NEURO-FUZZY CONTROL OF A ROBOTIC ARM

A Thesis

by

WALLACE EUGENE KELLY, III

Submitted to the College of Graduate Studies
Texas A&M University - Kingsville
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 1994

Major Subject: Electrical Engineering

NEURO-FUZZY CONTROL OF A ROBOTIC ARM

A Thesis

by

WALLACE EUGENE KELLY, III

Approved as to style and content by:

Rajab Challoo, Ph.D., P.E.
(Chairman of Committee)

John H. Painter, Ph.D., P.E.
(External Member)

S. Iqbal Omar, Ph.D., P.E.
(Member)

Robert McLauchlin, Ph.D., P.E.
(Member)

Yuan R. Wang, Ph.D., P.E.
(Head of Department)

Alberto M. Olivares, Ph.D.
Dean of College of Graduate Studies

August 1994

ABSTRACT

Neuro-fuzzy Control of a Robotic Arm

(August 1994)

Wallace Eugene Kelly, III, B.S., Texas A&I University

Chairman of Advisory Committee: Dr. Rajab Challoo

This thesis first outlines the theory, historical background, and application of *neural networks* and *fuzzy logic*. The review of neural networks and fuzzy logic is followed by a discussion of the combination of the two technologies -- *neuro-fuzzy* techniques. The two tools have been successfully combined to maximize their individual strengths and compensate for shortcomings. A survey is given of previous work done in applying these technologies to control systems.

The problem of moving a robotic arm in the presence of an obstacle is discussed. In particular, trajectory planning of a planar, redundant manipulator is studied. The primary weakness of previous methods for determining acceptable trajectories is the massive amount of computer time needed to obtain a solution. Neuro-fuzzy systems offer not only the benefit of the parallel nature of its computations, but also the ability to learn the control of an arm by following a human's example.

Several neuro-fuzzy controllers are trained using sample data obtained from a human's control of a robotic arm. Their performance is quantified and compared. It is shown that the definition of the fuzzy membership functions plays a significant role in the ability of the neuro-fuzzy controller to learn and generalize. Possible directions for future work are suggested.

Table of Contents

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The problem addressed in this thesis is that of moving a robotic arm in the presence of an obstacle. For robots to become effectively used in a wide range of applications, they must gain the ability to work in unpredictable and changing environments. Robots used in space exploration and construction will have to sense their environment and carry out their tasks regardless of the presence of objects in the work area. The ability to move an arm around an obstacle and to a goal is an intuitive skill for human beings. Translating that skill into instructions for a robotic arm, however, is not an easy task.

The ability of a machine to emulate human behavior has always been the goal of artificial intelligence. Neural networks and fuzzy logic systems are two of the most important results of research in the area of artificial intelligence. They have been effectively applied to everything from voice and image recognition to toasters and automobile transmissions. Neural networks are best known for their learning capabilities. Fuzzy logic is a method of using human skills and thinking processes in a machine.

While neural networks and fuzzy logic have added a new dimension to many engineering fields of study, their weaknesses have not been overlooked. In many applications, the training of a neural network requires millions of iterative calculations. Sometimes the network can not adequately learn the desired function. Fuzzy logic systems, on the other hand, acquire their knowledge from an expert who encodes his knowledge in a series of IF/THEN rules. Fuzzy logic systems are easy to understand because they mimic human thinking. The problem arises when systems have many inputs and outputs. Obtaining a rule base for large systems is difficult, if not impossible.

Prompted by the weaknesses inherent in the two technologies and their complementary strengths, researchers have looked at ways of combining neural networks

---

This thesis follows the style and format of the *IEEE Transactions on Control Systems Technology.*

and fuzzy logic. Due to the relative youth of this field of study, a consensus on the best way to utilize their individual strengths and compensate for their individual shortcomings has not yet been established. Consequently, research into neuro-fuzzy systems branches in many directions. The technique used in this work replaces the rule-base of a traditional fuzzy logic system with a back propagation neural network.

Using neuro-fuzzy techniques, a robotic arm can be trained to plan its movements to avoid a collision with obstacles in its vicinity.

CHAPTER II

NEURAL NETWORKS

"A neural network is a dynamical system with the topology of a directed graph that can carry out information processing by means of its state response to continuous or episodic input." -- Robert Hecht-Nielsen [9]

The attitudes of researchers toward neural networks have experienced an evolution since their inception in the early 1940s. According to Leon Cooper, these attitudes "progressed from skepticism through romanticism to what we have at present: general realistic acceptance of neural networks as the preferred -- most efficient, most economic -- solution to certain classes of problems" [7].

Neural networks are composed solely of two elements -- processing elements and interconnections. The processing elements are called *neurons* and the connections are termed *synapses*. A processing element generally has many inputs and a single output as shown in figure 2-1. The neural network performance is governed by the architecture of the processing element's interconnection, the transfer functions for the processing elements, and the learning law [3]. There are two popular models of neural networks -- the *feed-forward* model and the *feedback* model [25].

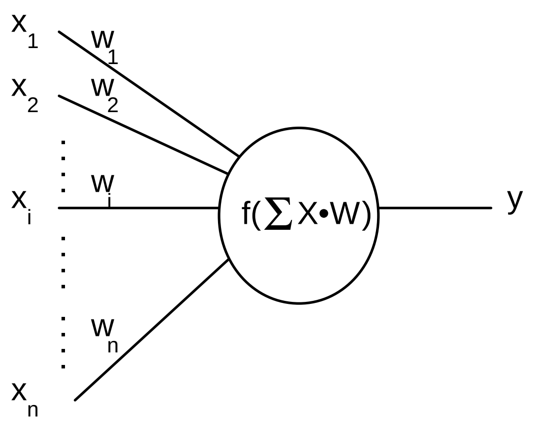


Figure 2-1. An artificial neuron.

The feed-forward model, as illustrated in figure 2-2, is composed of layers where the output from each level is the input for the next level. Input is applied to the input layer. The signals go to the hidden layers and then out to the output layer. Its operation is similar to the operation of a combinational logic circuit. Feed-forward neural networks work well for "natural" problems such as pattern recognition [25].

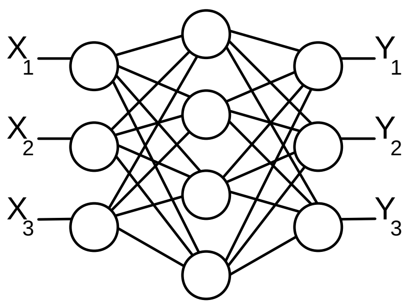


Figure 2-2. A feed-forward neural network.

Figure 2-3 illustrates the feedback model, which has connections between different levels, forward and backward. This is more like an asynchronous logic circuit in which the nodes evolve to a final state. Optimization problems (like the Traveling Salesman problem) are best implemented on feedback neural nets [25].

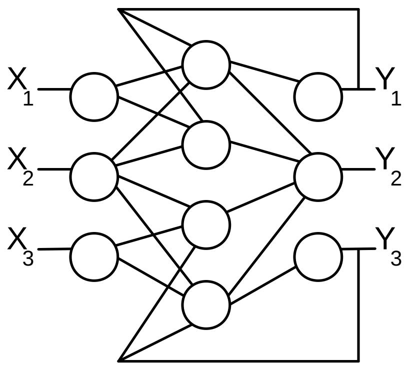


Figure 2-3. A feedback neural network.

Each input to a neuron has an associated weight. If the sum of all the weighted inputs is above a certain threshold, the neuron's output is triggered. An input that tends to lower the weighted sum (probably because of a negative weight) is inhibitory. Excitatory inputs increase the weighted sum. It is useful to consider the inputs, (x1, x2, ... xN) and the weights (w1, w2, ... wN) to be vectors **X** and **W** [3]. The output is then the dot product of the two vectors. This can give us a visual image of what goes on in a neural network. Consider the weight vector to be pointing in a direction in space. An **X** vector will produce a large dot product if that **X** vector points in the same direction as the **W** vector. Imagining the system in this way leads to the development of a neural network of *grandmother cells.*

In a neural network of grandmother cells, each cell has its weight vector set to a pattern which that cell is responsible for identifying [3]. For *n* patterns, *n* grandmother cells are needed. Unknown input is entered into all the grandmother cells. The one that has the highest value on the output has weights closest to the input. This is a very simple neural network. There is no learning involved.

Most neural network applications involve training. Training a neural network to provide the desired outputs can be either *supervised* or *graded* training [25]. Supervised training provides the input and the desired output to the network. Graded training provides the training input and a grade telling the network how close its output is to the desired output. The weights of the neuron inputs are adjusted during the learning process according to a *learning law.* An example of supervised learning is the Delta rule or Least Mean Squared training law, developed by Bernard Widrow and Ted Hoff [4].

To implement the LMS training law, the processing elements must be modified to provide the ability to compare its own output to the desired output. The input to the neuron that supplies the desired output is called the *mentor input.* For LMS training the weights are modified according to equation 1 [4].

$$W_{new} = W_{old} + b\frac{X}{|X|^2}(y_{desired} - y_{actual}) \qquad (1)$$

where W is the weight vector for a processing element

X is the input vector

b is a programmer defined constant between 0 and 1

y are the desired and actual outputs (scalar)

The value of *b* determines how fast the weighting matrices converge to the point of the minimum least square error. The larger *b*, the faster the learning curve should reach its final destination. If the training data is noisy, however, a large learning constant may prolong the training time. The "path" followed to the least squared error point is the negative gradient of the error hyperparabola. Therefore, the learning should always take the most efficient route to minimum error [4].

Once trained the network must be tested. If the network produces unacceptable results, there are several options. First, try more training. If that does not work, reevaluate the learning law and any programmed constants. Make sure that the training data is similar to the test data. Examine the architecture of the network. Are more nodes required? more layers? Is the coding scheme used for the input information adequate?

By far the most popular training method is *supervised back propagation*. A back propagation neural network consists of at least three layers. The input layer accepts the input from the outside world. Therefore, the number of nodes in the input layer is equal to the number of inputs. The output layer produces the result and must also have an appropriate number of nodes. The middle layers are sometimes called *hidden layers*.

Although there is no way to determine what is the best number of hidden nodes, there are some general rules of thumb. If there are too many neurons in the hidden layer, the network will have the tendency to memorize the input patterns rather than generalize the input into features. If on the other hand, the middle layer contains too few neurons, the accuracy of recall will decrease and the number of iterations required for training will increase significantly [5].

Before training begins, all the neuron weights are set to random numbers. Then, the inputs are fed to the neural network, a summation is carried out in each layer, and the results of each layer are passed as the input to the next layer. The summation is shown in equation 2.

$$I = f(\sum_i (w_i * x_i)) \tag{2}$$

where *i* is the input index

w is the weight of that input

x is the input signal

f(x) is the activation function

The *activation function, f(x),* determines the activity, or excitation level, generated in the neuron as a result of the magnitude of the input. "For a back propagation network, this function should be sigmoidal; that is, it must be continuous, S-shaped, monotonically increasing, and asymptotically approaching fixed values as the input approaches plus or minus infinity" [5]. If the neural network is used as a classifier, we must assume +1 or 0 values above and below a certain threshold on the output layer. Equation 3 and figure 2-4 is an example of such a function.

$$f(x) = \frac{1}{1+e^{-x}} \tag{3}$$

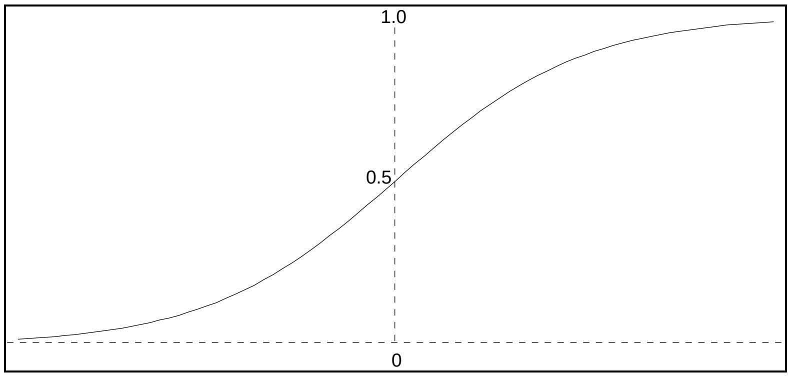


Figure 2-4. The graph of a sample activation function.

The delta rule can be used to adjust the weights of a back propagation neural network. The delta rule is based on minimizing the error. It is easy to apply to the output layer because we know the input and the desired output. However, there is no way for us to know what the desired output is in middle layers. Middle layers require a determination of their error based on the layer receiving their outputs. The term back propagation refers to the way that the error calculation is back propagated from the output to the input. The error is calculated with equations 4 and 5.

$$e_i = f'(I) * \sum_j (w_{ij} * E_j) \tag{4}$$

$$f'(x) \approx f(x)(1 - f(x)) \tag{5}$$

where *i* refers to a particular node in the middle layer

j refers to the nodes in the following layer

e_i is the error of a middle layer node

E_j is the error in the nodes of the following layer

w_{ij} is the weight between node i and j

f(x) is the activation function

The derivative term contributes to stability and helps prevent excessive blame being attached to the middle layer nodes. The training of a BPN therefore requires two passes through the network -- once forward, once backward. Because of this, BPNs are usually trained off-line to determine the weights. Once trained, the network can operate at much faster speeds.

The primary benefit of using a neural network is that the system "learns." Another benefit is that a problem does not have to be well understood before applying a neural network solution. The speed of computation (if many parallel processors act as the neurons) is a potential benefit as hardware implementation of neural networks improves. Finally, there is a better fault tolerance in parallel computer architecture than in the traditional von Neuman architecture [25].

Neural networks have their problems too. Neural nets are not good at precise mathematical computation. Therefore, neurocomputing will not replace algorithmic computing. The two have different, but complementary strengths. Artificial neural networks often require a huge number of iterations for training. They are not guaranteed to find the correct answer[2]. They may get stuck in a local minimum of the error versus possible weights curve. Most researchers agree that the starting point vector is as important as the other aspects of the network [7]. One way to help guard against the local

[2]It has been proven that for any set of input and desired outputs, a neural network exists that can exactly map the input to the output [5]. The problem is finding the correct weights and an efficient architecture.

minimum problem is to include a momentum term in the Delta rule. Equation 6 adds a momentum term to the delta rule of equation 1.

$$W_{new} = W_{old} + bE\frac{X}{|X^2|} + a(W_{new} - W_{old})_{prev} \quad (6)$$

where W is the weight vector for a processing element

X is the input vector

b is a programmer defined constant between 0 and 1

y are the desired and actual outputs (scalar)

a determines how much weight to put on momentum

CHAPTER III

FUZZY LOGIC

"If fuzziness is a genuine type of uncertainty, if fuzziness exists, the physical consequences are universal, and the sociological consequence is startling: Scientists, especially physicists, have overlooked an entire mode of reality." -- Bart Kosko [14]

Fuzzy logic has its roots in the work of renegade mathematicians who saw the value of a multivalent logic system. The credit for fuzzy logic's application to the areas of control and in engineering belongs solely to Lotfi Zadeh. In 1965, he formalized fuzzy set theory [27] and in 1973 brought fuzzy set theory into the context of control systems [28]. According to Lotfi Zadeh, fuzzy logic brings to control systems a "higher machine intelligence quotient" [6].

On a mathematical level, fuzzy logic abandons the strict bivalent logic of TRUE and FALSE, ONE and ZERO, ON and OFF. Fuzzy logic allows for half-truths. Take for an example the scientific classification of a whale. We are taught in grade school, much to our surprise, that a whale is a mammal. A whale is a mammal because, among other reasons, it is warm blooded, gives birth to live baby whales, feeds those young with milk, and also grows hair (or so we are told). This classification system is a perfect example of the traditional bivalent logic that has dominated science for centuries. Despite the fact that it looks like a fish, it swims like a fish, it smells like a fish, and every third grader in the country is skeptical when told that it is not a fish, the whale is 100% mammal, 0% fish. If a fuzzy logician were classifying the whale, he would allow the whale to belong to both the mammal set and the fish set, to certain degrees.

On an engineering level, fuzzy logic provides a platform for easily encoding human knowledge into the control of a system. It has been used in an increasing number of applications, especially in Japan. The Sendai railway in Japan is controlled by fuzzy logic controllers. Applications have been developed in tracking problems, tuning, interpolation, classification, handwriting, voice recognition, image stabilization in video cameras, washing machines, vacuum cleaners, air conditioners, electric fans, hot plates, and Lexus automatic transmissions. An inverted pendulum experiment was demonstrated in 1987 that "produced balancing responses nearly 100 times shorter than those of conventional PID controller" [26].

Fuzzy logic is a method of characterizing knowledge in terms of fuzzy sets and a rule base. A fuzzy system has one or more inputs that are fuzzified, a rule base that is evaluated according to the inputs, and one or more outputs that are defuzzified into "crisp" values. A fuzzy system structure is illustrated in figure 3-1. Bringing fuzzy logic to control problems is a way to use a human expert's knowledge about an analog process in a digital computer. Fuzzy logic is not always the best way to solve a control problem, but it offers several advantages.

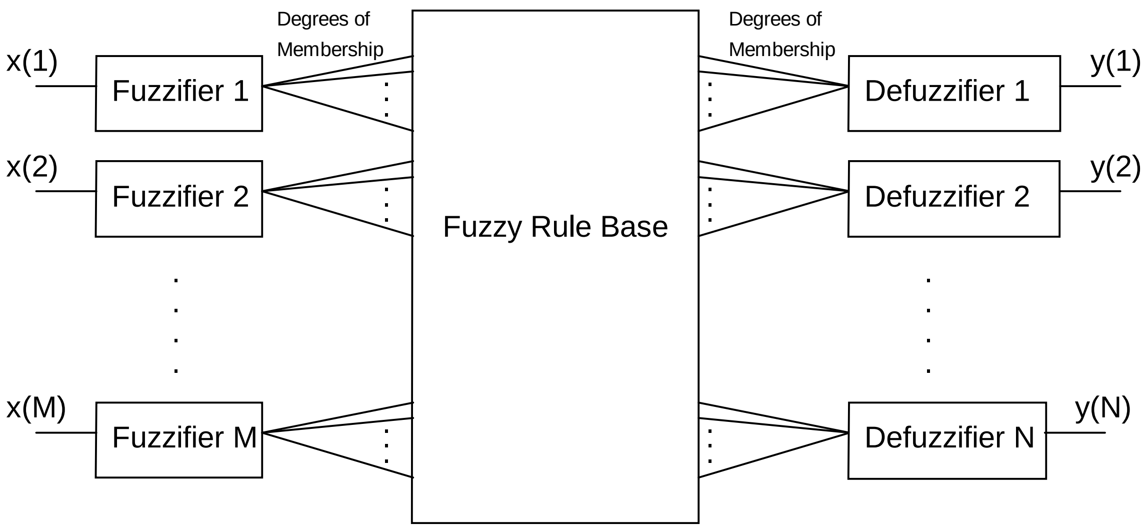


Figure 3-1. A typical fuzzy system.

Fuzzy sets are values to which a variable can belong. Fuzzy control involves describing the control procedure in terms of subjective descriptions like "very low",

"low", "just right", "high", and "very high". Suppose a human were to describe the operation of an elevator. If the elevator did not stop exactly on the desired floor, he might describe the elevator's error in terms of a fuzzy variable like the one above. If the elevator stopped three feet too high, the position of the elevator would definitely be "very high". Six inches too high might just be "high." The question is, "At what height did the elevator go from just being 'high' to being 'very high'?" Fuzzy logic avoids this problem by its multivalent nature. A fuzzy variable can have a certain degree of membership to "high" and a degree of membership to "very high." Often times these fuzzy sets are graphed as shown in figure 3-2.

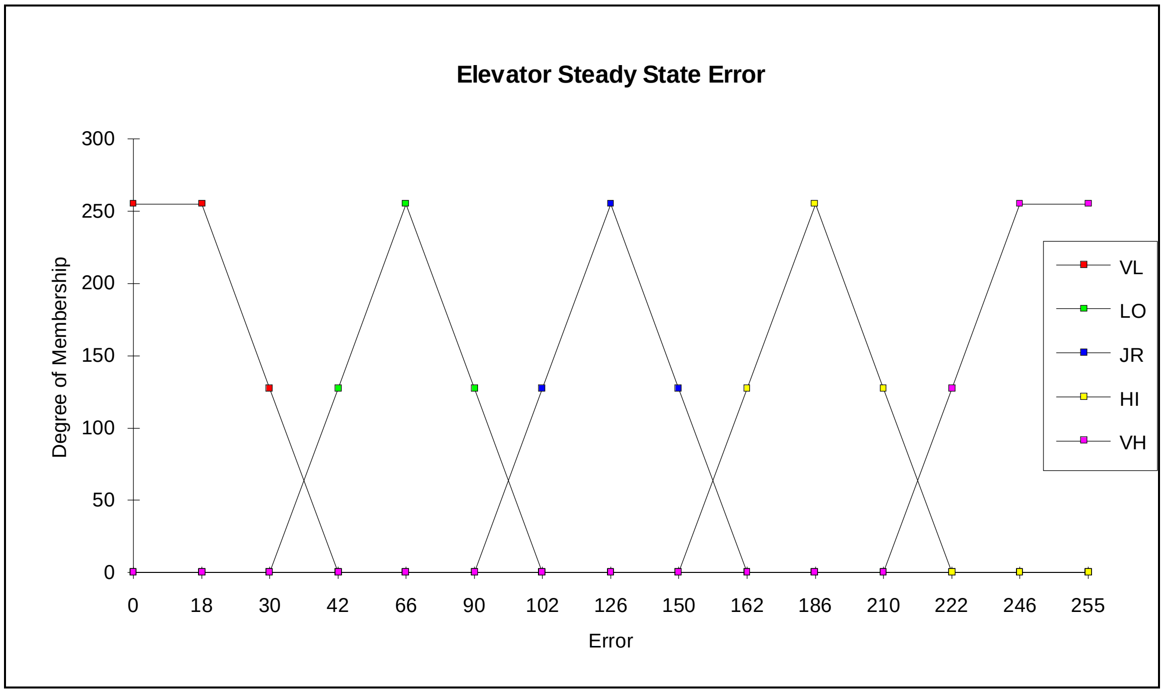


Figure 3-2. A sample fuzzy set.

The control rules are defined with these fuzzy sets. These control rules are usually in the form of IF/THEN statements. "If ERROR is VERY HIGH, then VOLTAGE is NEGATIVE LARGE." or "If ERROR is VERY LOW and LOAD is VERY HIGH, then

MOTOR POWER is POSITIVE LARGE." Combining several if-conditions necessitates the use of fuzzy logic operations. Zadeh defines these basic operations in equations 7, 8, and 9.

$$A \cap B = \min(A, B) \tag{7}$$

$$A \cup B = \max(A, B) \tag{8}$$

$$A' = 1 - A \tag{9}$$

Suppose a rule in the rule base stated, "If ERROR is VERY HIGH **and** LOAD is ZERO, MOTOR POWER is NEGATIVE SMALL." Consider for an example that the elevator was located a distance of 1.5 feet above the desired floor. The distance of 1.5 feet might have a 0.4 degree of membership in ERROR's fuzzy value VERY HIGH. The weight of a child in the elevator might cause a 0.5 degree of membership in LOAD's fuzzy value ZERO. According to equation 7, the rule's antecedent "if ... **and** ..." would evaluate as min (0.4, 0.5) = 0.4. Therefore, in determining the control, the output variable would have a 0.4 degree of membership in MOTOR POWER's fuzzy value of NEGATIVE SMALL. Usually, an output of a fuzzy system is determined by more than one rule and the total output is calculated according to the centroid of the output membership functions.

Sometimes a fuzzy variable has a degree of membership to a fuzzy value of 0.5. "The elevator is not quite low enough to qualify as 'high', but it isn't 'very high' either." For the special case of A = 0.5, equation 9 evalutates as *(not A)=A*. This contradicts traditional, bivalent logic. Notice, however, that traditional zero-or-one logic is a special case of fuzzy logic. If the state space is considered to be a hyperspace cube, whose axes correspond to individual fuzzy variables, traditional logic holds true for the corners of the cube. Fuzzy logic applies to all points in the state space [14].

When is fuzzy logic appropriate? Fuzzy logic is helpful in situations where the control variables are continuous. Fuzzy rules often take the place of a math model. Therefore, fuzzy logic is useful if a mathematical model of a process does not exist, is too difficult to encode, is too complex to be evaluated in real-time, or requires too much memory. Other situations that may make fuzzy control advantageous are when there are high ambient noise levels, it is important to use inexpensive sensors, or it is important to use low precision microcontrollers [6]. They are easier to prototype and implement and simpler to describe and verify. They can be maintained and extended with greater accuracy in less time.

What are the weaknesses of fuzzy logic control? One problem is that the control of some systems can not be easily specified in terms of an IF/THEN rule base. An example of such a system would be a robotic arm operating in the presence of an obstacle. Also, sometimes the 'experts' providing the rule base disagree among themselves. This was demonstrated at the Kawasaki Steel Corporation in Japan [29]. A fuzzy logic control system was installed to help operators make decisions regarding control of a blast furnace. The researchers summarized, "There is a slight difference in knowledge between a multiple number of experts... In the case of expert systems which process ill-structured problems, an 100% success could not be possible." Their data suggests that in practice, the operators ignored the expert system's suggestion over 15% of the time.

CHAPTER IV

NEURO-FUZZY SYSTEMS

"Integration combines separate entities into a whole more useful than any of its individual parts." -- David V. Hillman [10]

Recently, the combination of neural networks and fuzzy logic has received attention. The idea is to lose the disadvantages of the two and gain the advantages of both. Neural networks bring into this union the ability to learn. Fuzzy logic brings into this union a model of the system based on membership functions and a rule base.

This field of study is still in its infancy. Universally accepted techniques and a general consensus on the direction of research have not yet been established. Most of the work done in this area is still associated with individual researchers and has not been adopted as standard strategy. Therefore, much of the discussion of neuro-fuzzy techniques has been reserved for Chapter V in which previous work of other researchers is presented. What follows is a brief overview of the major approaches in neuro-fuzzy research.

Determining the fuzzy membership functions from sample data using a neural network is the most obvious method of using the two together. The definition of the membership function has a huge impact on the system response. Often, the programmer must use trial and error to find acceptable values. Assuming a certain shape and finding the beginning and endpoints for the fuzzy values in a fuzzy set is a neural network optimization problem [18]. Figure 4-1 is a diagram of such a system.

