Determining Finite Element Mesh Density from Problem Specification using Neural Networks

by

Derek Dyck, B.Eng. (Honours, Electrical)

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Engineering

Computational Analysis and Design Laboratory
Department of Electrical Engineering
McGill University
Montreal, Canada
November, 1990

Abstract

This thesis addresses the problem of how to determine the optimum level of mesh discretization required to solve a magnetic device accurately and efficiently using finite elements. Currently, most finite element packages require user intervention to assure that the mesh density is appropriate for the device. This requires that the user be knowledgable in finite-element analysis and magnetic device design.

The approach introduced here uses a neural network which is trained to recognize significant geometric features and material properties from the description of a magnetic device. Based on its knowledge of meshing rules the neural network computes the mesh density required for an optimum mesh of the device. The neural network acquires this knowledge from examples of "ideal" meshes.

The system requires no user intervention and can be used either independently or as a preprocessor to an adaptive mesh refinement system.

Résumé

Le sujet de cette thèse est la détermination de la densité optimale de maillage d'un composant magnétique requis pour une solution précise et efficace par la méthode des éléments finis. Actuellement, la plupart des logiciels d'éléments finis requièrent l'intervention de l'utilisateur pour s'assurer que la densité du maillage est adéquate. Ceci sous-entend, de la part de l'utilisateur, un niveau d'expertise en éléments finis et en analyse magnétique.

L'approche suivie ici consiste en un réseau de neurones qui a été exercé à reconnaître, à partir de la description du composant magnétique, les traits géométriques et les propriétés des matériaux significatifs. Se basant sur sa connaissance des règles de maillage, le réseau de neurones calcule la densité requise pour un maillage optimal du composant. Le réseau de neurones acquiert ce savoir à partir d'exemples de maillages optimaux.

Le système ne requiert aucune intervention de l'utilisateur, et peut être utilisé indépendemment, ou comme préprocesseur à un système adaptateur de raffinement de maillage.

Acknowledgements

I am indebted to many people without whose assistance this project would never have succeeded. My thanks and gratitude to my advisor, Prof. Dave Lowther, for starting me on the right track, and keeping me on it. Thanks also to Dr. Steve McFee, for sharing his experience in finite elements in the many enlightening conversations we had (and for donating his finite element code). And without the encouragement of my colleagues at the CADLAB, this project would not have been what it is. Thanks to all of you.

Thanks also goes to Prof. Dominique Pelletier and Jean-François Hétu, of l'Ecole Polytechnique de Montréal, for help with the use of their automatic mesh generator. Finally the financial support of Fonds pour la Formation des Chercheurs et Aide à la Recherche, Natural Sciences and Engineering Research Council of Canada, Centre de Recherche Informatique de Montréal, and McGill University Department of Engineering is gratefully acknowledged.

Table of Contents

Chapter 1
Introduction

1.1 Problem Description

This thesis grew out of an investigation into the applications of neural networks in electrical engineering design and analysis. The idea of using neural networks was first proposed as an alternative to expert systems in a project involving the user interface to a magnetics analysis program. From the investigation came the concept of using neural networks to eliminate the need for user interaction in the mesh generation stage of magnetic device analysis.

Currently the users of most magnetic device analysis packages must guide the mesh generation phase to ensure that the mesh has the proper level of discretization. This requires knowledge about finite element analysis, and about the properties of the solution to the problem. Without this guidance the mesh may be too coarse to model the solution accurately at certain critical points in the device domain, or over-discretized, requiring a large amount of computer time in solution and post-processing.

One approach which does not require user interaction uses an *adaptive solver* to refine an initial coarse mesh. The adaptive solver uses an error criterion to estimate the error of the crude mesh, and refines it where the error is estimated to be large. However this method starts out with essentially zero knowledge of meshing rules and the problem solution.

The premise of this thesis is that neural networks can learn the required knowledge, and function with a mesh generator to generate meshes without requiring any user input. This premise is validated by creating a system which computes a good approximation to the ideal mesh density, in the 2-D case and with linear materials, for steady state problems. The meshes generated by this system can be used as-is,

or they can be used as the initial mesh for an adaptive solver (to give it a head start).

1.2 Original Contributions

This thesis makes the following contributions to original research:

1) The application of neural networks to determine the finite element mesh density from the geometric specification and material properties of a magnetic device.

2) A local representation for the input to a neural network of a specified geometry for special symmetry conditions.

3) A general method of representing continuous real values for the output of a neural network.

4) A method of estimating the ideal element size at any point in the geometric domain of a magnetic device, based on the "exact" solution.

1.3 Literature Survey

This application of neural networks essentially bridges the gap between the field of finite elements, and the field of neural networks. The relevant literature is reviewed in the following sub-sections.

1.3.1 Neural Networks

From the field of neural networks an essential reference for this application is [Rumelhart, 1986], which presents the architecture used in this application. For an introduction to various neural network architectures, [Caudill, 1989] is a good source. The text: [Hecht-Nielsen, 1990] is indispensable for developing a neural network application. However no specific examples of other applications resembling this one

could be found.

In some ways this application is similar to visual pattern recognition. Much effort has been focused in this field; the following cover some of the more relevant material. The neocognitron [Fukashima, 1982] is a network optimized for character recognition. Some special architectures for dealing with symmetries have been developed by B. Widrow, see, for example, [Widrow, 1988a]. Also an application of neural networks using Fourier-Mellin spatial filters is described in [LeJeune, 1989]. Unfortunately, the special features and properties of the mesh density problem make most of these references inapplicable, as some results presented later will show. Also visual pattern recognition is largely concerned with the problem of *noise*. However, in this application, noise is not an issue since the device geometry and properties are typically specified with great precision.

1.3.2 Mesh Generation for Finite Elements

From the field of finite element analysis there is practically no useful literature. Virtually all the relevant mesh generation research ([Jin, 1990], [Fujita, 1988], [Baehmann, 1987]) is focused on generating a mesh assuming that the desired mesh density is specified. Most of these papers expect the mesh generator to form part of the loop in an adaptive solver (the mesh density is, in this case, determined using an error criterion applied to an approximate solution from a crude mesh). This step in the mesh generation process is also required by this system, but the difficult part is not generating a mesh based on density, but rather determining the proper density in the first place. Therefore this system is based on the existence of density driven mesh generators.

Other mesh generators use hard-coded rules (e.g., [Reichert, 1990], which, incidently, does *not* use expert system methods as claimed). These rules are used to ensure optimum grading and triangle shape, and do not take into account any properties of

the device or of the solution.

Some knowledge about finite element analysis of magnetic devices is used in this application. However this knowledge is very general, and can be found in any good text on finite elements for electromagnetics (e.g., [Lowther, 1986]).

