# Learning Systems for Nonlinear Mapping

by

Jordan Augustus Richardson

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 6, 2017

Keywords: Computational Intelligence, Machine Learning, Artificial Neural Networks, Radial Basis Functions, Fuzzy Systems, Splines

Copyright 2017 by Jordan Augustus Richardson

Approved by

Bogdan Wilamowski, Chair, Professor of Electrical and Computer Engineering
Michael Hamilton, Associate Professor of Electrical and Computer Engineering
Robert Dean, Professor of Electrical and Computer Engineering
Vitaly Vodyanoy, Professor of Anatomy, Physiology, and Pharmacy

Abstract

The many complex problems facing researchers and engineers demand innovative solutions. Machine learning techniques are growing in popularity due to their versatility and power. However, challenges remain. Popular machine learning algorithms such as Artificial Neural Networks are difficult to train, and require many designer choices that heavily impact the performance of the network. Furthermore, the randomized starting point of most ANN variants means that even if optimal choices are made, it may still take multiple trials to obtain satisfactory results. Fuzzy Systems are also widely used, but cannot tackle high dimensional problems or produce outputs of similar quality to neural networks. A novel defuzzification routine based on cubic splines seeking to improve the performance of FS is introduced, and compared to many state of the art machine learning techniques. The experimental results show the proposed algorithm performs competitively with popular machine learning methods, while not requiring a lengthy training process.

## Acknowledgements

First, I would like to acknowledge my advisor and committee chair Dr. Bogdan Wilamowski. Without his encouragement and guidance, I would likely never have begun pursuing my doctorate, much less completed it. It has been the privilege of a lifetime to get to work with and learn from him.

I want to thank my entire committee Dr. Hamilton, Dr. Dean, Dr. Vodyanoy, and my outside reader Dr. Chapman for their time and valuable feedback. The quality of this manuscript has been greatly enhanced as a result.

There have been many professors in the Electrical and Computer Engineering department who have helped me on this journey, and to whom I want to express my deepest gratitude.

I would like to thank Dr. Wentworth, who has been a mentor and friend for my entire academic career, and Dr. Nelson, with whom I have had the pleasure of teaching for the past few years.

I must also thank my parents for their love and support and for encouraging me from a young age to embrace my love of learning. I literally would not be here without you.

Lastly, I want to thank the love of my life for her superhuman patience, tireless support, and unwavering belief in me. Tiffany, this is for you.

Table of Contents


Abstract ..... ii

Acknowledgements ..... iii

List of Tables ..... vi

List of Figures ..... vii

Chapter 1 Introduction ..... 1

Chapter 2 Machine Learning Overview ..... 3

2.1 Artificial Neural Networks ..... 3

2.1.1 Neural Network Size ..... 8

2.1.2 Neural Network Architecture ..... 10

2.1.3 Neural Network Training ..... 15

2.1.4 Specialized Neural Networks ..... 21

2.2 Fuzzy Systems ..... 24

2.2.1 TSK Fuzzy Systems ..... 25

2.2.2 Adaptive Network-based Fuzzy Inference Systems ..... 26

Chapter 3 Nearest Neighbor Spline Approximation ..... 27

3.1 Polynomial Approximation Review ..... 27

3.1.1 Polynomial Approximation Derivation ..... 27

3.1.2 Polynomial Approximation Problems ..... 28

3.2 Cubic Spline Review ..... 32

3.2.1 Cubic Spline Derivation ..... 35

3.2.2 End Conditions ..... 39

3.2.3 Cubic Spline Problems ..... 40

3.3 Nearest Neighbor Spline Approximation ..... 40

3.3.1 One Dimensional TSK ..... 41

3.3.2 One Dimensional NNSA ..... 42

3.3.3 Extension to Multiple Dimensions ..... 51

3.3.4 Comments ..... 56

Chapter 4 Experimental Results ..... 56

4.1 Peaks Problem ........................................................................................................ 57
4.2 Forward Kinematics ................................................................................................ 60
4.3 Multidimensional Schwefel Function ....................................................................... 63
Chapter 5 Conclusions .................................................................................................. 73
References ..................................................................................................................... 75
Chapter 6 Appendices ................................................................................................... 79
6.1 Multidimensional Schwefel Data ............................................................................ 79
6.2 MATLAB Code ........................................................................................................ 82
6.2.1 Algorithm Functions ............................................................................................ 82
6.2.2 Support Functions ................................................................................................ 93
6.2.3 Peaks Experimental Results Code ...................................................................... 102
6.2.4 Forward Kinematics Results Code ...................................................................... 105
6.2.5 Schwefel Function Results Code ........................................................................ 106

List of Tables

**Table I**: Tabulated training and testing errors as the number of neurons increases. ...................... 9

**Table II**: Input patterns for the parity-3 problem. ........................................................................ 10

**Table III**: Comparing the largest parity problem that can be solved by various architectures with eight neurons. ........................................................................................................................ 14

**Table IV**: Results for peaks benchmark in terms of Training and Testing errors (RMSE), Training and Testing Times (s), and the number nodes/neurons. ................................................. 59

**Table V**: Results for Forward Kinematics problem X position in terms of Training and Testing errors (RMSE), Training and Testing Times (s), and the number nodes/neurons. ....................... 62

**Table VI**: Results for Forward Kinematics problem Y position in terms of Training and Testing errors (RMSE), Training and Testing Times (s), and the number nodes/neurons. ....................... 63

List of Figures

Figure 2.1.1 .......... 5

Figure 2.1.2 .......... 7

Figure 2.1.3 .......... 7

Figure 2.1.4 .......... 9

Figure 2.1.5 .......... 11

Figure 2.1.6 .......... 12

Figure 2.1.7 .......... 13

Figure 2.1.8 .......... 14

Figure 2.1.9 .......... 22

Figure 2.2.1 .......... 25

Figure 3.1.1 .......... 28

Figure 3.1.2 .......... 29

Figure 3.1.3 .......... 31

Figure 3.2.1 .......... 33

Figure 3.2.2 .......... 34

Figure 3.2.3 .......... 36

Figure 3.3.1 .......... 41

Figure 3.3.2 .......... 43

Figure 3.3.3 .......... 47

Figure 3.3.4 .......... 50

Figure 3.3.5 .......... 52

Figure 3.3.6 .......... 54

Figure 4.1.1 .......... 57

Figure 4.1.2 .......... 59

Figure 4.2.1 .......... 61

Figure 4.2.2 .......... 62

Figure 4.3.1 .......... 64

Figure 4.3.2 .......... 66

Figure 4.3.3 .......... 67

Figure 4.3.4 .......... 69

Figure 4.3.5 .......... 70

Figure 4.3.6 .......... 71

# Chapter 1 Introduction

Computers dominate our daily lives. Over the course of the last half century, they have evolved from massive, multi-room mainframes to personal computers, laptops, smart phones, tablets, and, recently, wearable computing devices such as smart watches and glasses, all with exponentially more power and capabilities than their vacuum tube forebears. This evolution has been driven by the continual shrinking of silicon process technology, with each reduction of the size of transistors allowing more devices to be packed into the same area. As computing power has grown, applications that once seemed like science fiction have become first possible, then feasible, and eventually commonplace. Despite this incredible progress, there are specific problems and classes of problems that remain intractable with conventional computing. Tasks that appear simple to humans—from moving through space, to making educated guesses based on incomplete information, to dealing with unexpected events—pose incredible difficulty to traditional digital computing.

Machine learning seeks to leverage the power of modern computers to solve these difficult problems. This family of algorithms can be broadly described as attempting to imitate the abilities of organic life to solve problems. The traits that get imitated vary between paradigms, and include movement, evolution, learning, reasoning, etc. Two of the more popular techniques, Artificial Neural Networks (ANN) and Fuzzy Systems (FS) attempt to emulate learning and reasoning, respectively. In theory, ANN and FS can approximate any function to an arbitrary degree of accuracy[1], [2]. In practice, both technologies have strengths and weaknesses.

Fuzzy systems are frequently used in the literature for nonlinear system modelling in which the problems are difficult to describe with mathematical models[3]–[6]. Fuzzy systems are also often used in industry for adaptive control algorithms[7]–[14]. The inherent drawbacks of fuzzy

membership functions result in approximations that lack smoothness. In addition, the process of designing FS can be difficult even for an experienced designer.

Traditional ANNs have recently been joined by radial basis function (RBF) networks, which are often trained by Support Vector Regression (SVR) or by Extreme Learning Machine (ELM) algorithms. Far better results can be obtained with ISO[15] and ErrCor[16] algorithms, which are capable of producing RBF networks more than 10 to 30 times smaller than SVM or ELM. These smaller networks are more suitable for hardware implementation.

There is also recent progress in ANN. It is much easier to train shallow architectures with a single hidden layer than it is to train deep networks[17]. But again, these shallow architectures often require network sizes 10 to 100 times larger than special deep architectures[18]. Unfortunately, these special ANN architectures, such as fully connected cascade (FCC) or bridged multilayer perceptron (BMLP), cannot be trained by commonly-available software, and a special NBN algorithm[19] for arbitrarily connected neural networks must be used.

With advanced training algorithms there is a loss of transparency in the relationship between trainable parameters and system output. A simple and transparent relationship—such as in a TSK[20]–[22] fuzzy system—is crucial for creation of adaptive systems. In traditional TSK fuzzy systems, for each selected area of operation, a specific value of the output is defined. This makes TSK systems very popular—especially with simple triangular membership functions—because this direct relationship between area of operation and desired output can easily be adjusted by a single parameter associated with the area.

The purpose of this work is to examine several state-of-the-art machine learning algorithms, and compare their performance to a new, improved defuzzification technique for zero-order TSK fuzzy systems. The proposed modification seeks to address one of the deficiencies of classic

TSK systems—the rawness of the output surfaces—while maintaining several desirable traits. Specifically, the modification maintains a transparent input-output relationship that allows for easy adjustments. In addition, the proposed approach does not require a lengthy and complicated training process. This has several benefits, including the ability to incorporate new data without necessitating retraining of the system. The newly-developed defuzzification scheme is based on spline-like local third-order polynomials. This local approach was developed to avoid the more complex computation required by global cubic splines [23].

# Chapter 2 Machine Learning Overview

The purpose of this section is to provide a conceptual review of popular machine learning techniques in order to give context to the research presented later. This review will focus on artificial neural networks and fuzzy systems.

## 2.1 Artificial Neural Networks

One of the defining features of life is the ability to adapt to changing environmental circumstances, which can be viewed as a form of problem solving. For simple organisms, natural selection is the primary adaptation mechanism. Natural selection is a slow process, requiring many generations for beneficial mutations or adaptions to occur and be propagated throughout the gene pool. In higher animals, the ability to learn new behaviors provides an alternative way of adapting to new challenges. Higher animals possess brains made up of many nerve cells. Very simplistically, nerve cells, or neurons, are specialized cells that can transmit electrical signals to other cells via connections called synapses. A neuron that receives sufficient input stimulus will fire in turn, thus propagating the signals. There are two features of neurons that are important to this discussion. The first is that a single neuron can form connections with many other neurons, all of which have their own connections and collectively form what is known as a neural

network. The second important feature is that repeated firing of a particular neuron has an effect on the connected neurons. In general, the effect is either excitatory or inhibitory, meaning that the connected neurons will either become more or less likely to fire. In this way, a neural network can adapt to produce responses to repeated stimuli. The power of biological neural networks is readily apparent when examining the complex behavior of higher animals, with the most striking example being humans. Recreating this ability for use in computers has many applications.

An artificial neural network (ANN) is a biologically inspired machine learning paradigm. As the name implies, ANNs attempt to replicate the process by which biological neural networks adapt and learn in order to solve problems. Rather than try to model all of the incredibly complex biological and chemical interactions, ANNs are an approximation of biological neural networks. Figure 2.1.1 shows a standard artificial neuron, called a perceptron, with N inputs $X_1, X_2, \ldots X_n$. Each input $X_i$ has an associated weight $w_i$, which get multiplied together to produce the value actually seen by the neuron. The total or net input to the neuron is calculated as a weighted sum of all the inputs (2.1-1). The output $O_1$ is then computed as the result of the activation function $f(Y)$ operating on the net.

$$Y = x_1 w_1 + \cdots + x_i w_i + \cdots + x_n w_n = \sum_{i=1}^{n} w_i x_i \tag{2.1-1}$$

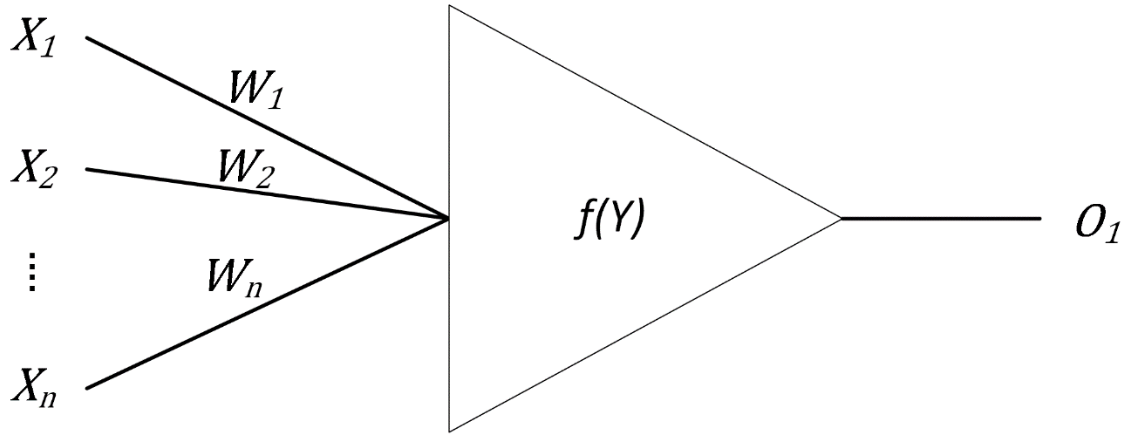



Figure 2.1.1: An artificial neuron with inputs N weighted inputs, activation function $f(Y)$, and one output.

Early artificial neuron models such as the McCullough-Pitts[24] neuron employed "hard" activation functions (essentially a step function). Hard activation functions are either unipolar (2.1-2) or bipolar (2.1-3). While the threshold can be variable, in practice most neurons will have an additional input connected to a constant value which is used to bias the net input for a threshold of 0 or 0.5 for bipolar and unipolar neurons, respectively.

$$O_1 = \begin{cases} 1 & if\ Y \geq T \\ 0 & if\ Y < T \end{cases} \tag{2.1-2}$$

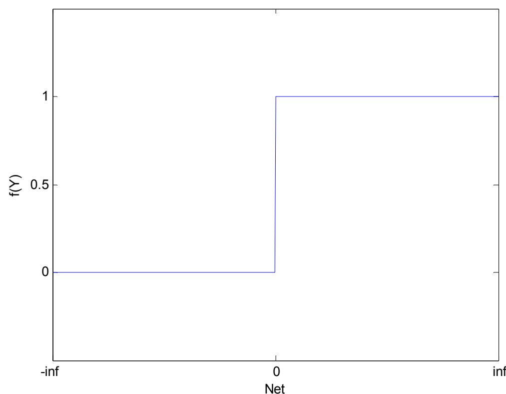
$$O_1 = \begin{cases} 1 & if\ Y \geq T \\ -1 & if\ Y < T \end{cases} \tag{2.1-3}$$

Hard activation functions emulate the all-or-nothing firing mechanism of biologic neurons and can implement basic logic functions such as AND, OR, and NOT, as well as solve other small problems with at most a single layer of neurons. Neurons with hard activation functions present a problem for training multiple layers in that they are opaque. Incremental changes to the

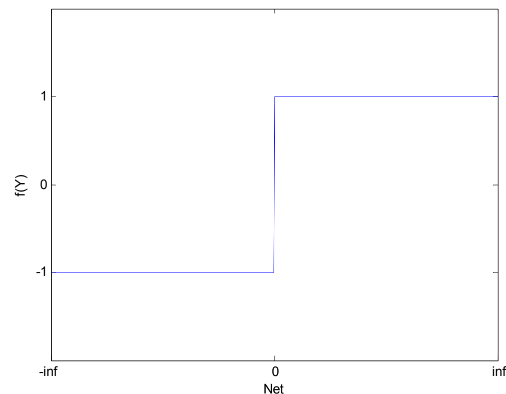
weights do not have a corresponding effect on the output. This is why “soft” sigmoidal activation functions are used in place of step functions. Sigmoidal activation functions behave the same in the limit as step functions, and can also be unipolar (2.1-4) or bipolar (2.1-5), but have a gradual transition that makes the neuron (or network) transparent for training, which can be seen in Figure 2.1.2. Note that $\lambda$ is a constant that controls the slope of the activation function.

$$O_1 = \frac{1}{1 + exp(-\lambda Y)} \tag{2.1-4}$$

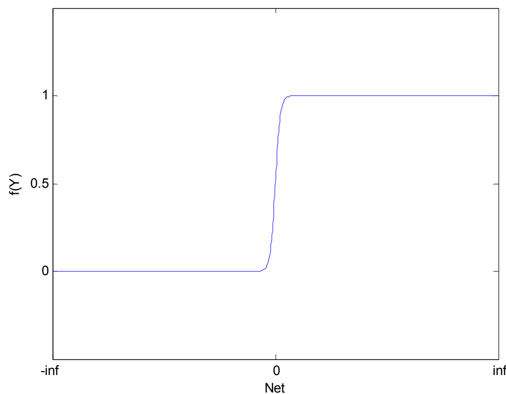
$$O_1 = \frac{2}{1 + exp(-\lambda Y)} - 1 \tag{2.1-5}$$



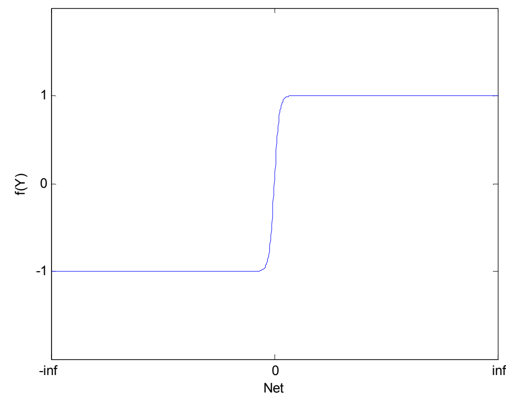

(a)




(b)




(c)




(d)

Figure 2.1.2: Typical activation functions. a) Hard unipolar. b) Hard bipolar. c) Soft unipolar. d) Soft bipolar.

As with biological neurons, a single artificial neuron is not particularly useful, and must be connected together in networks before any kind of learning can take place. The discussion here will be limited to feedforward networks, although there are special types of networks that make use of recurrent connections, such as Hopfield Networks[25]. Figure 2.1.3 shows a typical artificial neural network configured in a Multi-Layer Perceptron (MLP) architecture. This architecture has a number of what are called "hidden layers" in between the inputs and the network output.

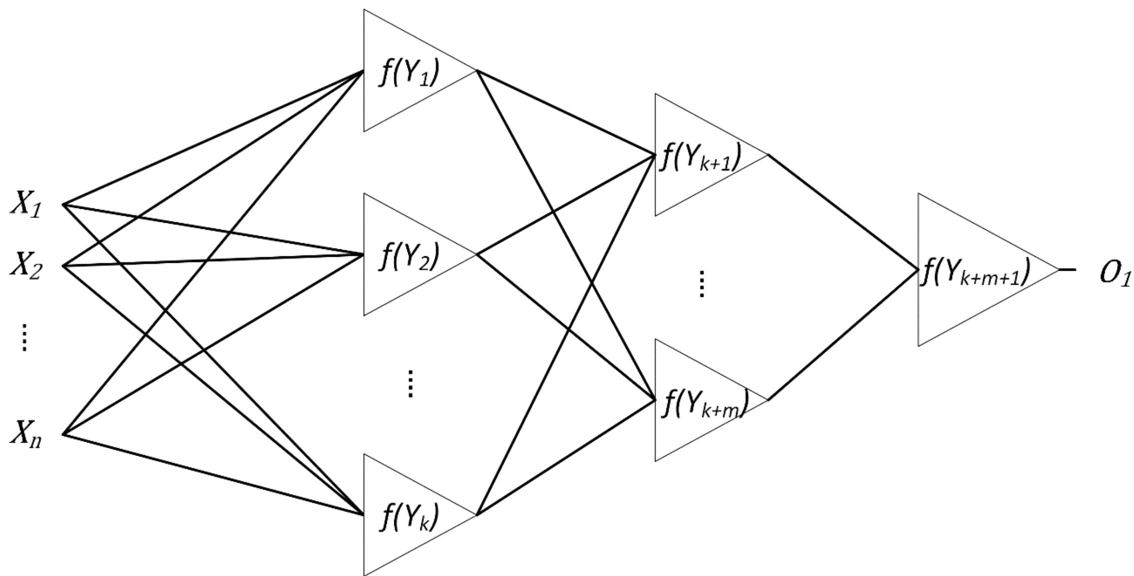
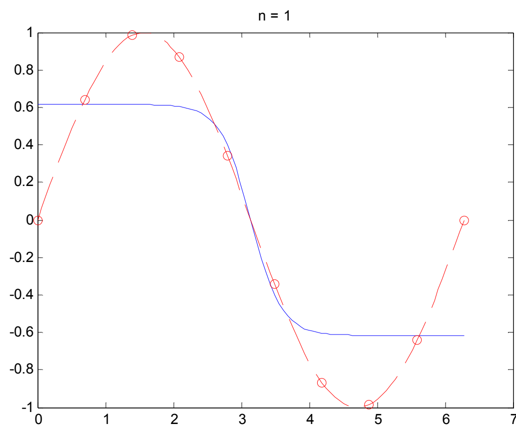



Figure 2.1.3: A Multi-Layer Perceptron artificial neural network.

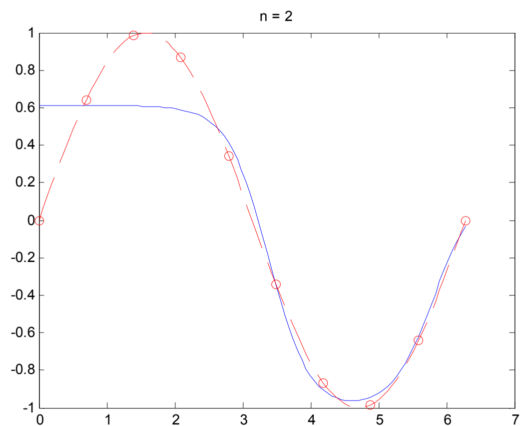
Artificial neural networks require several design decisions to be made, specifically: what architecture will be used, how large will the network be, and what algorithm will be used for training. These decisions will have an enormous impact on the network performance.

### 2.1.1 Neural Network Size

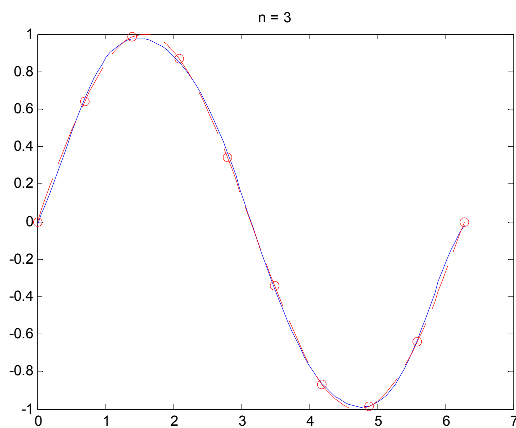
The size (number of neurons) and their configuration (the architecture or topology) have a direct impact the power of a neural network. Intuitively, the more neurons in a network, the more complex problems it can solve. This is easily confirmed experimentally in Figure 2.1.4(a)-(f) where a simple sinusoid is approximated by an ANN with increasing numbers of neurons. As the number of neurons increases, the output of the network more closely matches the desired sinusoid.



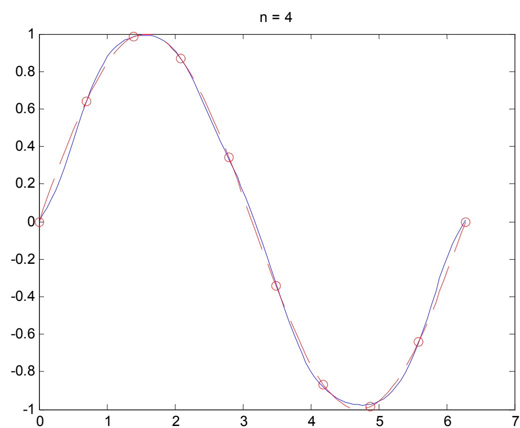

(a)



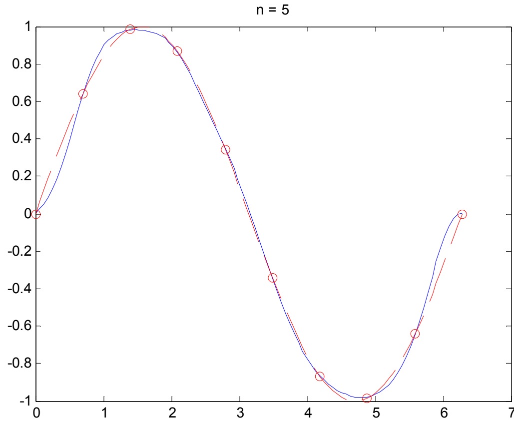

(b)



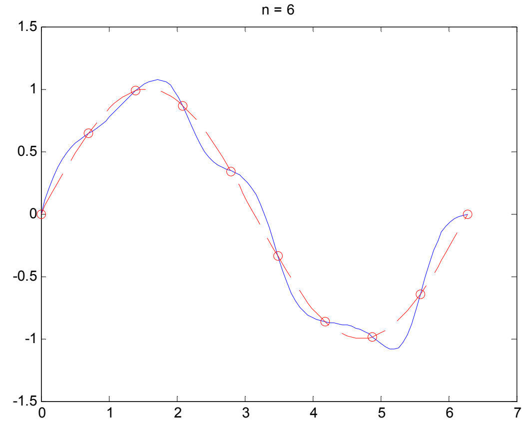

(c)




(d)




(e)




(f)

Figure 2.1.4: Results of training a neural network to approximate a sinusoid as number of neurons $n$ increases. a) $n = 1$ b) $n = 2$ c) $n = 3$ d) $n = 4$ e) $n = 5$ f) $n = 6$

However, it is readily apparent that the best results are produced by the network with three neurons, and adding more past that causes undesirable oscillations between the training points. These are similar to the oscillations that occur when using a high order polynomial for interpolation. Past the optimal number of neurons, the network will match the training points very closely, but the generalization ability suffers, as shown by the errors in **Table I**. Notice that the training errors decrease as more neurons are added, with the largest network producing the smallest training error. The testing errors verify what can be seen visually in Figure 2.1.4, with the errors decreasing as more neurons are added, reaching a minimum with three neurons, and then beginning to increase.

**Table I**: Tabulated training and testing errors as the number of neurons increases.

| n | Training RMSE | Testing RMSE |
|---|---|---|
| 1 | 0.3431 | 0.2736 |
| 2 | 0.2454 | 0.2000 |
| 3 | 0.0153 | 0.0219 |
| 4 | 0.0086 | 0.0327 |

| | | |
|---|---|---|
| 5 | 0.0039 | 0.0469 |
| 6 | 5.6417e-04 | 0.1078 |

In general, the best neural network results will be produced by a network that uses the fewest number of neurons possible while still meeting a minimum training error threshold[17], [19]. Such compact networks maintain good generalization capabilities when presented with inputs that were not used in training. The tradeoff for this performance is that smaller networks are harder to train than an overprovisioned network, and can even fail to converge.

### 2.1.2 Neural Network Architecture

In theory, there are an infinite number of ways to connect a given number of neurons, some subset of which will be optimal for a particular problem. In practice, there are several reasons why standardized architectures are preferred. First, not all training algorithms can handle arbitrarily connected networks. Second, standardized architectures make evaluating the effects of different network sizes and initial starting weights feasible. A comparison of the power of various architectures was done in [17]. The different architectures were evaluated based on their ability to solve parity-N problems. This problem has been demonstrated as an effective benchmark for comparing neural network efficiency[26]. The parity-N problem is a generalization of the XOR problem for *n* inputs, where the output is zero if there are an even number of one's in the input string, and one if there is an odd number. **Table II** shows the input patterns for the parity-3 problem.

**Table II**: Input patterns for the parity-3 problem.

| Binary Input | Output |
|---|---|
| 000 | 0 |
| 001 | 1 |
| 010 | 1 |
| 011 | 0 |
| 100 | 1 |
| 101 | 0 |

| 110 | 0 |
| --- | --- |
| 111 | 1 |

The Multilayer Perceptron architecture shown in the previous section (Figure 2.1.3) is the simplest architecture, and also one of the most popular. The defining property of an MLP network is that the neurons are organized into discreet layers, with no cross layer connections. Although there can be any number of layers, the special case of one hidden layer (known as a Single Layer Feedforward Network (SLFN)) shown in Figure 2.1.5 is widely used. The largest parity problem that can be solved by a SLFN is given by (2.1-6), where *k* is the number of neurons, and *N* is the degree of the parity problem.

$$N = k - 1 \tag{2.1-6}$$

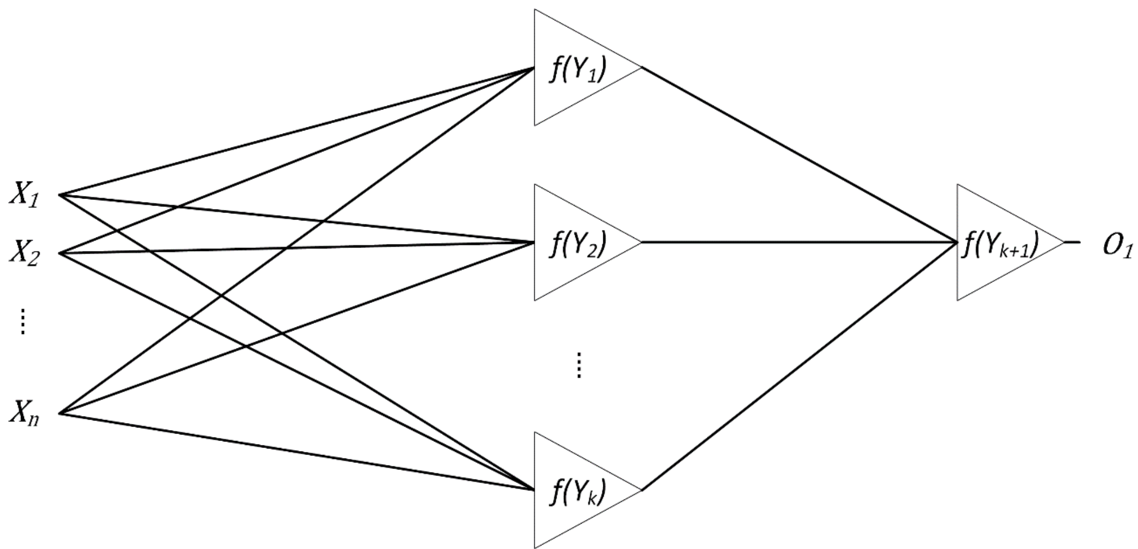



Figure 2.1.5: A Single Layer Feedforward Network (SLFN).

If cross layer connections are allowed, the Multilayer Perceptron becomes the Bridged Multilayer Perceptron (BMLP) architecture. A BMLP network with a single hidden layer is shown in Figure 2.1.6. The cross layer connections of the BMLP make it more powerful than the MLP, requiring fewer neurons to solve the same parity-N problem (2.1-7).

$$N = 2k - 1 \tag{2.1-7}$$

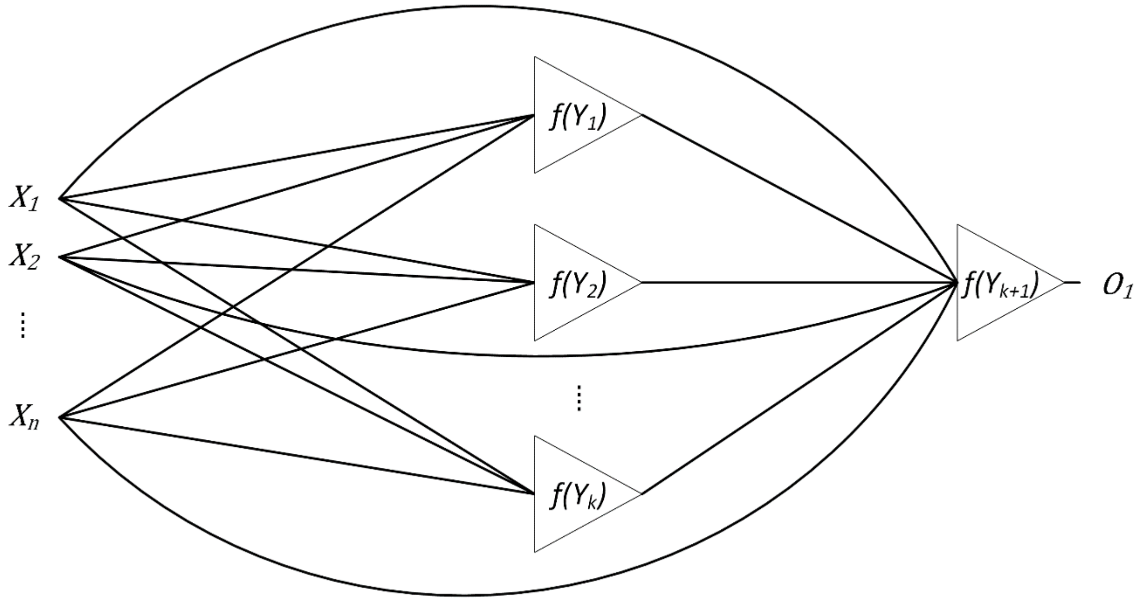



Figure 2.1.6: A BMLP network with a single hidden layer.

Multiple hidden layers can also be used in the BMLP architecture, and the network diagrams can quickly become difficult to follow, as shown in Figure 2.1.7, which only has two hidden layers. For a BMLP network with two hidden layers, the largest parity problem that can be solved is given by (2.1-8).

$$N = 2(k + 1)(m + 1) - 1 \tag{2.1-8}$$

For a BLMP network with an arbitrary number of hidden layers, the expression is a little more complex. The largest parity problem a network with *p* hidden layers can solve is shown by (2.1-9), where $n_i$ is the number of hidden neurons in the *i*th hidden layer.

$$N = 2\left(\prod_{i=1}^{i=p}(n_i + 1)\right) - 1 \tag{2.1-9}$$

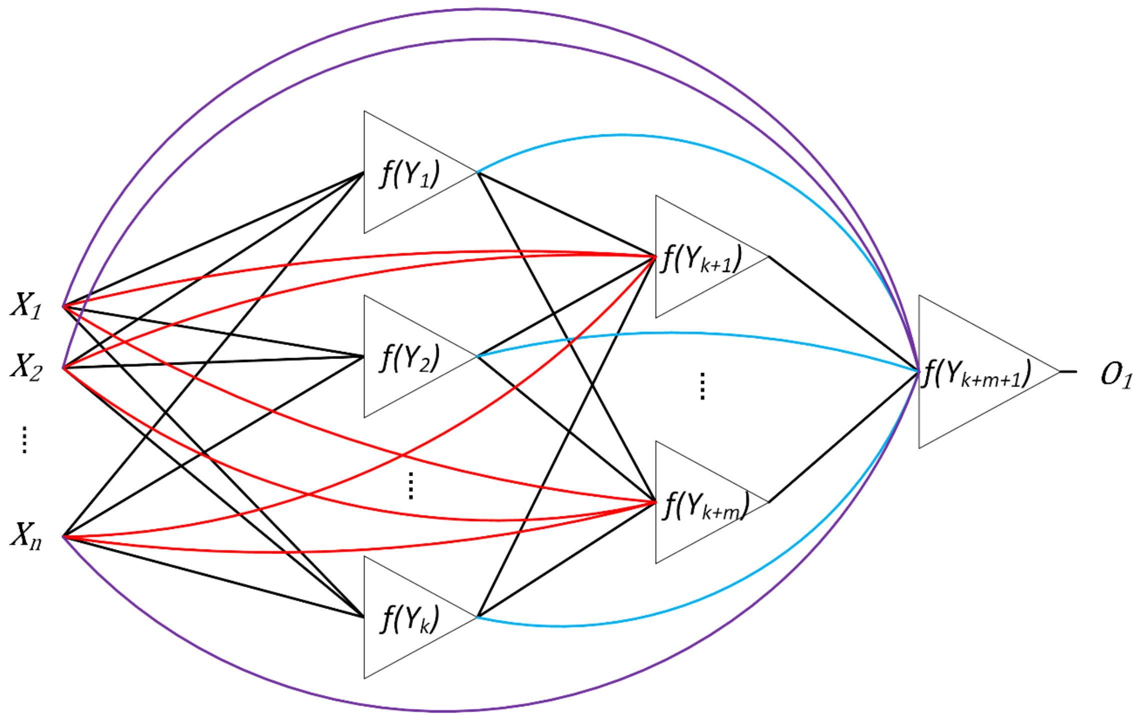



Figure 2.1.7: A BMLP network with two hidden layers.