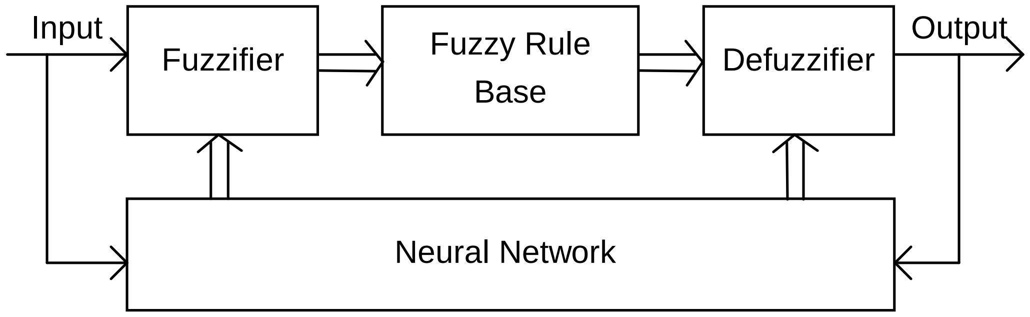


Figure 4-1. A fuzzy system whose membership functions are adjusted by a neural network.

Figure 4-2 shows a more complex integration -- the use of neural networks to determine both the fuzzy membership functions and the rule base. National Semiconductor scientists have developed a system that converts input and output data into nonlinear membership functions and a rule base [12]. The nonlinearity of the membership functions is unique to membership functions derived by neural networks. They help minimize the number of rules.

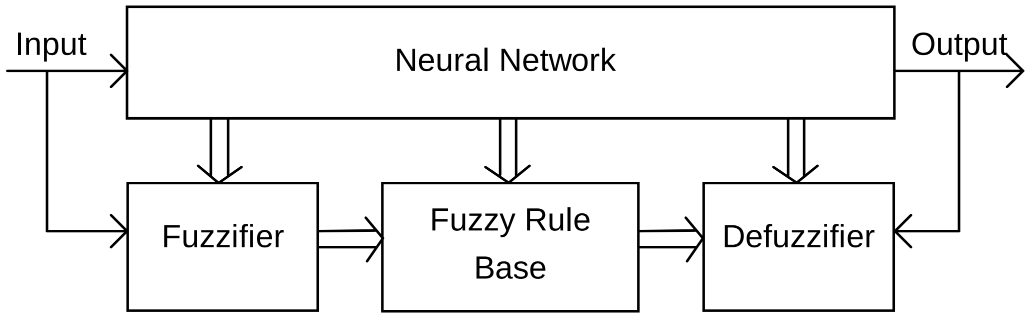


Figure 4-2. A fuzzy system defined by a neural network.

Another approach is to incorporate fuzzy logic into the neurons of the neural networks. This approached developed because of the original neuron model proposed by McCulloch and Pitts [30]. The McCulloch-Pitts cell produced an all-or-none output. It was quickly realized that neurons with output in the range of [0,1] produced much better results. The concept of a fuzzy neuron, however, has advanced beyond simply expanding the range of outputs on a crisp neuron. Some researchers have incorporated membership functions and rule bases into the individual neurons, as shown in figure 4-3.

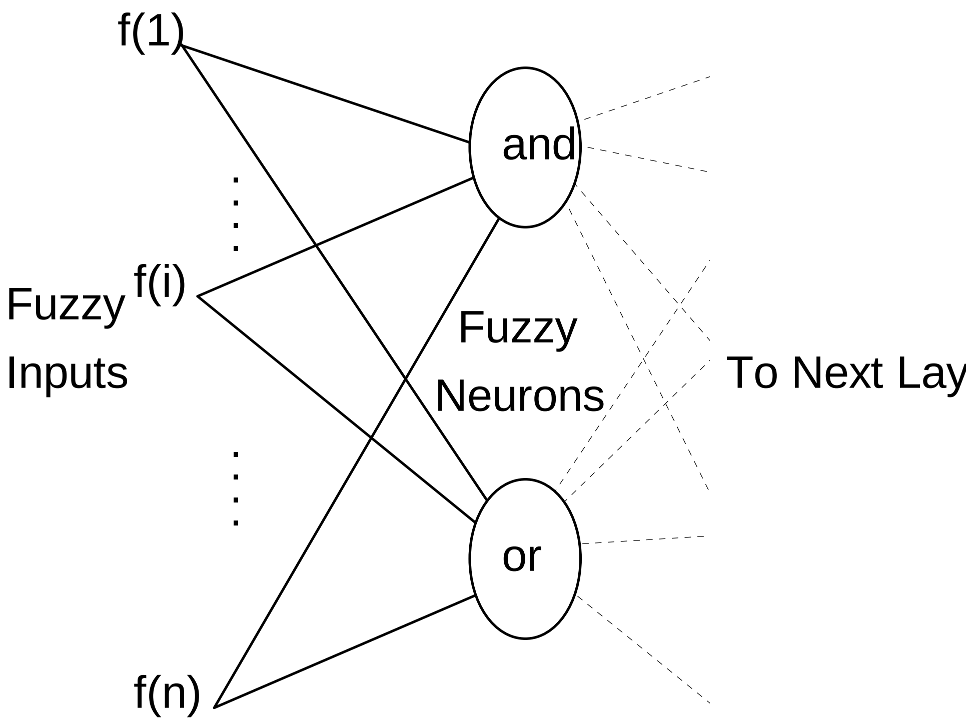


Figure 4-3. A neural network of fuzzy neurons.

Finally, the idea of fuzzification of control variables into degrees of membership in fuzzy sets has been integrated into neural networks. See figure 4-4. If the inputs and outputs of a neural network are fuzzified and defuzzified, significant improvements in the training time, in the ability to generalize, and in the ability to find minimizing weights can be realized. Also, the membership function definition gives the designer more control over the neural network inputs and outputs. It is this technique that is implemented in this thesis for the control of a robotic arm in the presence of an obstacle.

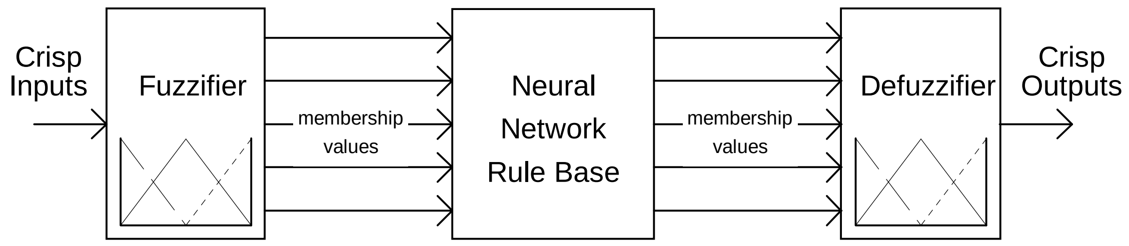


Figure 4-4. A fuzzy system with neural network rule base.

CHAPTER V

BACKGROUND

The use of neural networks, fuzzy logic and neuro-fuzzy technology in control has progressed in that order. Neural networks and fuzzy logic, used independently, have both demonstrated their value in control systems. More recently, neuro-fuzzy control has brought even more improvement to the quality of intelligent control. The following is a survey of the research done in the areas of intelligent control, followed by a survey of trajectory planning techniques.

NEURAL NETWORKS

Guez, Eilbert, and Kam propose an architecture for neural network control that can serve as an adaptive control system [8]. A comparison is made to traditional ***model reference adaptive control*** (MRAC). In their example of a robotic manipulator, the neural network approach shows significant improvement in performance over the MRAC. The neuro-controller is more general, making it possible to be trained for many problems. It shows the most improvement over MRAC as the system increased in order. It is also more stable and less sensitive to plant dynamics.

Psaltis, Sideris, and Yamamura address the problem of training a neurocontroller over a large state space [23]. They suggest training the neural net in two modes -- generalized training and specialized training. The generalized training session extracts the major features. The specialized training works to define detailed boundaries between samples. This decreases the iterations necessary to train the neuro-controller and provides better response in important operating regions.

Nguyen and Widrow use two neural networks to control a truck and trailer as it backs up to a dock from any initial position [20]. The first neural network, called an ***emulator***, learns to identify the system's dynamic characteristics. The second provides the actual control. After 20,000 simulated backups, the controller is able to move the truck to within 3% of the desired position. The network also develops a control strategy, as demonstrated by the fact that initial moves sometimes increase the current error in order to prepare the truck and trailer for ultimate success.

Newton and Xu outline the work being done at Carnegie Mellon University in the area of neural network control of a robotic arm [50]. A seven degree-of-freedom space manipulator is controlled with a neural network resulting in 85% less trajectory error than recorded under PID control. The neural network is trained on-line and made use of a *moving average feedback.* The feedback provides the network with the ability to plan current control based on both the current input and the recent response of the network to past inputs.

Arai, Rong, and Fukuda use neural networks to control a three link robotic arm that is manipulating a flexible plate [51]. The problem is complicated by the fact that the control must take into account not only the trajectory planning, but also the vibration of the flexible material being held. Four neural networks are used to learn the error caused by the vibrations of the flexible plate. The networks have 4, 10, and 1 nodes in the input, hidden and output layers, respectively. After training, the model of the system's error, contained in the neural networks, is used to improve control.

Liu and Asada describe a neural network control of a deburring robot [52]. Initially, a neural network is trained off-line to data obtained by recording a human's control of the robot. This training data teaches general control strategy and task planning. The final control system also makes neural network adjustments on-line based on data acquired during operation. This training fine tunes precise controlled motion. Because of this second learning capability, the neural network controller was able to exceed the performance of the human controller.

T. C. Hsia and Z. Mao propose a scheme for obstacle avoidance of redundant manipulators by neural network [53]. Their scheme, called ***q-system***, translates a desired Cartesian coordinate position, $\mathbf{X}_d$, into four joint angles, $\mathbf{q}$. The training samples are taken from known solutions given by the forward kinematics solution. In this manner, a neural network was trained for the inverse kinematics solution of a four-link arm. The system is improved by performing a coordinate transformation of $\mathbf{q}$, which localizes the search for an inverse solution.

Cooperstock and Milios incorporate neural network control into a vision guided robotic arm [54]. Neural networks take the place of complex numerical solution techniques for both the solution of the inverse kinematics of the arm and for the solution of the perspective projection of the stereo vision system. Cooperstock and Milios's controller is comprised of five neural networks. The networks specialize in the operations of approaching, centering, paralleling, reaching, and adjusting. The researchers describe their control system as being competitive with traditional systems.

Yegerlehner and Meckl utilize the learning capabilities of neural networks to adapt the control of a robot experiencing large changes in payload weight [55]. One neural network is trained for the inverse kinematics, while another is trained to estimate the mass of the payload. The authors reveal that the artificial neural network did not model the inverse kinematics as well as a comparison model of least squares fitting parameters. They point to the fact that the network inputs were only the joint angles and their two time derivatives. They suggest that a "richer set of inputs" would be necessary for improving performance.

FUZZY LOGIC

Li and Lau look at the application of fuzzy algorithms to servo system control [16]. They see the challenge of servo control lying in the accuracy and speed requirements. In comparing fuzzy control, PI control, and MRAC, they conclude that the fuzzy control is better than PI control and as good as MRAC. The authors note that their fuzzy controller could only offer an optimum solution to a narrow range of inputs.

Neffenger concludes that the value of fuzzy motor control is in the fact that a math model of the system is not needed [19]. A position control experiment is performed using a variable speed AC motor. Neffenger varied the mass of the object being controlled without the fuzzy system being affected. A change in system parameters is a drawback for model based control. The experiment demonstrates that a fuzzy controller can yield sufficient responses as parameters in the system change. Also, Neffenger points to the short design time required for fuzzy systems.

Kosko introduces the concept of FAM -- *fuzzy associative memory* [14]. It is based on his view of fuzzy state space as a hyperbolic unit cube. FAMs map input "balls" to output "balls". The balls are clusters of data in the state space associated with certain conditions. Adaptive fuzzy associative memories, AFAM, change over time as new data is sampled and processed.

Kosko uses his FAM concept to build on the work of Nguyen and Widrow by designing a fuzzy truck backer-upper [14]. His fuzzy controller consistently chose a smooth path to its final destination. Kosko points to the fact that Nguyen and Widrow's neural controller did not always converge. He also points out that the fuzzy controller did not require 20,000 iterations for training. Training was achieved by encoding "common sense" FAM rules.

Kong and Kosko extend Kosko's own fuzzy controller for a truck and trailer by implementing an adaptive fuzzy controller [13]. In adaptive FAM, the rules are inferred from sample data using ***differential competitive learning*** (DCL) clustering. The rule inference for the AFAM was much faster than the training required for neural net weight adjustments. The researchers suggest that their fuzzy controller is robust, as demonstrated by the fact that up to 50% of the control rules could be removed without a significant deterioration in performance.

Pacini and Kosko compare the use of AFAM and Kalman-filter for real-time target tracking [21]. The comparison shows that AFAM could track as well as a Kalman filter, but not better than. Again, DCL is used to cluster sample data into rules. The rules that were generated from the sample data produced nearly exactly the same results as the human rule base.

Berenji, Chen, Lee, Murugesan, and Jang perform experiments with fuzzy control and a cart-pole balancing problem [2]. In their comparison of the fuzzy logic controller and a state-feedback controller, the fuzzy logic controller scored high marks in the areas of ease of implementation, robustness and percentage overshoot. The state feedback controller had better settling times and ease of modification. They felt that the largest limitation of the fuzzy method is in the calibration of the membership functions. They suggest research in the area of "automatic learning of approximate control rules."

Shao addresses one of the problems associated with fuzzy systems -- sometimes the rule base is difficult to define. He presents a method for developing fuzzy systems that adjust their rule base to conform to predetermined system requirements. The strength of his approach lies in its application to nonlinear systems and systems with a large time lag. Shao applies his technique to the control of both a boiler and a DC motor.

Qioa, Wang, Heng, and Shan design a Rule Self-regulating Fuzzy Controller (RSFC) [42]. They are interested in adjusting a fuzzy rule base on-line in real-time. In order to give the system the ability to quickly modify the fuzzy controller's rule base, the authors use a special type of rule base which is more easily modified. The example given is that of a fuzzy logic controller adjusting the proportional and integral gain constants in a simple feedback control system.

Bagchi and Hatwal use fuzzy logic to plan the path of an object moving in the presence of obstacles [48]. The researchers justify the use of fuzzy logic for two reasons -- ease of design and the ability of a fuzzy system to work with inaccurate sensors. First, appropriate fuzzy variables and values are chosen. The researchers develop a collision avoidance strategy and encoded that strategy into a simple rule base. A simulation is performed with five objects moving in a plane.

Jou and Wang present their work in the area of adaptive fuzzy logic systems [56]. Error back propagation is applied to a fuzzy logic system to adjust the rule base according to a pre-specified training set. The specific application addressed is that of backing up a truck to a dock. The researcher's system performed as well as the neural network controller of Nguyen and Widrow [20], but required far fewer training sessions. They note that in order to develop a rule base capable of generalization, training samples that are "homogeneously distributed throughout the entire state space" are necessary.

NEURO-FUZZY TECHNOLOGY

Nauck, Klawonn, and Kruse research the fusing of neural networks and fuzzy systems in an attempt to overcome the disadvantages expressed by other researchers using only one of the technologies [18]. Their inverted pendulum application is similar to Berenji's, et al. [2]. The researchers' "neural network oriented fuzzy control" system adjusts its fuzzy set definitions. The application of this system to an inverted pendulum was able to balance the pendulum and reduce the rules required from 512 to fewer than 40. Not only did learning take place, as in a neural network, but the resulting system was defined in the linguistic variables.

Similarly, Jang designs a self-learning fuzzy controller based on temporal back propagation [11]. The current state of the system is compared to the desired state and the error is back propagated through the system to adjust individual fuzzy parameters. Tests on an inverted pendulum show that significant adjustments were made on membership function definitions. The trained system exhibited robustness and fault tolerance.

Archer and Wang propose a method for using neural networks to define membership functions [34]. Their algorithm incorporates what they term as a *Fuzzy Membership Model.* Neural networks are used to learn how to classify patterns that fall near regions of uncertainty in pattern space. In their example, one neural network is used to learn the sharp boundary between two classes, while two other neural networks are trained to determine a pattern's fuzzy membership in a particular class.

Berenji and Khedkar also use neural networks to define fuzzy membership functions [41]. They emphasize the importance of selecting the correct granularity for describing the values of each linguistic variable. Berenji and Khedkar implement a *generalized approximate-reasoning-based intelligent control* (GARIC). One neural network is used to evaluate the performance of the fuzzy system. One neural network is used to adjust the membership function based on the evaluation network's output. Simulations for the classical cart-pole balancing problem are performed.

Blanco and Delgado present their work in the area of neuro-fuzzy techniques [36]. They suggest that a neural network's strength lies in its ability to approximate a function from sample data. The parallel in fuzzy system applications would be the need to infer an output from a predefined rule base. Blanco and Delgado suggest training neural networks to the knowledge contained in a fuzzy rule base. The neural networks can then be used in the place of a rule base.

Keller, Yager and Tahani justify this approach by pointing to the fact that in fuzzy logic systems "as the number of antecedents' clauses increases, the storage and the computation in the inference process grow exponentially" [43, 44]. Their research indicates that for complex systems, a fuzzy rule base is more efficiently stored in a neural network. Also, due to the parallel nature of a neural network, the inference procedure would require less time in a parallel implementation.

Pedrycz takes a similar approach to combining neural networks and fuzzy logic [39]. His approach, however, does not involve training a generic neural network to emulate a rule base. He introduces two new classes of fuzzy neurons. The aggregative neurons realize AND, OR, and mixed AND/OR operations. The referential neurons realize binary relations of matching, difference, inclusion and dominance. The fuzzy rule base is "encoded" into a neural network by appropriate architecture design and weight selection using these two types of fuzzy neurons.

Pal and Mitra adapt neural networks to include fuzzy inputs and outputs [40]. In their combination of the two technologies, each input is first fuzzified according to the input membership functions. The desired outputs are also fuzzified in order to perform the back propagation during training. The resulting network produces fuzzy membership values for a given input. Pal and Mitra compared their neuro-fuzzy system of classifying vowel sounds to conventional speech recognition systems.

Rahman outlines his use of neuro-fuzzy technology in a practical, home appliance problem [24]. The problem is that a toaster's darkness setting does not take into account the initial temperature of the toaster. Rahman equips a toaster with a temperature sensor and microcontroller. The temperature is read by an eight bit microcontroller that controls the toaster coils. Data was collected off-line and used by a neural network to derive 52 rules and 3 membership functions. After training, the toaster produced toast with almost identical degrees of darkness regardless of the initial temperature of the toaster.

Altrock and Krause have researched neuro-fuzzy technology in embedded automotive control [1]. A 600 rule fuzzy control system was implemented in a 20 inch model and in a full size sedan to provide an anti-skid steering system. Due to the large number of rules and the real-time demands of such a system, the authors used the Gamma aggregational operator and FAM inference. Through a serial link to a PC, a neural network was able to optimize parameters during operation.

Lea, Jani, and Berenji explore the use of neuro-fuzzy control for space shuttle rendezvous and docking [15]. An architecture composed of two networks is used. The two networks are referred to as the *evaluation network* and the *action network.* The evaluation network is a multilayer back propagation neural network dedicated to the "if" side of a fuzzy rule base. Similarly, the action network is dedicated to the "then" side of the rule base.

S. C. Lee and E. T. Lee introduce the fuzzy neuron [38]. The fuzzy neuron incorporates fuzzy logic into the neurons of the neural networks. Their approach is based on the fact that the original neuron model proposed by McCulloch and Pitts produced an all-or-none output [30]. It was quickly realized that neurons with output in the range of [0,1] produced much better results. The authors say that their fuzzy neurons are best used in areas of "soft sciences, such as in prediction making, pattern recognition, and decision making processes." The examples presented are all feedback neural networks, which are best suited for that type of problem.

Pedrycz successfully trained a fuzzy-neuron neural network to control a system with two state variables and two control variables [22]. His approach attaches linguistic terms in the place of numeric weights between the individual processing elements. This makes it easier to interpret what goes on in the network. Pedrycz also points out that fuzzy-neuron controllers make it easier to understand and adjust the trade-off between information granularity and learning capabilities.

Keller and Hunt address the problem of a neural network classifier's inability to terminate training when training samples from two classes are nonseparable [37]. They suggest that training samples that are nonseparable are probably atypical of their respective class. Fuzzy techniques can be incorporated into the perceptron model to put less weight on these atypical samples during training. The researchers respond to the rhetorical question, "Why not just identify the atypical vectors and ignore them?" By making a sample vector's influence during training dependent on how closely it represents its class, the boundaries between the classes can be better characterized than if atypical vectors are ignored.

Trajectory Planning

W. J. Chung, W. K. Chung, and Y. Youm introduce a method for dealing with the problems of inverse kinematics in robotic manipulators with redundant degrees of freedom [33]. They propose the use of *virtual links* and *displacement distribution*. The virtual link scheme divides a redundant manipulator into a series of two or three link manipulators. The displacement distribution assists in planning movements by assigning responsibility for the end-effector's position to each joint. The researcher's simulate their algorithm for planar arms with as many as 10 degrees of freedom.

Paredis and Khosla also address the problem of solving inverse kinematics equations in redundant manipulators [47]. They suggest, however, a numerical approach for obtaining a solution. A system of inequalities and an objective equation describing the robotic arm's task is determined. Then the objective equation is optimized by the simulated annealing technique to iteratively solve for the arm's joint angles. The researcher's numerical solution for a two DOF serial link manipulator agrees with the analytical solution.

Lumelsky applies his dynamic path planning (DPP) approach to various types of two joint robotic arms operating in the presence of obstacles [31]. His non-heuristic approach is based on transforming the arm and obstacle arrangement into *image space*. The DPP image space is made up of regions of virtual boundaries caused by the obstacles in the environment. A path is planned in image space and then transformed into joint space. The DPP algorithm possesses a significant requirement for computation.

Wang and Hamam suggest trajectory planning and collision avoidance by representing the arm and objects as a set of convex polyhedra [46]. A *doubly connected edge table* (DCET) is used to determine the interaction between objects in three dimensional space. A path is iteratively calculated based on the geometric distances from each link of a robotic arm to the obstacle. Their FORTRAN simulation on a VAX II/GPX took "a few minutes" to find an acceptable path. The researchers suggest the use of heuristics to accelerate path planning.

Galicki examines the use of global optimization of potential functions to plan collision-free trajectories [32]. His specific application is that of a three link robotic arm operating in 2d-space in the presence of a single obstacle. Because Galicki's algorithm involves the optimization of both geometric trajectory path and the determination of the time parameterization of the trajectory, an iterative solution of nonlinear equations is necessary. Galicki summarizes, "Numerical solution of the problem ... is a complex and time-consuming task."

Lin and Fu take a divide-and-conquer approach to the collision avoidance and motion planning problem [49]. Their work deals with trajectory planning of an *n* degree of freedom manipulator where *n* is an arbitrary parameter of the system rather than a predefined, small constant. First, the space of operation is divided into subspaces and cells. Then, the problem of finding a collision free path is further decomposed into two separate problems. Because, the path planning is all done in configuration space, the researchers concede that there is not yet an efficient way to practically implement their algorithm.

Chuang and Ahuja use the Newtonian potential functions to plan the path of an object moving in the presence of obstacles [45]. The problem is first broken down into local planning problems and the global planning. For the local planning, the objects are considered to be charged. In this way, the space between the objects is characterized by the repulsive fields generated by the imaginary charge. The path is then planned by navigating a course that minimizes the function describing the fields in the space between the objects.

CHAPTER VI

ROBOTIC ARM PROBLEM

For robots to become effectively used in a wide range of application, they must gain the ability to work in unpredictable and changing environments. Robots used in space exploration and construction will have to sense their environment and carry out their tasks regardless of the presence of objects in the work area. This thesis addresses the problem of planning the trajectory of a three-link robotic arm in the presence of an obstacle.

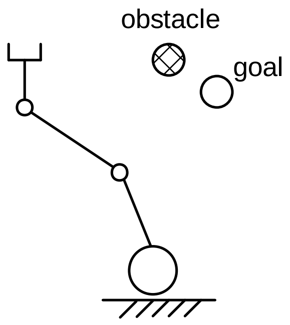


Figure 6-1. A diagram of a robotic arm in the presence of an obstacle.

For the purpose of designing the control system, the position of the obstacle and goal can be assumed to be available from sensory feedback. A vision system, for example, could be used to generate Cartesian coordinates of the objects. Also, the joint angle values are assumed to be available from position feedback sensors in the arm. The system configuration that inspired this problem is shown in figure 6-2.

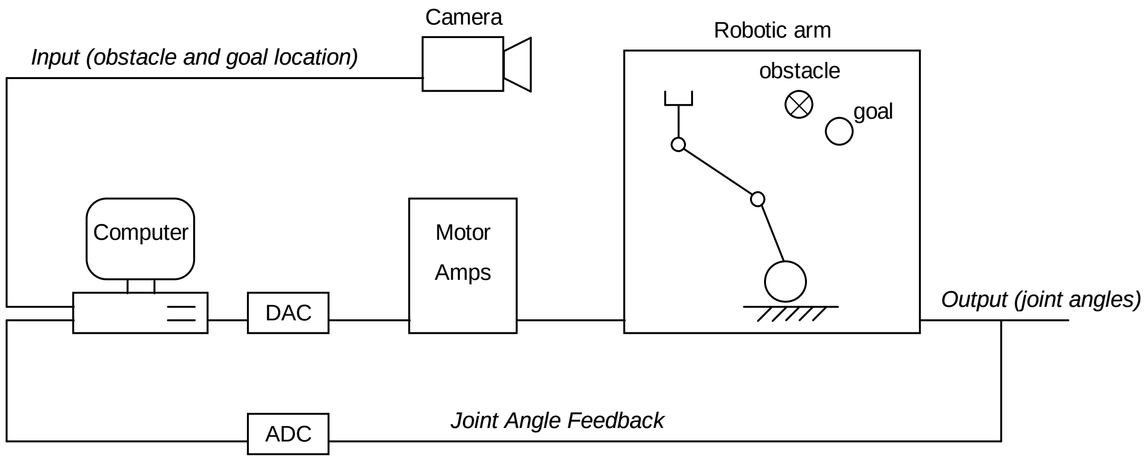


Figure 6-2. A sample configuration for the problem application.