Chapter 2
The Neural Network Approach

Recently neural networks have received increased attention because of their ability to learn from examples and to generalize. While neural networks are conceptually very simple, there are still many aspects that are poorly understood. This chapter describes neural networks and gives a constructive interpretation of how they work. Although much of what is presented in this chapter appears elsewhere in the literature ([Rumelhart, 1986], [Hecht-Nielsen, 1990]), it is included here for completeness. Even so, this thesis is necessarily terse in the background of neural networks. For a more thorough and readable introduction to neural networks, the reader is encouraged to read [Rumelhart, 1986]. This will form the basis for the system used to solve the mesh discretization problem.

2.1 General Definition of a Neural Network

Neural networks were inspired by biological computing systems as exemplified in animal brains. However the neural network concept has evolved to include a much broader range of computing systems. The following definition is taken from [Caudill, 1989]:

Definition: A neural network is a computing system made up of a number of simple, highly interconnected processing elements (nodes), which processes information by its dynamic state response to external inputs.

Neural networks are characterized by their ability to generalize from examples, to extract features present in a set of inputs, and to tolerate uncertainty. Neural networks are an inherently *parallel* computing architecture, and while some parallel hardware implementations exist, most neural network applications are developed on standard serial computers. This highlights the fact that the actual difference between

neural networks and conventional computing is the computing paradigm behind the network structure.

2.1.1 The Neural Network Paradigm

The conventional *procedural* computing paradigm emphasizes above all else the *algorithm* behind a data processing task. How data is represented plays a secondary role compared to how the data is manipulated.

In the neural network computing paradigm, on the other hand, the "algorithm" is essentially the same for different applications (for a given neural network architecture). The emphasis is instead placed on *representation*. Using a neural network paradigm, a data processing task is first reformulated in input/output terms. A representation is chosen for the ll the possible inputs and outputs, and then, if the representation is "good enough", a *mapping* between input and output can be determined automatically. Exactly what this means will become clearer in the following sections.

2.1.2 Neural Network Architectures

The previous statements apply to practically all neural network architectures. Many sophisticated architectures exist such as Fukushima's Neocognitron [Fukushima, 1982], Grossberg's adaptive resonance network [Grossberg, 1983], and Kohonen's associative map [Kohonen, 1984]; these architectures are surveyed in [Lippmann, 1987] and [Caudill, 1989]. For the application presented here one of the simpler and more established architectures proved to be suitable: the *backpropagation network*. The following analysis will focus on this network architecture.

2.2 The Backpropagation Network

The backpropagation network is also referred to as the "layered feedforward network", and "the mapping neural network". As the name suggests, the nodes in the network are arranged in layers (see Figure 1). The first layer consists of input nodes; their values are set externally and represent the information the network will use to determine the output of the data processing task. Each node in the following layers takes its input from the layer previous to it, and computes an output which is a bounded monotonic function of the weighted sum of these inputs. The nodes in the last layer are called output nodes. Once the computation of the node values has propagated through the layers, the output nodes represent the solution to the data processing task for the input presented to the input nodes.

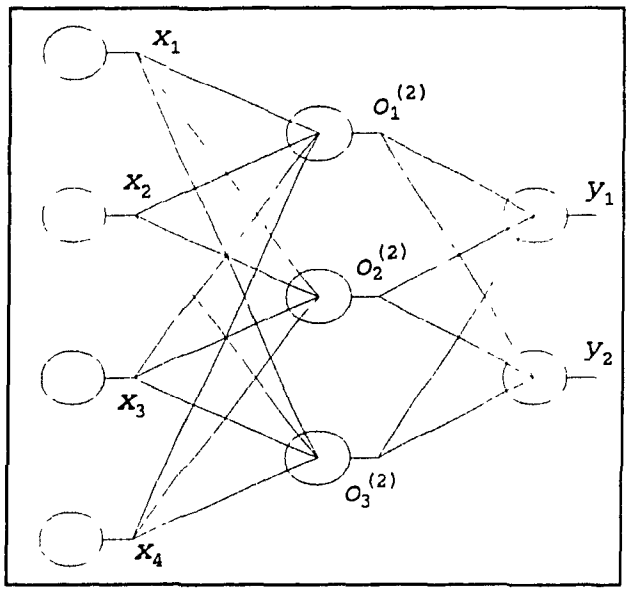

Figure 1 - A Small Backpropagation Network

2.2.1 Feedforward Mathematics

The following four equations express this mathematically:

$$\begin{aligned} \boldsymbol{o}^{(1)} &= \boldsymbol{x} \\ \boldsymbol{s}^{(p)} &= \boldsymbol{W}^{(p)}\boldsymbol{o}^{(p-1)} \\ o_i^{(p)} &= f_i^{(p)}\left(s_i^{(p)}\right), \quad 1 \le i \le N_p \\ \boldsymbol{y} &= \boldsymbol{o}^{(M)} \end{aligned} \tag{1}$$

where:

$\boldsymbol{o}^{(p)}$ is the (column) vector of node values of layer p, i.e.,

$o_i^{(p)}$ is the value of node i in layer p,
$\boldsymbol{x}$ is the vector of input node values,
$\boldsymbol{y}$ is the vector of output node values,
$\boldsymbol{W}^{(p)}$ is the matrix of weights, i.e.,
$w_{ij}^{(p)}$ is the connection weight from node j in layer p-1 to node i in layer p.

also:

N_p is the number of nodes in layer p,
M is the number of layers, and finally
$f_i^{(p)}()$ is the node transfer function.

A commonly used transfer function is:

$$f_i(s_i) = \frac{1}{1 + e^{-(s_i + a_i)}} \qquad (2)$$

where the superscript p has been dropped for clarity, and where a_i is an adjustable threshold parameter of the function. This parameter can be seen as a weight on a connection to a node which has a constant value of 1; therefore this parameter should be thought of as just another weight. Figure 2 shows the transfer characteristics in diagram form.

