SELF ADAPTIVE FINITE ELEMENT ANALYSIS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

In Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Shawhin Roudbari

January 2006

ABSTRACT

From the behavior of steel under thermal loads to the response of organic tissue to various stimuli, engineering research strives to develop constitutive models that allow us to understand and predict the physical world around us. Characterizing and understanding the constitutive behavior of materials is a pursuit limited by the expense and time associated with conducting and interpreting laboratory experiments. The focus of this thesis is to extend the development of an innovative computational method that aims to help circumvent the need for extensive tests as a basis for obtaining accurate and precise material response models.

Specifically, this research involves the Autoprogressive training of Neural Networks for the inverse estimation of heat transport material models. This methodology, the Self Adaptive Finite Element Analysis (SAFEA), combines a Neural Network based Constitutive Model (NNCM) with a nonlinear Finite Element Program in an algorithm which uses very basic conductivity measurements to produce a constitutive model of the material under study: Through manipulating a series of Neural Network embedded Finite Element Analyses, it is demonstrated that an accurate constitutive model for a highly nonlinear material can be evolved and retained as an object to be used in the analysis of any problem involving the material under study.

This thesis details the theoretical development of the SAFEA algorithm and provides a simulation of the SAFEA program through a steady-state non-linear heat transfer problem. The SAFEA program, coded in MATLAB, is included in appendix.

BIOGRAPHICAL SKETCH

The author currently practices structural engineering in San Francisco. Originally from Iran, he moved to the United States in 1997 for his continued studies. After a BS in Civil and Environmental Engineering from UC Berkeley, the author obtained an MS in Structural Engineering with minors in Geotechnical Engineering and Engineering Management from Cornell University.

The author plans to return to Iran in the future, with hopes of helping develop the country and region's infrastructure, and by this means helping strengthen ties and promote dialogue between regional states in an effort to improve the reliability and capabilities of the region's engineering and construction industries.

In his personal time, the author enjoys reading, cycling, martial arts, dance, hiking, and most recently guitar.

ACKNOWLEDGMENTS

The author extends his gratitude and sincere thanks to Dr. Wilkins Aquino, for his support, enthusiasm, and most of all his energy and patience over many long and suspenseful hours of developing this research, as a friend, professor, and as an advisor.

The author also deeply thanks his friends and colleagues for many useful discussions and their invaluable encouragement. John Brigham, Professor Kifle Gebremedhin, and Dr. Wu share in all aspects of the development of this research, and their hard work and enthusiasm are sincerely appreciated.

Finally, the author would like to thank his minor advisors and the faculty and administration of the department of Civil and Environmental Engineering for their wisdom, guidance, and patience.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

BC: Boundary Condition

BVP: Boundary Value Problem

FEA: Finite Element Analysis

HT: Heat Transfer

IBVP: Initial and Boundary Value Problem

NLFEA: Nonlinear Finite Element Analysis

NN: Neural Network

NNCM: Neural Network Constitutive Model

NNFEA: Neural Network embedded Finite Element Analysis

SAFEA: Self-Adaptive Finite Element Analysis

CHAPTER 1
INTRODUCTION

1.1. Background and Motivation

Of central importance in any engineering field is the knowledge of material behavior. Whether the behavior reflects strength, heat and mass transfer, electromagnetic, or other properties, the material response allows engineers to design and analyze physical entities. The behavior of steel under complex states of stress or under thermal loads in structural engineering, the response of organic tissue to surrounding fluids and local stimuli, or the response of superconductive media are examples of the important role that knowledge of material response presents in engineering applications.

Correspondingly, because understanding material response is so important, much research has gone into its quantification. From empirical equations obtained through physical testing (such as load-deformation tests) to mechanistic relations based on development of fundamental material laws, scientists and engineers are able to gather the required information for their purposes in analyzing and manipulating the materials we use everyday.

Many means exist to characterize structural and material behavior. Physical testing is one of the more common and familiar means. A civil engineer might set up a five-story structure on a shake table to measure its seismic response, or a mechanical engineer might test the stability of a new composite under extreme temperature in order to determine the material's suitability in aerospace applications. While these tests serve to validate a theoretical model or provide a response spectrum to be used in future

analyses, they are bound by the constraints of test setups and cost and time. Because lab testing often proves unfeasible, many researchers have looked into techniques that allow for inverse estimation of material behavior from the service conditions of structures.

The focus of this thesis is the development of an innovative computational method for inverse estimation of heat transport material models from very basic physical tests. This methodology rests on concepts of artificial intelligence (e.g. neural networks) and conventional computational mechanics tools such as finite element analysis.

1.2. Concept

The methodology presented in this thesis consists in having a neural network representation of a material model incorporated into a non-linear finite element code and training this neural network using the autoprogressive rule [7]. Autoprogressive training differs significantly from other training schemes commonly used in conventional neural network models in the sense that there is not a known material behavior to train the neural network a priori, but the material model is extracted from global measurements using non-linear finite element analysis. This new methodology for finding solutions to inverse problems is termed Self-Adaptive Finite Element Analysis (SAFEA) throughout this text. The title “self-adaptive finite element analysis” stems from the fact that the Neural Network (NN) produces its own training data during iterative finite element analyses. Once a neural network constitutive model is trained, it can be used as a conventional material model in any given numerical scheme (i.e. finite elements, finite differences, etc.).

1.3. State of Research

Although the application of neural networks to the general constitutive modeling of thermal bodies is new, material models involving neural networks have been used in elasticity and limitedly in heat and mass transfer [1, 5, 7, 13].

In their paper on the Autoprogressive Method, Ghaboussi et. al. [7] set the benchmark wherein they recover the nonlinear elastic behavior of complex structures and laminar geometries by training neural networks to capture basic element response. For example, by "teaching" a neural network constitutive model the behavior of a single steel rod in a basic tensile test, they were able to predict the response of an array of these single bars (arranged as a planar truss) as measured in a physical test of the structure in its nonlinear range. It is the same principals of the Autoprogressive method that have been applied in this thesis to the constitutive modeling in heat transfer problems – though there are subtle details differentiating the two applications (in elasticity and heat transfer) beyond what is evident from near identical governing equations; these details will be discussed throughout the following chapters.

As for research in heat transfer, other research groups have linked neural networks to material response, but not to constitutive modeling in the sense of determining the relationship between heat flux, temperature, and temperature gradient in general [1, 5, 12]. While their research provides many interesting insights and results from accurately training neural networks a material response in a given test, they fail to bridge the important gap that would allow a trained neural network constitutive model to predict a material's response in *any* boundary value problem. Put more concisely,

current research with neural networks in heat transfer focuses on reproducing individual lab test results; therefore the gap that remains is using neural networks to learn material response – in general – in heat transfer problems.

CHAPTER 2
NEURAL NETWORK CONSTITUTIVE MODELS IN FINITE ELEMENT ANALYSIS

2.1. Introduction

A central concept utilized in this research is the use of neural networks as the constitutive model in numerical analyses. Conventionally, constitutive models may be represented by numerical constants or algebraic equations, such as Young's Modulus in linear elasticity. Using Neural Network Constitutive Models (NNCMs) opens the way for incorporating the benefits of artificial intelligence into our analyses. In this chapter therefore, first neural networks, followed by the finite element formulation of transient heat transfer problems, and finally the combination of the two via constitutive models will be explained in depth.

The review of neural networks presented here is limited to those aspects that are pertinent to this research. Though the field of neural networks is vast and covers many applications, the aspects used here are limited. The topics covered include neural network structure, neural network training, and a discussion on the data given to and obtained from a neural network. For a detailed discussion on neural networks, texts such as Reed and Marks' Neural Smithing[11] should be referenced.

A review of the finite element formulation will follow the discussion of neural networks, in order to set the foundation for developing the incorporation of neural networks into a finite element analysis.

2.2. Neural Networks

In order to build a basic yet solid understanding of what a neural network is and how one functions, it is helpful to describe a neural network first by its structure, then its training, and finally how data is given to and obtained from the neural network.

2.2.1. Neural Network Structure

A neural network is a system of interconnected nodes (or neurons) that coarsely approximates the behavior of the human brain. Each node represents a simple functional calculation, i.e. a hyperbolic tangent, sigmoid, or any other function that spans from -1 to +1 in its range and a subset of real numbers as the domain [11]. Neural network nodes are attached by connections each of which also involves a function (Figure 1). The function at each connection consists of weight factors multiplied by the output value from the node at the beginning of the connection. The resultant value after these two operations, at the first neuron and then at the connection, is then plugged into the neuron at the terminal end of the connection, and so on through the network. The nodes and connections between them constitute the fundamental elements of a neural network.

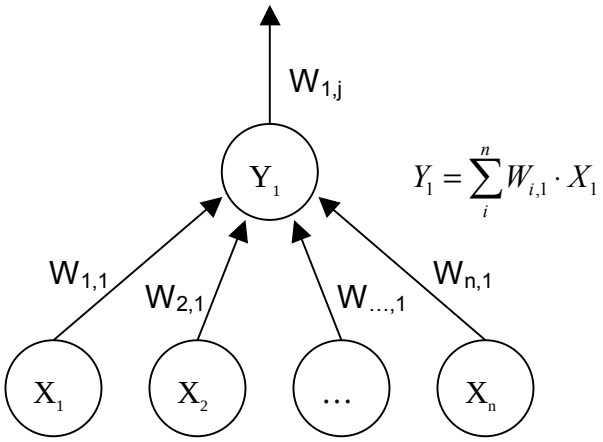


Figure 1. Neural network weights [11].

The layout or "architecture" of a neural network consists of nodes arrayed in layers; typically these layers consist of one input layer, a series of intermediate layers, and one output layer (Figure 2).

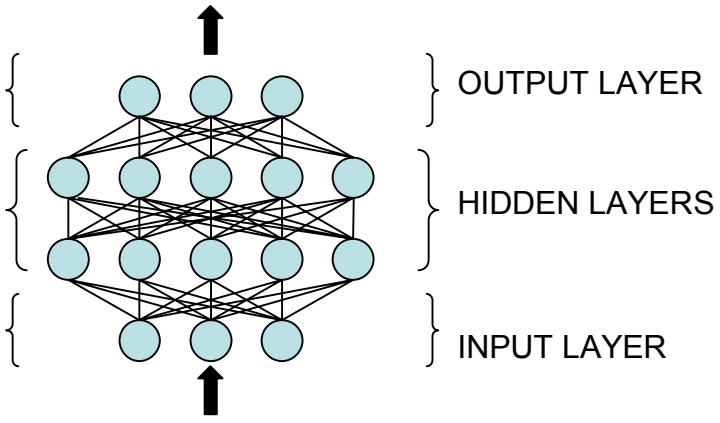


Figure 2. Neural network architecture [11].

The input data, or independent values, reside in the nodes of the input layer and that the output data, or dependent values, reside in the output layers (see Figure 2 above).

To clarify, consider the simple relationship between stress and strain: $\sigma=E\varepsilon$ (stress is equal to Young's Modulus times strain). Assume an experiment in which certain stresses are produced and the corresponding strains are measured. The imposed stresses are the independent values and the strains the dependent values. Plotting the dependents against the independents, strain vs. stress, and fitting a line to the data, a line with a slope of E is obtained (assuming a linear elastic response range of a material) as in Figure 3.

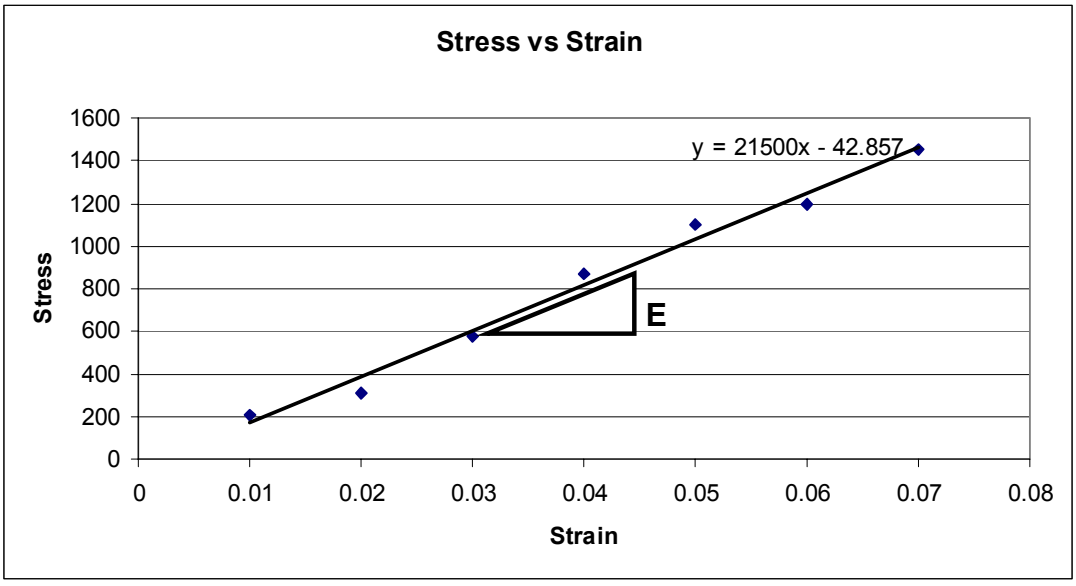


Figure 3. Relationship between stress and strain data.

In the context of the neural network approach, the independent data is fed into the input node of the neural network and the dependent data into the output node of the neural network (Figure 4). Note that in this example, since there is only one dependent variable and one independent variable, the input and output layers each consist of only one node.

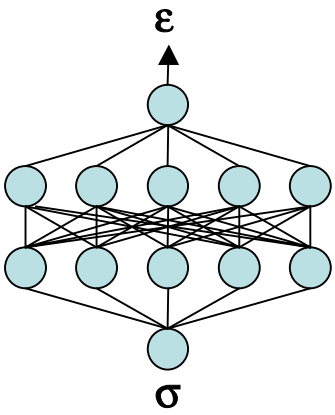


Figure 4. Schematic of a Neural Network [11].

When the neural network is trained, it will *learn* the relationship between the stress and strain data, thereby replacing Young's Modulus in the more conventional analysis described above. Ghaboussi et. al.[7] symbolize this relationship as:

$$\sigma = NN\{\ldots\}\varepsilon$$

Where the '…' in brackets is replaced by details of the NN architecture; specifically the number of nodes in the input and output layers as well as the number of nodes in the layers in between, termed "hidden layers."

In conclusion, it should be noted that the number of hidden layers and the number of nodes in those layers can be defined by the user and will affect the accuracy and precision by which the NN matches the data, and will affect how fast the NN trains. The number of nodes in the input and output layer are however fixed and reflect the number of independent and dependent variables respectively.

2.2.2. Neural Network Training

After establishing familiarity with the structure of a neural, the key element in manipulating one becomes understanding the training process: what values are fed to the neural network and in what stage of development? To what error tolerance is the data matched? How many nodes arrayed in how many hidden layers produces the most effective NN? These are among the many questions that translate into important heuristics when building models and performing analyses that involve developing neural networks.

As a neural network trains, it learns the relationship between inputs and outputs with increasing precision and accuracy. Different methods of training exist, but in keeping with the scope of this thesis, only the method used in this research is described.

The training of a neural network is a series of optimization iterations in which a *predicted* output is driven to match the *required* outputs. Referring to the stress-strain example above, the neural network is required to predict the dependent data from the independent data at the end of training, wherein stress values are input and the corresponding strain values are obtained. That is, from a given set of inputs an output set, Y_i, is predicted based on initial weight values (recall the weights at the connections of neurons

described above). Since the initial weights are random, the initial output estimation will be incorrect. Given that the true input, I_0, as well as the true output, Y_0, data is used during training – the error of the true output, Y_0, with respect to the initially estimated output data, Y_i is calculated – it is this error that is minimized to a predetermined error tolerance through iterations of modifying the weights at the connections between the neurons of the neural network. For a rigorous explanation of the process, Chapter 3 of Reed and Marks' Neural Smithing may be referenced [11].

Once the neural network is satisfactorily trained, it is expected that by feeding it only independent data, the NN will predict the desired outputs – based on "trained" weights – to a degree of accuracy contingent on the degree and error prescribed during training.

2.2.3. Neural Network Data Input and Output

With a background on neural network structure and training, the way data is fed to and obtained from a neural network can now be described. Designing and training a neural network, though complex in their own rite, only represent the initiation steps in using a neural network, for its main purpose is returning predicted results from given inputs. Thus, in order to use any NN, it is essential to understand how to input data to a neural network and how to read what comes out. In the process of this description, reference will be given to the earlier stress/strain example.

Consider that a neural network with specified architecture has been trained with measurement data: a set of input and outputs; stimulation and response; or stress and strain. At this point the neural network is sufficiently initialized to perform its function – to read new input data and output what is

estimated as the proper response. For example, if a neural network is given new stress values one by one, running the NN program will yield a list of corresponding strains. If the NN was trained well, the stress and strain values will plot close to the trend that was measured. Figure 5 gives a sample plot of inputs vs. NN outputs for this example. It should be noted that a NN is not generally used for simple linear relationships, which can be easily approximated by other means, and that this example is purely representational.

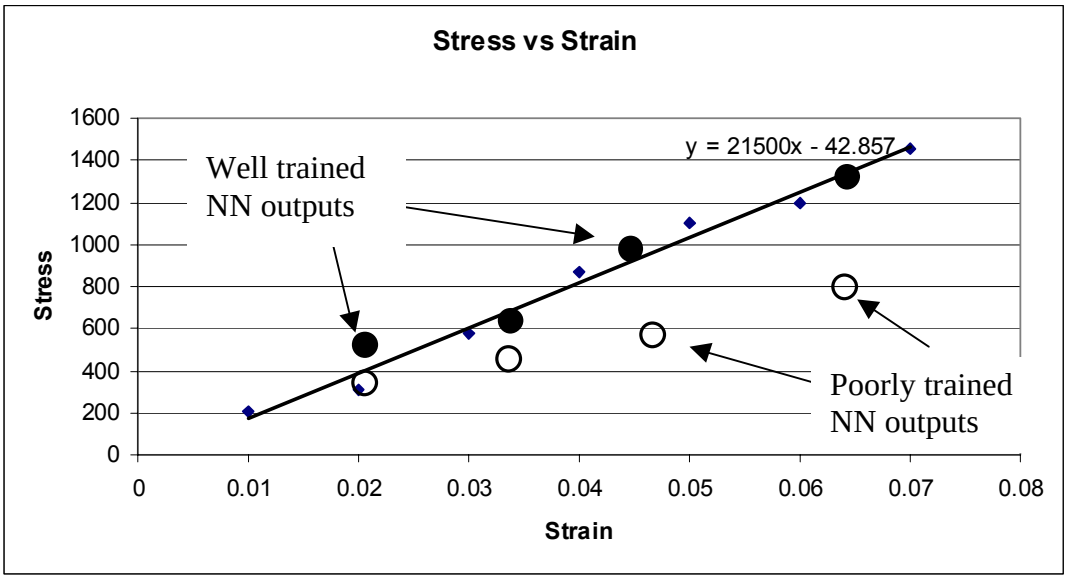


Figure 5. Hypothetical Neural Network input vs. output. (Compare with curve from Figure 3.)

Inputs fed to a neural network are normalized values that are arrayed in a vector. The neural network program runs a forward analysis with entries of the input vector. This run is a forward analysis because the data travels through the functions (at the nodes and the connections) of the neural

network only once from input layer toward output layer through all the neurons and all the connections. Thus this “feed forward” analysis, as it is termed, is performed on the input data and the neural network with its optimized connection weights will yield a vector of output values [11]. These output values are then de-normalized, according to the same normalizing factor used on the inputs, and plotted or processed as desired.

In this stage of neural network operation there are many heuristics inherent in the process that stem from variables such as the neuron function, normalizing factors, etc. These heuristics are paramount in successfully utilizing neural networks (and are discussed throughout this text).

This coverage of neural network inputs and outputs, together with the previous discussions of NN structure and training provide the required background to investigate next how to use a neural network constitutive model within a finite element analysis. Prior to that, however, a very brief review of the finite element formulation pertaining to the example of this research (nonlinear heat transfer) is presented. This formulation will allow a smoother subsequent discussion of where exactly the neural network constitutive model fits into the finite element analysis.

2.3. Finite Element Formulation

The finite element method is used to find approximate solutions to governing differential equations of a physical process. Note that although the problem tackled in this research is a nonlinear steady state heat transfer problem, the more general nonlinear *transient* formulation is presented.

Beginning with the strong form of the heat equation, we have the governing differential equation:

$$(\kappa T_{,i})_{,i} + q_B = \rho c \dot{T} \tag{1}$$

This equation follows from the conservation of energy and recognition that the net heat conducted out, in addition to the heat generated within, is equal to the change in energy stored within a thermal body.

Next, applying a virtual temperature [8] we obtain,

$$\overline{T}\{(\kappa \cdot T_{,i})_{,i} + q_B - \rho \cdot c \cdot \dot{T}\} = 0$$

Integrating the above expression over the domain, we get,

$$\int_V \overline{T}(\kappa \cdot T_i)_i dV + \int_V \overline{T} \cdot q_B dV - \int_V \overline{T} \cdot \rho \cdot c\dot{T} dV = 0$$

then using the product rule of differentiation on the first term, the above equation becomes,

$$\int_V (\overline{T} \cdot \kappa \cdot T_i)_i dV - \int_V \overline{T}_i (\kappa \cdot T_i) dV + \int_V \overline{T} \cdot q_B dV - \int_V \overline{T} \cdot \rho \cdot c\dot{T} dV = 0$$

Observing the divergence theorem on the first term, yields

$$\int_S \overline{T}(\kappa \cdot T_i) \cdot n_i dS - \int_V \overline{T}_i (\kappa \cdot T_i) dV + \int_V \overline{T} q_B dV - \int_V \overline{T} \cdot \rho \cdot c\dot{T} dV = 0$$

The following step utilizes shape functions and their derivatives, symbolized as **N** and **B** respectively.