The arm should operate in two dimensions in an environment containing a randomly placed obstacle. The starting position and desired position of the arm are arbitrary, as well. The arm will be modeled as a three-link planar manipulator. The controller will determine a series of joint angles, Θ(t), that move the end effector from a given starting position (x_s, y_s) to a desired final position (x_g, y_g) without colliding with the obstacle at (x_o, y_o).

The robotic arm's end-effector position in Cartesian space can be directly related to its link lengths and joint angles by the following equations and figure 6-3.

$$x_e = l_1 \cos q_1 + l_2 \cos(q_1 + q_2) + l_3 \cos(q_1 + q_2 + q_3) \qquad (10)$$

$$y_e = l_1 \sin q_1 + l_2 \sin(q_1 + q_2) + l_3 \sin(q_1 + q_2 + q_3) \qquad (11)$$

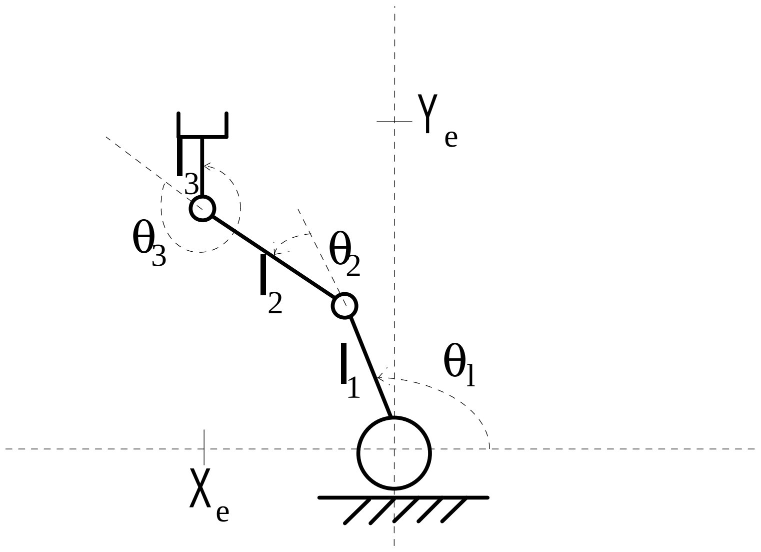


Figure 6-3. Robotic arm variable definitions.

Notice that because the model of the arm contains three joints and is operating in a two-dimensional plane, redundancy exists in possible configurations of the joint angles for a given Cartesian coordinates location for the end-effector. In one respect, this redundancy complicates the problem because there is no unique mapping available to convert a coordinate in Cartesian space to a specific point in joint space. In fact, given a desired position of the end-effector, there is an infinite number of solutions for possible joint angles. On the other hand, this gives the designer the freedom to select the arm movements based on power efficiency, or a straight line trajectory, or even on the observed behavior of an expert's control of an arm (as is the case in this thesis).

The physical characteristics of the arm will be based on those of a Remotec industrial robotic arm available in the Intelligent Control Systems Lab of Texas A&M University - Kingsville. The characteristics of importance for this problem are the link lengths and the physical constraints of the joints. Table 6-1 lists these values.

Physical Characteristic	Minimum Angle	Normalized Length	Maximum Angle

Link length, l_1	—	1	—
Link length, l_2	—	0.824	—
Link length, l_3	—	0.737	—
Joint constraints on q_1	0°	—	+180°
Joint constraints on q_2	-90°	—	+90°
Joint constraints on q_3	-25°	—	+25°

Table 6-1. The physical characteristics of a Remotec robotic arm.

The goal must be assumed to lie within the possible reach of the end-effector and the obstacle must be assumed to lie outside the path of link, l_1. Using the direct kinematics equations 10 and 11, the constraints in table 6-1, and one final constraint that $y_e \geq 0$, all possible locations for the end-effector can be determined and are illustrated in figure 6-4.

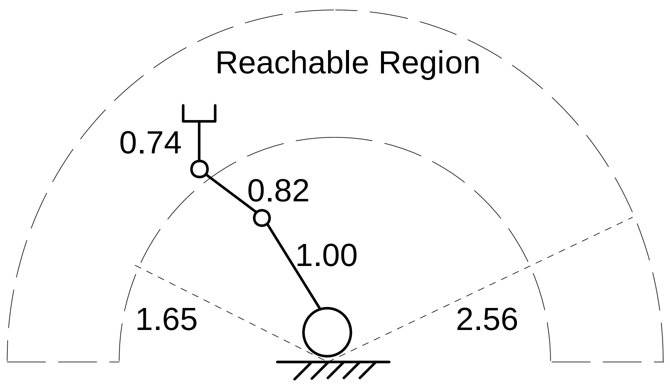


Figure 6-4. Normalized dimensions for Remotec arm and reachable region.

The problem therefore can be summarized as follows:

GIVEN A ROBOTIC ARM WITH:

- the current joint angles, $\Theta(0)$,
- an obstacle located at a known position, (x_o, y_o),
- and a desired position, or goal, (x_g, y_g);

DETERMINE A TRAJECTORY, $\Theta(t)$, SUCH THAT:

- the end-effector reaches the goal,
- the arm does not touch the obstacle,
- and the calculations can be performed in real-time with current hardware.

CHAPTER VII

NEURO-FUZZY SOLUTION

The problem described in the previous chapter and addressed in this thesis is actually a series of sub-problems. Planning the trajectory of a robotic arm has been itself a topic of many research projects. Planning the trajectory of a redundant robotic arm adds a new dimension to the problem -- more degrees of freedom, more nonlinear equations. Place an obstacle in the work area and the problem becomes a challenge of optimizing differential equations, iterative numerical solutions, and "guess and shoot" methods.

Chapter five gives a good summary of the work done in the areas of this problem. The largest disadvantage of previous solutions is the computer resources necessary to solve a system of equations or optimize a performance index. Researchers acknowledge that "minutes" are necessary to plan one trajectory on some of the world's fastest computers. Simply determining the equations to be solved is a formidable task. What is needed is an approach that is easy to apply and will work in real-time.

This problem is well addressed by neuro-fuzzy techniques because a solution is not easily found by analytical or numerical techniques. While an analytical technique is difficult, moving an arm in the presence of an obstacle can be instinctively performed by a child. Neuro-fuzzy systems excel in using sample data to determine an input-output relationship. As explained in Chapter four, neuro-fuzzy technology is the fusing of both neural networks and fuzzy logic. Neural networks bring to this solution the ability to learn while fuzzy logic is based on mimicking an expert's thinking. In addition, as hardware technology progresses, more and more value will be placed on solutions that can utilize parallel processing, like neural networks.

The field of neuro-fuzzy technology has gone in many directions. The neuro-fuzzy technique implemented in this thesis is that of replacing the fuzzy logic rule base of a traditional fuzzy logic system with a multilayer back propagation neural network. Figure 7-1 illustrates the type of neuro-fuzzy system used in this problem.

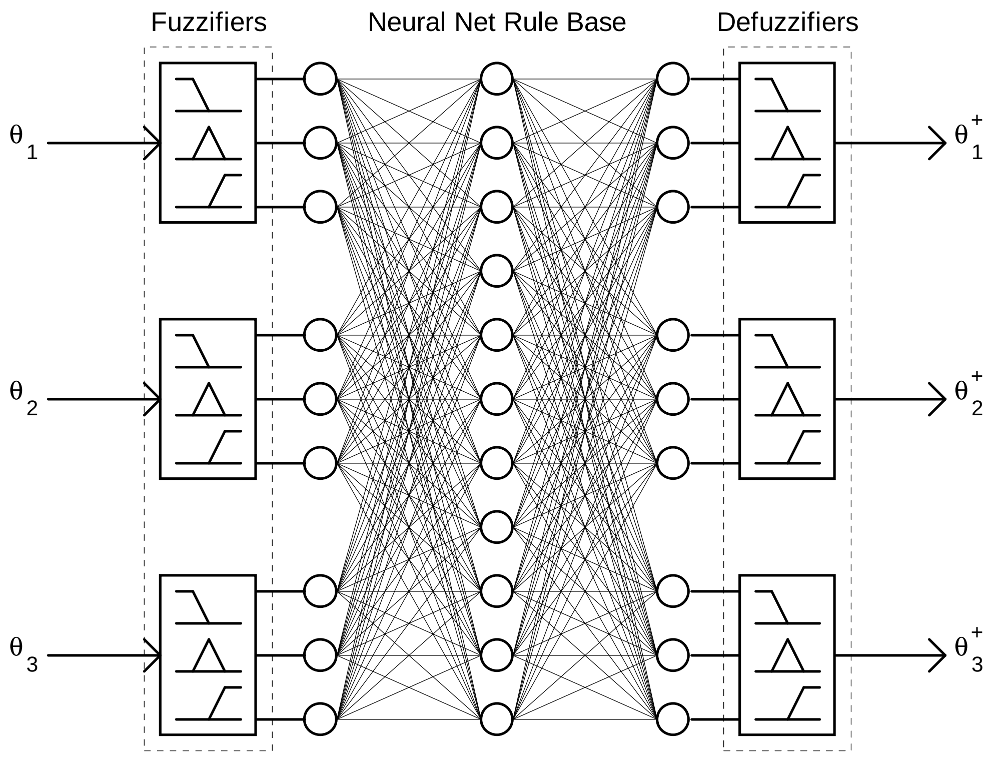


Figure 7-1. A sample neuro-fuzzy system.

This type of system is beneficial for several reasons. While it is true that a child is able to move an arm around an obstacle to reach a desired goal, that ability is intuitive. Putting the instructions for performing such a task into a neat, fuzzy logic, IF/THEN rule base is not easy. Thus, there is a necessity for the neural network to learn the rules. The fuzzifiers and defuzzifiers necessary for any fuzzy system provide an interface between an expert's control of a simulated arm and the neural network.

The problem was addressed in the following steps, which will serve as an outline for the remainder of this chapter. The state representation of the system, which would serve as the controller's input, was considered. A simulator of the Remotec robotic arm was written. The simulator included a graphical display of a three-link planar manipulator, an obstacle and a goal. The simulator was controllable from the keyboard and recorded all movements of the arm under "expert" control. A generic neuro-fuzzy system was programmed. The recorded data was used as the training set for the neuro-fuzzy system. The simulator and neuro-fuzzy controller were combined to simulate the arm under neuro-fuzzy control.

State Representation

One of the primary considerations in the design of the neuro-fuzzy controller is the representation of the state of the system. The ability of the final neuro-fuzzy controller to generalize a solution from training data will depend largely on the data representation scheme in the system. The question about whether to input joint angles, Cartesian coordinates, relative distances, or a combination of these into the neuro-fuzzy network will also determine whether or not the system can converge during training.

The following considerations guided the selection of state representation and are based on the fact that situations important to the trajectory chosen, should be an important part of the state representation scheme.

- The state representation must excite the system if the end-effector is not at the correct position. In other words, a steady state error of the end-effector's position must produce an error reducing response. Therefore, the distance from the end-effector to the desired location goal should be an input to the controller.
- The state representation should be similar for similar relative arrangements. For example, the two arrangements of arm, obstacle, and goal in figure 7-2 would have similar acceptable trajectories and should therefore have similar state representation. A scheme based solely on Cartesian coordinates may not allow a neuro-fuzzy system to generalize similar strategies for the two arrangements below.

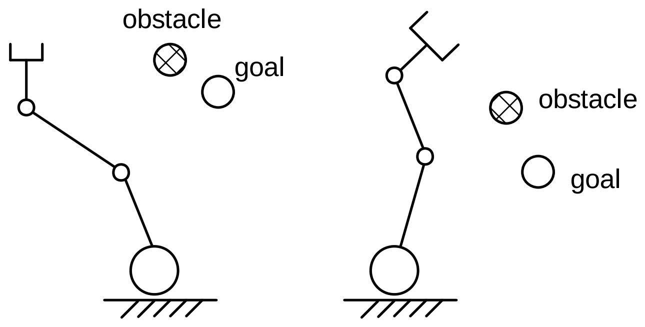


Figure 7-2. The relative position of the objects is the same.

- A scheme based on joint space might cause a neuro-fuzzy system to interpolate similar strategy for completely different situations. The representation should be different for the cases in figure 7-3.

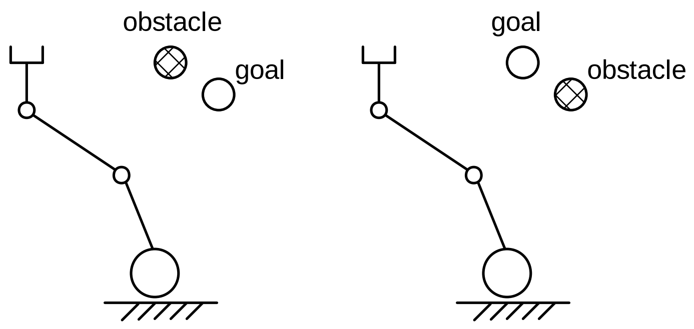


Figure 7-3. The obstacle and goal switch places.

- The representation should take into account the physical constraints so that interpolation into the table or past a joint's capabilities does not occur.
- The output of the system will be the three small changes in angle, $\Delta\theta_i$, that should be made to each of the three joints to move the end-effector toward the goal without hitting the obstacle.

These guidelines are not sufficiently met by using only joint space or only Cartesian coordinates. On the basis of these guidelines, the following inputs and outputs were chosen to represent the system and are illustrated in figure 7-4.

SEVEN INPUTS:

θ_1, joint angle of link1 to the base

θ_2, joint angle of link2 from link1's axis

θ_3, joint angle of link3 from link2's axis

x_o, horizontal distance from the end-effector to the obstacle

y_o, vertical distance from the end-effector to the obstacle

x_g, horizontal distance from the end-effector to the goal

y_g, vertical distance from the end-effector to the goal

THREE OUTPUTS:

$\Delta\theta_1$, a small change in the angle between link1 and the base

$\Delta\theta_2$, a small change in the angle between link2 and the link1 axis

$\Delta\theta_3$, a small change in the angle between link3 and the link2 axis

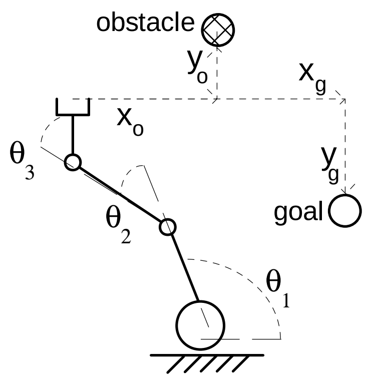


Figure 7-4. The state representation for the obstacle problem.

ROBOTIC ARM SIMULATION

In order to obtain the training data and test the robotic arm controller, it was necessary to write a simulation program of the robotic arm. The simulator was written in C on a PC, as was all the code for this thesis project. Appendix I contains a complete listing of the programming done during this project. Included in this chapter are the data structure definitions. The data structure definitions are the foundation of well-organized programming code and they provide insight into the nature of the rest of the program.

The primary data structure in the simulator was the *Link* structure. The *Link* structure is defined as shown in figure 7-5. A robotic arm is programmed as a linked list of *Link*s. The obstacle and goal are represented on the screen as circles. The simulated arm is controlled from the keyboard. The program records its data in two data files. One data file contained the state vectors which acted as the inputs to the neuro-fuzzy system during training. The other data file stored the "expert's" movements and served as the desired output of the training system. All the data was normalized to be in the range of (0, 1). Figure 7-6 shows a screen snap-shot of the robotic arm simulator and a sample trajectory while under "expert" control.

```
struct Link {             /* A link extending from a joint */
  double len;             /* Length of link in pixels */
  double theta;           /* Angle relative to previous link */
  double max_theta;       /* Max theta caused by joint constraint */
  double min_theta;       /* Min theta caused by joint constraint */
  double phi;             /* Angle relative to horizontal */
  struct Link *next;      /* Pointer to next link in linked list */
  };
```

Figure 7-5. The robotic arm LINK data structure.

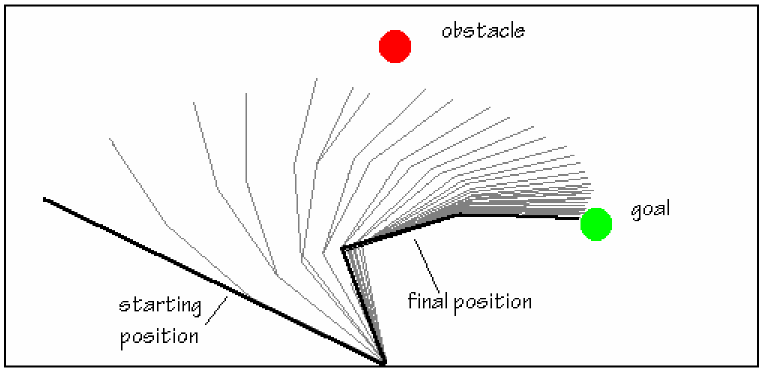


Figure 7-6. A sample training trajectory obtained from the simulator.

The training samples were obtained by cycling through a series of goal and obstacle positions. The training samples included two starting positions of the arm -- (θ_1 = 0; θ_2 = 0; θ_3 = 0) and (θ_1 = 180; θ_2 = 0; θ_3 = 0). Goal and obstacle positions were taken from the entire reachable region. As the training samples were recorded, each movement created a new input to the neuro-fuzzy system. Therefore, selecting starting positions of the arm at the two extremes of possible movements generated training samples that extended over the entire state space. Obtaining training samples over an entire range of possible state space is important for neuro-fuzzy design.

NEURO-FUZZY PROGRAMMING

Once the training samples were normalized, and in ASCII data files, the code to run the neuro-fuzzy system was written. The important data structures in the neuro-fuzzy code included *Node*, *Layer*, and *Net*. The definition of these data structures are shown in figure 7-7.

```
struct Node {              /* a neural network node */
  int numweights;          /* number of inputs to the node */
  double *win;             /* current weights array */
  double *winlast;         /* last weights array */
  double sum;              /* sum of inputs*weights */
  double output;           /* activation output */
  double error;            /* BPN error at this node */
  };

struct Layer {             /* an in, out, or hid layer */
  int numnodes;            /* number of nodes in this layer */
  struct Node *nodes;      /* ptr to next node in this layer */
  };

struct Net {               /* a neural network */
  int numlayers;           /* the number of layers in this NN */
  struct Layer *layers;    /* ptr to next layer in this NN */
  };
```

Figure 7-7. The neuro-fuzzy data structures.

The neuro-fuzzy engine is completely void of any constants or programming related to a robotic arm. Consequently, it can be used for any input/output data. The number of inputs, the number of nodes and even the fuzzy membership functions are defined for individual systems in an ASCII text file. This text file is termed the FNN data file. The weights for a particular training set are stored in a text file, which is also specified in the FNN data file. A sample FNN data file is shown in figure 7-8.

```
rem The number of inputs and outputs on the neuro-fuzzy network
inputs=7
outputs=3

rem The number of hidden layers and nodes
hidden=1
hidnodes=65

rem The activation slope constant and training parameters
alpha=1
beta=.5
gamma=.1
epochs=300
maxerror=.01

rem The data files
input_file=input.dat
samples_file=input.dat
desired_file=output.dat
weights_file=weights.dat
error_file=error.dat

rem Number of fuzzy values for each input and output
input_values=7 7 7 3 3 3 3
output_values=3 3 3

rem The fuzzy membership function data
input_max=
0.0 0.166 0.333 0.5 0.666 0.833 1.0
0.0 0.166 0.333 0.5 0.666 0.833 1.0
0.0 0.166 0.333 0.5 0.666 0.833 1.0
0.0 0.5 1.0
0.0 0.5 1.0
0.0 0.5 1.0
0.0 0.5 1.0

output_max=
0.0 0.5 1.0
0.0 0.5 1.0
0.0 0.5 1.0
```

Figure 7-8. A sample FNN data file.

THE NEURO-FUZZY CONTROLLER

Once training samples are obtained and a neuro-fuzzy system is trained to this data, the controller can be tested on the robotic arm simulator. The tests are performed by placing the objects at random locations and assigning a random starting position for the arm. The controller simulation must quantify the performance of the proposed neuro-fuzzy controller. To do so, the original design specifications of the problem statement should be addressed. The design specification of Chapter 6 listed the following as the goals of the system:

DETERMINE A TRAJECTORY, Θ(t), SUCH THAT:

- the end-effector reaches the goal,
- the arm does not touch the obstacle,
- and the calculations can be performed in real-time with current hardware.

In order to judge the controllers based on these guidelines, the simulated neuro-fuzzy controller determines the percentage error in reaching the goal, determines whether or not a collision occurred, and performs all the necessary calculations so that a judgment can be made about the possibility of real-time application. The first and third criteria are straight-forward. A collision with the obstacle on any part of the arm can be detected from the joint angles and obstacle position.

Given a robotic arm, composed of a series of links, operating in the presence of an obstacle whose Cartesian coordinate position is known, the shortest distance between the obstacle and the arm is the minimum of the distance between the obstacle and each link of the arm. The distance, *d*, from a link to an obstacle is determined by equations 12 through 17 where all the variables are defined in figure 7-9.

$$d_{12} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{12}$$

$$d_{10} = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2} \tag{13}$$

$$d_{02} = \sqrt{(x_0 - x_2)^2 + (y_0 - y_2)^2} \tag{14}$$

$$q_1 = \arccos(\frac{d_{12}{}^2 + d_{10}{}^2 - d_{02}{}^2}{2d_{12}d_{10}}) \tag{15}$$

$$q_2 = \arccos(\frac{d_{02}{}^2 + d_{10}{}^2 - d_{12}{}^2}{2d_{02}d_{10}}) \tag{16}$$

$$\begin{aligned} d &= d_{10} && \text{for } (q_1 > \tfrac{p}{2}), \\ d &= d_{02} && \text{for } (q_2 > \tfrac{p}{2}), \\ d &= d_{10}\sin(q_1) && \text{other wise} \end{aligned} \tag{17}$$

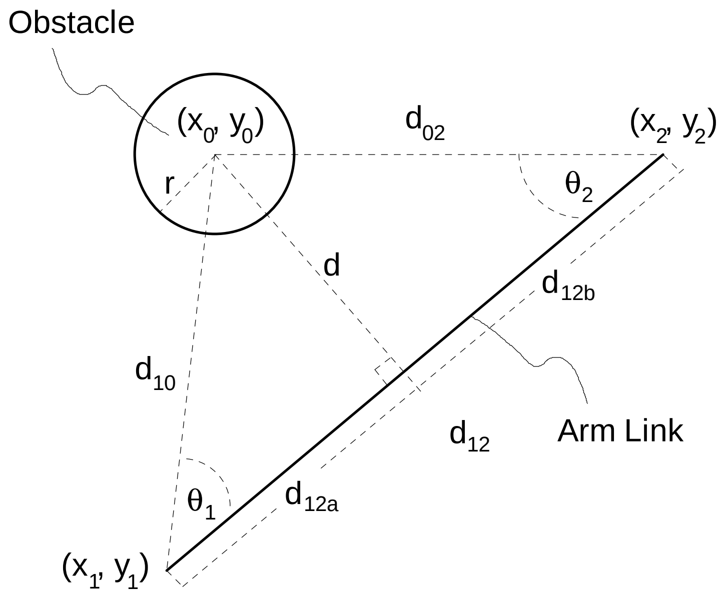


Figure 7-9. Determination of a collision between the obstacle and a link.

If *d* < *r*, a collision has occurred. The function that checks for a collision is shown in figure 7-10.

```
/*********************************************************************
*
*  Determine if the arm is intersecting an object
**********************************************************************
/
int Collision(struct Link *links, struct Point *obstacle, double
margin)
{
  double d, d12, d10, d02, d12a, d12b, t0, t1, t2;
  struct Link *l;
  struct complex p0, p1, p2;
  struct Point p;

  Pol2Rec(obstacle, &p0);
  p1.x = 0;
  p1.y = 0;

  for(l = links; l!=NULL; l = l->next)  {
    p.rho = l->len;
    p.phi = l->phi;
    Pol2Rec(&p, &p2);
    p2.x += p1.x;
    p2.y += p1.y;

    d12 = Distance(&p1, &p2);
    d10 = Distance(&p1, &p0);
    d02 = Distance(&p0, &p2);
    t1 = acos( (d12 * d12 + d10 * d10 - d02 * d02) / (2 * d12 * d10));
    t2 = acos( (d12 * d12 + d02 * d02 - d10 * d10) / (2 * d12 * d02));

    d = d10 * sin(t1);
    if(t1 > M_PI_2) d = d10;
    if(t2 > M_PI_2) d = d02;
    if(d < margin) return ON;
    p1.x = p2.x; p1.y = p2.y;  }
    return OFF;
}
```

Figure 7-10. Source code for collision detection function.

CHAPTER VIII

EXPERIMENTAL RESULTS

To test the use of neuro-fuzzy systems on control of a robotic arm, first a simulator was used to generate training samples from a human's example of controlling of an arm. Then, the training samples were used to train seven different neuro-fuzzy controllers. The seven controllers are named FNN1, FNN2, ... FNN7 for reference. The controllers varied in the definition of their fuzzy membership functions. The actual text files which defined the fuzzy membership functions for each controller are shown in Appendix II. This chapter contains the results of the training sessions and the testing of each controller on the simulated arm.

The results are primarily shown in three different types of graphs. The fuzzy membership function graphs define the way the crisp values are converted into fuzzy values. There is a fuzzy membership function graph for each of the seven controllers. The RMS training error graph indicates how well each neuro-fuzzy controller was able to learn the training data. The performance histograms quantify the controller's performance in moving the simulated arm from the starting position to the goal without touching the obstacle.

Figure 8-1 is a graph of the FNN1 fuzzy membership function. It assigns a crisp value's degree of membership to three fuzzy values -- NEG, ZERO, and POS. The fuzzy membership functions used in the controllers FNN2 and FNN3, on the other hand, have seven fuzzy values -- NL, NM, NS, ZERO, PS, PM, and PL. Figures 8-2 and 8-3 illustrate the difference between FNN2 and FNN3. Fnn3 has a concentration of membership values near ZERO. This concentration of values near the center of the function definition is motivated by the fact that the motion strategy changes as the arm approaches the obstacle or goal. Figure 8-4 shows the learning capabilities of these three controllers. Figure 8-5 compares their performance.

