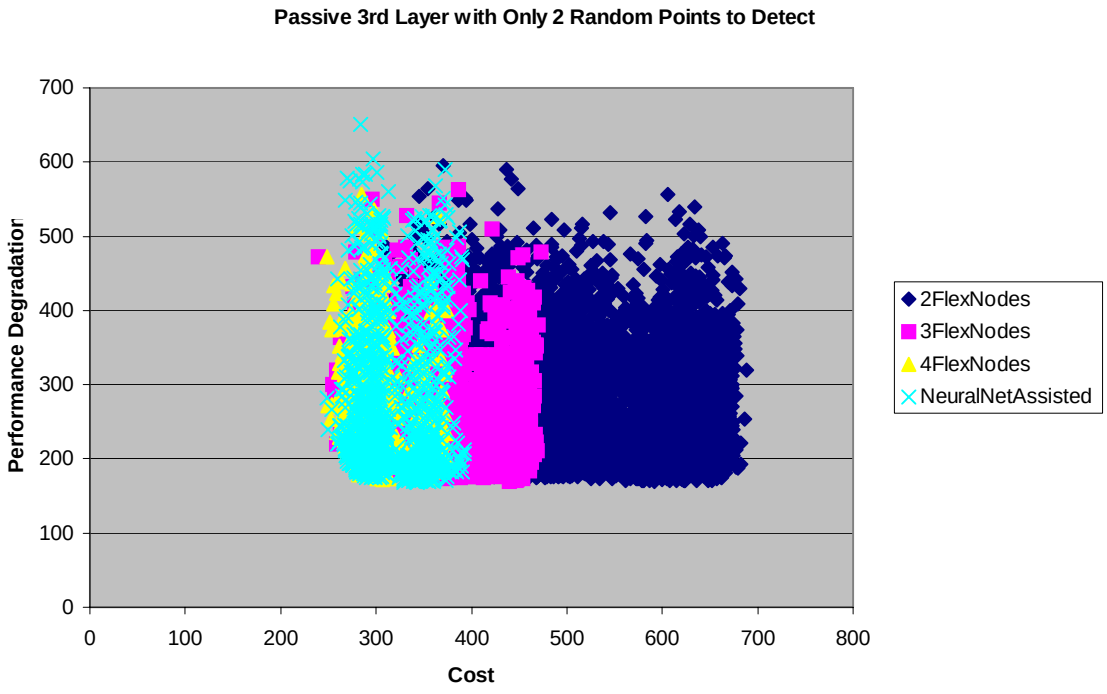
The 2 layer fuzzyParetoSet design candidates are now extended to 3 layer designs using the genetic algorithm:

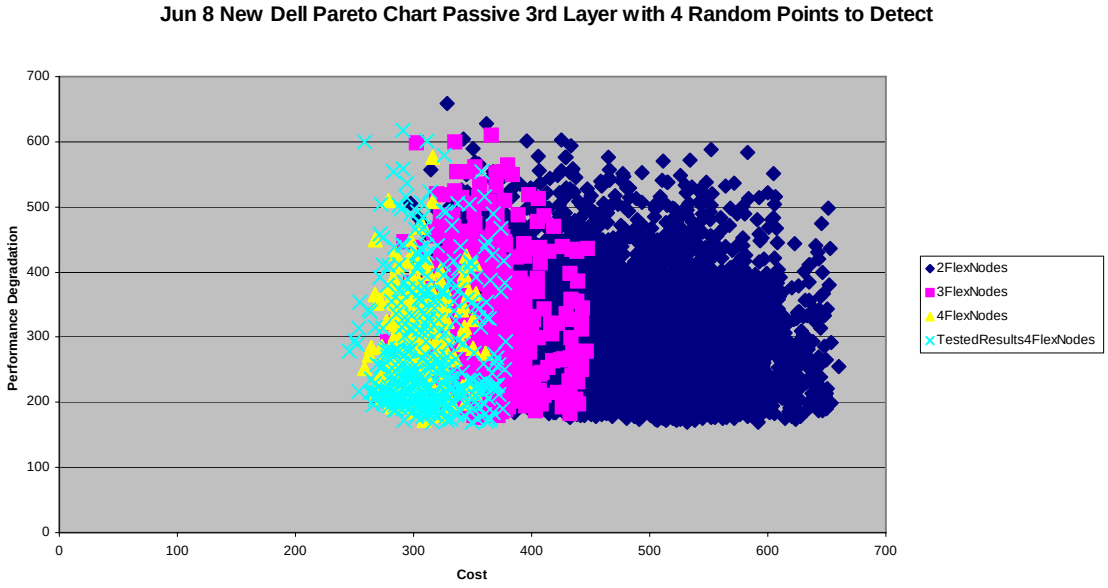
	LowerLimit	Step Size	UpperLimit
w_3rdLayer	0.03	0.05	0.2
n_3rdLayer	0.2	0.2	0.8
nrOfRandomRuns_3rdLayer	1		