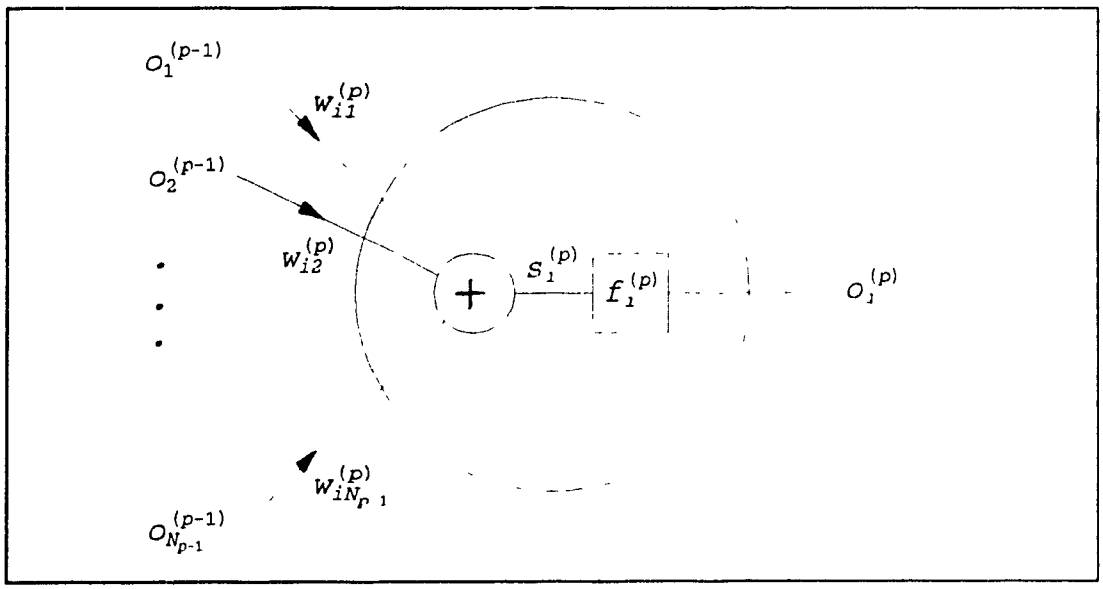

Figure 2 - Node Transfer Function

2.2.2 Network Topology

The topology of a backpropagation network is the specification of the number of layers, the number of nodes in each layer, and the form of the transfer function of each node. The weights themselves are not considered to be part of the architecture, and are initially given random values (the final values for the weights are determined through a process called training). The topology itself has no direct relationship to the computational problems a given network is meant to solve (except for fixing the number of input and output nodes), although the topology will influence the performance. There is, as of yet, no analytical method of determining the optimal topology. Since the topology is not modified during learning, all the knowledge acquired by the network about a certain data processing task is encoded in the weights during the training phase.

2.3 Training the Backpropagation Network

The backpropagation network acquires its knowledge from a set of *training examples*. The training examples are input-output pairs, where each pair consists of the input values of one example and the corresponding target output values. During the training phase, one of the inputs from the set of training examples (selected at random) is presented to the network and the output values are computed. The weights are then modified to reduce the error between the actual output values and the target output values. This process is repeated until the network response is sufficiently close to the targets for all the examples in the training set. How close depends on the application; however it may also happen that the network never reaches a point where the error is sufficiently small. This phenomena can be understood most easily from the numerical perspective presented in the next section.

2.3.1 The Error Backpropagation Rule

The backpropagation network gets its name from the method used to train the network, the *error backpropagation rule*. This rule was discovered independently by several researchers including [Rumelhart, 1986]. In this training scheme, the network is trained by backward propagation of the error from the outputs to the inputs. This algorithm defines the error of the neural network with respect to the k-th training example as:

$$E^{(k)} = \|\boldsymbol{y}^{(k)} - \boldsymbol{t}^{(k)}\|^2 \qquad (3)$$

where $\boldsymbol{y}^{(k)}$ is the output of the network when training input $\boldsymbol{x}^{(k)}$ is applied to the inputs of the neural network, and $\boldsymbol{t}^{(k)}$ is the target output for that same training input. The *total* error is then defined as $E = \Sigma E^{(k)}$. The algorithm minimizes this total error with respect to the weights using a gradient descent algorithm.

This algorithm was implemented exactly as described in [Rumelhart, 1986], using these two equations:

$$\Delta w_{ij}^{(k)} = \alpha \Delta w_{ij}^{(k-1)} + \beta \frac{\partial E^{(k)}}{\partial w_{ij}} \qquad (4)$$

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \Delta \boldsymbol{w}^{(k)}$$

where α is a momentum parameter (approximately equivalent to the successive over-relaxation parameter in relaxation terminology), and β is the "step size" of each update. Interested readers can refer to [Rumelhart, 1986] for more details on the backpropagation algorithm.

The previous section mentioned the possibility that the network never attains a sufficiently small error. The reason for this is due to the fact that the backpropagation algorithm is a minimization algorithm, but the surface of the function being minimized (the total error) does not have a single minimum. In fact the error surface typically has many local minima which can be near or far from the global minimum. Since the backpropagation always travels "downhill", it is liable to

get stuck in one of these minima. The next section discusses this problem in terms of other minimization methods.

2.3.2 Modifying the Backpropagation Algorithm

As mentioned in the previous paragraph, the backpropagation algorithm minimizes the error by gradient descent. Since it is well known that gradient descent methods are inferior to other minimization algorithms in many applications, it is tempting to try some of these superior algorithms to minimize the error of the neural network.

The conjugate gradient method is one such method; it is well documented in the literature (e.g Section 10.6 of [Press, 1988]), and features quadratic convergence on a certain class of problems. Also one application, [Lapedes, 1988], uses the conjugate gradient method (demonstrating that it is workable), but without comparison to the gradient descent method. Accordingly the conjugate gradient algorithm was implemented, and the results compared to the gradient descent algorithm.

The results were disappointing; the conjugate gradient method did converge to a minimum in many fewer iterations, but the minimum it converged to was far from global. The gradient descent method, in general, finds much better minima. There are several possible explanations for this. First of all, the momentum parameter in the weight update step allows the gradient descent algorithm to escape a local minimum in some cases by carrying it through the minimum. Secondly the fact that the weight update occurs after each individual example means that the error "landscape" changes at each iteration. This means that what is a local minimum for one example may not be for the next. Finally the order of the examples is chosen at random; this adds a certain amount of "noise" to the direction taken, again so that escape from local minima is possible. Of course in theory these three factors could also cause it to escape from a better minimum to a worse one, but in practice this does not happen often enough to impair the superior performance of the gradient

descent algorithm.

Attempts at using other minimization methods such as simulated annealing [Press, 1988], and genetic algorithms [Montana, 1989] have also been attempted (but not by this author) without much success. These methods are good for minimizing functions with many local minima, however they have only limited usefulness because they converge much more slowly than even the gradient descent method (although the minimum they find might be closer to a global minimum). The reason for this is that each set of weights is chosen almost independently of the others. This means that the chances of finding a good set of weights are almost vanishingly small. To illustrate this, consider a network of only 75 weights (a rather small network). If an implementation of these algorithms tested one million weight matrices per second (a figure only specialized parallel hardware could acheive), it would take one *billion* years just to test one weight vector in each "quadrant" of the weight space. Considering that the estimated geological age of the earth is under five billion years, it is clear that an exhaustive search of this kind is quite infeasible. These experimental results favour the standard backpropagation algorithm. Also, although no formal analysis on convergence properties has been carried out, the gradient descent method is the method of choice in practically all applications of the backpropagation neural network.