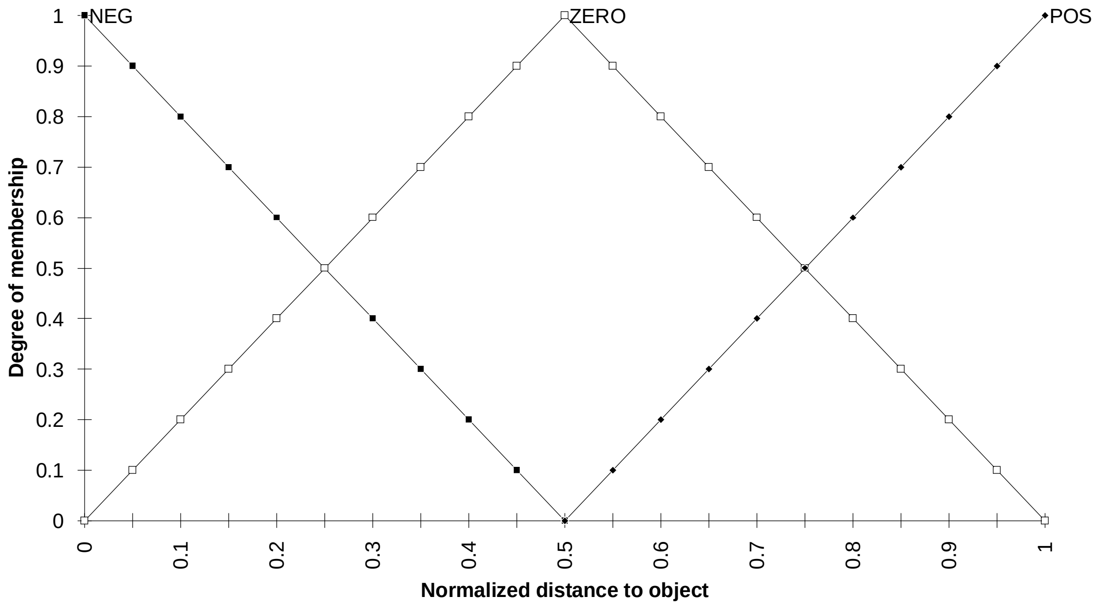


Figure 8-1. The fuzzy membership function definition FNN1.

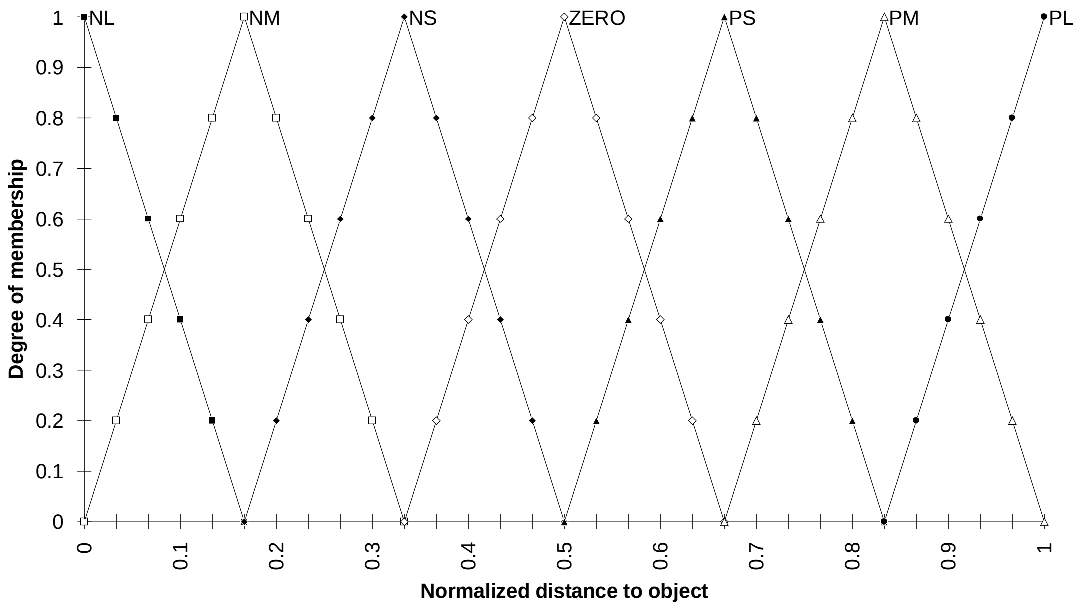


Figure 8-2. The fuzzy membership function definition FNN2.

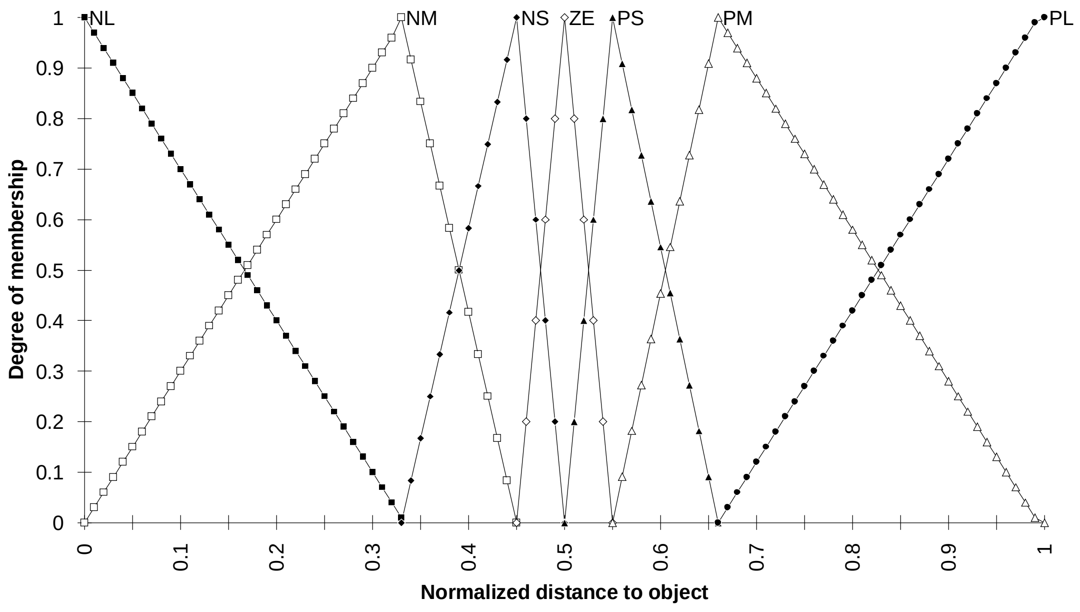


Figure 8-3. The fuzzy membership function definition FNN3.

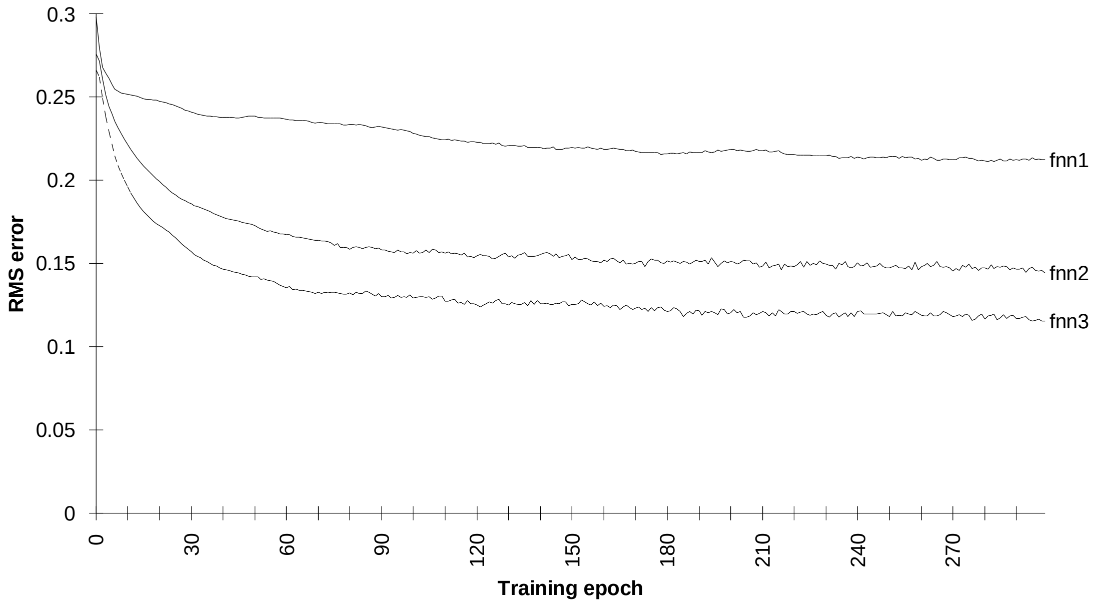


Figure 8-4. The RMS training error of FNN1, FNN2, and FNN3.

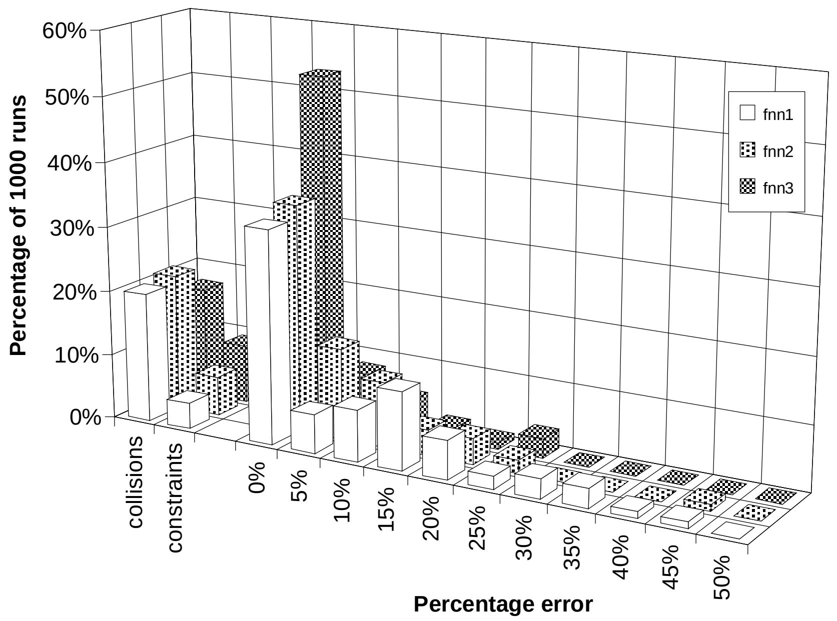
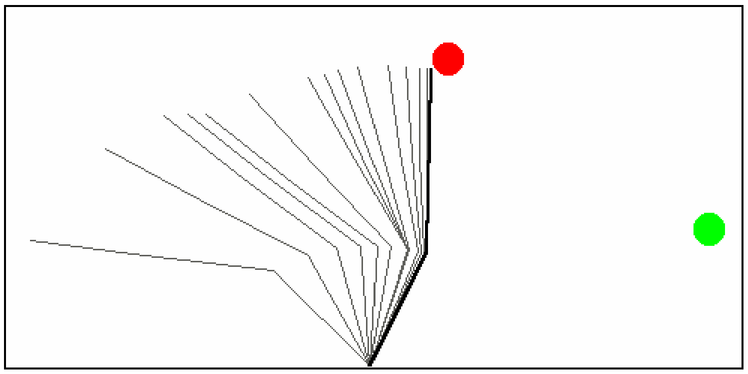
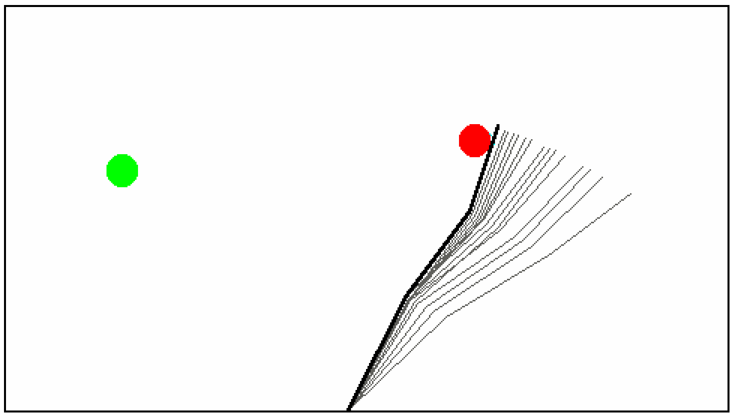


Figure 8-5. Performance histogram of the neuro-fuzzy controller for FNN1, FNN2 and FNN3.

A comparison of the different controllers can be made from the performance histograms. The "collisions" category of the histogram records the percentage of runs that resulted in a collision with the obstacle. Figure 8-6 (a) and (b) are screen-shots of the simulator under neuro-fuzzy control. It shows samples of the controller moving the arm into the obstacle. When the neuro-fuzzy controller attempted to move the arm past the joint's capabilities, the "constraint" category was incremented. Figure 8-7 shows samples of this case. The percentage error categories record how close the end effector was to the goal at the arm's final position. Figure 8-8 (a) and (b) are actual trajectories planned by the neuro-fuzzy controller that incremented the 0%-5% error category.

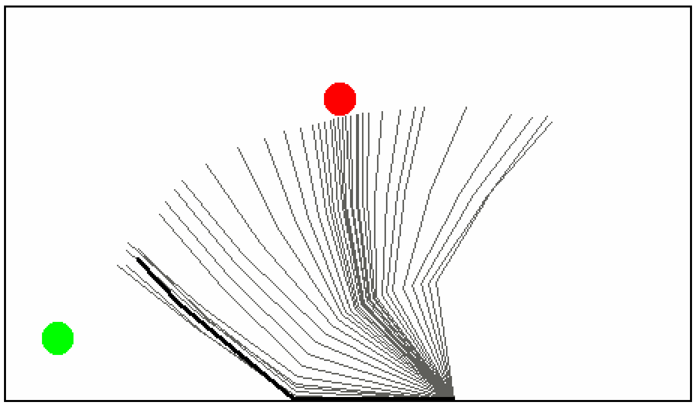


(a)

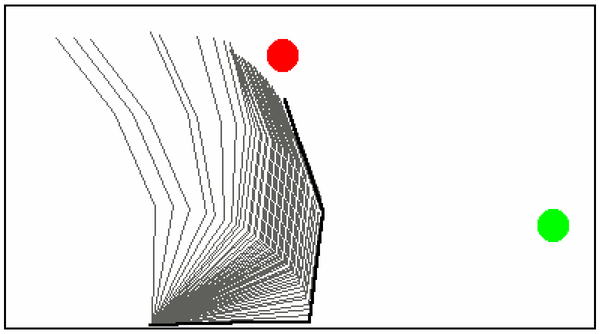


(b)

Figure 8-6. Samples of collisions while under neuro-fuzzy control.

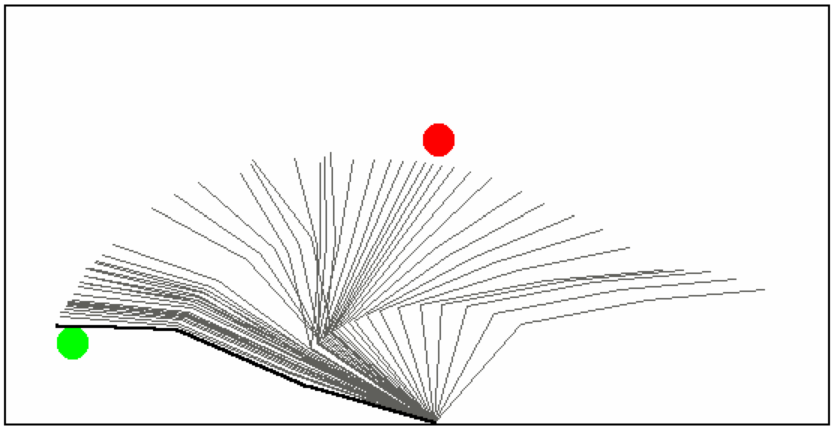


(a)

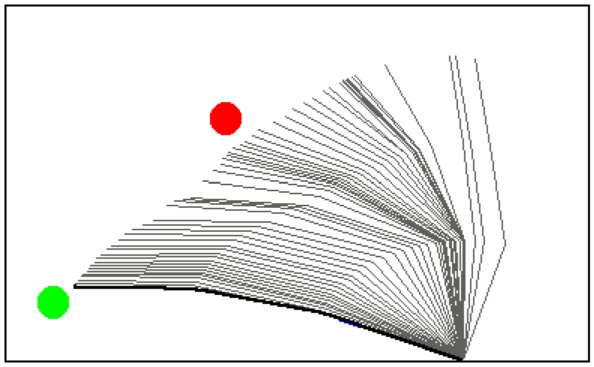


(b)

Figure 8-7. Samples of neuro-fuzzy control in which constraint limits were exceeded.



(a)



(a)

Figure 8-8. Samples of successful control by the neuro-fuzzy controllers.

The fuzzy membership function definitions FNN4 and FNN5 increase the granularity to 11 fuzzy values as shown in figures 8-9 and 8-10. Similar to FNN3, FNN5 emphasizes the region near the center of operation. The training data and performance histogram are shown in figures 8-11 and 8-12, respectively.

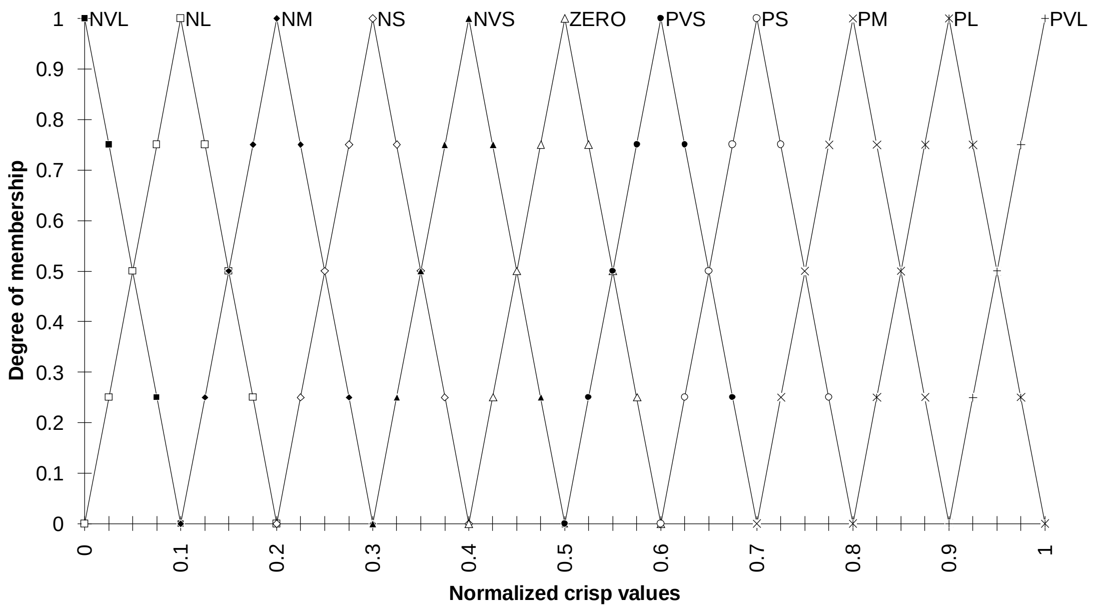


Figure 8-9. The fuzzy membership function definition FNN4.

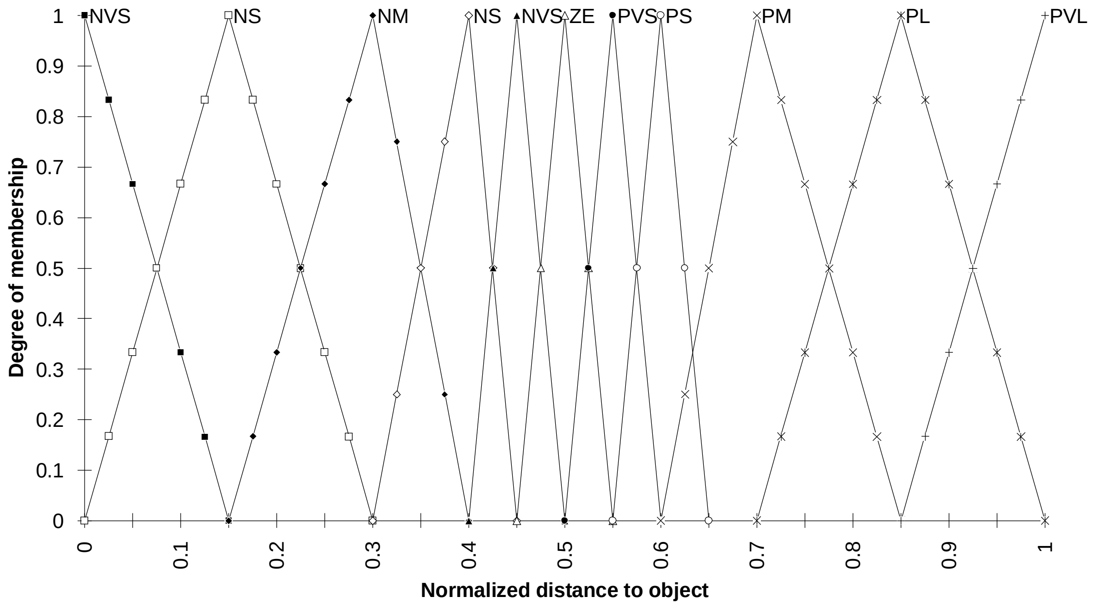


Figure 8-10. The fuzzy membership function definition FNN5.

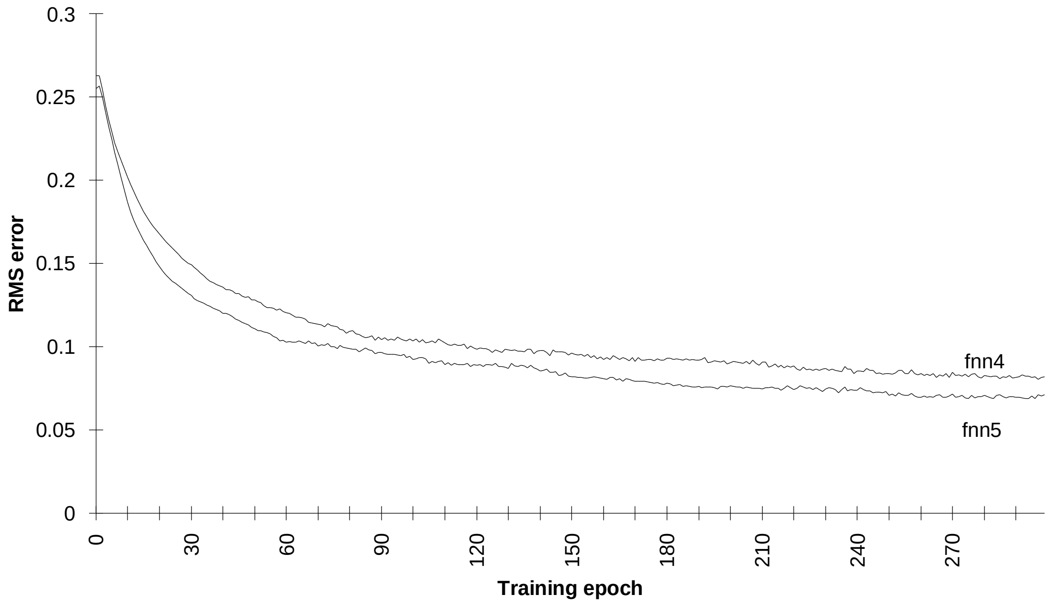


Figure 8-11. The RMS training error of FNN4 and FNN5.

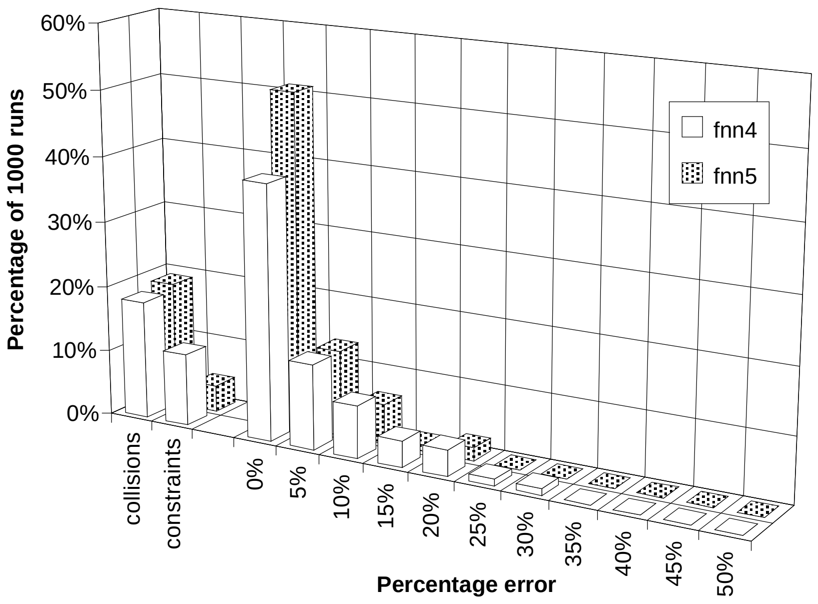


Figure 8-12. Performance histogram of the neuro-fuzzy controller for FNN4 and FNN5.

The last two controllers increased the number of fuzzy values to fifteen. FNN6 and FNN7 are shown in figures 8-13 and 8-14, respectively. Figures 8-15 and 8-16 show the controllers' error curves and performance histograms.