The most powerful network architecture evaluated is the Fully Connected Cascade (FCC) architecture. Shown in Figure 2.1.8, the FCC can be thought of as a special case of the BMLP where each hidden layer only has a single neuron. This gives an FCC network the maximum possible depth with a minimum total number of neurons. The highest order parity-N problem an FCC network with *n* neurons can solve is given by (2.1-10).

$$N = 2^k - 1 \qquad (2.1\text{-}10)$$

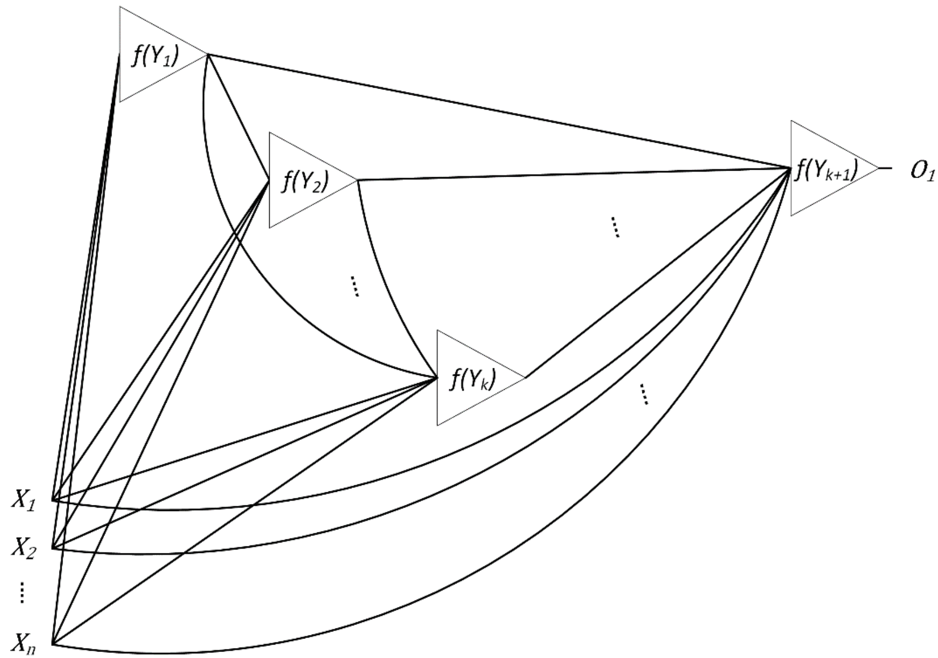



Figure 2.1.8: A FCC network with *n* inputs and *k* neurons.

Some concrete numbers can help demonstrate the difference in power between these architectures. **Table III** shows the maximum parity problem that can be solved by the various architectures if we fix the network size at eight neurons. Note that BMLP (4-4) indicates that the network has two hidden layers with four neurons each, and BMLP (2-2-2-2) has four hidden layers with two neurons each. It should be clear that the difference in power between the network topologies is substantial, with the FCC network able to solve problems of much greater complexity than simple MLP networks.

**Table III**: Comparing the largest parity problem that can be solved by various architectures with eight neurons.

| Architecture | Parity-N |
|---|---|
| MLP | 7 |
| BMLP (4-4) | 15 |
| BMLP (2-2-2-2) | 161 |
| FCC | 255 |

### 2.1.3 Neural Network Training

For a small number of neurons, it is possible to design the weights to create the desired output. Designing quickly becomes impossible as network size increases, and some form of training is necessary to automatically adjust the weights to meet some criteria. Learning methods can be broadly classified as supervised, where each training pattern has a desired output, or unsupervised, where the training patterns do not have desired outputs. The goal of unsupervised learning is typically to explore a dataset and perform clustering, such as with a Kohonen Self Organizing Map (SOM)[27]. The focus here will be on supervised learning.

In supervised learning, each input pattern has a desired output. When a pattern is applied to the network, the actual output will, in general, be different from the desired output. The optimal network weights for that pattern will be the set of weights that minimizes the error between the desired and actual outputs. Thus, training the network can be reformulated to an optimization problem, with the goal of minimizing the total error for all patterns. Assume there are $P$ total training patterns, and the $p$th pattern has the form of (2.1-11), with $n$ input dimensions and the desired output $d$.

$$x_p = \{(x_{p,1}, x_{p,2}, \dots, x_{p,n}) \mid d_p\} \tag{2.1-11}$$

After applying the $p$th pattern, the error $e_p$ for that pattern is computed as the difference between the desired output $d_p$ and the actual output $o_p$ (2.1-12).

$$e_p = d_p - o_p \tag{2.1-12}$$

Since the goal of neural network training is to minimize the error for all training patterns, a measure for the total error is needed. The Sum Squared Error (SSE), abbreviated as simply E,

shown in (2.1-13), is widely utilized. Note that the one half factor is there to cancel the power during differentiation later, and could be omitted.

$$E = \frac{1}{2}\sum_{p=1}^{P}\left(d_p - o_p\right)^2 \tag{2.1-13}$$

The training algorithms described in the following sections are all based on some form of gradient descent optimization to find the minimum SSE.

#### *2.1.3.1 Error Backpropagation*

Early training algorithms were limited to either a single neuron or a single layer of neurons (as in no hidden layers). As a result, neural networks were very limited in ability and application. The first algorithm that could train networks with one or more hidden layers was Error Backpropagation (EBP). Originally described in 1974 by Werbos [28], EBP was first applied to ANN training by Rumelhart et al. [29] in 1986.

EBP is an iterative algorithm, with each iteration consisting of three steps. The first consists of forward computation, where a training pattern is applied to the network inputs. The net values for each neuron in the first layer are computed, and the output of each neuron is calculated using the activation function. Then the outputs of the first layer are used as the inputs to the next layer, and the process is repeated until the final output of the network is obtained. In the second step, the error between the desired output and the actual output is found. The error then propagates back through the network (the titular backpropagation), by calculating the partial derivative of the total error with respect to each weight. The third and final step calculates the change in each network weight using the input pattern and the partial error terms.

Error Backpropagation is a steepest descent algorithm. As such, the weight update rule can be written as (2.1-14).

$$\boldsymbol{w}^* = \boldsymbol{w} - \alpha \boldsymbol{g} \tag{2.1-14}$$

Where $\boldsymbol{w}$ is the vector of network weights, $\boldsymbol{w}^*$ is the updated weight vector for the next iteration, $\alpha$ is a learning constant, and $\boldsymbol{g}$ is the gradient of the error function taken with respect to the weights, and has the form of (2.1-15).

$$\boldsymbol{g} = \frac{\partial E(\boldsymbol{x}, \boldsymbol{w})}{\partial \boldsymbol{w}} = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_N} \end{bmatrix} \tag{2.1-15}$$

Finding the partial derivative values for the gradient is the difficult part of the algorithm. We will assume a SLFN network with one output, such as the one shown in Figure 2.1.5. Furthermore, we shall refer to the set of weights between the inputs and hidden layer as $\boldsymbol{u}$**,** the weights between the hidden layer and the output layer as $\boldsymbol{v}$, and output of the neurons in the hidden layer as $f(\boldsymbol{Y})$.

Recall that the goal is to minimize the error function. Evaluating (2.1-13) for an input pattern $x_p$ and expanding yields (2.1-16).

$$E(\boldsymbol{x_p}, \boldsymbol{w}) = \frac{1}{2}(d^2 - 2do + o^2) \tag{2.1-16}$$

The partial derivative of (2.1-16) with respect to the output weights $\boldsymbol{v}$ gives (2.1-17).

$$\frac{\partial E(\boldsymbol{x_p}, \boldsymbol{v})}{\partial \boldsymbol{v}} = -(d - o)f'\big(\boldsymbol{v}f'(\boldsymbol{Y})\big)f(\boldsymbol{Y}) \tag{2.1-17}$$

It is convenient to define a term $\delta$ to be the partial error (2.1-18), so that the weight change in for the output layer can be written as (2.1-19).

$$\delta = (d - o)f'(\boldsymbol{vy}) \tag{2.1-18}$$

$$\Delta \boldsymbol{v} = -\frac{\partial E(\boldsymbol{x_p}, \boldsymbol{v})}{\partial \boldsymbol{v}} = \alpha \delta \boldsymbol{Y} \tag{2.1-19}$$

This process repeats for the weights in the input layer (2.1-20)

$$\Delta \boldsymbol{u} = -\frac{\partial E(\boldsymbol{x_p}, \boldsymbol{u})}{\partial \boldsymbol{u}} = \alpha \delta \boldsymbol{v} f'(\boldsymbol{u}\boldsymbol{x_p})\boldsymbol{x_p} \tag{2.1-20}$$

This process is either repeated for each training pattern, with the weights either being updated after each pattern is applied, or held constant until all patterns are applied, and the sum of all weight changes used to update the weights.

As a first order gradient descent algorithm, EBP is stable, but suffers from slow convergence and a tendency to get stuck in local minima. Modifications of EBP such as adaptive learning rate [30], momentum [31], and Resilient Error Backpropagation (RPROP)[32] seek to address one or both of these issues. However, even with these modifications EBP is simply not powerful enough to solve some problems, requires a large number of neurons, or produces very poor results when tested with patterns that were not used for training. Furthermore, EBP can only train MLP networks, which are not very powerful. More powerful algorithms that can train arbitrarily connected networks are needed.

#### *2.1.3.2 Levenberg-Marquardt Approach*

The goal of a gradient descent algorithm is to find the global minimum of an error function. This process can be visualized as attempting to find the lowest point in a valley while only being able to see one step in any direction. Getting to the bottom of the valley then becomes a series of "steps", and the direction and size of each step must be chosen. In a first order algorithm like EBP, the direction of the next "step" is selected to be whichever is the steepest. This is a locally greedy heuristic, and as such can cause the algorithm to get stuck in local minima. EBP can also become trapped in flat spots where the gradient is very small. The size of each step is determined by the learning constant $\alpha$. A larger learning constant produces a larger change in the network weights for a given iteration, and makes the algorithm less likely to become trapped in local

minima. However, larger values also cause the algorithm to become less stable, and may prevent it from ever reaching the global minimum by causing it to always step from one side of the "valley" to the other. A smaller learning constant causes slower convergence, but is more stable. Fast, stable convergence therefor requires optimal selection of the learning constant at each iteration. First order methods do not have enough information about the surface of the error function to optimally choose the learning constant at each step. Towards that end, second order methods are needed.

Newton's method is the starting point of second order training algorithms. In Newton's method, the weight update rule of steepest descent in (2.1-14) is modified to (2.1-21).

$$\boldsymbol{w}^* = \boldsymbol{w} - \boldsymbol{H}^{-1}\boldsymbol{g} \tag{2.1-21}$$

Where $\boldsymbol{H}$ is the Hessian matrix of the form in (2.1-22).

$$\boldsymbol{H} = \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_N} \\ \frac{\partial^2 E}{\partial w_2 \partial w_1} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_2 \partial w_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 E}{\partial w_N \partial w_1} & \frac{\partial^2 E}{\partial w_N \partial w_2} & \cdots & \frac{\partial^2 E}{\partial w_N^2} \end{bmatrix} \tag{2.1-22}$$

Newton's method gives fast convergence, but calculating the second derivatives of the error function for the Hessian matrix can be computationally infeasible. Newton's method is also unstable in some cases. In order to reduce the computational complexity, the Gauss-Newton method introduces the Jacobian matrix (2.1-23). Note that $P$ refers to the number of patterns, and $M$ refers to the number of outputs.

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w_1} & \frac{\partial e_{1,1}}{\partial w_2} & \cdots & \frac{\partial e_{1,1}}{\partial w_N} \\ \frac{\partial e_{1,2}}{\partial w_1} & \frac{\partial e_{1,2}}{\partial w_2} & \cdots & \frac{\partial e_{1,2}}{\partial w_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_{1,M}}{\partial w_1} & \frac{\partial e_{1,M}}{\partial w_2} & \cdots & \frac{\partial e_{1,M}}{\partial w_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_{P,1}}{\partial w_1} & \frac{\partial e_{P,1}}{\partial w_2} & \cdots & \frac{\partial e_{P,1}}{\partial w_N} \\ \frac{\partial e_{P,2}}{\partial w_1} & \frac{\partial e_{P,2}}{\partial w_2} & \cdots & \frac{\partial e_{P,2}}{\partial w_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_{P,M}}{\partial w_1} & \frac{\partial e_{P,M}}{\partial w_2} & \cdots & \frac{\partial e_{P,M}}{\partial w_N} \end{bmatrix} \tag{2.1-23}$$

The gradient is related to the Jacobian by (2.1-24). The vector $\boldsymbol{e}$ has the form of (2.1-25).

$$\boldsymbol{g} = \boldsymbol{J}\boldsymbol{e} \tag{2.1-24}$$

$$\boldsymbol{e} = \begin{bmatrix} e_{1,1} \\ e_{1,2} \\ \vdots \\ e_{1,M} \\ \vdots \\ e_{P,1} \\ e_{P,2} \\ \vdots \\ e_{P,M} \end{bmatrix} \tag{2.1-25}$$

The Hessian matrix can be approximated by (2.1-26).

$$\boldsymbol{H} \approx \boldsymbol{J}^T\boldsymbol{J} \tag{2.1-26}$$

Combining (2.1-21), (2.1-24), and (2.1-26), the weight update rule for the Gauss-Newton method and can be written as (2.1-27).

$$\boldsymbol{w}^* = \boldsymbol{w} - (\boldsymbol{J}^T\boldsymbol{J})^{-1}\boldsymbol{J}\boldsymbol{e} \tag{2.1-27}$$

The Gauss-Newton method does not require the second derivative calculations of the Newton method, but still faces convergence problems. Specifically, the approximation of the Hessian, $\boldsymbol{J}^T\boldsymbol{J}$, is not guaranteed to be invertible.

The Levenberg-Marquardt (LM) algorithm[33] modifies the Gauss-Newton method by replacing the Hessian approximation in (2.1-26) with (2.1-28). Note that $\mu$ is a positive constant called the combination coefficient and $\boldsymbol{I}$ is the identity matrix.

$$\boldsymbol{H} \approx \boldsymbol{J}^T\boldsymbol{J} + \mu\boldsymbol{I} \tag{2.1-28}$$

The addition of the $\mu\boldsymbol{I}$ term ensures that the diagonal of the Hessian approximation will always be non-zero, and thus invertible. Finally, the weight update rule for the LM algorithm can be seen in (2.1-29).

$$\boldsymbol{w}^* = \boldsymbol{w} - (\boldsymbol{J}^T\boldsymbol{J} + \mu\boldsymbol{I})^{-\mathbf{1}}\boldsymbol{J}\boldsymbol{e} \tag{2.1-29}$$

First applied to neural network training in [34], the LM algorithm combines the fast convergence of the Gauss-Newton method with the stability of EBP. LM training has many advantages, however, there are still problems. The Jacobian matrix that must be stored is of size $(P \times M) \times N$, where $P$ is the number of patterns, $M$ is the number of outputs, and $N$ is the number of weights. For large training sets and/or network sizes, the Jacobian can become too large for the available memory, and the speed gains over EBP will be lost. Furthermore, the LM algorithm can only be applied to certain network architectures, such as MLP. The recently developed Neuron-by-Neuron (NBN) algorithm[19] offers several advantages over LM. First, NBN can train arbitrarily connected networks, which is important if efficient architectures are to be used. Second, NBN utilizes a novel computation scheme that avoids ever having the entire Jacobian matrix in memory at one time.

### 2.1.4 Specialized Neural Networks

This section will briefly describe several machine learning techniques that are related to neural networks, but differ from classic neural networks in key ways.

### *2.1.4.1 Radial Basis Function Networks*

Moody and Darken [35] first showed that Radial Basis Function (RBF) networks were universal approximators. RBF networks differ from traditional neural networks in that instead of neurons with sigmoidal activation functions, RBF networks are composed of RBF units with one of several activation functions, typically a multidimensional Gaussian function such as (2.1-30). The vector $\boldsymbol{x}$ is the multidimensional input, $\boldsymbol{c}$ is a vector with the location of the center of the Gaussian for each dimension, $\sigma$ is a parameter that controls the width, and the gain $\beta$ controls the height of the Gaussian.

$$f(\boldsymbol{x}) = \beta \exp\left(-\left(\frac{\|\boldsymbol{x}-\boldsymbol{c}\|}{\sigma^2}\right)\right) \tag{2.1-30}$$

RBF networks typically use a SLFN architecture, shown in Figure 2.1.9. The output layer of an RBF network is usually a summation, rather than another RBF unit.

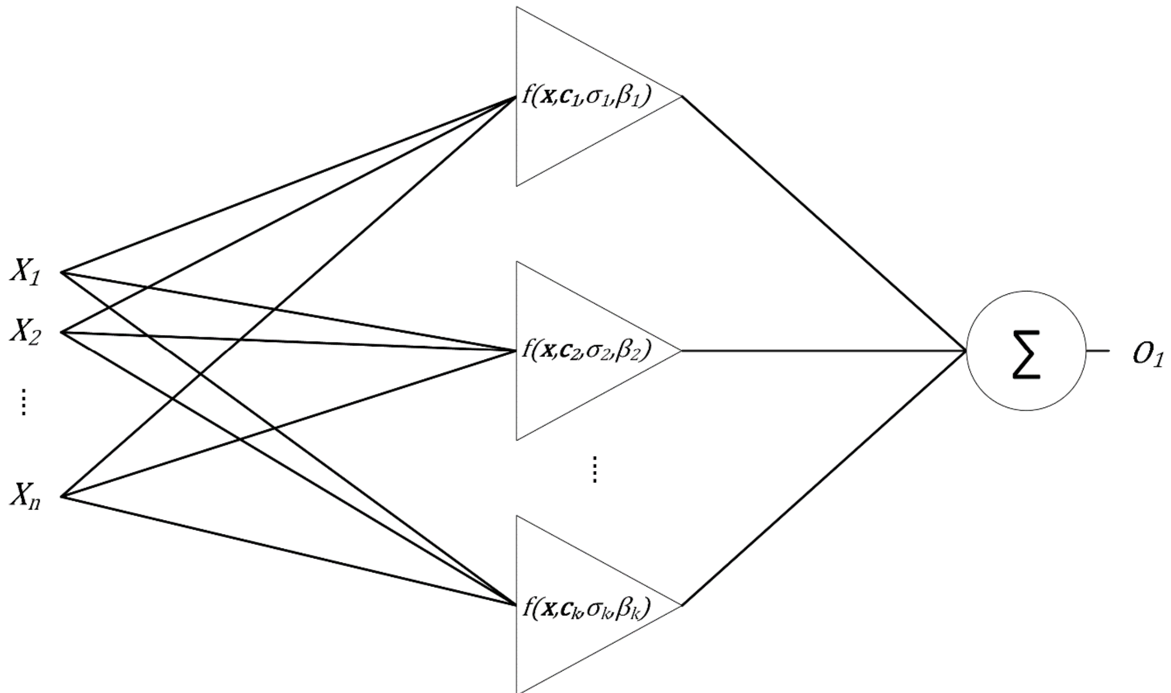



Figure 2.1.9: A SLFN RBF network with *k* RBF units and a single output.

In contrast to finding the weights in classic ANN, the goal of training RBF networks is to find optimal values for $\boldsymbol{c}, \sigma,$ and $\beta$ for each RBF unit. Because the number of tunable parameters is larger than in a traditional ANN of the same size, developing training algorisms is difficult. Typically, only first order algorithms have been adapted to train RBF networks, although recent advances have applied second order LM training to RBF networks[15].

#### *2.1.4.2 Extreme Learning Machines*

Extreme Learning Machines (ELM) are a relatively recent development. Originally published by Huang et al. [36]–[38], ELM are a type of RBF network. ELM sidestep the difficulties of training RBF networks by randomly generating (and leaving fixed) the $\sigma$ and $\boldsymbol{c}$ parameters for each RBF unit, and then solving for the $\beta$ by simple pseudo-inversion. ELM have very fast training times and are capable of producing surprisingly good results considering only one parameter is trained. However, the fast training times of ELM are mitigated by the fact that many trials may be necessary before acceptable results are obtained. In addition, ELM require far larger networks to reach similar error levels vs. other algorithms that take advantage of all trainable parameters.

#### *2.1.4.3 Support Vector Machines*

Support Vector Machines (SVM) [39] are an attempt to perform learning on a dataset using a minimum number of nodes. This is done by selecting and training only the most essential patterns, and using those for training. This is done by performing optimization on a more complex cost function than the one used in classic ANN. For a detailed description of SVM, see [40]. SVM can use a variety of kernels such as sigmoid or Gaussian. A trained SVM with a Gaussian kernel is functionally equivalent to an RBF network. While SVM are capable of producing very good results, they require the user to provide several parameters that will greatly

impact the performance. Searching for appropriate values for these parameters is very time consuming, requiring trial and error.

## 2.2 Fuzzy Systems

Fuzzy logic can be thought of as a generalization of classical set theory. First proposed by Zadeh[41], Fuzzy logic attempts to formalize and account for the uncertainty inherent in the real world. In Boolean logic (and in classical set theory on which it is based), the only allowed values are zero and one. An expression is either true or false; an item either belongs to a set or it does not. As the foundation of digital computing, this kind of reasoning is powerful, but also has clear limitations. At its heart, set theory relies on certainty. It requires that there be no ambiguity in observations or measurements, which is not a problem in the abstract realm of theory. But we intuitively understand that uncertainty and ambiguity exist intrinsically in the real world. Fuzzy logic models this by allowing a continuous range of values between zero and one.

There are a few important concepts necessary to understanding the Fuzzy Systems described in the next section. The first are the fuzzy version of the AND, OR, and NOT operations, which are defined by (2.2-31), (2.2-32), and (2.2-33), respectively.

$$A \cap B = \min(A, B) \tag{2.2-31}$$

$$A \cup B = \max(A, B) \tag{2.2-32}$$

$$\bar{A} = 1 - A \tag{2.2-33}$$

The second is the idea of the membership function. A membership function defines a fuzzy set by taking in an input $x$ and producing a value indicating degree to which $x$ belongs to the set. Notice that in the limit, where $x$ either fully belongs to a set or fully does not, fuzzy logic reduces to Boolean logic. The range of a variable can be divided into any number of membership functions. These functions can overlap as long as the sum never exceeds one. Typically,

trapezoidal, triangular, or Gaussian membership functions are used. The process of computing the membership values of a crisp (or non-fuzzy) value $x$ for each membership function is called fuzzification. The reverse process of taking a set of membership values and producing a crisp output is called defuzzification.

### 2.2.1 TSK Fuzzy Systems

Fuzzy Systems (FS) refers to a family of techniques that utilize fuzzy logic to perform classification, approximation, and control. Mamdani[42] published the first example of a FS applied to controlling a steam engine. This idea was improved upon in the Takagi-Sugeno-Kang (TSK) FS [20], which generally produces better results. The general structure of any FS can be seen in Figure 2.2.1.

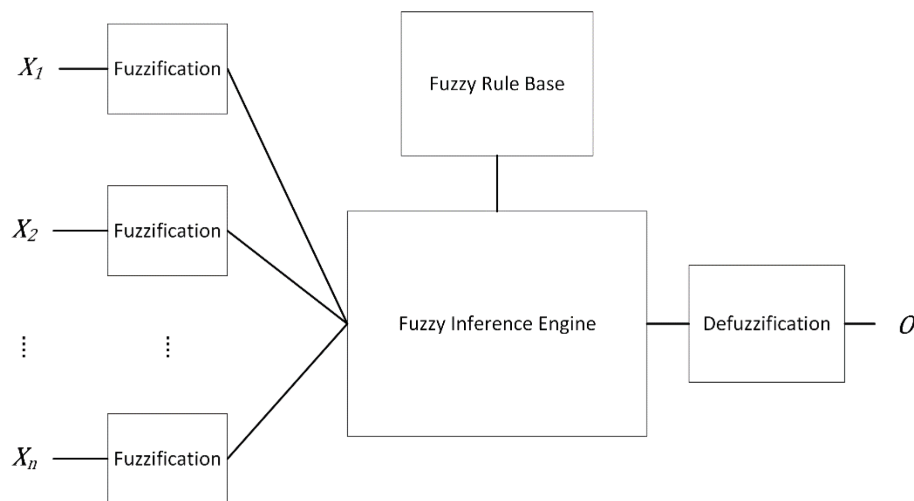



Figure 2.2.1: The basic component of any fuzzy system.

The inputs $X_1, X_2, \dots X_n$ are crisp (or non-fuzzy) values. These crisp values are passed through a fuzzification step. After fuzzification, the fuzzy values are passed to the inference engine. The inference engine evaluates the rule base using the fuzzy inputs and aggregates the results. Formally, the rule base is a series of IF-THEN statements that produce a certain output when activated, as shown in (2.2-34). Note that $m_i(X_1)$ indicates the output of the $i$th

membership function of input $X_1$, $m_j(X_2)$ is the output of the *j*th membership function of input $X_2$, and so on. The value C can be a constant or a function of the inputs depending on the system being used.

$$IF\ m_i(X_1)\ AND\ m_j(X_2)\ AND\ \dots m_k(X_n)\ THEN\ O = C \tag{2.2-34}$$

The last step is to pass the results of the inference engine through a defuzzifier to produce a single, crisp output.

One major difference between FS and ANN is that FS are designed, while ANN are trained. By adjusting the number and shape of membership functions as well as the rule base, the desired output can be obtained. The design process can take into account expert knowledge and intuition, but generally requires a good deal of trial and error. Broadly speaking, FS are fast compared to neural networks, at least in cases where there are only a few input dimensions. Neural Networks are trainable and do not have a limitation on the number of input dimensions, but they suffer from convergence problems. One trend involves applying optimization techniques to the design of FS—essentially leading to trainable FS, often referred to as neuro-fuzzy systems.

### 2.2.2 Adaptive Network-based Fuzzy Inference Systems

One of the foundational ANN FS hybrid techniques is called Adaptive Network-based Fuzzy Inference Systems (ANFIS). Proposed by Jang[43], [44], ANFIS is essentially a TSK FS with Gaussian membership functions and linear rule consequents. In ANFIS, the parameters of the membership functions are tuned using a gradient descent algorithm. This hybridization gives ANFIS the potential to maintain the human interpretability of a fuzzy system, while being able to learn from training data in a manner similar to ANN. In a later paper, Jang proved that ANFIS and RBF networks are functionally equivalent [45]. Nevertheless, ANFIS is a powerful learning technique.

# Chapter 3 Nearest Neighbor Spline Approximation

Before describing the Nearest Neighbor Spline Approximation (NNSA) algorithm, some review is in order. As the name implies, NNSA relies on splines. So we will first, briefly, review polynomial interpolation as an introduction to splines, move onto a description of splines, and then present the NNSA algorithm.

## 3.1 Polynomial Approximation Review

Polynomial interpolation has long been used to approximate functions from known data points. This technique is useful if the underlying function is unknown, or too computationally complex to be used directly. One historical example is the computation of trigonometric functions. For hundreds of years, the main technique for computing values for trigonometric functions relied on precomputed tables of values for known angles, and then applying linear interpolation to obtain the function output for the desired input. Modern processors use a variety of techniques to compute trigonometric functions, but use of look up tables and interpolation remain common, especially in applications where speed is essential.

### 3.1.1 Polynomial Approximation Derivation

Polynomials have a known and predictable form. For a polynomial $p$ of degree $n$, the form is defined by (3.1-35)

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x^1 + a_0 \quad (3.1\text{-}35)$$

So, the problem of polynomial interpolation is one of solving for the values of the unknown coefficients. More formally, given a set of $n + 1$ data points of the form $(x_i, y_i)$, find the polynomial $p$ with degree of $n$ which satisfies the interpolation condition (3.1-36).

$$p(x_i) = y_i \quad , \qquad i = 0 \ldots n \quad (3.1\text{-}36)$$

The interpolation condition means that the polynomial must be pass through the data points. This also ensures that the solution will be unique. In order to solve for the coefficients, the known polynomial form (3.1-35), and the data points are inserted into (3.1-36) to produce a system of equations shown in (3.1-37).

$$\begin{bmatrix} x_0^n & x_0^{n-1} & \cdots & x_0^1 & 1 \\ x_1^n & x_1^{n-1} & \cdots & x_1^1 & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ x_n^n & x_n^{n-1} & \cdots & x_n^1 & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} \tag{3.1-37}$$

This system of equations has $n$ unknowns and $n$ equations, meaning it has a unique solution that can be found by simple pseudo-inversion.

### 3.1.2 Polynomial Approximation Problems

This process works well in certain situations, as demonstrated by the example in Figure 3.1.1. On the left, we can see the desired function, a simple sinusoid, and seven evenly spaced data points to interpolate. On the right is the resulting curve produced by polynomial interpolation. Since seven data points were used, then the interpolating polynomial is sixth order.

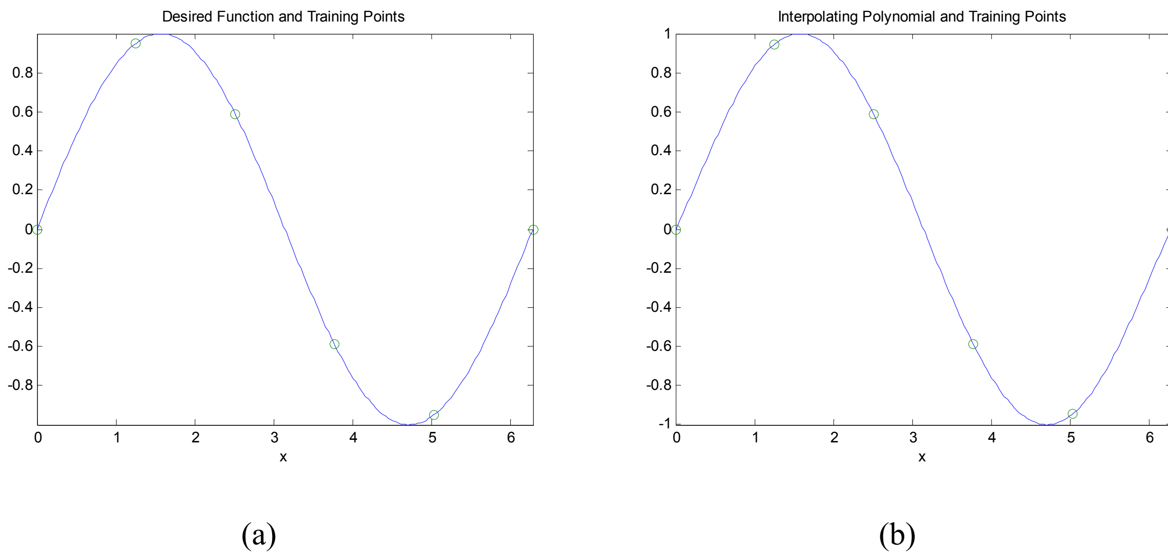



Figure 3.1.1: Polynomial interpolation example. (a) The desired function $y = sin(x)$. (b) The interpolating polynomial.

On first inspection, the results look fairly good, as it is hard to visually distinguish between the two curves. Upon further inspection, several problems become apparent. First, the difference between the curves is greater than it first appears, as shown in Figure 3.1.2.

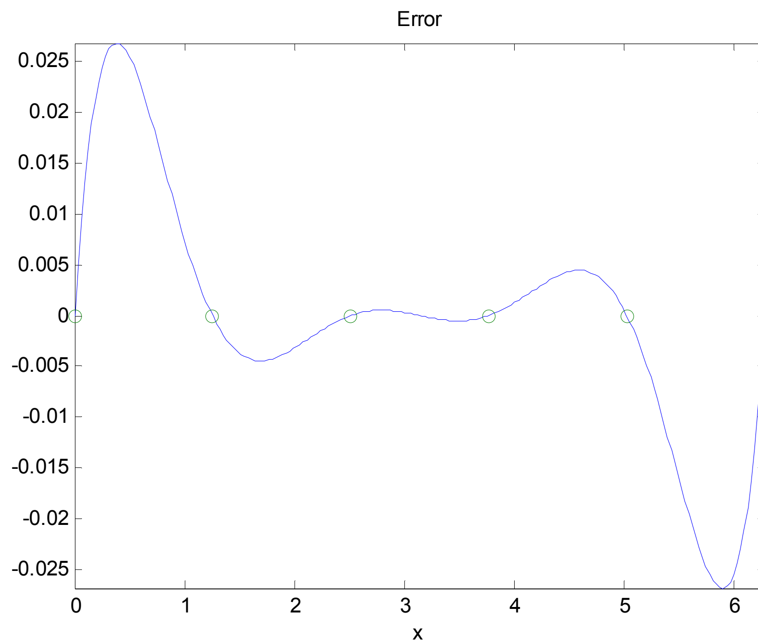
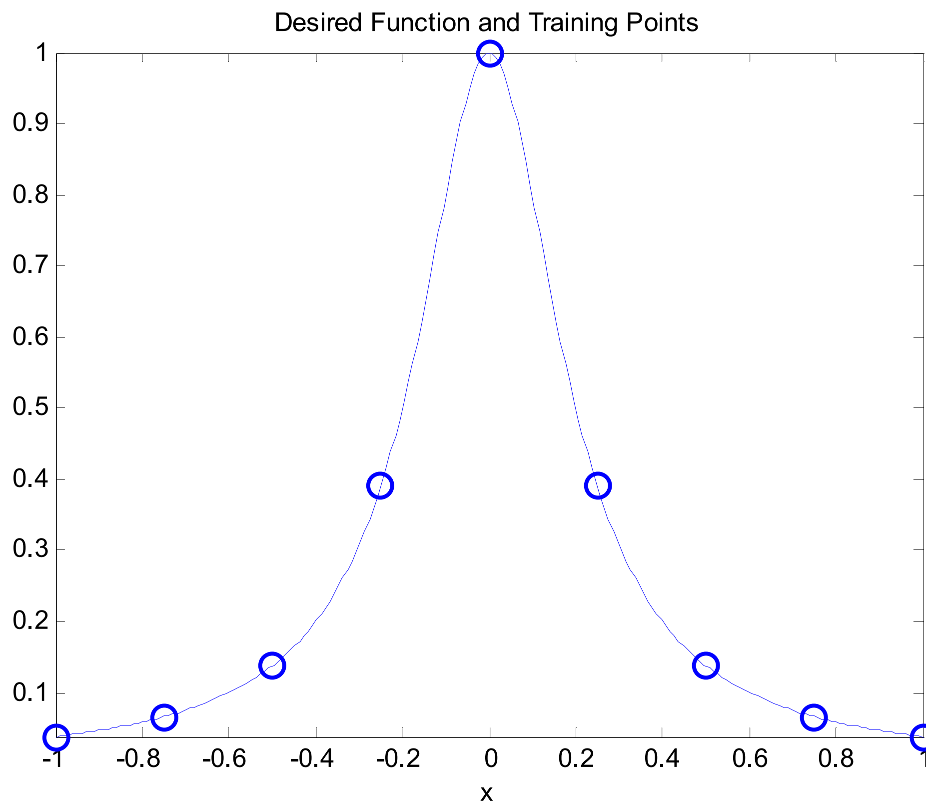



Figure 3.1.2: The differences between the desired curve and the interpolated curve at every evaluation point.

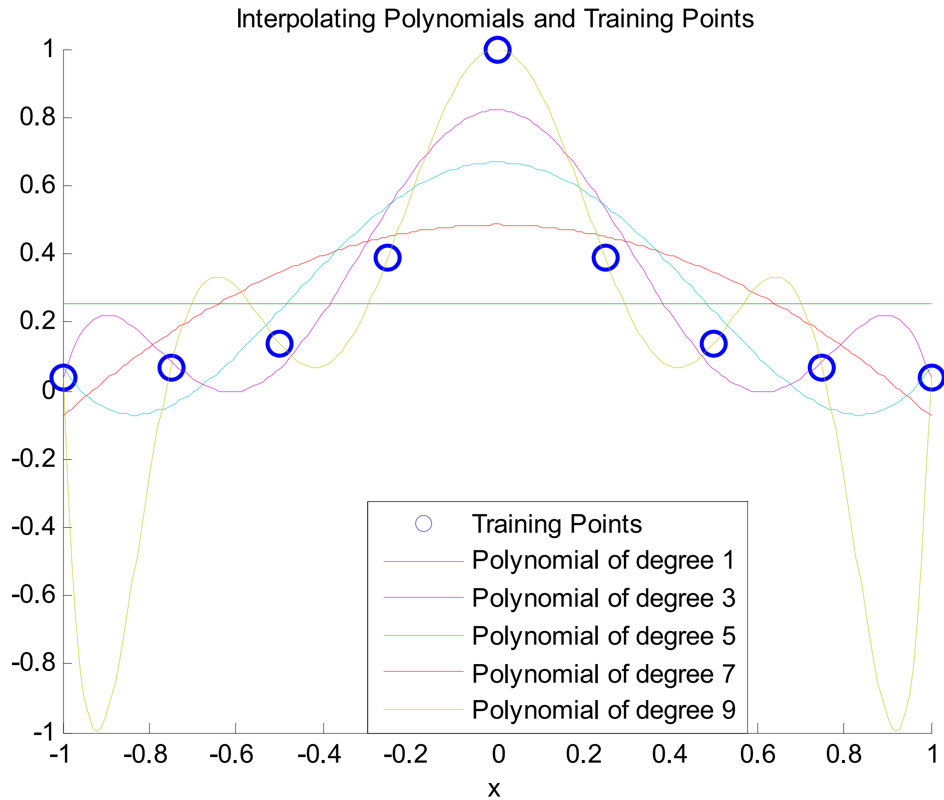
There are significant errors in between the data points. The logical solution to this problem is to use more data points. Assuming that more data points are available (which is a large assumption in any real-world application), this creates several more problems.

The first is simply one of computation time. Because the order of the interpolating polynomial is tied to the number of data points, adding more data points will result in a higher order polynomial, which will in turn require more multiplications to evaluate. Depending on the order of the polynomial and the number of points that need to be evaluated, this computation cost can become too expensive.

The second problem is more serious, and a fundamental problem rather than a practical one. This issue has to do with the behavior of high order polynomials in between data points of certain function. Specifically, increasing the order of the interpolating polynomial produces a better match at the data points, while oscillating wildly in between points. This is known as Runge's Phenomenon. It is best illustrated through an example, shown in Figure 3.1.3, which shows the results of polynomial interpolation of nine data points with polynomials of order one through nine. Note that only odd polynomial orders were used, because for this particular problem, the even power polynomial coefficients always evaluate to zero.




(a)




(b)

Figure 3.1.3: Runge's Phenomenon example. (a) Runge's Function and set of training points. (b) Interpolating polynomials of increasing order. Notice that as the order increases, the polynomials come closer to matching the training points, but have undesirable behavior in between.

As higher order polynomials are used to try and increase accuracy, the interpolating polynomials start diverging more and more from the desired function.

It is clear that simple polynomials are not suitable to use on complex problems. However, polynomials, especially of low order, have some desirable properties. For this reason, one of the most popular modifications utilizes piecewise polynomial interpolation. In other words, rather than attempt to define a single polynomial over an entire curve, a separate, low order polynomial is obtained for each segment. If certain continuity conditions are imposed when deriving the

individual segments, the result is known as a spline. If each segment is a third order polynomial, then this is known as a cubic spline.

## 3.2 Cubic Spline Review

Cubic splines have been in use for decades due to their simplicity and performance. The term spline derives from a thin, flexible strip of metal used by drafters to draw smooth curves. The spline would be bent around anchor points, forcing it into the desired shape. Similarly, mathematical splines can be thought of as having control points which constrain their behavior.