2.4 Numerical versus Knowledge Based Interpretation

The previous sections described neural networks in very minimal terms. The description specified, in mathematical terms, the operation of the network and the method used to modify the weights during training. In principle this is sufficient to implement a neural network, however it does not give much insight into the behaviour of the network as a whole.

This emergent behaviour is often described using different terms from the artificial

intelligence field such as pattern recognition, feature extraction, and generalization. However, especially in this application, the emergent behaviour is best described from a numerical analysis perspective.

2.4.1 Emergent Behaviour: Numerical Perspective

The mathematics describing the neural network presented in the previous section suggest that a neural network can be interpreted as a functional mapping between the inputs and the outputs. In this type of mapping, the input space is the N_1 dimensional space of input points, and the output space is the N_M dimensional space of output points. Specifically, a backpropagation neural network can be defined as follows:

Definition: A *backpropagation neural network* is a multi-valued function of many variables; the solution to a computing problem is found by *evaluating* this function at a specified point.

The "specified point" is the input to the neural network, and the values of the function correspond to the output of the neural network (the solution to the computing problem). Using this definition, many of the properties of the neural network can be explained in terms describing functional mappings. In particular, the ability of the network to generalize can been seen as *interpolation* between the training examples in the output space. This has implications regarding input and output representation, which is discussed in the next chapter.

2.4.2 Learning as Modelling

If the neural network is interpreted as a function, then the process of learning can be interpreted as *modelling*. The weights can be seen as the *parameters* of a modelling function, and the training examples as *data points* in the multi-dimensional

(input-output) space of this function. Then the error backpropagation rule is identical to a *least squares fit* (minimized by gradient descent) of the neural network "modelling function" to these data points. This puts neural networks on more familiar ground, and allows the large body of knowledge about modelling theory to be applied to neural networks.

2.4.3 Using Other Modelling Functions

If the neural network can be interpreted as a modelling function, then it is appropriate to ask whether other modelling functions might not be more suitable for implementing the functional mapping between inputs and outputs. While no answer to this question is presented here, some background information might prove to be illuminating. This question is addressed in two papers by different authors. Both papers describe a solution to the same problem, but each uses a different approach. The first paper, [Lapedes, 1988], implements a solution using neural networks. The second paper, [Farmer, 1988], uses an explicit interpolation function which is *local*, i.e., the output at a specified input point is determined by interpolating between the output of the nearest neighbours to the input point. In this explicit scheme it turns out that the way in which nearest neighbours are chosen has a significant impact on the accuracy of the interpolation. Also the local interpolation scheme is superior to several different types of global interpolation schemes attempted by the authors.

What is of interest here is that the neural network approach gives very similar results to the local interpolation scheme. This suggests that in fact neural networks are implementing some type of local interpolation scheme. This also suggests that when the neural network learns the input-output pairs, it also learns the features of the input which are important in selecting the "best" nearest neighbours.

2.5 Summary

This chapter outlines the neural network computing paradigm. The system presented in the following chapters is based on the backpropagation neural network architecture. Four key facts about neural networks are introduced:

1) Method of computation: in parallel.

2) Information representation: real numbers.

3) Knowledge representation: in the connection weights.

4) Knowledge acquisition: learns from examples.

Chapter 3
Input and Output Representation

The most difficult aspect of developing a neural network application (not counting the generation of training examples) is finding a workable representation for the input and output of the neural network. Thus most of the experimental work was aimed at determining appropriate representations. The results are the subject of this chapter; more is said on the simulations themselves in Chapter 5.

3.1 General Considerations

The only explicit constraint imposed by the neural network is the format: the input and the output must each be encoded as a set of real numbers. The size of the set (the number of input or output nodes) is arbitrary. The range of each value is also arbitrary, though in practice the values are usually constrained to be in the interval [0,1] or [-1,1]. However the principle on which the neural network operates, as explained in Chapter 2, means that not all representations are equal. In fact the representation chosen is critical to the success of a neural network application.

3.1.1 Rules for Deriving Representations

Unfortunately, the field of neural networks is still very young, and there is as yet no rigorous approach to finding a good representation (this applies to other aspects of neural networks as well!). There is even an scarcity of general rules of thumb or other guidelines to assist developers of neural network applications. The previous chapter, however, does give some idea of what the input and output representation must be capable of. Specifically, since generalization is equivalent to interpolation between training examples, the input and output representation must make this interpolation possible. One implication of this is that the input and the output should be encoded so that similar inputs produce similar outputs.

One general question is: should the input representation favour a simple encoding using a large number of nodes, or a complex encoding using a small number of nodes. In the application presented here, an example of the former - simple coding using many nodes - would be a pixel image of the input geometry. An example of the latter - complex coding using only a few nodes - would be an encoding based on relationships between the line segments making up each geometric object. The same question applies to the output.

The answers to these questions were found through software experiments. In the process, insight was gained into the workings of back-propagation networks. From this insight a set of guidelines was developed for creating input and output representations for neural networks in the general case. These guidelines are presented at the end of this chapter, and are one of the contributions of this thesis.

3.2 Representing the Output Mesh

At first glance, the complexity of a finite element mesh makes it difficult to imagine that a workable encoding exists at all. The idea of a pixel image of a complete finite element mesh appearing at the output of the neural network makes one shudder at the computational cost that would be required. However the reason a neural network was chosen for this application was because of its pattern recognition capabilities. But in this application the key need for pattern recognition is *not* for Delaunay triangulation or optimum grading of the mesh. There already exist efficient algorithmic solutions to these aspects of the meshing process. The aspect of the process which is as yet unsolved, and for which pattern recognition is required, is in determining the optimum *density* of the mesh. Here, and in the following, the density of a mesh at a given point is the *node* density in the vicinity of the point. This is measured by the distance between the nodes, (or, equivalently, the length of the side of the element). The ideal mesh density evenly distributes the error in the solution

computed on the mesh. Later, the chapter on training examples will give more precise definitions for mesh density and for the ideal mesh density. From the density specification it is straightforward to generate a mesh, and software packages already exist which do exactly that [Hétu, 1990]. Therefore it is sufficient for the neural net to compute the density of the mesh everywhere (or at a selection of sample points) in the domain of the mesh.