Given that:

$$T = NT_e$$

$$\dot{T} = N\dot{T}_e$$

$$\overline{T} = N\overline{T}_e$$

$$T_i = BT_e$$

$$\kappa T_i = -q_i$$

$$q_i = h \cdot (T - T_f)$$

therefore:

$$\int_S N\overline{T_e}(\kappa \cdot {}_i) \cdot n_i dS - \int_V B\overline{T_e}(\kappa \cdot T_i)dV + \int_V N\overline{T_e} \cdot q_B dV - \int_V N\overline{T_e} \cdot \rho \cdot c\dot{T} dV = 0$$

and:

$$\int_S N^T h(NT_e - T_f) \cdot n_i dS - \int_V B^T \kappa B T_e dV + \int_V N q_B dV - \int_V N^T \rho c N \dot{T}_e dV = 0$$

Let:

$$k = \int_V B^T \kappa B T_e dV \tag{2}$$

$$r_H = \int_S N^T h T_f dS \tag{3}$$

$$h = \int_S N^T h N dS \tag{4}$$

$$c = \int_V N^T \rho c N dV \tag{5}$$

$$r_Q = \int_V N q_B dV \tag{6}$$

Therefore,

$$hT_e - r_H - kT_e + r_Q - c\dot{T}_e = 0 \tag{7}$$

or,

$$K_T T_e + c\dot{T}_e = R_T \tag{8}$$

where

$$K_T = k - h$$

$$R_T = r_Q - r_H$$

This final equation, a matrix equation, is used to compute the temperature, gradient, and flux fields in a given initial and boundary value problem.

When programming the finite element analysis, Equations 8, along with Equations 2 through 6 are calculated using Gaussian Quadrature – a numerical method of integration [10]. It is at these integration points that the Neural Network Constitutive Model is used to evaluate heat flux given the temperature and temperature gradient. When neural networks are used as constitutive models in finite element analyses, material parameters such as thermal conductivity are immaterial. However, the derivatives of the heat flux with respect to the temperature gradient are used during the solution of the nonlinear system of equations in (7). These derivatives are equivalent to thermal conductivity in materials that obey Fourier's law.

CHAPTER 3
THE SELF-ADAPTIVE FINITE ELEMENT ANALYSIS

3.1. Introduction

This chapter gives a comprehensive development of the SAFEA starting with the physical setup, i.e. lab test, of a material for which the constitutive behavior is desired, through the algorithm of the SAFEA, to the end product: a nonlinear constitutive model that captures the response of the material under examination. The nonlinear constitutive model is a neural network that can be utilized in subsequent finite element analyses of the material in any boundary value problem.

3.2. Test Setup for SAFEA

A solid understanding of the SAFEA method requires familiarity with the physical test setup upon which the analysis is performed. While some fundamental quantities and procedures require definition in order to proceed with a description of the SAFEA method, the intricacies of the laboratory setup are not in the scope of this thesis.

The experimental setup involves a body composed of a chosen material under investigation. There are discrete measurement points along a boundary of the material at which temperatures are measured. Heat flux is applied along the same boundary at (multiple) load steps. The other boundaries are kept at a constant temperature (Figure 6).

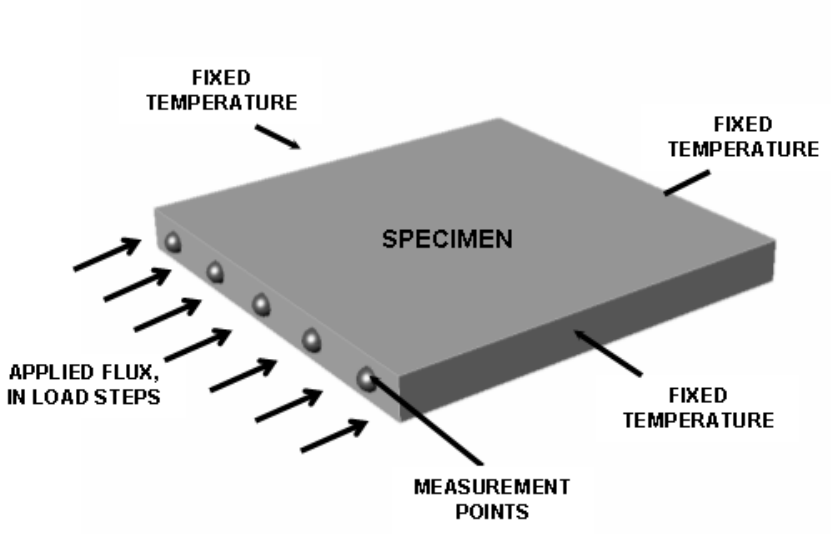


Figure 6. Laboratory test setup.

In the case of a steady-state heat transfer analysis (as in this thesis) the body is subjected to an applied flux, for example *load step 1* along the chosen boundary, all other boundaries are maintained at a fixed temperature, and once equilibrium is reached at these conditions, the temperatures at the measurement points are recorded (along the same boundary where the flux load was applied).

Next, this procedure is repeated with a higher applied flux – *load step 2*. Again the other boundaries maintain the same fixed temperature as in the previous load step. When equilibrium is reached, measurement point temperatures are measured and recorded.

This process is iterated for *n load steps*, resulting in *n* sets of temperature readings. An analogous test would be that of applying a lateral force on a structural frame in increments (load steps) and recording displacements at different points in the structure at each load increment.

The self-adaptive finite element analysis begins after test measurements and corresponding loading values are obtained, after which the data is input to the SAFEA program. The program will perform iterations of neural network embedded finite element analyses until the neural network constitutive model is adequately trained to replicate the response of the material under investigation.

3.3. Self-Adaptive Finite Element Analysis Structure

It is helpful to first understand the structure of the SAFEA program before focusing on the individual steps in the analysis.

The steps in the SAFEA program are as follows:

I. *Create a finite element representation of the physical test setup:* body dimensions, meshing, and boundary conditions are all specified.

II. *Pre-train the Neural Network:* the neural network constitutive model is trained to a linear material response.

III. *First Pass:* each pass is an iterative cycle that reproduces all the load steps of the lab test.

   i. First load step:

      a) First local iteration:

         1) Run a flux controlled Neural Network embedded finite element analysis

         2) Compare the error between the finite element analysis of step (1) and the results of the lab test at the first load step. If the error is satisfactory, then skip

to the next load step (return to step (i)); otherwise continue.

3) Run temperature controlled neural network embedded finite element analysis (perturb solution).
4) Train NN constitutive model to learn the effect of the perturbance.

Return to (1)

Next local iteration…

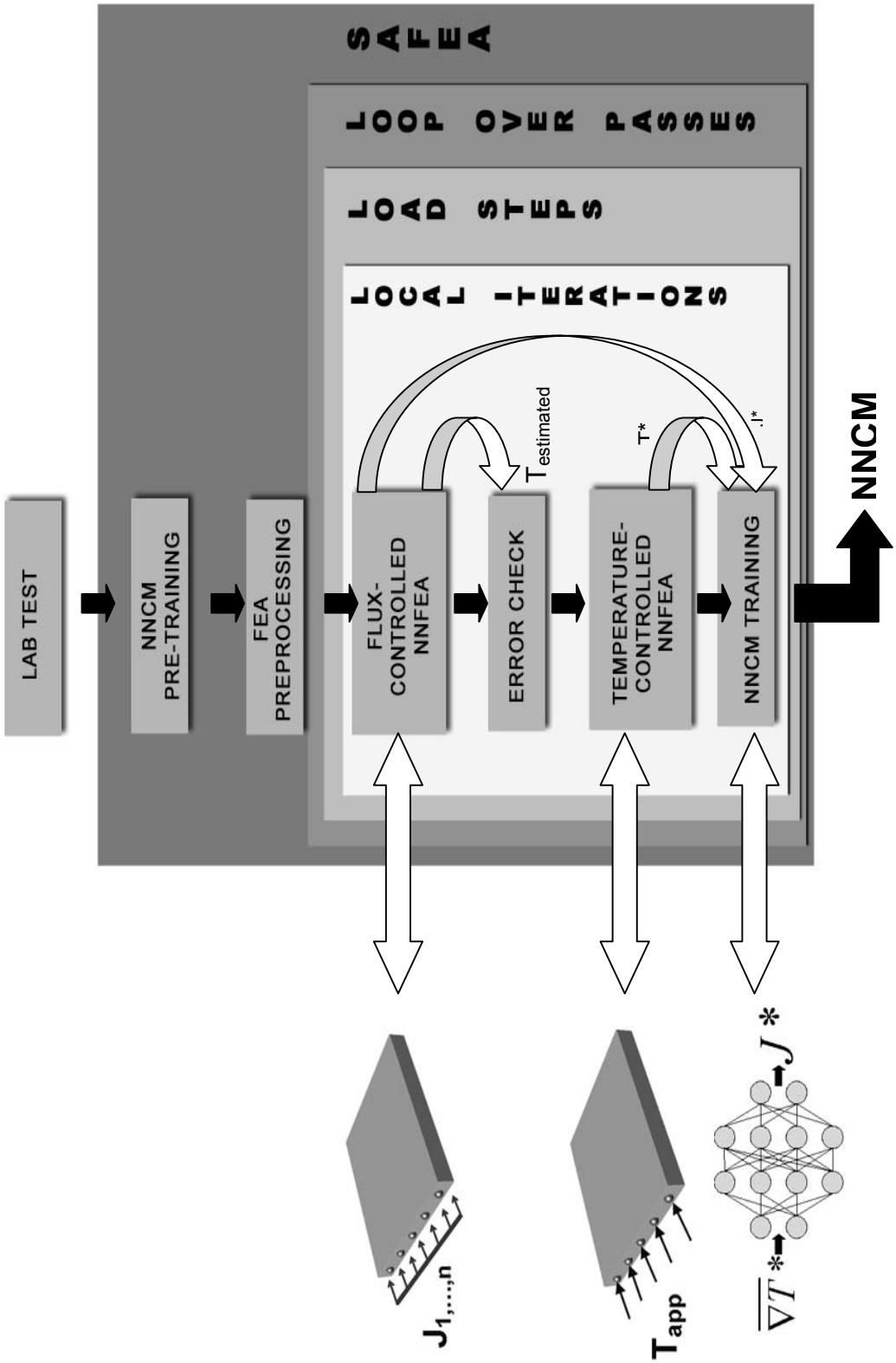
Next load step…

Next Pass…

Post-process results

Note that there are three levels of iteration presented here: *passes*, *load steps*, and *local iterations*. While these are the three loops of the SAFEA algorithm, the program also includes iterations in the nonlinear finite element analysis algorithm, training of the neural network, and in other secondary operations.

Figure 7.Schematic of the SAFEA algorithm.


SAFEA
LOOP OVER PASSES
LOAD STEPS
LOCAL ITERATIONS
LAB TEST
NNCM PRE-TRAINING
FEA PREPROCESSING
FLUX-CONTROLLED NNFEA
ERROR CHECK
TEMPERATURE-CONTROLLED NNFEA
NNCM TRAINING
NNCM
T estimated
T*
J*
J1,...,n
Tapp
J*
∇T*

3.4. The SAFEA Algorithm

After the physical test is carried out and the measurement data is fed to the SAFEA program, first a finite element representation of the setup is created. Afterwards, through the algorithm presented below, the program runs a series of finite element analyses and neural network trainings from which a constitutive model is created. The trained neural network is capable of replicating the experimental material response, and can potentially be used as a conventional constitutive model in other finite element analyses.

3.4.1. Pre-training the Neural Network

Due to the nature of neural networks, prior to using the neural network constitutive model in a nonlinear finite element analysis, it is required to train the neural network to learn an arbitrary linear material behavior. This avoids premature convergence problems in the nonlinear analysis.

On an error surface (Figure 8) with many local minima and only one global minimum, a neural network without pre-training may lie anywhere, but the pre-trained NN is placed in the neighborhood of the global minimum, and therefore if there is any convergence, it is will be toward a global minimum.

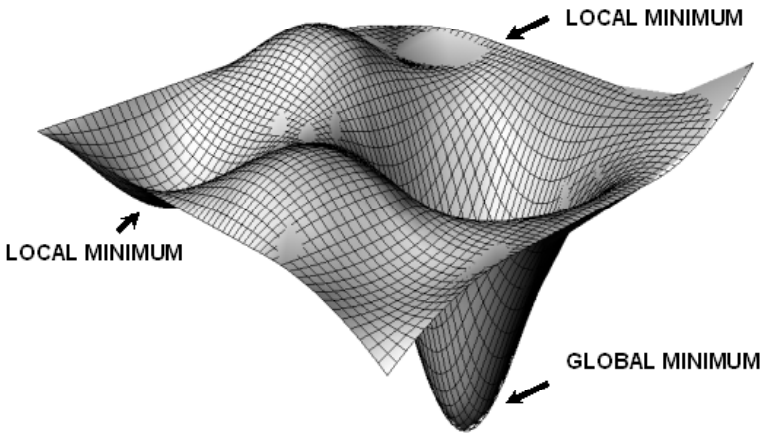


Figure 8. Error surface.

This behavior can be explained by understanding that the linear material response assumption is in the neighborhood – or otherwise a simplification – of the nonlinear response solution. A solution based on a linear neural network constitutive model is likely to be similar to the nonlinear response, especially if the material response were not driven far into the nonlinear range. This is precisely why it is helpful to begin the SAFEA iterations at lower load-step values, because a lower initial applied flux is not likely to drive temperatures into a highly nonlinear range.

An alternative view of this step is that the role of pre-training the neural network is to expose its weights to an approximate solution to the boundary value problem (BVP). This means that a constant value for conductivity is assumed. The value for conductivity can be taken as any commonly referenced constant material conductivity. As the algorithm progresses, the neural network decreasingly reflects and relies on this initially assumed conductivity. With the progression of the algorithm, the NN

is trained away from the linear assumption as it gradually learns the true nonlinear conductivity. This can be pictured as a solution that starts at the linear dashed line in Figure 9, and progresses toward the nonlinear solid line.

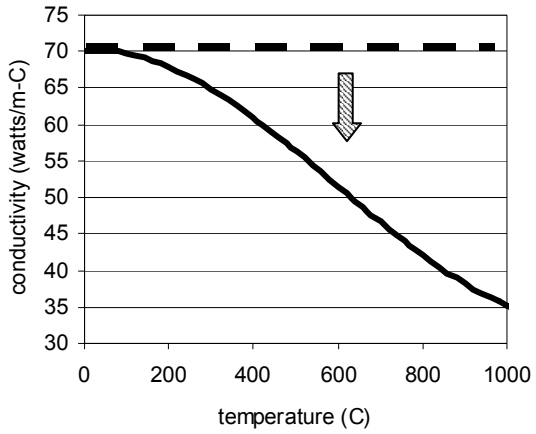


Figure 9. Linear (dashed line) and nonlinear (solid line) conductivity.

3.4.2. Parallel Finite Element Analyses

The most central operations in the SAFEA program are parallel FEA, using the Neural Network material model. The first is a flux-controlled analysis where, identical to the physical test, a flux boundary condition is applied. From this analysis, the temperatures at location(s) corresponding to the measurement points in the physical test are compared to the experimental measurements. If the error between these is sufficiently small, then the analysis for the current load step is complete and it is understood that the NNCM can sufficiently replicate material response at this load level. This is often true at low load steps where material behavior is linear, and thus the linearly pre-trained NNCM performs adequately in replicating that response.

If the error is not sufficiently small, a second analysis is carried out in which a correction is made to the neural network finite element analysis solution: this analysis is a temperature-controlled analysis where, unlike the physical test with an applied flux, a series of temperatures are applied to the boundary. These temperatures are a correction to the measurements from the experiment at the corresponding flux load step. By imposing the measured temperatures, a correction is made to the entire temperature and temperature gradient fields resulting from the second analysis. However slight the correction is, it is in the direction of the true solution. Thus the neural network is gradually trained to learn the behavior of the corrected system.

Prior to discussing the neural network training, a closer analysis of the mechanics, inputs, and outputs of the flux and temperature controlled analyses is given:

- In the flux-controlled analysis, a value of flux is input and after completion of the analysis, the resulting temperature field, gradient field, and internal fluxes that are in weak equilibrium with the applied flux are obtained. It is important to note that the flux field produced in this analysis is independent of the material model – it is only dependent on the applied flux. This means that the flux field solution is independent of the constitutive model used and as a result reflects the true flux distribution within the body. This flux field, $\overline{J}^*$, is recorded and further in the algorithm it is shown how $\overline{J}^*$ is used in training the NNCM.

- In the temperature-controlled analysis, the temperatures and gradients are improved over the domain. From this analysis, the temperature and gradient fields are corrected such that – relative to the temperature and gradient fields of the flux-controlled analysis – they more closely reflect the response from the lab test. This is accomplished by providing applied temperatures, at the locations of the measurement points, which are perturbed by a small magnitude in the direction of the correct solution. These applied temperatures, $T_{applied}$, comprise of those temperatures from the measurement points in the flux-controlled analysis, $T_{estimated}$ (obtained above), plus an increment of temperature equal to a fraction of the difference between $T_{estimated}$ and the true temperature at the measurement points, from the physical experiment, $T_{experimental}$:

$$\mathbf{T_{applied} = T_{estimated} + (T_{experimental} - T_{estimated}) / n}$$

Applied in temperature controlled FEA

From physical experiment

From flux controlled FEA

From this analysis, the temperature, *T*, and temperature gradients, $\overline{\nabla T}^{*}$, are recorded in order to be used in NNCM training.

3.4.3. Training the Neural Network Constitutive Model

Having obtained a true flux-field from the flux-controlled NNFEA, $\overline{J}*$, and a corrected temperature and temperature gradient fields from the temperature-controlled NNFEA, the NNCM is now trained to learn the relationship between heat fluxes, temperature gradient, and temperature. The values T, $\overline{\nabla T}*$ and $\overline{J}*$ assume the roles of neural network inputs and outputs in this training (Figure 10).

$$\overline{J}* = NN\{...\}\overline{\nabla T}*$$

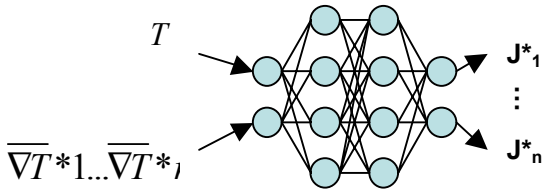


Figure 10. Inputs and outputs of NNCM training.

Noting that the temperature and temperature gradient values are corrected values, it is understood that the NNCM response will begin incrementally deviating from the pre-trained linear response toward the true nonlinear response, and as the SAFEA algorithm progresses to further load steps, the NNCM response continues moving toward a nonlinear solution.

CHAPTER 4

APPLICATION OF THE SELF ADAPTIVE FINITE ELEMENT ANALAYSIS TO A NONLINEAR STEADY-STATE HEAT TRANSFER PROBLEM

4.1. Introduction

This chapter will detail the procedures of the Self-Adaptive Finite Element Analysis method by using it to determine the constitutive model of a hypothetically unknown material. If the SAFEA method produces a neural network constitutive model that accurately matches the constitutive model of the material specimen, then the program is successful.

In order to determine if the neural network constitutive model is accurate and precise, a hypothetical material specimen will be used with a predetermined constitutive model. Instead of performing a physical lab test, for the purpose of validation and verification, the hypothetical lab test is created by a conventional finite element analysis with a custom constitutive model, in order to replicate what would happen in a lab. By using this FEA substitute, it is possible to provide an exact constitutive model with which to test the validity of the SAFEA produced neural network constitutive model.

The example that will be used is a nonlinear, steady-state heat transfer problem. The constitutive quantity under consideration is the material's conductivity, K. For a linear problem, K is a constant, but in this nonlinear problem, K is represented by a higher order equation. Here, K is a function of temperature only and is represented by the equation:

$$K = (5 \cdot 10^{8})T^{3} - (9 \cdot 10^{-5})T^{2} + (5.3 \cdot 10^{-3})T + 70.07$$

Plotting this relation yields the curve in Figure 11. The goal of the SAFEA method is to produce a neural network constitutive model that will output the same curve. Once that is accomplished, that NNCM can be used in any FEA to accurately represent material response. Recall that in practice, the NNCM is derived from a physical test on a material for which the constitutive model is poorly understood or not known at all.

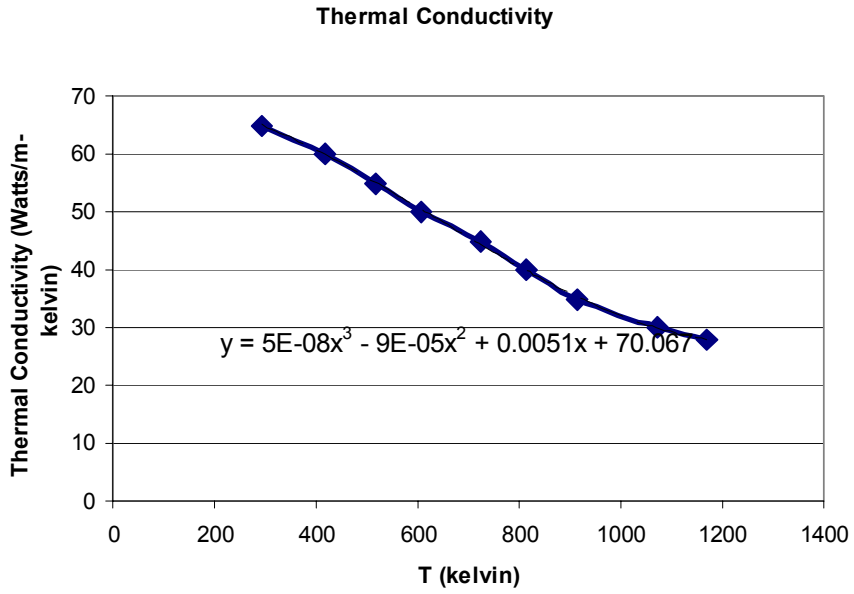


Figure 11. Specified thermal conductivity of steel plate specimen.