Splines are a special type of piecewise polynomial function. Piecewise polynomials avoid the issues of defining a single polynomial for an entire curve by finding a different polynomial for each segment (i.e. between each pair of data points). There are several benefits to this approach. First, because each polynomial segment only has to perform well between two points, much lower order polynomials can be used, which avoid the oscillation problem. Second, computation time is saved by not having to evaluate high order polynomial terms. However, directly using piecewise polynomials can result in undesirable behavior with rapidly changing functions, as shown in Figure 3.2.1, in which a cubic piecewise polynomial interpolates some data. The piecewise interpolant performs much better than simple polynomial interpolation, but the quality of the curve suffers at the data points, as one cubic segment transitions to another.

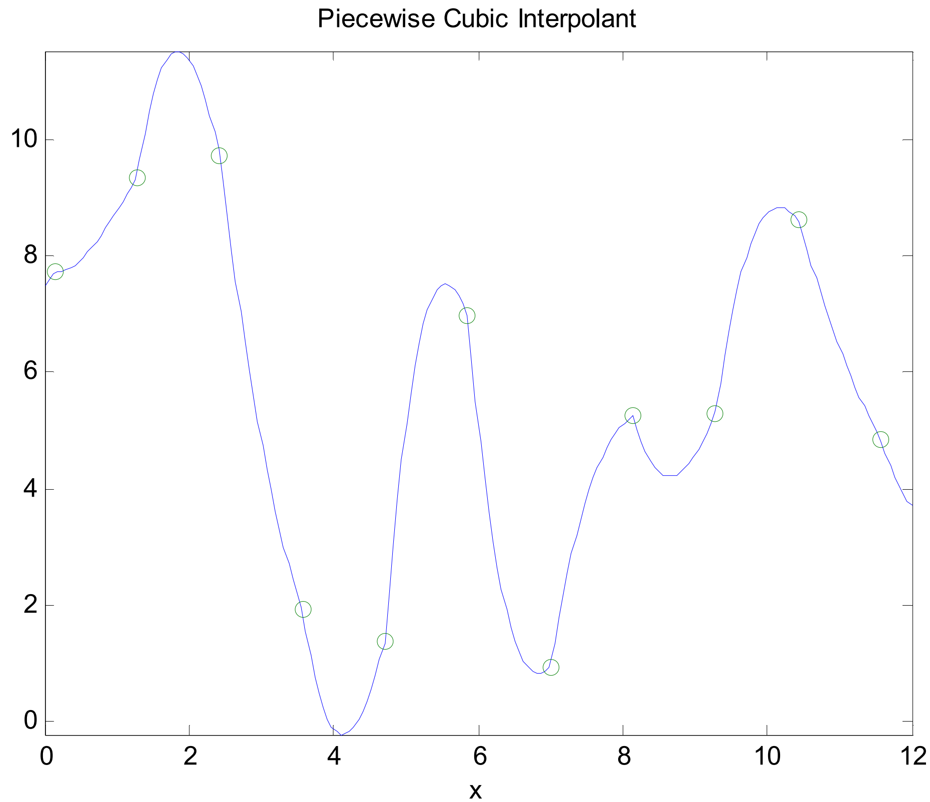



Figure 3.2.1: A piecewise cubic interpolant.

Although the interpolant is continuous, it is not differentiable at the data points because the derivative of the interpolant is not continuous. As a consequence, the curve lacks smoothness, which is displeasing visually, and cannot be numerically differentiated, which is problematic for analysis. This is what separates splines from piecewise polynomials. The difference can be observed by comparing Figure 3.2.1 with Figure 3.2.2. The spline curve is much smoother, with gradual transitions at the boundary points.

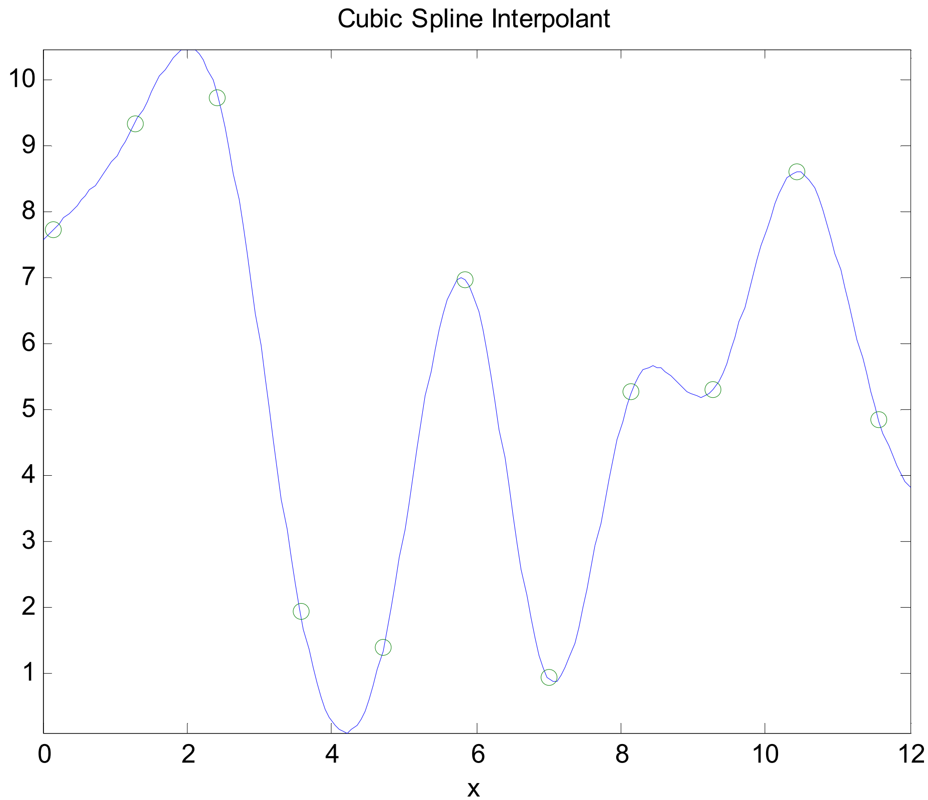



Figure 3.2.2: Cubic spline interpolant

Simply put, splines are piecewise polynomials that have constraints placed on their derivatives. Although any order of polynomial can be used, third order, or cubic, splines are by far the most common. Cubic polynomials are of a high enough order that they can match most nonlinearities over a short domain, and low order enough that they are generally well behaved. Cubic splines posses $C^2$ continuity, which means that both the first and second derivatives are continuous. Cubic spline interpolation can be extended to multiple dimensions though the use of tensor products [46], however, the cost associated with solving for the coefficients increases rapidly as the number of dimensions and data points increase.

### 3.2.1 Cubic Spline Derivation

The process for solving for the coefficients of the spline segments is a little more involved than with polynomial interpolation. If there are $N + 1$ data points, then there are $N$ segments, so the overall spline can be represented as the piecewise function (3.2-38).

$$S(x) = \begin{Bmatrix} s_1(x) & x_1 \leq x \leq x_2 \\ s_2(x) & x_3 \leq x \leq x_4 \\ \vdots & \vdots \\ s_i(x) & x_i \leq x \leq x_{i+1} \\ \vdots & \vdots \\ s_N(x) & x_N \leq x \leq x_{N+1} \end{Bmatrix} \tag{3.2-38}$$

Each segment $s_i(x)$ is a third order polynomial with the form of (3.2-39).

$$s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \tag{3.2-39}$$

For convenience, we will define $h_i = (x - x_i)$, so (3.2-39) becomes (3.2-40). Figure 3.2.3 gives a visual representation of the notation used in this derivation.

$$s_i(x) = a_i + b_i h_i + c_i h_i^2 + d_i h_i^3 \tag{3.2-40}$$

It is also useful to define here the form of the first and second derivatives of (3.2-40).

$$s_i'(x) = b_i + 2c_i h_i + 3d_i h_i^2 \tag{3.2-41}$$

$$s_i''(x) = 2c_i + 6d_i h_i \tag{3.2-42}$$

As stated earlier, $N + 1$ data points gives $N$ spline segments, each of which has four unknown coefficients $\{a_i, b_i, c_i, d_i\}$, which makes a total of $4N$ unknowns. The notation used in this section can also be seen in Figure 3.2.3.

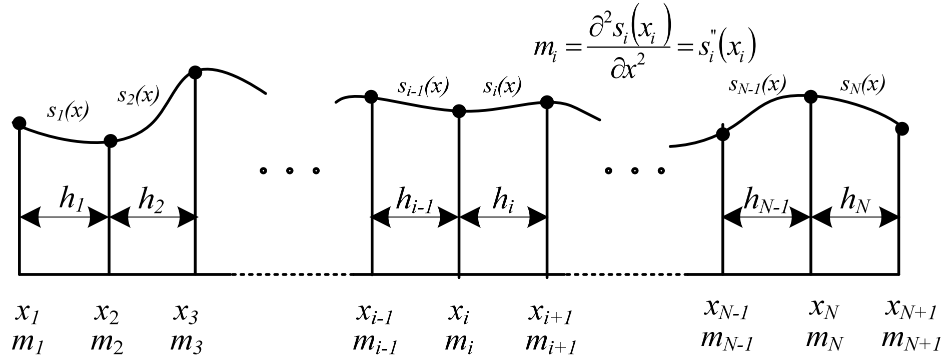



Figure 3.2.3: Illustration of notation.

In order to solve for the $4N$ unknowns, $4N$ equations are needed. To obtain the equations, we will impose the constraints of cubic splines. The first condition is that the spline must pass through every data point. This is also called the interpolation condition (3.2-43).

$$S(x_i) = \begin{Bmatrix} s_i(x_i) = y_i \ , & i \in [1, N] \\ s_N(x_{N+1}) = y_{N+1} \ , & i = N+1 \end{Bmatrix} \tag{3.2-43}$$

The second condition is the continuity condition (3.2-44), which forces the left and right hand spline segments to be equal at the data points.

$$s_i(x_{i+1}) = s_{i+1}(x_{i+1}) \quad i \in [1, N-1] \tag{3.2-44}$$

The last two conditions (3.2-45) (3.2-46) enforce the continuity of the first and second derivatives at the junctions.

$$s_i{}'(x_{i+1}) = s_{i+1}{}'(x_{i+1}) \quad i \in [1, N-1] \tag{3.2-45}$$

$$s_i{}''(x_{i+1}) = s_{i+1}{}''(x_{i+1}) \ i \in [1, N-1] \tag{3.2-46}$$

With the conditions established, it is time to derive equations for the unknowns. First, by substituting (3.2-40) into (3.2-43) and (3.2-44) we obtain (3.2-47) and (3.2-48), respectively.

$$s_i(x_i) = a_i = y_i \tag{3.2-47}$$

$$s_i(x_{i+1}) = y_{i+1} = a_i + b_i h_i + c_i h_i^2 + d_i h_i^3 \tag{3.2-48}$$

Similarly, using (3.2-41) and (3.2-42) with (3.2-45) and (3.2-46) yields (3.2-49) and (3.2-50).

$$s_i{}'(x_{i+1}) = b_i + 2c_ih_i + 3d_ih_i^2 = b_{i+1} \tag{3.2-49}$$

$$s_i{}''(x_{i+1}) = 2c_i + 6d_ih_i = 2c_{i+1} \tag{3.2-50}$$

Although (3.2-47), (3.2-48), (3.2-49), and (3.2-50) define a system of equations that can be used to solve for the spline coefficients, the computation can be greatly simplified by evaluating (3.2-42) at $x = x_i$. Recall that we defined $h_i = x - x_i$, so by setting $x = x_i$, then $h_i = 0$ and (3.2-42) simplifies to (3.2-51).

$$s_i{}''(x_i) = 2c_i \tag{3.2-51}$$

For additional convenience, we will define a new variable $m_i$ (3.2-52) to be the second derivative of the curve.

$$m_i = s_i{}''(x_i) \tag{3.2-52}$$

Now we can derive expressions for each coefficient in terms of $m_i$ and $y_i$. From (3.2-47) we already have an expression for $a_i$, rewritten here as (3.2-53)

$$a_i = y_i \tag{3.2-53}$$

We can also immediately obtain the function for $c_i$, using (3.2-51) and (3.2-52) to produce (3.2-54).

$$c_i = \frac{m_i}{2} \tag{3.2-54}$$

Next, by substituting (3.2-54) into (3.2-50), we get (3.2-55)

$$2\left(\frac{m_i}{2}\right) + 6d_ih_i = 2\left(\frac{m_{i+1}}{2}\right) \tag{3.2-55}$$

Which simplifies to (3.2-56)

$$d_i = \left(\frac{m_{i+1} - m_i}{6h_i}\right) \tag{3.2-56}$$

The last coefficient is $b_i$, whose expression can be found using the values for $a_i$, $c_i$, and $d_i$, (3.2-53), (3.2-54), and (3.2-56), respectively, in (3.2-48), giving (3.2-57), which finally simplifies down to (3.2-58).

$$y_{i+1} = y_i + b_i h_i + \left(\frac{m_i}{2}\right) h_i^2 + \left(\frac{m_{i+1} - m_i}{6h_i}\right) h_i^3 \tag{3.2-57}$$

$$b_i = \frac{y_{i+1} - y_i}{h_i} - \frac{h_i}{6}(m_{i+1} - m_i) - \frac{h_i}{2} m_i \tag{3.2-58}$$

With equations for the four coefficients in terms of $m_i$, we now just have to set up a system of equations to solve for $m$, and can easily calculate the values for the coefficients of every spline segment. Fortunately, we have one last equation to use. If we replace $b_i$, $c_i$, and $d_i$ with (3.2-54), (3.2-56), (3.2-58) in (3.2-49) (and perform a rather tedious amount of simplification which has been omitted here), we can obtain (3.2-59), which can be used to populate the system of equations in (3.2-60).

$$h_i m_i + 2(h_i + h_{i+1}) m_{i+1} + h_{i+1} m_{i+2} = 6\left[\frac{y_{i+2} - y_{i+1}}{h_{i+1}} - \frac{y_{i+1} - y_i}{h_i}\right] \tag{3.2-59}$$

$$\begin{bmatrix} h_1 & 2(h_1 + h_2) & h_2 & \cdots & & 0 \\ \vdots & h_2 & 2(h_2 + h_3) & h_3 & \cdots & \vdots \\ & & & \ddots & & \\ 0 & & \cdots & h_{N-1} & 2(h_{N-1} + h_N) & h_N \end{bmatrix} \begin{bmatrix} m_2 \\ m_3 \\ \vdots \\ m_N \end{bmatrix} = \begin{bmatrix} 6\left[\frac{y_3 - y_2}{h_2} - \frac{y_2 - y_1}{h_1}\right] \\ 6\left[\frac{y_3 - y_2}{h_2} - \frac{y_2 - y_1}{h_1}\right] \\ \vdots \\ 6\left[\frac{y_{N+1} - y_N}{h_N} - \frac{y_N - y_{N-1}}{h_{N-1}}\right] \end{bmatrix} \tag{3.2-60}$$

Sharp-eyed readers will note that this system of equations is missing values for $m_1 and\ m_{N+1}$, or in other words has two fewer rows than columns. The reason for this becomes obvious when examining (3.2-59). Solving for the value of $m_i$ (recall, this is the second derivative of a spline segment) requires the values for $m$ in the adjacent segments, which in turn rely on the second derivative values of their neighbors. In this fashion, the segments are interlocking, and finding a particular value requires solving the entire curve. At the limits, $i = 1$

and $i = N + 1$, the adjacent values of $m$ depend on data that does not exist. Therefore, it is necessary to impose end conditions.

### 3.2.2 End Conditions

The end condition choice will impact the performance of the spline curve most strongly at the boundaries. The simplest and perhaps most obvious option is called a natural or free spline, in which the first and last second derivative values are set to zero (3.2-61).

$$s_1''(x) = s_N''(x) = 0 \tag{3.2-61}$$

Obviously, this boundary condition assumes that the second derivative at node $x_1$ and $x_{N+1}$ equals zero, and will obviously not perform well when the second derivative has a large magnitude.

For periodic functions, it makes sense to match the curves on the left and right hand boundaries, giving rise to the periodic end condition (3.2-62) and (3.2-63).

$$s_1''(x) = s_N''(x) \tag{3.2-62}$$

$$s_1'(x) = s_N'(x) \tag{3.2-63}$$

The last frequently used end condition we will look at forces the third derivative to be continuous across the first and second segments (3.2-64), and across the second to last and last segments (3.2-65).

$$s_1'''(x_2) = s_2'''(x_2) \tag{3.2-64}$$

$$s_{N-1}'''(x_N) = s_N'''(x_N) \tag{3.2-65}$$

There is no end condition that will be the best choice in all cases. This is because the problem boils down to extrapolation, a fundamentally hard problem.

### 3.2.3 Cubic Spline Problems

Cubic splines work well, but have several deficiencies. The solution of a spline surface is global. That is, in order to approximate the value of a single point, the coefficients for the entire curve must be calculated. This can get computationally expensive for large numbers of data points, and in high dimensions. It is also problematic if the training points change, perhaps as new data becomes available.

## 3.3 Nearest Neighbor Spline Approximation

Nearest Neighbor Spline Approximation [47] (NNSA) is an algorithm that attempts to emulate the desirable properties of cubic splines, such as curve smoothness and interpolation ability, while only using local data. NNSA can also be thought of as a modified defuzzification algorithm of a TSK Fuzzy System. The following sections will describe a one dimensional TSK system, show how this can be enhanced with NNSA defuzzification, and finally examine how this technique can be extended to higher dimensions.

### 3.3.1 One Dimensional TSK

Let us consider a one dimensional case of a zero order TSK system with non-uniform membership functions as shown in Figure 3.3.1.

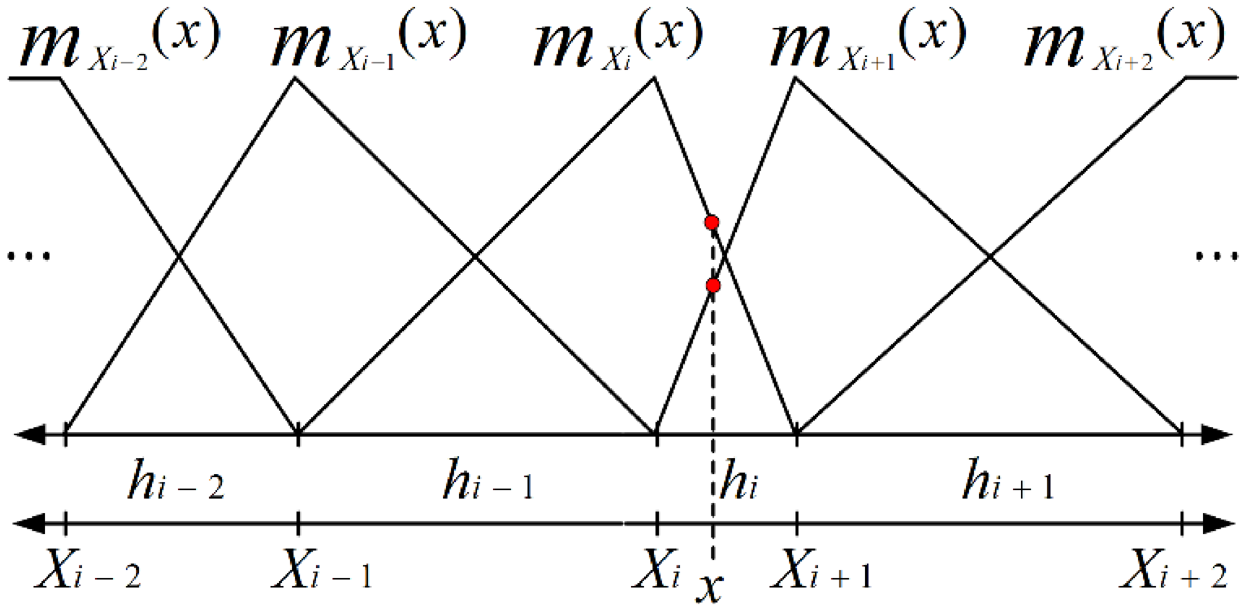



Figure 3.3.1: Showing the non-uniform triangular membership functions $m_{X_i}$ for a single input x. Note that only two membership functions $m_{X_i}$ and $m_{X_{i+1}}$are activated, and that their values sum to 1.

For an input x, only two membership functions $m_{X_i}(x)$ and $m_{X_{i+1}}(x)$ have nonzero values, given by (3.3-66) and (3.3-67).

$$m_{X_i}(x) = \frac{|x - X_i|}{h_i} \tag{3.3-66}$$

$$m_{X_{i+1}}(x) = \frac{|x - X_{i+1}|}{h_{i+1}} \tag{3.3-67}$$

Notice that since the sum of overlapping membership functions must membership functions must equal one, (3.3-66) and (3.3-67) become (3.3-68) and (3.3-69).

$$m_{X_i}(x) + m_{X_{i+1}}(x) = 1 \tag{3.3-68}$$

$$m_{X_{i+1}}(x) = 1 - m_{X_i}(x) \tag{3.3-69}$$

If membership functions fulfill equations (3.3-66) to (3.3-69), then values of $m_{X_i}$ can be also used as “normalized” distances. This way, for approximation purposes, the non-uniform case is simplified to a uniform one.

The commonly used defuzzification process for a zero-order TSK FS leads to the system output of (3.3-70).

$$g(x) = \frac{f(X_i)m_{X_i}(x) + f(X_{i+1})m_{X_{i+1}}(x)}{m_{X_i}(x) + m_{X_{i+1}}(x)} \tag{3.3-70}$$

Where $f(X_i)$ and $f(X_{i+1})$ are values associated with each membership function and corresponding grid nodes. Because of (3.3-68) and (3.3-69), equation (3.3-70) can be simplified to remove the division entirely, and a use a single multiplication in (3.3-72).

$$g(x) = f(X_i)m_{X_i}(x) + f(X_{i+1})m_{X_{i+1}}(x) \tag{3.3-71}$$

$$g(x) = m_{X_i}(x)[f(X_i) - f(X_{i+1})] + f(X_{i+1}) \tag{3.3-72}$$

Notice that with this approach, the output value may be obtained using a defuzzification process that, for a one dimensional case, uses one membership function $m_{X_i}(x)$, and two neighboring values $f(X_i)$ and $f(X_{i+1})$.

### 3.3.2 One Dimensional NNSA

We will now describe the NNSA defuzzification process, and highlighting the similarities and differences compared to the previous section.

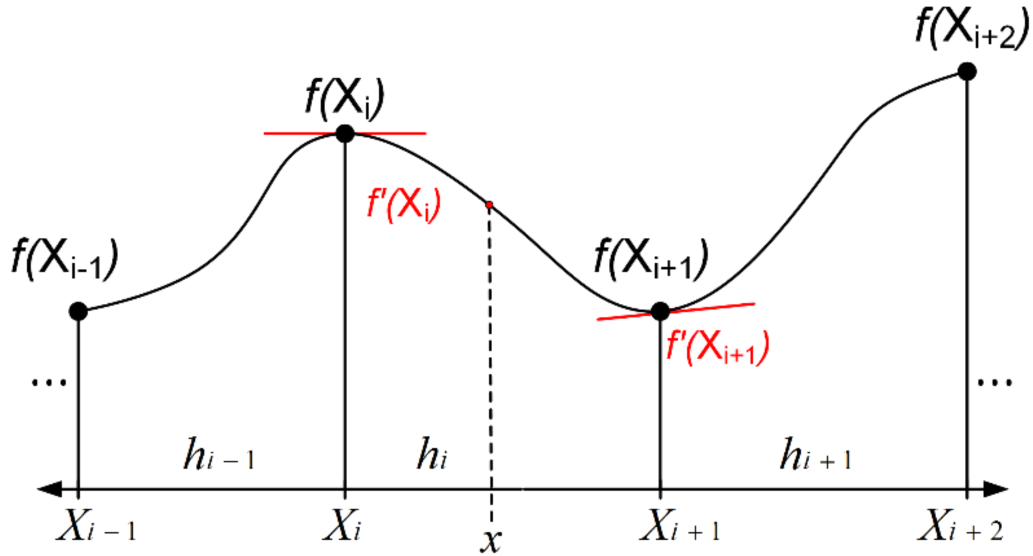



Figure 3.3.2: The NNSA algorithm constructs a third-order polynomial using the four nearest node value and two slope values.

In Figure 3.3.2, the range between $X_i$ and $X_{i+1}$ can be approximated by the third order polynomial $g(x)$:

$$g(x) = a + b\left(\frac{x - X_i}{h_i}\right) + c_i\left(\frac{x - X_i}{h_i}\right)^2 + d_i\left(\frac{x - X_i}{h_i}\right)^3 \tag{3.3-73}$$

Or, by using (3.3-66),

$$g(x) = a + b\, m_{X_i}(x) + c\, m_{X_i}(x)^2 + d\, m_{X_i}(x)^3 \tag{3.3-74}$$

The first derivative will also be needed:

$$g'(x) = b\ + 2c\, m_{X_i}(x)^2 + 3d\, m_{X_i}(x)^2 \tag{3.3-75}$$

In order to solve for the four unknown coefficients, four constraints are necessary. The first two constraints will be that the interpolating function $g(x)$ should pass through the values $f(X_i)$ and $f(X_{i+1})$. For the second two constraints, we will enforce that the derivative of the interpolating function $g'(x)$ should match the derivative of the underlying function $f'(x)$ at $X_i$

and $X_{i+1}$. Finding values for the derivatives of the underlying function is addressed in Section 3.3.2.1.

By evaluating (3.3-74) and (3.3-75) at $X_i$ and $X_{i+1}$, we obtain (3.3-76), (3.3-77), (3.3-78), and (3.3-79), which define a system of equations that can be solved for the coefficients.

$$\mathrm{g}(\mathrm{X_i}) = \mathrm{f}(\mathrm{X_i}) = \mathrm{a} \tag{3.3-76}$$

$$\mathrm{g}'(\mathrm{X_i}) = \mathrm{f}'(\mathrm{X_i}) = \mathrm{b} \tag{3.3-77}$$

$$\mathrm{g}(\mathrm{X_{i+1}}) = \mathrm{f}(\mathrm{X_{i+1}}) = \mathrm{a} + \mathrm{b}\ \mathrm{m_{X_i}}(\mathrm{X_{i+1}}) + \mathrm{c}\ \mathrm{m_{X_i}}(\mathrm{X_{i+1}})^2 + \mathrm{d}\ \mathrm{m_{X_i}}(\mathrm{X_{i+1}})^3 \tag{3.3-78}$$

$$\mathrm{g}'(\mathrm{x}) = \mathrm{f}'(\mathrm{X_{i+1}}) = \mathrm{b}\ + 2\mathrm{c}\ \mathrm{m_{X_i}}(\mathrm{X_{i+1}}) + 3\mathrm{d}\ \mathrm{m_{X_i}}(\mathrm{X_{i+1}})^2 \tag{3.3-79}$$

The system can be solved analytically. Clearly, (3.3-76) and (3.3-77) give the first two unknowns directly. Rearranging (3.3-79) to solve for *c* gives (3.3-80).

$$c = \frac{\mathrm{f}'(\mathrm{X_{i+1}}) - \mathrm{b} - 3\mathrm{d}\ \mathrm{m_{X_i}}(\mathrm{X_{i+1}})^2}{2\ \mathrm{m_{X_i}}(\mathrm{X_{i+1}})} \tag{3.3-80}$$

Replacing *c* in (3.3-78) and solving for *d* gives (3.3-81).

$$\mathrm{d} = \frac{\mathrm{f}(\mathrm{X_{i+1}}) - \mathrm{a} - \mathrm{b}\ \mathrm{m_{X_i}}(\mathrm{X_{i+1}}) - c\ \mathrm{m_{X_i}}(\mathrm{X_{i+1}})^2}{\mathrm{m_{X_i}}(\mathrm{X_{i+1}})^3} \tag{3.3-81}$$

Substituting (3.3-66), (3.3-76) and (3.3-77) into (3.3-80) and (3.3-81) and simplifying yields the final form of the expressions for the coefficients (3.3-82)-(3.3-85).

$$a = f(X_i) \tag{3.3-82}$$

$$b = f'(X_i) \tag{3.3-83}$$

$$c = \frac{3\big(f(X_{i+1}) - f(X_i)\big)}{{h_i}^2} - \frac{\big(2f'(X_{i+1}) + f'(X_i)\big)}{h_i} \tag{3.3-84}$$

$$d = \frac{\big(f'(X_{i+1}) + f'(X_i)\big)}{{h_i}^2} - \frac{2\big(f(X_{i+1}) - f(X_i)\big)}{h_i^3} \tag{3.3-85}$$

With expressions for all four coefficients, values for the derivatives are needed before the algorithm can be used.

#### *3.3.2.1 Derivative Approximation*

The above formulation requires values for the derivatives of the underlying function at the points $X_i$ and $X_{i+1}$. In almost any real world example, the underlying function is either unknown, or of sufficient complexity that computing the derivative directly is unfeasible. As a result, the best solution is to approximate the derivative from the available data. The easiest and most obvious solution is to use simple divided differences (3.3-86) and (3.3-87).

$$f'(X_i) = \frac{f(X_{i+1}) - f(X_i)}{h_i} \tag{3.3-86}$$

$$f'(x_{i+1}) = \frac{f(X_{i+2}) - f(X_{i+1})}{h_{i+1}} \tag{3.3-87}$$

The quality of the curve will be dependent on the accuracy of the derivative approximations. Inaccurate values will result in a curve with a discontinuous second derivative. This means that single sided divided difference formulas will not suffice. Obtaining a more accurate derivative requires using the Taylor Expansion (3.3-88) and some algebraic manipulation.

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} * (x-a)^n = \cdots \tag{3.3-88}$$

$$= f(a) + \frac{f^{(1)}(a)}{1!} * (x-a)^1 + \frac{f^{(2)}(a)}{2!} * (x-a)^2 + \cdots$$

Setting $a = x_0$ yields (3.3-89).

$$f(x) = f(x_0) + f'(x_0) * (x - x_0) + \frac{f^{(2)}(x_0)}{2} * (x - x_0)^2 + \cdots \tag{3.3-89}$$

Expressions for forward (3.3-90) and backward (3.3-91) differences can be obtained by setting $x = x_0 + h$ and $x = x_0 - h$, respectively, and solving for $f'(x_0)$

$$f'(x_0) = \frac{f(x_0+h) - f(x_0)}{h} - \frac{f''(x_0)h}{2} - \frac{f'''(x_0)h^2}{6} + \cdots \tag{3.3-90}$$

$$f'(x_0) = \frac{f(x_0) - f(x_0-h)}{h} + \frac{f''(x_0)h}{2} - \frac{f'''(x_0)h^2}{6} + \cdots \tag{3.3-91}$$

The upper bound of the error of the single sided difference formulas is $O(h)$, caused by the truncation of all the higher order terms. Adding (3.3-90) to (3.3-91) produces the centered difference formula (3.3-92).

$$f'(x_0) = \frac{f(x_0+h) - f(x_0-h)}{2h} - \frac{f'''(x_0)h^2}{6} - \frac{f^4(x_0)h^4}{120} + \tag{3.3-92}$$

Which reduces the error from $O(h)$ to $O(h^2)$ by cancelling out the errors from the terms with odd powers of $h$. Truncating the higher order terms and substituting $X_i$ and $X_{i+1}$for $x_o$ (as well as replacing $h$ with $h_i$ and $h_{i+1}$) yields the needed derivative approximations (3.3-93) and (3.3-94).

$$f'(X_i) = \frac{f(X_{i+1}) - f(X_{i-1})}{2h_i} \tag{3.3-93}$$

$$f'(X_{i+1}) = \frac{f(X_{i+2}) - f(X_i)}{2h_{i+1}} \tag{3.3-94}$$

The above formulas assume that the data points are evenly spaced. Unevenly spaced points complicate the derivation a bit, but (3.3-95) and (3.3-96) are the result.

$$f(X_i) = \frac{\frac{h_i}{h_{i-1}}(f(X_i) - f(X_{i-1}) - \frac{h_{i-1}}{h_i}\left(f(X_{i+1}) - f(X_i)\right)}{h_i + h_{i-1}} \tag{3.3-95}$$

$$f(X_{i+1}) = \frac{\frac{h_{i+1}}{h_i}\left(f(X_{i+1}) - f(X_i)\right) - \frac{h_i}{h_{i+1}}(f(X_{i+2}) - f(X_{i+1}))}{h_{i+1} + h_i} \tag{3.3-96}$$

These formulas provide an approximation of the actual derivative of the underlying function at the specified points. The errors in the approximation can cause the resulting curve to have small discontinuities in the second derivative, as shown in Figure 3.3.3.

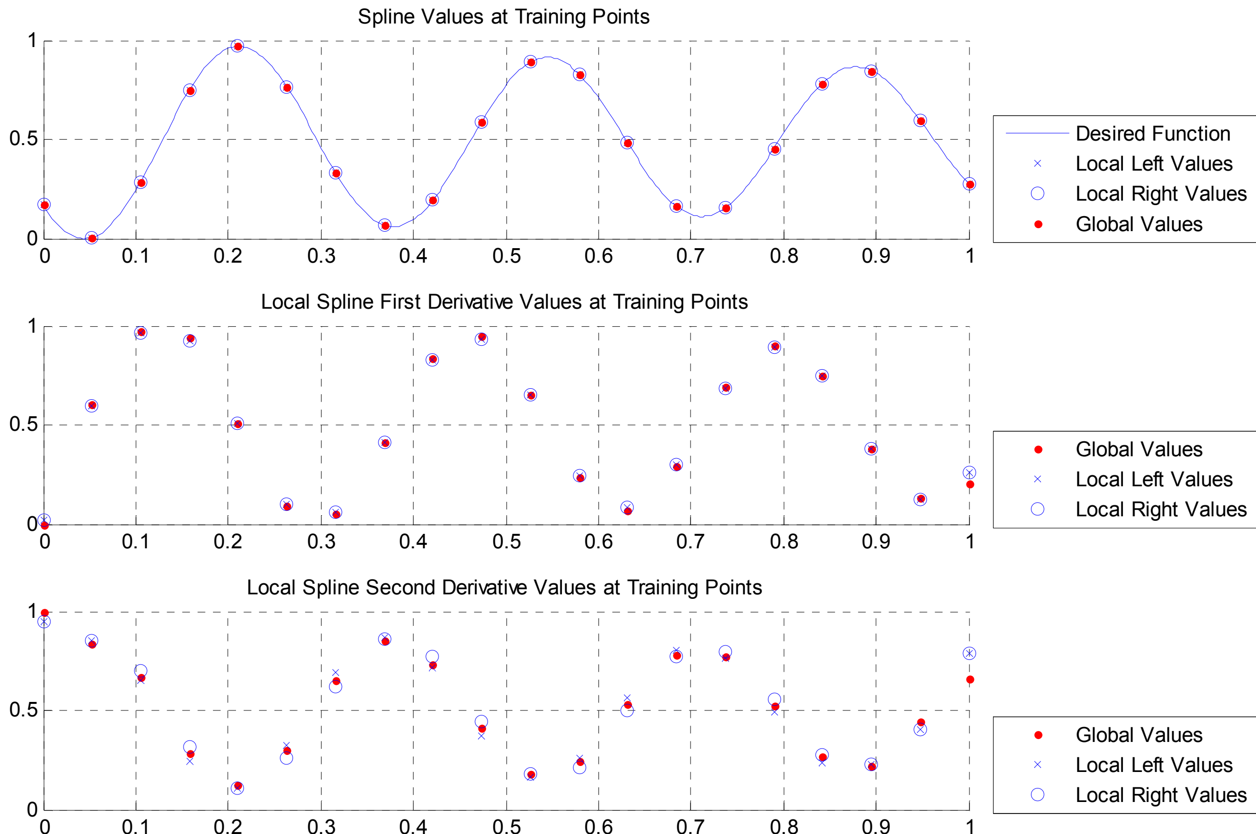



Figure 3.3.3: Comparing the continuity of various derivatives between global spline and NNSA. The global values are from a global spline routine. The local left and right values are from the NNSA algorithm on the left and right hand sides of each data point.

The issue boils down to the fact that when using centered difference derivative approximations, moving from one NNSA segment to the next brings in one new data point and removes another. That means fully half of the data points are different, and the result is a sometimes discontinuous second derivative. Higher order divided difference approximations such as (3.3-97) that use more data points can help reduce the error, with the extreme case of using all of the available data points.

$$f'(x_i) = \frac{-f(x_{i+2}) + 8f(x_{i+1}) - 8f(x_{i-1}) + f(x_{i-2})}{12h_i} - O(h^4) \tag{3.3-97}$$

However, increasing the number of points required for the algorithm increases the computational overhead, and gets away from the localized solution that was the original goal. In addition, using more points exacerbates the problems that must be dealt with at the edge of data sets. For those reasons, centered divided difference derivative approximation will be sufficient.

Provided the four surrounding points are available, the equations for the coefficients in the previous section and the formula for the derivative approximations above can be used to approximate any arbitrary function. However, it should be apparent that problems appear at the edges for the data set, and so special care must be taken.