3.2.1 Properties of the Mesh Density

Before discussing different options for representing the mesh density, it is instructive to study the properties of the mesh density as a function of the input geometry. These properties are defined with respect to points, lines, or the whole domain, whichever is most convenient. Specifically, the mesh density function has the following properties:

1) Translational covariance Translating the input geometry results in an equivalent translation of the mesh density as a whole.

2) Rotational invariance Rotating the input about a point does not change the mesh density at that point.

3) Scale covariance Scaling the input geometry (about a point) results in an equivalent scaling of the mesh density (element size) at that point.

4) Mirror invariance Mirroring the input geometry about a line does not change the mesh density along that line.

Many of these properties are also exploited in other fields of image processing such as character recognition. However there are some important differences. In most pattern recognition tasks of this type, the output is independent of orientation, scale and position of the input. This is not the same as translation and scale *covariance*,

which is used here to mean that the output is transformed in the same way the input was transformed. Therefore techniques used in image processing are not readily applicable to this application.

3.2.2 Representing the Mesh Density at a Point

If the neural network is appropriately set up, the above properties will be *enforced* explicitly. Otherwise they will only be present implicitly in the training examples, and the neural network will attempt to learn these properties from the training examples. The first property has the most relevance to the representation chosen for the output, because it implies that the neural network only has to compute the mesh density at a single point. The mesh density at any point in the domain can then be computed by the appropriate translation of the input geometry. To make this clearer, imagine that the neural network is set up to compute the mesh density at the exact center of a "snapshot" of the input device. Then the mesh density at any point can be computed by "panning" the device across the input to the network such that the center of the snapshot corresponds to the desired point: the output of the neural network will be the required mesh density at that point.

Representing a single value is not difficult. The most obvious representation uses a single output node, with the output value proportional to the size of the element at the output point. But there are still several options to consider. Care must be taken to ensure that the full dynamic range of the output is being used. In fact the reason mesh density was defined in terms of element length is because the distribution of element lengths over the geometric domain is relatively uniform. Figure 3 shows the distributions of two encodings: length and area. These distributions are derived from the ideal sizes at 547 uniformly distributed points in the geometric domain of the C-core device (see Chapter 5). Note how the distribution of the encoding using element area is skewed towards the smaller size values. The opposite is true for the

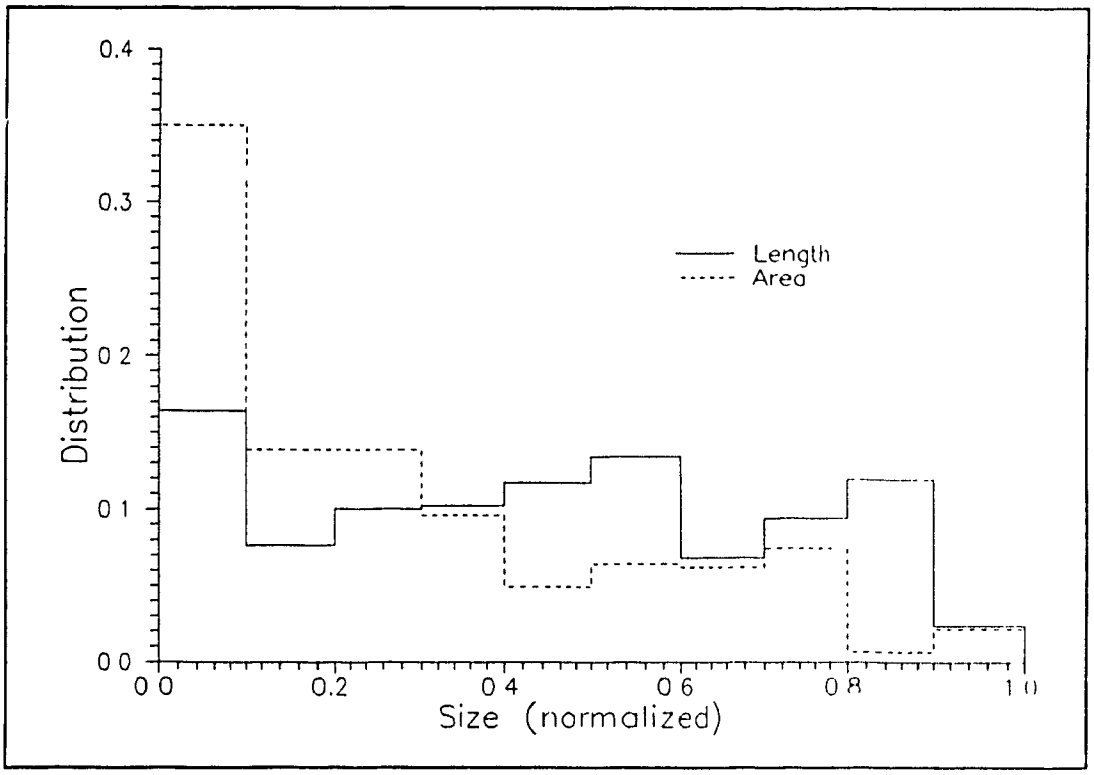

Figure 3 - Distribution of Different Encodings of Size

encoding based on number of elements per unit area (since this is simply the inverse of the encoding using element area), i.e., this encoding is skewed towards the larger size values. For other devices this rule may not hold, but since the device in this case was chosen to be *representative* (in terms of important geometric features and material properties, see Chapter 5), this distribution is as close as practical to the expected distribution.

However, even with a uniform distribution, the neural network has difficulty learning to model a continuous valued output. It proved to be impossible to obtain sufficient accuracy (less than about 5% error on the training examples, or less than 10% on test examples). Why this is so is not obvious; perhaps the neural network fails at continuous interpolation because it has too few degrees of freedom to model every example closely, but too many degrees of freedom to be smooth between examples. In any case, this difficulty is neatly circumvented by using a coding process which

allows for error correction. The next section describes the output encoding and decoding process which reduces this error within acceptable bounds.

3.3 Representing a Continuous Output

The encoding of the output is based on the hypothesis that *recognition* and *classification* are the strong points of neural networks. Accordingly, a representation was developed which takes advantage of these abilities: the output is *discretized*, and each discrete value is represented by a separate output. Conceptually, each output represents all the continuous values in the interval around the corresponding discrete value. To decode this output, a voting system can be used to select the element size corresponding to the output with the largest value. This encoding provides a much greater error immunity, since all that matters is which output is *maximum*, and not the precise value of each output. From the pattern recognition point of view, it means that the network only has to *classify* a given input geometry into one of the bins representing element sizes. Alternatively, each output can be seen as representing a confidence level that the input geometry requires the corresponding element size. This method of representing a continuous value is not restricted to this application. The next two sub-sections explain the encoding and decoding process in detail, for the general case of a continuous valued output.