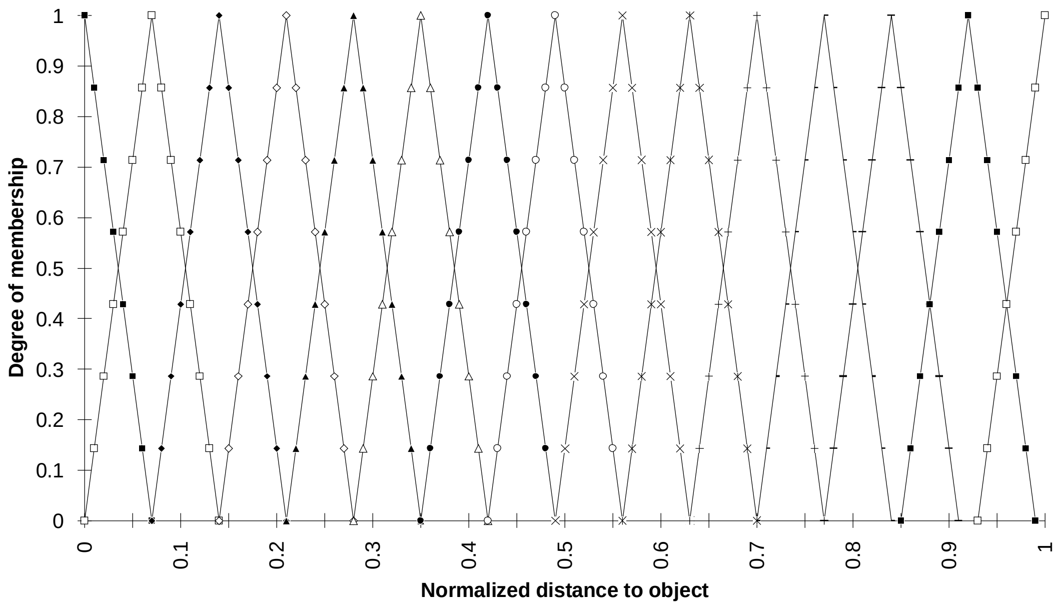


Figure 8-13. The fuzzy membership function definition FNN6.

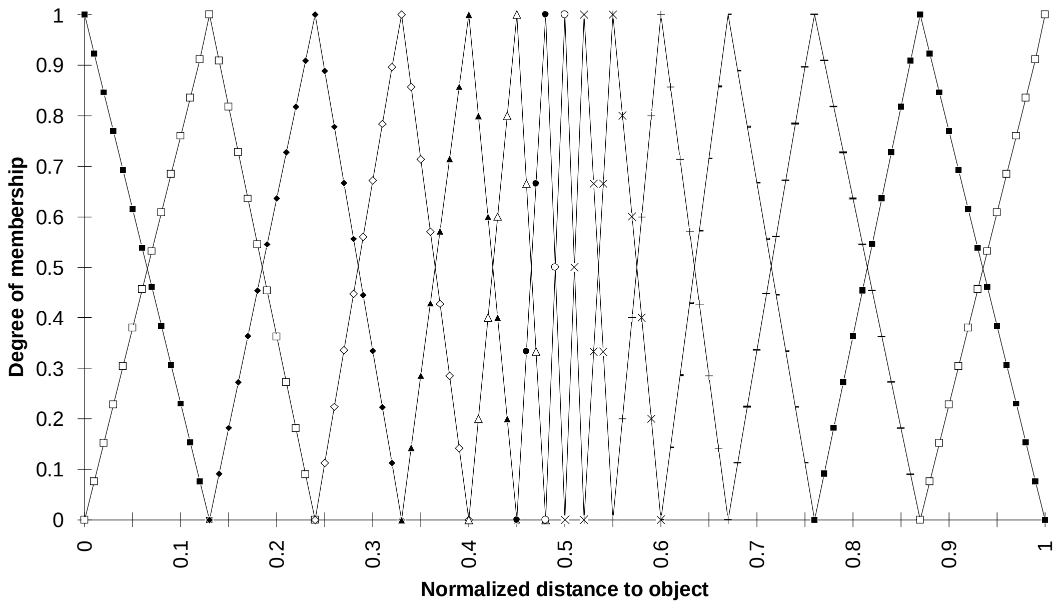


Figure 8-14. The fuzzy membership function definition FNN7.

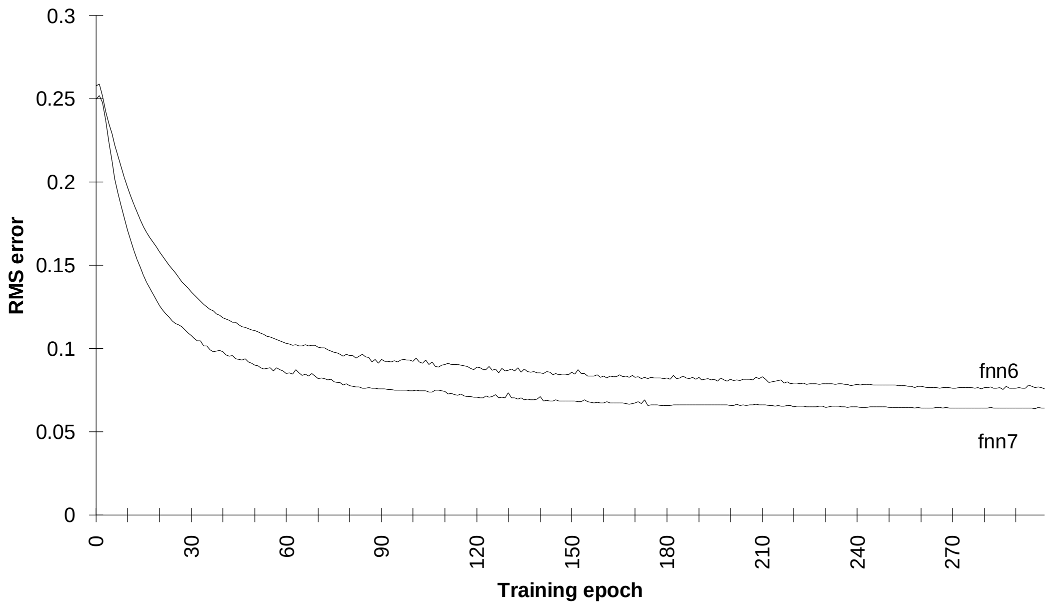


Figure 8-15. The RMS training error of FNN6 and FNN7.

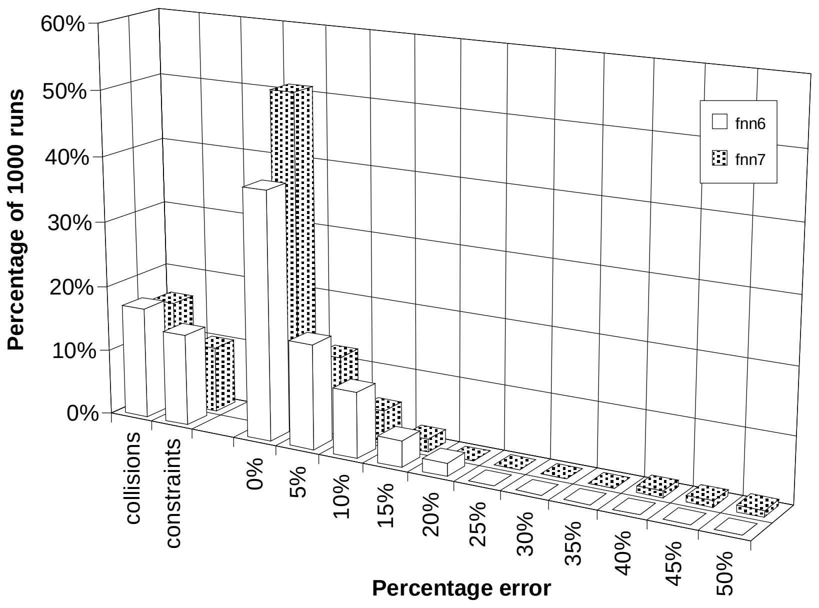


Figure 8-16. Performance histogram of the neuro-fuzzy controller for FNN6 and FNN7.

Two more charts are helpful in interpreting the experimental results. Figure 8-17 compares the final RMS training error of the seven controllers. Figure 8-20 compares the collisions and success rate for the seven controllers.

Finally, many attempts were made to train a back propagation neural network to control the simulated arm. The neural networks without the fuzzification layer were not able to learn from the sample data.

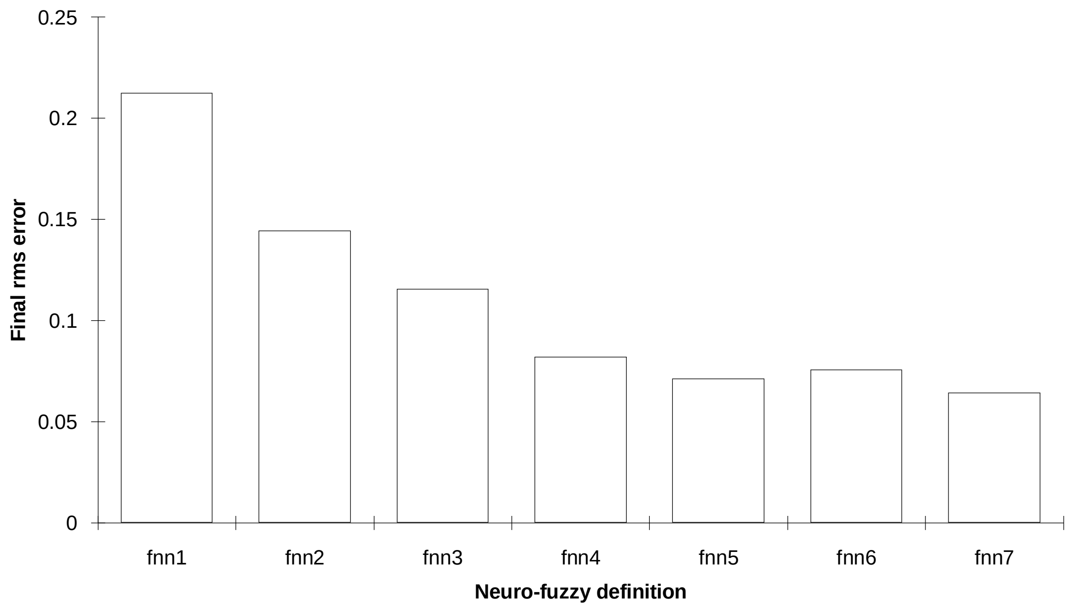


Figure 8-17. A comparison of the final RMS training errors.

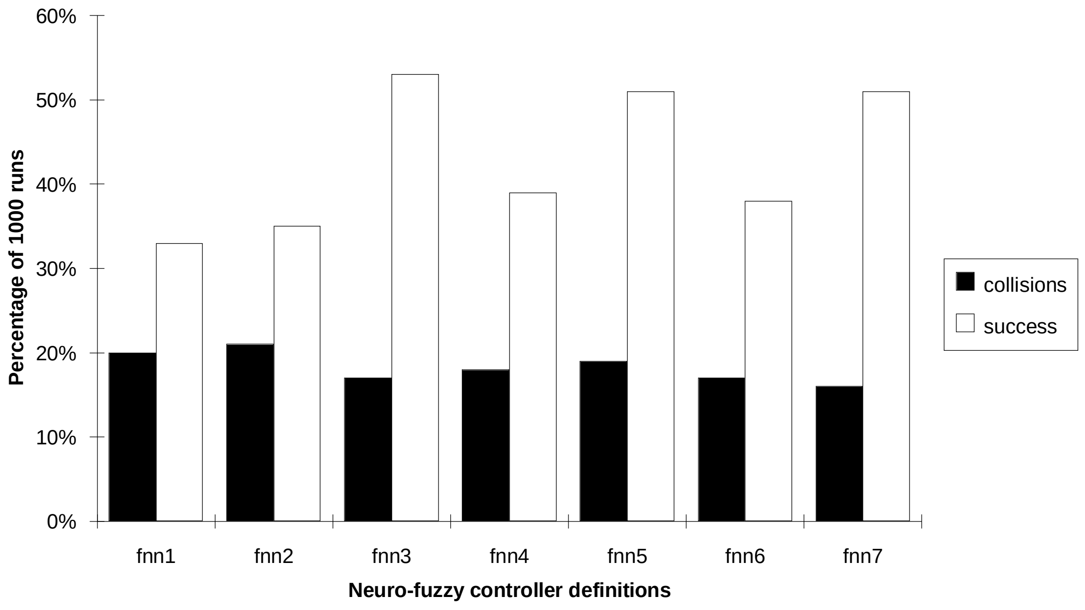


Figure 8-18. A comparison of the performance of all the controllers.

CHAPTER IX

CONCLUSION AND FUTURE WORK

The field of neuro-fuzzy technology will become an important part of intelligent control. The ability to learn how to control a process from sample data is its biggest asset. In this thesis, seven neuro-fuzzy controllers were trained to emulate a human's example control of a robotic arm. From the experimental results of Chapter VIII, two conclusions can be drawn about the application of neuro-fuzzy control.

First, the membership function definitions are an important part of the neuro-fuzzy system. Figure 8-17, on page 61, shows that the final training error was reduced 50% by increasing the number of fuzzy values in a fuzzy set. Figure 8-18, also on page 61, shows that membership functions, defined with an important aspect of the problem in mind, consistently improved performance by more than 25%.

Second, the fuzzification of a neural network's inputs and outputs allows neural networks to learn more complex functions than ever before. The function being learned in this thesis project is very complex. An AI researcher would have a difficult time training a traditional neural network to the sample data used in this work. Not only were the neuro-fuzzy networks able to converge to a solution, they did so in a relatively few number of training epochs and with as many as 100 hidden nodes.

The results of Chapter VIII give a good indication of the nature of neuro-fuzzy systems. The performance of the neuro-fuzzy controllers in this specific application, however, is less than perfect. Even the best controller had a collision rate of 17%. What would improve the performance? What more needs to be understood about this new technology?

Programming some heuristic rules into the control of the arm could improve performance. A large percentage of the "failures" resulted from the controller attempting to move past the physical joint constraints. The control program could move the arm away from such limitations. Another situation that decreases performance occurs when the arm starts oscillating between two points. While ideally, these situations would naturally not occur in the neural network, a few heuristics in the control program could decrease these problems.

A trained neuro-fuzzy system is only as good as the training data used to train it. The seven FNN controllers of this project were trained to 2150 training samples. Future work in this area should look into determining the adequacy of training samples. Are there enough training samples? Are all areas of the state space represented? A study of the relationship of the training samples and the fuzzy membership functions would be particularly helpful.

The use of neuro-fuzzy systems for control has been examined. It is the opinion of this researcher that fuzzification of a neural network's inputs and outputs will become standard procedure in neural network applications.

REFERENCES

[1] C. Altrock and B. Krause, "Fuzzy Logic and Neurofuzzy Technologies in Embedded Automotive Applications", *Proceedings of Fuzzy Logic '93*, pp. A113-1 - A113-9.

[2] H. Berenji, Y. Chen, C. Lee, S. Murugesan, and J. Jang, "An Experiment-based Comparative Study of Fuzzy Logic Control" *Proceedings of the American Association for Artificial Intelligence Conference*, 1988, pp. 2751 - 2753.

[3] M. Caudill, "Neural Networks Primer, Part I", *AI Expert*, (December, 1987), pp. 46-52.

[4] M. Caudill, "Neural Networks Primer, Part II", *AI Expert,* (February, 1988), pp. 55-61.

[5] M. Caudill, "Neural Networks Primer, Part III", *AI Expert* (June, 1988), pp. 53-59.

[6] E. Cox, "Fuzzy Fundamentals" *IEEE Spectrum*, (October 1993), pp. 58-61.

[7] L. N. Cooper, "Hybrid Neural Network Architectures: Equilibrium Systems That Pay Attention", in *Neural Networks: Theory and Applications* (ed. Richard J. Mammone and Yehoshua Y. Zeevi) Boston: Academic Press, Inc., 1991, pp. 81-96.

[8] A. Guez, J. Eilbert, and M. Kam, "Neural Network Architecture for Control", *IEEE Control Systems Magazine* (April, 1988), pp. 22 - 25.

[9] R. Hecht-Nielson, "Neurocomputer Applications" in *Proceedings of the 1987 IEEE Asilomar Signals & Systems Conference*. IEEE Press, 1988.

[10] D. V. Hillman, "Integrating Neural Nets and Expert Systems", *AI Expert,* (June, 1990) pp. 54-59.

[11] J. Jang, "Self-Learning Fuzzy Controllers Based on Temporal Back Propogation", *IEEE Transactions on Systems, Man, and Cybernetics*, 1992.

[12] E. Khan, "An Elegant Combination of Fuzzy Logic & Neural Nets", *Proceedings of Fuzzy Logic '93*, pp. A223-1 - A223-7.

[13] S. Kong and B. Kosko, "Comparison of Fuzzy and Neural Truck Backer-Upper Control Systems" in *Neural Networks and Fuzzy Systems*. Englewood Cliffs: Prentice Hall Inc., 1992.

[14] B. Kosko, *Neural Networks and Fuzzy Systems*. Englewood Cliffs: Prentice Hall Inc., 1992.

[15] R. Lea, Y. Jani, and H. Berenji, "Fuzzy Logic Controller with Reinforcement Learning for Proximity Operations and Docking", *Fifth IEEE International Symposium on Intelligent Control*, 1990.

[16] Y. Li and C. Lau, "Development of Fuzzy Algorithms for Servo Systems", in *IEEE Control Systems Magazine*, (April, 1990), pp. 65 - 71.

[17] L. McLauchlin, *Supervised and Unsupervised Learning Applied to Robotic Manipulator Control*, (Master's Thesis) Kingsville, TX: Texas A&I University, August, 1993.

[18] D. Nauck, F. Klawonn and R. Kruse, "Combining Neural Networks and Fuzzy Controllers" *Fuzzy Logic in Artificial Intelligence (FLAI93),* ed. Klement, Erich Peter and Slany, Wolfgang, pp. 35-46, 1993.

[19] C. Neffenger, "Fuzzy Logic in Motor Control", *Fuzzy Logic '93 Proceedings*, pp. A111-1 - A111-10.

[20] D. Nguyen and B. Widrow, "Neural Networks for Self Learning Control Systems", *IEEE Control Systems Magazine*, (April, 1990), pp. 18 - 23.

[21] P. Pacini and B. Kosko, "Comparison of Fuzzy and Kalman-Filter Target-Tracking Control Systems" in *Neural Networks and Fuzzy Systems*. Englewood Cliffs: Prentice Hall Inc., 1992.

[22] W. Pedrycz, "Fuzzy Sets and Neurocomputations: Knowledge Representation and Processing in Intellingent Controllers", *Fifth IEEE International Symposium on Intelligent Control*, 1990, pp. 626 - 630.

[23] D. Psaltis, A. Sideris, and A. Yamamura, "A Multilayered Neural Network Controller", *IEEE Control Systems Magazine*, (April, 1988), pp. 17-21.

[24] S. Rahman, "Neural-Fuzzy Consumer Appliance Applications", *Proceedings of Fuzzy Logic '93*, pp. M234-1 - M234-7.

[25] M. Sadiku and M. Mazzara. "Computing with Neural Networks" *IEEE Potentials*, (October, 1993), pp. 14-16.

[26] D. G. Schwartz and G. J. Klir. "Fuzzy Logic Flowers in Japan" *IEEE Spectrum* (July, 1992).

[27] L. A. Zadeh. "Fuzzy Sets" *Information and Control,* (June, 1965), pp. 338 - 353.

[28] L. A. Zadeh. "Outline of a New Approach to the Analysis of Complex Systems and Decision Processes" *IEEE Transactions on Systems, Man, and Cybernetics* (January 1973), pp. 28 - 44.

[29] O. Iida, M. Sato, M. Kiguchi, T. Iwamura, S. Fukumura. "Blast Furnace Control by Artificial Intelligence", *Artificial Intelligence in Real-time Control* (ed. M. G. Rodd and G. J. Suski) Swansea, UK: Pergamon Press, 1988, p. 73.

[30] W. S. McCulloch and W. Pitts, "A Logical Calculus of the Ideas Imminent in Nervous Activity," *Bulletin of Mathematical Biophysics,* Vol 5, 1943.

[31] V. J. Lumelsky, "Effect of Kinematics on Motion Planning for Planar Robot Arms Moving Amidst Unknown Obstacles," *Journal of Robotics and Automation* Vol. RA-3, (June, 1987), pp. 207-222.

[32] M. Galicki, "Optimal Planning of a Collision-free Trajectory of Redundant Manipulators," *The International Journal of Robotics Research,* Vol 11, No. 6, (December, 1992), pp. 549-559.

[33] W. J. Chung, W. K. Chung, and Y. Youm, "Inverse Kinematics of Planar Redundant Manipulators Using Virtual Links and Displacement Distribution Schemes," *Proceedings of the 1991 IEEE International Conference on Robotics and Automation,* April 1991, pp. 926 - 932.

[34] K. P. Archer and S. Wang, "Fuzzy Set Representation of Neural Network Classification Boundaries", *IEEE Transactions on Systems, Man, and Cybernetics, (*July/August, 1991), pp. 735-742.

[35] S. Shao, "Fuzzy Self-organzing Controller and Its Application for Dynamic Processes", *Fuzzy Sets and Systems,* (May, 1988), pp. 151-164.

[36] A. Blanco and M. Delgado, "A Direct Fuzzy Inference Procedure By Neural Networks", *Fuzzy Sets and Systems,* (September 1993), pp. 133-141.

[37] J. M. Keller and D. J. Hunt, "Incorporating Fuzzy Membership Functions into the Perceptron Algorithm", *IEEE Transactions on Pattern Analysis and Machine Intelligence,* (November, 1985), pp. 693-699.

[38] S. C. Lee, and E. T. Lee, "Fuzzy sets and neural networks" *Journal of Cybernetics Vol 4.* pp. 83 -103, 1974.

[39] W. Pedrycz, "Fuzzy Neural Networks and Neurocomputations", *Fuzzy Sets and Systems,* Vol. 56, (May 1993), pp. 1-28.

[40] S. K. Pal and S. Mitra, "Multilayer Perceptron, Fuzzy Sets, and Classification", *IEEE Transactions on Neural Networks, Vol 3.* (September 1992), pp. 683-697.

[41] Berenji, Hamid R. and Khedkar, Pratap. "Learning and Tuning Fuzzy Logic Controllers Through Reinforcements" *IEEE Transactions on Neural Networks Vol. 3.* pp. 724 - 740, 1992.

[42] W. Z. Qiao, W. P. Zhuang, T. H. Heng, S. S. Shan, "A Rule Self-Regulating Fuzzy Controller" *Fuzzy Sets and Systems,* pp. 13-21, 1992.

[43] J. M. Keller, R. R. Yager, and H. Tahani, "Neural Network Implementation of Fuzzy Logic" *Fuzzy Sets and Systems* (Vol 45), pp. 1-12, 1992.

[44] J. M. Keller, H. Tahani, "Backpropagation Neural Networks for Fuzzy Logic" *Information Sciences* Vol 62, pp. 205-221, 1992.

[45] J. H. Chuang and N. Ahuja, "Path Planning Using the Newtonian Potential", *Proceedings of the 1991 IEEE International Conference on Robotics and Automation* (April 1991), pp. 558 - 563.

[46] D. Wang and Y. Hamam, "Optimal Trajectory Planning of Manipulators With Collision Detection and Avoidance", *The International Journal of Robotics Research, Vol 11, No. 5* (October 1992), pp. 460 - 468.

[47] C. J. J. Paredis and P. K. Khosla, "On Kinematic Design of Serial Link Manipulators", *Proceedings of the 30th IEEE Conference on Decision and Control*, (1991), pp. 517 - 531.

[48] A. Bagchi and H. Hatwal, "A Solution Strategy for Collsion Avoidance of Multiple Bodies Moving on a Plane Using Fuzzy Logic", *Proceedings of the 29th IEEE Conference on Decision and Control*, (1990), pp. 452 - 461

[49] C. H. Lin and L. C. Fu, "Motion Planning of Robot Manipulators with Arbitrary Number of DOF", *Proceedings of the 30th IEEE Conference on Decision and Control,* (1990), pp. 359 - 370.

[50] R. T. Newton and Y. Xu, "Real-time Implementation of Neural Network Learning Control of a Flexible Space Manipulator", *Proceedings of the IEEE International Conference on Robotics and Automation, Vol. 1,* (May 1993), pp. 135 - 141.

[51] F. Arai, L. Rong, and T. Fukuda, "Trajectory Control of Flexible Plate Using Neural Network", *Proceedings of the IEEE International Conference on Robotics and Automation, Vol 1* (May 1993), pp. 155 - 160.

[52] S. Liu and H. Asada, "Teaching and Learning of Deburring Robots Using Neural Networks", *Proceedings of the IEEE International Conference on Robotics and Automation, Vol 3* (May 1993), pp. 339 - 345.

[53] T. C. Hsia and Z. Mao, "Obstacle Avoidance Inverse Kinematics Solution of Redundant Manipulators by Neural Networks", *Proceedings of the IEEE*

International Conference on Robotics and Automation Vol. 3 (May 1993), p. 1014.

[54] J. R. Cooperstock and E. E. Milios, "An Efficiently Trainable Neural Network Based Vision-Guided Robot Arm", *Proceedings of the IEEE International Conference on Robotics and Automation Vol. 2* (May 1993), pp. 738 - 743.

[55] J. D. Yegerlehner and P. H. Meckl, "Experimental Implementation of Neural Network Controller for Robot Undergoing Large Payload Changes", *Proceedings of the IEEE International Conference on Robotics and Automation, Vol. 2.* (May 1993), pp. 744 - 749.

[56] C. C. Jou and N. C. Wang, "Training a Fuzzy Controller to Back Up an Autonomous Vehicle", *Proceedings of the IEEE International Conference on Robotics and Automation Vol. 1,* (May 1993), pp. 923 - 928.

APPENDIX ONE

All the code for this thesis project is written in C and is the original work of the author. The code can be divided into three groups -- arm simulation, neuro-fuzzy engine, and neuro-fuzzy controller.