#### *3.3.2.2 Handling Edges*

Evaluating (3.3-95) at $i = 1$ and (3.3-96) for $i = N$ reveals the problem with edge values, as shown in (3.3-98) and (3.3-99).

$$f'(X_1) = \frac{f(X_2) - f(X_{-1})}{2h_1} \tag{3.3-98}$$

$$f'(X_{N+1}) = \frac{f(X_{N+2}) - f(X_N)}{2h_{N+1}} \tag{3.3-99}$$

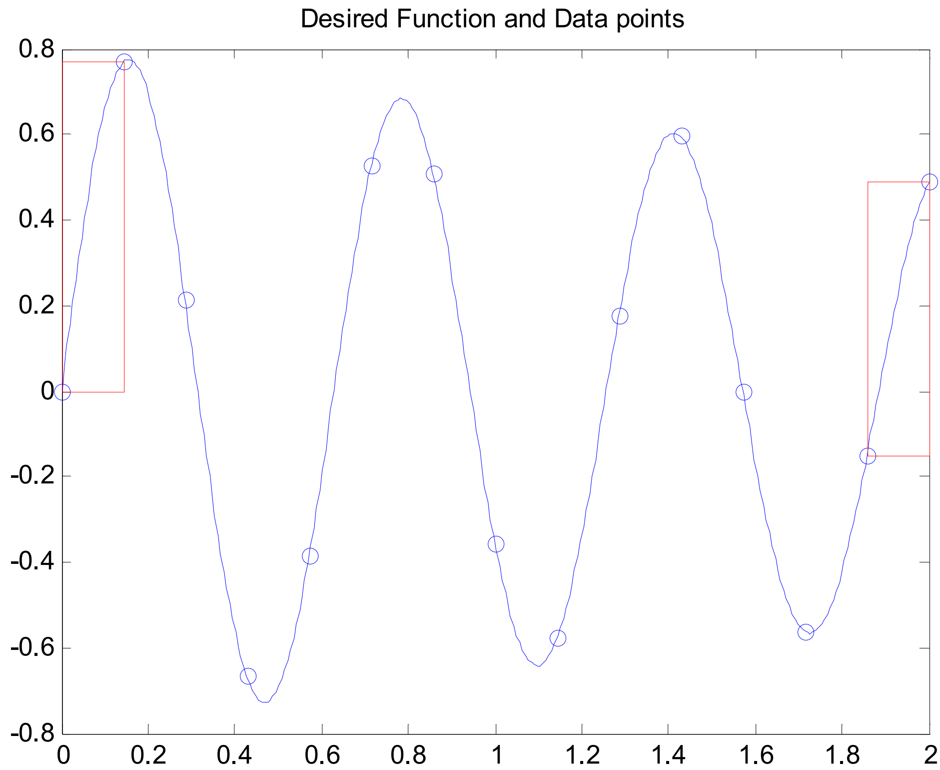
Since the values at $f(X_{-1})$ and $f(X_{N+2})$ are unavailable, some method of obtaining approximations of the derivatives is needed at the boundaries.

There are two obvious strategies for dealing with the edges. The first is to simply use first order forward/backward approximations of the derivative at the boundaries, such as (3.3-86) and (3.3-87). The benefit of this approach is simplicity, but forward and backward difference approximations generate rather large errors, and must be generated on the fly. The second solution is to use the available data points to extrapolate an additional “virtual” point on each end. While this could also be done on the fly, it has the benefit of being easily accomplished as a preprocessing step. The next issue is what extrapolation method to use.

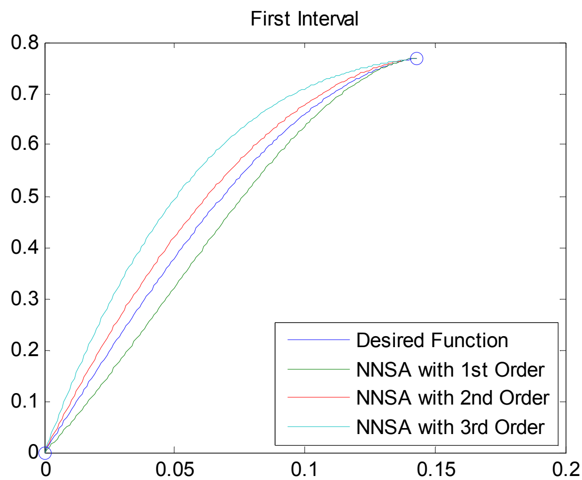
Given the limited number of data points available, and the nature of the NNSA algorithm, some form of polynomial extrapolation seems appropriate. The NNSA algorithm only uses at most four data points, so it makes sense to limit the highest order polynomial to a third order polynomial. Furthermore, as discussed in section 3.1.2, using high degree polynomials can cause wild behavior. A zero order polynomial is a constant, and would simply duplicate the points on the boundaries, which will rarely give good results, so we will limit the discussion to polynomial extrapolation with first, second, and third order polynomials. The benchmark function was an exponentially decaying sinusoid described by (3.3-100) and shown in Figure 3.3.4(a).

$$f(x) = 0.8\exp(-0.2x)\sin(10x) \tag{3.3-100}$$

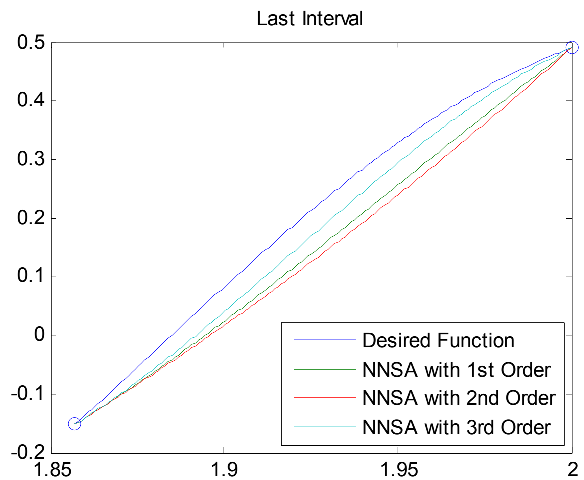
No matter what extrapolation method is used, the NNSA algorithm will produce identical results for all but the first and last segments, indicated by the red boxes. For that reason, Figure 3.3.4(b) and Figure 3.3.4(c) show the results of applying NNSA for the first and last segments, respectively.




(a)




(b)




(c)

Figure 3.3.4: Showing the benchmark function and results. (a) The function and data points, with the first and last segments highlighted. (b) Resulting NNSA surface with different extrapolation methods for the first interval. (c) Resulting NNSA surface with different extrapolation methods for the last interval.

In the first segment, second order extrapolation appears to be closest to the desired curve, while in the last segment the third order extrapolation performs the best. It is clear that the behavior of the curve at the edge will determine which method gives the best results. By shifting the start and stop values through one full period of the sinusoid and averaging the results for each method, it was found that a third order extrapolation produced the lowest errors, and is therefore the default extrapolation technique in the NNSA algorithm. As a final note, extrapolation is a fundamentally hard problem. No single method will be optimal in every case, and the goal here was to identify a method that would perform reasonably well.

### 3.3.3 Extension to Multiple Dimensions

The defuzzification described in section 3.3.1 can be extended for multiple dimensions. The multidimensional case can be handled as multiple steps of the one dimensional case described by the formulas (3.3-82)-(3.3-85), (3.3-93), and (3.3-94). In the two-dimensional example shown in Figure 3.3.5, the point $(x, y)$ is associated with two membership functions $m_{X_i}, m_{X_{i+1}}$ in the x direction, and $m_{Y_j}, m_{Y_{j+1}}$ in the y direction.

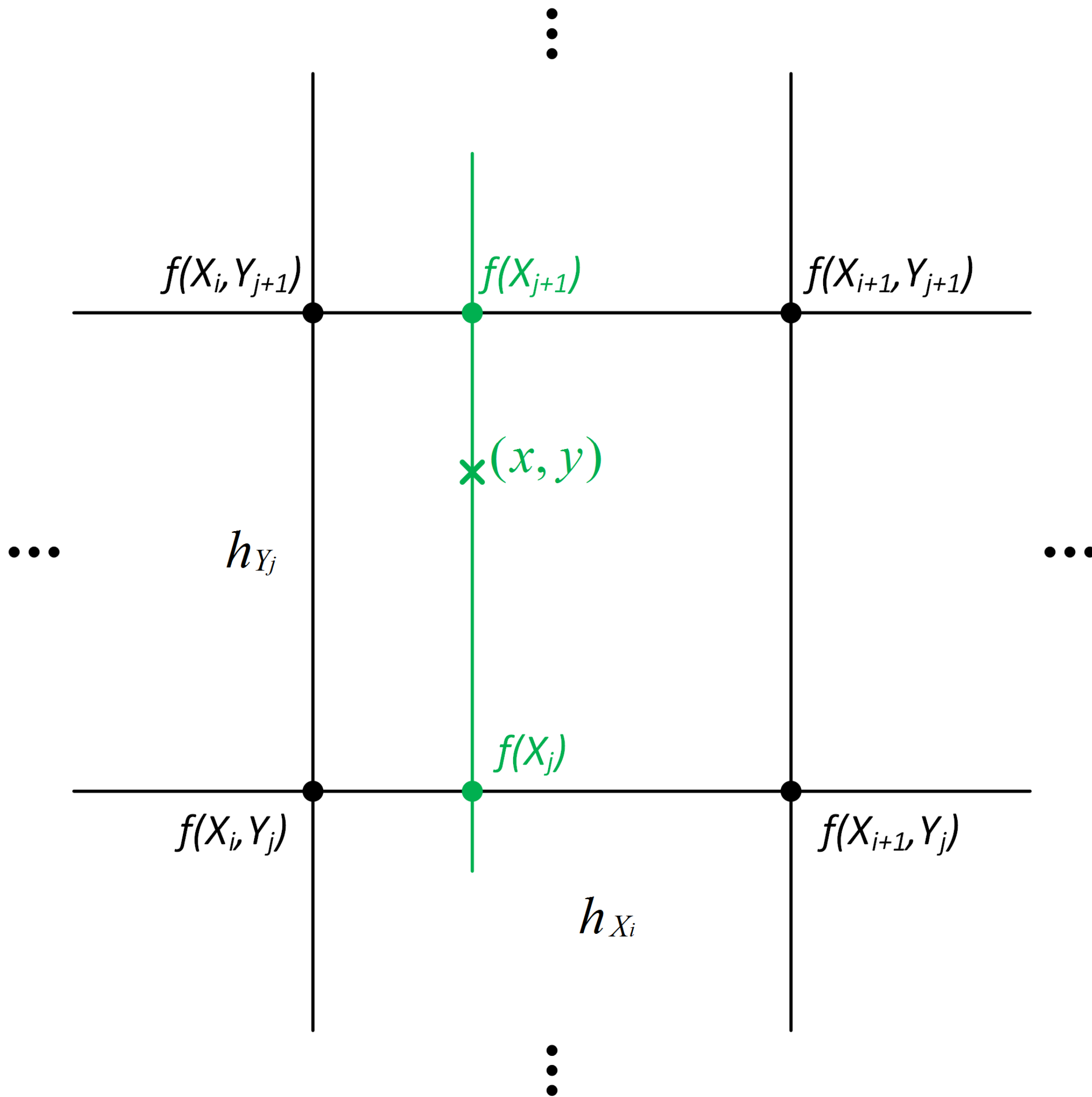



Figure 3.3.5: Demonstrating the defuzzification process in two dimensions. In this case, one-dimensional defuzzification is performed in the $x$ direction in order to obtain the intermediate points $f(X_j)$ and $f(X_{j+1})$. Defuzzification in the $y$ direction is then performed on the intermediate points to obtain the final value.

With x and y as inputs, the output value is calculated as

$$g(x,y) = m_{X_i}(x)m_{Y_j}(y)f(X_i, Y_j) + m_{X_{i+1}}(x)m_{Y_j}(y)f(X_{i+1}, Y_j) \\ + m_{X_i}(x)m_{Y_{j+1}}(y)f(X_i, Y_{j+1}) + m_{X_{i+1}}(x)m_{Y_{j+1}}(y)f(X_{i+1}, Y_{j+1}) \tag{3.3-101}$$

Because of (3.3-68), equation (3.3-101) can be rewritten as

$$g(x,y) = f(X_i,Y_j)\left[m_{X_i}(x)m_{Y_j}(y)\right] \tag{3.3-102}$$

$$+ f(X_{i+1},Y_j)\left[\left(1-m_{X_i}(x)\right)m_{Y_j}(y)\right]$$

$$+ f(X_i,Y_{j+1})\left[m_{X_i}(x)\left(1-m_{Y_j}(y)\right)\right]$$

$$+ f(X_{i+1},Y_{j+1})\left[m_{X_{i-1}}(x)\left(1-m_{Y_j}(y)\right)\right]$$

Notice again that in the two dimensional case, the output value can be calculated as a function of the four nearest grid point values and two membership functions $m_{X_i}(x)$ and $m_{Y_j}(y)$.

Let us first calculate the values of intermediate points $f(X_j)$ and $f(X_{j+1})$:

$$f(X_j) = f(X_i,Y_j)m_{X_i}(x) + f(X_{i+1},Y_j)m_{X_{i+1}}(x) \tag{3.3-103}$$

$$f(X_{j+1}) = f(X_i,Y_{j+1})m_{X_i}(x) + f(X_{i+1},Y_{j+1})m_{X_{i+1}}(x)$$

$$f(X_j) = f(X_i,Y_j)m_{X_i}(x) + f(X_{i+1},Y_j)\left(1-m_{X_i}(x)\right) \tag{3.3-104}$$

$$f(X_{j+1}) = f(X_i,Y_{j+1})m_{X_i}(x) + f(X_{i+1},Y_{j+1})\left(1-m_{X_i}(x)\right)$$

Then the output value at (x,y) can be calculated as:

$$g(x,y) = f(X_j)m_{Y_j}(y) + f(X_{j+1})\left(1-m_{Y_j}(y)\right) \tag{3.3-105}$$

Inserting (3.3-104) into (3.3-105) leads to the same formula (3.3-102). It is not important if the first calculation is done in the x or in y direction. The formula will be the same and leads to the well-known fuzzy product encoding formula (3.3-102).

The two-dimensional case described by (3.3-102), (3.3-103), (3.3-104), and (3.3-105) can be extended to multidimensional cases, which can be simply computed as a series of one dimensional calculations.

In a very similar fashion, the NNSA algorithm can also be applied in higher dimensions. The approach is illustrated in Figure 3.3.6 for the 2-dimensional case. Given function values at the grid points, the following steps must be taken in order to calculate a single function value located outside of the grid.

1. Find the intermediate y values along the all grid lines (red dash lines) in in the x direction using values of function at a grid points and (3.3-74).

2. Once intermediate y values at the points where the dotted blue line crosses the grids are known, the value at the desired point D can be approximated.

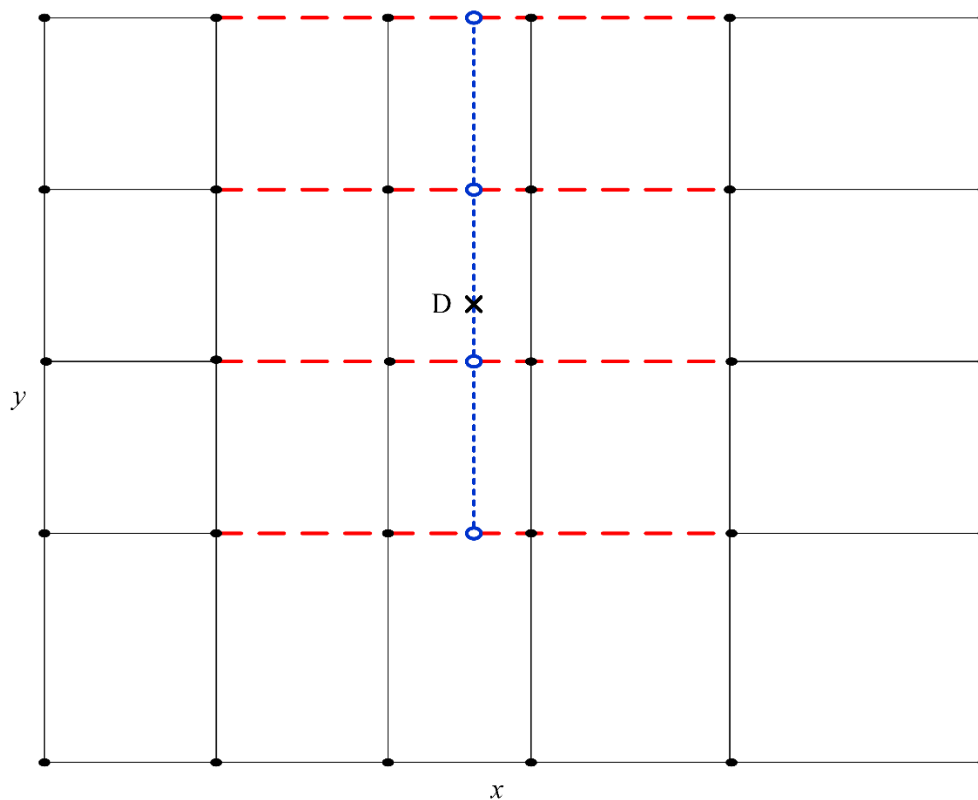



Figure 3.3.6: Illustration of the values that must be computed for the two dimensional NNSA1 algorithm.

In the above example, computation was done first in the $x$ direction and then in the $y$ direction. Of course, the process can be reversed—the $y$ direction is solved first, followed by the $x$ direction—with similar results.

This concept can be extended for multiple dimensions such that the dimensionality is reduced one by one in subsequent steps. The algorithm for arbitrary dimension d is as follows:

1. Inputs
    a. x: an array of length d which contains the location of the desired point
    b. Xt: a matrix with $n^d$ rows – where n is the number of training points per dimension - and d +1 columns. The first d columns contain the input points, while the last column holds the function value. Note that the points arranged in a grid pattern.
    c. npoints: the number of points required per dimension. For NNSA this is 4.
2. For i = d down to 1
    a. b=$npoints^{i-1}$
    b. Out=array of length b
    c. For j = 1 to b
        i. For k=1 to npoints
            1. Xa(k)=Xt(j+(k-1)*b,i)
            2. Ya(k)=Xt(j+(k-1)*b,i+1)
        ii. end
    d. Calculate Out(j) using (3.3-74).
    e. end
3. Xt=[Xt(1 to b,1 to i-1), out]

4. end
5. After the loop exits, Xt will contain a single value that is the approximated value at location x.

### 3.3.4 Comments

The proposed algorithm requires exactly the same parameters as the zero-order TSK with triangular membership functions and node values. Only the defuzzification process is different, as given by equations (3.3-82)–(3.3-85).

The name Nearest Neighbor Spline Approximation is adopted because only the nearest four neighbors are needed in the calculation process. The only difference between zero-order TSK and NNSA is that in TSK, only the two nearest node values are required, while NNSA uses the nearest four. As a consequence, any algorithm which has already been developed to tune or train TSK system can also be applied to train or tune the NNSA system, because exactly the same parameters are adjusted in both systems.

# Chapter 4 Experimental Results

For comparison, the proposed algorithm was tested on a variety of problems against popular machine learning and approximation techniques ANN (MLP trained with EBP), ANN (FCC trained with NBN), ELM, SVM, ANFIS, TSK FS, and Global Spline. For the algorithms that required user parameters, a search was performed to find the parameters that minimized the testing error. For example: size and starting weights for the ANNs, the input weight and bias range for ELM; the cost constant, C, size of epsilon insensitive tube, $\varepsilon$, and radius of the RBF kernel, $\gamma$ for SVR, and the number of input membership functions for ANFIS. All experiments were performed using MATLAB, running on a Windows machine with an Intel i5-2300 operating at 2.80 GHz and 8GB of RAM.

### 4.1 Peaks Problem

Various learning algorithms were tested on a highly nonlinear peaks benchmark problem described by (4.1-106) and shown in Figure 4.1.1.

$$z(x,y) = -\frac{1}{30}\exp(-1-6x-9x^2-9y^2) - (0.6x-27x^3-243y^5)\exp(-9x^2-9y^2) + (0.3-1.8x+2.7x^2)\exp(-1-6y-9x^2-9y^2) \tag{4.1-106}$$

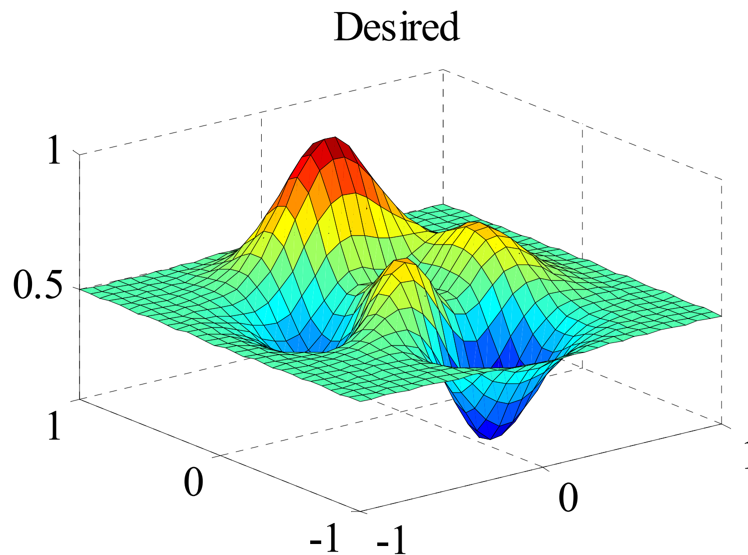



Figure 4.1.1: Peaks benchmark surface plot.

For the benchmark trials, 2000 random patterns were generated for training, and a separate set of 1000 random patterns was used for testing. For the algorithms that require node values on a regular grid, an 8x8 grid was approximated from the random training data. The time required to create the grid was counted as the "training" time for the TSKFS, SPLINE, and NNSA algorithms. For the algorithms that have randomized starting conditions, ten trials were run, and

the best results were used. For the algorithms that require user set parameters, the following settings were used:

ANN-MLP: 40 hidden neurons.

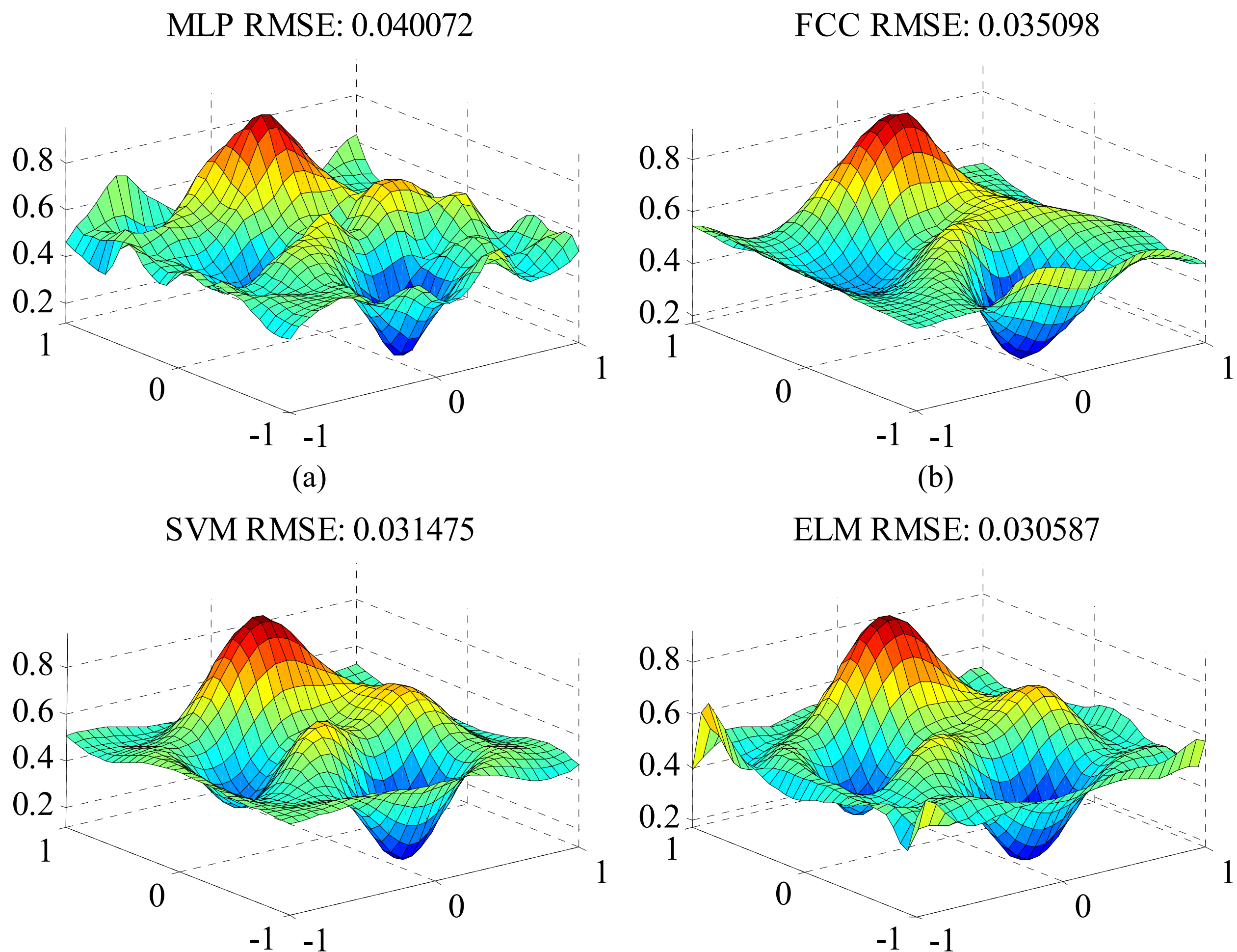
ANN-FCC: 10 hidden neurons.

ELM: 60 RBF units.

SVR: $\gamma = 3$, $C = 10$

ANFIS: 3 generalized bell membership functions per input dimension.

The training set was supplied to each of the described algorithms, and the test set applied after training. Figure 4.1.2 shows the surface produced by each algorithm, and the results are summarized in **Table IV** in terms of error and execution time for both training and testing.



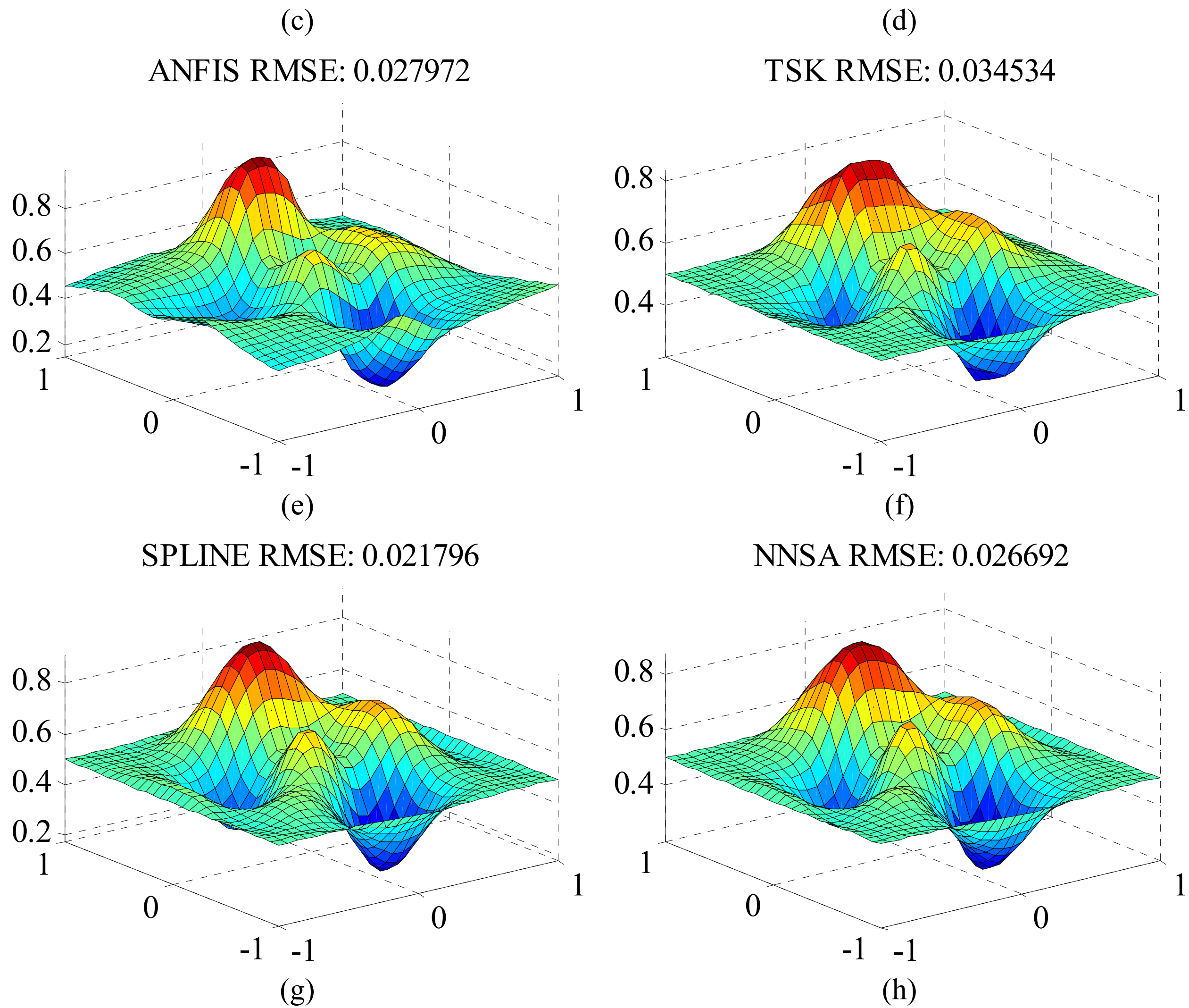



Figure 4.1.2: Surface plots for peaks benchmark for (a) ANN-MLP (b) ANN- FCC (c) SVM (d) ELM (e) ANFIS (f) TSK FS (g) Global Spline (h) NNSA.

**Table IV**: Results for peaks benchmark in terms of Training and Testing errors (RMSE), Training and Testing Times (s), and the number nodes/neurons.

| Algorithm | Training RMSE | Testing RMSE | Training Time (s) | Testing Time(s) | *nn* |
|---|---|---|---|---|---|
| ANN-MLP | 0.0381 | 0.0401 | 7.0832 | 0.0238 | 40 |
| ANN-FCC | 0.0329 | 0.0351 | 13.2805 | 0.0569 | 10 |
| SVM | 0.0311 | 0.0315 | 39.6671 | 0.0973 | 79 |
| ELM | 0.0301 | 0.0306 | 7.0258 | 0.0907 | 60 |
| ANFIS | 0.0238 | 0.0280 | 0.4186 | 0.0811 | 6 |

| | | | | | |
|---|---|---|---|---|---|
| TSK | 0.0369 | 0.0345 | 0.0105 | 0.0881 | 16 |
| SPLINE | 0.0369 | 0.0218 | 0.0105 | 0.4909 | 16 |
| NNSA | 0.0286 | 0.0267 | 0.0105 | 0.4531 | 16 |

As one can see from **Table IV**, traditional MLP neural networks trained with EBP require a very large number of nodes to train, and the errors obtained are still larger than in other methods. Better results can be obtained with SVM, but the training is very time consuming—especially if optimal training parameters must be found using the grid search method. On the other hand, ELM needs a large number of RBF units, but the number of trainable parameters is still smaller than in the case of ANN. The ANFIS algorithm produces very good results with a very short training time. The TSK fuzzy system is very competitive with the methods that required training. The only disadvantage of the TSK system is that the output surface is rawer than that obtained with the learning methods. Global Spine and NNSA produced the lowest and second lowest testing errors, respectively.

### 4.2 Forward Kinematics

Control of robotics and motors is a commonly-used benchmark for approximation methods. The forward kinematics problem is useful in robotics and computer animation. Presented here is forward 2D kinematics problem in which the position of a manipulator, such as a robot arm or animated wire frame, must be found given the lengths and angles between the links. Shown in Figure 4.2.1, the desired X and Y position of the arm is given as an input, and the angles alpha and beta must be calculated to move the manipulator to the specified XY coordinate.

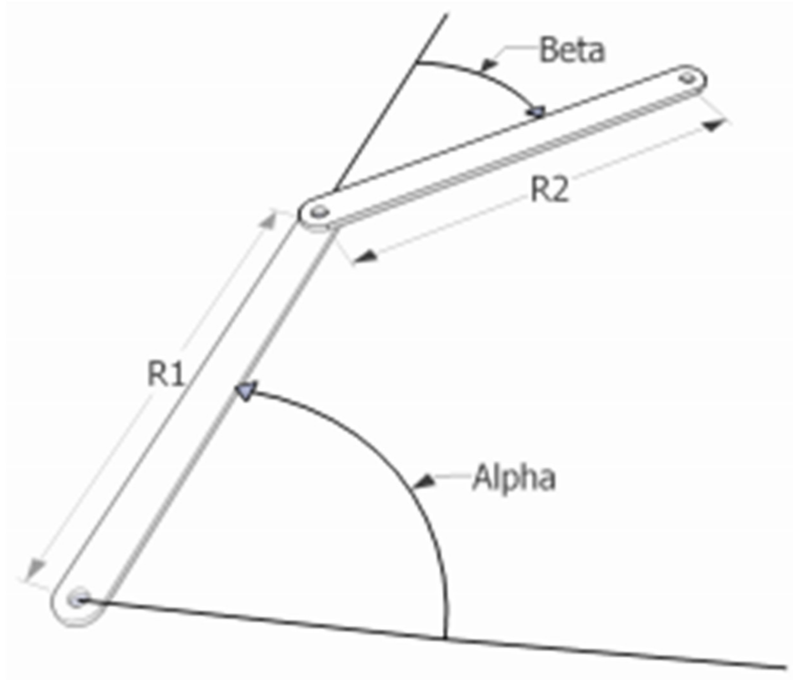



Figure 4.2.1: A two link manipulator with lengths R1 and R2, and angles alpha and beta.

This system is governed by the forward kinematic equations (4.2-107) and (4.2-108).

$$x = R1\ cos(\alpha) + R2cos(\alpha + \beta) \tag{4.2-107}$$

$$y = R1\ sin(\alpha) + R2sin(\alpha + \beta) \tag{4.2-108}$$

The control surfaces for both X and Y are shown in Figure 4.2.2. For this work, the lengths R1 and R2 are normalized to be 1 and 1 respectively. For the experiments performed, a uniform grid of size 8x8 with was used for training. A separate set of 900 data points were used for testing. All inputs and outputs were normalized to [-1,+1].

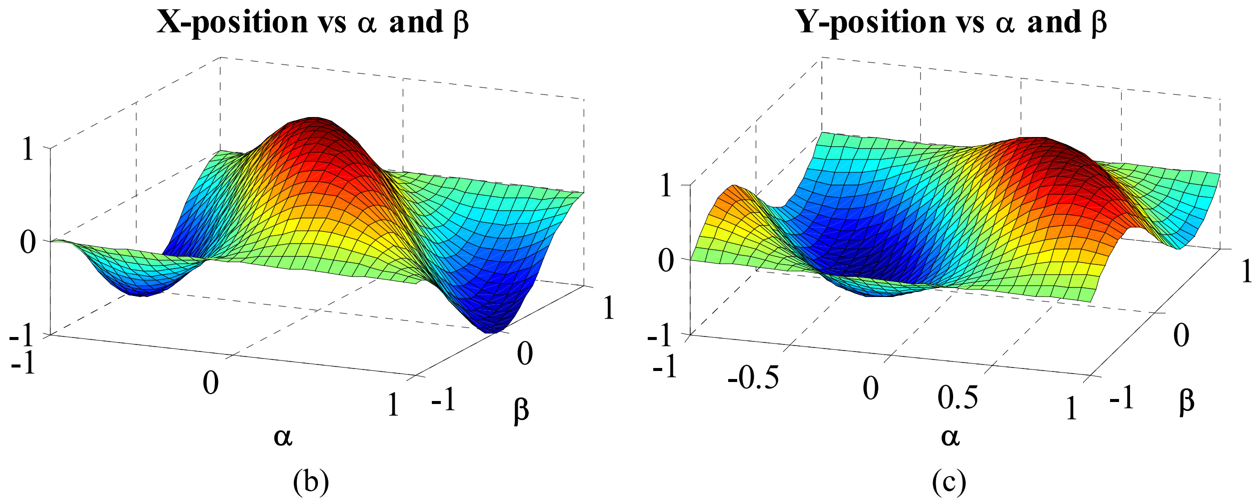



Figure 4.2.2: Output surface for the (a) X-position and (b) Y-position.

For the algorithms that require user set parameters, the following settings were used:

ANN-MLP: 10 hidden neurons.

ANN-FCC: 8 hidden neurons.

ELM: 60 RBF units.

SVR: $\gamma = 1, C = 64$

ANFIS: 4 generalized bell membership functions per input dimension.

The results for the X-position can be seen in **Table V**, and the Y-position in **Table VI**.

**Table V**: Results for Forward Kinematics problem X position in terms of Training and Testing errors (RMSE), Training and Testing Times (s), and the number nodes/neurons.

| Algorithm | Training RMSE | Testing RMSE | Training Time (s) | Testing Time(s) | Nodes |
|---|---|---|---|---|---|
| ANN-MLP | 0.1443 | 0.0888 | 13.7827 | 0.0214 | 10 |
| ANN-FCC | 0.0688 | 0.0533 | 1.7226 | 0.0570 | 8 |
| SVM | 0.0723 | 0.0376 | 0.0482 | 0.0014 | 53 |
| ELM | 0.0104 | 0.0277 | 0.4012 | 0.0876 | 60 |
| ANFIS | 0.0896 | 0.0721 | 0.1239 | 0.0846 | 16 |
| TSK | 0.0000 | 0.0352 | 0.0000 | 0.0046 | 64 |

| | | | | | |
|---|---|---|---|---|---|
| SPLINE | 0.0000 | 0.0319 | 0.0000 | 1.6829 | 64 |
| NNSA | 0.0000 | 0.0239 | 0.0000 | 0.3481 | 64 |

**Table VI**: Results for Forward Kinematics problem Y position in terms of Training and Testing errors (RMSE), Training and Testing Times (s), and the number nodes/neurons.

| Algorithm | Training RMSE | Testing RMSE | Training Time (s) | Testing Time(s) | Nodes |
|---|---|---|---|---|---|
| ANN-MLP | 0.0980 | 0.0740 | 12.7502 | 0.0210 | 20 |
| ANN-FCC | 0.0765 | 0.0486 | 2.0770 | 0.1326 | 8 |
| SVM | 0.0585 | 0.0332 | 0.0315 | 0.0012 | 43 |
| ELM | 0.0092 | 0.0155 | 0.4696 | 0.1086 | 60 |
| ANFIS | 0.0329 | 0.0453 | 0.1021 | 0.0658 | 16 |
| TSK | 0.0000 | 0.0431 | 0.0000 | 0.0046 | 64 |
| SPLINE | 0.0000 | 0.0198 | 0.0000 | 1.6863 | 64 |
| NNSA | 0.0000 | 0.0127 | 0.0000 | 0.3525 | 64 |

Again, we can see that the neural networks require a lengthy training process, and the SVM, ELM, and ANFIS are much faster. TSK, SPLINE, and NNSA do not require training, although NNSA and SPLINE require a relatively lengthy testing time. The MLP network has the poorest performance, both in training time and testing error. For both X and Y outputs, NNSA performs very well, producing the lowest testing error of any algorithm. In contrast, the TSK FS has a higher testing error, but it also has the lowest execution time.