3.3.1 Encoding a Continuous Output

Since the neural network requires target values for the outputs during training, it is necessary to be able to encode a single continuous target value into many target values, one for each of the output nodes. Before specifying what these values should be, the following formalizes this representation. As stated in the previous paragraph, each output node is assigned to represent a different (discrete) value. These discrete values are chosen in this case to correspond to a linear scale (i.e., each interval has

the same width), although there is no reason that other scales couldn't be used in other applications. In the simplest case, the output of the neural network is decoded by selecting the node with the largest value. If the output nodes are numbered from 1 to N, then if the n-th output has the largest value, the corresponding continuous value is:

$$x = \left(n - \tfrac{1}{2}\right)w \qquad (1)$$

where x is the continuous value, and w, the width of the intervals, is the conversion factor between the node index and the continuous value. In the application presented here, this interval width is defined as:

$$w = \frac{2\bar{x}}{N} \qquad (2)$$

where $\bar{x}$ is the average over all the examples of the continuous values (the sizes in this case), and N is the number of intervals (the number of output nodes). This relation was found empirically to give a uniform distribution over the output nodes, i.e., each node recognizes its share of input geometries. The choice is logical if the values are uniformly distributed. Note however that this implies that values greater than $2\bar{x}$ would map to an index larger than N. In this case the value is *squashed* to $2\bar{x}$; artifacts of this step are discussed in Chapter 4.

To encode a continuous value requires the *inverse* of the mapping from node output values to the continuous value by taking the maximum output. However this inverse mapping is not unique, since the operation of taking the maximum output is not unique. In fact any mapping is valid which assigns the maximum value to the output whose corresponding value is closest to the continuous value. The simplest such mapping assigns 1 this output, and 0 to all other outputs. However this mapping is a bit too simple, and leads to problems during the learning phase. The reason for this is the *discontinuity* in the target output values when the continuous value is at the boundary between intervals. If the continuous value is perturbed slightly, one output node value will jump from 0 to 1, while the neighbouring node value jumps from 1

to 0. This discontinuity is arbitrary, yet causes considerable difficulty when the neural network attempts to model the discontinuity. The next section describes a more appropriate strategy based on the principle of Gaussian error distributions.

3.3.2 Distributing Output Node Values *Normally*

Since this encoding strategy discretizes a continuous value, it has a basic error of half the interval width. Accordingly, it is appropriate to train the neural network assuming that each continuous value has an error which is normally distributed about that value with a standard deviation σ of half the interval width:

$$\sigma = \tfrac{1}{2}w \tag{3}$$

Then the value of each output node is the *probability* that the continuous value falls into the interval corresponding to each node. The probability that a continuous value x falls into interval n is:

$$p_n(x) = \mathcal{P}\{(n-1)w \le x \le nw\} \tag{4}$$

Using the Gaussian probability distribution for the "error" of the continuous value, this probability is:

$$p_n(x) = \mathcal{P}\left\{\frac{(n-1)w - x}{\sigma} \le z \le \frac{nw - x}{\sigma}\right\} \tag{5}$$

where z is normally distributed with a mean of zero and a variance equal to one. Defining the function prob(z):

$$\mathrm{prob}(z) = \mathcal{P}\{\zeta \le z\} = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{z}\exp\left(-\tfrac{1}{2}\zeta^2\right)d\zeta \tag{6}$$

and substituting this and $\sigma=\frac{1}{2}$ into the equation for $p_n(x)$ gives:

$$p_n(x) = \mathrm{prob}(2n - 2x/w) - \mathrm{prob}(2n - 2 - 2x/w) \tag{7}$$

The target output value of node n is this value scaled so that the maximum possible value is unity, which occurs when $x = \left(n-\frac{1}{2}\right)w$:

$$p_{\max} = p_n\left(\left(n-\tfrac{1}{2}\right)w\right) = \mathrm{prob}(1) - \mathrm{prob}(-1) \approx 0.68 \tag{8}$$

i.e., the probability that the error is within one standard deviation of the mean. So

the final expression for output n is:

$$o_n = \frac{p_n(x)}{p_{max}} = \frac{\text{prob}(2n - 2x/w) - \text{prob}(2n - 2 - 2x/w)}{\text{prob}(1) - \text{prob}(-1)} \qquad (9)$$

This mapping has no discontinuities. This mapping also has the advantage of training the outputs for partial recognition, or, more precisely, not *discouraging* an output node as severely if it partially recognizes an input configuration. In terms of a modelling function: this encoding smooths the modelling function, thus making interpolation easier. Also the decoding scheme presented in the next sub-section takes advantage of the extra information available in this encoding to refine the decoded value.

3.3.3 Decoding the Output

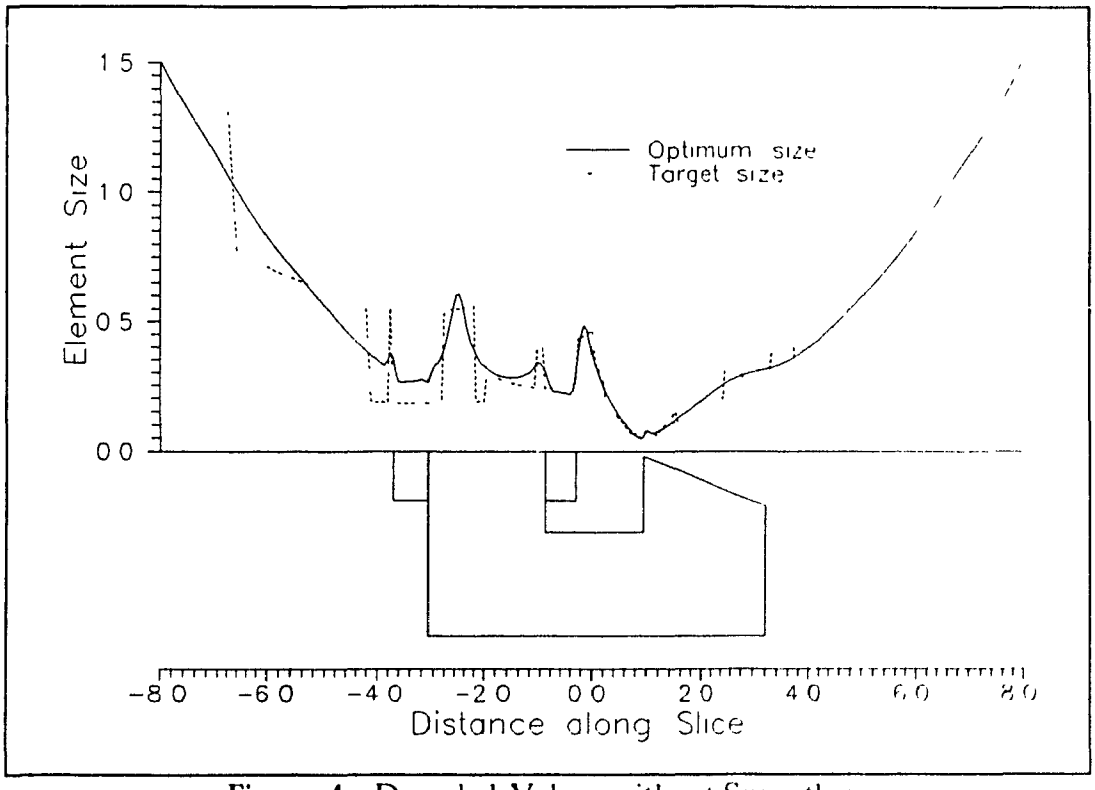

Figure 4 - Decoded Values without Smoothing

As described above, the simplest decoding process simply finds the maximum output