ARM SIMULATION

```
/***********************************************************************
********
*  FARM.H
*  the structure definitions and function definitions for
*  the arm simulation under neuro-fuzzy control
*  by Wallace Kelly
*  February 13, 1994
*  revision on 6/14/94 for minor changes
***********************************************************************
*******/
#define OFF 0
#define ON 1
#define NO 0
#define YES 1
#define READ 0
#define WRITE 1
#define APPEND 2
#define FAILURE 0
#define SUCCESS 1
#define CLIPOFF 0
#define CLIPON 1
#define MAXFILENAME 64
#define MAXLEN 640
#define pi2 6.283185314

#define strequ !strcmp

struct Link {           /* A link extending from a revolute joint */
  double len;         /* Length of link in pixels */
  double theta;         /* Angle relative to previous link, in radians
*/
  double max_theta;     /* Maximum theta caused by joint constraint */
  double min_theta;     /* Minumum theta caused by joint constraint */
  double phi;         /* Angle relative to horizontal */
  struct Link *next;    /* Pointer to next link in linked list */
  };

struct Point {        /* A structure for polar coordinates */
  double rho;         /* the distance from the origin */
  double phi;         /* the angle from the x axis to the radius vector
*/
```

```
  };

/**********************************************************************
*******
*  Initialize the link data structure for an arm
**********************************************************************
******/
struct Link *IniLinks(void);

/**********************************************************************
*******
*   Add a link to a link struct
**********************************************************************
******/
int AddLink(struct Link *links, double len, double theta, double
max_theta, double min_theta);

/**********************************************************************
*******
*  Set the 'linknum' joint to 'theta' degrees
**********************************************************************
******/
int SetLink(struct Link *links, int linknum, double theta_norm);

/**********************************************************************
*******
*  Move the 'linknum' joint by 'd_theta' degrees
**********************************************************************
******/
int MoveLink(struct Link *links, int linknum, double d_theta);

/**********************************************************************
*******
*  Draw the arm
**********************************************************************
******/
void DrawArm(struct Link *links, int color);

/**********************************************************************
*******
*  Calculates the Cartesian coordinates of the end-effector
**********************************************************************
******/
void EndEffector (struct Link *links, struct complex *p);

/**********************************************************************
*******
*  Determine if the arm is intersecting with an object
**********************************************************************
******/
int Collision(struct Link *links, struct Point *obstacle, double
margin);

/**********************************************************************
*******
*  Draw a circle to simulate an obstacle or goal
```

```
************************************************************************
******/
void Circle(double x, double y, double radius, int color);

/***********************************************************************
*******
*  The function, radius and color for a goal
************************************************************************
******/
void DrawGoal(struct Point *g);

/***********************************************************************
*******
*  The function, radius and color for an obstacle
************************************************************************
******/
void DrawObstacle(struct Point *p);

/***********************************************************************
********
*  Generate a random number between min and max
************************************************************************
*******/
double Random(double min, double max);

/***********************************************************************
*******
* Returns the distance between two points
************************************************************************
******/
double Distance(struct complex *p1, struct complex *p2);

/***********************************************************************
********
*  Return the distance between to polar points
************************************************************************
*******/
double PDistance(struct Point *p1, struct Point *p2);

/***********************************************************************
********
*  Determine the rectangular equivalent of a polar coordinate
************************************************************************
*******/
void Pol2Rec(struct Point *p, struct complex *r);

/***********************************************************************
*******
* Copy the current state into an array
************************************************************************
******/
void GetState(double *state, struct Link *links, struct Point *goal,
struct Point *obstacle);


/***********************************************************************
********
```

```
*  FNN.H
*  the structure definitions and function definitions for
*  a basic backpropagation fuzzy neural network
*  by Wallace Kelly
*  February 9, 1994
*  adapted for fuzzy 2/22/94
*  revision on 5/17/94 for flexibility
*  revision on 6/14/94 for minor changes
***********************************************************************
*******/
#define OFF 0
#define ON  1
#define NO 0
#define YES 1
#define READ 0
#define WRITE 1
#define APPEND 2
#define FAILURE 0
#define SUCCESS 1
#define CLIPOFF 0
#define CLIPON 1
#define MAXFILENAME 64
#define pi2 6.283185314

#define strequ !strcmp

struct Node {              /* a neural network node */
  int numweights;          /* number of inputs to the node */
  double *win;             /* current weights array */
  double *winlast;       /* last weights array */
  double sum;            /* sum of inputs*weights */
  double output;         /* activation output */
  double error;            /* BPN error at this node */
  };

struct Layer {           /* an in, out, or hid layer */
  int numnodes;            /* number of nodes in this layer */
  struct Node *nodes;    /* ptr to next node in this layer */
  };

struct Net {               /* a neural network */
  int numlayers;         /* the number of layers in this NN */
  struct Layer *layers;    /* ptr to next layer in this NN */
  };

struct NetworkInfo {       /* all the info to start a net */
  int numinputs;
  int numhidlayers;
  int numoutputs;
  int *numhidnodes;
  int *numnodes;
  int numlayers;
  double alpha;
  double beta;
  double gamma;
  char *input;
  char *samples;
  char *desired;
  char *weights;
  char *error;
  int *input_values;
  int *output_values;
  double **input_max;
  double **output_max;
  unsigned epochs;
  double maxerror;  };

/**********************************************************************
********
*  malloc function with the advantage of error catching
**********************************************************************
*******/
void *MemAlloc(size_t size);

/**********************************************************************
********
*  free function with benefit of nulling pointers
**********************************************************************
*******/
void Free(void *ptr);

/**********************************************************************
********
*  Initialize the nodes in a layer
**********************************************************************
*******/
struct Node *IniNodes(int nodes_prev, int numnodes);

/**********************************************************************
********
*  Initialize the layers in a network
**********************************************************************
*******/
struct Layer *IniLayers(int numlayers, int *numnodes);

/**********************************************************************
********
*  Initialize a neural network
**********************************************************************
*******/
struct Net *IniNet(int numlayers, int *numnodes);

/**********************************************************************
********
*  Input a string, and attach a default extension if necessary
**********************************************************************
*******/
char *GetFileName(const char *string);

/**********************************************************************
********
*  fopen function with advantage of error catching, etc
**********************************************************************
*******/
```

```
FILE *Fopen(char *filename, int m);

/*********************************************************************
********
*  Store the neural network weights in a file
*********************************************************************
*******/
void StoreWeights(char *filename, struct Net *net);

/*********************************************************************
********
*  Load the neural network weights from a file
*********************************************************************
*******/
void LoadWeights(char *filename, struct Net *net);

/*********************************************************************
********
*  Input the values in an array into the input layer's output
*********************************************************************
*******/
void InputArray(double *array, struct Net *net);

/*********************************************************************
********
*  Perform the forward propagation
*********************************************************************
*******/
void RunNN(struct Net *net);

/*********************************************************************
********
*  Perform the backpropagation
*********************************************************************
*******/
void BackPropagate(double *desired, struct Net *net);

/*********************************************************************
********
*  Calculate the root mean square error of all the inputs
*********************************************************************
*******/
double ErrorNN(double *inputs, double *desired, struct Net *net, struct
NetworkInfo *info);

/*********************************************************************
********
*  Print the values of the outputs of the last neurons
*********************************************************************
*******/
void ShowOutput(struct Net *net, struct NetworkInfo *info);

/*********************************************************************
********
*  Set the GLOBAL constants - alpha, beta, gamma
*********************************************************************
*******/
```

```
void BPN_Constants(double a, double b, double g);

/**********************************************************************
********
*  Get the network information from a file
***********************************************************************
*******/
struct NetworkInfo *GetNetInfo(FILE *fp);

/**********************************************************************
********
*  Load an array of size numbers
***********************************************************************
*******/
int DoubleIn(FILE *fp, double *a, int size);

/**********************************************************************
********
* Fuzzify the values of an array
***********************************************************************
*******/
void Fuzzify(double *in, double *fuzzified, double **max_points);

/**********************************************************************
********
* Defuzzify the values of an array
***********************************************************************
*******/
void Defuzzify(double *f, double *out, double **max_points);

/**********************************************************************
*******
* Graphics function
***********************************************************************
******/
int InitializeScreen(void);

/**********************************************************************
********
* Return the output of the network
***********************************************************************
*******/
void ReadOutput(struct Net *net, struct NetworkInfo *info, double
*out);

/**********************************************************************
*******
*  ARM.C
*  An include file for simulating a 2-D revolute robotic arm with
arbitrary
*  number of links
*  by Wallace Kelly
*  January 20, 1994
***********************************************************************
******/
#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>
#include <graphics.h>

#include "fnn.h"
#include "farm.h"

/**********************************************************************
*******
*  Initialize the link data structure for an arm
**********************************************************************
******/
struct Link *IniLinks(void)
{
  struct Link *links;

  links = MemAlloc(sizeof(struct Link));
  if(links == NULL) return NULL;

  links->len = 0;       /* Necessary for the AddLink function */
  links->next = NULL;
  return links;
}

/**********************************************************************
*******
*   Add a link to a link struct
**********************************************************************
******/
int AddLink(struct Link *links, double len, double theta,
         double max_theta, double min_theta)
{
  struct Link *l;
  double phi = 0;
  l = links;

  if(len == 0) {
    printf("\nError: Link lengths of zero not permitted.\n");
    return FAILURE;  }

  /* This section for arms with links already existing */
  if(l->len != 0) {
    while (l->next != NULL)
      l = l->next;  /* Advance pointer to last existing link */

    phi = l->phi;    /* Stored to calculate new link's phi */
    l->next = MemAlloc(sizeof(struct Link));
    if(l->next == NULL) return FAILURE;

    l = l->next; }

  /* Insert link parameters in to the data structure */
  l->len = len;
  l->theta = theta;
  l->phi = fmod( phi + theta, pi2);
  l->max_theta = max_theta;
  l->min_theta = min_theta;
  l->next = NULL;
  return SUCCESS;
};

/**********************************************************************
*******
*  Set the 'linknum' joint to 'theta' degrees
***********************************************************************
******/
int SetLink(struct Link *links, int linknum, double theta_norm)
{
  int loop;
  double temp = 0, theta;
  struct Link *l;
  l = links;

  /* Advance to the 'linknum' joint and sum heighths */
  for(loop = 0; loop < linknum; loop = loop + 1) {
    temp = l->phi;
    if(l->next != NULL) l = l->next;  }

  theta = theta_norm * (l->max_theta - l->min_theta) + l->min_theta;
  if(theta > l->max_theta) theta = l->max_theta;
  if(theta < l->min_theta) theta = l->min_theta;

  l->theta = theta;

  for(; l!=NULL; l=l->next) {  /* Recalculate phi's all the way to the
tip */
     l->phi = fmod( temp + l->theta, pi2);
     temp = l->phi; }

  return SUCCESS;
}

/**********************************************************************
*******
*  Move the 'linknum' joint by 'd_theta' degrees
***********************************************************************
******/
int MoveLink(struct Link *links, int linknum, double d_theta)
{
  int loop;
  double y = 0, angle = 0; /* New height and angle for the joints */
  struct Link *ltemp, *l;
  l = links;

  /* Advance to the 'linknum' joint and sum heighths */
  for(loop = 0; loop < linknum; loop = loop + 1) {
    y = y + l->len * sin( l->phi );
    if(l->next != NULL) l = l->next;  }

  /* Test a Cartesian constraint that each joint's y > 0 */
  ltemp = l;
  for(; ltemp!=NULL; ltemp = ltemp->next)  {
    y = y + ltemp->len * sin( ltemp->phi + d_theta );
    if(y < 0.0) return FAILURE;  }
  /* Test angle constraint of the joint to be moved */
  ltemp = l;
  angle = fmod( ltemp->theta + d_theta, pi2);
  if(angle > ltemp->max_theta) d_theta = ltemp->max_theta - ltemp-
>theta;
  if(angle < ltemp->min_theta) d_theta = ltemp->min_theta - ltemp-
>theta;


  /* Physical limits won't be exceeded, OK, make changes to data */
  l->theta = fmod( l->theta + d_theta, pi2);
  for(; l!=NULL; l=l->next)  /* Recalculate phi's all the way to the
tip */
    l->phi = fmod( l->phi + d_theta, pi2);

  return SUCCESS;
}

/***********************************************************************
*******
*  Move to a location taking into account the aspect ratio and Y axis
***********************************************************************
******/
void MoveTo(int x, int y)
{
  moveto(x*4/3, -y);
}

/***********************************************************************
*******
*  Draw to a location taking into account the aspect ratio and Y axis
***********************************************************************
******/
void LineTo(int x, int y)
{
  lineto(x*4/3, -y);
}

/***********************************************************************
*******
*  Draw the arm
***********************************************************************
******/
void DrawArm(struct Link *links, int color)
{
  int x=0, y=0;  /* Assumes base is at (0, 0) */
  struct Link *l;

  setcolor(color);
  MoveTo(x, y);
  for(l = links; l!=NULL; l = l->next) {
    x += l->len * cos(l->phi);
    y += l->len * sin(l->phi);
    LineTo(x, y);
    }
}
```

```
/**********************************************************************
*******
*  Calculates the Cartesian coordinates of the end-effector
**********************************************************************
******/
void EndEffector (struct Link *links, struct complex *p)
{
  struct Link *l;

  if(!p) {printf("ERROR in programming EndEffector()");  exit(1); }

  p->x = 0;
  p->y = 0;

  for(l = links; l!=NULL; l = l->next)  {
    p->x = p->x + l->len * cos (l->phi);
    p->y = p->y + l->len * sin (l->phi); }

  return;
}

/**********************************************************************
*******
*  Determine if the arm is intersecting with an object
**********************************************************************
******/
int Collision(struct Link *links, struct Point *obstacle, double
margin)
{
  double d, d12, d10, d02, d12a, d12b, t0, t1, t2;
  struct Link *l;
  struct complex p0, p1, p2;
  struct Point p;

  Pol2Rec(obstacle, &p0);
  p1.x = 0;
  p1.y = 0;

  for(l = links; l!=NULL; l = l->next)  {
    p.rho = l->len;
    p.phi = l->phi;
    Pol2Rec(&p, &p2);

    p2.x += p1.x;
    p2.y += p1.y;

    d12 = Distance(&p1, &p2);
    d10 = Distance(&p1, &p0);
    d02 = Distance(&p0, &p2);

    t1 = acos( (d12 * d12 + d10 * d10 - d02 * d02) / (2 * d12 * d10));
    t2 = acos( (d12 * d12 + d02 * d02 - d10 * d10) / (2 * d12 * d02));

    d = d10 * sin(t1);      /* distance from obstacle's center to the
link */
    if(t1 > M_PI_2) d = d10;
```

```
     if(t2 > M_PI_2) d = d02;

     if(d < margin) return ON;

     p1.x = p2.x; p1.y = p2.y;  }

     return OFF;
}

/**********************************************************************
*******
*  Copy the current state into an array
**********************************************************************
******/
void GetState(double *state, struct Link *links, struct Point *goal,
struct Point *obstacle)
{
  int index = 1;
  double x1, y1, x2, y2;
  struct Link *l;
  struct complex end;
  struct complex g, o;

  Pol2Rec(goal, &g);
  Pol2Rec(obstacle, &o);

  for(l = links; l!=NULL; l=l->next)  {
     state[index] = (l->theta - l->min_theta)/(l->max_theta - l-
>min_theta);
     index++;  }

  EndEffector(links, &end);
  x1 = (g.x - end.x) / (2.0 * MAXLEN) + 0.5;
  state[index] = x1;
  index++;

  y1 = (g.y - end.y) / MAXLEN + 0.5;
  state[index] = y1;
  index++;

  x2 = (o.x - end.x) / (2.0 * MAXLEN) + 0.5;
  state[index] = x2;
  index++;

  y2 = (o.y - end.y) / MAXLEN + 0.5;
  state[index] = y2;
  index++;

  return;
}

/**********************************************************************
********
*  OBJECTS.C
*  source code for the objects -- both the goal and obstacle
*  by Wallace E. Kelly, III
*  revision on June 16, 1994
```

```
**********************************************************************
*******/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <graphics.h>

#include "fnn.h"
#include "farm.h"

/*********************************************************************
*******
*  Draw a circle to simulate an obstacle or goal
**********************************************************************
******/
void Circle(double x, double y, double radius, int color)
{
  setcolor(color);
  circle(x * 4 / 3, - y, radius);
}

/*********************************************************************
*******
*  The function, radius and color for a goal
**********************************************************************
******/
void DrawGoal(struct Point *g)
{
  int radius = 10;
  int color = EGA_LIGHTGREEN;
  double x, y;

  x = cos(g->phi) * g->rho;
  y = sin(g->phi) * g->rho;

  Circle(x, y, radius, color);
  setfillstyle(SOLID_FILL, color);
  floodfill(x * 4 / 3, -y, color);
}

/*********************************************************************
*******
*  The function, radius and color for an obstacle
**********************************************************************
******/
void DrawObstacle(struct Point *o)
{
  int radius = 10;
  int color = EGA_LIGHTRED;
  double x, y;

  x = cos(o->phi) * o->rho;
  y = sin(o->phi) * o->rho;

  Circle(x, y, radius, color);
}
```

```
/**********************************************************************
********
*  Generate a random number between min and max
***********************************************************************
*******/
double Random(double min, double max)
{
  double n;
  n = ((double)(rand()) / (double)RAND_MAX) * (max - min) + min;
  return n;
}

/**********************************************************************
*******
* Returns the distance between two points
***********************************************************************
******/
double Distance(struct complex *p1, struct complex *p2)
{
  double d;

  d = (double)( (double)(p1->x - p2->x) * (double)(p1->x - p2->x)
        + (double)(p1->y - p2->y) * (double)(p1->y - p2->y));
  d = sqrt(d);
  return d;
}

/**********************************************************************
********
*  Return the distance between to polar points
***********************************************************************
*******/
double PDistance(struct Point *p1, struct Point *p2)
{
  struct complex rp1, rp2;

  rp1.x = p1->rho * cos(p1->phi);
  rp1.y = p1->rho * sin(p1->phi);
  rp2.x = p2->rho * cos(p2->phi);
  rp2.y = p2->rho * sin(p2->phi);

  return Distance(&rp1, &rp2);
}

/**********************************************************************
********
*  Determine the rectangular equivalent of a polar coordinate
***********************************************************************
*******/
void Pol2Rec(struct Point *p, struct complex *r)
{
  if(!p || !r) {printf("ERROR in programming Pol2Rec."); exit(1); }

  r->x = p->rho * cos(p->phi);
  r->y = p->rho * sin(p->phi);
}
```