### 4.3 Multidimensional Schwefel Function

The Schwefel [48] function is highly complex nonlinear benchmark function, and can be extended to arbitrary dimensions. For the purposes of this research, the Schwefel function was modified so that the sine function is used instead of cosine, a parameter called alpha to control nonlinearity was added, and inputs and outputs were normalized to the range (-1,1). The function is described by (4.3-109).

$$y = \sum_{i=1}^{d} -x_i * \alpha * \sin\left(\sqrt{|x_i * \alpha|}\right) \tag{4.3-109}$$

Note that $x$ is a vector of length $d$, and each element $x_i$ corresponds to a different dimension. The alpha parameter allows the shape of the resulting function to be changed, as shown in Figure 4.3.1.

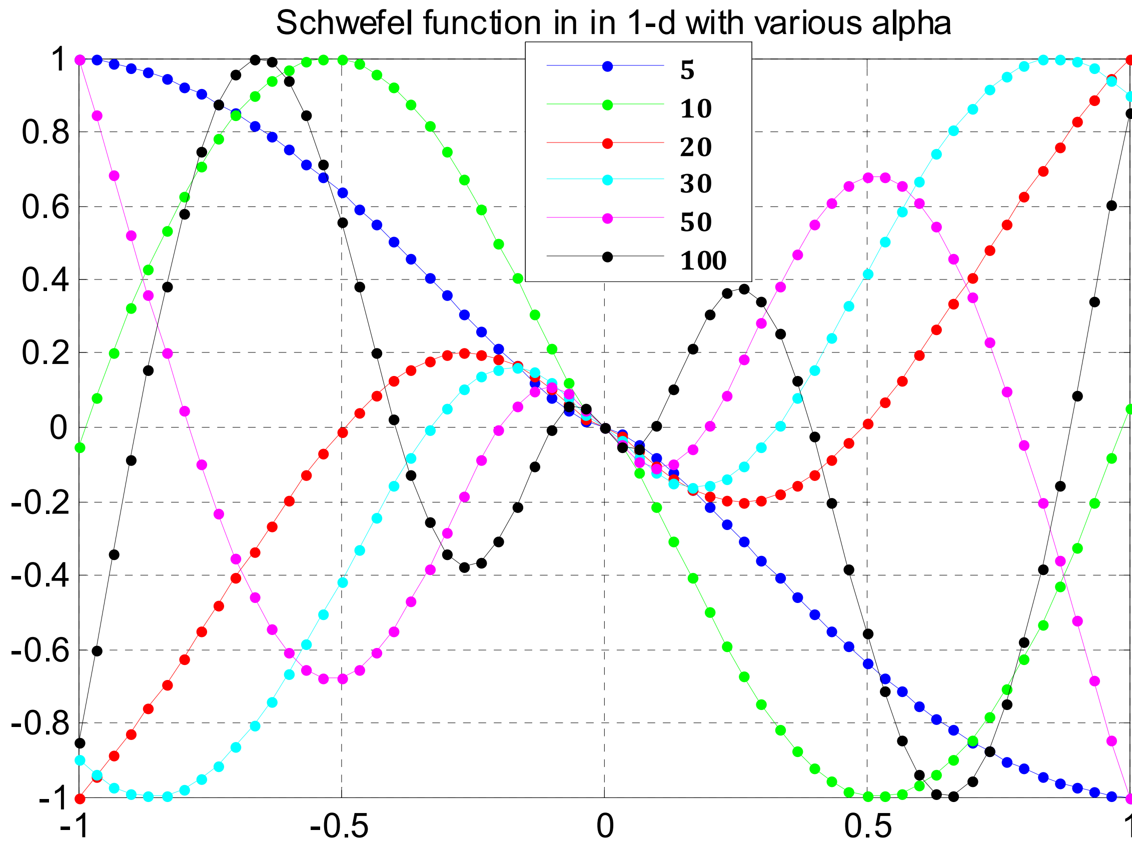
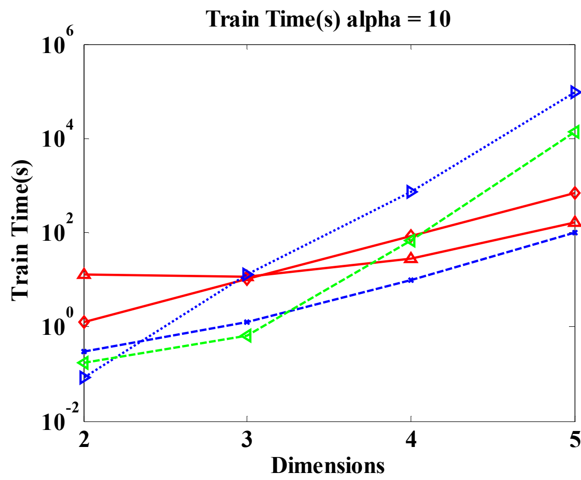



Figure 4.3.1: Schwefel function with different alpha values for one dimensional cases for different values of Alpha (Alpha = 10, Alpha = 20, Alpha = 30, Alpha = 50, Alpha = 100.)

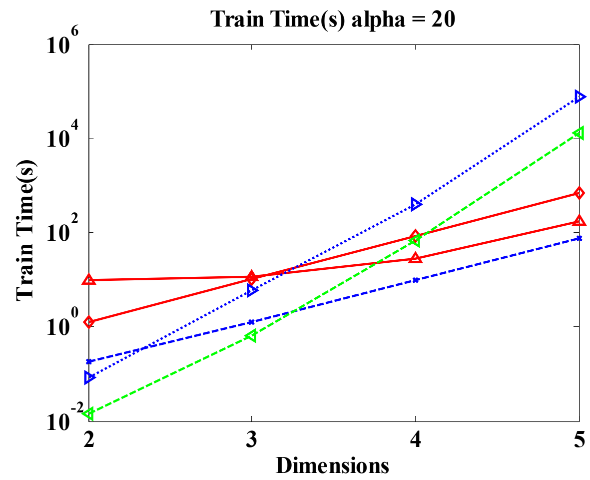
The NNSA algorithm was compared to the same machine learning algorithms listed in the previous section using the Schwefel function in multiple dimensions with several different uniform training grid sizes. For each training grid size N, an N-1 grid of test points was generated. The test point grid was offset from the training grid so that each test point resided in the center of a hypercube defined by the closest training points. This meant that the test points

were as far as possible from the available training points. The purpose of this was to test the ability of the algorithms to generalize between the available data points.

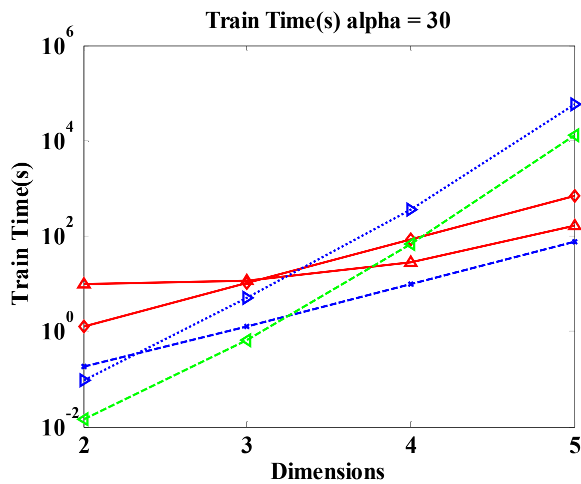
The performance of the algorithms was evaluated by comparing the training and testing times (Figure 4.3.2 and Figure 4.3.3), as well as the training and testing error (Figure 4.3.5 and Figure 4.3.6). Selected graphs have been included in this section for analysis of the data. For the full listing, see Multidimensional Schwefel Data in Appendix 6.1.



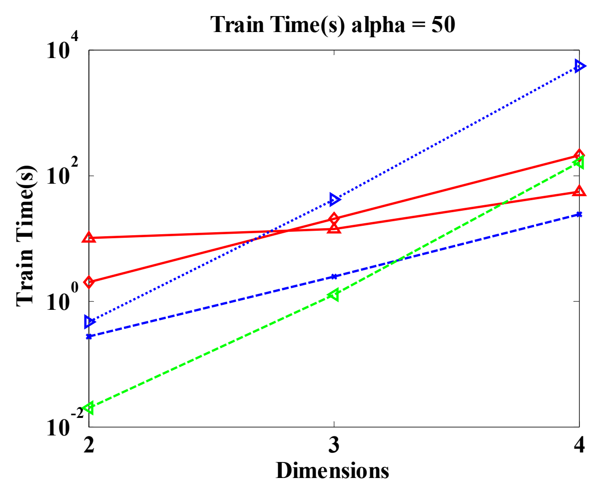

(a)



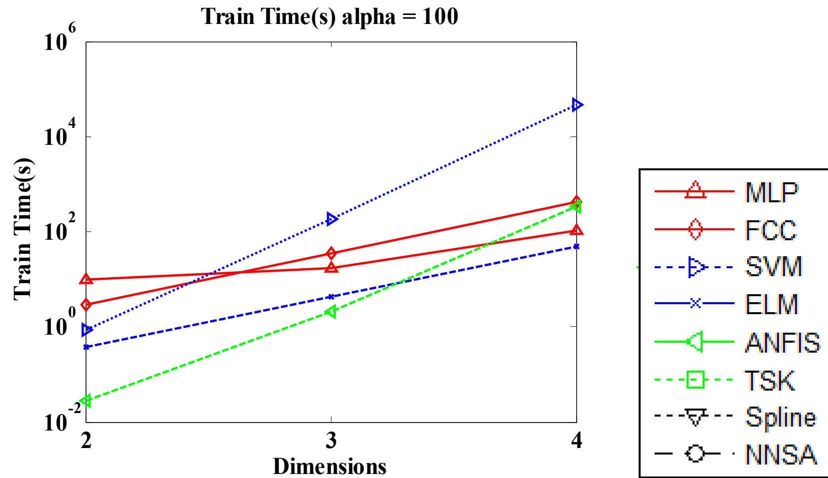

(b)




(c)

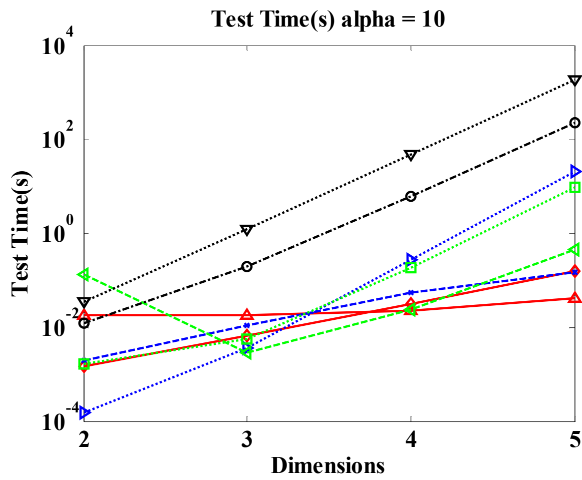



(d)

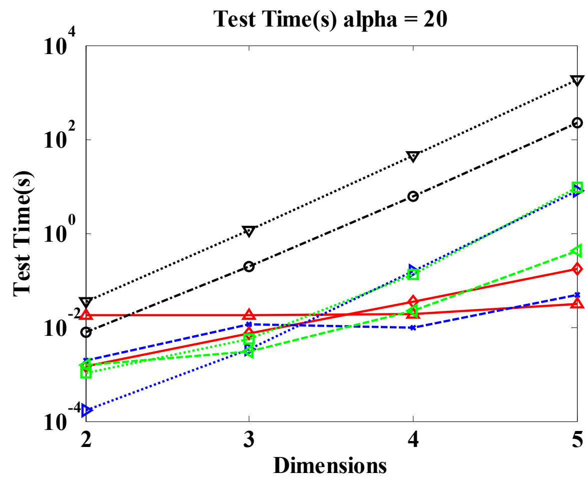



(e)

Figure 4.3.2: Semi-log plots showing training times for the algorithms tested as the number of dimensions increased. (a) Alpha = 10 (b) Alpha = 20 (c) Alpha = 30 (d) Alpha = 50 (e) Alpha = 100.




(a)




(b)

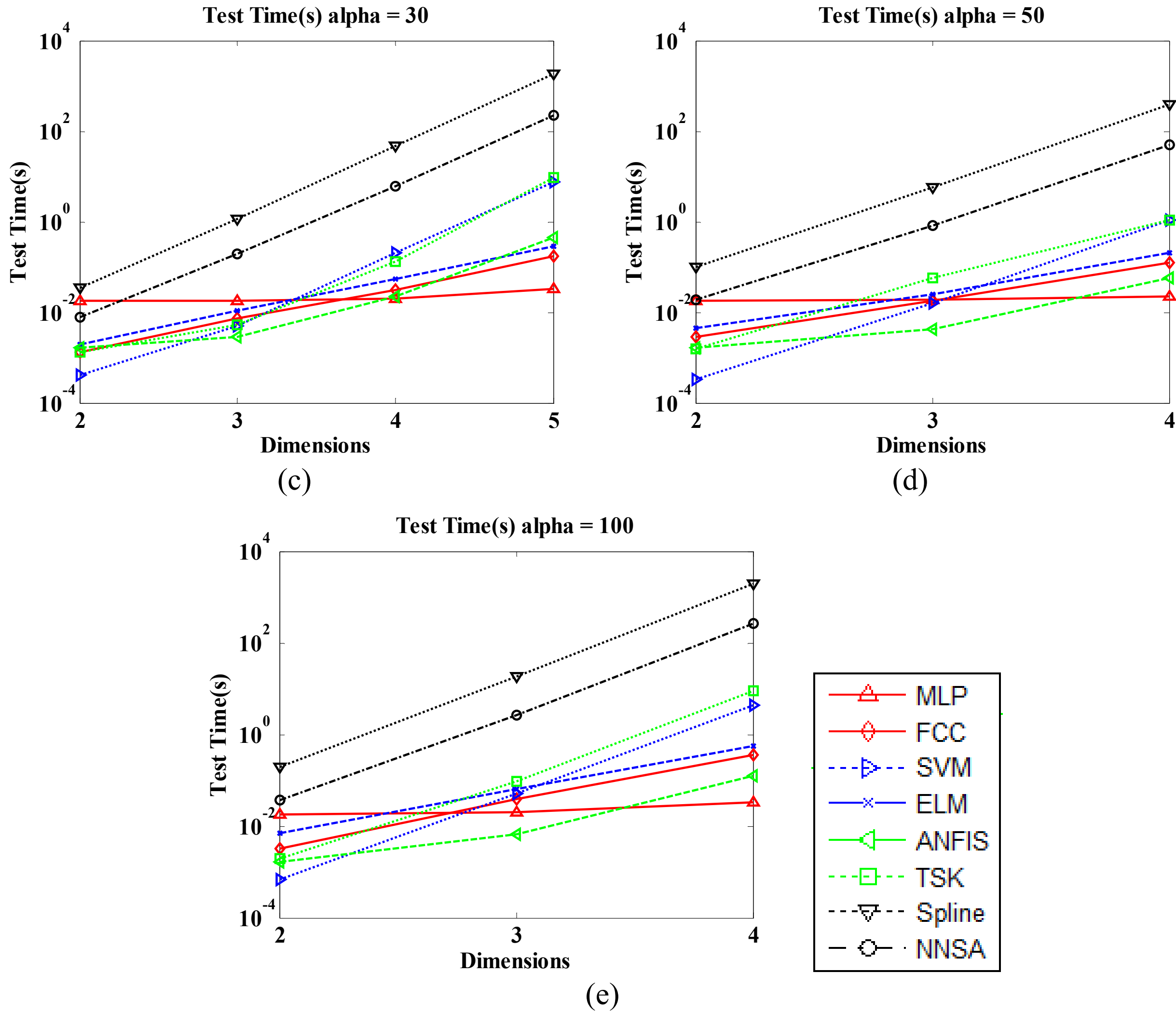
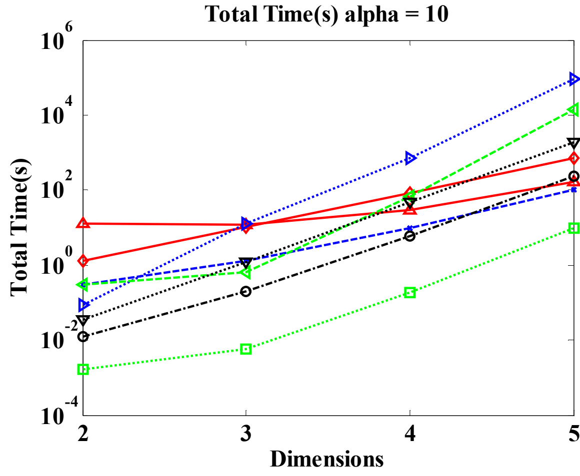
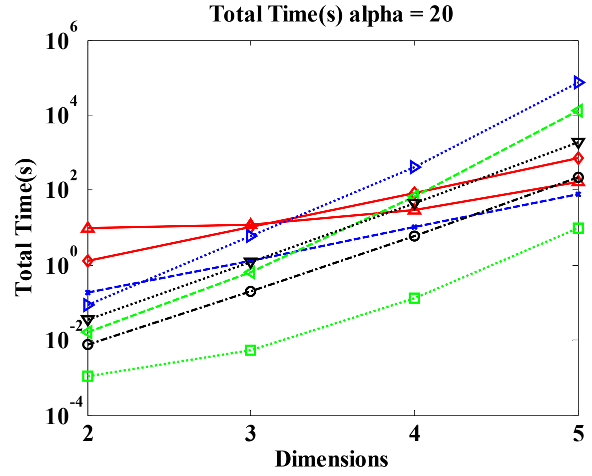


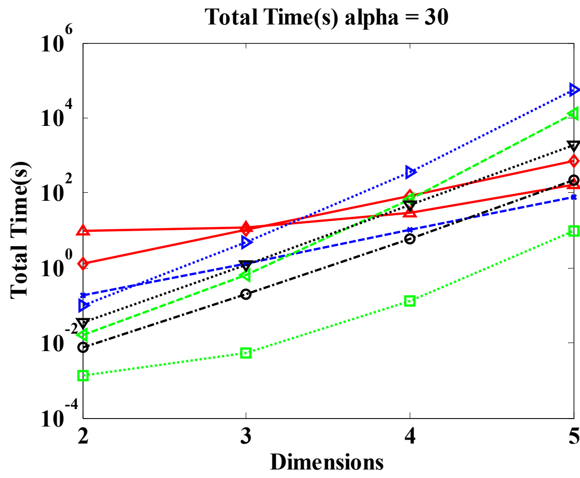

Figure 4.3.3: Semi-log plots showing testing times for the algorithms tested as the number of dimensions increased. (a) Alpha = 10 (b) Alpha = 20 (c) Alpha = 30 (d) Alpha = 50 (e) Alpha = 100.



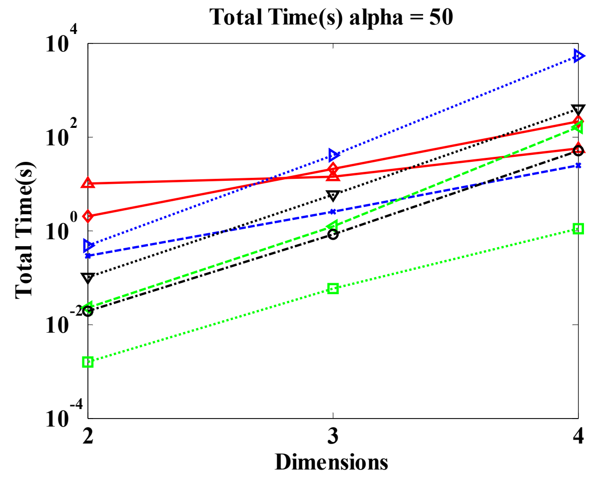

(a)



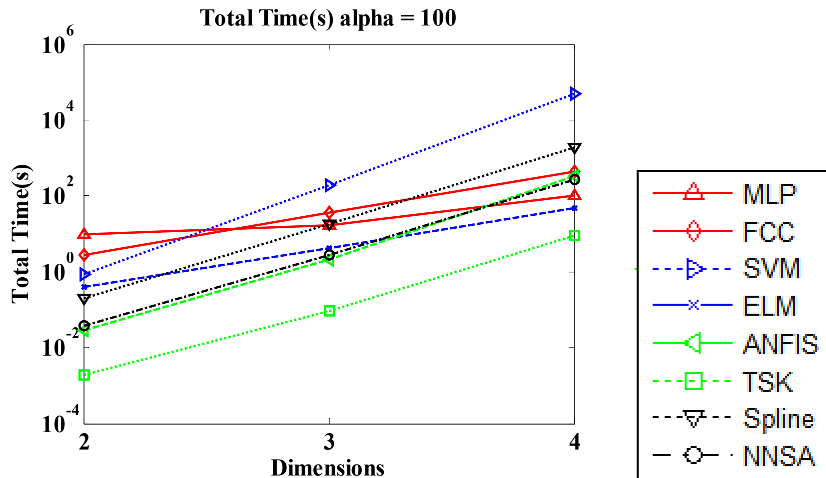

(b)




(c)

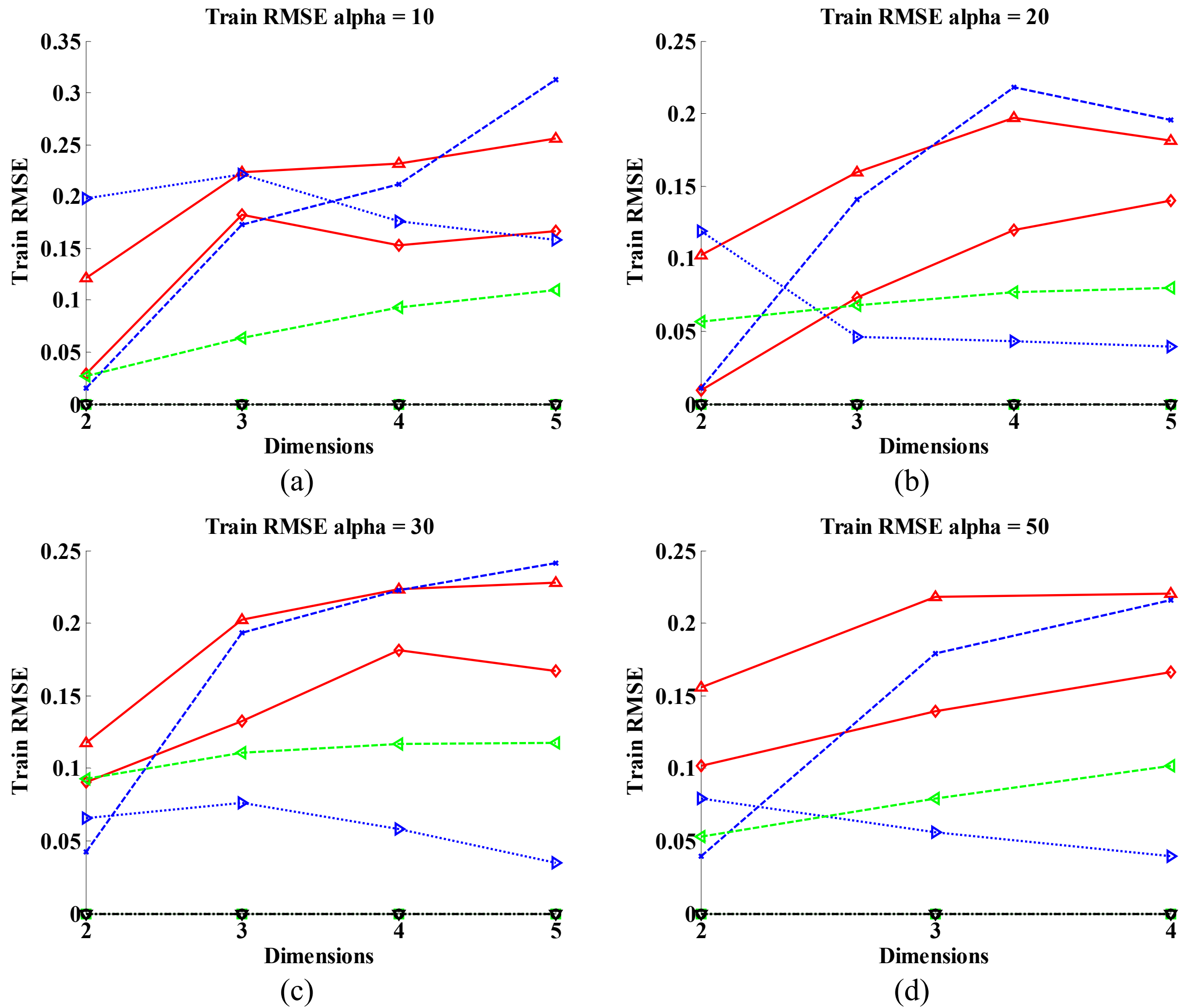


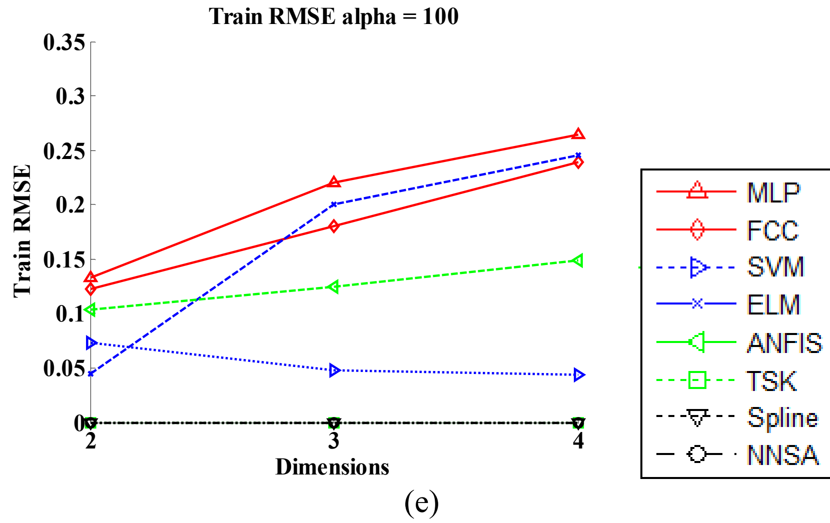

(d)




(e)

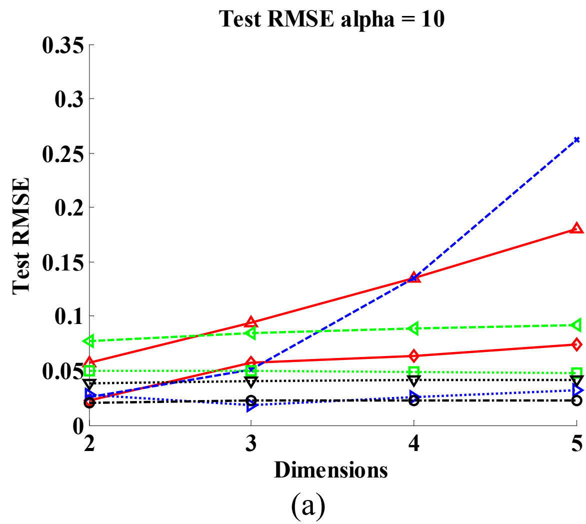
Figure 4.3.4: Semi-log plots showing total times for the algorithms tested as the number of dimensions increased. (a) Alpha = 10 (b) Alpha = 20 (c) Alpha = 30 (d) Alpha = 50 (e) Alpha = 100.



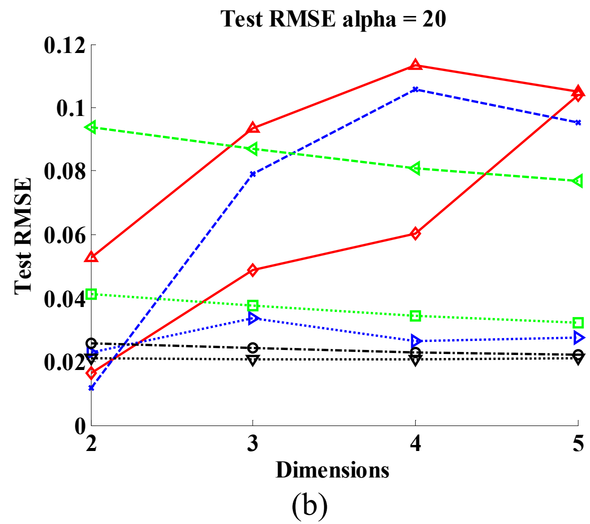



(e)

Figure 4.3.5: Plots showing training errors for the algorithms tested as the number of dimensions increased. (a) Alpha = 10 (b) Alpha = 20 (c) Alpha = 30 (d) Alpha = 50 (e) Alpha = 100.




(a)




(b)

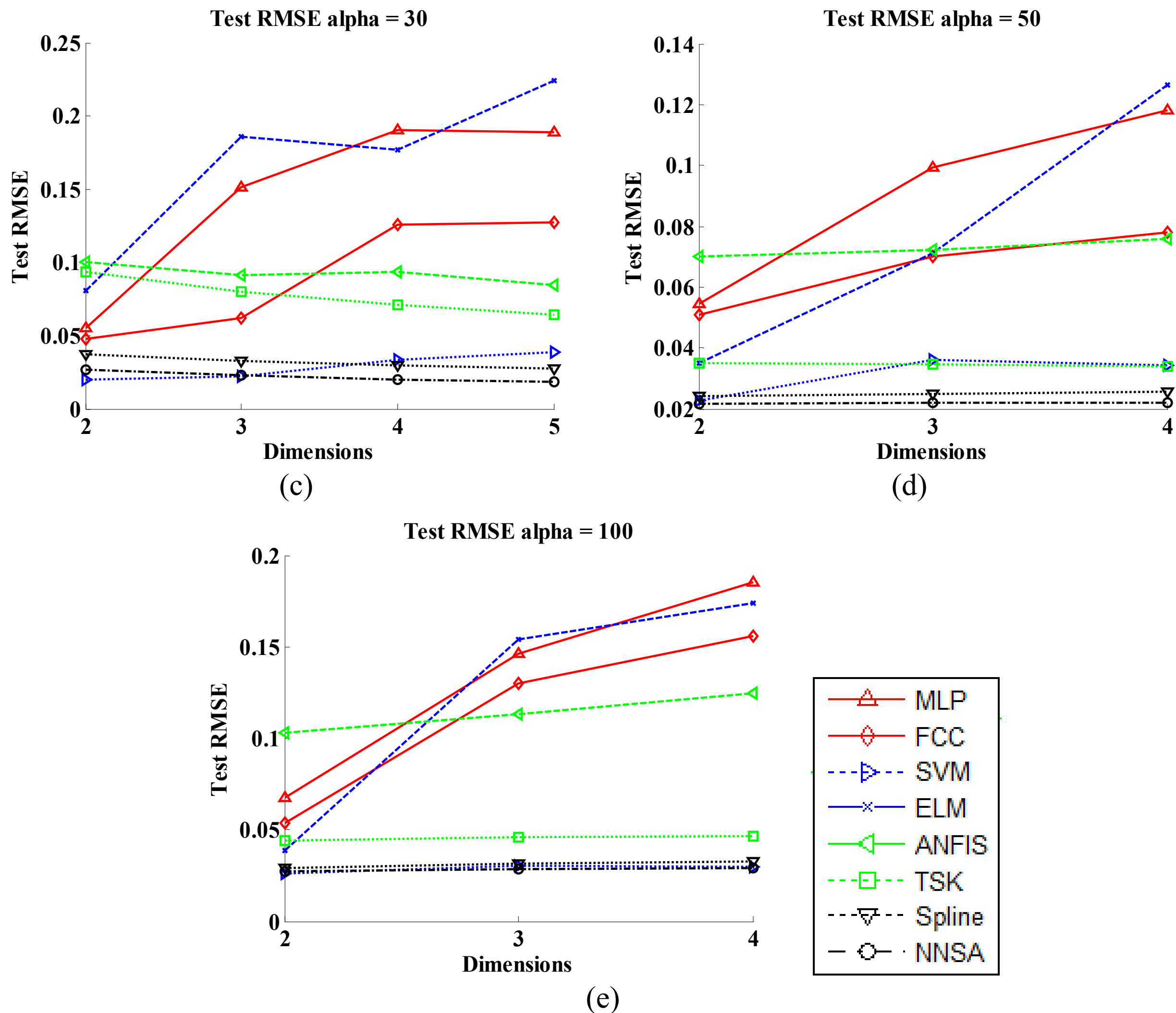



Figure 4.3.6: Plots showing training errors for the algorithms tested as the number of dimensions increased. (a) Alpha = 10 (b) Alpha = 20 (c) Alpha = 30 (d) Alpha = 50 (e) Alpha = 100.

The training and testing times can be seen in Figure 4.3.2 and Figure 4.3.3, respectively. The SPLINE, TSKFS, and NNSA algorithms do not require training, and cannot be seen in Figure 4.3.2. The training times of all algorithms increase as the number of dimensions increases, which makes sense seeing as the number of training points is growing exponentially with each additional dimension. Note that the plots use a log scale on the Y-axis, so the linear appearance of the trends is in fact exponential growth. Figure 4.3.2(a)–(c) are very similar because the same number of points in each dimension were used. Figure 4.3.2(d) and (e) differ slightly as more

training points were generated to account for the increasing complexity of the function. As a result, Figure 4.3.2(d) and (e) stop at the fourth dimension, as the simulations for the fifth dimension required an infeasible amount of time to complete.

The testing times in Figure 4.3.3(a)–(e) also increase as the number of dimensions increase, although the MLP network appears nearly constant. The only exception is ANFIS in Figure 4.3.3(a). The fact that this behavior does not repeat with any of the other values of alpha indicates that this is an anomaly. It is clear that SPLINE performance suffers the most as the number of dimensions increases, followed by NNSA and TSK. This is a result of the memory and computation requirements of each algorithm that scale with the number of dimensions.

A fairer comparison of the algorithms can be seen in Figure 4.3.4(a)–(e), which shows the total time as a sum of training and testing time for each algorithm. Several trends are apparent. First, the total time for all algorithms increases exponentially (recall that the scale for the Y-axis is semi-logarithmic) as the number of dimensions increases. The fastest algorithm appears to be the TSKFS, which is a result of simplicity of the defuzzification process when compared with NNSA. SVM performs the slowest, probably due to the necessity of searching for appropriate parameters for each training set. Lastly, other than TSKFS, NNSA performs the fastest, although it suffers as the number of dimensions increases.

The training errors of the tested algorithms can be seen in Figure 4.3.5(a)–(e). Training error gives a measurement of how well an algorithm has matched the training data. The training error of TSKFS, SPLINE, and NNSA are all zero, as they match the training points by design. For the other algorithms, the training errors generally increase as the number of dimensions increase, with the exception of SVM, with which training errors actually decrease. The errors also appear

fairly consistent as alpha increases, with the MLP network and ELM performing the worst, FCC and ANFIS in the middle, and SVM generally performing the best.

Perhaps the most important performance criterion is testing or validation error, shown in Figure 4.3.6(a)–(e). This measures how an algorithm performs on data that was not used for training. There are several interesting trends to notice. The errors for all the algorithms are fairly close when there are only two input dimensions. The errors for MLP, FCC, and ELM generally increase as the number of dimensions increases, especially compared to the other algorithms. In every case, NNSA produces either the lowest, or close to the lowest errors.

# Chapter 5 Conclusions

The power of modern computing has the potential to tackle engineering and scientific problems that were infeasible only a few years ago. Even with this potential, some tasks are simply too complex for traditional first principles analysis. Data driven machine learning techniques represent a different paradigm for applying the capabilities of computers. By drawing inspiration from the biological world, machine learning seeks to recreate the problem-solving ability of living organisms. The varieties of ANN and FS all have advantages and drawbacks. Questions such as network size, architecture, training algorithm, number, and type of membership function, rule base, etc. can lead to frustration. The NNSA algorithm presented in this work offers another tool for researchers to utilize.

The experimental results demonstrate the capabilities of NNSA when compared to other machine leaning methods. It is clear that the performance of NNSA suffers as the number of input dimensions grows. Further work remains to be done to make the algorithm viable in higher dimensions. In addition, the grid-based nature of the algorithm makes it unsuitable for some problems. However, the smoothness and accuracy of the NNSA algorithm is superior to

traditional TSK FS, and of similar quality to ANN, even without optimization. The quality of the output surfaces produced by NNSA approaches that of cubic splines, while avoiding the computational cost associated with traditional splines, such as solving for the entire surface by matrix inversion. The forward-only nature of the computation means that new training data can be easily incorporated through the adjustment of the appropriate node values, thus avoiding the computationally intensive retraining process required by learning systems. In theory, this also allows NNSA to be tuned to produce more accurate results, as it is a simple matter to trace inaccurate output values back to the associated node values, which can then be modified to reduce the error.

## References

[1] K. Hornik, “Approximation Capabilities of Multilayer Feedforward Networks,” *Neural Netw*, vol. 4, no. 2, pp. 251–257, Mar. 1991.

[2] B. Kosko, “Fuzzy systems as universal approximators,” *IEEE Trans. Comput.*, vol. 43, no. 11, pp. 1329–1333, Nov. 1994.

[3] E. Kayacan, E. Kayacan, and M. A. Khanesar, “Identification of Nonlinear Dynamic Systems Using Type-2 Fuzzy Neural Networks -A Novel Learning Algorithm and a Comparative Study,” *IEEE Trans. Ind. Electron.*, vol. 62, no. 3, pp. 1716–1724, Mar. 2015.

[4] S. Simani, S. Farsoni, and P. Castaldi, “Fault Diagnosis of a Wind Turbine Benchmark via Identified Fuzzy Models,” *IEEE Trans. Ind. Electron.*, vol. 62, no. 6, pp. 3775–3782, Jun. 2015.

[5] F. Lin, K. Lu, T. Ke, B. Yang, and Y. Chang, “Reactive Power Control of Three-Phase Grid-Connected PV System during Grid Faults Using Takagi-Sugeno-Kang Probabilistic Fuzzy Neural Network Control,” *IEEE Trans. Ind. Electron.*, vol. PP, no. 99, pp. 1–1, 2015.

[6] J. Yu, P. Shi, W. Dong, and H. Yu, “Observer and Command Filter-based Adaptive Fuzzy Output Feedback Control of Uncertain Nonlinear Systems,” *IEEE Trans. Ind. Electron.*, vol. PP, no. 99, pp. 1–1, 2015.

[7] R. Wai, M. Chen, and Y. Liu, “Design of Adaptive Control and Fuzzy Neural Network Control for Single-Stage Boost Inverter,” *IEEE Trans. Ind. Electron.*, vol. PP, no. 99, pp. 1–1, 2015.