4.2. Problem Description

The physical problem in this example involves a homogeneous plate composed of a form of steel with the conductivity relation described above. The plate is a 20cmx20cm square, with a thickness of 1cm. The boundary conditions and geometry are summarized in Figure 12; they include fixed temperature values of 100K along the east, south, and west boundaries, and an applied flux at the north boundary. The flux is applied in four load steps:

2.5MW/m^2, 4.0MW/m^2, 5.5MW/m^2, and 7.0MW/m^2; meaning that first, a flux of 2.5MW/m^2 is applied at the northern boundary (with all other boundaries fixed at a temperature of 100K), equilibrium is achieved then temperature readings at the measurement points are taken. Next, a flux of 4.0MW/m^{2}is applied (again, with all other boundaries at 100K), equilibrium is reached and measurements are taken, and so on through the other two load steps.

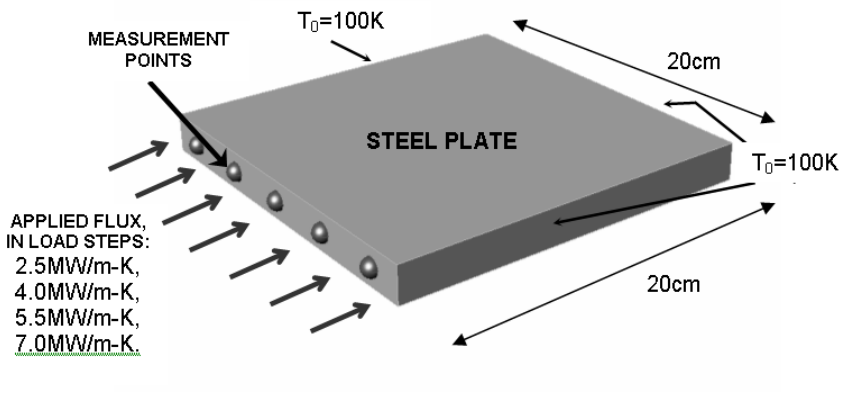


Figure 12. Schematic of specimen setup.

After equilibrium is attained in each flux load step and temperatures are read at measurement points along the northern boundary, then the temperature data is linked with the applied flux value and stored for use in the SAFEA program both in checking the error of the flux-controlled NNFEA and in obtaining input values for the temperature-controlled NNFEA. These steps are detailed in the following sections.

4.3. Applying the SAFEA

Once the test is performed and measurements are taken, the procedures of the SAFEA begin. In this section a detailed account of each step is given, including important heuristics inherent in the method.

4.3.1. Pre-training and Preprocessing

First the NNCM is pre-trained. In this example, our material is a form of steel, therefore an assumed conductivity of K=70W/m-K is used (which is a common value for steel). This constant conductivity will pre-train the neural network constitutive model to behave linearly.

Pre-training involves:

- Defining NN architecture
- Selecting a vector of inputs, [temperature, *T*, and temperature gradient, ∇T], and a vector of outputs, [flux, *J*], that obey the equation $J = K \cdot \nabla T$, where K=70MW/m-K, or more accurately, a diagonal matrix with 70MW/m-K on the diagonals.
- Pre-train the NN to learn the linear constitutive model.
- Storing the pre-trained NN for use as the NNCM in the parallel FEA of the first load step, in the first pass.
- After the first iteration of the first pass, the pre-trained NNCM will no longer be used because the NN will have evolved to capture a higher order equation for K.

Following NN pre-training, the program reads specimen geometry and boundary conditions (input by the user to match the physical test setup). The geometry is then meshed.

4.3.2. Parallel FEA and NNCM training

After NNCM pre-training and preprocessing, the SAFEA iterations begin, following the sequence below:

- The first pass begins, commencing with the first load step of $J=2.5MW/m^2$.
- A flux-controlled NNFEA is performed using an applied flux along the northern boundary of $2.5MW/m^2$. This analysis uses the linearly pre-trained NNCM and thus yields a linear response. The resulting flux field, $\bar{J}*$ (independent of constitutive model), is recorded for use in NNCM training.
- The temperatures obtained in this analysis are compared to temperatures measured at the same applied-flux value in the lab test. The steady-state temperatures at those nodes are compared against the measured values and a percent difference error is calculated. If the error is satisfactory, the second load step begins. However, if the error is not small enough, the program proceeds to the temperature-controlled NNFEA.
- In the temperature-controlled NNFEA, corrected temperature values are applied to the nodes of the northern boundary. These corrected temperatures equal the temperatures from the flux-controlled analysis, $T_{estimated}$ (which are incorrect but reflect a flux field that is in weak equilibrium with the applied fluxes), plus a small perturbation equal to a fraction of the difference between the measured temperature values, $T_{measured}$, and the $T_{estimated}$ values. Applying these corrected temperatures drives the resulting temperature gradient field closer to the true solution field. This incremental change in gradient is sufficient

to train the NN to move toward a nonlinear response; therefore in this analysis the temperature, *T*, and the gradient field, $\overline{\nabla T}^*$, are recorded for use in NNCM training.

- In training the NNCM, $\overline{J}^*$ and $\overline{\nabla T}^*$ act as NN outputs and inputs respectively. The NNCM is trained to capture a relationship that involves true fluxes and increasingly nonlinear temperature gradients. Note: It is important for proper training that a rich field of data be collected from the physical test. Recalling the nature of NN training, the broader the training data, the broader the ability of the NN to recognize relationships between inputs and outputs.
- At this point, in order to monitor the progress of NNCM training, a forward analysis is run and conductivity is plotted against temperature in order to obtain a curve similar to that of Figure 11. Prior to NNCM training, the pre-trained NNCM will result in a horizontal line, corresponding to the constant value of K, similar to the horizontal line in Figure 9. After NNCM training, that line is expected to gravitate toward the specified nonlinear relation of the material. In this case, where a hypothetical material with a known conductivity relation exists, the NNCM conductivity development is plotted against the known conductivity as will be seen in Figure 13. The goal then is to have the NNCM generated curve match the specified conductivity curve.
- The next step is to return to a second local iteration, at the same load step. Everything in subsequent local iterations is the same except the NNCM, which becomes better in each iteration. Once the number of local iterations reaches a predetermined maximum or the error after

the flux-controlled FEA is satisfied, the local iterations of the next load step, 4.0MW/m^2, are commenced. In the higher load steps, the NNCM is gradually introduced to increasingly higher temperature fields – meaning increasingly nonlinear behavior.

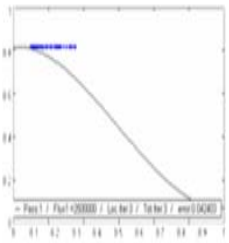
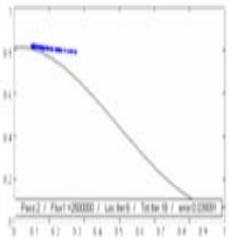
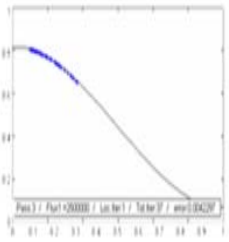
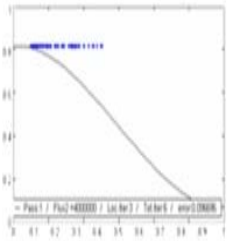
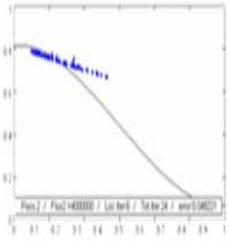
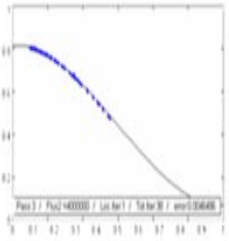
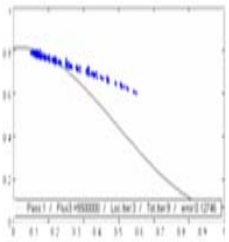
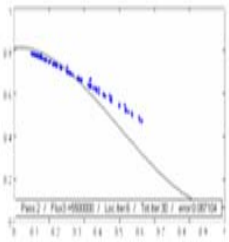
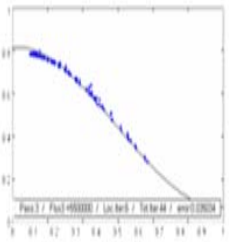
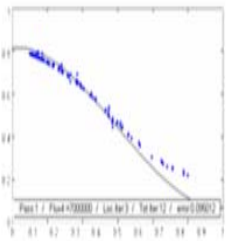
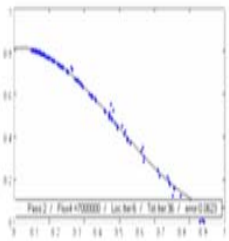
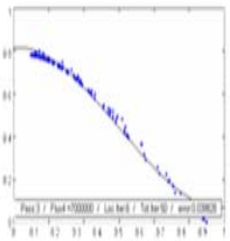
After all load steps have been processed, if the NNCM does not perform satisfactorily it might be necessary to run a second pass through the load steps. The benefit of subsequent passes lies solely in starting with a better NNCM, thus allowing for accelerated NN training behavior.

4.4. Results Analysis and Validation

4.4.1. SAFEA Results

The performance of the SAFEA algorithm in the above steps can be traced through the curves plotted after each NNCM training. Figure 13 shows the development of the NNCM conductivity through three passes containing the four load steps outlined above. The development is traced top to bottom (in one pass) and then left to right (over four load steps). The solid black curve represents the specified conductivity equation that is to be matched. The dots are NNCM outputs. An analysis of these figures follows.

Figure 13. SAFEA results over three passes and four load steps.

	Pass 1	Pass 2	Pass 3
Load Step 1			
Load Step 2			
Load Step 3			
Load Step 4			

Pass 1

Load Steps 1 and 2: the NNCM behaves linearly in reflection of the linear pre-training to a constant conductivity value.

Load Step 3: it is observed that the NNCM begins gravitating from a linear response toward a nonlinear response

Load Step 4: the NNCM correctly models conductivity at lower temperatures, but is not capable of matching conductivity at higher temperatures where there is a stronger nonlinearity, necessitating a second pass.

Pass 2

Load Steps 1, 2, and 3: the NNCM still behaves somewhat linearly – exposing the fact that training has yet to be perfected. This behavior can be explained by the NNCM having been trained exclusively to higher temperature response in the final load step of the previous pass.

Load Step 4: the NNCM replicates the nonlinear conductivity equation well. A final load pass is still required in order to insure proper behavior at early temperatures.

Pass 3

Load Steps 1, 2, 3, and 4: In all load steps, the NNCM has accurately and quite precisely evolved to match the specified conductivity equation.

At this point the Self Adaptive Finite Element Analysis is complete. A Neural Network Constitutive Model has been generated that closely captures the nonlinear material model. This NNCM is finally stored as an object that can be used in other analyses involving the material under investigation.

4.5. Error Behavior

An important component in the implementation of the SAFEA algorithm is monitoring the error behavior. There are two errors monitored in the analysis. The first error reflects the precision to which the NN is trained to meet the supplied target outputs; this is the *NN training error*. The second error is the *NNFEA error*, which reflects how well, relative to our experimental data, the NNCM is able to capture the temperatures at the measurement nodes. This NNFEA error is the error that is monitored in the step between the flux-controlled NNFEA and the temperature-controlled NNFEA, whereas the NN training error is monitored in the NN training step of the SAFEA algorithm.

The *NNFEA error* is observed to decrease steadily within one load step, but to increase once a new load step is reached. This is expected and attributed to the new information that the as yet untrained NN is exposed to in a new load step where higher temperatures are encountered. After all load steps are passed, and the next pass begins, similar error behavior is observed (that is, increased error at each load step, but decreasing error within local iterations within a given load step) but with lower magnitude. For clarification, Figure 14 gives a qualitative picture of the error behavior over the three passes and four load steps per pass of the sample problem

.

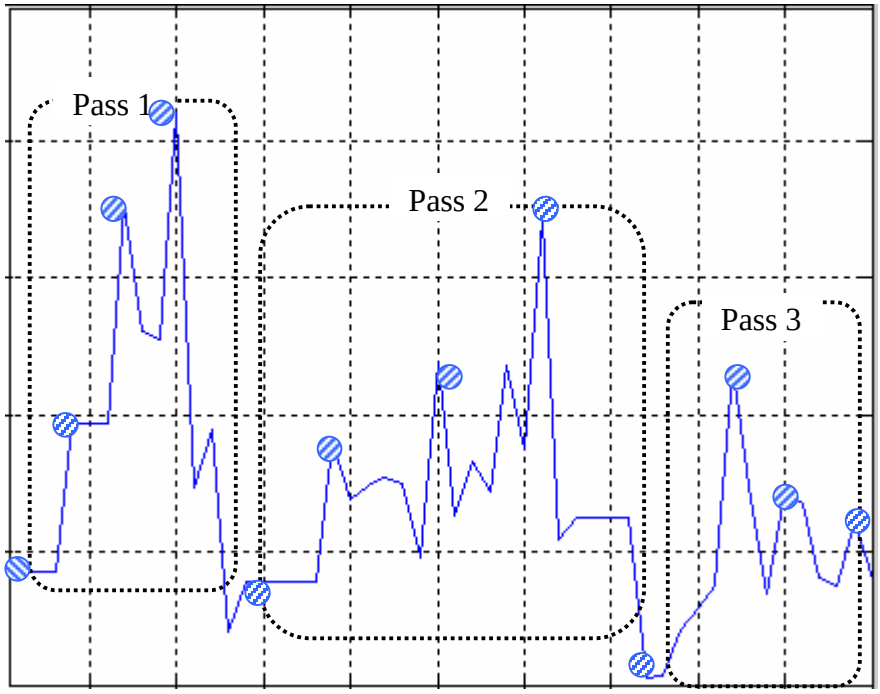


Figure 14. Schematic of error behavior over three SAFEA passes.

4.6. Validation

To verify the SAFEA produced neural network constitutive model, a finite element analysis with the obtained NNCM is performed and compared to the results of a finite element analysis with the prescribed hypothetical constitutive model. Validation follows if the results from both analyses match closely. Using a new set of boundary conditions and a different geometry proves that the NNCM understands the constitutive behavior of the material and not merely the input-output relationship of a single boundary value problem.

To demonstrate how the NNCM based FEA results match the results of the predefined FEA, the temperatures are plotted for each node of the

finite element mesh. Figure 15 shows results from three finite element analyses for each node of the finite element mesh: the linear constitutive model, the neural network constitutive model, and the specified hypothetical constitutive model. In this figure, the triangles represent the nonlinear FEA results of the specified conductivity and the circles represent the results from the FEA with the SAFEA produced NNCM. It is observed that the NNCM and hypothetical conductivity match well.

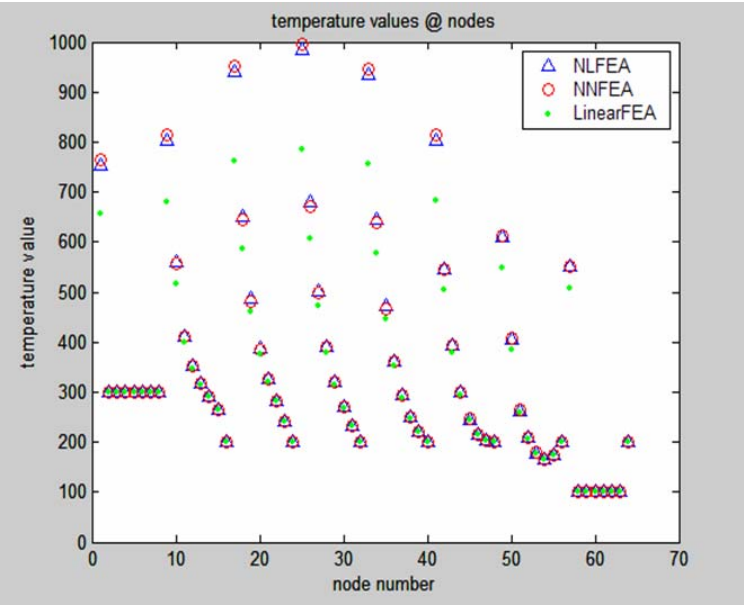


Figure 15. Temperature values at finite element nodes, for three constitutive models: linear, NNCM, and the hypothetical specified model.

Furthermore, in Figure 16, conductivity is plotted against temperature for each analysis, offering a visual for the degree of accuracy and precision with which the NNCM matches the specified conductivity definition.

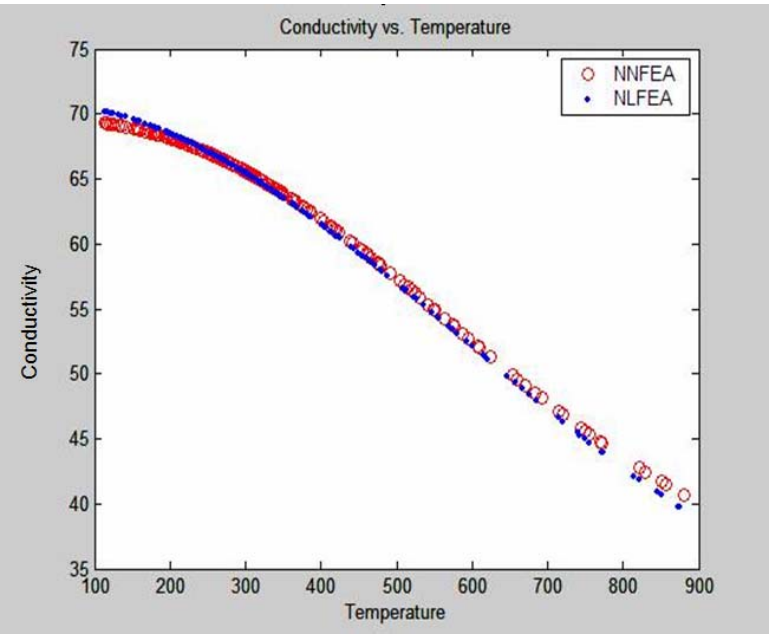


Figure 16. Comparison of the SAFEA based constitutive model with the true constitutive model.

4.7. Conclusion and Discussion

The performance of the SAFEA method in this steady-state, nonlinear heat transfer problem demonstrates the method's ability to capture a constitutive model from simple lab tests. More notably, the method demonstrates that the SAFEA can successfully be applied in inverse problems to determine nonlinear material behavior.

It is important to understand that this method does not merely obtain a response relationship for a given experiment. By focusing on matching the material's *constitutive relationship*, the SAFEA obtains a response relationship for any boundary value problem containing the studied material. For example with the information found in the problem above, any heat

transfer boundary value analysis involving the steel material can be carried out with accurate results. The NNCM obtained is not limited to reproducing behavior for problems with similar geometry and boundary conditions.

In using the SAFEA program it has become clear that certain heuristics play a central role in obtaining desired results. The most important of these heuristics are described below:

- *Load Step Values:* Initial load step values should fall within or very near to the linear response range; meaning that the first applied flux value should be low enough to generate temperatures that do not push into the non-constant conductivity range. This ensures that NNCM error remains in the neighborhood of the global minimum on the error surface. If the first load step produces highly nonlinear conductivity, the NNCM training will concentrate on highly nonlinear response without reinforcing the linear behavior at low flux values; as a result, in subsequent iterations the SAFEA error might grow highly unstable.
- *Allowable Number of Local Iterations:* The number of allowable local iterations should start small and increase with each pass. This step prevents the NN from over-training in earlier stages, when the NNCM is still inaccurate, and encourages more rigorous training in later passes when it is clear that the error is converging to a global minimum. Over-training to an undesired response will prevent the NNCM from learning the correct response in future iterations.
- *NN Training Error:* For the same reason as above, the NN training error should be relaxed in early passes and grow more stringent in

higher passes, again to prevent over-training the NNCM to an inaccurate response.

In general, most of the heuristics strive to increase stability in the SAFEA program. Ultimately, the nature of neural network training controls the stability of the program and the measures mentioned above either directly or indirectly attempt to moderate NN training. Offering attention to these heuristics will allow for a stable analysis such as in the example above.

In conclusion the SAFEA method circumvents the issue of extensive and elaborate laboratory testing toward attaining complex material behavior. By using intelligent tools to understand and generalize relationships between the different fields in a boundary value problem, this method enables a highly optimal use of straight forward lab test data to evolve a complete material model in the form of a Neural Network Constitutive Model, which can then be exported as an object for use in any analysis involving the tested material. Furthermore, the capability of the method in solving inverse problems opens the door to many other applications for which solutions are often overly expensive or simply impractical.

The procedures described herein can also be readily extended to model more complex nonlinear relationships or ultimately to capture coupled multiphysics processes such as coupled heat and mass transfer. With the SAFEA algorithm serving as a standing foundation. The method is an agile tool with broad potential.

APPENDIX A

PROGRAMMING THE SELF-ADAPTIVE FINITE ELEMENT ANALYSIS

A.1 Introduction

This section offers guidance for the end user of the SAFEA program. In order to make the program more user-friendly, a graphical user interface (GUI) was developed. Using the GUI is mostly self explanatory or otherwise documented for the user when the GUI is run. With that in mind, this section points out some aspects of the program that are important to be aware of.

The complete MATLAB code for the SAFEA program, including brief descriptions for each subroutine, is included in Appendix C.

A.2 GUI Fields

The SAFEA GUI is shown in Figure 17. Required inputs include FEA parameters, SAFEA (or "AUTOP") parameters, and NN training parameters. Below are some important considerations for the GUI user:

- Anytime a new meshing scheme is being used, or new experimental data is entered, the check box marked "Is this a new mesh seed?" must be checked in order to generate newly formatted NN training data sets.
- Under the SAFEA parameters, both error tolerance and a maximum number of iterations must be specified. The "Tolerances" field is a vector with a single tolerance entry for each SAFEA pass. The "maximum local iterations" field is also a vector with the same number of entries.