NEURO-FUZZY ENGINE

```
/**********************************************************************
********
*  NN1.C
*  the basic initialization routines for a BPN
*  by Wallace Kelly
*  February 9, 1994
**********************************************************************
*******/
#include <stdio.h>
#include <stdlib.h>

#include "fnn.h"

/**********************************************************************
********
*  Initialize the nodes in a layer
**********************************************************************
*******/
struct Node *IniNodes(int nodes_prev, int numnodes)
{
  int loop, loop2;
  struct Node *nodes = NULL;

  nodes = MemAlloc(sizeof(struct Node) * numnodes);

  for(loop = 0; loop<numnodes; loop++)  {
    nodes[loop].win = MemAlloc(sizeof(double) * nodes_prev);
    nodes[loop].winlast = MemAlloc(sizeof(double) * nodes_prev);
    for(loop2 = 0; loop2 < nodes_prev; loop2++) {
      nodes[loop].win[loop2] = 0;
      nodes[loop].winlast[loop2] = 0; }
    nodes[loop].numweights = nodes_prev;
    nodes[loop].sum = 0;
    nodes[loop].output = 0;
    nodes[loop].error = 0;  }

  return nodes;
}

/**********************************************************************
********
*  Initialize the layers in a network
**********************************************************************
*******/
struct Layer *IniLayers(int numlayers, int *numnodes)
{
  int loop, prev;
  struct Layer *layers = NULL;

  layers = MemAlloc(sizeof(struct Layer) * numlayers);

  for(loop = 0; loop < numlayers; loop++)  {
    layers[loop].numnodes = numnodes[loop];
    prev = numnodes[loop-1];
```

```
     if(loop == 0) prev = 0;
     layers[loop].nodes = IniNodes(prev, numnodes[loop]);  }

  return layers;
}

/***********************************************************************
********
*  Initialize a neural network
************************************************************************
*******/
struct Net *IniNet(int numlayers, int *numnodes)
{
  struct Net *net = NULL;

  net = MemAlloc(sizeof (struct Net));

  net->numlayers = numlayers;
  net->layers = IniLayers(numlayers, numnodes);
  return net;
}

/***********************************************************************
********
*  Input the values in an array into the input layer's output
************************************************************************
*******/
void InputArray(double *array, struct Net *net)
{
  int loop, numnodes;
  struct Node *node = NULL;

  numnodes = net->layers[0].numnodes;
  if(array[0] != numnodes) {
     printf("\nThe input array is the wrong dimension for this neural
network.\n");
     exit(1);  }

  node = net->layers[0].nodes;
  for(loop = 0; loop < numnodes; loop++)
     node[loop].output = array[loop+1];
}

/***********************************************************************
********
*  FNN_IO.C
*  Input/output functions
*  by Wallace Kelly
*  February 9, 1994
*  adapted for fuzzy on 2/22/94
*  revision on 5/17/94 for more flexibility
*  revision on 6/14/94 for minor changes
************************************************************************
*******/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <math.h>
#include <alloc.h>
#include <graphics.h>

#include "fnn.h"

/*********************************************************************
********
*  Input a string, and attach a default extension if necessary
**********************************************************************
*******/
char *GetFileName(const char *string)
{
  int done=YES;
  char *filename = NULL, dot = '.', space = ' ', *ext = ".DAT";

  do {
    printf("\nEnter the name of the file %s: ", string);
    filename = MemAlloc(sizeof(char)*MAXFILENAME);
    strupr(gets(filename));

    if(strlen(filename) == 0) {
      Free(filename);
      return NULL; }

    if(strchr(filename, space)) {
      done = NO;
      printf("\nInvalid filename.\n");
      Free(filename);
      } } while(!done);

  if(!strchr(filename, dot)) strcat(filename, ext);
  return filename;
}

/*********************************************************************
********
*  fopen function with advantage of error catching, etc
**********************************************************************
*******/
FILE *Fopen(char *filename, int m)
{
  char mode;
  FILE *fp = NULL;

  /* Determine the mode code */
  switch(m) {
    case READ:  mode = 'r'; break;
    case WRITE: mode = 'w'; break;
    case APPEND:  mode = 'a'; break;
    default:  printf("\nError in call to Fopen().\n");
          exit(1);  };

  fp = fopen(filename, &mode);
  if(!fp) {
    printf("\nError opening a file: %s\n", filename);
    exit(1);  }
  return fp;
}

/*********************************************************************
********
*  Print the values of the outputs of the last neurons
*********************************************************************
*******/
void ShowOutput(struct Net *net, struct NetworkInfo *info)
{
  int loop, numnodes;
  double *fuzzy, *defuzzified;
  struct Layer *layer = NULL;
  struct Node *node = NULL;

  layer =  net->layers;
  loop = net->numlayers - 1;
  numnodes = layer[loop].numnodes;

  fuzzy = MemAlloc(sizeof(double) * (numnodes+1));
  fuzzy[0] = numnodes;

  defuzzified = MemAlloc(sizeof(double) * (info->numoutputs+1));
  defuzzified[0] = info->numoutputs;

  node = layer[loop].nodes;

  for(loop = 1; loop <= numnodes; loop++)
    fuzzy[loop] = node[loop-1].output;

  Defuzzify(fuzzy, defuzzified, info->output_max);
  for(loop = 1; loop <= defuzzified[0]; loop++)
    printf("%0.8lf ", defuzzified[loop]);
  printf("\n");
}

/*********************************************************************
********
*  Store the neural network weights in a file
*********************************************************************
*******/
void StoreWeights(char *filename, struct Net *net)
{
  int loop, loop2, loop3;
  struct Node *node = NULL;
  struct Layer *layer = NULL;
  FILE *fp = NULL;

  if(filename != NULL) {
    fp = Fopen(filename, WRITE);

    for(loop = 0; loop < net->numlayers; loop++)  {
      layer =  net->layers;
      for(loop2 = 0; loop2 < layer[loop].numnodes; loop2++)  {
        node = layer[loop].nodes;
        for(loop3 = 0; loop3 < node[loop2].numweights; loop3++)
```

```
            fprintf(fp, "%+0.8lf %+0.8lf\n", node[loop2].win[loop3],
node[loop2].winlast[loop3]);  } }

  fclose(fp);  }
}

/***********************************************************************
********
*  Load the neural network weights from a file
************************************************************************
*******/
void LoadWeights(char *filename, struct Net *net)
{
  int loop, loop2, loop3;
  struct Node *node = NULL;
  struct Layer *layer = NULL;
  FILE *fp = NULL;

  fp = Fopen(filename, READ);

  for(loop = 1; loop < net->numlayers; loop++)  {
    layer =  net->layers;
    for(loop2 = 0; loop2 < layer[loop].numnodes; loop2++)  {
      node = layer[loop].nodes;
      for(loop3 = 0; loop3 < node[loop2].numweights; loop3++)
        if(  fscanf(fp, "%lf %lf", &node[loop2].win[loop3],
&node[loop2].winlast[loop3] ) == 0) {
            printf("\nError loading weights from file %s", filename);
            exit(1);  } } }
  fclose(fp);
}

/***********************************************************************
********
*  Read an array of size numbers
************************************************************************
*******/
int DoubleIn(FILE *fp, double *a, int size)
{
  int flag = 1, loop;

  if(feof(fp)) flag = 0;

  a[0] = size;
  for(loop = 1; loop <= size; loop++)
    flag = (fscanf(fp, "%lf\n", &a[loop]) && (flag));

  if(!flag) rewind(fp);
  return flag;
}

/***********************************************************************
********
*  Read the network information from the network info file
************************************************************************
*******/
struct NetworkInfo *GetNetInfo(FILE *fp)
```

```
{
  int flag = 0, flag1 = 0, flag2 = 0, loop, loop2;
  char buf[255], *equal = "=", *space = " ", *cr = "\n", *value,
*value2;
  struct NetworkInfo *info;

  char *numinputs = "inputs";
  char *numoutputs = "outputs";
  char *numhidlayers = "hidden";
  char *numhidnodes = "hidnodes";
  char *alpha = "alpha";
  char *beta = "beta";
  char *gamma = "gamma";
  char *input = "input_file";
  char *samples = "samples_file";
  char *desired = "desired_file";
  char *weights = "weights_file";
  char *error = "error_file";
  char *epochs = "epochs";
  char *maxerror = "maxerror";
  char *input_values = "input_values";
  char *output_values = "output_values";
  char *input_max = "input_max";
  char *output_max = "output_max";


  info = MemAlloc(sizeof(struct NetworkInfo));

  while(fgets(buf, 255, fp) != NULL) {
    strlwr(buf);
    strtok(buf, equal);
    value = strtok(NULL, cr);

    if(strequ(buf, numinputs))  {
      info->numinputs = atoi(value);
      flag1 = 1;  }

    if(strequ(buf, numoutputs))  {
      info->numoutputs = atoi(value);
      flag2 = 1;  }

    if(strequ(buf, numhidlayers)) {
      info->numhidlayers = atoi(value);
      flag = 1;  }

    if(strequ(buf, numhidnodes)) {
      if(flag == 0) {
        printf("\nERROR: The 'numhidnodes='line should come after
'numhidlayers='\n");
        exit(1);  }
      info->numhidnodes = MemAlloc(sizeof(int)* info->numhidlayers);
      for(loop = 0; loop < info->numhidlayers; loop++)  {
        value2 = strtok(value, space);
        info->numhidnodes[loop] = atoi(value2);
        value = NULL;  }  }

    if(strequ(buf, alpha)) info->alpha = atoi(value);
```

```
     if(strequ(buf, beta)) info->beta = atof(value);

     if(strequ(buf, gamma)) info->gamma = atof(value);

     if(strequ(buf, input))  {
       info->input = MemAlloc(sizeof(char) * (strlen(value)+1));
       strcpy(info->input, value);  }

     if(strequ(buf, samples)) {
       info->samples = MemAlloc(sizeof(char) * (strlen(value) + 1));
       strcpy(info->samples, value);  }

     if(strequ(buf, desired))  {
       info->desired = MemAlloc(sizeof(char) * (strlen(value)+1));
       strcpy(info->desired, value);  }

     if(strequ(buf, weights)) {
       info->weights = MemAlloc(sizeof(char) * (strlen(value)+1));
       strcpy(info->weights, value);  }

     if(strequ(buf, error)) {
       info->error = MemAlloc(sizeof(char) * (strlen(value)+1));
       strcpy(info->error, value);  }

     if(strequ(buf, epochs)) info->epochs = atoi(value);

     if(strequ(buf, maxerror)) info->maxerror = atof(value);

     if(strequ(buf, input_values)) {
       if(!flag1) {
         printf("MEMBERSHIP ERROR: The inputs= must precede
inputs_values=");
         exit(1);  }
       flag1=2;
       info->input_values = MemAlloc(sizeof(int)* (info->numinputs+1));
       info->input_values[0] = info->numinputs;
       for(loop = 1; loop <= info->numinputs; loop++)  {
         value2 = strtok(value, space);
         info->input_values[loop] = atoi(value2);
         value = NULL;  }  }

     if(strequ(buf, output_values)) {
       if(!flag2) {
         printf("MEMBERSHIP ERROR: The inputs= must precede
inputs_values=");
         exit(1);  }
       flag2=2;
       info->output_values = MemAlloc(sizeof(int)* (info-
>numoutputs+1));
       info->output_values[0] = info->numoutputs;
       for(loop = 1; loop <= info->numoutputs; loop++)  {
         value2 = strtok(value, space);
         info->output_values[loop] = atoi(value2);
         value = NULL;  }  }

     if(strequ(buf, input_max)) {
       if(flag1!=2) {
         printf("MEMBERSHIP ERROR: The input_values= must precede
input_max=");
         exit(1);  }
       info->input_max = MemAlloc(sizeof(double*) * info->numinputs);
       for(loop2 = 0; loop2 < info->numinputs; loop2++)  {
         fgets(buf, 255, fp);
         strlwr(buf);
         value = buf;
         info->input_max[loop2] =
MemAlloc(sizeof(double)*(input_values[0]+1));
         info->input_max[loop2][0] = info->input_values[loop2+1];
         for(loop = 1; loop <= info->input_values[loop2+1]; loop++)  {
           value2 = strtok(value, space);
           info->input_max[loop2][loop] = atof(value2);
           value = NULL;  }  }  }

     if(strequ(buf, output_max)) {
       if(flag2!=2) {
         printf("MEMBERSHIP ERROR: The output_values= must precede
output_max=");
         exit(1);  }
       info->output_max = MemAlloc(sizeof(double*) * info->numoutputs);
       for(loop2 = 0; loop2 < info->numoutputs; loop2++)  {
         fgets(buf, 255, fp);
         strlwr(buf);
         value = buf;
         info->output_max[loop2] =
MemAlloc(sizeof(double)*(output_values[0]+1));
         info->output_max[loop2][0] = info->output_values[loop2+1];
         for(loop = 1; loop<=info->output_values[loop2+1]; loop++)  {
           value2 = strtok(value, space);
           info->output_max[loop2][loop] = atof(value2);
           value = NULL; }  }  }}

   /* calculate number of layers and nodes */
   info->numlayers = info->numhidlayers+2;
   info->numnodes = MemAlloc(sizeof(int) * info->numlayers);

   /* count input nodes */
   info->numnodes[0] = 0;
   for(loop=1; loop<=info->input_values[0]; loop++)
     info->numnodes[0] += info->input_values[loop];

   /* count hidden nodes */
   for(loop=1; loop<=info->numhidlayers; loop++)
     info->numnodes[loop] = info->numhidnodes[loop-1];

   /* count output nodes */
   info->numnodes[info->numlayers-1] = 0;
   for(loop=1; loop<=info->output_values[0]; loop++)
     info->numnodes[info->numlayers-1] += info->output_values[loop];

   return info;
}
```

```
/*********************************************************************
********
*   malloc function with the advantage of error catching
*********************************************************************
*******/
void *MemAlloc(size_t size)
{
  void *ptr = NULL;
  if(size != 0) {
    ptr = malloc(size);
    if (!ptr) {
      printf("\nOut of memory error.\n");
      exit(1);  }  }
  return ptr;
}

/*********************************************************************
********
*  free function with benefit of nulling pointers
*********************************************************************
*******/
void Free(void *ptr)
{
  if(ptr)
    free(ptr);
  ptr = NULL;
}

/*********************************************************************
*******
*  Graphics function
*********************************************************************
******/
int InitializeScreen(void)
{
  /* Settings for paging */
  int gdriver = VGA, gmode = VGAMED, gerr;

  registerbgidriver(EGAVGA_driver);
  initgraph(&gdriver, &gmode, "");
  gerr = graphresult();
  if(gerr != grOk)  {
  printf("BGI error: %s\n", grapherrormsg(gerr));
  return(gerr);}

  /* Redefine coordinate system by moving viewport */
  setviewport(getmaxx()/2, getmaxy(), getmaxx(), getmaxy(), CLIPOFF);
  setlinestyle(SOLID_LINE, 0, NORM_WIDTH);
  return 0;
}

/*********************************************************************
********
*  Return the output of the network
*********************************************************************
*******/
void ReadOutput(struct Net *net, struct NetworkInfo *info, double *out)
```

```
{
  int loop, numnodes;
  double *fuzzy;
  struct Layer *layer = NULL;
  struct Node *node = NULL;

  layer =  net->layers;
  loop = net->numlayers - 1;
  numnodes = layer[loop].numnodes;
  if (out[0] != info->numoutputs) {printf("ERROR in programming
ReadOutput."); exit(1); }

  fuzzy = MemAlloc(sizeof(double) * (numnodes+1));
  fuzzy[0] = numnodes;

  node = layer[loop].nodes;
  for(loop = 1; loop <= numnodes; loop++)
    fuzzy[loop] = node[loop-1].output;

  Defuzzify(fuzzy, out, info->output_max);
  Free(fuzzy);
}

/**********************************************************************
********
*  FNN_BPN.C
*  the basic neural network include file for BPNs
*  by Wallace Kelly
*  February 9, 1994
*  adapted for fuzzy 2/22/94
*  revision on 5/17/94 to add flexibility
*  revision on 6/14/94 for minor changes
***********************************************************************
*******/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>

#include "fnn.h"

/**********************************************************************
********
*  Global variables:    activation constant, alpha
*                             learning rate, beta
*                     and momentum constant, gamma
***********************************************************************
*******/
double alpha, beta, gamma;
void BPN_Constants(double a, double b, double g)
{
  alpha = a;
  beta = b;
  gamma = g;
}
```

```
/*********************************************************************
********
*  The activiation function
*********************************************************************
*******/
double Activation(double x)
{
  double y;
  y = 1 / (1 + exp(- alpha *x));
  return y;
}

/*********************************************************************
********
*  Perform the forward propagation
*********************************************************************
*******/
void RunNN(struct Net *net)
{
  int loop, loop2, loop3;
  struct Layer *layer = NULL;
  struct Node *node = NULL, *nodeprev = NULL;

  for(loop = 1; loop < net->numlayers; loop++)  {
    layer =  net->layers;
    for(loop2 = 0; loop2 < layer[loop].numnodes; loop2++)  {
      node = layer[loop].nodes;
      nodeprev = layer[loop-1].nodes;
      node[loop2].sum = 0;
      for(loop3 = 0; loop3 < node[loop2].numweights; loop3++)
        node[loop2].sum += nodeprev[loop3].output *
node[loop2].win[loop3];
      node[loop2].output = Activation(node[loop2].sum);   } }
}

/*********************************************************************
********
*  Perform the backpropagation
*********************************************************************
*******/
void BackPropagate(double *desired, struct Net *net)
{
  int loop, loop2, loop3, numlayers, numnodes, numnodesprev;
  double output, sum, temp;
  struct Layer *layer = NULL;
  struct Node *node = NULL, *nodeprev = NULL;

  /* set layer and node variables to the last layer */
  numlayers = net->numlayers;
  layer = net->layers;
  numnodes = layer[numlayers-1].numnodes;
  node = layer[numlayers-1].nodes;

  /* calulate error on output layer */
  for(loop2 = 0; loop2 < numnodes; loop2++)  {
    output = node[loop2].output;
    node[loop2].error = output * (1 - output) * (desired[loop2+1] -
output); }

  /* calculate error in hidden nodes */
  for(loop = numlayers-1; loop > 1; loop--)  {
    numnodes = layer[loop].numnodes;
    numnodesprev = layer[loop-1].numnodes;
    node = layer[loop].nodes;
    nodeprev = layer[loop-1].nodes;
    for(loop2 = 0; loop2 < numnodesprev; loop2++)  {
      sum = 0;
      output = nodeprev[loop2].output;
      for(loop3 = 0; loop3 < numnodes; loop3++)
        sum += node[loop3].error * node[loop3].win[loop2];
      nodeprev[loop2].error = output * (1 - output) * sum;  }  }

  /* adjust the weights */
  for(loop = numlayers-1; loop>0; loop--)  {
    numnodes = layer[loop].numnodes;
    numnodesprev = layer[loop-1].numnodes;
    node = layer[loop].nodes;
    nodeprev = layer[loop-1].nodes;
    for(loop2 = 0; loop2 < numnodesprev; loop2++)  {
      for(loop3 = 0; loop3 < numnodes; loop3++)  {
        temp = node[loop3].win[loop2];
        node[loop3].win[loop2] = node[loop3].win[loop2]
          + beta * node[loop3].error * nodeprev[loop2].output
          + gamma * (temp - node[loop3].winlast[loop2]);
        node[loop3].winlast[loop2] = temp; } } }
}

/***********************************************************************
********
*  Calculate the root mean square error of an input
************************************************************************
*******/
double ErrorNN(double *inputs, double *desired, struct Net *net, struct
NetworkInfo *info)
{
  int loop2;
  double error = 0, *defuzzified, *fuzzified, *fuzzy;
  struct Layer *layer= NULL;
  struct Node *node = NULL;

  layer = net->layers;
  node = layer[net->numlayers - 1].nodes;

  fuzzified = MemAlloc(sizeof(double) * (layer[0].numnodes+1));
  fuzzified[0] = layer[0].numnodes;

  fuzzy = MemAlloc(sizeof(double) * (layer[net->numlayers -
1].numnodes+1));
  fuzzy[0] = layer[net->numlayers - 1].numnodes;

  defuzzified = MemAlloc(sizeof(double) * (info->numoutputs +1));
  defuzzified[0] = info->numoutputs;
  Fuzzify(inputs, fuzzified, info->input_max);
  InputArray(fuzzified, net);
  RunNN(net);

  for(loop2 = 1; loop2 <= fuzzy[0]; loop2++)
    fuzzy[loop2] = node[loop2-1].output;
  Defuzzify(fuzzy, defuzzified, info->output_max);
  for(loop2 = 0; loop2 < defuzzified[0]; loop2++)
    error += pow((defuzzified[loop2+1] - desired[loop2+1]), 2.0);

  error = sqrt(error / defuzzified[0]);

  Free(defuzzified);
  Free(fuzzy);
  Free(fuzzified);

  return error;
}

/**********************************************************************
********
*  FUZZIFY.C
*  code to fuzzify input to a neural network
*  by Wallace Kelly
*  2/22/94
*  revision 5/17/94 added MembershipInfo file
*  revision 6/14/94 minor changes
**********************************************************************
*******/
#include <stdio.h>
#include <stdlib.h>

#include "fnn.h"

/**********************************************************************
********
* Fuzzify the values of an array
**********************************************************************
*******/
void Fuzzify(double *in, double *fuzzified, double **max_points)
{
  int loop, loop2, k=0;
  double **m;

  m = max_points;

  /* initialize all the membership values to zero */
  for(loop = 1; loop <= fuzzified[0]; loop++)
    fuzzified[loop] = 0;

  /* step through all the crisp inputs */
  for(loop = 1; loop <= in[0]; loop++)  {

    /* if the fuzzifier simply passes a crisp value through... */
    if(m[loop-1][0] == 1) { k++; fuzzified[k] = in[loop];  }

    /* for the case of membership function defined */
```

```
     else {
       for(loop2=1; loop2<=m[loop-1][0]-1; loop2++)  {
         k++;
         if(m[loop-1][loop2] <= in[loop] && m[loop-1][loop2+1] >=
in[loop])  {
           fuzzified[k] =  (m[loop-1][loop2+1] - in[loop]) / (m[loop-
1][loop2+1] - m[loop-1][loop2]);
           fuzzified[k+1] = 1 - fuzzified[k];  }  }
       k++; }  }
  return;
}

/*********************************************************************
********
* Defuzzify the values of an array
**********************************************************************
*******/
void Defuzzify(double *f, double *out, double **max_points)
{
  int loop, loop2, k=0;
  double **m;
  double sum1, sum2;

  m = max_points;

  for(loop2=1; loop2<=out[0]; loop2++) {

    /* if the defuzzifier simply passes a crisp value...*/
    if(m[loop2-1][0] == 1)  {
      k++; out[loop2] = f[k];  }

    else {
      sum1 = 0;   sum2 = 0;
      for(loop=1; loop<=m[loop2-1][0]; loop++)  {
        k++;
        sum1 += f[k];
        sum2 += f[k] * m[loop2-1][loop];  }
      out[loop2] = sum2 / sum1; }
    }

  return;
}

/*********************************************************************
********
*  TRAIN.C
*  trains a neuro-fuzzy network
*  by Wallace Kelly
**********************************************************************
*******/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>
#include <conio.h>
```

```
#include "fnn.h"

main(int argv, char *argc[])
{
  int flag;
  unsigned loop, loop2;
  char *filename = NULL;
  double *arrays = NULL, *desired = NULL, *fuzzified = NULL,
*defuzzified = NULL;
  double error, maxerror, rmserror;
  struct Net *net = NULL;
  struct NetworkInfo *info = NULL;
  FILE *fp = NULL, *fp_in = NULL, *fp_desired = NULL;

/***********************************************************************
********
*  Determine the name of the file containing the network information
************************************************************************
*******/
  clrscr();
  if( argv < 2 )  {
    if( (filename = GetFileName("containing network information")) ==
NULL)
      exit(1);  }
  else {
    filename = MemAlloc(sizeof(char) * strlen(argc[1]));
    strcpy(filename, argc[1]);  }

/***********************************************************************
********
*  Read in the network information
************************************************************************
*******/
  fp = Fopen(filename, READ);
  if(!fp) { printf("ERROR opening %s.", filename); exit(1); }
  info = GetNetInfo(fp);
  fclose(fp); fp = NULL;
  Free(filename);  filename = NULL;

/***********************************************************************
********
*  Initialize the network, weights, fuzzifiers, arrays, and training
constants
************************************************************************
*******/
  net = IniNet(info->numlayers, info->numnodes);
  LoadWeights(info->weights, net);

  arrays = MemAlloc(sizeof(double) * (info->numinputs +1));
  arrays[0] = info->numinputs;

  desired = MemAlloc(sizeof(double) * (info->numoutputs +1));
  desired[0] = info->numoutputs;

  fuzzified = MemAlloc(sizeof(double) * (info->numnodes[0]+1));
  fuzzified[0] = info->numnodes[0];
  defuzzified = MemAlloc(sizeof(double) * (info->numnodes[info-
>numlayers-1]+1));
  defuzzified[0] = info->numnodes[info->numlayers-1];

  BPN_Constants(info->alpha, info->beta, info->gamma);

/**********************************************************************
********
*  Open the files for error, input, and desired data
***********************************************************************
*******/
  fp = Fopen(info->error, APPEND);
    if(!fp) { printf("Error opening %s", info->error); exit(1);  }
  fp_in = Fopen(info->samples, READ);
    if(!fp_in) { printf("Error opening %s", info->samples); exit(1);  }
  fp_desired = Fopen(info->desired, READ);
    if(!fp_desired) {printf("Error opening %s", info->desired);
exit(1); }

/**********************************************************************
********
*  The main training loop
***********************************************************************
*******/
  clrscr(); loop = 0; loop2 = 0;  error = 0;  maxerror = 0;  rmserror =
0;

  do {

/**********************************************************************
********
*  Read in the state and desired vectors
***********************************************************************
*******/
     flag = DoubleIn(fp_in, arrays, info->numinputs);
     if(    DoubleIn(fp_desired, desired, info->numoutputs) != flag)  {
       printf("\nThe input and desired files are of different
lengths.\n");
       exit(1);  }

/**********************************************************************
********
*  In the case that the end of the files were reached (flag==0), ...
***********************************************************************
*******/
     if(flag == 0) {
       if(!kbhit())
         fprintf(fp, "%0.8lf %0.8lf\n", maxerror, (
sqrt(rmserror/loop)));
       loop = 0;  loop2++;   error = 0;  maxerror = 0;  rmserror = 0;
  }

/**********************************************************************
********
*  Fuzzify, load, propogate, and backpropogate
***********************************************************************
*******/
```

```
     loop++;
     printf("\n\nTraining loop #%d.  Input vector #%d.", loop2, loop);

     Fuzzify(arrays, fuzzified, info->input_max);
     Fuzzify(desired, defuzzified, info->output_max);

     InputArray(fuzzified, net);
     RunNN(net);
     BackPropagate(defuzzified, net);

/***********************************************************************
********
*  Calculate and show errors
***********************************************************************
*******/
     error = ErrorNN(arrays, desired, net, info);
     if(maxerror < error) maxerror = error;
     rmserror += pow(error, 2.0);
     printf("\nLast Error:    %0.8lf", error);
     printf("\nMaximum Error: %0.8lf", maxerror);
     printf("\nRMS Error:     %0.8lf", ( sqrt(rmserror/loop)));
     gotoxy(1,1);

/***********************************************************************
********
*  If keyboard not hit and we haven't exceeded the max iterations,
continue
***********************************************************************
*******/
     } while( loop2 < info->iterations && !kbhit());

/***********************************************************************
********
*  Do the final housekeeping
***********************************************************************
*******/
   StoreWeights(info->weights, net);
   fclose(fp);
   if(kbhit()) getch();
   return 0;
}
```

NEURO-FUZZY CONTROL

```
/***********************************************************************
*******
*  FARM.C
*  Source code for a program to simulate a robotic arm
*  The arm is simulated as a 3-link revolute arm in 2 dimensions
*  Physical constraints are included in this model as well
*  as a goal and obstacle to be avoided.
*  The arm is controlled by a fuzzy neural network
*  This simulation does random configurations and returns a histogram
*  of the distances from the goal and the percentages of collision free
*  trajectories, etc...
*  by Wallace Kelly
*  March 22, 1994
*  revision on June 16, 1994
**********************************************************************
******/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>
#include <graphics.h>
#include <conio.h>
#include <ctype.h>

#include "fnn.h"
#include "farm.h"

#define LEN 80
#define OBJECT_MAX 2.65
#define OBJECT_MIN 2.25
#define OBJECT_MARGIN 100
#define HUNDRED_PERCENT 400
#define HISTOGRAM_RESOLUTION .05
#define MAX_MOVES 100
#define RUNS 100

main(int argv, char *argc[])
{
  int runs = 0, loop, loop2, pause = ON, drawing = ON, c;
  int constraints = 0, collisions = 0, success, *histogram;
  char *filename = NULL, *drawoff = "off";
  double *fuzzified = NULL, *defuzzified = NULL;
  double state[8], delta[4], d, d1, d2, min_distance;
  struct Net *net = NULL;
  struct NetworkInfo *info = NULL;
  struct Link *arm = NULL;
  struct complex end, g, o;
  struct Point goal, obstacle;
  FILE *fp = NULL;

  histogram = MemAlloc(sizeof(int) * (1 / HISTOGRAM_RESOLUTION + 1));

/*********************************************************************
********
*  Determine the name of the file containing the network information
**********************************************************************
*******/
  clrscr();
  if( argv < 2 )  {
    if( (filename = GetFileName("containing network information")) ==
NULL)
      exit(1);  }
  else {
    filename = MemAlloc(sizeof(char) * strlen(argc[1]));
    strcpy(filename, argc[1]);  }
  if( argv == 3)
    if( strequ(argc[2], drawoff)) { drawing = OFF; pause = OFF; }

/**********************************************************************
********
*  Read in the network information
***********************************************************************
*******/
  fp = Fopen(filename, READ);
  if(!fp)  { printf("ERROR opening %s.", filename); exit(1); }
  info = GetNetInfo(fp);
  fclose(fp); fp = NULL;
  Free(filename);

/**********************************************************************
********
*  Check for correct dimensions in FNN specified
***********************************************************************
*******/
  if(info->numinputs != 7) {
    printf("\nThe fuzzy neural network must have seven inputs.");
    exit(1);  }
  if(info->numoutputs != 3) {
    printf("\nThe fuzzy neural network must have three outputs.");
    exit(1);  }
  state[0] = 7;
  delta[0] = 3;

/**********************************************************************
********
*  Initialize the network, weights, fuzzifiers, arrays, and training
constants
***********************************************************************
*******/
  net = IniNet(info->numlayers, info->numnodes);
  LoadWeights(info->weights, net);

  fuzzified = MemAlloc(sizeof(double) * (info->numnodes[0]+1));
  fuzzified[0] = info->numnodes[0];

  defuzzified = MemAlloc(sizeof(double) * (info->numnodes[info-
>numlayers-1]+1));
  defuzzified[0] = info->numnodes[info->numlayers-1];

  BPN_Constants(info->alpha, info->beta, info->gamma);
  for(loop = 0; loop < 20; loop++)
    histogram[loop] = 0;

/**********************************************************************
********
*  Initialize the arm simulation
***********************************************************************
*******/
  randomize();
  if(drawing) {
    InitializeScreen();
```

```
      setbkcolor(EGA_BLACK);  }

    /* Initialize the arm */
    arm = IniLinks();
    AddLink(arm, LEN, 0, M_PI, 0);
    AddLink(arm, LEN*0.824, 0, M_PI_2, -M_PI_2);
    AddLink(arm, LEN*0.737, 0, M_PI*0.135, -M_PI*0.135);

/***********************************************************************
********
*  The main loop
***********************************************************************
*******/
    clrscr();

    for(loop = 0; loop < RUNS; loop++)  {

      /* Select random locations for obstacle and goal */
      do {
        obstacle.rho = Random(OBJECT_MIN * LEN, OBJECT_MAX * LEN);
        obstacle.phi = Random(.1, 3.04);
        goal.rho = Random(OBJECT_MIN * LEN, OBJECT_MAX * LEN);
        goal.phi = Random(.1, 3.04);
        } while(PDistance(&obstacle, &goal) < OBJECT_MARGIN);

      /* Select a random configuration of the arm */
      do {
        SetLink(arm, 0, Random(.25, .75));
        SetLink(arm, 1, Random(.25, .75));
        SetLink(arm, 2, Random(.25, .75));
        EndEffector(arm, &end);
        } while(end.y < 0 || Collision(arm, &obstacle, 75) ||
Collision(arm, &goal, 75));

      if(drawing) {
        cleardevice();
        DrawObstacle(&obstacle);
        DrawGoal(&goal);
        DrawArm(arm, EGA_YELLOW);  }

      else printf(".");

      min_distance = 9999;

/***********************************************************************
********
*  The loop for moving the arm for a given configuration
***********************************************************************
*******/
      for(loop2 = 0; loop2 < MAX_MOVES; loop2++)  {
        GetState(state, arm, &goal, &obstacle);

        Fuzzify(state, fuzzified, info->input_max);
        InputArray(fuzzified, net);
        RunNN(net);
        ReadOutput(net, info, delta);
        if(drawing) DrawArm(arm, EGA_LIGHTGRAY);
        EndEffector(arm, &end);

        Pol2Rec(&goal, &g);
        Pol2Rec(&obstacle, &o);
        d1 = Distance(&end, &g);
        d2 = Distance(&end, &o);
        if(d1 < d2) d = d1;
        else d = d2;
        if(d1 < min_distance) min_distance = d1;

        success = 1;
        success = (success && MoveLink(arm, 0, (delta[1]*2 - 1) * d /
1000));
        success = (success && MoveLink(arm, 1, (delta[2]*2 - 1) * d /
1000));
        success = (success && MoveLink(arm, 2, (delta[3]*2 - 1) * d /
1000));
        if(drawing) DrawArm(arm, EGA_YELLOW);

        if(!success) {
          printf("\a");
          loop2 = MAX_MOVES;
          constraints++; }

        else
          if(Collision(arm, &obstacle, 10)) {
            printf("\a");
            loop2 = MAX_MOVES;
            collisions++;  }

        if(d1 / HUNDRED_PERCENT < 0.05) {
          loop2 = MAX_MOVES;  }

        if(kbhit()) c = tolower(getch());
        }
  if(!Collision(arm, &obstacle, 10) && success)
    histogram[(int)floor(min_distance / HUNDRED_PERCENT /
HISTOGRAM_RESOLUTION)]++;

  switch(c) {
    case 'q': loop = RUNS; pause = OFF; break;
    case 'f': pause = 1 - pause; c = 0; break;  }

  if(pause || !drawing) c = tolower(getch());
  }

/***********************************************************************
********
*  Do the final housekeeping
************************************************************************
*******/
  if(drawing) closegraph();
  printf("collisions %0.2lf\n", (double)collisions / (double)RUNS);
  printf("constraints %0.2lf\n", (double)constraints / (double)RUNS);
  for(loop = 0; loop < 20; loop++)
```

```
     printf("%0.2lf %0.2lf\n", (double)loop * HISTOGRAM_RESOLUTION,
(double)histogram[loop] / (double)RUNS);
  return 0;
}
```

APPENDIX II

NEURO-FUZZY ENGINE TESTING

The programming and debugging of the neuro-fuzzy engine was an extensive process. This appendix includes the data from that testing process and verifies the validity of the neuro-fuzzy engine shown in Appendix I.