[8] C. Lin, Y. Chang, C. Hung, C. Tu, and C. Chuang, “Position Estimation and Smooth Tracking with a Fuzzy Logic-Based Adaptive Strong Tracking Kalman Filter for Capacitive Touch Panels,” *IEEE Trans. Ind. Electron.*, vol. PP, no. 99, pp. 1–1, 2015.

[9] H. H. Choi, H. M. Yun, and Y. Kim, “Implementation of Evolutionary Fuzzy PID Speed Controller for PM Synchronous Motor,” *IEEE Trans. Ind. Inform.*, vol. 11, no. 2, pp. 540–547, Apr. 2015.

[10] D. Q. Dang, Y. Choi, H. H. Choi, and J. Jung, “Experimental Validation of a Fuzzy Adaptive Voltage Controller for Three-Phase PWM Inverter of a Standalone DG Unit,” *IEEE Trans. Ind. Inform.*, vol. 11, no. 3, pp. 632–641, Jun. 2015.

[11] Q. Jia, W. Chen, Y. Zhang, and H. Li, “Fault Reconstruction and Fault-Tolerant Control via Learning Observers in Takagi-Sugeno Fuzzy Descriptor Systems With Time Delays,” *IEEE Trans. Ind. Electron.*, vol. 62, no. 6, pp. 3885–3895, Jun. 2015.

[12] F. Luo *et al.*, "Advanced Pattern Discovery-based Fuzzy Classification Method for Power System Dynamic Security Assessment," *IEEE Trans. Ind. Inform.*, vol. 11, no. 2, pp. 416–426, Apr. 2015.

[13] C.-F. Juang, Y.-H. Chen, and Y.-H. Jhan, "Wall-Following Control of a Hexapod Robot Using a Data-Driven Fuzzy Controller Learned Through Differential Evolution," *IEEE Trans. Ind. Electron.*, vol. 62, no. 1, pp. 611–619, Jan. 2015.

[14] Z. Li, C.-Y. Su, L. Wang, Z. Chen, and T. Chai, "Nonlinear Disturbance Observer-Based Control Design for a Robotic Exoskeleton Incorporating Fuzzy Approximation," *IEEE Trans. Ind. Electron.*, vol. 62, no. 9, pp. 5763–5775, Sep. 2015.

[15] T. Xie, H. Yu, J. Hewlett, P. Rozycki, and B. Wilamowski, "Fast and Efficient Second-Order Method for Training Radial Basis Function Networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 23, no. 4, pp. 609–619, Apr. 2012.

[16] H. Yu, P. D. Reiner, T. Xie, T. Bartczak, and B. M. Wilamowski, "An Incremental Design of Radial Basis Function Networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 25, no. 10, pp. 1793–1803, Oct. 2014.

[17] B. M. Wilamowski, "Neural network architectures and learning algorithms," *IEEE Ind. Electron. Mag.*, vol. 3, no. 4, pp. 56–63, Dec. 2009.

[18] D. Hunter, H. Yu, M. S. Pukish, J. Kolbusz, and B. M. Wilamowski, "Selection of Proper Neural Network Sizes and Architectures-A Comparative Study," *IEEE Trans. Ind. Inform.*, vol. 8, no. 2, pp. 228–240, May 2012.

[19] B. M. Wilamowski and H. Yu, "Neural Network Learning Without Backpropagation," *IEEE Trans. Neural Netw.*, vol. 21, no. 11, pp. 1793–1803, Nov. 2010.

[20] T. Takagi and M. Sugeno, "Fuzzy identification of systems and its applications to modeling and control," *IEEE Trans. Syst. Man Cybern.*, vol. SMC-15, no. 1, pp. 116–132, Jan. 1985.

[21] M. Sugeno and G. T. Kang, "Structure Identification of Fuzzy Model," *Fuzzy Sets Syst*, vol. 28, no. 1, pp. 15–33, Oct. 1988.

[22] H. Ying, "General SISO Takagi-Sugeno fuzzy systems with linear rule consequent are universal approximators," *IEEE Trans. Fuzzy Syst.*, vol. 6, no. 4, pp. 582–587, Nov. 1998.

[23] J. Richardson, P. Reiner, and B. M. Wilamowski, "Cubic spline as an alternative to methods of machine learning," in *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, 2015, pp. 110–115.

[24] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bull. Math. Biophys.*, vol. 5, no. 4, pp. 115–133, Dec. 1943.

[25] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proc. Natl. Acad. Sci.*, vol. 79, no. 8, pp. 2554–2558, Apr. 1982.

[26] B. M. Wilamowski, H. Yu, and K. T. Chung, “Parity-N Problems as a Vehicle to Compare Efficiency of Neural Network Architectures,” in *Industrial Electronics Handbook*, 2nd ed., vol. 5, CRC Press, pp. 10-1 to 10-2.

[27] T. Kohonen, “Self-organized formation of topologically correct feature maps,” *Biol. Cybern.*, vol. 43, no. 1, pp. 59–69, Jan. 1982.

[28] Werbos and P. J. (Paul John, “Beyond regression : new tools for prediction and analysis in the behavioral sciences /,” *ResearchGate*, Jan. 1974.

[29] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986.

[30] C. T. Kim and J. J. Lee, “Training Two-Layered Feedforward Networks With Variable Projection Method,” *IEEE Trans. Neural Netw.*, vol. 19, no. 2, pp. 371–375, Feb. 2008.

[31] V. V. Phansalkar and P. S. Sastry, “Analysis of the back-propagation algorithm with momentum,” *IEEE Trans. Neural Netw.*, vol. 5, no. 3, pp. 505–506, May 1994.

[32] M. Riedmiller and H. Braun, “A direct adaptive method for faster backpropagation learning: the RPROP algorithm,” in *IEEE International Conference on Neural Networks*, 1993, pp. 586–591 vol.1.

[33] K. Levenberg, “A method for the solution of certain non-linear problems in least squares,” *Q. Appl. Math.*, vol. 2, pp. 164–168, 1944.

[34] M. T. Hagan and M.-B. Menhaj, “Training feedforward networks with the Marquardt algorithm,” *IEEE Trans. Neural Netw.*, vol. 5, no. 6, pp. 989–993, 1994.

[35] J. Moody and C. J. Darken, “Fast Learning in Networks of Locally-tuned Processing Units,” *Neural Comput*, vol. 1, no. 2, pp. 281–294, Jun. 1989.

[36] G.-B. Huang, Z. Bai, L. L. C. Kasun, and C. M. Vong, “Local Receptive Fields Based Extreme Learning Machine,” *IEEE Comput. Intell. Mag.*, vol. 10, no. 2, pp. 18–29, May 2015.

[37] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, “Extreme learning machine: Theory and applications,” *Neurocomputing*, vol. 70, no. 1–3, pp. 489–501, Dec. 2006.

[38] S. Y. Wong, K. S. Yap, H. J. Yap, S. C. Tan, and S. W. Chang, “On Equivalence of FIS and ELM for Interpretable Rule-Based Knowledge Representation,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 26, no. 7, pp. 1417–1430, Jul. 2015.

[39] V. N. Vapnik, “An overview of statistical learning theory,” *IEEE Trans. Neural Netw.*, vol. 10, no. 5, pp. 988–999, Sep. 1999.

[40] A. J. Smola and B. Schölkopf, “A tutorial on support vector regression,” *Stat. Comput.*, vol. 14, no. 3, pp. 199–222, Aug. 2004.

[41] L. A. Zadeh, “Fuzzy sets,” *Inf. Control*, vol. 8, no. 3, pp. 338–353, Jun. 1965.

[42] E. H. Mamdani, “Application of Fuzzy Logic to Approximate Reasoning Using Linguistic Synthesis,” *IEEE Trans. Comput.*, vol. C-26, no. 12, pp. 1182–1191, Dec. 1977.

[43] J.-S. R. Jang, “ANFIS: adaptive-network-based fuzzy inference system,” *IEEE Trans. Syst. Man Cybern.*, vol. 23, no. 3, pp. 665–685, May 1993.

[44] J.-S. R. Jang and C.-T. Sun, “Neuro-fuzzy modeling and control,” *Proc. IEEE*, vol. 83, no. 3, pp. 378–406, Mar. 1995.

[45] J.-S. R. Jang and C.-T. Sun, “Functional equivalence between radial basis function networks and fuzzy inference systems,” *IEEE Trans. Neural Netw.*, vol. 4, no. 1, pp. 156–159, Jan. 1993.

[46] C. de Boor, “Efficient Computer Manipulation of Tensor Products,” *ACM Trans Math Softw*, vol. 5, no. 2, pp. 173–182, Jun. 1979.

[47] J. Richardson, J. Korniak, P. D. Reiner, and B. M. Wilamowski, “Nearest-Neighbor Spline Approximation (NNSA) Improvement to TSK Fuzzy Systems,” *IEEE Trans. Ind. Inform.*, vol. 12, no. 1, pp. 169–178, Feb. 2016.

[48] H.-P. Schwefel, *Numerical Optimization of Computer Models*. New York, NY, USA: John Wiley & Sons, Inc., 1981.

# Chapter 6 Appendices

## 6.1 Multidimensional Schwefel Data

| Algorithm | Dimension | Alpha | Train Time (s) | Test Time (s) | Total Time (s) | Train RMSE | Test RMSE | Nodes |
|---|---|---|---|---|---|---|---|---|
| MLP | 2 | 10 | 12.94021 | 0.017668 | 12.95788 | 0.121646 | 0.057196 | 15 |
| FCC | 2 | 10 | 1.277369 | 0.001482 | 1.278852 | 0.029302 | 0.022392 | 88 |
| SVM | 2 | 10 | 0.084975 | 0.000155 | 0.08513 | 0.197627 | 0.027843 | 26 |
| ELM | 2 | 10 | 0.297913 | 0.001967 | 0.29988 | 0.015431 | 0.025549 | 50 |
| ANFIS | 2 | 10 | 0.169048 | 0.135201 | 0.304249 | 0.02677 | 0.07732 | 9 |
| TSK | 2 | 10 | 0 | 0.001662 | 0.001662 | 0 | 0.049552 | 36 |
| Spline | 2 | 10 | 0 | 0.034441 | 0.034441 | 0 | 0.038061 | 64 |
| NNSA | 2 | 10 | 0 | 0.012179 | 0.012179 | 0 | 0.020946 | 64 |
| MLP | 2 | 20 | 9.533187 | 0.018249 | 9.551436 | 0.102822 | 0.052718 | 10 |
| FCC | 2 | 20 | 1.275765 | 0.001484 | 1.277249 | 0.009014 | 0.016296 | 88 |
| SVM | 2 | 20 | 0.085635 | 0.000172 | 0.085808 | 0.119255 | 0.022908 | 26 |
| ELM | 2 | 20 | 0.186096 | 0.001962 | 0.188058 | 0.010574 | 0.011705 | 50 |
| ANFIS | 2 | 20 | 0.014293 | 0.001537 | 0.01583 | 0.056705 | 0.093826 | 9 |
| TSK | 2 | 20 | 0 | 0.00107 | 0.00107 | 0 | 0.04136 | 36 |
| Spline | 2 | 20 | 0 | 0.034048 | 0.034048 | 0 | 0.021066 | 64 |
| NNSA | 2 | 20 | 0 | 0.00789 | 0.00789 | 0 | 0.025755 | 64 |
| MLP | 2 | 30 | 9.553677 | 0.017545 | 9.571222 | 0.117325 | 0.055211 | 15 |
| FCC | 2 | 30 | 1.282617 | 0.001323 | 1.28394 | 0.090229 | 0.047413 | 75 |
| SVM | 2 | 30 | 0.095954 | 0.000419 | 0.096373 | 0.06564 | 0.020103 | 50 |
| ELM | 2 | 30 | 0.183874 | 0.001968 | 0.185842 | 0.042202 | 0.080674 | 50 |
| ANFIS | 2 | 30 | 0.014358 | 0.001606 | 0.015964 | 0.092787 | 0.099857 | 9 |
| TSK | 2 | 30 | 0 | 0.001315 | 0.001315 | 0 | 0.093282 | 36 |
| Spline | 2 | 30 | 0 | 0.034027 | 0.034027 | 0 | 0.036846 | 64 |
| NNSA | 2 | 30 | 0 | 0.007869 | 0.007869 | 0 | 0.026711 | 64 |
| MLP | 2 | 50 | 9.866466 | 0.017431 | 9.883896 | 0.155482 | 0.054535 | 10 |
| FCC | 2 | 50 | 2.017227 | 0.00284 | 2.020067 | 0.10179 | 0.050774 | 88 |
| SVM | 2 | 50 | 0.466546 | 0.000331 | 0.466877 | 0.079549 | 0.022621 | 58 |
| ELM | 2 | 50 | 0.274767 | 0.004475 | 0.279242 | 0.039195 | 0.035099 | 60 |
| ANFIS | 2 | 50 | 0.020172 | 0.001613 | 0.021785 | 0.053061 | 0.070035 | 9 |
| TSK | 2 | 50 | 0 | 0.001529 | 0.001529 | 0 | 0.03492 | 64 |
| Spline | 2 | 50 | 0 | 0.097538 | 0.097538 | 0 | 0.023999 | 100 |
| NNSA | 2 | 50 | 0 | 0.018577 | 0.018577 | 0 | 0.02154 | 100 |
| MLP | 2 | 100 | 9.918065 | 0.018211 | 9.936276 | 0.13268 | 0.067406 | 15 |
| FCC | 2 | 100 | 2.861763 | 0.003218 | 2.864981 | 0.122154 | 0.053686 | 52 |
| SVM | 2 | 100 | 0.877246 | 0.000683 | 0.877929 | 0.07292 | 0.025967 | 80 |
| ELM | 2 | 100 | 0.379914 | 0.007136 | 0.38705 | 0.044361 | 0.038643 | 60 |
| ANFIS | 2 | 100 | 0.027089 | 0.001683 | 0.028772 | 0.103398 | 0.103106 | 9 |
| TSK | 2 | 100 | 0 | 0.001971 | 0.001971 | 0 | 0.044214 | 100 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Spline | 2 | 100 | 0 | 0.195435 | 0.195435 | 0 | 0.029356 | 144 |
| NNSA | 2 | 100 | 0 | 0.037775 | 0.037775 | 0 | 0.027175 | 144 |
| MLP | 3 | 10 | 11.67602 | 0.017682 | 11.69371 | 0.223443 | 0.094427 | 15 |
| FCC | 3 | 10 | 10.43696 | 0.006565 | 10.44353 | 0.182259 | 0.056854 | 85 |
| SVM | 3 | 10 | 12.99226 | 0.003508 | 12.99577 | 0.221449 | 0.018689 | 220 |
| ELM | 3 | 10 | 1.28128 | 0.011016 | 1.292296 | 0.172382 | 0.050919 | 60 |
| ANFIS | 3 | 10 | 0.643649 | 0.002904 | 0.646553 | 0.063526 | 0.084563 | 27 |
| TSK | 3 | 10 | 0 | 0.00566 | 0.00566 | 0 | 0.049921 | 216 |
| Spline | 3 | 10 | 0 | 1.217751 | 1.217751 | 0 | 0.04016 | 512 |
| NNSA | 3 | 10 | 0 | 0.198288 | 0.198288 | 0 | 0.022087 | 512 |
| MLP | 3 | 20 | 11.6582 | 0.017831 | 11.67603 | 0.159723 | 0.093526 | 15 |
| FCC | 3 | 20 | 10.40526 | 0.007341 | 10.4126 | 0.073409 | 0.048747 | 99 |
| SVM | 3 | 20 | 6.064509 | 0.003309 | 6.067819 | 0.046382 | 0.033754 | 202 |
| ELM | 3 | 20 | 1.285099 | 0.011284 | 1.296383 | 0.140609 | 0.079254 | 60 |
| ANFIS | 3 | 20 | 0.640177 | 0.003091 | 0.643268 | 0.068292 | 0.087144 | 27 |
| TSK | 3 | 20 | 0 | 0.00554 | 0.00554 | 0 | 0.037631 | 216 |
| Spline | 3 | 20 | 0 | 1.176986 | 1.176986 | 0 | 0.02081 | 512 |
| NNSA | 3 | 20 | 0 | 0.196855 | 0.196855 | 0 | 0.02419 | 512 |
| MLP | 3 | 30 | 11.82266 | 0.018017 | 11.84067 | 0.202482 | 0.151629 | 25 |
| FCC | 3 | 30 | 10.53178 | 0.00722 | 10.539 | 0.132451 | 0.062261 | 99 |
| SVM | 3 | 30 | 4.939962 | 0.005136 | 4.945098 | 0.075979 | 0.022402 | 330 |
| ELM | 3 | 30 | 1.277036 | 0.011005 | 1.288041 | 0.193019 | 0.185437 | 60 |
| ANFIS | 3 | 30 | 0.647471 | 0.002924 | 0.650395 | 0.11093 | 0.091414 | 27 |
| TSK | 3 | 30 | 0 | 0.005423 | 0.005423 | 0 | 0.079976 | 216 |
| Spline | 3 | 30 | 0 | 1.180454 | 1.180454 | 0 | 0.032455 | 512 |
| NNSA | 3 | 30 | 0 | 0.197245 | 0.197245 | 0 | 0.022763 | 512 |
| MLP | 3 | 50 | 13.9658 | 0.018649 | 13.98444 | 0.218229 | 0.099286 | 25 |
| FCC | 3 | 50 | 20.35409 | 0.017786 | 20.37188 | 0.139455 | 0.070053 | 85 |
| SVM | 3 | 50 | 40.31731 | 0.016089 | 40.3334 | 0.056144 | 0.036059 | 500 |
| ELM | 3 | 50 | 2.476887 | 0.025589 | 2.502475 | 0.178807 | 0.071072 | 50 |
| ANFIS | 3 | 50 | 1.233366 | 0.00428 | 1.237646 | 0.079535 | 0.072193 | 27 |
| TSK | 3 | 50 | 0 | 0.056827 | 0.056827 | 0 | 0.034738 | 512 |
| Spline | 3 | 50 | 0 | 5.667674 | 5.667674 | 0 | 0.025043 | 1000 |
| NNSA | 3 | 50 | 0 | 0.834155 | 0.834155 | 0 | 0.022016 | 1000 |
| MLP | 3 | 100 | 17.28814 | 0.020393 | 17.30853 | 0.220541 | 0.146163 | 60 |
| FCC | 3 | 100 | 35.29679 | 0.037954 | 35.33474 | 0.179835 | 0.129798 | 85 |
| SVM | 3 | 100 | 187.8772 | 0.05173 | 187.9289 | 0.047501 | 0.030291 | 856 |
| ELM | 3 | 100 | 4.237248 | 0.064341 | 4.301589 | 0.199897 | 0.154177 | 60 |
| ANFIS | 3 | 100 | 2.114776 | 0.0065 | 2.121276 | 0.125058 | 0.113305 | 27 |
| TSK | 3 | 100 | 0 | 0.094961 | 0.094961 | 0 | 0.046213 | 1000 |
| Spline | 3 | 100 | 0 | 18.53888 | 18.53888 | 0 | 0.031442 | 1728 |
| NNSA | 3 | 100 | 0 | 2.701356 | 2.701356 | 0 | 0.028755 | 1728 |
| MLP | 4 | 10 | 28.35486 | 0.022919 | 28.37778 | 0.231954 | 0.135059 | 25 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| FCC | 4 | 10 | 85.23116 | 0.031815 | 85.26298 | 0.152566 | 0.063653 | 95 |
| SVM | 4 | 10 | 723.1443 | 0.2785 | 723.4228 | 0.175609 | 0.025627 | 2382 |
| ELM | 4 | 10 | 9.943142 | 0.055251 | 9.998393 | 0.211735 | 0.135045 | 60 |
| ANFIS | 4 | 10 | 66.04318 | 0.023018 | 66.0662 | 0.093438 | 0.088782 | 81 |
| TSK | 4 | 10 | 0 | 0.184057 | 0.184057 | 0 | 0.048957 | 1296 |
| Spline | 4 | 10 | 0 | 46.03165 | 46.03165 | 0 | 0.041206 | 4096 |
| NNSA | 4 | 10 | 0 | 6.033782 | 6.033782 | 0 | 0.022663 | 4096 |
| MLP | 4 | 20 | 28.31941 | 0.019033 | 28.33844 | 0.197213 | 0.113244 | 10 |
| FCC | 4 | 20 | 85.06787 | 0.034637 | 85.1025 | 0.11942 | 0.060266 | 110 |
| SVM | 4 | 20 | 402.6522 | 0.152874 | 402.805 | 0.042911 | 0.026338 | 1313 |
| ELM | 4 | 20 | 10.00417 | 0.009647 | 10.01382 | 0.218395 | 0.105801 | 10 |
| ANFIS | 4 | 20 | 68.40435 | 0.02277 | 68.42712 | 0.077212 | 0.081042 | 81 |
| TSK | 4 | 20 | 0 | 0.131341 | 0.131341 | 0 | 0.034564 | 1296 |
| Spline | 4 | 20 | 0 | 45.9078 | 45.9078 | 0 | 0.020877 | 4096 |
| NNSA | 4 | 20 | 0 | 6.032537 | 6.032537 | 0 | 0.023059 | 4096 |
| MLP | 4 | 30 | 28.29759 | 0.020012 | 28.3176 | 0.223361 | 0.190555 | 25 |
| FCC | 4 | 30 | 85.50003 | 0.030924 | 85.53096 | 0.181146 | 0.125596 | 95 |
| SVM | 4 | 30 | 352.0603 | 0.210318 | 352.2706 | 0.058419 | 0.033476 | 1814 |
| ELM | 4 | 30 | 9.949425 | 0.054983 | 10.00441 | 0.222647 | 0.177087 | 60 |
| ANFIS | 4 | 30 | 69.23301 | 0.022929 | 69.25594 | 0.116801 | 0.093651 | 81 |
| TSK | 4 | 30 | 0 | 0.131906 | 0.131906 | 0 | 0.071039 | 1296 |
| Spline | 4 | 30 | 0 | 46.05314 | 46.05314 | 0 | 0.029394 | 4096 |
| NNSA | 4 | 30 | 0 | 6.042726 | 6.042726 | 0 | 0.020083 | 4096 |
| MLP | 4 | 50 | 55.73015 | 0.022774 | 55.75293 | 0.220229 | 0.118194 | 15 |
| FCC | 4 | 50 | 208.4648 | 0.122344 | 208.5871 | 0.166434 | 0.077965 | 95 |
| SVM | 4 | 50 | 5389.336 | 1.107214 | 5390.443 | 0.039466 | 0.03439 | 3612 |
| ELM | 4 | 50 | 23.78374 | 0.211077 | 23.99482 | 0.215717 | 0.126312 | 60 |
| ANFIS | 4 | 50 | 161.5137 | 0.056375 | 161.5701 | 0.101769 | 0.07598 | 81 |
| TSK | 4 | 50 | 0 | 1.108912 | 1.108912 | 0 | 0.033772 | 4096 |
| Spline | 4 | 50 | 0 | 380.1863 | 380.1863 | 0 | 0.025567 | 10000 |
| NNSA | 4 | 50 | 0 | 49.33926 | 49.33926 | 0 | 0.02203 | 10000 |
| MLP | 4 | 100 | 105.3967 | 0.033225 | 105.4299 | 0.264329 | 0.18534 | 25 |
| FCC | 4 | 100 | 430.3563 | 0.357924 | 430.7142 | 0.238833 | 0.155644 | 110 |
| SVM | 4 | 100 | 47576 | 4.379837 | 47580.38 | 0.043271 | 0.029839 | 6504 |
| ELM | 4 | 100 | 48.59702 | 0.563259 | 49.16028 | 0.24534 | 0.173948 | 60 |
| ANFIS | 4 | 100 | 331.3938 | 0.122643 | 331.5165 | 0.149129 | 0.124757 | 81 |
| TSK | 4 | 100 | 0 | 8.733116 | 8.733116 | 0 | 0.046541 | 10000 |
| Spline | 4 | 100 | 0 | 1940.6 | 1940.6 | 0 | 0.03247 | 20736 |
| NNSA | 4 | 100 | 0 | 267.5006 | 267.5006 | 0 | 0.029341 | 20736 |
| MLP | 5 | 10 | 164.608 | 0.040163 | 164.6481 | 0.256136 | 0.180209 | 25 |
| FCC | 5 | 10 | 701.3773 | 0.159256 | 701.5365 | 0.166944 | 0.073857 | 105 |
| SVM | 5 | 10 | 93743.83 | 20.17087 | 93764 | 0.158525 | 0.032145 | 20242 |
| ELM | 5 | 10 | 98.60218 | 0.148691 | 98.75087 | 0.312352 | 0.262447 | 25 |

| ANFIS | 5 | 10 | 13872.62 | 0.451701 | 13873.07 | 0.109332 | 0.092275 | 243 |
|---|---|---|---|---|---|---|---|---|
| TSK | 5 | 10 | 0 | 9.405557 | 9.405557 | 0 | 0.047568 | 7776 |
| Spline | 5 | 10 | 0 | 1886.991 | 1886.991 | 0 | 0.041853 | 32768 |
| NNSA | 5 | 10 | 0 | 227.5315 | 227.5315 | 0 | 0.023028 | 32768 |
| MLP | 5 | 20 | 170.8636 | 0.031217 | 170.8948 | 0.181 | 0.104928 | 10 |
| FCC | 5 | 20 | 710.2176 | 0.176799 | 710.3944 | 0.139779 | 0.103971 | 121 |
| SVM | 5 | 20 | 76271.13 | 8.009599 | 76279.14 | 0.039603 | 0.027568 | 7866 |
| ELM | 5 | 20 | 77.05893 | 0.048053 | 77.10698 | 0.195917 | 0.095323 | 10 |
| ANFIS | 5 | 20 | 12863.51 | 0.429435 | 12863.94 | 0.079761 | 0.076789 | 243 |
| TSK | 5 | 20 | 0 | 9.362086 | 9.362086 | 0 | 0.032082 | 7776 |
| Spline | 5 | 20 | 0 | 1890.666 | 1890.666 | 0 | 0.021256 | 32768 |
| NNSA | 5 | 20 | 0 | 220.0561 | 220.0561 | 0 | 0.022327 | 32768 |
| MLP | 5 | 30 | 164.9082 | 0.033286 | 164.9415 | 0.228211 | 0.188709 | 15 |
| FCC | 5 | 30 | 696.1101 | 0.171432 | 696.2815 | 0.166718 | 0.127076 | 121 |
| SVM | 5 | 30 | 57440.51 | 7.775694 | 57448.28 | 0.03498 | 0.03898 | 8339 |
| ELM | 5 | 30 | 76.29271 | 0.28198 | 76.57469 | 0.241528 | 0.224315 | 60 |
| ANFIS | 5 | 30 | 12856.47 | 0.444522 | 12856.92 | 0.117707 | 0.084495 | 243 |
| TSK | 5 | 30 | 0 | 9.419723 | 9.419723 | 0 | 0.064533 | 7776 |
| Spline | 5 | 30 | 0 | 1887.493 | 1887.493 | 0 | 0.027043 | 32768 |
| NNSA | 5 | 30 | 0 | 220.4 | 220.4 | 0 | 0.018107 | 32768 |

## 6.2 MATLAB Code

### 6.2.1 Algorithm Functions

Name: MLPResults.m

```
function [output,RMSETR,RMSETS,trainTime,testTime,network,nodes]=MLPResults(TrainData,TestData,sizes,ntrial,epoch_n)


RMSETS=100;
RMSETR=100;
ns=length(sizes);
trainTimes=zeros(1,ns);
testTimes=zeros(1,ns);
tt=1;
for j=1:ns
nsize=sizes(j);
for i=1:ntrial
tic;
inputs = TrainData(:,1:end-1).';
targets = TrainData(:,end)';

% Create a Fitting Network
hiddenLayerSize = nsize;
trainFcn='trainrp';
net = fitnet(hiddenLayerSize,trainFcn);
net.trainParam.showWindow = false;

% Setup Division of Data for Training, Validation, Testing
net.divideParam.trainRatio = 100/100;
net.divideParam.valRatio = 0/100;
net.divideParam.testRatio = 0/100;

net.trainParam.epochs = epoch_n;

% Train the Network

[net,tr] = train(net,inputs,targets);

trainTimes(tt)=toc;
```

```
tic;

OO=net(TestData(:,1:end-1).');


% Test the Network
outputs = net(inputs);
testTimes(tt)=toc;
% times(i)=toc;



[~,RMSETR1]=computeRMSE(targets,outputs);
[~,RMSETS1]=computeRMSE(TestData(:,end),OO.');
if (RMSETS1<RMSETS)
    RMSETS=RMSETS1;


network=net;

    testTime=testTimes(tt);
    nodes=nsize;
    RMSETR=RMSETR1;
    output=OO.';
end
tt=tt+1;
end
end
% time=sum(times)/length(times);
nntraintool('close')

% time=toc;
trainTime=sum(trainTimes);
```

# Name: FCCResults.m

```
function
[output,RMSETR,RMSETS,trainTime,testTime,nodes,topo,best_w,act,gain,paramt,iw]=FCCResults(TrainData,TestData,nsize,ntrial,maxite,
ntest)

trainTime = nan;
testTime = nan;
topo = nan;
best_w = nan;
act = nan;
gain = nan;
paramt = nan;
iw = nan;
nodes = nan;

types={'FCC'};
ntypes=length(types);
data = [TrainData ;TestData];
[m,n]=size(data); ninp=n-1;

maxerr=1e-3;

RMSETS=100*ones(1,ntypes);
RMSETR=100*ones(1,ntypes);
output=zeros(length(TestData(:,end)),ntypes);
trainTimeTotal = 0;
for k=1:length(types)
    type=types{k};
    for i = 1:length(nsize)
        h=nsize(i);
        network=[ninp ones(1,h) 1]; nFig=11+h;
        [RMSETS1,RMSETR1,output1,trainTime1,testTime1,topo1,best_w1,act1,gain1,paramt1,iw1]=nbn(data, type, network, ntrial,
maxite, maxerr,nFig,ntest);
        trainTimeTotal = trainTimeTotal + trainTime1;
        if (RMSETS1 < RMSETS)
            output = output1;
            RMSETR = RMSETR1;
            RMSETS = RMSETS1;
            testTime = testTime1;
            topo = topo1;
            best_w = best_w1;
            act = act1;
            gain = gain1;
            paramt = paramt1;
            iw = iw1;
            nodes = nsize(i);
        end


    end
end
trainTime = trainTimeTotal;

function [bestTSRMSE,bestTRRMSE,out,trainTime,testTime,topo,best_w,act,gain,paramt,iw]=nbn(data, type, network, ntrial, maxite,
maxerr,nFig,ntest)
format compact;  warning off; %#ok<WNOFF>
```

```
topo=gen_topo(type,network);
[m,n]=size(data);
Ti = data(:,1:n-1);
Td = data(:,n);

ind = 1:1:m;
Tnp = m-ntest;
Ti_tst = Ti(ind(Tnp+1:end),:);  Ti = Ti(ind(1:Tnp),:);
Td_tst = Td(ind(Tnp+1:end));    Td = Td(ind(1:Tnp));

Tnp=size(Ti,1);   Tnp_tst = m-Tnp;
nd=size(Ti,2);

%% neural network NBN      FCC
%%%%%%%%%%%%%%%%%%%%%% set train parameter %%%%%%%%%%%%%%%%%%%%%%%
% maxite = 10;          % max iteration
mu = 0.01;             % mu
muH = 1e15;            % high bound of mu
muL = 1e-15;           % low bound of mu
scale = 10;            % scale
% maxerr = 1e-3;         % max required error
setting = [maxite,mu,muH,muL,scale,maxerr];
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[np,ni,no,nw,nn,iw] = checkingInputs(Ti,Td,topo);
act=5*ones(1,nn); act(nn)=0;
gain=1*ones(1,nn);
param=[np,ni,no,nw,nn];

threshold=0.09;   % threshold for success rate evaluation
RMSE_rc=zeros(ntrial,1);  time_rc=zeros(ntrial,1);   is=0;   RMSEt_rc=zeros(ntrial,1);  ttime_rc=zeros(ntrial,1);

bestTSRMSE=inf;
best_w=generate_weights(nw);


for tr=1:ntrial
    w_ini=generate_weights(nw);
    tic;
    [w,iter,SSE] = Trainer(Ti,Td,topo,w_ini,act,gain,param,iw,setting);
    t=toc;

    paramt=param;  paramt(1)=Tnp_tst;
    tic;
    SSEt=calculate_error(Ti_tst,Td_tst,topo,w,act,gain,paramt,iw);  % scalar
    tt=toc;
    outtr=calc_fwd(Ti,topo,w,act,gain,paramt,iw);
    outts=calc_fwd(Ti_tst,topo,w,act,gain,paramt,iw);
    [~,RMSE]=computeRMSE(Td,outtr);
    [~,RMSEt_rc(tr)]=computeRMSE(Td_tst,outts);
    if (RMSEt_rc(tr) < bestTSRMSE)
        bestTSRMSE = RMSEt_rc(tr);
        bestTRRMSE=RMSE;
        best_w=w;
    end
    if RMSE<threshold
        is=is+1;
    end;




    RMSE_rc(tr)=RMSE;
    time_rc(tr)=t;
    ttime_rc(tr)=tt;
end;
sr=is/ntrial;                                   % success rate
time_ave=sum(time_rc)/ntrial;                   % average training time
ttime_ave=sum(ttime_rc)/ntrial;                 % average testing time
RMSE_ave=sum(RMSE_rc)/ntrial;                     % average training RMSE
RMSEt_ave=sum(RMSEt_rc)/ntrial;                   % average testing RMSE
RMSE_std=std(RMSE_rc);                           % std of all the training RMSE
RMSEt_std=std(RMSEt_rc);                         % std of all the testing RMSE
trainTime=time_ave;
testTime=ttime_ave;

tic;
out=calc_fwd(Ti_tst,topo,best_w,act,gain,paramt,iw);
testTime=toc;

function [ww,iter,SSE] = Trainer(inp,dout,topo,w,act,gain,param,iw,setting)
ww = w;                    % weight
nw = param(4);             % number of weights
maxite = setting(1);       % max iteration
mu = setting(2);           % mu
muH = setting(3);          % high bound of mu
muL = setting(4);          % low bound of mu
scale = setting(5);        % scale
maxerr = setting(6);       % max requred error

TER = calculate_error(inp,dout,topo,ww,act,gain,param,iw);
SSE=zeros(maxite,1);
SSE(1) = TER;
I = eye(nw);
for iter = 2:maxite
```

```
    jw = 0;
    [gradient,hessian] = Hessian(inp,dout,topo,ww,act,gain,param,iw);
    ww_backup = ww;
    while 1
        ww = ww_backup - ((hessian+mu*I)\gradient)';
        TER = calculate_error(inp,dout,topo,ww,act,gain,param,iw);
        SSE(iter) = TER;
        if TER <= SSE(iter-1)
            if mu > muL
                mu = mu/scale;
            end;
            break;
        end;
        if mu < muH
            mu = mu*scale;
        end;
        jw = jw + 1;
        if jw > 30
            break;
        end;
    end;
    if SSE(iter) < maxerr
        break;
    end;
    if (SSE(iter-1)-SSE(iter))/SSE(iter-1)<0.000000000000001
        break;
    end;
end;
return;

function [gradient,hessian] = Hessian(inp,dout,topo,ww,act,gain,param,iw)
% param(1)-------np--------number of pattern
% param(2)-------ni--------number of input
% param(3)-------no--------number of output
% param(4)-------nw--------number of weights
% param(5)-------nn--------number of neurons
np=param(1);
ni=param(2);
no=param(3);
nw=param(4);
nn=param(5);

gradient = zeros(nw,1);
hessian = zeros(nw,nw);

for p = 1:np
    node(1:ni) = inp(p, 1:ni);%pobierz wiersz

    for n = 1:nn
        j = ni + n;
        net = ww(iw(n));
        for i = (iw(n)+1):(iw(n+1)-1)
            net = net + node(topo(i))*ww(i);
        end;
        [out,de(j)]=actFuncDer(n,net,act,gain);
        node(j) = out;
    end;

    for k = 1:no           % for each output
        error = dout(p,k) - node(nn+ni-no+k);
        J = zeros(1, nw);  % Jacobian row
        o = nn + ni - no + k;
        s = iw(o-ni);
        J(s) = -de(o);   %%% modify de depending on sign of error and net
        delo=zeros(1,nn+ni-no+1);

        for i = (s+1):(iw(o+1-ni)-1)
            J(i) = node(topo(i))*J(s);
            delo(topo(i)) = delo(topo(i))-ww(i)*J(s);
        end;

        for n = 1:(nn-no)                 %hidden neurons in the reverse order
            j = nn+ni-no + 1 - n;   %node number
            s = iw(j-ni);
            J(s) = -de(j)*delo(j);                     %for bias of hidden neurons
            for i = (s+1):(iw(j-ni+1)-1)        %for weights of hidden neurons
                J(i) = node(topo(i))*J(s);
                delo(topo(i)) = delo(topo(i)) - ww(i)*J(s);
            end;
        end;

        gradient = gradient + J'*error;
        hessian = hessian + J'*J;

    end;
end;
return;

function topo=gen_topo(type,network)
%  MLP ,network=>  ninp 3 4 2 1
%  SLP ,network=>  ninp 17 1
%  FCC ,network=>  ninp 1 1 1 1 1 1
%  MLP ,network=>  ninp 3 4 2 1
topo=[];
nl=length(network);
```

```
for i=2:nl                   %   for number of layers
    s=sum(network(1:i-1));       %   starting a new layer
    for j=1:network(i)           %   in each layer
        switch type
            case 'SLP'
                topo=[topo, s+j, s-network(i-1)+1:s];
            case 'MLP'
                topo=[topo, s+j, s-network(i-1)+1:s];
            case 'FCC'
                topo=[topo, s+j, 1:s];   %s+j node number and j is always 1
            case 'BMLP'
                topo=[topo, s+j, 1:s];   %s+j node number
        end;
    end;
end;
return;

function [y] = calc_fwd(inp,topo,w,act,gain,param,iw)
np = size(inp,1);            % number of pattern
ni = param(2);            % number of input
no = param(3);            % number of output
nn = param(5);            % number of neurons
y = zeros(np,no);
for p = 1:np      % number of patterns
    node(1:ni) = inp(p,1:ni);
    for n = 1:nn % number of neurons
        j = ni + n;
        net = w(iw(n));
        for i = (iw(n)+1):(iw(n+1)-1)
            net = net + node(topo(i))*w(i);
        end;
        out=actFunc(n,net,act,gain);
        node(j) = out;
    end;
    y(p,:)=node(ni+nn-no+1:ni+nn);
end;

function [err] = calculate_error(inp,dout,topo,w,act,gain,param,iw)
np = param(1);            % number of pattern
ni = param(2);            % number of input
no = param(3);            % number of output
nn = param(5);            % number of neurons
err = 0;
for p = 1:np      % number of patterns
    node(1:ni) = inp(p,1:ni);
    for n = 1:nn % number of neurons
        j = ni + n;
        net = w(iw(n));
        for i = (iw(n)+1):(iw(n+1)-1)
            net = net + node(topo(i))*w(i);
        end;
        out=actFunc(n,net,act,gain);
        node(j) = out;
    end;
    for k = 1:no
        err = err + (dout(p,k)-node(nn+ni-no+k))^2;                % calculate total error
    end;
end;

function [np,ni,no,nw,nn,iw]= checkingInputs(inp,dout,topo)
iw = findiw(topo);
[np,ni]=size(inp);
[y,no]=size(dout);
if (np ~= y) error('input and output patterns are not equal'); end;
nw=length(topo);
y=length(topo); nn=length(iw)-1;

if (min(min(sign(topo))')<1)
    error('all elements of topo must be positive');
end;

if (nw==0)
    error('weights must not be zero');
end;
return;

function iw = findiw(topo)
nmax=0; j=0;
for i=1:length(topo),
    if topo(i)>nmax,
        nmax=topo(i);
        j=j+1; iw(j)=i;
    end;
end;
iw(j+1)=i+1;
return

function [weight] = generate_weights(nw)
for i = 1:nw             % number of weights
    ra = 2*rand(1)-1;          % generate random weights between -1 and 1
    while( ra == 0 )
        ra = 2*rand(1)-1;
    end;
    weight(i) = ra;
end;
```

```
function out=actFunc(n,net,act,gain)
de=0;
switch act(n)
    case 0, out = gain(n)*net;                          % linear neuron
    case 1, out = 1/(1+exp(-gain(n)*net));              % unipolar neuron
    case 2, out = tanh(gain(n)*net);                    % bipolar neuron
    case 3, out = gain(n)*net/(1+gain(n)*abs(net));     % bipolar elliot neuron
    case 4, out = 2*gain(n)*net/(1+gain(n)*abs(net))-1; % unipolar elliot neuron
    case 5, out = 2/(1+exp(-gain(n)*net))-1;            % bipolar from NBN 2.08
    case 6,
%          out = sign(gain(n)*net);
        if (abs(gain(n)*net)>=1)
            out = sign(gain(n)*net);            % hard activation
        else
            out = gain(n)*net;
        end
end;

function [out,der]=actFuncDer(n,net,act,gain)
de=0;
switch act(n)
    case 0, out = gain(n)*net;  der = gain(n);                                          % linear neuron
    case 1, out = 1/(1+exp(-gain(n)*net)); der = gain(n)*(1-out)*out;                   % unipolar neuron   % log-
likelyhood cost function: der = gain(n)/(1-out)/out;
    case 2, out = tanh(gain(n)*net);  der = gain(n)*(1-out*out);                        % bipolar neuron
    case 3, out = gain(n)*net/(1+gain(n)*abs(net));der = 1/((gain(n)*abs(net)+1)^2);    % bipolar elliot neuron
    case 4, out = 2*gain(n)*net/(1+gain(n)*abs(net))-1; der = 2*gain(n)/(gain(n)*abs(net)+1)^2;      % unipolar elliot neuron
    case 5, out = 2/(1+exp(-gain(n)*net))-1; der = gain(n)*(1-out*out)/2;               % bipolar from NBN 2.08
    case 6,
        if (abs(gain(n)*net)>=1)
            out = sign(gain(n)*net);            % hard activation
            der = 0;
        else
            out = gain(n)*net;der = gain(n);
        end

end;
der=der+de;
```