and sets the decoded output value to the value corresponding to this output. However the discretization error, especially with scaling (see Chapter 4), can be quite large, as Figure 4 illustrates. However a simple modification to this basic process can reduce the error to a negligible value.

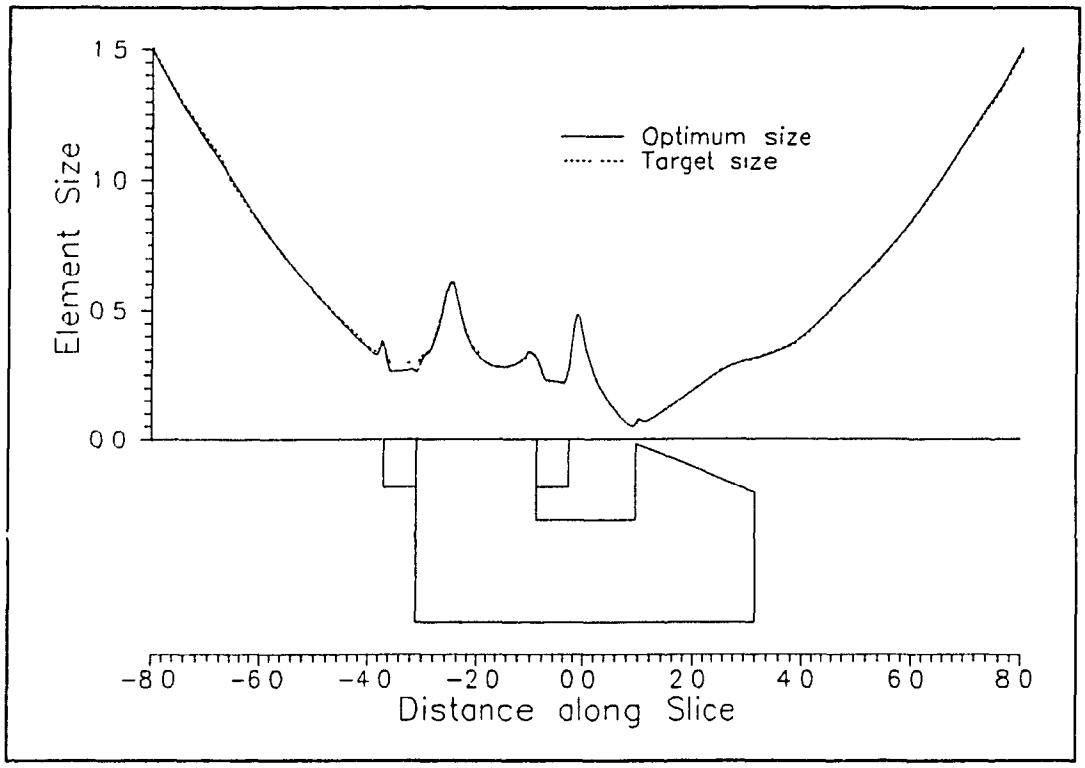

Figure 5 - Decoded Values with Smoothing

The idea is simply to take the weighted average of the values corresponding to the maximum output and its two neighbours. The weighting for the average is the actual value of each output. Thus if a value is on the boundary between two intervals, the adjacent output will be approximately equal to the maximum output, and averaging their corresponding values will produce the correct decoded value. Explicitly, the decoded value is calculated with (c.f. equation (1)):

$$x = \frac{\left(n-\frac{3}{2}\right)o_{n-1} + \left(n-\frac{1}{2}\right)o_n + \left(n+\frac{1}{2}\right)o_{n+1}}{o_{n-1} + o_n + o_{n+1}} \cdot w \qquad (10)$$

Figure 5 shows how closely the decoded values match the original value.

There is one possible snag in the decoding process. It is possible that none of the output values are large enough to be differentiated from noise. The threshold value for this application is 0.2; if all the outputs are less than this value, then it is assumed that the neural network does not recognize the input. This is still preferable to the single output continuous representation, however, since in this case the output can be estimated by other means (e.g., averaging the decoded output at the neighbouring sample points). Other error checks are possible; for example, a check that only one output (and possibly its neighbours) has a large value.

3.3.4 Evaluation of the Encoding Scheme

Table 1 - Error of Different Output Representations

	RMS Error of Continuous Output	RMS Error of Discrete Outputs	RMS Error of Decoded Output
Training Examples	14.3%	12.4%	5.0%
Test Examples	23.9%	24.1%	9.9%

The performance of the output representation compared very favourably to the single output scheme. To compare the two representations, the topology for the (continuous valued) single output network was derived from the (discretized) 10 output network by adding an extra hidden layer with 10 nodes. This way the continuous output network was actually more powerful in principle than the discrete output network. The comparison of the performance is very interesting. Table 1 shows the root mean squared (rms) error of both networks. Note in particular that the rms error before decoding was almost identical to the error of the single

(continuous) output network (23.9% v.s. 24.1% for the test examples). However the rms error of the decoded outputs was much lower for both the training examples and the test examples. This reduction in error is entirely due to the representation of the output. It is expected that similar gains can be made by using this representation in other applications involving a continuous output.

3.4 Representing the Input Geometry

This section describes the representation of the input. Several encodings of the input were evaluated, and what failed to work is almost as important as the final encoding. Accordingly, the following sections describe the evolution of the representation from a "historical" perspective.

3.4.1 Representations based on Image Recognition

At the start, the representation for the input geometry was based on the work of other neural network researchers who were also dealing with geometric type recognition tasks ([Widrow, 1988a], [Fukashima, 1982]). The input in these cases was a pixel image of the geometry. Representing the input in this manner meant that the properties of the mesh density stated in the previous section (rotational invariance, etc.) would have to be *learned* by the neural network. At the time this did not appear to be a significant disadvantage.