LINEAR FUNCTION

The first function used to test the neuro-fuzzy system was $y = x$. After 500 training epochs, the neuro-fuzzy system produced the output shown in figure A2-1.

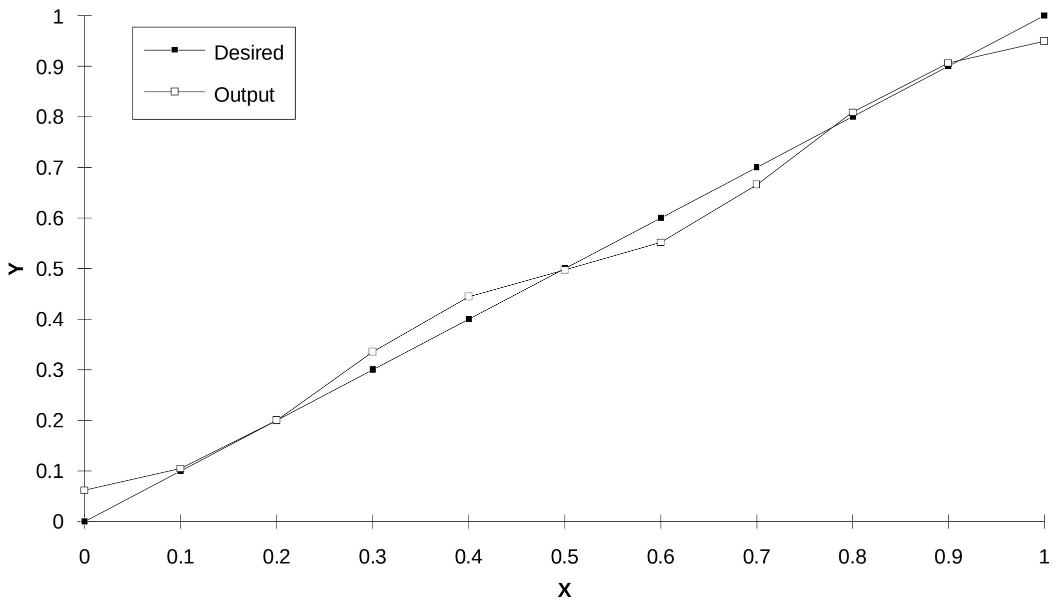


Figure A2-1. The performance of the neuro-fuzzy system on a linear function.

SINUSOIDAL FUNCTION

The ability of the neuro-fuzzy system to learn more complex data was tested with the function $y = sin(x)$. Figure A2-2 shows the desired output and the network's output after 250 training epochs.

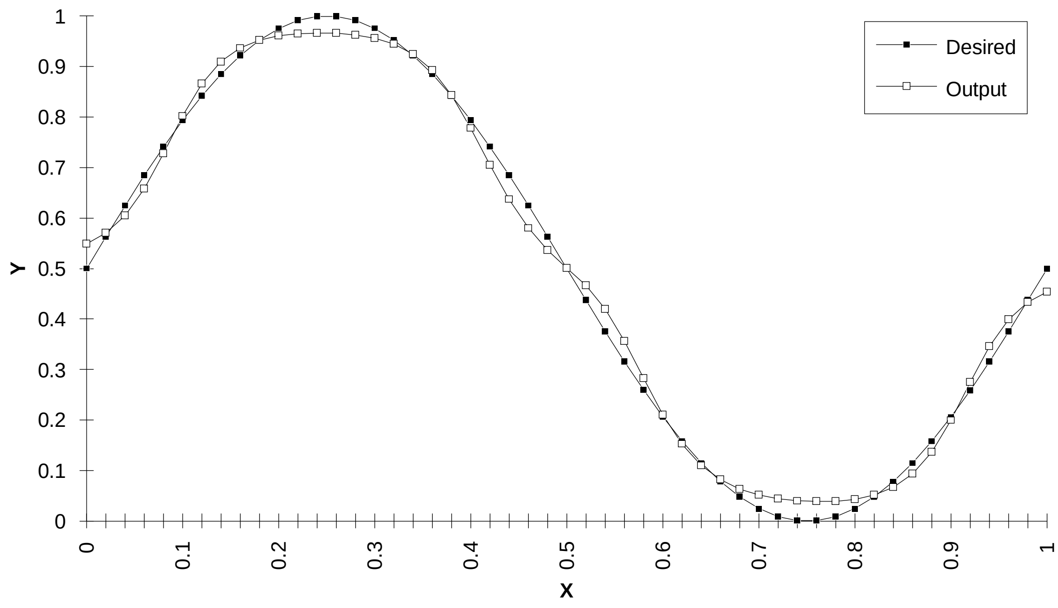
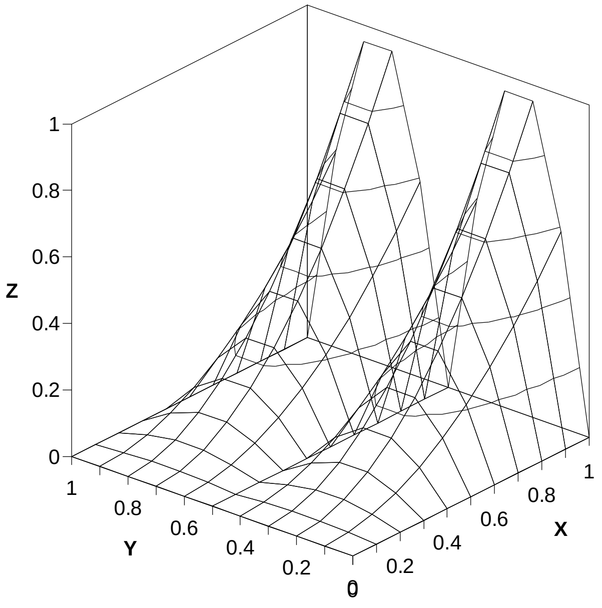


Figure A2-2. The performance of the neuro-fuzzy system on a sinusoidal function.

TWO-DIMENSIONAL FUNCTION

A programming bug existed in the neuro-fuzzy engine that was not a problem for single-input, single-output functions. Testing with $z = x^2|sin(y)|$ revealed the error. After correcting the programming bug and training the network for 400 epochs, the network learned the function, as shown in figure A2-3.



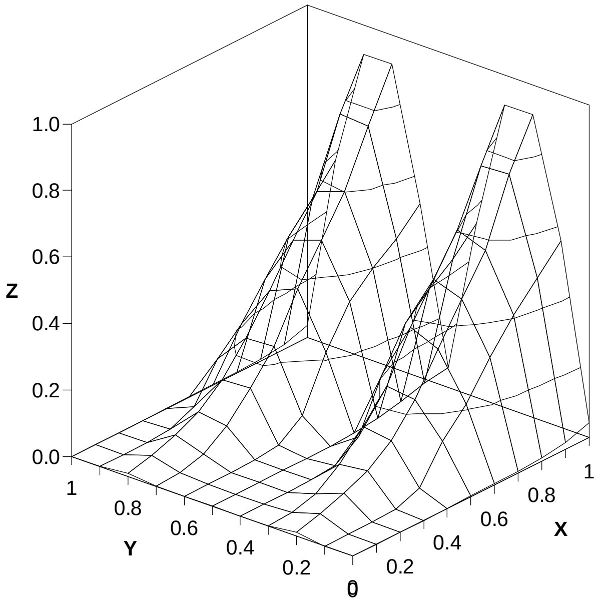


Figure A2-3. The desired output (top) and the actual output for the 2D test data.

The Learning Rate Constant

Another important part of any system involving a back propagation neural network is the learning rate constant. The learning rate constant, β, determines how much the weights in the neural network are adjusted as the error is back propagated. Equation 6 on page 10 shows the learning equation used in most BPNs. The effect of the learning rate constant on three training sessions for the sine function is shown in figures A2-4. All three training sessions began the networks with the same initial weights. The training error decreases faster for the network with the higher learning rate.

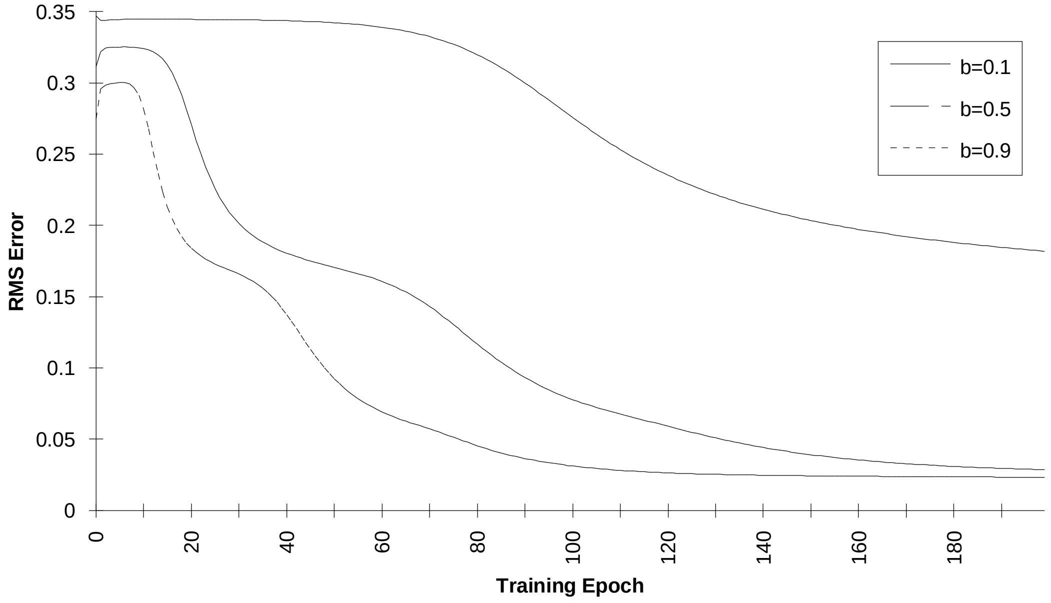


Figure A2-4. The effect of the learning rate constant.

THE MOMENTUM CONSTANT

It became apparent during the early stages of this thesis project that the neuro-fuzzy system would benefit from the addition of a momentum term in the learning rule. Experiments show that the momentum term causes the system to converge more quickly. Figure A2-5 is a comparison of the network output after two training sessions that were identical except for the inclusion of a momentum term.

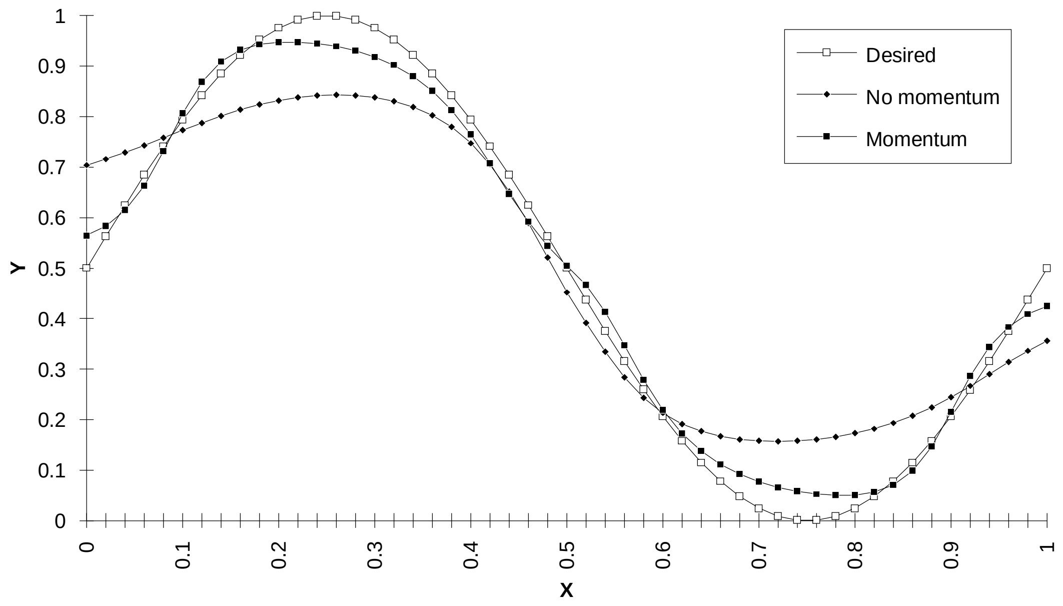


Figure A2-5. The effect of the momentum term.

THE FUZZY MEMBERSHIP FUNCTIONS

In order to test the fuzzy membership function features, the neuro-fuzzy engine was tested for several functions with varying membership function definitions. Figure A2-6 reveals the effect of the fuzzy membership function definition on a network training to a sinusoidal function. The "error1" line corresponds to the output error of the network with input and output fuzzy membership function definition #1, shown in figure A2-7(a). The plot of "error2" corresponds to definition #2, shown in figure A2-7(b). The error was less for the fuzzifier/defuzzifier with the higher granularity.

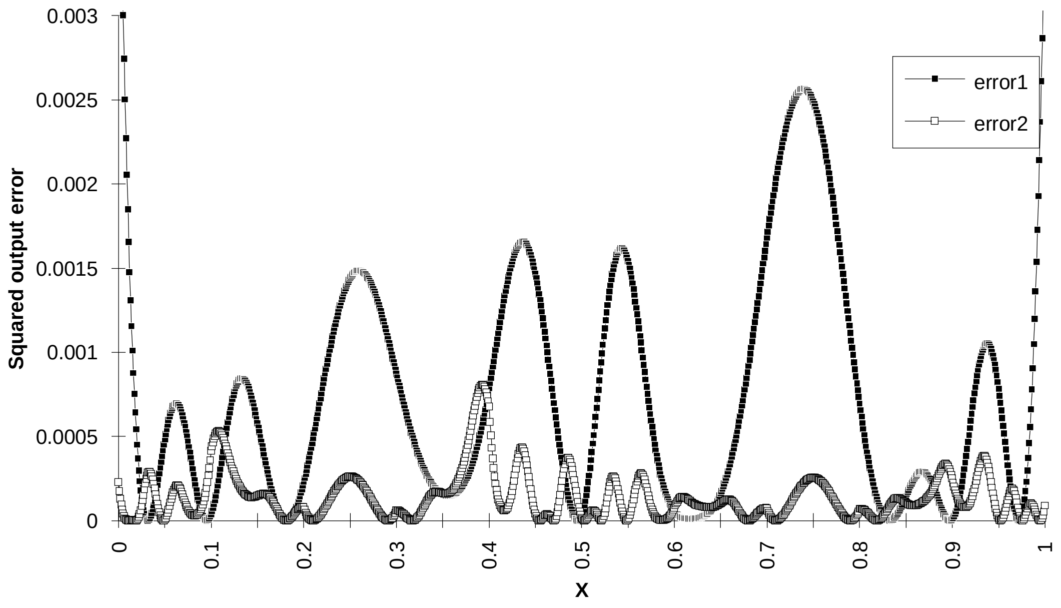


Figure A2-6. The effect of the fuzzy membership function definition.

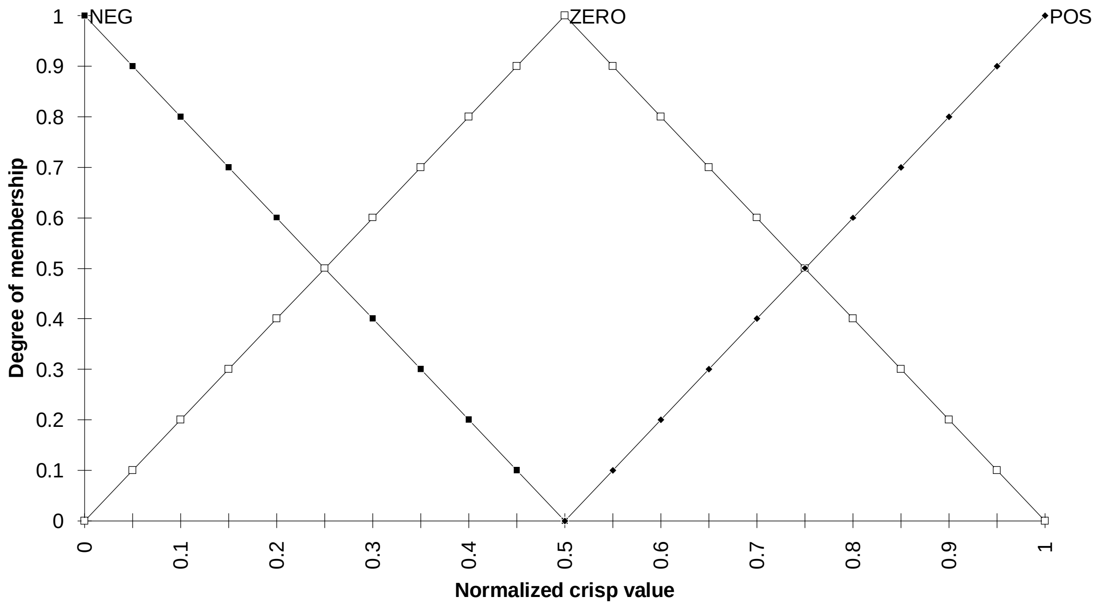


Figure A2-7(a). The fuzzy membership function, test definition #1.

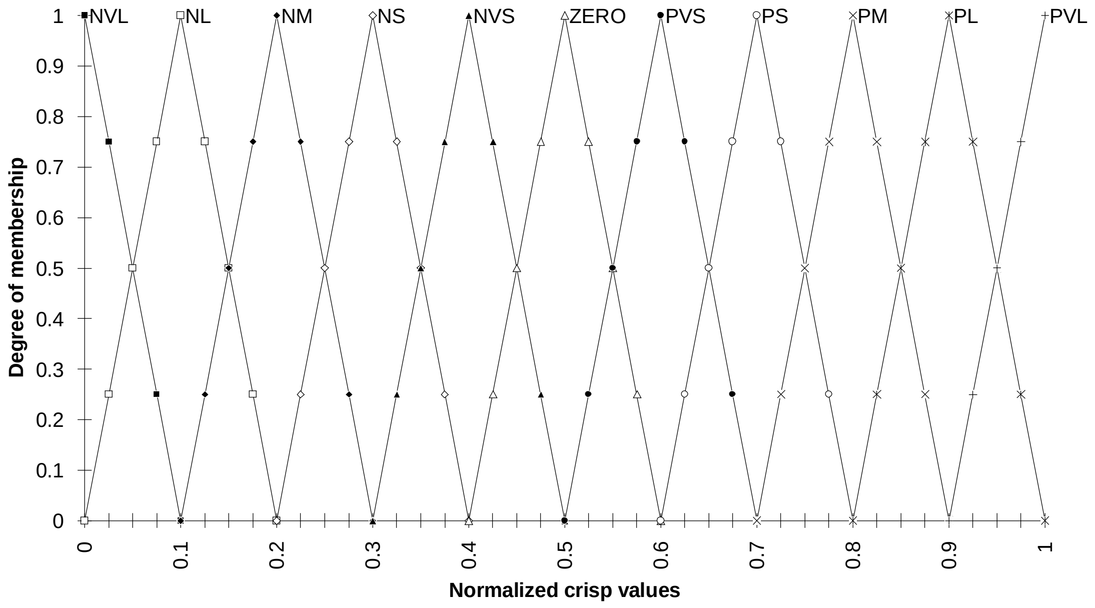


Figure A2-7(b). The fuzzy membership function, test definition #2.

APPENDIX III

THE FNN DATA FILES

This appendix contains the ASCII text files which defined the neuro-fuzzy controllers tested in the experiments of Chapter VIII. Of primary interest is the fuzzy membership function definitions. All the fuzzy membership functions are assumed to be triangular, with a heighth of unity, and a peak at the values labeled "input_max" and "output_max."

ARM1.FNN
```
inputs=7
outputs=3
hidden=1
hidnodes=35
alpha=1
beta=.9
gamma=.1
iterations=300
maxerror=.01
input_file=input.dat
samples_file=input.dat
desired_file=output.dat
weights_file=weights.dat
error_file=error.dat
input_values=3 3 3 3 3 3 3
output_values=3 3 3
input_max=
0.0 0.5 1.0
0.0 0.5 1.0
0.0 0.5 1.0
0.0 0.5 1.0
0.0 0.5 1.0
0.0 0.5 1.0
0.0 0.5 1.0
output_max=
0.0 0.5 1.0
0.0 0.5 1.0
0.0 0.5 1.0
```

ARM2.FNN
```
inputs=7
outputs=3
hidden=1
hidnodes=65
alpha=1
beta=.9
gamma=.1
iterations=300
maxerror=.01
```

```
input_file=input.dat
samples_file=input.dat
desired_file=output.dat
weights_file=weights.dat
error_file=error.dat
input_values=7 7 7 7 7 7 7
output_values=3 3 3
input_max=
0.0 0.166 0.333 0.5 0.666 0.833 1.0
0.0 0.166 0.333 0.5 0.666 0.833 1.0
0.0 0.166 0.333 0.5 0.666 0.833 1.0
0.0 0.166 0.333 0.5 0.666 0.833 1.0
0.0 0.166 0.333 0.5 0.666 0.833 1.0
0.0 0.166 0.333 0.5 0.666 0.833 1.0
0.0 0.166 0.333 0.5 0.666 0.833 1.0
output_max=
0.0 0.5 1.0
0.0 0.5 1.0
0.0 0.5 1.0
```

ARM3.FNN

```
inputs=7
outputs=3
hidden=1
hidnodes=30 16
hidnodes=65
alpha=1
beta=.9
gamma=.1
iterations=300
maxerror=.01
input_file=input.dat
input_file
samples_file=input.dat
samples_file inputs
desired_file=output.dat
weights_file=weights.dat
error_file=error.dat
input_values=11 11 11 7 7 7 7
input_values=7 7 7 7 7 7 7
output_values=3 3 3
output_values=3 3 3
input_max=
0.0 0.166 0.333 0.5 0.666 0.833 1.0
0.0 0.166 0.333 0.5 0.666 0.833 1.0
0.0 0.166 0.333 0.5 0.666 0.833 1.0
0.0 0.333 0.450 0.5 0.550 0.666 1.0
0.0 0.333 0.450 0.5 0.550 0.666 1.0
0.0 0.333 0.450 0.5 0.550 0.666 1.0
0.0 0.333 0.450 0.5 0.550 0.666 1.0
output_max=
0.0 0.5 1.0
0.0 0.5 1.0
0.0 0.5 1.0
```

ARM4.FNN

```
inputs=7
outputs=3
hidden=1
hidnodes=30 16
hidnodes=85
alpha=1
beta=.9
gamma=.1
iterations=300
maxerror=.01
input_file=input.dat
input_file
samples_file=input.dat
samples_file inputs
desired_file=output.dat
weights_file=weights.dat
error_file=error.dat
input_values=11 11 11 11 11 11 11
output_values=3 3 3
input_max=
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
output_max=
0.0 0.5 1.0
0.0 0.5 1.0
0.0 0.5 1.0
```

ARM5.FNN

```
inputs=7
outputs=3
hidden=1
hidnodes=85
alpha=1
beta=.9
gamma=.1
iterations=300
maxerror=.01
input_file=input.dat
input_file
samples_file=input.dat
desired_file=output.dat
weights_file=weights.dat
error_file=error.dat
input_values=11 11 11 11 11 11 11
output_values=3 3 3
input_max=
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
0.0 0.15 0.3 0.4 0.45 0.5 0.55 0.6 0.7 0.85 1.0
0.0 0.15 0.3 0.4 0.45 0.5 0.55 0.6 0.7 0.85 1.0
```

```
0.0 0.15 0.3 0.4 0.45 0.5 0.55 0.6 0.7 0.85 1.0
0.0 0.15 0.3 0.4 0.45 0.5 0.55 0.6 0.7 0.85 1.0
output_max=
0.0 0.5 1.0
0.0 0.5 1.0
0.0 0.5 1.0
```

ARM6.FNN

```
inputs=7
outputs=3
hidden=1
hidnodes=100
alpha=1
beta=.9
gamma=.1
iterations=300
maxerror=.01
input_file=input.dat
samples_file=input.dat
desired_file=output.dat
weights_file=weights.dat
error_file=error.dat
input_values=11 11 11 15 15 15 15
output_values=3 3 3
input_max=
0.0 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 1.0
0.0 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 1.0
0.0 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 1.0
0.0 0.07 0.14 0.21 0.28 0.35 0.43 0.50 0.57 0.65 0.72 0.79 0.86 0.93
1.0
0.0 0.07 0.14 0.21 0.28 0.35 0.43 0.50 0.57 0.65 0.72 0.79 0.86 0.93
1.0
0.0 0.07 0.14 0.21 0.28 0.35 0.43 0.50 0.57 0.65 0.72 0.79 0.86 0.93
1.0
0.0 0.07 0.14 0.21 0.28 0.35 0.43 0.50 0.57 0.65 0.72 0.79 0.86 0.93
1.0
output_max=
0.0 0.5 1.0
0.0 0.5 1.0
0.0 0.5 1.0
```

ARM7.FNN

```
inputs=7
outputs=3
hidden=1
hidnodes=100
alpha=1
beta=.9
gamma=.1
iterations=300
maxerror=.01
input_file=input.dat
samples_file=input.dat
desired_file=output.dat
weights_file=weights.dat
error_file=error.dat
```

```
input_values=11 11 11 15 15 15 15
output_values=3 3 3
input_max=
0.0 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 1.0
0.0 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 1.0
0.0 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 1.0
0.0 0.13 0.24 0.33 0.40 0.45 0.48 0.50 0.52 0.55 0.60 0.67 0.76 0.87
1.0
0.0 0.13 0.24 0.33 0.40 0.45 0.48 0.50 0.52 0.55 0.60 0.67 0.76 0.87
1.0
0.0 0.13 0.24 0.33 0.40 0.45 0.48 0.50 0.52 0.55 0.60 0.67 0.76 0.87
1.0
0.0 0.13 0.24 0.33 0.40 0.45 0.48 0.50 0.52 0.55 0.60 0.67 0.76 0.87
1.0
output_max=
0.0 0.5 1.0
0.0 0.5 1.0
0.0 0.5 1.0
```