## Name: ELMResults.m

```
function [output,RMSETR,RMSETS,trainTime,testTime,inw1,outw1,bias1,nodes1]=ELMResults(TrainData,TestData,nodeV,ntrial)

 wRange=[-1 1]; bRange=[-1 1];

trainTimes=zeros(1,ntrial*length(nodeV));
testTimes=zeros(1,ntrial*length(nodeV));

RMSETS=100;
RMSETR=100;

tt=1;
for na=1:length(nodeV)
    nodes=nodeV(na);
for i=1:ntrial
tic;
    [inw, outw, bias, outputs, error]=ELM(TrainData(:,1:end-1),TrainData(:,end),wRange,bRange,nodes);
    trainTimes(tt)=toc;
    tic;

    [~,O]=calcO(TestData(:,1:end-1),TestData(:,end),inw,outw,bias,nodes);
    testTimes(tt)=toc;

    [~,RMSETR1]=computeRMSE(TrainData(:,end),outputs.');
    [~,RMSETS1]=computeRMSE(TestData(:,end),O.');

if (RMSETS1<RMSETS)
    RMSETS=RMSETS1;
    RMSETR=RMSETR1;
    inw1=inw;
    outw1=outw;
    bias1=bias;
    nodes1=nodes;
    testTime=testTimes(tt);

    output=O.';
end
  tt=tt+1;

end


end

trainTime=sum(trainTimes);

%% Original ELM (not incremental ELM)
% Inputs *******************
% x are the training vectors
% y are the targets
```

```
% wRange is a 1x2 matrix containing the lower and upper bounds for the
% range of the input weights
% bRange is the same as wRange but pertaining to the input bias
% nodes is the number of nodes in the network
% output = sum of outw*g(inw*x+bias)
function [inw outw bias outputs error]=ELM(x,y,wRange,bRange,nodes)
[np,nd]=size(x);
inw=(wRange(2)-wRange(1))*rand(nodes,nd)+wRange(1);
bias=(bRange(2)-bRange(1))*rand(nodes,1)+bRange(1);
for i=1:nodes
    for j=1:np
        H(j,i)=1/(1+exp(-(inw(i,:)*x(j,:)'+bias(i))));
    end
end
%Calculate Moore-Penrose generalized inverse of H
Ht=pinv(H);
%Calculate output weights
outw=Ht*y;
outputs=outw'*H';
error=y-outputs';

function [SSE,O]=calcO(x, y, inw, outw, bias, nodes)
[np,nd]=size(x);
for i=1:nodes
    for j=1:np
        H(j,i)=1/(1+exp(-(inw(i,:)*x(j,:)'+bias(i))));
    end
end
O=outw'*H';
er=y-O'; SSE=er'*er;
```

## Name: SVRResults.m

```
function
[output,RMSETR,RMSETS,trainTime,testTime,model_bst,nodes,C_bst,gamma_bst]=SVRResults(TrainData,TestData,gamma_list,C_list)
%% instruction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% train %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Usage: model = svmtrain(training_label_vector, training_instance_matrix, 'libsvm_options');
% libsvm_options:
% -s svm_type : set type of SVM (default 0)
%   0 -- C-SVC
%   1 -- nu-SVC
%   2 -- one-class SVM
%   3 -- epsilon-SVR
%   4 -- nu-SVR
% -t kernel_type : set type of kernel function (default 2)
%   0 -- linear: u'*v
%   1 -- polynomial: (gamma*u'*v + coef0)^degree
%   2 -- radial basis function: exp(-gamma*|u-v|^2)
%   3 -- sigmoid: tanh(gamma*u'*v + coef0)
%   4 -- precomputed kernel (kernel values in training_instance_matrix)
% -d degree : set degree in kernel function (default 3)
% -g gamma : set gamma in kernel function (default 1/num_features)
% -r coef0 : set coef0 in kernel function (default 0)
% -c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default 1)
% -n nu : set the parameter nu of nu-SVC, one-class SVM, and nu-SVR (default 0.5)
% -p epsilon : set the epsilon in loss function of epsilon-SVR (default 0.1)
% -m cachesize : set cache memory size in MB (default 100)
% -e epsilon : set tolerance of termination criterion (default 0.001)
% -h shrinking : whether to use the shrinking heuristics, 0 or 1 (default 1)
% -b probability_estimates : whether to train a SVC or SVR model for probability estimates, 0 or 1 (default 0)
% -wi weight : set the parameter C of class i to weight*C, for C-SVC (default 1)
% -v n : n-fold cross validation mode
% -q : quiet mode (no outputs)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% predict %%%%%%%%%%%%%%%%%%%%%%%%%%%
% Usage: [predicted_label, accuracy, decision_values/prob_estimates] = svmpredict(testing_label_vector, testing_instance_matrix,
model, 'libsvm_options')
% Parameters:
%   model: SVM model structure from svmtrain.
%   libsvm_options:
%     -b probability_estimates: whether to predict probability estimates, 0 or 1 (default 0); one-class SVM not supported yet
% Returns:
%   predicted_label: SVM prediction output vector.
%   accuracy: a vector with accuracy, mean squared error, squared correlation coefficient.
%   prob_estimates: If selected, probability estimate vector.
%% support vector regression train
ng = length(gamma_list);
nc = length(C_list);
TrainData(:,end)=2*TrainData(:,end)-1;TestData(:,end)=2*TestData(:,end)-1;
Td=TrainData(:,end); Ti=TrainData(:,1:end-1);
rms_bst = 10;
figcnt=1;
trainTimes=zeros(1,ng*nc);
testTimes=zeros(1,ng*nc);
for i = 1:ng
    for j = 1:nc
        gamma = gamma_list(i);   C = C_list(j);
        option = sprintf('-s 3 -t 2 -h 0 -q -g %f -c %f', gamma, C);
        tic;

        model = svmtrain(Td, Ti, option);
```

```
        trainTimes(figcnt)=toc;
        tic;
        [y, ~, ~] = svmpredict(Td, Ti, model);
        [yt, ~, ~] = svmpredict( TestData(:,end),TestData(:,1:end-1), model);
        testTimes(figcnt)=toc;
        [~,trainRMS]=computeRMSE((TrainData(:,end)+1)/2,(y+1)/2);
        [~,testRMS]=computeRMSE((TestData(:,end)+1)/2,(yt+1)/2);
      if testRMS<rms_bst
            rms_bst = testRMS;
            model_bst = model;
            C_bst = C;
            gamma_bst = gamma;
            RMSETR=trainRMS;
            RMSETS=testRMS;
            testTime=testTimes(figcnt);
            output=yt;
      end;
      figcnt=figcnt+1;
    end;
end;

trainTime=sum(trainTimes);
output=(output+1)/2;

nodes=model_bst.totalSV;
```

## Name: fuzzyResults.m

```
function
[output,RMSETR,RMSETS,trainTime,testTime,out_fis,nodes]=fuzzyResults(TrainData,TestData,numMFs,inmftype,outmftype,epoch_n)

nodes=prod(numMFs);
tic;
in_fis = genfis1(TrainData,numMFs,inmftype,outmftype);
trnOpt=[epoch_n,0,0.1,0.9,1.1];
dispOpt=[0,0,0,0];

out_fis = anfis(TrainData,in_fis,trnOpt,dispOpt);
trainTime=toc;
tic;
output=evalfis(TestData(:,1:end-1),out_fis);
TrainOut=evalfis(TrainData(:,1:end-1),out_fis);
testTime=toc;
[~,RMSETR]=computeRMSE(TrainData(:,end),TrainOut);

[~,RMSETS]=computeRMSE(TestData(:,end),output);
```

## Name: FuzzyTSKResults.m

```
function [output,RMSETR,RMSETS,trainTime,testTime,nodes]=FuzzyTSKResults(xt,TrainData,TestData,trainn)

ndim = size(TrainData,2)-1;
trainTime=0;
testTime=0;


numMFs = zeros(1,ndim);
mfType=char(zeros(ndim,5));
for i=1:ndim
    numMFs(i) = length(xt{i});
%     mfType = [mfType;'trimf'];
    mfType(i,:) = 'trimf';
end

nodes=prod(numMFs);

% grid=reshape(TrainData(:,end),numMFs).'; % For 2D
grid=reshape(TrainData(:,end),numMFs);


outmftype='constant';
fismat = genfis1(TrainData,numMFs,mfType,outmftype);
for i=1:nodes
fismat.output.mf(i).params=grid(i);
end
tic;
output=evalfis(TestData(:,1:end-1),fismat);
testTime=testTime+toc;

RMSETR=0;
[~,RMSETS]=computeRMSE(TestData(:,end),output);
```

## Name: splineResults.m

```
function [output,RMSETR,RMSETS,trainTime,testTime,nodes]=splineResults(TrainData,TestData,trainn,xtE,nsegE,xv)
trainTime=0;
testTime=0;
nodes=prod(nsegE);
% xv{2}=xv{2}.';
output=zeros(size(TestData(:,end)));
tic;
for i=1:size(TestData,1)

point = num2cell(TestData(i,1:end-1));

output(i) = SplineND(xtE, TrainData, point,nsegE);
end
testTime=testTime+toc;

RMSETR = 0;
[~,RMSETS]=computeRMSE(TestData(:,end),output);

function splineValues = SplineND(ranges, values, points,nsegE)


dimensions=size(values,2)-1;



splineStorageSize=num2cell(nsegE);
splineStorageSize{end}=length(points{1});
splineStorage=reshape(values(:,end),nsegE);
for i=1:dimensions
    resolution=length(points{i});
    index1=1:nsegE(i);
    index2=1:resolution;
    [splineIndex,dataIndex]=generateIndices(dimensions,i,index1,index2);
    splineStorageSize{end-i+1}=resolution;

    tempMatrix=zeros(splineStorageSize{:});
    for j=1:length(dataIndex(:,1))
        currentDataIndex=dataIndex(j,:);
        currentSplineIndex=splineIndex(j,:);

        tempMatrix(currentSplineIndex{:})=SPLINE1D(ranges{i},splineStorage(currentDataIndex{:}),points{i});
    end
    splineStorage=tempMatrix;
end
splineValues=splineStorage;



function [splineIndex,dataIndex]=generateIndices(dimensions,i,index1,index2)


vectorsToGrid=cell(1,dimensions-1);
te=i-1;

vectorsToGrid(1:end-te)={index1};
vectorsToGrid(end-te+1:end)={index2};


    temp1=cell(1,dimensions-1);
    [temp1{:}]=ndgrid(vectorsToGrid{:});

    baseIndex=cell(numel(temp1{1}),dimensions-1);
    for j=1:dimensions-1


        baseIndex(:,j)=num2cell(temp1{:,j}(:));
    end
splineIndex=cell(size(baseIndex,1),dimensions);

splineIndex(:,1:end~=(end-i+1))=baseIndex;
dataIndex=splineIndex;

dataIndex(:,end-i+1)={index1};
splineIndex(:,end-i+1)={index2};

function output = SPLINE1D(x, y, interpPoints)

x=x(:).';
y=y(:).';
% % Mirror edge values to give better fit at boundry
x=[x(1)-(x(2)-x(1)),x,x(end)+(x(end)-x(end-1))];
y=[y(1),y,y(end)];


n=length(x);


h=diff(x);

A=zeros(n,n);
f=zeros(n,1);
```

```
% Setting up matrix for g''(x)
for i=1:n
    % The first and last segments require special conditions
    if ( i == 1 )
        A(i,i)=-h(i+1);
        A(i,i+1)=h(i)+h(i+1);
        A(i,i+2)=-h(i);

    elseif( i == n)
        A(i,i-2)=-h(i-1);
        A(i,i-1)=h(i-2)+h(i-1);
        A(i,i)=-h(i-2);
    else
        A(i,i-1)=h(i-1);
        A(i,i)=2*(h(i-1)+h(i));
        A(i,i+1)=h(i);
        f(i)=6*((y(i+1)-y(i))/h(i)-(y(i)-y(i-1))/h(i-1));
    end
end

m=A\f;


% With the g''(x) for each interval, solve for the coeficients
coefs=zeros(n-1,4);

for i=1:n-1
    coefs(i,1)=y(i);
    coefs(i,2)=(y(i+1)-y(i))/h(i)-h(i)/2*m(i)-h(i)/6*(m(i+1)-m(i));
    coefs(i,3)=m(i)/2;
    coefs(i,4)=(m(i+1)-m(i))/(6*h(i));
end

output = EvaluatePoints(coefs, x, interpPoints);


function value = EvaluatePoints(coefs, range, points)
% EvaluatePoints  Computes polynomial values
% -----------------------------------------------------------------------
%
% Parameters:
%
%   value = EvaluatePoints(coefs, points)
%
%    INPUT:
%
%      coefs - (n-1)x4 matrix of coefeicient values for each segment.
%
%
%      ranges - vector of length n conataining the break values of the piecewise polynomial
%
%
%      points - vector of the desired points at which the piecewise polynomial should
%      be evaluated.  All of the values should be contained with
%      [range(1),range(end)]
%
%    OUTPUT:
%
%      value - vector of the same length as input points containing the
%      function values corresponding to each point

% dimensions=size(points,2);
n=length(range);
value=zeros(size(points));
for i=1:n-1
    activePoints=points(points>=range(i) & points<=range(i+1))-range(i);
    value(points>=range(i) & points<=range(i+1))=coefs(i,1)+activePoints.*(coefs(i,2) + activePoints.*(coefs(i,3) +
coefs(i,4).*activePoints));
end
```

# Name: NNSA1Results.m

```
function [output,RMSETR,RMSETS,trainTime,testTime,nodes]=NNSA1Results(Xt,Xv,xtE,indx,method_apr)
% INPUTS
%         ndim => number of input dimensions for function
%         xtE - cell with extended grid vectors
%         xxa => desired point in for [x1 x2 ... xndim] where
%           x1,x2,...,xndim are the coordinates in each dimension
%         indx - index array
%         X  - training points (last column is the output)
%         method_apr=> method used for approximation
% OUTPUTS
%         out  - approximated value at xxa
trainTime=0;
testTime=0;

[np,ndim]=size(Xv);
nodes=length(xtE{1})^(ndim-1);
```

```
output=zeros(np,1);
tic;
for i=1:np
output(i)=localWrapper(ndim-1,xtE,Xv(i,1:end-1),indx,Xt,method_apr);
end
testTime=testTime+toc;


RMSETR=0;
[~,RMSETS]=computeRMSE(Xv(:,end),output);

function out=localWrapper(ndim,xtE,xxa,indx,X,method_apr)
% INPUTS
%         ndim => number of input dimensions for function
%         xtE - cell with extended grid vectors
%         xxa => desired point in for [x1 x2 ... xndim] where
%           x1,x2,...,xndim are the coordinates in each dimension
%         indx - index array
%         X  - training points (last column is the output)
%         method_apr=> method used for approximation
% OUTPUTS
%         out  - approximated value at xxa

    X4=extractSarray(ndim,xtE,xxa,indx,X);
    out=findValueFrom4array(X4,xxa,method_apr);

function [Xn,In]=extractSarray(ndim,xtE,xx,indx,X)
[nn,~]=size(indx);
kk=zeros(1,ndim);
for j=1:ndim

    kk(j)=find_range(xtE{j},xx(j));
end
sw=zeros(1,ndim);
Xn=zeros(4^ndim,ndim+1);
In=zeros(4^ndim,ndim);
it=1;
for i=1:nn
    for j=1:ndim
        a=indx(i,j);
        k=kk(j);
        if a>=k-1 && a<=k+2
            sw(j)=1;
        else
            sw(j)=0;
        end
    end;
    if all(sw),
        Xn(it,:)=X(i,:);
        In(it,:)=indx(i,:);
        it=it+1;
    end
end


function kk=find_range(Xa,x)
%   Xa long vector
%   find index for x
n=length(Xa);
kk=0;
for k=1:n-1,
    if x>=Xa(k) && x<=Xa(k+1),
        kk=k;
        break
    end
end
if kk==1, kk=2; end;

function out=findValueFrom4array(Xr,xx,method_apr)
%  Xr column array of 4x4x4x...
%  xx value of point
nc=length(xx);
while nc > 0
    Xr=reduceDim4(Xr,xx,nc,method_apr);
    nc=nc-1;
end
out=Xr;


function Xt_new=reduceDim4(Xt,xx,nc,method_apr)
% x  => location of a point
% Xt  => traning data colum vectors with input and output
% nc => index for the last column to be eliminated
% [ni,n2]=size(Xt); % n2=ndim+1  ni number of points
% ndim=n2-1;
x=xx(nc); %single value
nii=4^(nc-1);
out=zeros(nii,1);
Ya=zeros(1,4);
Xa=Ya;
for i=1:nii
    for j=1:4  % this js just to 4x4x4x.... arrays
        Ya(j)=Xt(i+(j-1)*nii,end);   %just for output (end)
        Xa(j)=Xt(i+(j-1)*nii,nc);
    end;
```

```
    switch method_apr
        case 1,  out(i)=linearA(Xa,Ya,x);
        case 2,  out(i)=Spline1d4point(Xa,Ya, x);
        case 3,  out(i)=Spline1d4point4(Xa,Ya, x);
        case 4,  out(i) = Spline_Local_Optimized1(Xa, Ya, x);
    end;
end;
% rearange arrays for output
Xt_new=[Xt(1:nii,1:nc-1),out];

function out=linearA(Xa,YY,x)
kk=find_range(Xa,x);
% kk
x2=Xa(kk+1); x1=Xa(kk);
h=x2-x1; del=x-x1;
out=(YY(kk)*(h-del)  + YY(kk+1)*del)/h;


function output = Spline1d4point(Xa,Ya, x)
%    Xa,Ya are 4 elements vectors
%    x scalar where value maust be calculated
% SPLINE1D_Local  Computes cubic spline interpolating curve
% using central approximation

h=diff(Xa); h1=h(1); h2=h(2); h3=h(3);
dy=diff(Ya); y1=Ya(1);y2=Ya(2);y3=Ya(3); y4=Ya(4);
m2=(dy(2)/h2-dy(1)/h1)/(0.5*(h2+h1));
m3=(dy(3)/h3-dy(2)/h2)/(0.5*(h3+h2));
d=(m3-m2)/(6*h2);
c=(m2+m3)*0.25;
% c=(m2+m3)*0.3;
h2x=h2*h2*0.25;
b=(y3-y2)/h2-d*h2x;
a=(y2+y3)*0.5-c*h2x;
x=x-(Xa(2)+Xa(3))*0.5;
output = a+x*(b+x*(c+x*d));


function output = Spline1d4point4(Xa,Ya, x)
%    Xa,Ya are 4 elements vectors
X(:,4)=ones(4,1);
X(:,3)=Xa;
X(:,2)=X(:,3).*X(:,3);
X(:,1)=X(:,2).*X(:,3);
p=(X\Ya');
output = p(4)+x*(p(3)+x*(p(2)+x*p(1)));


function output = Spline_Local_Optimized1(Xa, Ya, x)
% Given input data x and y, approximates value of y at desired point
% inputs: Xa,Ya:  arrays containing knownn data points.
%       x: value of x at which Ya will be approximated.

dx=diff(Xa); dx1=dx(1); dx2=dx(2); dx3=dx(3);
dy=diff(Ya); dy1=dy(1); dy2=dy(2); dy3=dy(3);
y1=Ya(1);y2=Ya(2);y3=Ya(3); y4=Ya(4);
x1=Xa(1);x2=Xa(2);x3=Xa(3); x4=Xa(4);
x=x-x2;

a=1;
fPrime2=a*(dx1/dx2*dy2+dx2/dx1*dy1)/(dx1+dx2);
fPrime3=a*(dx2/dx3*dy3+dx3/dx2*dy2)/(dx2+dx3);


h2=dx2*dx2; h3=dx2*h2;
a=y2;
b=fPrime2;
d=(dx2*(fPrime3+b)+2*(a-y3))/h3;
c=(y3-a-b*dx2)/h2-d*dx2;

output=a+x*(b+x*(c+x*d));
```

### 6.2.2 Support Functions

Name: computeRMSE.m

```
function [difference,RMSE]=computeRMSE(desired,actual)
difference=desired(:)-actual(:);
SSE=sum(difference.^2);
MSE=SSE/(length(difference));
RMSE=sqrt(MSE);
```

Name: fig.m

```
function h = fig(varargin)
% FIG - Creates a figure with a desired size, no white-space, and several other options.
%
```

```
%       All Matlab figure options are accepted.
%       FIG-specific options of the form FIG('PropertyName',propertyvalue,...)
%       can be used to modify the default behavior, as follows:
%
%       -'units'    : preferred unit for the width and height of the figure
%                      e.g. 'inches', 'centimeters', 'pixels', 'points', 'characters', 'normalized'
%                      Default is 'centimeters'
%
%       -'width'    : width of the figure in units defined by 'units'
%                      Default is 14 centimeters
%                      Note: For IEEE journals, one column wide standard is
%                      8.5cm (3.5in), and two-column width standard is 17cm (7 1/16 in)
%
%       -'height'   : height of the figure in units defined by 'units'
%                      Specifying only one dimension sets the other dimension
%                      to preserve the figure's default aspect ratio.
%
%       -'font'     : The font name for all the texts on the figure, including labels, title, legend, colorbar, etc.
%                      Default is 'Times New Roman'
%
%       -'fontsize' : The font size for all the texts on the figure, including labels, title, legend, colorbar, etc.
%                      Default is 14pt
%
%       -'border'   : Thin white border around the figure (compatible with export_fig -nocrop)
%                      'on', 'off'
%                      Default is 'off'
%
%   FIG(H) makes H the current figure.
%   If figure H does not exist, and H is an integer, a new figure is created with
%   handle H.
%
%   FIG(H,...) applies the properties to the figure H.
%
%   H = FIG(...) returns the handle to the figure created by FIG.
%
%
% Example 1:
%   fig
%
% Example 2:
%   h=fig('units','inches','width',7,'height',2,'font','Helvetica','fontsize',16)
%
%
% Copyright   2012 Reza Shirvany,  matlab.sciences@neverbox.com
% Source:    http://www.mathworks.com/matlabcentral/fileexchange/30736
% Updated:   05/14/2012
% Version:   1.6.5
%



% default arguments
width=14;
font='Times New Roman';
fontsize=14;
units='centimeters';
bgcolor='w';
sborder='off';
flag='';
Pindex=[];

%%%%%%%%%%% process optional arguments
optargin = size(varargin,2);
if optargin>0

% check if a handle is passed in
if isscalar(varargin{1}) && isnumeric(varargin{1})
    flag=[flag '1'];
    i=2;
     if ishghandle(varargin{1})==1
        flag=[flag 'i'];
    end
else
    i=1;
end

% get the property values
while (i <= optargin)
    if (strcmpi(varargin{i}, 'border'))
        if (i >= optargin)
            error('Property value required for: %s', num2str(varargin{i}));
        else
            sborder = varargin{i+1};flag=[flag 'b'];
            i = i + 2;
        end
    elseif (strcmpi(varargin{i}, 'width'))
        if (i >= optargin)
            error('Property value required for: %s', num2str(varargin{i}));
        else
            width = varargin{i+1};flag=[flag 'w'];
            i = i + 2;
```

```
        end
    elseif (strcmpi(varargin{i}, 'height'))
        if (i >= optargin)
            error('Property value required for: %s', num2str(varargin{i}));
        else
            height = varargin{i+1};flag=[flag 'h'];
            i = i + 2;
        end
    elseif (strcmpi(varargin{i}, 'font'))
        if (i >= optargin)
            error('Property value required for: %s', num2str(varargin{i}));
        else
            font = varargin{i+1};flag=[flag 'f'];
            i = i + 2;
        end
    elseif (strcmpi(varargin{i}, 'fontsize'))
        if (i >= optargin)
            error('Property value required for: %s', num2str(varargin{i}));
        else
           fontsize = varargin{i+1};flag=[flag 's'];
            i = i + 2;
        end
    elseif (strcmpi(varargin{i}, 'units'))
        if (i >= optargin)
            error('Property value required for: %s', num2str(varargin{i}));
        else
            units = varargin{i+1};flag=[flag 'u'];
            i = i + 2;
        end
    elseif (strcmpi(varargin{i}, 'color'))
        if (i >= optargin)
            error('Property value required for: %s', num2str(varargin{i}));
        else
            bgcolor = varargin{i+1};flag=[flag 'c'];
            i = i + 2;
        end
    else
        %other figure properties
        if (i >= optargin)
            error('A property value is missing.');
        else
        Pindex = [Pindex i i+1];
        i = i + 2;
        end
    end

end

end

% We use try/catch to handle errors
try

% creat a figure with a given (or new) handle
if length(strfind(flag,'1'))==1
    s=varargin{1};
    if ishandle(s)==1
    set(0, 'CurrentFigure', s);
    else
        figure(s);
    end
else
    s=figure;
end

flag=[flag 's'];

% set other figure properties
if ~isempty(Pindex)
    set(s,varargin{Pindex});
end


% set the background color
set(s, 'color',bgcolor);

% set the font and font size
set(s, 'DefaultTextFontSize', fontsize);
set(s, 'DefaultAxesFontSize', fontsize);
set(s, 'DefaultAxesFontName', font);
set(s, 'DefaultTextFontName', font);

%%%%%%%%%%% set the figure size
% set the root unit
old_units=get(0,'Units');
set(0,'Units',units);

% get the screen size
scrsz = get(0,'ScreenSize');

% set the root unit to its default value
set(0,'Units',old_units);

% set the figure unit
set(s,'Units',units);
```

```
% get the figure's position
pos = get(s, 'Position');
old_pos=pos;
aspectRatio = pos(3)/pos(4);

% set the width and height of the figure
if length(strfind(flag,'w'))==1 && length(strfind(flag,'h'))==1
    pos(3)=width;
    pos(4)=height;
elseif isempty(strfind(flag,'h'))
    pos(3)=width;
    pos(4) = width/aspectRatio;
elseif isempty(strfind(flag,'w')) && length(strfind(flag,'h'))==1
    pos(4)=height;
    pos(3)=height*aspectRatio;
end

% make sure the figure stays in the middle of the screen
diff=old_pos-pos;

 if diff(3)<0
 pos(1)=old_pos(1)+diff(3)/2;
     if pos(1)<0
         pos(1)=0;
     end
 end
 if diff(4)<0
 pos(2)=old_pos(2)+diff(4);
    if pos(2)<0
         pos(2)=0;
     end
 end

% warning if the given width (or height) is greater than the screen size
if pos(3)>scrsz(3)
warning(['Maximum width (screen width) is reached! width=' num2str(scrsz(3)) ' ' units]);
end

if pos(4)>scrsz(4)
warning(['Maximum height (screen height) is reached! height=' num2str(scrsz(4)) ' ' units]);
end

% apply the width, height, and position to the figure
set(s, 'Position', pos);
if strcmpi(sborder, 'off')
    set(s,'DefaultAxesLooseInset',[0,0,0,0]);
end



% handle errors
catch ME
    if isempty(strfind(flag,'i')) && ~isempty(strfind(flag,'s'))
    close(s);
    end
   error(ME.message)
end

s=figure(s);
% return handle if caller requested it.
  if (nargout > 0)
        h =s;
  end
%
% That's all folks!
%
%flag/1iwhfsucsb
```

# Name: generateSchwefelTrainValidation.m

```
function [Xt,xt,Xv,xv,Y_,xtE,nsegE,indx,n]=generateSchwefelTrainValidation(nPoints,alpha,method_gen)
% INPUTS
%         nPoints - vector with number of grid points for each dimension
%         alpha - scalar nonlinear parameter for Schwefel function
%         method_gen - method used to hand edges via extrapolation, see
%                      GenTpoints for details
% OUTPUTS
%         Xt  - training points (last column is the output)
%         xt  - cells of grid vectors
%         Xv  - validation points (last column is the desired output)
%         xv  - cells of vectors
%            Validation point are automaticaly selected in the middle between grids
%            Each time you run the same data point should be generated
%         Y_  - Training point function values for extended grid
%         xtE - cell with extended grid vectors
%         nsegE - extended grid size
%         indx - index array
%         n   - number of points in extended grid
[xt,xv]=initEqual(nPoints);
```

```
[Y_,xtE,nsegE]=GenTpoints(nPoints, xt, alpha,method_gen);

[Y_,scal,offset]=normaliArray(Y_, 2);

[Xv,xv]=GenVpoints(nPoints, alpha, xv);
Xv(:,end)=scal*Xv(:,end)-offset;
Xt=genInps(Y_,xtE,nsegE);
[indx,n]=FindInd(nsegE);

function [xt,xv]=initEqual(nPoints)
%% Description: Creates the grid vectors for training and testing data sets
% INPUTS
%         nPoints - vector with number of grid points for each dimension
% OUTPUTS
%         xt  - cell array with each entry containing the grid vector for
%               the corresponding dimension of the training set
%         xv  - cell array with each entry containing the grid vector for
%               the corresponding dimension of the validation set
%%
ndim=length(nPoints);
xt = cell(1,ndim);
xv = cell(1,ndim);
for j=1:ndim
    xa=linspace(-1,1,nPoints(j));
    xb=xa(1:end-1)+0.5*diff(xa);
    xt{j}=xa;
    xv{j}=xb;
end


function [TrainData,xtE,nPointsE]=GenTpoints(nPoints, xt, alpha,method)
%% Description: Generate extended training dataset
% INPUTS
%         nPoints - vector with number of grid points for each dimension
%         xt      - cell array with each entry containing the grid vector for
%                   the corresponding dimension of the training set
%         alpha   - scalar nonlinear parameter for Schwefel function
%         method  - select the method for edge extrapolation
% OUTPUTS
%         TrainData  - matrix with training points (last column is the output)
%         xtE  - extended cell array with each entry containing the grid vector for
%                the corresponding dimension of the training set
%         nPointsE - vector with number of grid points for each dimension
%                    after extrapolation
%%
ndim=length(nPoints);
nPointsE=nPoints+2; % add beginning and ending values for each dimension
xtE = cell(1,ndim);
for i=1:ndim,  % extend cells form xt => xxt
    point=xt{i};
    xl=2*point(1)-point(2);     xr=2*point(end)-point(end-1); % calculate left and right values
    xtE{i} = [xl,point,xr];
end;

[indt,nt]=FindInd(nPointsE); % get index matrix
pointer =findPointer(nPointsE); % get jumo size for each dimension

TrainData=zeros(1,nt); % initialize array to hold training data
for i=1:nt
    ind=indt(i,:); % ind is current index
    loc=checkForBE(ind,nPointsE);

    %% Calculate values at training points, or extrapolation values for the edges
    XX_=[]; Y_=[];
    if loc>0,  %calculate Y values only if found a single one
        X_=xtE{loc};  lx=length(X_);
        for j=1:ndim
            XX_(j,:)=ones(1,lx)*xtE{j}(ind(j));
        end
        XX_(loc,:)=X_;
        for k=2:lx-1
            point=XX_(:,k)';
            Y_(k) = Schwefel(point,alpha);
        end
        Xa=X_(2:5);         Ya=Y_(2:5);
        ya=findVirtualpoint(Xa,Ya,0,method);
        Y_(1)=ya;
        Xa=X_(lx-4:lx-1);         Ya=Y_(lx-4:lx-1);
        yb= findVirtualpoint(Xa,Ya,1,method);
        Y_(lx)=yb;
        for k=1:lx
            ind(loc)=k;
            iii=CalcI(ind,pointer);
            TrainData(iii)=Y_(k);
        end
    end
end


function [Xv,xv]=GenVpoints(nPoints, alpha, xv)
%% Description: Validation point are automaticaly selected in the middle between grids
% INPUTS
%         nPoints - vector with number of grid points for each dimension
%         alpha - scalar nonlinear parameter for Schwefel function
%         xv  - cell array with each entry containing the grid vector for
```

```
%               the corresponding dimension of the validation set
% OUTPUTS
%         TestData  - matrix with validation points (last column is the output)
%         xv  - matrix with training points (last column is the output)
%%
ndim=length(nPoints);
[indv,~]=FindInd(nPoints-1);
point = zeros(1,ndim);
Xv = zeros(size(indv,1),ndim+1);
for i=1:size(indv,1)
    for j=1:ndim
        point(j)=xv{j}(indv(i,j));
    end
     y = Schwefel(point,alpha);
    Xv(i,:)=[point,y];
end


function [indx,n]=FindInd(nPointsE)
%% Description: Creates a matrix with the indices for the training data
% INPUTS
%         nPointsE - vector with number of grid points for each dimension
%                    after extrapolation
% OUTPUTS
%         indx  - matrix of indices
%         n  - number of indices
%%
ndim=length(nPointsE); n=prod(nPointsE);
index=ones(n,ndim); ind=ones(1,ndim);
for i=1:n
    index(i,:)=ind;
    ind(1)=ind(1)+1;
    k=1;
    while(ind(k)>nPointsE(k))
        ind(k)=1;
        k=k+1;
        if k==ndim+1
            break;
        end;
        ind(k)=ind(k)+1;
    end;
end;
indx=int16(index);


function pointer =findPointer(nPointsE)
%% Description: Finds pointer is a vector of size of nsegE indicating size of jumps.
% INPUTS
%         nPointsE - vector with number of grid points for each dimension
%                    after extrapolation
% OUTPUTS
%         pointer  - array with the jump size for each dimension
%%
ndim = length(nPointsE);
pointer = zeros(1,ndim);
pointer(2)=nPointsE(1);
for i=3:ndim
    pointer(i)=pointer(i-1)*nPointsE(i-1);
end;


function loc=checkForBE(ind,nPointsE)
%% Description: Check for occurances of ones and max in ind.
% INPUTS
%         ind - current index
%         nPointsE - vector with number of grid points for each dimension
%                    after extrapolation
% OUTPUTS
%         loc  - returns location of first 1 in ind, or zero if there are
%                none or any ind value is at max
%%
ndim = length(ind);  s1 = 0;
for i=1:ndim
    if (ind(i)==1)
        s1 = s1+1;
        ii=i;
    end
end
for i=1:ndim,
    if ind(i)==nPointsE(i)
        s1=0;
    end; %exclude end case
end

if s1==1,
    loc=ii;
else
    loc=0;
end;

function  [yv,xv]=findVirtualpoint(Xa,Ya,sw,method_gen)
%% Description: Calculates linear index given ind array and pointer (offsets).
% INPUTS
%         Xa - vector of length 4 with the x locations
%         Ya - vector of length 4 with the function values
```

```
%         sw - switch for left/right end of the array. sw=0 for left, 1 for
%              right
%         method_gen - extrapolation method
% OUTPUTS
%         yv - extrapolated y value
%         xv - corresponding x value
%%
if length(Xa)~=length(Ya), disp('error in findVirtualpoint length(Xa)~=length(Ya)'); pause; end;
if sw, Xa=fliplr(Xa); Ya=fliplr(Ya); end; % switching R_edge to L_edge
xv=2*Xa(1)-Xa(2);
% Xa
% Ya
% pause
switch method_gen,
    case 1, % linear
        yv=2*Ya(1)-Ya(2);
    case 2 %method from the paper to find quadratic =>faster
        % for gamma=1 gives the same results as quadratic but it is much faster
        dX=diff(Xa);dY=diff(Ya); h0=dX(1); y1=Ya(1);
        k1=dY(1)/dX(1); k2=dY(2)/dX(2);
        del1=h0*k1; del2=(k2-k1)*h0;
        yv=y1-del1+del2;
    case 3 % using the 3-rd order interpolation faster
        X(1,:)=Xa.^3; X(2,:)=Xa.^2; X(3,:)=Xa; X(4,:)=ones(1,4);   p=(X'\Ya')';
        yv = polyval(p,xv);
    case 4 % using the quadratic faster => actualy slower
        xx=Xa(1:3); yy=Ya(1:3);
        X(2,:)=Xa(1:3); X(1,:)=X(2,:).^2; X(3,:)=ones(1,3);   p=(X'\yy')';
        yv = polyval(p,xv);
    case 5 %method from the paper with gamma parameter
        % for gamma=1 gives the same results as quadratic but it is much faster
        gamma=1;
        dX=diff(Xa);dY=diff(Ya); h0=dX(1); y1=Ya(1);
        k1=dY(1)/dX(1); k2=dY(2)/dX(2);
        del1=h0*k1; del2=(k2-k1)*h0;
        yv=y1-del1+gamma*del2;
    case 6 % using the 3-rd order interpolation - slower
        p = polyfit(Xa(1:4),Ya(1:4),3);  %can be faster with a line below
        yv = polyval(p,xv);

    case 7, % quadratic slower => atualy is much faster
        p = polyfit(Xa(1:3),Ya(1:3),2);  %can be faster with a line below
        yv = polyval(p,xv);
    case 8 %method from the paper to find linear =>faster
        % for gamma=0 gives the same results as linear
        dX=diff(Xa);dY=diff(Ya); h0=dX(1); y1=Ya(1);
        k1=dY(1)/dX(1);
        del1=h0*k1;
        yv=y1-del1;
    otherwise, disp('error method is not specified')
end

return

function index=CalcI(ind,pointer)
%% Description: Calculates linear index given ind array and pointer (offsets).
% INPUTS
%         point - vector with each entry corresponding to the loction in
%         each dimension of the desired point.
%         ind - current multidimensional index
% OUTPUTS
%         index  - linear index
%%
index=ind(1);  ind=ind-1; ndim=length(ind);
for i=2:ndim,     index=index+ind(i)*pointer(i);  end;

function  X=genInps(Y_,xtE,nPointsE)
%% Description: Generates input matrix.
% INPUTS
%         Y_  - Training point function values for extended grid
%         xtE - cell with extended grid vectors
%         nPointsE - vector with number of grid points for each dimension
%                    after extrapolation
% OUTPUTS
%         X  - Matrix with training data
%%
[indx,n]=FindInd(nPointsE);

ndim=length(nPointsE);
X = zeros(n,ndim+1);
for i=1:n,
    for j=1:ndim
        X(i,j)=xtE{j}(indx(i,j));
    end;
end;
X(:,end)=Y_';


function out = Schwefel(point,alpha)
%% Description: Calculates the value of the Schwefel function for the given point and alpha value.
% INPUTS
%         point - vector with each entry corresponding to the loction in
%         each dimension of the desired point.
%         alpha - nonlinear parameter for Schwefel function
% OUTPUTS
```

```
%         out  - value of the Schwefel function at the desired point
%%
ndim=length(point);
out=0;
for i=1:ndim
    x=point(i)*alpha;
    out=out-x.*sin(sqrt(abs(x)));
end


function [Xn,scal,offset]=normaliArray(X, type)
% typ   0  => STD; 1  => UNI;  2  => BIP;
X_=X(:);
ma=max(X_) ;
mi=min(X_) ;
dell=ma-mi;
sd=std(X);  avg=mean(X);
switch type
    case 0           % xn=x/std  STD
        scal=1./sd; offset=avg./sd;
    case 1           % (0, +1)  UNI
        scal=1./dell;   offset= mi./dell;
    case 2           % (-1, +1) BIP
        scal=2/dell;   offset= 2*mi/dell+1;
end;
Xn=scal*X-offset;
return;
```