- Under NN training parameters, the user has the option to select the method of NN data transfer as any of “current”, “window”, or “all” options. The “current” option only saves training data from the most recent iteration thereby resulting in a faster analysis. The “window” option saves training data from a specified number of prior iterations given as the “window size”. Finally, the “all” option saves training data from all iterations resulting in the most comprehensive database but also the slowest running program. In the work done for this research, it was found that the “current” option provided best results.

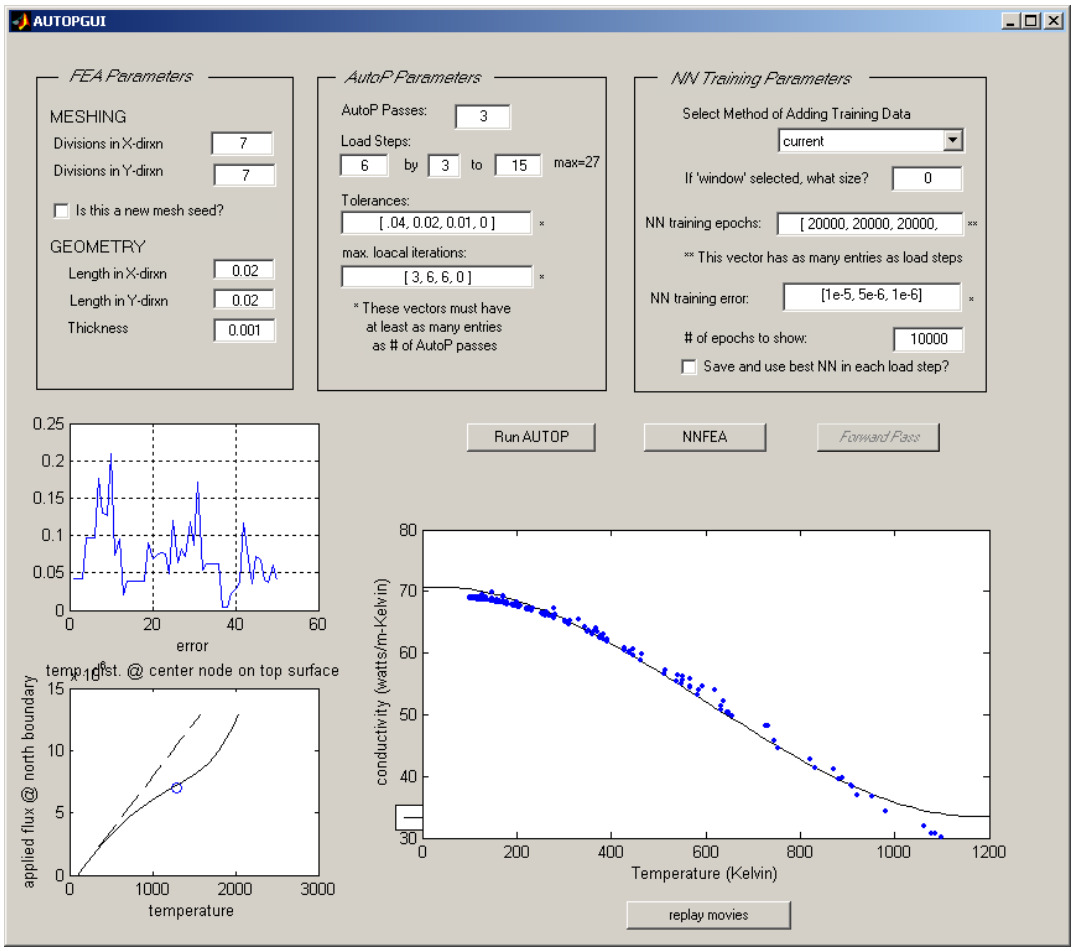


Figure 17. The SAFEA graphical user interface with the outputs graphed in three windows.

A.3 Outputs

After the user inputs all parameters, and runs the program, results begin plotting. Real time plots of the thermal conductivity curve, the accuracy of the predicted temperature fields at one sample measurement node, and the SAFEA error are all output in real time. After the program is complete, the user has the option of running a forward analysis that uses the most

updated NN generated by the SAFEA program in order to see how well the NNCM predicts the test behavior.

APPENDIX B

PROGRAMMING THE FINITE ELEMENT ANALYSIS

B.1 Introduction

The research presented in this thesis necessitated the development of a custom finite element analysis program that could be freely and extensively manipulated. This section offers validation for the finite element analysis program that was developed.

B.2 Verification

Closed-Form Verification of Nonlinear Solution for One-Dimensional Problem:

First, the simple one dimensional problem was investigated. The physical problem is as shown in Figure 18. With consistency in units, the calculations and analyses yield identical results.

Figure 18. Sample problem boundary conditions.

Closed Form Solution:
The conductivity obeys the following relation:

$$\kappa(T) = a_0 + a_1 T$$

where $a_0 = 77.52$ and $a_1 = -4.44 \cdot 10^{-2}$

$$\int \kappa dT = \int c_1 dx$$

$$a_o T + \frac{1}{2} a_1 T^2 + c_2 = c_1 x + c_3$$

$$\frac{1}{2} a_1 T^2 + a_0 T - (c\ x + c_3) = 0$$

The boundary conditions state that:

$$T(0) = T_1 = 10 \quad \text{and} \quad T(L) = T_2 = 10{,}000$$

Enforcing the BC's, we obtain:

$$c_1 = -144.56 \cdot 10^4 \cdot \frac{1}{L} \qquad \text{and} \qquad c_3 = 772.98$$

Solving the differential equation and incorporating coefficients c_1 and c_2, at the center of the bar we have:

$$T\left(\frac{L}{2}\right) = 7709.9$$

The same problem solved with the FEA code has the solution shown in Figure 19 where the temperature at mid-length is 7710.0 – an accurate and relatively precise approximation.

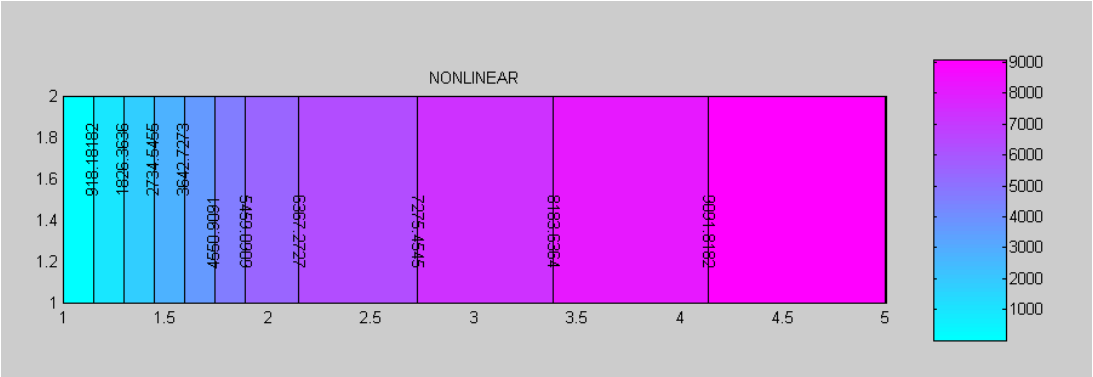


*Figure 19. Sample problem solution. The results are as follows: T_e2 = 10^4 * [0.001, 0.001, 0.6138, 0.6138, 0.7710, 0.7710, 0.8947, 0.8947, 1.0, 1.0].*

Comparative Verification of Nonlinear Solution for Two-Dimensional Problem:

For more complex geometries, a finite element analysis was performed using well known commercial software (ANSYS). Comparisons were performed for two problems:

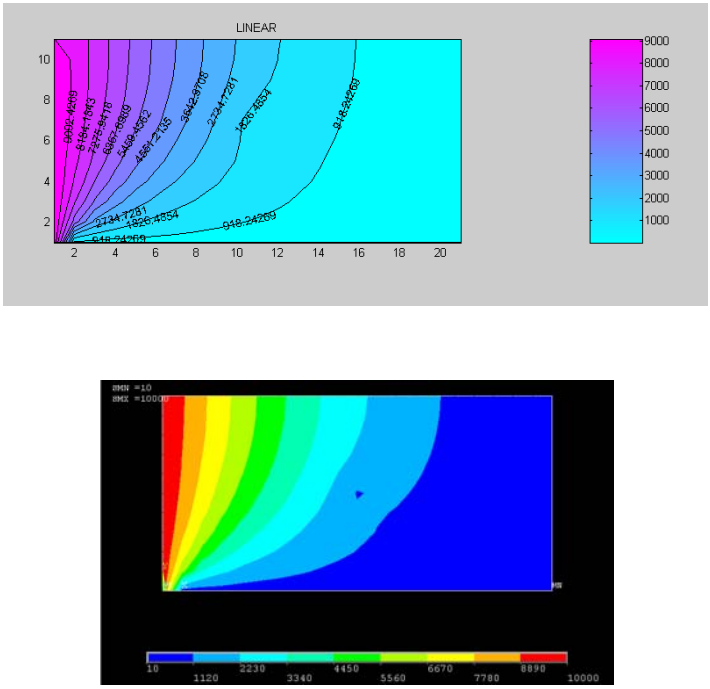


Figure 20. Graphical comparison of results from (top) programmed FEA and (bottom) commercial FEA program (ANSYS ©).

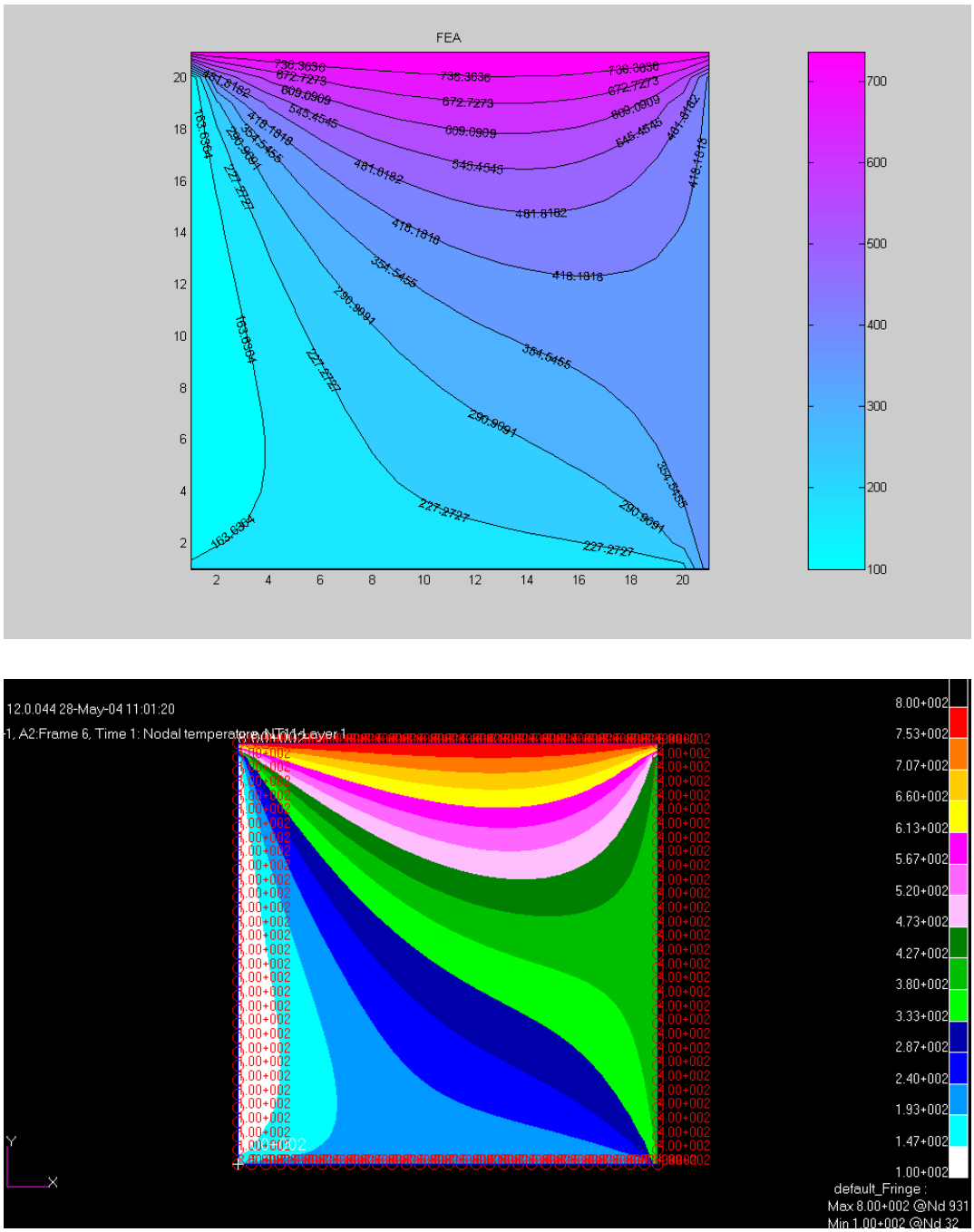


Figure 21. Graphical comparison of results from (top) programmed FEA and (bottom) commercial FEA program (ANSYS ©).

The figures show that the custom developed finite element analysis program successfully recreates commercially available finite element analysis software results.

APPENDIX C

MATLAB CODE AND DESCRIPTIONS

Subroutine Descriptions:

MATLAB Code:

add_data.m

General Description
add_data.m will take vectorized NN training data and add it to a spreadsheet file such as “trainingset.wk1.” There are three ways in which to add new data to pre-existing training data. The first method is to append data to the original file; that is, without overwriting the original training data but merely adding to it. The benefit of this method is a large body of training data with values that span a large space – hence a wider spectrum for the NN to draw from. However this poses a drawback as well – that is the NN will have a data set that is not particularly all good data. Another drawback is the size of the file.

Inputs and Outputs
The inputs are T, DT, and flux. These are supplied from the autop.m program. T and dT are vectors containing temperature and temperature gradient distributions from the Run2 analysis that involved the input of several measured temperatures in addition to the boundary conditions. The flux vector contains flux distributions from the Run1 analysis that involved only the applied boundary conditions without the additional measured points.

appliedflux.m

General Description
This function converts temperature or flux data (hence the name is not entirely appropriate for it operates on temperature as well as flux) from a user input to a vector of values per node of the body being analyzed.

Inputs and Outputs
The function takes as inputs either the temperature or flux values along one of the four edges of the plate and outputs these scalar values as an applied temperature or applied flux vector with entries corresponding to the nodes of the mesh.

applytemp.m

General Description
This function converts temperature or flux data (hence the name is not entirely appropriate for it operates on temperature as well as flux) from a user input to a vector of values per node of the body being analyzed.

Inputs and Outputs
The function takes as inputs either the temperature or flux values along one of the four edges of the plate and outputs these scalar values as an applied temperature or applied flux vector with entries corresponding to the nodes of the mesh.

aquinomethod.m

General Description
This function is the platform for finding the nonlinear finite element solution to our problem. The direct method of nonlinear iteration is used (described in chapter on nonlinear solution). Note that in this nonlinear solution, the neural network is used as the constitutive model.

Inputs and Outputs
The function inputs applied fluxes (fb), a vector of the temperature distribution from the linear solution (T_e1) as a first approximation, the temperature vector of applied temperatures (T), and the current neural network object. The outputs are the solution to the nonlinear problem, which is comprised of vectors representing temperature distribution (T1), directional temperature gradient distribution (gradT), and directional flux distribution (fluxdirxn) over the elements.

aquinomethodNN.m

General Description
This function is the platform for finding the nonlinear finite element solution to our problem. The direct method of nonlinear iteration is used (described in chapter on nonlinear solution). Note that in this nonlinear solution, the neural network is used as the constitutive model.

Inputs and Outputs
The function inputs applied fluxes (fb), a vector of the temperature distribution from the linear solution (T_e1) as a first approximation, the temperature vector of applied temperatures (T), and the current neural network object. The outputs are the solution to the nonlinear problem, which is comprised of vectors representing temperature distribution (T1), directional temperature gradient distribution (gradT), and directional flux distribution (fluxdirxn) over the elements.

Explanation of Code
Lines 9 through 12 are a set of basic initializations for the loop and vectors. There is one major loop in the direct convergence method. This do loop terminates once a tolerance or maximum iteration is reached. Before the first run of the loop, an assumed temperature field is initialized as the field obtained from the linear solution (line 12). Within the loop, the linear type problem is solved again for nonlinear conductivities (a function of temperature) corresponding to the temperature values obtained from the linear solution (line 12). The difference between the two results is obtained as “err” in line 19. Then, the original temperature distribution is reset to the

latest temperature distribution. And we iterate. Once the loop is completed, in lines 24 through 26 the error convergence is plotted. This is a very important plot to observe anytime the program is run, because in some conditions, the direct method will not converge and it important to make sure in all solutions that there is indeed convergence.

Discussion on Modifications
Other convergence schemes, such as the Newton-Raphson technique were attempted unsuccessfully. This method does not take much time to converge, and thus it was deemed a suitable alternative.

autop.m

General Description
Autop is the hub program for this project and it is the origin and destination of all subroutines and functions. It runs the Autoprogressive algorithm from pre-training a

Inputs and Outputs
There are no inputs to the program. Outputs include plots and temperature, flux, and gradient distributions.

AUTOPGUI.m

General Description
Autop is the hub program for this project and it is the origin and destination of all subroutines and functions. It runs the Autoprogressive algorithm from pre-training a

Inputs and Outputs
There are no inputs to the program. Outputs include plots and temperature, flux, and gradient distributions.

BC.m

General Description
This function is allows the user to choose the set of boundary conditions to use in the problem.

Inputs and Outputs
The only input is a flag that determines if the applied BC's are to only include user defined (typically edge) conditions, or if the applied BC's will also

include several point loads that are determined by measured data points in a previous analysis – this is the case in Run2 from in the Autoprogressive algorithm (described in chapter on the Autoprogressive algorithm). The function outputs two vectors that contain the temperature and flux values at all the nodes

Explanation of Code
In lines 13 through 16 and 26 through 29, temperatures and fluxes are chosen respectively along the north, east, south, and west edges of the plate. In lines 17 and 30, these values are vectorized. Lines 18 through 22 allow the program or user to define temperatures at nodes on the surface (interior nodes) of the plate. Finally, lines 31 and 32 cause flux boundary conditions to overwrite temperature boundary conditions. This is done because….

Discussion on Modifications
The programmer may elect not to have fluxes override temperatures. In that case, lines 31 and 32 should be discarded.

boundary.m

General Description
This function modifies the conductivity matrix in the finite element solution such that prescribed temperatures and fluxes (boundary conditions) are taken into account.

Inputs and Outputs
This subroutine reads the unmodified conductivity matrix, applied temperature, and applied flux vectors.

frwdpass.m

General Description
This subroutine functions as a verification of the resulting NN from a completed SAFEA run. It takes the final NN and runs a NNFEA, comparing results with an assumed model (i.e. lab experiment).

geometry.m

General Description
In this subroutine, the user defined geometry and mesh characteristics are transformed into matrices and vectors that the remainder of the FE program can efficiently read.

Inputs and Outputs
Plate dimensions and mesh specifications are input to the function in order to retrieve two vectors: the node coordinate vector and the element connectivity vector.

Gettestdata.m

General Description
In order to run the SAFEA problem in this research, hypothetical lab test data had to be generated. Further, this data had to correctly match FE mesh points with test measurement locations. This subroutine generates the data within the required format for any mesh size.

Inputs and Outputs
The locations of the measurement points are input in order to obtain nonlinear and linear test data

Indicator.m

General Description
This function assigns vector that indicates which nodes of the mesh have applied boundary conditions.

KAPPA.m

General Description
This function houses the hypothetical nonlinear (or linear) functional relationship between temperature and conductivity.

Kelement3.m

General Description
This function obtains the element conductivity matrix (un-modified for boundary conditions) for the linear FE solution.

Kelement3modified.m

General Description
This function obtains the element conductivity matrix (un-modified for boundary conditions) for the nonlinear FE solution. Note also that this function pertains to the non-NN solution.

Kelement3modifiedNN.m

General Description
This function obtains the element conductivity matrix (un-modified for boundary conditions) for the nonlinear NN-FE solution.

Kglobal.m

General Description
This function organizes the element conductivity matrices compiled in the linear element conductivity subroutine above, into a global conductivity matrix for the entire body.

Inputs and Outputs
This function outputs both the global conductivity matrix (un-modified for boundary conditions) as well as the global flux vector.

Kglobalmodified.m

General Description
This function organizes the element conductivity matrices compiled in the either of the non-linear element conductivity subroutines above, into a global conductivity matrix for the entire body.

Inputs and Outputs
This function outputs both the global conductivity matrix (un-modified for boundary conditions) as well as the global flux vector.

KglobalmodifiedNN.m

General Description
This function is the same as Kglobalmodified.m, except that it works for the NNFEA.

linearFEA.m

General Description
This function performs a simple linear steady state FEA for purposes of solution comparison and as a first iteration in the nonlinear iterations performed in the more complex FEA solution.

Master.m

General Description
This function is the platform for the regular nonlinear FEA. It calls all pertinent functions in order to obtain the field solution to the given BVP.

Inputs and Outputs
This function outputs vectors containing values that make up the field solutions to the user defined finite element problem.

Maxandmins.m

General Description
This function identifies and reads maximum and minimum values of training data that is fed to the NN. These values are essential for normalizing the NN input and output values.

NLFEA.m

General Description
This script implements the NLFEA. This script was created for interfacing with the GUI.