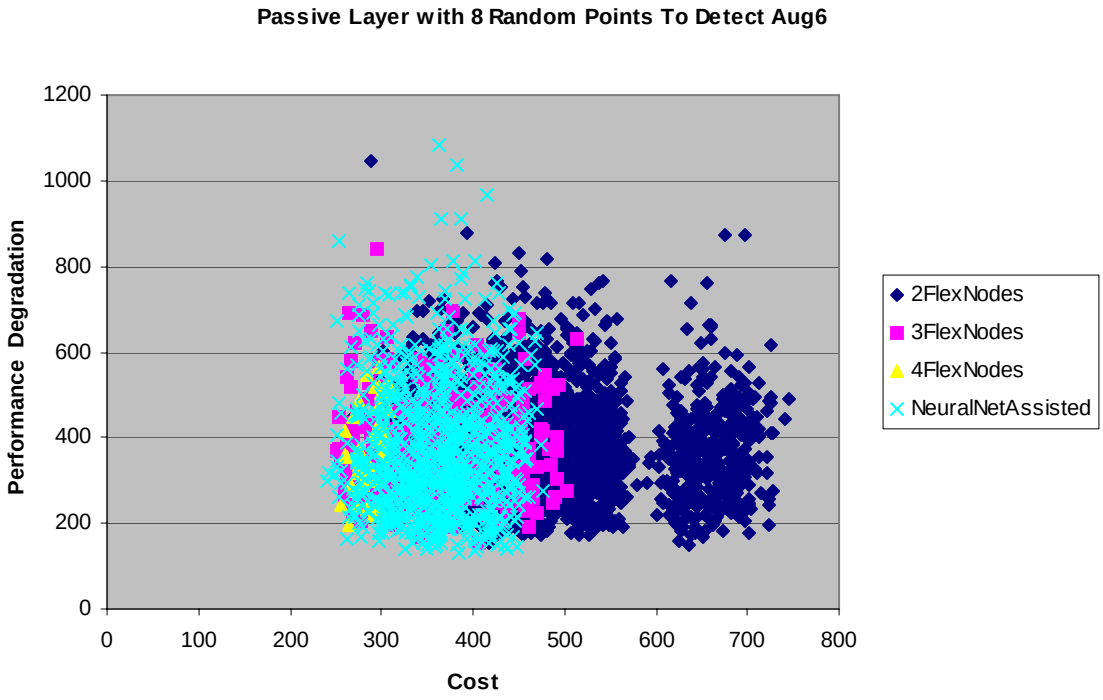
	LowerLimit	Step Size	UpperLimit
w_Get3From2	0.03	0.05	0.2
n_Get3From2	0.1	0.2	0.8
nrOfRandomRuns_Get3From2	1		

Appendix F

Experiment with different numbers of random points to detect






Passive Layer with 8 Random Points To Detect Aug6
Performance Degradation
1200
1000
800
600
400
200
0
0
100
200
300
400
500
600
700
800
Cost
2FlexNodes
3FlexNodes
4FlexNodes
NeuralNetAssisted

References

1 L. Nunes de Castro, F. Von Zuben, *Recent Developments in Biologically Inspired Computing*, Idea Group Publishing, 2005.

2 O. de Weck, K Wilcox, Multidisciplinary System Design Optimization, Lecture 1, 2004.

3 AIAA Technical Committee on Multidisciplinary Design Optimization (MDO) White Paper on Current State of the Art January 15, 1991 http://endo.sandia.gov/AIAA_MDOTC/sponsored/aiaa_paper.html.