# Name: GenTpoints.m

```
function [xtE,X,indt]=GenTpoints(nseg, xt,X,method,F)
% INPUTS
%         nseg => vector with number of gird points for each dimention
%         alpha => nonlinear parameter for Schwefel function
%         xt=> cells of vectors  (if not specified then linspace is created
% OUTPUTS
%         Xt  - training points (last column is the output)
%% inputs:
% -nseg - a vector containing the number of grid points in each dimension
%
% -xt - cell array of vectors that define a grid
%
% -X - matrix of n row and d columns, where n is the number of grid points
% and d is the number of input dimensions.  should be in the ndgrid format.
%  Example: a=linspace(-1,1,5);
%           b=linspace(-1,1,5);
%           [aa,bb]=ndgrid(a,b);
%           X=[a(:), bb(:)];
%
% -method - method used for edge value extrapolation, see findVirtualpoint
% function

% -F - column vector of length n containing function values or
% 'pole heights' at each point in X
%  Example: for i=1:n
%               F(i)=g(X(i,:)); % assume g() is some function
%           end
%% outputs:
%
% -xtE - cell array of vectors that define a grid
%
% -X - matrix of n row and d+1 columns.  X(:,1:end-1)=locations, X(:end)=values
%
% -indt - matrix with prod(nsegE) rows and d columns.  Each row is a
% subscript into the grid defined by the grid vectors, and coressponds to
% each row in X
%%
ndim=length(nseg);
nsegE=nseg+2; % add begining and ending values for each dimention
for i=1:ndim,  % extend cells form xt => xxt
    xx=xt{i};     xl=2*xx(1)-xx(2);     xr=2*xx(end)-xx(end-1);
    xxx=[xl,xx,xr];     xtE{i}=xxx;
end;
[indt,nt]=FindInd3(nsegE);
indt=int16(indt);
pointer =findPointer(nsegE);
% Mscal=findScal(alpha);
% xtE{1}
% xtE{2}
% YY=zeros(1,nt);
YY=zeros(1,nt);
for i=1:nt
    ind=indt(i,:);
    loc=checkForBE(ind,nsegE);
% loc=checkForBE(ind,nseg);
%         loc=checkForOnes(ind)
    XX_=[]; Y_=[];
    if loc>0,  %calculate Y values only if found a single one
        X_=xtE{loc};  lx=length(X_);
        for j=1:ndim
```

```
            XX_(j,:)=ones(1,lx)*xtE{j}(ind(j));
        end
        XX_(loc,:)=X_;

        for k=2:lx-1
            xx=XX_(:,k)';

%             Y_(k)=Schwefel2(xx, alpha, Mscal) ;

%         Y_(k)=fun(xx,alpha,functionSwitch);
% Y_(k)=F1(xx(1),xx(2));
%         for p=1:ndim
%             subs(p)=xt{i}
%         end
%         index=find(TrainData(:,1:end-1)==xx);
        [~,index]=ismember(xx,X,'rows');
        Y_(k)=F(index);
%         xp(count,:)=[xx count];
%         count=count+1;
        end
%         count=count+1;
        Xa=X_(2:5);         Ya=Y_(2:5);
        ya=findVirtualpoint(Xa,Ya,0,method);
        Y_(1)=ya;
        Xa=X_(lx-4:lx-1);         Ya=Y_(lx-4:lx-1);
        yb= findVirtualpoint(Xa,Ya,1,method);
        Y_(lx)=yb;
        for k=1:lx
            ind(loc)=k;
            iii=CalcI(ind,pointer);
%             finalI(cc)=iii;
%             cc=cc+1;
            YY(iii)=Y_(k);
        end
    end
end

X=genInps(YY,xtE,nsegE);

function [indx,n]=FindInd3(max)
%% inx  -  without removal duplicates
% to get full list use index
b=length(max); n=prod(max);
index=ones(n,b); ind=ones(1,b);
for i=1:n
    index(i,:)=ind;
    ind(1)=ind(1)+1;
    k=1;
    while(ind(k)>max(k))
        ind(k)=1;
        k=k+1;
        if k==b+1
            break;
        end;
        ind(k)=ind(k)+1;
    end;
end;
indx=int16(index);

function pointer =findPointer(nsegE)
% pointer is a vector of size of nsegE indicating size of jumps
% pointer is used in the following way to find iii
%         iii=ind(1);  ind=ind-1;
%         for i=2:ndim,     iii=iii+ind(i)*pointer(i);  end;
ndim= length(nsegE); pointer(2)=nsegE(1);
for i=3:ndim,
    pointer(i)=pointer(i-1)*nsegE(i-1);
end;

function loc=checkForBE(ind,nsegE)
% cheking occurances of ones and max in the string: ind
% returning location of one
lx=length(ind);  s1=0;
for i=1:lx,
    if ind(i)==1, s1=s1+1; ii=i; end;
end
for i=1:lx,
    if ind(i)==nsegE(i), s1=0; end; %exclude end case
end

if s1==1, loc=ii; else loc=0; end;


function  [yv,xv]=findVirtualpoint(Xa,Ya,sw,method_gen)
% returning value and posintion if virtual point
% Xa and Ya are 4 elements vectors
% sw==1 for upper end
if length(Xa)~=length(Ya), disp('error in findVirtualpoint length(Xa)~=length(Ya)'); pause; end;
if sw, Xa=fliplr(Xa); Ya=fliplr(Ya); end; % switching R_edge to L_edge
xv=2*Xa(1)-Xa(2);
% Xa
% Ya
% pause
switch method_gen,
    case 1, % linear
```

```
        yv=2*Ya(1)-Ya(2);
    case 2 %method from the paper to find quadratic =>faster
        % for gamma=1 gives the same results as quadratic but it is much faster
        dX=diff(Xa);dY=diff(Ya); h0=dX(1); y1=Ya(1);
        k1=dY(1)/dX(1); k2=dY(2)/dX(2);
        del1=h0*k1; del2=(k2-k1)*h0;
        yv=y1-del1+del2;
    case 3 % using the 3-rd order interpolation faster
        X(1,:)=Xa.^3; X(2,:)=Xa.^2; X(3,:)=Xa; X(4,:)=ones(1,4);   p=(X'\Ya')';
        yv = polyval(p,xv);
    case 4 % using the quadratic faster => actualy slower
        xx=Xa(1:3); yy=Ya(1:3);
        X(2,:)=Xa(1:3); X(1,:)=X(2,:).^2; X(3,:)=ones(1,3);   p=(X'\yy')';
        yv = polyval(p,xv);
    case 5 %method from the paper with gamma parameter
        % for gamma=1 gives the same results as quadratic but it is much faster
        gamma=1;
        dX=diff(Xa);dY=diff(Ya); h0=dX(1); y1=Ya(1);
        k1=dY(1)/dX(1); k2=dY(2)/dX(2);
        del1=h0*k1; del2=(k2-k1)*h0;
        yv=y1-del1+gamma*del2;
    case 6 % using the 3-rd order interpolation - slower
        p = polyfit(Xa(1:4),Ya(1:4),3);  %can be faster with a line below
        yv = polyval(p,xv);

    case 7, % quadratic slower => atualy is much faster
        p = polyfit(Xa(1:3),Ya(1:3),2);  %can be faster with a line below
        yv = polyval(p,xv);
    case 8 %method from the paper to find linear =>faster
        % for gamma=0 gives the same results as linear
        dX=diff(Xa);dY=diff(Ya); h0=dX(1); y1=Ya(1);
        k1=dY(1)/dX(1);
        del1=h0*k1;
        yv=y1-del1;
    otherwise, disp('error method is not specified')
end

return

function iii=CalcI(ind,pointer)
iii=ind(1);  ind=ind-1; ndim=length(ind);
for i=2:ndim,     iii=iii+ind(i)*pointer(i);  end;

function  X=genInps(Y_,xtE,nsegE)
[indx,n]=FindInd3(nsegE);
ndim=length(nsegE);
for i=1:n,
    for j=1:ndim
        X(i,j)=xtE{j}(indx(i,j));
    end;
end;
X=[X,Y_'];
```

### 6.2.3 Peaks Experimental Results Code

Name: m_PeaksRandomTest.m

```
clear all;
testn=1000;trainn=2000;
load peaksTrain2400.mat
load peaksTest1000.mat
%%

trainTime=0;
testTime=0;
tic;
gridn=8;
xt{1}=linspace(-1,1,gridn);xt{2}=linspace(-1,1,gridn);

tic;
[xgrid,ygrid]=ndgrid(xt{1},xt{2});
zgrid=griddata(TrainData(:,1),TrainData(:,2),TrainData(:,3),xgrid,ygrid);
trainTime=trainTime+toc;
X=[xgrid(:), ygrid(:)]; F=zgrid(:);

nt=trainn;
[np,d]=size(TestData);

nseg=zeros(1,d-1);
for i=1:d-1
    nseg(i)=length(xt{i});
end


ndim = size(TrainData,2)-1;
n=sqrt(size(TrainData,1));
method=3;
[xtE,X,indx]=GenTpoints(nseg, xt,X,method,F);
nsegE = nseg + 2;
%%
splineTraintime = toc;
```

```
tskTraintime = splineTraintime;
nnsaTraintime = splineTraintime;
%%
xv = xtE;
ztemp = reshape(X(:,end),nsegE);
ztemp = ztemp.';
[~,~,splineRMSETR,~,~,~]=splineResults([X(:,1:2) ztemp(:)],TrainData,trainn,xtE,nsegE,xv);
%% Fuzzy TSK Results

[~,~,tskRMSETR,~,~,~]=FuzzyTSKResults(xtE,X,TrainData,trainn);
%% NNSA Results
method_apr=4;
[~,~,nnsaRMSETR,~,~,~]=NNSA1Results(X,TrainData,xtE,indx,method_apr);

%%


ntrial=10;
%% Strip out values outside of [-1,+1]
TrainData2=TrainData(1:trainn,:);
%% MLP Results
sizes=[40];
epoch_n=100;
[MLP_Result,mlpRMSETR,mlpRMSETS,mlpTraintime,mlpTesttime,net,mlpNodes]=MLPResults(TrainData2,TestData,sizes,ntrial,epoch_n);
%% FCC Results
sizes=[10];
epoch_n=20;
[FCC_Result,fccRMSETR,fccRMSETS,fccTraintime,fccTesttime,fccNodes,topo,best_w,act,gain,paramt,iw]=FCCResults(TrainData2,TestData,
sizes,ntrial,epoch_n,testn);
%% SVR Results
gamma_list=[0.01,0.1,1,3,10];
C_list=[10,50,100,150,300];
[SVR_Result,svrRMSETR,svrRMSETS,svrTraintime,svrTesttime,model,svrNodes,C_bst,gamma_bst]=SVRResults(TrainData2,TestData,gamma_lis
t,C_list);
%% ELM Results
 nodeV=[10 15 25 50 60];
[ELM_Result,elmRMSETR,elmRMSETS,elmTraintime,elmTesttime,inw,outw,bias,elmNodes]=ELMResults(TrainData2,TestData,nodeV,ntrial);

%% ANFIS Results
numMFs=[3,3]; inmftype='gbellmf'; outmftype='linear'; epoch_n = 10;
[anfis_Result,anfisRMSETR,anfisRMSETS,anfisTraintime,anfisTesttime,out_fis,anfisNodes]=fuzzyResults(TrainData2,TestData,numMFs,in
mftype,outmftype,epoch_n);

%% Global Spline Results
xv = xtE;
ztemp = reshape(X(:,end),nsegE);
ztemp = ztemp.';

[Global_Result,~,splineRMSETS,~,splineTesttime,splineNodes]=splineResults([X(:,1:2) ztemp(:)],TestData,trainn,xtE,nsegE,xv);
%% Fuzzy TSK Results

[TSK_Result,~,tskRMSETS,~,tskTesttime,tskNodes]=FuzzyTSKResults(xtE,[X(:,1:2) ztemp(:)],TestData,trainn);
%% NNSA Results
method_apr=4;
[NNSA_Result,~,nnsaRMSETS,~,nnsaTesttime,nnsaNodes]=NNSA1Results(X,TestData,xtE,indx,method_apr);

%%
save('peaksTrialsData')
%%
plotSurfaces()
%% Print Time, Train, and Test Error to console
clear all;
load peaksTrialsData

disp('FCC');
disp(sprintf('trTime %.4f, tsTime %.4f, RMSETR=%.4f, RMSETS=%.4f,
Nodes=%d',fccTraintime,fccTesttime,fccRMSETR,fccRMSETS,fccNodes));
disp('MLP');
disp(sprintf('trTime %.4f, tsTime %.4f, RMSETR=%.4f, RMSETS=%.4f,
Nodes=%d',mlpTraintime,mlpTesttime,mlpRMSETR,mlpRMSETS,mlpNodes));
disp('SVM');
disp(sprintf('trTime %.4f, tsTime %.4f, RMSETR=%.4f, RMSETS=%.4f,
Nodes=%d',svrTraintime,svrTesttime,svrRMSETR,svrRMSETS,svrNodes));
disp('ELM');
disp(sprintf('trTime %.4f, tsTime %.4f, RMSETR=%.4f, RMSETS=%.4f,
Nodes=%d',elmTraintime,elmTesttime,elmRMSETR,elmRMSETS,elmNodes));
disp('ANFIS');
disp(sprintf('trTime %.4f, tsTime %.4f, RMSETR=%.4f, RMSETS=%.4f,
Nodes=%d',anfisTraintime,anfisTesttime,anfisRMSETR,anfisRMSETS,anfisNodes));
disp('Spline');
disp(sprintf('trTime %.4f, tsTime %.4f, RMSETR=%.4f, RMSETS=%.4f,
Nodes=%d',splineTraintime,splineTesttime,splineRMSETR,splineRMSETS,splineNodes));
disp('TSK');
disp(sprintf('trTime %.4f, tsTime %.4f, RMSETR=%.4f, RMSETS=%.4f,
Nodes=%d',tskTraintime,tskTesttime,tskRMSETR,tskRMSETS,tskNodes));
disp('NNSA');
disp(sprintf('trTime %.4f, tsTime %.4f, RMSETR=%.4f, RMSETS=%.4f,
Nodes=%d',nnsaTraintime,nnsaTesttime,nnsaRMSETR,nnsaRMSETS,nnsaNodes));
```

Name: plotSurfaces.m

```
function plotSurfaces
load peaksTrialsData
```

```
load peaksPlot900

%% Global Results
[Global_Result,~,~,~,~]=splineResults([X(:,1:2) ztemp(:)],PlotData,trainn,xtE,nsegE,xv);

%% NNSA Results
[NNSA_Result,~,~,~,~,~]=NNSA1Results(X,PlotData,xtE,indx,method_apr);

%% Fuzzy TSK Results
[TSK_Result,~,~,~,~,~]=FuzzyTSKResults(xtE,[X(:,1:2) ztemp(:)],PlotData,trainn);

%% Fuzzy Results
Fuzzy_Result=evalfis(PlotData(:,1:end-1),out_fis);

%% MLP Results
MLP_Result=net(PlotData(:,1:end-1).').';

%% FCC Results
FCC_Result=calc_fwd(PlotData(:,1:end-1),topo,best_w,act,gain,paramt,iw);

%% SVR Results
[SVR_Result, ~, ~] = svmpredict( 2*PlotData(:,end)-1,PlotData(:,1:end-1), model);
SVR_Result=(SVR_Result+1)/2;

%% ELM Results
[~,ELM_Result]=calcO(PlotData(:,1:end-1),PlotData(:,end),inw,outw,bias,elmNodes);ELM_Result=ELM_Result.';

%% In color
res=30;
string=strcat('Desired');
h=figure(1);clf;
surf(reshape(PlotData(:,1),res,res),reshape(PlotData(:,2),res,res),reshape(PlotData(:,3),res,res));
view(3); axis tight;hold all; title(string); %scatter3(xi(:),yi(:),zi(:),'r*');
fig(h,'units','inches','width',3.5)

string=strcat('SPLINE RMSE:',32,num2str(splineRMSETS));
h=figure(2);clf;
surf(reshape(PlotData(:,1),res,res),reshape(PlotData(:,2),res,res),reshape(Global_Result,res,res));
view(3); axis tight;hold all; title(string); %scatter3(xi(:),yi(:),zi(:),'r*');
fig(h,'units','inches','width',3.5)

string=strcat('NNSA RMSE:',32,num2str(nnsaRMSETS));
h=figure(3);clf;
surf(reshape(PlotData(:,1),res,res),reshape(PlotData(:,2),res,res),reshape(NNSA_Result,res,res));
view(3); axis tight;hold all; title(string); %scatter3(xi(:),yi(:),zi(:),'r*');
fig(h,'units','inches','width',3.5)

string=strcat('TSK RMSE:',32,num2str(tskRMSETS));
h=figure(4);clf;
surf(reshape(PlotData(:,1),res,res),reshape(PlotData(:,2),res,res),reshape(TSK_Result,res,res));
view(3); axis tight;hold all; title(string); %scatter3(xi(:),yi(:),zi(:),'r*');
fig(h,'units','inches','width',3.5)

string=strcat('ANFIS RMSE:',32,num2str(anfisRMSETS));
h=figure(5);clf;
surf(reshape(PlotData(:,1),res,res),reshape(PlotData(:,2),res,res),reshape(Fuzzy_Result,res,res));
view(3); axis tight;hold all; title(string); %scatter3(xi(:),yi(:),zi(:),'r*');
fig(h,'units','inches','width',3.5)

string=strcat('MLP RMSE:',32,num2str(mlpRMSETS));
h=figure(6);clf;
surf(reshape(PlotData(:,1),res,res),reshape(PlotData(:,2),res,res),reshape(MLP_Result,res,res));
view(3); axis tight;hold all; title(string); %scatter3(xi(:),yi(:),zi(:),'r*');
fig(h,'units','inches','width',3.5)

string=strcat('SVM RMSE:',32,num2str(svrRMSETS));
h=figure(7);clf;
surf(reshape(PlotData(:,1),res,res),reshape(PlotData(:,2),res,res),reshape(SVR_Result,res,res));
view(3); axis tight;hold all; title(string); %scatter3(xi(:),yi(:),zi(:),'r*');
fig(h,'units','inches','width',3.5)

string=strcat('ELM RMSE:',32,num2str(elmRMSETS));
h=figure(8);clf;
surf(reshape(PlotData(:,1),res,res),reshape(PlotData(:,2),res,res),reshape(ELM_Result,res,res));
view(3); axis tight;hold all; title(string); %scatter3(xi(:),yi(:),zi(:),'r*');
fig(h,'units','inches','width',3.5)

string=strcat('FCC RMSE:',32,num2str(fccRMSETS));
h=figure(10);clf;
surf(reshape(PlotData(:,1),res,res),reshape(PlotData(:,2),res,res),reshape(FCC_Result,res,res));
view(3); axis tight;hold all; title(string); %scatter3(xi(:),yi(:),zi(:),'r*');
fig(h,'units','inches','width',3.5)


function [y] = calc_fwd(inp,topo,w,act,gain,param,iw)
np = size(inp,1);           % number of pattern
ni = param(2);           % number of input
no = param(3);           % number of output
nn = param(5);           % number of neurons
y = zeros(np,no);
for p = 1:np     % number of patterns
    node(1:ni) = inp(p,1:ni);
    for n = 1:nn % number of neurons
        j = ni + n;
        net = w(iw(n));
```

```
        for i = (iw(n)+1):(iw(n+1)-1)
            net = net + node(topo(i))*w(i);
        end;
        out=actFunc(n,net,act,gain);
        node(j) = out;
    end;
    y(p,:)=node(ni+nn-no+1:ni+nn);
end;

function [SSE,O]=calcO(x, y, inw, outw, bias, nodes)
[np,nd]=size(x);
for i=1:nodes
    for j=1:np
        H(j,i)=1/(1+exp(-(inw(i,:)*x(j,:)'+bias(i))));
    end
end
O=outw'*H';
er=y-O'; SSE=er'*er;

function out=actFunc(n,net,act,gain)
de=0;
switch act(n)
    case 0, out = gain(n)*net;                          % linear neuron
    case 1, out = 1/(1+exp(-gain(n)*net));              % unipolar neuron
    case 2, out = tanh(gain(n)*net);                    % bipolar neuron
    case 3, out = gain(n)*net/(1+gain(n)*abs(net));     % bipolar elliot neuron
    case 4, out = 2*gain(n)*net/(1+gain(n)*abs(net))-1; % unipolar elliot neuron
    case 5, out = 2/(1+exp(-gain(n)*net))-1;            % bipolar from NBN 2.08
    case 6,
%         out = sign(gain(n)*net);
        if (abs(gain(n)*net)>=1)
            out = sign(gain(n)*net);            % hard activation
        else
            out = gain(n)*net;
        end
end;
```

### 6.2.4 Forward Kinematics Results Code

Name: m_ForwardKinematicsTest.m

```
clear all; format compact;
% test_2d_with_edges

filename = 'normForwardKinTrainData.dat';
TrainData = dlmread(filename);

filename = 'normForwardKinTestData.dat';
TestData = dlmread(filename);
% TrainData = TrainData(:,[1 2 3]);
% fnamebase = 'forwardKinX2d';
% TestData = TestData(:,[1 2 3]);
TrainData = TrainData(:,[1 2 4]);
fnamebase = 'forwardKinY2d';
TestData = TestData(:,[1 2 4]);
%%
ndim = size(TrainData,2)-1;
n=sqrt(size(TrainData,1));

xt = cell(1,ndim);
nseg = zeros(1, ndim);
for i = 1:ndim
    xt{i} = TrainData(1:n,1).';
    nseg(i) = length(xt{i});
end
[xtE,TrainData,indx]=GenTpoints(nseg, xt,TrainData(:,1:ndim),3,TrainData(:,end));
nsegE = nseg + 2;

method_apr=4;
trainn = size(TrainData,1);
testn = size(TestData, 1);
ntrial=10;

%% MLP Results
sizes=[10,20,50,60,70];
epoch_n=100;
[~,mlpRMSETR,mlpRMSETS,mlpTraintime,mlpTesttime,~,mlpNodes]=MLPResults(TrainData,TestData,sizes,ntrial,epoch_n);
%% FCC Results

sizes=[4,8,10];
epoch_n=20;
[~,fccRMSETR,fccRMSETS,fccTraintime,fccTesttime,fccNodes,~,~,~,~,~,~]=FCCResults(TrainData,TestData,sizes,ntrial,epoch_n,testn);
%% SVR Results

 gamma_list=2.^[0 -3 -4];
 C_list=2.^[0 3 6];

[~,svrRMSETR,svrRMSETS,svrTraintime,svrTesttime,~,svrNodes,C_bst,gamma_bst]=SVRResults(TrainData,TestData,gamma_list,C_list);
```

```
%% ELM Results

 nodeV=[10 15 25 50 60];
[~,elmRMSETR,elmRMSETS,elmTraintime,elmTesttime,~,~,~,elmNodes]=ELMResults(TrainData,TestData,nodeV,ntrial);
%% ANFIS Results
numMFs = 4*ones(size(nseg));
inmftype='gbellmf'; outmftype='linear'; epoch_n = 10;
[~,anfisRMSETR,anfisRMSETS,anfisTraintime,anfisTesttime,~,anfisNodes]=fuzzyResults(TrainData,TestData,numMFs,inmftype,outmftype,e
poch_n);
%% Global Spline Results
xv = xtE;
[Global_Result,splineRMSETR,splineRMSETS,splineTraintime,splineTesttime,splineNodes]=splineResults(TrainData,TestData,trainn,xtE,
nsegE,xv);
Global_Result=reshape(Global_Result,sqrt(size(TestData,1)),sqrt(size(TestData,1)));Global_Result=Global_Result.';Global_Result=Gl
obal_Result(:);
[~,splineRMSETS]=computeRMSE(TestData(:,end),Global_Result(:));
%% Fuzzy TSK Results
[TSK_Result,tskRMSETR,tskRMSETS,tskTraintime,tskTesttime,tskNodes]=FuzzyTSKResults(xtE,TrainData,TestData,trainn);
TSK_Result=reshape(TSK_Result,sqrt(size(TestData,1)),sqrt(size(TestData,1)));TSK_Result=TSK_Result.';TSK_Result=TSK_Result(:);
[~,tskRMSETS]=computeRMSE(TestData(:,end),TSK_Result(:));
%% NNSA Results

[~,nnsaRMSETR,nnsaRMSETS,nnsaTraintime,nnsaTesttime,nnsaNodes]=NNSA1Results(TrainData,TestData,xtE,indx,method_apr);
%%

algorithmNames = {'MLP';'FCC';'SVM';'ELM';'ANFIS';'TSK';'Spline';'NNSA'};
trainTime = [mlpTraintime;fccTraintime;svrTraintime;elmTraintime;anfisTraintime;tskTraintime;splineTraintime;nnsaTraintime];
testTime = [mlpTesttime;fccTesttime;svrTesttime;elmTesttime;anfisTesttime;tskTesttime;splineTesttime;nnsaTesttime];
totalTime = trainTime + testTime;
trainRMSE = [mlpRMSETR;fccRMSETR;svrRMSETR;elmRMSETR;anfisRMSETR;tskRMSETR;splineRMSETR;nnsaRMSETR];
testRMSE = [mlpRMSETS;fccRMSETS;svrRMSETS;elmRMSETS;anfisRMSETS;tskRMSETS;splineRMSETS;nnsaRMSETS];
nodes = [mlpNodes;fccNodes;svrNodes;elmNodes;anfisNodes;tskNodes;splineNodes;nnsaNodes];
trialResults = table(algorithmNames,trainTime,testTime,totalTime,trainRMSE,testRMSE,nodes);

%%
writetable(trialResults,strcat(fnamebase,'.xlsx'))
```

### 6.2.5 Schwefel Function Results Code

Name: m_Schwefel_multidim_batch.m

```
clear;
fnameBase = 'schwefelResults';

% alphas = [10,20,30,50,100];
% ndims = [2,3,4,5];
% points = [6,6,6,8,10];
alphas = [10,20];
ndims = [2,3];
points = [6,6];
method_gen = 3;

for i=1:length(ndims)
    for j=1:length(alphas)
        alpha=alphas(j);
        nPoints = points(j)*ones(1,ndims(i));

        trialResults = runSchwefelTrial(nPoints,alpha,method_gen);
        filename = strcat(fnameBase,'Dim',num2str(ndims(i)),'Alpha',num2str(alpha),'Npt',num2str(points(j)));
        writetable(trialResults,filename)
    end
end
```

Name: runSchwefelTrial.m

```
clear;
fnameBase = 'schwefelResults';
algorithmNames = {'MLP';'FCC';'SVM';'ELM';'ANFIS';'TSK';'Spline';'NNSA'};
nAlg = length(algorithmNames);
% alphas = [10,20,30,50,100];
% ndims = [2,3,4,5];
% points = [6,6,6,8,10];

alphas = [10,20];
ndims = [2,3];
points = [6,6];

trialResults = table();
for i=1:length(ndims)
    for j=1:length(alphas)

        alpha=alphas(j);
        filename = strcat(fnameBase,'Dim',num2str(ndims(i)),'Alpha',num2str(alpha),'Npt',num2str(points(j)));
        trial = readtable(filename);
        trialResults = vertcat(trialResults,trial);
    end
```

```
end

writetable(trialResults,fnameBase)

trialResults = readtable(fnameBase);

%%
writetable(trialResults,strcat(fnameBase,'.xlsx'))
%%
figureWidth=3.5;
fontSize = 15;
lineWidth = 2;
fields = {'trainTime','testTime','totalTime','trainRMSE','testRMSE'};
titles = {'Train Time(s)','Test Time(s)','Total Time(s)','Train RMSE','Test RMSE'};
plotStyles = {'-r^','-rd',':b>','--bx','--g<',':gs',':kv','-.ko'};
for k=1:length(fields)
for j=1:length(alphas)
h=figure(k*10+j);clf;hold all;
for i = 1:nAlg
title(strcat(titles{k},32,'alpha =',32,num2str(alphas(j))));
rows1 = strcmp(trialResults.Algorithm, algorithmNames{i});
rows2 = trialResults.Alpha==alphas(j);
rows = rows1 & rows2;
t1 = trialResults(rows,fields{k});
%%
plot(ndims(1:length(t1{:,fields{k}})),t1{:,fields{k}},plotStyles{i},'LineWidth',lineWidth)
set(gca,'xtick', ndims)
set(gca,'xticklabel',{'2','3','4','5'})
xlabel('Dimensions');
ylabel(titles{k});
end
% fig(h,'units','inches','width',figureWidth)
% legend(algorithmNames,'Location','best');
set(gca,'FontSize',fontSize,'fontWeight','bold','FontName','Times New Roman')
set(findall(gcf,'type','text'),'FontSize',fontSize,'fontWeight','bold','FontName','Times New Roman')
end
end
%%


%%
fields = {'trainTime','testTime','totalTime'};
titles = {'Train Time(s)','Test Time(s)','Total Time(s)'};
plotStyles = {'-r^','-rd',':b>','--bx','--g<',':gs',':kv','-.ko'};
for k=1:length(fields)
for j=1:length(alphas)
h=figure(k*10+j);clf;
for i = 1:nAlg

rows1 = strcmp(trialResults.Algorithm, algorithmNames{i});
rows2 = trialResults.Alpha==alphas(j);
rows = rows1 & rows2;
t1 = trialResults(rows,fields{k});

% plot(ndims(1:length(t1{:,fields{k}})),t1{:,fields{k}},plotStyles{i})
semilogy(ndims(1:length(t1{:,fields{k}})),t1{:,fields{k}},plotStyles{i},'LineWidth',lineWidth);hold all;

end
title(strcat(titles{k},32,'alpha =',32,num2str(alphas(j))));
set(gca,'xtick', ndims)
set(gca,'xticklabel',{'2','3','4','5'})
xlabel('Dimensions');
ylabel(titles{k});
% fig(h,'units','inches','width',figureWidth)
% legend(algorithmNames,'Location','best');
set(gca,'FontSize',fontSize,'fontWeight','bold','FontName','Times New Roman')
set(findall(gcf,'type','text'),'FontSize',fontSize,'fontWeight','bold','FontName','Times New Roman')
end
end
```

## Name: m_Schwefel_multidim_data_analysis.m

```
function trialResults = runSchwefelTrial(nPoints,alpha,method_gen)
[TrainData,xt,TestData,xv,~,xtE,nsegE,indx,n]=generateSchwefelTrainValidation(nPoints,alpha,method_gen);
method_apr=4;
trainn = size(TrainData,1);
testn = size(TestData, 1);
ntrial=10;
%% MLP Results
sizes=[10 15 25 50 60];
epoch_n=100;
[~,mlpRMSETR,mlpRMSETS,mlpTraintime,mlpTesttime,~,mlpNodes]=MLPResults(TrainData,TestData,sizes,ntrial,epoch_n);
%% FCC Results
sizes=[3,5,7,9,10];
epoch_n=20;
[~,fccRMSETR,fccRMSETS,fccTraintime,fccTesttime,fccNodes,~,~,~,~,~,~]=FCCResults(TrainData,TestData,sizes,ntrial,epoch_n,testn);
%% SVR Results
gamma_list=[0.01,0.1,1,3,10];
C_list=[10,50,100,150,300];
[~,svrRMSETR,svrRMSETS,svrTraintime,svrTesttime,~,svrNodes,~,~]=SVRResults(TrainData,TestData,gamma_list,C_list);
%% ELM Results
```

```
 nodeV=[10 15 25 50 60];
[~,elmRMSETR,elmRMSETS,elmTraintime,elmTesttime,~,~,~,elmNodes]=ELMResults(TrainData,TestData,nodeV,ntrial);
%% ANFIS Results

numMFs=3*ones(1,length(nPoints));
inmftype='gbellmf'; outmftype='linear'; epoch_n = 10;

[~,anfisRMSETR,anfisRMSETS,anfisTraintime,anfisTesttime,~,anfisNodes]=fuzzyResults(TrainData,TestData,numMFs,inmftype,outmftype,e
poch_n);
%% Global Spline Results
[~,splineRMSETR,splineRMSETS,splineTraintime,splineTesttime,splineNodes]=splineResults(TrainData,TestData,trainn,xtE,nsegE,xv);
%% Fuzzy TSK Results
[~,tskRMSETR,tskRMSETS,tskTraintime,tskTesttime,tskNodes]=FuzzyTSKResults(xtE,TrainData,TestData,trainn);
%% NNSA Results
[~,nnsaRMSETR,nnsaRMSETS,nnsaTraintime,nnsaTesttime,nnsaNodes]=NNSA1Results(TrainData,TestData,xtE,indx,method_apr);

%%
Algorithm = {'MLP';'FCC';'SVM';'ELM';'ANFIS';'TSK';'Spline';'NNSA'};
trainTime = [mlpTraintime;fccTraintime;svrTraintime;elmTraintime;anfisTraintime;tskTraintime;splineTraintime;nnsaTraintime];
testTime = [mlpTesttime;fccTesttime;svrTesttime;elmTesttime;anfisTesttime;tskTesttime;splineTesttime;nnsaTesttime];
trainRMSE = [mlpRMSETR;fccRMSETR;svrRMSETR;elmRMSETR;anfisRMSETR;tskRMSETR;splineRMSETR;nnsaRMSETR];
testRMSE = [mlpRMSETS;fccRMSETS;svrRMSETS;elmRMSETS;anfisRMSETS;tskRMSETS;splineRMSETS;nnsaRMSETS];
nodes = [mlpNodes;fccNodes;svrNodes;elmNodes;anfisNodes;tskNodes;splineNodes;nnsaNodes];
Dim = size(TrainData,2) -1;
Dimension = Dim*ones(size(Algorithm));
totalTime = trainTime+testTime;
Alpha = alpha*ones(size(Algorithm));
trialResults = table(Algorithm,Dimension,Alpha,trainTime,testTime,totalTime,trainRMSE,testRMSE,nodes);
```