NNFEA.m

General Description
This script implements the NNFEA. This script was also created for interfacing with the GUI.

NNFEAGUI.m

General Description
This GUI implements a NNFEA problem.

Inputs and Outputs
The GUI reads geometry, boundary condition, and meshing values and returns field solutions and graphical contour plots, etc.

NNvsNL.m

General Description
This script generates a comparison of solutions from the NNFEA and the NLFEA. This is especially useful in verifying the results.

Plotgrid.m

General Description
This function assembles, both graphically and in matrix form, the finite element mesh that the user defines.

R.m

General Description
This function produces the flux vector used in the vector equation that is solved as part of the FE procedure.

Inputs and Outputs
This subroutine reads the global conductivity matrix, global flux, and applied temperature vectors and outputs an intermittent flux vector.

Resultplot.m

General Description
This subroutine is the major postprocessor for the FE programs.

Inputs and Outputs
The subroutine reads vectorized solutions and outputs contour plots.

Shape2.m

General Description
This function computes the shape functions, their derivatives, and the Jacobian of the shape function matrix, which are the feature quantities of the FE method.

trainNN.m

General Description
This important function performs the training of the NN. It includes the settings that determine how the NN is trained including number of epochs, minimum error, minimum error gradient, and target error.

usermat.m

General Description
This central function interfaces the NN with the FE program calculations. This function fits into integration point calculations in the FE analysis from where it calls the NNCM and substitutes those conductivity and fluxes into the parent FE program. This function also houses the most current NN as trained in the SAFEA program.

add_data.m

```
function [valuesmonitor,
datamonitor]=add_data(kappa_temp,dT,flux,micro_iter,macro_iter,i,lsa
,iter,adddata,windowsize,valuesmonitor, datamonitor,pass);

%subroutine that selects a method for adding training data for the
NN
%created on: 6/7/2004 by Shawhin Roudbari
%last updated: 6/14/2004

global L nodes ndx ndy

if adddata==1      %add CURRENT data
%     'adding current'
      [values] = wk1read('ts1');
    valuesmonitor{iter,1}=values';
      L=length(values);
      T_nonlin=values(1:L,1);
      gradT=values(1:L,2:3);             %assign data from
spreadsheet to specific variables...
      fluxdirxn=values(1:L,5:6);         %these vectors/matrices
have dimensions of nx1 (for T) or nx2 (grad,flux)
      kappa=values(1:L,4);
      %update training data
      data(:,1)=[T_nonlin; kappa_temp(:,3)];
      data(:,2:3)=[gradT; dT];
      data(:,4)=[kappa; kappa_temp(:,4)];
      data(:,5:6)=[fluxdirxn; flux];
    datamonitor{iter,1}=data';
      wk1write('ts2',data);


elseif adddata==2  %add WINDOW of data
    [values1]=wk1read('ts1');
      [values2]=wk1read('ts2');
%     pass
%     macro_iter
%     size_ts2=size(values2)
    if macro_iter==1
        !del ts2.wk1
    end
      if micro_iter==1 & iter==1      %read training data file
        L=length(values1);
      elseif micro_iter==1 & iter~=1 & pass==1
        L=length(values2);
      end
      if macro_iter>windowsize & micro_iter==1
%         'case1'
%         skip_to=length(values1)+(4*ndx*ndy+1)
%         up_to=length(values2)
        T_nonlin=[values1(1:length(values1),1) ;
values2(length(values1)+(4*ndx*ndy+1):length(values2),1)];
```

```
        gradT=[values1(1:length(values1),2:3) ;
values2(length(values1)+(4*ndx*ndy+1):length(values2),2:3)];
%assign data from spreadsheet to specific variables...
        fluxdirxn=[values1(1:length(values1),5:6) ;
values2(length(values1)+(4*ndx*ndy+1):length(values2),5:6)];
%these vectors/matrices have dimensions of nx1 (for T) or nx2
(grad,flux)
        kappa=[values1(1:length(values1),4) ;
values2(length(values1)+(4*ndx*ndy+1):length(values2),4)];

%L=length(values1)+length(values2(length(values1)+(4*ndx*ndy+1):leng
th(values2)));
        L=length(values2)-(4*ndx*ndy);
        valuesmonitor{iter,1}=[T_nonlin,gradT,kappa,fluxdirxn]';
      elseif iter==1
%         'case2'
%         L
        T_nonlin=values1(1:L,1);
            gradT=values1(1:L,2:3);            %assign data from
spreadsheet to specific variables...
            fluxdirxn=values1(1:L,5:6);        %these
vectors/matrices have dimensions of nx1 (for T) or nx2 (grad,flux)
            kappa=values1(1:L,4);
        valuesmonitor{iter,1}=[T_nonlin,gradT,kappa,fluxdirxn]';
      else
%         'case3'
%         L
        T_nonlin=values2(1:L,1);
            gradT=values2(1:L,2:3);            %assign data from
spreadsheet to specific variables...
            fluxdirxn=values2(1:L,5:6);        %these
vectors/matrices have dimensions of nx1 (for T) or nx2 (grad,flux)
            kappa=values2(1:L,4);
        valuesmonitor{iter,1}=[T_nonlin,gradT,kappa,fluxdirxn]';
      end
      %update training data
      data(:,1)=[T_nonlin; kappa_temp(:,3)];
      data(:,2:3)=[gradT; dT];
      data(:,4)=[kappa; kappa_temp(:,4)];
      data(:,5:6)=[fluxdirxn; flux];
    datamonitor{iter,1}=data';
      wk1write('ts2',data);
      % length(values1)
      % [values2]=wk1read('trainingset2');
      % length(values2)


elseif adddata==3  %add ALL data
%     'adding all'
      if macro_iter==1      %read training data file
        [values] = wk1read('ts1');
        !del ts2.wk1
      else
        [values] = wk1read('ts2');
      end
```

```
      if micro_iter==1
        L=length(values);
      end
    valuesmonitor{iter,1}=values';
      T_nonlin=values(1:L,1);
      gradT=values(1:L,2:3);            %assign data from
spreadsheet to specific variables...
      fluxdirxn=values(1:L,5:6);        %these vectors/matrices
have dimensions of nx1 (for T) or nx2 (grad,flux)
      kappa=values(1:L,4);
      %update training data
      data(:,1)=[T_nonlin; kappa_temp(:,3)];
      data(:,2:3)=[gradT; dT];
      data(:,4)=[kappa; kappa_temp(:,4)];
      data(:,5:6)=[fluxdirxn; flux];
    datamonitor{iter,1}=data';
      wk1write('ts2',data);
end
```

appliedflux.m

```
function [Rb]=appliedflux(fb)

%fucntion to obtain element flux vectors as effected by externally
applied
%boundary fluxes, fb. This function also globalizes the element
applied
%flux vectors
%created on: 6/1/2004 by Shawhin Roudbari
%last updated: 6/4/2004

global nodes connect lx ly ndx ndy thc
Rb=zeros(length(nodes),1);
coord=[-1/sqrt(3) 1/sqrt(3)];
weight=1;
count=0;
lxe=lx/ndx;
%length of element in x direction
lye=ly/ndy;
%length of element in y direction
for el=1:length(connect)
    fbel=fb(connect(el,:));
      Rbel=zeros(4,1);
      g1=nodes(connect(el,:),2)==0;
%element lies on north boundary
      g2=nodes(connect(el,:),2)==-ly;
%element lies on south boundary
    g3=nodes(connect(el,:),1)==0;
%element lies on west boundary
      g4=nodes(connect(el,:),1)==lx;
%element lies on east boundary
```

```
%     gboundary=g1|g2|g3|g4;
      if max(g1)==1
%north
        Rbel=[lxe*fbel(1); 0; 0; lxe*fbel(4)]*thc/2;
      elseif max(g2)==1
%south
        Rbel=[0; lxe*fbel(2); lxe*fbel(3); 0]*thc/2;
      elseif max(g3)==1
%west
        Rbel=[lye*fbel(1); lye*fbel(2); 0; 0]*thc/2;
      elseif max(g4)==1
%east
        Rbel=[0; 0; lye*fbel(3); lye*fbel(4)]*thc/2;
    end

    %the following section is NOT totally accurate if different
sides of
    %the element have different lengths
    if max(g1)==1 & max(g4)==1
%north-east corner
        Rbel=[lxe*fbel(1); 0; lye*fbel(3); lye*fbel(4)]*thc/2;
      elseif max(g1)==1 & max(g3)==1
%north-west corner
        Rbel=[lxe*fbel(1); lye*fbel(2); 0; lxe*fbel(4)]*thc/2;
      elseif max(g2)==1 & max(g4)==1
%south-east corner
        Rbel=[0; lxe*fbel(2); lxe*fbel(3); lye*fbel(4)]*thc/2;
      elseif max(g2)==1 & max(g3)==1
%south-west corner
        Rbel=[lye*fbel(1); lxe*fbel(2); lxe*fbel(3); 0]*thc/2;
    end
    if max(g1|g2|g3|g4)==1
        for k=1:4    %if the element stradles the body boundary
            Rb(connect(el,k),1)=Rb(connect(el,k),1)+Rbel(k);
        end
    end
end
```

applytemp.m

```
function [T]=applytemp(Tw,Ts,Te,Tn,flag)

%funciton to apply desired temperatures on one of four faces of a
%rectangular plate
%****NOTE*****: temperatures at corner nodes can be the sum of the
temperatures of
%the two faces they stradle.
%created on: 3/2/2004 by Shawhin Roudbari
%last updated: 3/12/2004

global nodes lx ly ndx ndy
T=zeros(length(nodes),1);

%assign temps at faces that were indicated
```

```
T(1:ndy+1)=Tw;                          %west face (including edges)
T(ndy+1:ndy+1:length(nodes))=Ts;        %south face
T(ndx*(ndy+1)+1:length(nodes))=Te;      %east face
T(1:ndy+1:length(nodes))=Tn;            %north face

if flag==1      % sum corner nodes that stradle two faces
      T(1)=Tw+Tn;
      T(1+ndy)=Tw+Ts;
      T(ndx*(ndy+1)+1)=Tn+Te;
      T((ndx+1)*(ndy+1))=Te+Ts;
else              %For corner nodes, if not summing applied
conditions, then choose max
      T(1)=max(Tw,Tn);
      T(1+ndy)=max(Tw,Ts);
      T(ndx*(ndy+1)+1)=max(Tn,Te);
      T((ndx+1)*(ndy+1))=max(Te,Ts);
end
```

aquinomethod.m

```
function [T1,gradT,fluxdirxn,kappa_temp]=aquinomethod(fb,To,T);

%nonlinear solver to substitue N-R method and also to use NN model
%created on: 3/31/2004 by shawhin roudbari
%last updated: 6/14/2004

i=1;
tolerance=0.0001;
maxiterations=30;
err(i)=1;
while (err(i) > tolerance) & (i<maxiterations)          %while error
is more than tolerance
    [kg,gradT,fluxdirxn,kappa_temp]=kglobalmodified(To);
%calculate global conductivity accouting for nonlinearity @ temp =
To
    [Rb]=appliedflux(fb);
    k_bc=boundary(kg,T,fb) ;                            %modify the
conductivity to account for BC's
    flux=R(T,k_bc,kg,Rb);                               %calculate
right side of KT=R equation (flux=R): use applied T in this
function, because only BC values of T are used and they should be
equall to those applied
    T1=k_bc\flux;                                       %solve for
new value of T vector
    err(i+1)=norm(T1-To);                               %error is
difference of the norms of initial and updated temperature vectors
    erri=err(i+1);
    i=i+1;
    To=T1;                                              %write over
previous To vector, to save space
end
if err(i)>1
    disp('WARNING -- poor convergence in nonlinear iterations; see
aquinomethod*.m');
```

```
end
% figure
% plot(err)                                                 %plot the
error convergence
% title('error plot')
```

aquinomethodNN.m

```
function
[T1,gradT,fluxdirxn,kappa_temp]=aquinomethodNN(fb,T_e1,T,net);

%same as aquinomethod.m, except this function calls
kglobalmodifiedNN.m,
%not Kglobalmodified.m
%nonlinear solver to substitue N-R method and also to use NN model
%created on: 3/31/2004 by shawhin roudbari
%last updated: 4/8/2004

i=1;
tolerance=0.001;
maxiterations=20;
To=T_e1;                                                  %solve
system for with initial K (already solved in master.m
err(i)=1;
while (err(i) > tolerance) & (i<maxiterations)          %while error
is more than tolerance
    [kg,Rb,gradT,fluxdirxn,kappa_temp]=kglobalmodifiedNN(To,fb,net);
%calculate global conductivity accouting for nonlinearity @ temp =
To
    [Rb]=appliedflux(fb);
    k_bc=boundary(kg,T,fb) ;                              %modify the
conductivity to account for BC's
    flux=R(T,k_bc,kg,Rb);                                 %calculate
right side of KT=R equation (flux=R): use applied T in this
function, because only BC values of T are used and they should be
equall to those applied
    T1=k_bc\flux;                                         %solve for
new value of T vector
    err(i+1)=norm(T1-To);                                 %error is
difference of the norms of initial and updated temperature vectors
    erri=err(i+1);
    i=i+1;
    To=T1;                                                %write over
previous To vector, to save space
end
if err(i)>1
    disp('WARNING -- poor convergence in nonlinear iterations; see
aquinomethodNN.m');
end
% figure
% plot(err)                                                 %plot the
error convergence
```

```
% title('error plot')
```

autop.m

```
%autop.m

%program that performs an autoprogressive analysis for nonlinear
heat transfer
%in a two dimensional plate, using NNFEA as the finite element
analysis tool.
%created on: 5/25/2004 by Shawhin Roudbari
%last updated: 6/16/2004

%information passed by the GUI
% passes = 1;
% lsa = 2;
% lsskip= 3;
% lsb = 28;
% tols =[0.001];
% cycles = [10];
% epochs = [10000,10000,10000,10000,10000];
% trainerror=[1e-6, 1e-6];
% showepoch =5000;
% adddata=3;
% savebest=0;
% newmesh=0;
% windowsize = 1;
% ndx = 7;
% ndy = 7;
% lx = 2;
% ly = 2;
% thc = 1;

tic
% global  loc ndx ndy nodes T_NLmeas lx ly thc %T_NL1  kappa_temp

%PRETRAIN ANN
load NN0;
maxandmins(1);

%GEOMETRY AND MESH
[nodes,connect]=geometry;

%EXPERIMENTAL DATA
loc=zeros(1,2*(ndy+1));
for i=1:(ndy+1);
    loc(2*i-1:2*i)=[(i-1)*(ndx+1)+1, (i-1)*(ndx+1)+2];
end
if newmesh==1
    [expdata,lindata]=gettestdata(loc);
else
    load expdata
```

```
    load lindata
end
fluxnorth=expdata(1,:);
dum=size(expdata);
experiments=expdata(2:dum(1),:);

%kappa plot info
Tplot=linspace(0,1200,100);
kappaplot= (5*10^-8)*Tplot.^3 - (9*10^-5)*Tplot.^2 + 0.0051*Tplot +
70.67*ones(length(Tplot),1)';

%AUTOPROGRESSIVE LOOP TO CONVERGENCE
iter=1;
valuesmonitor{iter,1}=0;
datamonitor{iter,1}=0;
for pass=1:passes
%     !del ts2.wk1
      macro_iter=1;
      %tols=[0.015 0.009 0.007 0.006];
    tolerance=tols(pass);
      %cycles=[4 4 4 4];
    maxiterations=cycles(pass);
      for i=lsa:lsskip:lsb
        clear temp_data
        T_NLmeas=experiments(:,i);
        fbn=fluxnorth(i);
%         fbn=0;
        Tn=0;
%         Tn=Tnorth(i);
        micro_iter=1;
        while micro_iter<=maxiterations

            %first analysis
            [T,fb]=BC(0, 100,100,100,Tn, 0,0,0,fbn);

[T_lin1,T_NL1,gradT_1,flux_1,kappa_temp1]=masterNN(T,fb,net,0);
            TNLsaver(:,iter)=[fluxnorth(i);i;T_NL1];
            Tlinsaver(:,iter)=[fluxnorth(i);i;T_lin1];
            KTsaver(:,4*iter-3:4*iter)=[fluxnorth(i) fluxnorth(i)
fluxnorth(i) fluxnorth(i);i i i i;kappa_temp1];

load_ctrl_data{iter,1}=[kappa_temp1(:,3),gradT_1,kappa_temp1(:,4),fl
ux_1]';

            %error of analysis vs. measurement
            dif=T_NL1(loc)-T_NLmeas;
            err=dif./T_NLmeas;
            error1(iter)=max(abs(err));
            me=error1(length(error1));
            avgerror(iter)=sum(dif.^2)/length(loc);
            ae=avgerror(length(error1));

            %plot error real time
            axes(handles.errplot1)
            plot(error1)
```

```
                xlabel('error')
                grid

            %plot kappa vs. T real time
            axes(handles.movie1)

      plot(Tplot,kappaplot,'k',kappa_temp1(:,3),kappa_temp1(:,4),'b.
')
            b=['legend (''Pass:' int2str(pass) '  /   Flux'
int2str(macro_iter) ' =' int2str(fbn)...
                    '  /   Loc.Iter:' int2str(micro_iter)...
                    '  /   Tot.Iter:' int2str(iter) '  /   error:'
num2str(me) ''',4)'];
                eval(b);
            ylabel ('conductivity (watts/m-Kelvin)');
xlabel('Temperature (Kelvin)')
            M(iter)=getframe;
            pause(.01)
%             set(h,'visible','off')

                %plot flux vs nodal temp for a given node real
time
            axes(handles.tempnode)
                targetnode=((ndy+1)*floor((ndx+1)/2))+1;

      plot(T_NL1(targetnode),fbn,'bo',expdata((length(loc)/2)+2,:),e
xpdata(1,:),'k-', lindata((length(loc)/2)+2,:),lindata(1,:),'k--');
            xlabel('temperature'); ylabel('applied flux @ north
boundary')
            title('temp. dist. @ center node on top surface');
            MT(iter)=getframe;
            pause(.01)

%             figure;
%
      plot(Tplot,kappaplot,'k',kappa_temp1(:,3),kappa_temp1(:,2),'b.
')
%             b=['legend (''Pass:' int2str(pass) '  /   Flux'
int2str(macro_iter) ' =' int2str(fbn)...
%                     '  /   Loc.Iter:' int2str(micro_iter)...
%                     '  /   Tot.Iter:' int2str(iter) '  /
error:' num2str(me) ''',4)'];
%                 eval(b);
%             ylabel ('conductivity - 2 - (watts/m-Kelvin)');
xlabel('Temperature (Kelvin)')
%             title ('[2, 2] of kappa diagonals');
%             pause(.01)
%
%             figure;
%
      plot(Tplot,kappaplot,'k',kappa_temp1(:,3),kappa_temp1(:,1),'b.
')
%             b=['legend (''Pass:' int2str(pass) '  /   Flux'
int2str(macro_iter) ' =' int2str(fbn)...
%                     '  /   Loc.Iter:' int2str(micro_iter)...
```

```
%                     '  /   Tot.Iter:' int2str(iter) '  /
error:' num2str(me) ''',4)'];
%                eval(b);
%             ylabel ('conductivity - 2 - (watts/m-Kelvin)');
xlabel('Temperature (Kelvin)')
%             title ('[1, 1] of kappa diagonals');
%             pause(.01)

            %break loop if...
            if error1(iter) <= tolerance
                temp_data (:,(6*micro_iter)-5:6*micro_iter) =
[kappa_temp1(:,3), gradT_1, kappa_temp1(:,4), flux_1];
                [valuesmonitor,
datamonitor]=add_data(kappa_temp1,gradT_1,flux_1,micro_iter,macro_it
er,i,lsa,iter,adddata,windowsize,valuesmonitor, datamonitor,pass);
                a=['save NNa' int2str(pass) int2str(i)
int2str(micro_iter) '_' int2str(iter) ' net'];
                      eval(a);
                monitor{iter,1}=wk1read('ts2');
                micro_iter=micro_iter+1;
                iter=iter+1;
                break
            end

            %second analysis
            [T,fb]=BC(1, 100,100,100,0, 0,0,0,0);

[T_lin2,T_NL2,gradT_2,flux_2,kappa_temp2]=masterNN(T,fb,net,0);

temp_ctrl_data{iter,1}=[kappa_temp2(:,3),gradT_2,kappa_temp2(:,4),fl
ux_1]';

            %add data to training set and train NN
            [valuesmonitor,
datamonitor]=add_data(kappa_temp2,gradT_2,flux_1,micro_iter,macro_it
er,i,lsa,iter,adddata,windowsize,valuesmonitor, datamonitor,pass);

      net=trainNN(1,iter,net,epochs(macro_iter),showepoch,trainerror
(pass));
            save 'NNet' net;

            %other
            temp_data (:,(6*micro_iter)-5:6*micro_iter) =
[kappa_temp2(:,3), gradT_2, kappa_temp2(:,4), flux_1];
            a=['save NNa' int2str(pass) int2str(i)
int2str(micro_iter) '_' int2str(iter) ' net'];
                  eval(a);
            monitor{iter,1}=wk1read('ts2');
            micro_iter=micro_iter+1;
            iter=iter+1;
        end %local iterations
        if savebest==1
            minerr=min(error1(iter-micro_iter+1 : iter-1)) ;
            pos=find(error1(iter-micro_iter+1 : iter-1)==minerr);
            pos_iter=pos+iter-micro_iter;
```

```
            a=['load NNa' int2str(pass) int2str(i) int2str(pos) '_'
int2str(pos_iter)];
            eval(a);
            a=['save bestNNa' int2str(pass) int2str(i) int2str(pos)
'_' int2str(pos_iter) ' net'] ;
            eval(a);
            getbest=temp_data(:,(6*pos)-5:6*pos);
            [valuesmonitor, datamonitor]=add_data([
zeros(length(getbest(:,4))),zeros(length(getbest(:,4))),
getbest(:,1),getbest(:,4)],getbest(:,2:3),getbest(:,5:6),micro_iter,
macro_iter,i,lsa,iter,adddata,windowsize,valuesmonitor,
datamonitor,pass);
         end
         macro_iter=macro_iter+1;
       end %load steps
end %passes
me=error1;
% ae=avgerror
monitor'
save movie M;
save movietemp MT;
save valuesmonitor valuesmonitor;
save datamonitor datamonitor;
save me me;
save temp_ctrl_data temp_ctrl_data;
save load_ctrl_data load_ctrl_data;
toc
```