4 O. de Weck, K Wilcox, Multidisciplinary System Design Optimization, Lecture 1, 2004.

5 O. de Weck, K Wilcox, Multidisciplinary System Design Optimization, Lecture 1, 2004.

6 E. Suh, I. Kim, O. de Weck, D. Chang, "Design for Flexibility: Performance and Economic Optimization of Product Platform Components", AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, 30 August - 1 September 2004, Albany, New York, AIAA 2004-4310.

7. D. Frey, X. Lim *Evaluating Robust Design Methods Using A Model Of Interactions In Complex Systems*, Engineering Systems Symposium, http://esd.mit.edu/symposium/pdfs/papers/frey.pdf, March 31, 2004.

8 D. Frey, X Li, "Evaluating Robust Design Methods Using A Model Of Interactions In Complex Systems" Engineering Systems Symposium http://esd.mit.edu/symposium/pdfs/papers/frey.pdf, March 31, 2004.

9 D. Rumelhart, J. McClelland and the PDP Research Group, *Parallel Distributed Processing Explorations in the Microstructure of Cognition V1, pg 9*, MIT Press, Cambridge MA, 1986.

10 C. Booth and S. Buluswar "The Return of Artificial Intelligence", The McKinsey Quarterly, 2002, Number 4.

11 D. Rumelhart, J. McClelland and the PDP Research Group, *Parallel Distributed Processing Explorations in the Microstructure of Cognition V1,* MIT Press, Cambridge MA, 1986

12 Matlab Genetic Algorithm help reference.

13 O. de Weck, K Wilcox, Multidisciplinary System Design Optimization, 2004.

14 J. Holland, "Adaptation in natural and artificial systems", University of Michigan Press, 1975.

15 D. Goldberg, "Genetic Algorithms in Search, Optimization and Machine Learning", Addison Wesley, 1989.

16 T. Baeck, *Evolutionary Algorithms in Theory and Practice,* Oxford, N.Y. 1996.

17 A. Zalzala, P. Fleming, "Genetic Algorithms in Engineering Systems" Control Engineering Series 55, The Institution of Electrical Engineers (IEE), 1997.

18 G. Leyland. *Multi-Objective Optimization Applied to Industrial Energy Problems*. PhD thesis, Ecole Polytechnique F´ed´erale de Lausanne, 2002.

19 N. Senin, D. Wallace, N. Borland, "Objected-based Design Modeling and Optimization with Genetic Algorithms", Genetic and Evolutionary Computation Conference, 1999.

20 Matlab help reference.

21 Matlab help reference.

22 Matlab help reference.

23 D. Jourdan and O. de Weck, "Multi-objective genetic algorithm for the automated planning of a wireless sensor network to monitor a critical facility", in Proc. SPIE Defense and Security Symposium, Vol. 5403, pp. 565-575, Orlando, Florida, April 12-16, 2004.

24 A. Mesac, C. Mattson, "Development of a Pareto Based Concept Selection Method", AIAA 2002-1231, Denver, 2002.

25 R. Smaling, O. de Weck, "Fuzzy Pareto Frontiers in Multidisciplinary System Architecture Analysis", 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference 30 August - 1 September 2004, Albany, New York AIAA 2004-4553 American Institute of Aeronautics and Astronautics.

26 O. de Weck, K Wilcox, Multidisciplinary System Design Optimization, Lecture 12, 2004.

27 O. de Weck, K Wilcox, Multidisciplinary System Design Optimization, 2004.

28 R. Smaling, O de Weck, "Fuzzy Pareto Frontiers in Multidisciplinary System Architecture Analysis", 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference 30 August - 1 September 2004, Albany, New York AIAA 2004-4553 American Institute of Aeronautics and Astronautics.

29 O. de Weck, K Wilcox, Multidisciplinary System Design Optimization, Lecture 15, 2004.

30 G. Leyland. *Multi-Objective Optimization Applied to Industrial Energy Problems*, PhD thesis, Ecole Polytechnique F´ed´erale de Lausanne, 2002.

31 N. Senin, D. Wallace, and N. Borland. "Distributed object-based modeling in design simulation marketplace". *ASME Journal of Mechanical Design*, 125:2–13, 2003.

32 N. Senin, D. Wallace, N. Borland, Objected-based Design Modeling and Optimization with Genetic Algorithms, Genetic and Evolutionary Computation Conference, 1999.