AUTOPGUI.m

```
function varargout = AUTOPGUI(varargin)
% AUTOPGUI M-file for AUTOPGUI.fig
%      AUTOPGUI, by itself, creates a new AUTOPGUI or raises the
existing
%      singleton*.
%
%      H = AUTOPGUI returns the handle to a new AUTOPGUI or the
handle to
%      the existing singleton*.
%
%      AUTOPGUI('CALLBACK',hObject,eventData,handles,...) calls the
local
%      function named CALLBACK in AUTOPGUI.M with the given input
arguments.
%
%      AUTOPGUI('Property','Value',...) creates a new AUTOPGUI or
raises the
%      existing singleton*.  Starting from the left, property value
pairs are
%      applied to the GUI before AUTOPGUI_OpeningFunction gets
called.  An
```

```
%      unrecognized property name or invalid value makes property
application
%      stop.  All inputs are passed to AUTOPGUI_OpeningFcn via
varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows
only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help AUTOPGUI

% Last Modified by GUIDE v2.5 16-Jun-2004 14:55:44

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @AUTOPGUI_OpeningFcn, ...
                   'gui_OutputFcn',  @AUTOPGUI_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin & isstr(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before AUTOPGUI is made visible.
function AUTOPGUI_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to AUTOPGUI (see VARARGIN)

% Choose default command line output for AUTOPGUI
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes AUTOPGUI wait for user response (see UIRESUME)
% uiwait(handles.figure1);
clear all

% --- Outputs from this function are returned to the command line.
function varargout = AUTOPGUI_OutputFcn(hObject, eventdata, handles)
```

```
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes during object creation, after setting all properties.
function pass_CreateFcn(hObject, eventdata, handles)
% hObject    handle to pass (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function pass_Callback(hObject, eventdata, handles)
% hObject    handle to pass (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of pass as text
%        str2double(get(hObject,'String')) returns contents of pass
as a double
pass=str2double(get(hObject,'String'));
handles.pass=pass;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function lsA_CreateFcn(hObject, eventdata, handles)
% hObject    handle to lsA (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
```

```
end



function lsA_Callback(hObject, eventdata, handles)
% hObject    handle to lsA (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of lsA as text
%        str2double(get(hObject,'String')) returns contents of lsA
as a double
lsa=str2double(get(hObject,'String'));
handles.lsa=lsa;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function lsB_CreateFcn(hObject, eventdata, handles)
% hObject    handle to lsB (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function lsB_Callback(hObject, eventdata, handles)
% hObject    handle to lsB (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of lsB as text
%        str2double(get(hObject,'String')) returns contents of lsB
as a double
lsb=str2double(get(hObject,'String'));
handles.lsb=lsb;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function tols_CreateFcn(hObject, eventdata, handles)
% hObject    handle to tols (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
```

```
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function tols_Callback(hObject, eventdata, handles)
% hObject    handle to tols (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of tols as text
%        str2double(get(hObject,'String')) returns contents of tols
as a double
tols=str2num(get(hObject,'String'));
handles.tols=tols;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function cycles_CreateFcn(hObject, eventdata, handles)
% hObject    handle to cycles (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function cycles_Callback(hObject, eventdata, handles)
% hObject    handle to cycles (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of cycles as text
%        str2double(get(hObject,'String')) returns contents of
cycles as a double
cycles=str2num(get(hObject,'String'));
handles.cycles=cycles;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
```

```
function adddata_CreateFcn(hObject, eventdata, handles)
% hObject    handle to adddata (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end


% --- Executes on selection change in adddata.
function adddata_Callback(hObject, eventdata, handles)
% hObject    handle to adddata (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = get(hObject,'String') returns adddata contents
as cell array
%        contents{get(hObject,'Value')} returns selected item from
adddata

contents = get(hObject,'Value');
handles.adddata=contents;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function epochs_CreateFcn(hObject, eventdata, handles)
% hObject    handle to epochs (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function epochs_Callback(hObject, eventdata, handles)
% hObject    handle to epochs (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
% Hints: get(hObject,'String') returns contents of epochs as text
%        str2double(get(hObject,'String')) returns contents of
epochs as a double
epochs=str2num(get(hObject,'String'));
handles.epochs=epochs;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function showepoch_CreateFcn(hObject, eventdata, handles)
% hObject    handle to showepoch (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function showepoch_Callback(hObject, eventdata, handles)
% hObject    handle to showepoch (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of showepoch as text
%        str2double(get(hObject,'String')) returns contents of
showepoch as a double
showepoch=str2double(get(hObject,'String'));
handles.showepoch=showepoch;
guidata(hObject, handles);

% --- Executes on button press in savebest.
function savebest_Callback(hObject, eventdata, handles)
% hObject    handle to savebest (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of savebest
savebest=get(hObject,'Value');
handles.savebest=savebest;
guidata(hObject, handles);

% --- Executes on button press in autop.
function autop_Callback(hObject, eventdata, handles)
% hObject    handle to autop (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
global  loc ndx ndy nodes T_NLmeas lx ly thc
passes = handles.pass;
lsa = handles.lsa;
lsskip= handles.lsskip;
lsb = handles.lsb;
tols = handles.tols;
cycles = handles.cycles;
epochs = handles.epochs;
trainerror = handles.trainerror;
showepoch = handles.showepoch;
adddata=handles.adddata;
savebest=handles.savebest;
newmesh=handles.newmesh;
windowsize = handles.windowsize;
ndx = handles.ndx;
ndy = handles.ndy;
lx = handles.lx;
ly = handles.ly;
thc = handles.thc;
axes(handles.errplot1)
cla
axes(handles.movie1)
cla
axes(handles.tempnode)
cla

autop   %run autoP

figure  %plot error on seperate figure
plot(me)
xlabel('error')
grid

axes(handles.errplot1)  %plot error in GUI
plot(me)
xlabel('error')
grid

axes(handles.movie1)    %plot KvsT in GUI
movie(M,1,1)
handles.M=M;
guidata(hObject, handles);

axes(handles.tempnode)  %plot fbnvsT in GUI
movie(MT,1,1)
handles.MT=MT;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function popupmenu1_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called
```

```
% Hint: popupmenu controls usually have a white background on
Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end


% --- Executes on selection change in popupmenu1.
function popupmenu1_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = get(hObject,'String') returns popupmenu1
contents as cell array
%        contents{get(hObject,'Value')} returns selected item from
popupmenu1


% --- Executes on button press in pushbutton2.
function pushbutton2_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)


% --- Executes on button press in replay.
function replay_Callback(hObject, eventdata, handles)
% hObject    handle to replay (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
M = handles.M;
axes(handles.movie1)
movie(M,1,1)

MT = handles.MT;
axes(handles.tempnode)
movie(MT,1,2)

% --- Executes during object creation, after setting all properties.
function monitor_CreateFcn(hObject, eventdata, handles)
% hObject    handle to monitor (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
```

```
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end


% --- Executes on selection change in monitor.
function monitor_Callback(hObject, eventdata, handles)
% hObject    handle to monitor (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = get(hObject,'String') returns monitor contents
as cell array
%        contents{get(hObject,'Value')} returns selected item from
monitor

monitor = evalin('base','monitor')


% --- Executes during object creation, after setting all properties.
function windowsize_CreateFcn(hObject, eventdata, handles)
% hObject    handle to windowsize (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function windowsize_Callback(hObject, eventdata, handles)
% hObject    handle to windowsize (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of windowsize as
text
%        str2double(get(hObject,'String')) returns contents of
windowsize as a double
windowsize=str2double(get(hObject,'String'));
handles.windowsize=windowsize;
guidata(hObject, handles);


% --- Executes during object creation, after setting all properties.
```

```
function ls_skip_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ls_skip (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function ls_skip_Callback(hObject, eventdata, handles)
% hObject    handle to ls_skip (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of ls_skip as text
%        str2double(get(hObject,'String')) returns contents of
ls_skip as a double

lsskip=str2double(get(hObject,'String'));
handles.lsskip=lsskip;
guidata(hObject, handles);


% --- Executes during object creation, after setting all properties.
function ndx_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ndx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function ndx_Callback(hObject, eventdata, handles)
% hObject    handle to ndx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
% Hints: get(hObject,'String') returns contents of ndx as text
%        str2double(get(hObject,'String')) returns contents of ndx
as a double
ndx=str2double(get(hObject,'String'));
handles.ndx=ndx;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function ndy_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ndy (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function ndy_Callback(hObject, eventdata, handles)
% hObject    handle to ndy (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of ndy as text
%        str2double(get(hObject,'String')) returns contents of ndy
as a double
ndy=str2double(get(hObject,'String'));
handles.ndy=ndy;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function lx_CreateFcn(hObject, eventdata, handles)
% hObject    handle to lx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end
```

```
function lx_Callback(hObject, eventdata, handles)
% hObject    handle to lx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of lx as text
%        str2double(get(hObject,'String')) returns contents of lx as
a double
lx=str2double(get(hObject,'String'));
handles.lx=lx;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function ly_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ly (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function ly_Callback(hObject, eventdata, handles)
% hObject    handle to ly (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of ly as text
%        str2double(get(hObject,'String')) returns contents of ly as
a double
ly=str2double(get(hObject,'String'));
handles.ly=ly;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function thc_CreateFcn(hObject, eventdata, handles)
% hObject    handle to thc (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
```

```
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function thc_Callback(hObject, eventdata, handles)
% hObject    handle to thc (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of thc as text
%        str2double(get(hObject,'String')) returns contents of thc
as a double
thc=str2double(get(hObject,'String'));
handles.thc=thc;
guidata(hObject, handles);

% --- Executes on button press in newmesh.
function newmesh_Callback(hObject, eventdata, handles)
% hObject    handle to newmesh (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of newmesh
newmesh=get(hObject,'Value');
handles.newmesh=newmesh;
guidata(hObject, handles);


% --- Executes on button press in NNFEA.
function NNFEA_Callback(hObject, eventdata, handles)
% hObject    handle to NNFEA (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

NNFEAGUI


% --- Executes on button press in frwrdpss.
function frwrdpss_Callback(hObject, eventdata, handles)
% hObject    handle to frwrdpss (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global  loc ndx ndy nodes  lx ly thc
passes = handles.pass;
lsa = handles.lsa;
lsskip= handles.lsskip;
lsb = handles.lsb;
ndx = handles.ndx;
ndy = handles.ndy;
```

```
lx = handles.lx;
ly = handles.ly;
thc = handles.thc;
frwdpass


% --- Executes during object creation, after setting all properties.
function trainerror_CreateFcn(hObject, eventdata, handles)
% hObject    handle to trainerror (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function trainerror_Callback(hObject, eventdata, handles)
% hObject    handle to trainerror (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of trainerror as
text
%        str2double(get(hObject,'String')) returns contents of
trainerror as a double
trainerror=str2num(get(hObject,'String'));
handles.trainerror=trainerror;
guidata(hObject, handles);
```

BC.m

```
function [T,fb]=BC(flag, Tw,Ts,Te,Tn, fbw,fbs,fbe,fbn);

%function to choose boundary conditions
%created on: 4/10/2004 by Shawhin Roudbari
%last updated: 6/14/2004

global T_NLmeas loc

%BOUNDARY CONDITIONS
%---------------------
%indicate T=TEMPERATURE [C] boundary conditions at desired faces
```

```
%***nonlinearity in conductivity is not defined for Temperature
exceeding
%1000 degrees celcius
% Tw=100;                                    %temperature on west
face
% Ts=100;                                    %temp on south face
% Te=100;                                    %temp on east face
% Tn=Tn;                                    %temp on north face
T=applytemp(Tw,Ts,Te,Tn,0);
if flag==1
    for i=1:length(loc)
        T(loc(i))=T_NLmeas(i);            %apply some known temps
from analysis above
    end
end

%indicate fb=PRESCRIBED FLUX [Watts/m^2] boundary conditions at
desired faces
%   ***    NOTE: FLUX BC's will overwrite Temp. BC's    ***
% fbw=0;                                    %flux on west face
% fbs=0;                                    %...  on south face
% fbe=0;                                    %... on east face
% fbn=fbn;                                    %... on north face
fb=applytemp(fbw,fbs,fbe,fbn,1);
fbind=(fb==0);                            %have applied fb overwrite
applied T
T=T.*fbind;
```

Boundary.m

```
function [Kg_bc]=boundary(Kg,T,fb)

%function to impose boundary conditions
%created on: 2/23/2004 by Shawhin Roudbari
%updated: 3/12/2004

global nodes
I_temp=indicator(T);
I_flux=indicator(fb);
for i=1:length(nodes)
    if I_temp(i)==1                              %if there is a temp
applied @ node
        Kg(1:i-1,i)=0;                  %zero column, less the
diagonal term
        Kg(i+1:length(nodes),i)=0;
        Kg(i,1:i-1)=0;                  %zero row, less the diagonal
term
        Kg(i,i+1:length(nodes))=0;
    end
end
Kg_bc=Kg;
```

Frwdpass.m

```
%frwdpass.m

%runs a forward pass through all flux load leves with final NN from
autop
%created on: 6/15/2004 by Shawhin Roudbari
%last updated: 6/17/2004

% clear all
global ndx ndy lx ly thc
load NNa3154_48
maxandmins(1);
load expdata
fluxnorth=expdata(1,:);
load lindata
ndx=7; ndy=7; lx=.02; ly=.02; thc=1;
lsa=6; lsskip=3; lsb=15;
[nodes,connect]=geometry;
loc=zeros(1,2*(ndy+1));
Tplot=linspace(0,1500,100);
kappaplot= (5*10^-8)*Tplot.^3 - (9*10^-5)*Tplot.^2 + 0.0051*Tplot +
70.67*ones(length(Tplot),1)';
counter=1;
iter=1;
figure

for i=lsa:lsskip:lsb
    fbn=fluxnorth(i);
    Tn=0;
    [T,fb]=BC(0, 100,100,100,Tn, 0,0,0,fbn);
    [T_lin1,T_NL1,gradT_1,flux_1,kappa_temp1]=masterNN(T,fb,net,0);

    for j=1:ndx+1
        targetnode=((ndy+1)*(j-1))+1;
        subplot(((lsb-lsa)/lsskip+1),ndx+1,iter)

plot(T_NL1(targetnode),fbn,'bo',expdata(2*(j),:),expdata(1,:),'k-',
lindata(2*(j),:),lindata(1,:),'k--');
        axis([0 2000 0 15e4])
        if j==1
            b=['ylabel (''Flux=' int2str(fbn) ''')'];
                  eval(b);
        end
%         if j==ndx+1
%             xlabel('temperature');
%         end
%         if j==1
%             title('temp. dist. @ center node on top surface');
%             ylabel('applied flux @ north boundary')
%         end
%         MT(counter)=getframe;
        pause(.01)
```

```
        iter=iter+1;
    end
    hold on
    counter=counter+1;
end

figure
plot(Tplot,kappaplot,'k',kappa_temp1(:,3),kappa_temp1(:,2),'b.')

% figure
% movie(MT,1,1)
```

Geometery.m

```
function [nodes,connect]=geometry;

%program to input problem geometry (for a rectangular sample)
%and impose desired mesh
%inputs:
%lx=length of sample in x direction
%ly=length of sample in y direction
%ndx=number of element discretizations in x direction
%ndy=number of element discretizations in y direction
%created on: 2/3/2004 by Shawhin Roudbari
%last updated: 3/12/2004

global nodes connect lx ly ndx ndy

% divide element
dx=lx/ndx;                              %dx is element width in x
dy=ly/ndy;                              %dy is element length in y
direction

% number of elements and nodes (CCW starting from top left)
num_nodes=(ndx+1)*(ndy+1);              %total number of nodes
num_el=ndx*ndy;                         %number of finite elements

%create cell array (ndx*ndy) that containes node data for each cell
el_num=0;                 %element number counter
for i=1:ndx
    for j=1:ndy
        el_num=el_num+1;                %element number
        TL=(i-1)*(ndy+1)+j;             %top left node number
        TR=i*(ndy+1)+j;                 %top right node number
        BL=TL+1;                        %bottom left node number
        BR=TR+1;                        %bottom right node number
        elements(el_num,:)=[el_num, TL, BL, BR, TR];
    end
end
connect=elements(:,2:5);                %make element connectivity
matrix (CCW nodes starting from top left)
%figure
```

```
[nodes]=plotgrid(dx,dy,elements);
```

Gettestdata.m

```
function [expdata,lindata]=gettestdata(loc);

% function to generate hypothetical experimental data at specified
measurement points on the body (loc)
% for a given mesh and geometry
% created on: 6/11/2004 by Shawhin Roudbari
% last updated: 6/15/2004

fluxes=[0:500000:13000000];
expdata=zeros(length(loc)+1, length(fluxes));
for i=1:length(fluxes)
    [T,fb]=BC(0, 100,100,100,0, 0,0,0,fluxes(i));
    [T_lin,T_NL,gradT,fluxdirxn,kappa_temp]=master(T,fb,0);
    resultplot(T_NL,1)
    expdata(:,i)=[fluxes(i); T_NL(loc)];
    lindata(:,i)=[fluxes(i); T_lin(loc)];
%     '_____________________________'
%     appflux=fluxes(i)
%     maxgrad=max(max(gradT))
%     maxflux=max(max(fluxdirxn))
end
appflux=fluxes(i)
maxgrad=max(max(gradT))
maxflux=max(max(fluxdirxn))
save expdata expdata
save lindata lindata
```

Indicator.m

```
function [I]=indicator(T);

%function to create indicator vector signaling nodes with BC applied
%created on: 2/26/2004 by Shawhin Roudbari
%updated: 3/12/2004

global nodes
indicator=zeros(1,length(nodes));
for node=1:length(nodes)
        if T(node)~= 0
        indicator(node)=1;
    end
end
I=indicator;
```

KAPPA.m

```
function [kappa]=KAPPA(T);

global nodes
%kappa=(1./(T+ones(length(T),1))).^0.5;
%kappa= -0.0444*T + 77.52*ones(length(T),1);
kappa= (5*10^-8)*T.^3 - (9*10^-5)*T.^2 + 0.0051*T +
70.67*ones(length(T),1);
%kappa=70;
%kappa= -.1444*T + 77.52*ones(length(T),1);
```

Kelement3.m

```
function [Kel,Rbel]=Kelement3(el,fbel,kappa)

%master program that obtains K for a single element
%Created: 3/6/2004 by Shawhin Roudbari
%Last updated: 3/12/2004

global nodes connect lx ly ndx ndy thc
coord=[-1/sqrt(3) 1/sqrt(3)];           %xi and eta coordinates for
second order gauss quadrature
weight=1;                               %weight factor
x=nodes(connect(el,:),1);               %element nodal coordinates
y=nodes(connect(el,:),2);
Kel=zeros(4);
Rbel=zeros(4,1);
for i=1:2                               %loop over xi coordinates
    xi=coord(i);
    for j=1:2                           %loop over eta coordinates
        eta=-coord(j);                  %NOTE: nodes count CCW from
top left corner
        [J,B,N]=shape2(xi,eta,x,y);
        Kel=Kel + B'*kappa*B*thc*J*weight^2;

        %calculate Rb=HEAT FLUX VECTOR for prescribed boundary flux
        %for all elements that lie on the boundary of the body

      g=((nodes(connect(el,:),1))==0)|((nodes(connect(el,:),1))==lx)
...

|((nodes(connect(el,:),2))==0)|((nodes(connect(el,:),2))==-ly);
            if max(g)==1                    %if the element stradles
the body boundary
            sum1=N'*fbel(1)*thc*J*weight+N'*fbel(2)*thc*J*weight+...
                N'*fbel(3)*thc*J*weight+N'*fbel(4)*thc*J*weight;
            Rbel=Rbel+sum1;             %UNCERTAIN about using
scalar fb_element, fbel
        end
    end
```

```
end
```

Kelement3modified.m

```
function
[Kel,gradT_el,fluxdirxn_el,kappa_temp]=Kelement3modified(el,T_el);

%master program that obtains K for a single element. For use with
the NN
%material model version of NLFEA
%Created: 4/4/2004 by Shawhin Roudbari
%Last updated: 6/5/2004

global nodes connect  thc
coord=[-1/sqrt(3) 1/sqrt(3)];           %xi and eta coordinates for
second order gauss quadrature
weight=1;                               %weight factor
x=nodes(connect(el,:),1);               %element nodal coordinates
y=nodes(connect(el,:),2);
Kel=zeros(4);
nodenum=1;                              %set counter to be used in
assigning nodal kappa
T_ip=zeros(4,1);
kappa_temp=zeros(2,4);
for i=1:2                               %loop over xi coordinates
    xi=coord(i);
    for j=1:2                           %loop over eta coordinates
        eta=-coord(j);                  %NOTE: nodes count CCW from
top left corner
        [J,B,N]=shape2(xi,eta,x,y);
        T_ip(nodenum)=N*T_el;               %Temp at integration
point = (N)x(temp at element nodes)
        Bholder(2*nodenum-1:2*nodenum, :)=B; %store all B matrices
for use in calculating temp gradient, shown below
        kappa=KAPPA(T_ip(nodenum));             %kappa is a
function of temp @ integration points
        Kholder(nodenum)=kappa;                      %store all
kappa values to be used in flux calculations below
        Kel=Kel+B'*kappa*B*thc*J*weight^2;
        nodenum=nodenum+1;
    end
end
for j=1:4       %loop for creating gradT and fluxdirxn vectors; to
be passeed to Kglobalmodified
    gradT_el(:,j)=Bholder(2*j-1:2*j, :)*T_ip;
    fluxdirxn_el(:,j)=-Kholder(j)*gradT_el(:,j);
    kappa_temp(:,j)=[Kholder(j); T_ip(j)];  %2x4 matrix (top
row=kappa; bot row=temps)
end
```

Kelement3modifiedNN.m

```
function
[Kel,Rbel,gradT_el,fluxdirxn_el,kappa_temp]=Kelement3modifiedNN(el,f
bel,T_el,net)

%same as Kelement3modified.m with changes for ANN model:
%master program that obtains K for a single element. For use with
the NN
%material model version of NLFEA
%Created: 4/4/2004 by Shawhin Roudbari
%Last updated: 5/26/2004

global nodes connect lx ly ndx ndy thc %kappa_temp
coord=[-1/sqrt(3) 1/sqrt(3)];               %xi and eta coordinates
for second order gauss quadrature
weight=1;                                   %weight factor
x=nodes(connect(el,:),1);                   %element nodal
coordinates
y=nodes(connect(el,:),2);
Kel=zeros(4);
Rbel=zeros(4,1);
nodenum=1;                                  %set counter to be used
in assigning nodal kappa
T_ip=zeros(4,1);
kappa_temp=zeros(4,4);
for i=1:2                                   %loop over xi
coordinates
    xi=coord(i);
    for j=1:2                               %loop over eta
coordinates
        eta=-coord(j);                      %NOTE: nodes count CCW
from top left corner
        [J,B,N]=shape2(xi,eta,x,y);         %N[1x4]
        T_ip(nodenum)=N*T_el;               %Temp at integration
point = (N [1x4])x(temp at element nodes[4x1] = [1x1])
        Bholder(2*nodenum-1:2*nodenum, :)=B; %store all B matrices
for use in calculating temp gradient, shown below

g=((nodes(connect(el,:),1))==0)|((nodes(connect(el,:),1))==lx)|((nod
es(connect(el,:),2))==0)|((nodes(connect(el,:),2))==-ly);
            if max(g)==1                        %if the element
stradles the body boundary

sum1=N'*fbel(1)*thc*J*weight+N'*fbel(2)*thc*J*weight+N'*fbel(3)*thc*
J*weight+N'*fbel(4)*thc*J*weight;
            Rbel=Rbel+sum1;                 %Rb prescribed heat flux
vector for elements on boundary of body. UNCERTAIN about using
scalar fb_element, fbel
        end
        nodenum=nodenum+1;
    end
end
```

```
%warning off MATLAB:divideByZero            %this is for the error
that is temporarily patched in the usermat.m function call below
for k=1:4
    gradT_el(:,k)=Bholder(2*k-1:2*k, :)*T_ip;   %2x4 matrix
    [Jflux,kappa]=usermat(gradT_el(:,k),T_ip(k),net);
    %temporary1=[kappa(1,1) kappa(2,2) T_ip(k)]
%PRINT out used while programming
    Kel=Kel+Bholder(2*k-1:2*k, :)'*[kappa(1,1) 0; 0
kappa(2,2)]*Bholder(2*k-1:2*k, :)*thc*J*weight^2;
    Kholder(k)=mean([kappa(1,1) kappa(2,2)]);   %kholder gives mean
of directional conductivities (the same for isotropic material)
    Kholder1(k)=kappa(1,1);
    Kholder2(k)=kappa(2,2);
    Jxholder(k)=Jflux(1); %1x4
    Jyholder(k)=Jflux(2); %1x4
end
for j=1:4                                   %loop for creating gradT
and fluxdirxn vectors; to be passeed to KglobalmodifiedNN
    gradT_el(:,j)=Bholder(2*j-1:2*j, :)*T_ip;   %2x4 matrix
    fluxdirxn_el(:,j)=[Jxholder(j); Jyholder(j)];
%     fluxdirxn_el(:,j)=[-Kholder1(j)*gradT_el(1,j); -
Kholder2(j)*gradT_el(2,j)];    %2x4 matrix
    kappa_temp(:,j)=[Kholder1(j); Kholder2(j); T_ip(j); Kholder(j)];
%3x4 matrix (top row=flux; bot row=temps)
%    kappa_temp(:,j)=[Kholder(j); T_ip(j)];
end
```

Kglobal.m

```
function [Kg,Rb]=Kglobal(kappa,fb)

%was previously
[Kg]=Kglobal(lx,ly,ndx,ndy,nodes,connect,T,kappa,thc)
%program that obtains global K
%Created: 2/23/2004 by Shawhin Roudbari
%Last updated: 3/12/2004

global nodes connect lx ly
Kg=zeros(length(nodes));
%initialize conductivity matrix
Rb=zeros(length(nodes),1);
%initialize boundary flux vector
for el=1:length(connect)                                    %loop
over all elements
    fbel=fb(connect(el,:));
    [Kel,Rbel]=Kelement3(el,fbel,kappa);
%obtain element conductivity
    for i=1:4
        for j=1:4                                           %loops
over element nodes
            Kg(connect(el,i),connect(el,j))=...
                Kg(connect(el,i),connect(el,j))+Kel(i,j);
```

```
        end

g=((nodes(connect(el,:),1))==0)|((nodes(connect(el,:),1))==lx)...

|((nodes(connect(el,:),2))==0)|((nodes(connect(el,:),2))==-ly);
            if max(g)==1                                        %if
the element stradles the body boundary
            Rb(connect(el,i),1)=Rb(connect(el,i),1)+Rbel(i);
%calculate Rb=HEAT FLUX VECTOR for prescribed boundary flux
        end
    end
end
```

Kglobalmodified.m

```
function [Kg,gradT,fluxdirxn,kappa_temp]=Kglobalmodified(T)

%was previously
[Kg]=Kglobal(lx,ly,ndx,ndy,nodes,connect,T,kappa,thc)
%program that obtains global K for a vector kappa
%Created: 4/5/2004 by Shawhin Roudbari
%Last updated: 6/6/2004

global nodes connect
Kg=zeros(length(nodes));
%initialize conductivity matrix
gradT=zeros(4*length(connect),2);                   %initialize Temp
Gradient vectors in (nx2) matrix where column 1 is in x direction,
col 2 in y dirxn
fluxdirxn=zeros(4*length(connect),2);               %initialize
directional flux similar to gradT
kappa_temp=zeros(4*length(connect),2);
T_el=ones(4,1);                                     %set dimensions
of T_el
count=0;    %counter for globalizing gradT and fluxdirxn
for el=1:length(connect)                                    %loop
over all elements
    T_el=T(connect(el,:));                          %assign temp to
nodes of element (used for kappa in Kelement3modified.m)
    T_el(3)=T(connect(el,4));                       %Make
corrections to temperature assignment because of different...
    T_el(4)=T(connect(el,3));                       %...order of
node counting in Kelement3modified.m

[Kel,gradT_el,fluxdirxn_el,kappa_temp_el]=Kelement3modified(el,T_el)
;           %obtain element conductivity
    for i=1:4
        for j=1:4                                           %loops
over element nodes
            Kg(connect(el,i),connect(el,j))=...
%GLOBALIZE conductivity matrix
                Kg(connect(el,i),connect(el,j))+Kel(i,j);
```

```
        end
    end
    for m=1:4;              %4 IPs per element -- note, values of
gradT, etc are @ IP's, thus they will not be summed across elements
(as they would for nodes)
        gradT(4*count+m,:)=gradT(4*count+m,:)+gradT_el(:,m)';
%GLOBALIZE gradT matrix (two vectors, side by side)

fluxdirxn(4*count+m,:)=fluxdirxn(4*count+m,:)+fluxdirxn_el(:,m)';
%GLOBALIZE fluxdirxn matrix   "

kappa_temp(4*count+m,:)=kappa_temp(4*count+m,:)+kappa_temp_el(:,m)';
%globalize conductivity and temp values at ea. IP
      end
    count=count+1;
      %****NOTE: the arrays for gradT and fluxdirxn are interpreted
4 rows at a
      %time. The four rows represent the 4 IP's of one element
end
```

KglobalmodifiedNN.m

```
function
[Kg,Rb,gradT,fluxdirxn,kappa_temp]=KglobalmodifiedNN(T,fb,net)

%same as kglobalmodified.m except that this calls
Kelement3modifiedNN.m
%instead of Kelement3modified.m
%was previously
[Kg]=Kglobal(lx,ly,ndx,ndy,nodes,connect,T,kappa,thc)
%program that obtains global K for a vector kappa
%Created: 4/5/2004 by Shawhin Roudbari
%Last updated: 5/25/2004

global nodes connect lx ly
Kg=zeros(length(nodes));
%initialize conductivity matrix
Rb=zeros(length(nodes),1);
%initialize boundary flux vector
gradT=zeros(4*length(connect),2);
%initialize Temp Gradient vectors in (nx2) matrix where column 1 is
in x direction, col 2 in y dirxn
fluxdirxn=zeros(4*length(connect),2);
%initialize directional flux similar to gradT
fluxdirxn2=zeros(4*length(connect),2);
kappa_temp=zeros(4*length(connect),4);
%initialize nx2 matrix containing IP conductivities in the first
column and IP temps in the second
T_el=ones(4,1);                                          %set
dimensions of T_el
count=0;                                                 %counter
for globalizing gradT and fluxdirxn
```

```
for el=1:length(connect)                                        %loop
over all elements
    fbel=fb(connect(el,:));                                     %assign
flux to nodes of element in consideration
    T_el=T(connect(el,:));                                      %assign
temp to nodes of element (used for kappa in Kelement3modified.m)
    T_el(3)=T(connect(el,4));                                   %Make
corrections to temperature assignment because of different...
    T_el(4)=T(connect(el,3));
%...order of node counting in Kelement3modified.m

[Kel,Rbel,gradT_el,fluxdirxn_el,kappa_temp_el]=Kelement3modifiedNN(e
l,fbel,T_el,net);           %obtain element conductivity
    for i=1:4
        for j=1:4                                               %loops
over element nodes

Kg(connect(el,i),connect(el,j))=Kg(connect(el,i),connect(el,j))+Kel(
i,j);   %GLOBALIZE conductivity matrix
        end

g=((nodes(connect(el,:),1))==0)|((nodes(connect(el,:),1))==lx)|((nod
es(connect(el,:),2))==0)|((nodes(connect(el,:),2))==-ly);
%GLOBALIZE Rb vector
            if max(g)==1                                            %if
the element stradles the body boundary
            Rb(connect(el,i),1)=Rb(connect(el,i),1)+Rbel(i);
%calculate Rb=HEAT FLUX VECTOR for prescribed boundary flux
        end
    end
    for m=1:4;                                                  %4 IPs
per element -- note, values of gradT, etc are @ IP's, thus they will
not be summed across elements (as they would for nodes)
        gradT(4*count+m,:)=gradT(4*count+m,:)+gradT_el(:,m)';
%GLOBALIZE gradT matrix (two vectors, side by side)

fluxdirxn(4*count+m,:)=fluxdirxn(4*count+m,:)+fluxdirxn_el(:,m)';
%GLOBALIZE fluxdirxn matrix   "

kappa_temp(4*count+m,:)=kappa_temp(4*count+m,:)+kappa_temp_el(:,m)';
%globalize conductivity and temp values at ea. IP
    end
    count=count+1;
            %****NOTE: the arrays for gradT and fluxdirxn are
interpreted 4 rows at a
            %time. The four rows represent the 4 IP's of one element
end
```

linearFEA.m

```
function [Kg_bc,flux]=linearFEA(fb,T,kappa);

%linear steady state FEA
```

```
%reads flux from nonlinear iteration fuction (master.m)
%returns modigied global conductivity, Kg_bc, and flux vector
%created: 3/8/2004 by Shahwin Roudbari
%last updated: 6/4/2004

[Kg,Rb]=Kglobal(kappa,fb);                    %obtain global K
matrix and Rb (prescribed flux vector)
clear Rb
[Rb]=appliedflux(fb);
Kg_bc=boundary(Kg,T,fb);               %then modify K with BC's
implemented
flux=R(T,Kg_bc,Kg,Rb);                 %get load (flux) vector, R,
the right hand side of linear equation to be solved (K*T=R)
```

master.m

```
function
[T_e1,T_e2,gradT,fluxdirxn,kappa_temp]=master(T,fb,plotflag);

%Nonlinear Finite element analysis.
%created on: 4/10/2004 by Shawhin Roudbari
%last updated: 6/14/2004


global nodes
%ANALYSIS
%--------
%Linear solution
kappa=70.5;                         %linear kappa for initial
estimate of flux. Units: [W/m*C] (see pp 470 cook, 4th ed.)
[Kg_bc,flux]=linearFEA(fb,T,kappa);
T_e1=Kg_bc\flux;                        %solve for T

%Nonlinear solution
[T_e2,gradT,fluxdirxn,kappa_temp]=aquinomethod(fb,T_e1,T);

%POST PROCESSOR
%--------------
if plotflag==1
      resultplot(T_e1,1);                     %plot contour for
linear analysis
      title ('LINEAR');
      resultplot(T_e2,2);                     %plot contour for
nonlinear analysis
      title ('NONLINEAR');

      figure                                %compare temperature for
linear and nonlinear
      plot((1:length(nodes)),T_e1,(1:length(nodes)),T_e2,'r')
      legend ('linear approximation', 'nonlinear solution')
      title ('temperature values @ nodes')
      xlabel ('node number'); ylabel ('temperature value');
```

```
end
```

masterNN.m

```
function
[T_e1,T_e2,gradT,fluxdirxn,kappa_temp]=masterNN(T,fb,net,plotflag);

%Nonlinear Finite element analysis. Same as master, but uses NN as
material model
%created on: 4/10/2004 by Shawhin Roudbari
%last updated: 4/10/2004

global nodes

%ANALYSIS
%--------
%Linear solution
kappalinear=70.5;                          %linear kappa for initial
estimate of flux. Units: [W/m*C] (see pp 470 cook, 4th ed.)
[Kg_bc,flux]=linearFEA(fb,T,kappalinear);
T_e1=Kg_bc\flux;                          %solve for T

%Nonlinear solution
%T_e2=NNR(fb,T,flux);                       %call NR solver and supply
initial flux estimate (from linear kappa)
[T_e2,gradT,fluxdirxn,kappa_temp]=aquinomethodNN(fb,T_e1,T,net);


%POST PROCESSOR
%--------------
if plotflag==1
      resultplot(T_e1,1);                        %plot contour for
linear analysis
      title ('LINEAR');
      resultplot(T_e2,2);                        %plot contour for
nonlinear analysis
      title ('NONLINEAR');

      figure                                  %compare temperature for
linear and nonlinear
      plot((1:length(nodes)),T_e1,(1:length(nodes)),T_e2,'r')
      legend ('linear approximation', 'nonlinear solution')
      title ('temperature values @ nodes')
      xlabel ('node number'); ylabel ('temperature value');
end
```

maxandmins.m

```
function [inputn, Jn, dTdx, dTdy, Jx, Jy]=maxandmins(flag);
```

```
%function to read file of training data and obtain max and min
values for
%normalization and denormalization. All variables are set to global
%variables for use in the entire program
%created on: 5/11/2004 by Shawhin Roudbari
%last updated: 7/6/2004

global mxi mxj
if flag==1
    [values] = WK1READ('ts0');     %1 %read training data from wk1
file: trainingset1.wk1, created on excel
else
    [values] = WK1READ('ts0');     %2  %read training data from wk1
file: trainingset1.wk1, created on excel
end

size_of_data_read=size(values);

%global Jn minJ maxJ dTdxmin dTdxmax dTdymin dTdymax Tmin Tmax top
%[values] = WK1READ('trainingset0');      %read training data from
wk1 file: trainingset1.wk1, created on excel
gradT=values(1:length(values),2:3);             %assign data from
spreadsheet to specific variables...
fluxdirxn=values(1:length(values),5:6);         %these
vectors/matrices have dimensions of nx1 (for T) or nx2 (grad,flux)
T_nonlin=values(1:length(values),1);

%generate input data for training: T,x T,y and T (if nonlinear)
dTdx=gradT(:,1)';               %1xn vectors
dTdy=gradT(:,2)';
T=T_nonlin';
input=[dTdx;dTdy;T];            %3xn matrix of inputs, arranged in
rows

%generate ouput data: Jx and Jy
Jx=fluxdirxn(:,1)';             %1xn vectors
Jy=fluxdirxn(:,2)';
J=[Jx;Jy];                      %2xn matrix of targets with top
row=Jx and bottom row=Jy

%normalize values
% mxi=max(abs([max(input')',min(input')']))';
% mxi=1.25*mxi;
% mxi=[2300 ; 2300; 2500];
mxi=[100000; 100000; 10000];
inputn(1:3,:)=[input(1,:)/mxi(1); input(2,:)/mxi(2);
input(3,:)/mxi(3)];

% mxj=max(abs([max(J')',min(J')']))';
% mxj=1.25*mxj;
% mxj=[160000; 160000];
mxj=[2500000; 2500000];
Jn(1:2,:)=[J(1,:)/mxj(1); J(2,:)/mxj(2)];
```

NLFEA.m

```
%NLFEA.m

%script implements the NLFEA on an arbitrary (user
%defined) problem
%created on: 6/16/2004 by shawhin roudbari
%last updated: 6/16/2004

clear all
global ndx ndy lx ly thc
ndx=7; ndy=7; lx=.02; ly=.02; thc=1;
Tw=100; Te=300; Ts=400; Tn=0; fbn=5000000; fbs=0; fbe=0; fbw=0;

[nodes,connect]=geometry;

[T,fb]=BC(0, Tw,Ts,Te,Tn, fbw,fbs,fbe,fbn);
[T_lin1,T_NL1,gradT_1,flux_1,kappa_temp1]=master(T,fb,1);
```

NNFEA.m

```
%NNFEA.m

%script implements the result of autop training on an arbitrary
(user
%defined) problem
%created on: 6/14/2004 by shawhin roudbari
%last updated: 6/14/2004

clear all
global ndx ndy lx ly thc
load NNa2153_23
maxandmins(1);
ndx=7; ndy=7; lx=2; ly=2; thc=1;
Tw=100; Te=300; Ts=400; Tn=0; fbn=50000; fbs=0; fbe=0; fbw=0;

[nodes,connect]=geometry;

[T,fb]=BC(0, Tw,Ts,Te,Tn, fbw,fbs,fbe,fbn);
[T_lin1,T_NL1,gradT_1,flux_1,kappa_temp1]=masterNN(T,fb,net,1);
```

NNFEAGUI.m

```
function varargout = NNFEAGUI(varargin)
% NNFEAGUI M-file for NNFEAGUI.fig
%      NNFEAGUI, by itself, creates a new NNFEAGUI or raises the
existing
%      singleton*.
```

```
%
%      H = NNFEAGUI returns the handle to a new NNFEAGUI or the
handle to
%      the existing singleton*.
%
%      NNFEAGUI('CALLBACK',hObject,eventData,handles,...) calls the
local
%      function named CALLBACK in NNFEAGUI.M with the given input
arguments.
%
%      NNFEAGUI('Property','Value',...) creates a new NNFEAGUI or
raises the
%      existing singleton*.  Starting from the left, property value
pairs are
%      applied to the GUI before NNFEAGUI_OpeningFunction gets
called.  An
%      unrecognized property name or invalid value makes property
application
%      stop.  All inputs are passed to NNFEAGUI_OpeningFcn via
varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows
only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help NNFEAGUI

% Last Modified by GUIDE v2.5 16-Jun-2004 15:06:05

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @NNFEAGUI_OpeningFcn, ...
                   'gui_OutputFcn',  @NNFEAGUI_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin & isstr(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before NNFEAGUI is made visible.
function NNFEAGUI_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to NNFEAGUI (see VARARGIN)

% Choose default command line output for NNFEAGUI
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes NNFEAGUI wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command line.
function varargout = NNFEAGUI_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes during object creation, after setting all properties.
function ndx_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ndx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function ndx_Callback(hObject, eventdata, handles)
% hObject    handle to ndx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of ndx as text
%        str2double(get(hObject,'String')) returns contents of ndx
as a double
ndx=str2double(get(hObject,'String'));
handles.ndx=ndx;
guidata(hObject, handles);
```

```
% --- Executes during object creation, after setting all properties.
function ndy_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ndy (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function ndy_Callback(hObject, eventdata, handles)
% hObject    handle to ndy (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of ndy as text
%        str2double(get(hObject,'String')) returns contents of ndy
as a double
ndy=str2double(get(hObject,'String'));
handles.ndy=ndy;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function lx_CreateFcn(hObject, eventdata, handles)
% hObject    handle to lx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function lx_Callback(hObject, eventdata, handles)
% hObject    handle to lx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
% Hints: get(hObject,'String') returns contents of lx as text
%        str2double(get(hObject,'String')) returns contents of lx as
a double
lx=str2double(get(hObject,'String'));
handles.lx=lx;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function ly_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ly (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function ly_Callback(hObject, eventdata, handles)
% hObject    handle to ly (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of ly as text
%        str2double(get(hObject,'String')) returns contents of ly as
a double
ly=str2double(get(hObject,'String'));
handles.ly=ly;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function thc_CreateFcn(hObject, eventdata, handles)
% hObject    handle to thc (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end
```

```
function thc_Callback(hObject, eventdata, handles)
% hObject    handle to thc (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of thc as text
%        str2double(get(hObject,'String')) returns contents of thc
as a double
thc=str2double(get(hObject,'String'));
handles.thc=thc;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function Tn_CreateFcn(hObject, eventdata, handles)
% hObject    handle to Tn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function Tn_Callback(hObject, eventdata, handles)
% hObject    handle to Tn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of Tn as text
%        str2double(get(hObject,'String')) returns contents of Tn as
a double
Tn=str2double(get(hObject,'String'));
handles.Tn=Tn;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function Te_CreateFcn(hObject, eventdata, handles)
% hObject    handle to Te (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
```

```
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function Te_Callback(hObject, eventdata, handles)
% hObject    handle to Te (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of Te as text
%        str2double(get(hObject,'String')) returns contents of Te as
a double
Te=str2double(get(hObject,'String'));
handles.Te=Te;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function Ts_CreateFcn(hObject, eventdata, handles)
% hObject    handle to Ts (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function Ts_Callback(hObject, eventdata, handles)
% hObject    handle to Ts (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of Ts as text
%        str2double(get(hObject,'String')) returns contents of Ts as
a double
Ts=str2double(get(hObject,'String'));
handles.Ts=Ts;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function Tw_CreateFcn(hObject, eventdata, handles)
```

```
% hObject    handle to Tw (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function Tw_Callback(hObject, eventdata, handles)
% hObject    handle to Tw (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of Tw as text
%        str2double(get(hObject,'String')) returns contents of Tw as
a double
Tw=str2double(get(hObject,'String'));
handles.Tw=Tw;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function fbn_CreateFcn(hObject, eventdata, handles)
% hObject    handle to fbn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function fbn_Callback(hObject, eventdata, handles)
% hObject    handle to fbn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of fbn as text
```

```
%        str2double(get(hObject,'String')) returns contents of fbn
as a double
fbn=str2double(get(hObject,'String'));
handles.fbn=fbn;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function fbe_CreateFcn(hObject, eventdata, handles)
% hObject    handle to fbe (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function fbe_Callback(hObject, eventdata, handles)
% hObject    handle to fbe (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of fbe as text
%        str2double(get(hObject,'String')) returns contents of fbe
as a double
fbe=str2double(get(hObject,'String'));
handles.fbe=fbe;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function fbs_CreateFcn(hObject, eventdata, handles)
% hObject    handle to fbs (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end
```

```
function fbs_Callback(hObject, eventdata, handles)
% hObject    handle to fbs (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of fbs as text
%        str2double(get(hObject,'String')) returns contents of fbs
as a double
fbs=str2double(get(hObject,'String'));
handles.fbs=fbs;
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function fbw_CreateFcn(hObject, eventdata, handles)
% hObject    handle to fbw (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function fbw_Callback(hObject, eventdata, handles)
% hObject    handle to fbw (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of fbw as text
%        str2double(get(hObject,'String')) returns contents of fbw
as a double
fbw=str2double(get(hObject,'String'));
handles.fbw=fbw;
guidata(hObject, handles);

                  % % --- Executes during object creation, after
setting all properties.
                  % function NNFEA_CreateFcn(hObject, eventdata,
handles)
                  % % hObject    handle to NNFEA (see GCBO)
                  % % eventdata  reserved - to be defined in a
future version of MATLAB
                  % % handles    empty - handles not created until
after all CreateFcns called
                  %
```

```
                % % Hint: edit controls usually have a white
background on Windows.
                % %       See ISPC and COMPUTER.
                % if ispc
                %     set(hObject,'BackgroundColor','white');
                % else
                %
set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
                % end
                %
                %
                %
                % function NNFEA_Callback(hObject, eventdata,
handles)
                % % hObject    handle to NNFEA (see GCBO)
                % % eventdata  reserved - to be defined in a
future version of MATLAB
                % % handles    structure with handles and user
data (see GUIDATA)
                %
                % % Hints: get(hObject,'String') returns contents
of NNFEA as text
                % %        str2double(get(hObject,'String'))
returns contents of NNFEA as a double
                % global ndx ndy lx ly nodes thc
                % ndx=handles.ndx;
                % ndy=handles.ndy;
                % lx=handles.lx;
                % ly=handles.ly;
                % thc=handles.thc;
                % Tn=handles.Tn;
                % Te=handles.Te;
                % Ts=handles.Ts;
                % Tw=handles.Tw;
                % fbn=handles.fbn;
                % fbe=handles.fbe;
                % fbs=handles.fbs;
                % fbw=handles.fbw;
                %
                % % NNFEA
                % load NNet;
                % [nodes,connect]=geometry;
                % [T,fb]=BC(0, Tw,Ts,Te,Tn, fbw,fbs,fbe,fbn);
                %
[T_lin,T_NL,gradT,flux,kappa_temp1]=masterNN(T,fb,net,0);
                %
                % axes(handles.tempdistNN)  %plot temp dist in GUI
                % resultplot(T_NL,2);
                %
                % axes(handles.KvsTNN)    %plot KvsT in GUI
                % plot(kappa_temp(:,3),kappa_temp(:,2),'b^')
                % title('Conductivity vs. Temperature')
                % xlabel('Temperature')
                %
```

```
                % axes(handles.nodaltempsNN)  %plot nodal temps
                %
plot((1:length(nodes)),T_lin,(1:length(nodes)),T_NL,'r')
                % title ('temperature values @ nodes')
                % legend ('linear approximation', 'nonlinear
solution')
                % xlabel ('node number'); ylabel ('temperature
value');



                % % --- Executes during object creation, after
setting all properties.
                % function NLFEA_CreateFcn(hObject, eventdata,
handles)
                % % hObject    handle to NLFEA (see GCBO)
                % % eventdata  reserved - to be defined in a
future version of MATLAB
                % % handles    empty - handles not created until
after all CreateFcns called
                %
                % % Hint: edit controls usually have a white
background on Windows.
                % %       See ISPC and COMPUTER.
                % if ispc
                %     set(hObject,'BackgroundColor','white');
                % else
                %
set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
                % end
                %
                %
                %
                % function NLFEA_Callback(hObject, eventdata,
handles)
                % % hObject    handle to NLFEA (see GCBO)
                % % eventdata  reserved - to be defined in a
future version of MATLAB
                % % handles    structure with handles and user
data (see GUIDATA)
                %
                % % Hints: get(hObject,'String') returns contents
of NLFEA as text
                % %        str2double(get(hObject,'String'))
returns contents of NLFEA as a double
                % global ndx ndy lx ly nodes thc
                % ndx=handles.ndx;
                % ndy=handles.ndy;
                % lx=handles.lx;
                % ly=handles.ly;
                % thc=handles.thc;
                % Tn=handles.Tn;
                % Te=handles.Te;
                % Ts=handles.Ts;
```

```
                    % Tw=handles.Tw;
                    % fbn=handles.fbn;
                    % fbe=handles.fbe;
                    % fbs=handles.fbs;
                    % fbw=handles.fbw;
                    %
                    % % NLFEA
                    % [nodes,connect]=geometry;
                    % [T,fb]=BC(0, Tw,Ts,Te,Tn, fbw,fbs,fbe,fbn);
                    % [T_lin,T_NL,gradT,flux,kappa_temp]=master(T,fb);
                    %
                    % axes(handles.tempdistNL)  %plot temp dist in GUI
                    % resultplot(T_NL,2);
                    %
                    % axes(handles.KvsTNL)    %plot KvsT in GUI
                    % plot(kappa_temp(:,2),kappa_temp(:,1),'b^')
                    % title('Conductivity vs. Temperature')
                    % xlabel('Temperature')
                    %
                    % axes(handles.nodaltempsNL)  %plot nodal temps
                    %
plot((1:length(nodes)),T_lin,(1:length(nodes)),T_NL,'r')
                    % title ('temperature values @ nodes')
                    % legend ('linear approximation', 'nonlinear
solution')
                    % xlabel ('node number'); ylabel ('temperature
value');


% --- Executes on button press in NNvsNLFEA.
function NNvsNLFEA_Callback(hObject, eventdata, handles)
% hObject    handle to NNvsNLFEA (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

global ndx ndy lx ly nodes thc
ndx=handles.ndx;
ndy=handles.ndy;
lx=handles.lx;
ly=handles.ly;
thc=handles.thc;
Tn=handles.Tn;
Te=handles.Te;
Ts=handles.Ts;
Tw=handles.Tw;
fbn=handles.fbn;
fbe=handles.fbe;
fbs=handles.fbs;
fbw=handles.fbw;
NNID=handles.NNID;

NNvsNL

axes(handles.tempdistNN)  %plot temp dist in GUI
resultplot(TNL_NL,1);
```

```
title ('NL-FEA');
resultplot(TNL_NN,2);
title ('NN-FEA');

axes(handles.nodaltempsNN)  %plot nodal temps
plot((1:length(nodes)),TNL_NL,'b^',(1:length(nodes)),TNL_NN,'ro')
legend ('NLFEA', 'NNFEA')
title ('temperature values @ nodes')
xlabel ('node number'); ylabel ('temperature value');

axes(handles.KvsTNN)    %plot KvsT in GUI
plot(kappa_tempNN(:,3),kappa_tempNN(:,2),'ro',
kappa_tempNL(:,2),kappa_tempNL(:,1),'b^')
legend ('NNFEA', 'NLFEA')
title('Conductivity vs. Temperature')
xlabel('Temperature')

% --- Executes during object creation, after setting all properties.
function NNID_CreateFcn(hObject, eventdata, handles)
% hObject    handle to NNID (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else

set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor
'));
end



function NNID_Callback(hObject, eventdata, handles)
% hObject    handle to NNID (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of NNID as text
%        str2double(get(hObject,'String')) returns contents of NNID
as a double

NNID=get(hObject,'String');
handles.NNID=NNID;
guidata(hObject, handles);
```

NNvsNL.m

```
%NNvsNL.m

%script to compare NNFEA vs NLFEA for a userdefined NN and a user
defined
%set of BC's
%created on: 6/16/2004 by Shawhin Roudbari
%last update: 6/16/2004

% global ndx ndy lx ly thc
% load NNa2153_23
a=['load NNa' NNID];
eval(a);
maxandmins(1);
% ndx=7; ndy=7; lx=2; ly=2; thc=1;
% Tw=100; Te=300; Ts=400; Tn=0; fbn=50000; fbs=0; fbe=0; fbw=0;
[nodes,connect]=geometry;
[T,fb]=BC(0, Tw,Ts,Te,Tn, fbw,fbs,fbe,fbn);
[Tlin_NL,TNL_NL,gradT_NL,flux_NL,kappa_tempNL]=master(T,fb,0);
[Tlin_NN,TNL_NN,gradT_NN,flux_NN,kappa_tempNN]=masterNN(T,fb,net,0);

%CONTOUR PLOTS
% resultplot(TNL_NL,1);
% title ('NL-FEA');
% resultplot(TNL_NN,2);
% title ('NN-FEA');
%
% %NODAL TEMP PLOTS
% figure                                  %compare temperature for
linear and nonlinear
% plot((1:length(nodes)),TNL_NL,'b^',(1:length(nodes)),TNL_NN,'ro')
% legend ('NLFEA', 'NNFEA')
% title ('temperature values @ nodes')
% xlabel ('node number'); ylabel ('temperature value');
```

Plotgrid.m

```
function [nodes]=plotgrid(dx,dy,elements)

%subroutine to plot mesh
%created on: 2/4/2004 by Shawhin Roudbari
%last updated: 3/12/2004

global nodes lx ly ndx ndy

%loop to create array of node coordinates
index1=0;                                       %counter for
plotting coordinates
for i=0:ndx
    for j=0:ndy
        index1=index1+1;
```

```
        pt(index1,:)=[index1, i*dx, -j*dy];     %create matrix "pt"
with coordinate data
    end
end
nodes=pt(:,2:3);                                %create array of
node coordinates
%plot(nodes(:,1),nodes(:,2),'go')                %plot node points on
grid
%axis ([-1 lx+1 -ly-1 1])

%loops to plot (continuous) horizontal and vertical lines of grid
for i=1:ndy+1
    x=[0, lx];
    y=[nodes(i,2), nodes(i,2)];
    %line(x,y);
end
dum1=1;
for j=1:ndx+1
    x=[nodes(dum1,1), nodes(dum1,1)];
    y=[0, -ly];
    %line(x,y);
    dum1=dum1+ndy+1;
end
```

R.m

```
function [flux]=R(T,Kg_bc,Kg,Rb);

%function to create flux vector (right hand side of system of
%equations to be solved)
%created on: 2/26/2004 by Shawhin Roudbari
%last updated: 3/12/2004

global nodes Tindicator
I=indicator(T);                     %find indicator vector that
identifies nodes with BC's
flux=zeros(length(nodes),1);
for i=1:length(nodes)               %run loop once to get
sum(K_ii*Ts_i) on all BC nodes
    if I(i)==1
        flux(i)=Kg_bc(i,i)*T(i);
    end
end
for i=1:length(nodes)
    for j=1:length(nodes)
        if I(j)==1
            dum1=Kg(i,j)*T(j);
            if I(i)==0
                flux(i)=flux(i)-dum1;
            end
        end
    end
```

```
end
flux=flux+Rb;                          %add prescribed boundary flux
Tindicator=I;           %variable is used on other subroutines in
nonlinar analysis
```

Resultplot.m

```
function resultplot(T_e,position)

%post processor function to plot temperature results
%created on: 2/27/2004 by Shawhin Roudbari
%last updated: 3/12/2004

global nodes ndx ndy
Z=zeros(ndy+1,ndx+1);
for i=0:ndy
    Z(i+1,:)=T_e(ndy+1-i:ndy+1:length(nodes)-i)';
end
if position == 1;
    figure
end
subplot(3,1,position)
colormap cool;
[C,h] = contourf(Z,5);
% clabel(C,h);
axis image
colorbar ('vert')
```

Shape2.m

```
function [J,B,N]=shape2(xi,eta,x,y)

%subroutine to compute shape function matrix, B matrix, and Jacobian
%Created: 3/6/2004 by Shawhin Roudbari
%Last updated: 3/6/2004

global nodes connect lx ly ndx ndy kappa thc
N=ones(1,4);
%find shape functions, N
dum1=[-1 -1 1 1];   %pattern indicator vectors for finding the shape
functions below
dum2=[1 -1 -1 1];
for i=1:4
    N(i)=(1/4)*(1+dum1(i)*xi).*(1+dum2(i)*eta);    %calculate four
shape functions
end

%find gradient matrix, B
dN=(1/4)*[-(1+eta) -(1-eta) (1-eta) (1+eta);...
```

```
       (1-xi) -(1-xi) -(1+xi) (1+xi)];     %derivative of shape
functions wrt xi and eta
Jac=dN*[x(1) y(1); x(2) y(2); x(3) y(3); x(4) y(4)];  %jacobian
J=abs((x(4)-x(1))*(y(2)-y(1)))/4;   %det(J)=Area of element/4  for
rectangluar elements
B=inv(Jac)*dN;    %B matrix
```

trainNN.m

```
function [net]=trainNN(flag,flag2,net,epochs,showepoch,trainerror);

%trainNN.m
%train NN for use in subroutine usermat.m
%start:1/17/2004 by Shawhin Roudbari
%last update: 6/16/2004

global mxi mxj
if flag==0
    [inputn, Jn, dTdx, dTdy, Jx, Jy]=maxandmins(1);
    top=[3,10,10,2];    %NN topoplogy = [number of input nodes,
number of nodes per hidden layer..., number of output nodes]
      net=newff([minmax(inputn)], [top(2),top(3),top(4)],
{'tansig','tansig','tansig'}, 'trainrp'); %****NOTE**** amplified
input normalized values by 1.25
else
    if flag2==1
        load NN0
%     else
%         load NNet
    end
    [inputn, Jn, dTdx, dTdy, Jx, Jy]=maxandmins(0);
end

%EITHER set network parameters and train
net.trainParam.epochs = epochs;
net.trainParam.show = showepoch;
net.trainParam.goal = trainerror;
net=train(net,inputn,Jn);       %train NN

%run forward prop network
Yn=sim(net,inputn);                                  %normalized
results
Y(1,:)=mxj(1)*Yn(1,:);     %denormalize results
Y(2,:)=mxj(2)*Yn(2,:);

%plot results
% figure
% subplot(2,1,1), plot(dTdx,Jx,'^',dTdx,Y(1,:),'o')
% legend ('Measured data','NN approxiamtion')
% ylabel ('flux in x direction'); xlabel ('gradient in x direction')
% subplot(2,1,2), plot(dTdy,Jy,'^',dTdy,Y(2,:),'o')
% ylabel ('flux in y direction'); xlabel ('gradient in y direction')
% title('NN training')
```

usermat.m

```
function [J,k]=usermat(dT,T,net);

%usermat.m
%Obtain Jx, Jy, and conductivity k using pretrained NN, net.m
%INPUTS: dT=2x1 matrix containing temperature gradient at point
%T = scalar tamperature at a point
%net= trained neural netwrok object with
%k is calculated by a forward difference operated on NN simulations
%start:1/17/2004 by Shawhin Roudbari
%last update: 5/25/2004

global mxi mxj
%global minJ maxJ dTdxmin dTdxmax dTdymin dTdymax Tmin Tmax

dTdx=dT(1);
dTdy=dT(2);
inputn=[dTdx/mxi(1); dTdy/mxi(2); T/mxi(3)];
%normalize inputs with respect to maximum values from pretraining
**NOTE** amplified input (by 1.25)

%INITIAL SIMULATION
Jn=sim(net,inputn);
%normalized results
J=[Jn(1)*mxj(1); Jn(2)*mxj(2)];
%J=postmnmx(Jn, minJ, maxJ);                                    %de-
normalize results

%SIMULATION WITH ADDED INCREMENTS
partition=1000;
if dTdx==0 & dTdy==0
%if both gradiaents are zero, use a minimum step
    dTdx_incr=1/partition;
    dTdy_incr=1/partition;
else
%otherwise, use a partition of the max of the absolute values
    dTdx_incr=dTdx+(max(abs(dTdx),abs(dTdy))/partition);
    dTdy_incr=dTdy+(max(abs(dTdx),abs(dTdy))/partition);
end
% dT_incrx=[dTdx_incr;dTdy;T];
% dT_incry=[dTdx;dTdy_incr;T];  %inputs for NN with incremented
values
dT_incrx_n=[dTdx_incr/mxi(1); dTdy/mxi(2); T/mxi(3)];
%normalize inputs
dT_incry_n=[dTdx/mxi(1); dTdy_incr/mxi(2); T/mxi(3)];
J_incrx_n=sim(net,dT_incrx_n);
%runs NN with incremented points, normalized results
J_incry_n=sim(net,dT_incry_n);
J_incrx=[J_incrx_n(1)*mxj(1); J_incrx_n(2)*mxj(2)];
J_incry=[J_incry_n(1)*mxj(1); J_incry_n(2)*mxj(2)];
% J_incr=[J_incrn(1)*mxj(1); J_incrn(2)*mxj(2)];
%de-normalize results
% J_incrx=J_incr(1,:);
```

```
% J_incry=J_incr(2,:);

%PERFORM DIFFERENTIATION

k=zeros(2,2);
k(:,1)= -(J_incrx - J) ./ (dTdx_incr-dTdx);
k(:,2)= -(J_incry - J) ./ (dTdy_incr-dTdy);
% k(1,1)= -(J_incrx - J(1)) / (dTdx_incr-dTdx);
% k(2,2)= -(J_incry - J(2)) / (dTdy_incr-dTdy);
```

BIBLIOGRAPHY

1. D. Gawin, M. Lefik and B.A. Schrefler, 'ANN approach to sorption hysteresis within a coupled hygro-thermo-mechanical FE analysis'*, Int. J. Numer. Meth. Engng*, 50, 299-323 (2001).

2. D.E. Sidarta and J. Ghaboussi, 'Constitutive modeling from non-uniform material tests', *Computers and Geotechnics*, 22, 53-71 (1998).

3. D.E. Sidarta, 'Neural network-based constitutive modeling of granular material', *Ph.D. Thesis*, Department of Civil Engineering, University of Illinois at Urbana-Champaign, 2000.

4. H. S. Shin and G. N. Pande, 'On self-learning finite element codes based on monitored response of structures', *Computers and Geotechnics*, 27, 161-178 (2000).

5. I. Farkas, P. Remenyi and A. Biro, 'A neural network topology for modeling grain drying', *Computers and Electronics in Agriculture*, 26, 147-158 (2000).

6. I.M. Smith and D.V. Griffiths, 'Programming the Finite Element Method', 3rd ed. *John Wiley and Sons*, England, 1999.

7. J. Ghaboussi, D. A. Pecknold, M. Zhang and R. M. Haj-Ali, 'Autoprogressive training of neural network constitutive models', *Int. J. Numer. Meth. Engng.* 42, 105-126 (1998).

8. K.J. Bathe, 'Finite Element Procedures', *Prentice-Hall*, New Jersey, 1996.

9. M.M. Zhang, 'Neural network material models determined from structural tests', *Ph.D. Thesis*, Department of Civil Engineering, University of Illinois at Urbana-Champaign, 1996.

10. R.D. Cook, D.S. Malkus, M.F. Plesha and R.J. Witt, 'Concepts and Applications of Finite Element Analysis' 4th ed. John Wiley and Sons, 2002.

11. R.D. Reed and R.J. Marks II, 'Neural Smithing', *The MIT Press*, 1999.

12. S.K. Kim, B.S. Jung, H.J. Kim and W.I. Lee, 'Inverse estimation of thermophysical properties for anisotropic composites', *Exp. Therm. and Fluid Sci.* 27, 697-704 (2003).

13. V. Dumek, M. Druckmuller and M. Raudensky, 'Novel approaches to the IHCP: neural networks and expert systems', *Inverse Problems in Engineering: Theory and Practice, ASME,* 275-282 (1993).

14. Y.M.A. Hashash, C. Marulanda, J. Ghaboussi ans S. Jung, 'Systematic update of a deep excavation model using field performance data', *Computers and Geotechnics*, 30, 477-488 (2003).