A unique topology for the network (based on a structure proposed by B. Widrow) enabled the translational symmetry property to be enforced in a manner that economized on hidden layer nodes. The same weights were used at every output point, and the connections were arranged so that the hidden layer nodes were shared between neighbouring points. The topology also partially enforced the rotational invariance property by duplicating the weights around each point so that the same

output resulted when the input was rotated by multiples of 60 degrees. Scale covariance and mirror invariance were properties that the neural network had to learn from the examples.

As previously mentioned the idea was to represent the input geometry as a pixel image. However as the network architecture evolved, even before the first simulations were run, it became obvious that sampling the input geometry at a fine enough resolution to capture the essential features (such as corners) would be impractical. This was due simply to the computational cost of evaluating the network on a standard (serial) computer (a SUN IV workstation). It was also obvious that a representation of the entire input geometry was also impractical, and that a representation of the geometry local to a specific point would have to be sufficient. Exactly how local the representation can be made and yet still produce useful results is an open question, although the success of the implemented system does set an upper bound of sorts.

Using this architecture several different representations were attempted, the most successful being a *radial* sampling of the input geometry, where each input node represented the distance to the geometry in a different direction (see Figure 6). The results were still far from satisfactory, however, especially when generalization to new input geometries was attempted.

3.4.2 Enhancing the Performance

One other approach that was tested in conjunction with the radial sampling representation uses two additional inputs to encode the direction and magnitude of the magnetic field at each sample point. This may sound like cheating, since the purpose of the mesh generation is to be able to compute the magnetic field! However the magnetic field in this case is derived from a crude solution computed on a near-minimal mesh. A minimal mesh does not have any nodes that are not

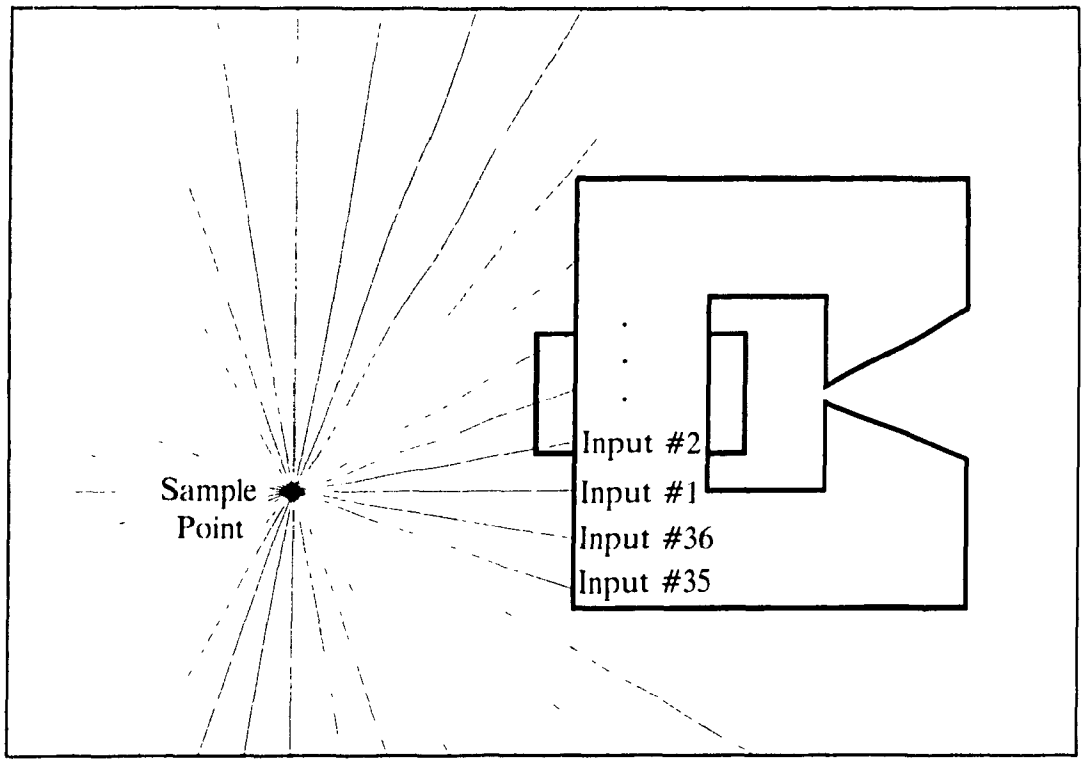

Figure 6 - Input Representation Example

already part of the geometry of a device (i.e., all nodes are at corners of the device). The cost of this solution is small, yet gives the neural network a rough idea of the form of the solution. However this approach was abandoned because the improvement in the performance of the neural network was not significant. This is not to say that this approach is a dead end. In fact this approach may be required when the system is extended to non-linear materials.

One aspect of this process of selecting the best input representation is that it was done in parallel with the selection of the output representation. The ideas presented in the previous sections were all tested in conjunction with the output representation which used only a single continuous output. This means that these input representations may work with the discretized encoding. However, lack of resources prohibits the complete evaluation of all combinations of these representations in this thesis. This is unfortunate, since it leaves many of loose ends, but this is the nature

of scientific research.

In any case, it was decided at this point that the properties of scale covariance and rotational invariance were too important to leave un-enforced, especially since enforcing them would mean that the neural network wouldn't have to learn them. Also the experience gained with different input representations suggested that an appropriate encoding could explicitly enforce these properties.

3.4.3 Enforcing the Properties of the Mesh Density

Accordingly the interleaved, partially rotationally invariant network topology was scrapped, and a standard backpropagation network used instead, with the geometry and element size at each sample point treated as a separate input-output pair. In such a network the input encoding becomes very complex, since it enforces all four of the properties of the mesh density stated above. In this encoding scheme, the sample point is taken as the origin of a local coordinate system (for translational invariance). The input geometry is rotated and scaled so that the closest corner in iron falls on the x-axis a unit distance from the origin (for rotation and scale invariance). Finally the input geometry is flipped, if necessary, so that the corner has a positive orientation with respect to this local x-axis (for mirror invariance). In this coordinate system the local geometry can be described while maintaining all the properties of the mesh density mapping (for the translation and scaling the same transformation is applied to the target sizes)

The final encoding scheme describes the relationship between the two nearest corners of the magnetic device, and requires only eight input nodes. Figure 7 shows the angles and distances used to define the input values. In the figure, all the angles are positive except for φ_2. Corner 1 is always chosen to be the closest corner in iron on the closest segment to the sample point. Corner 2 is always chosen to be the closest corner distinct from Corner 1 (e.g., Corner 2 could on the other side of an air gap).

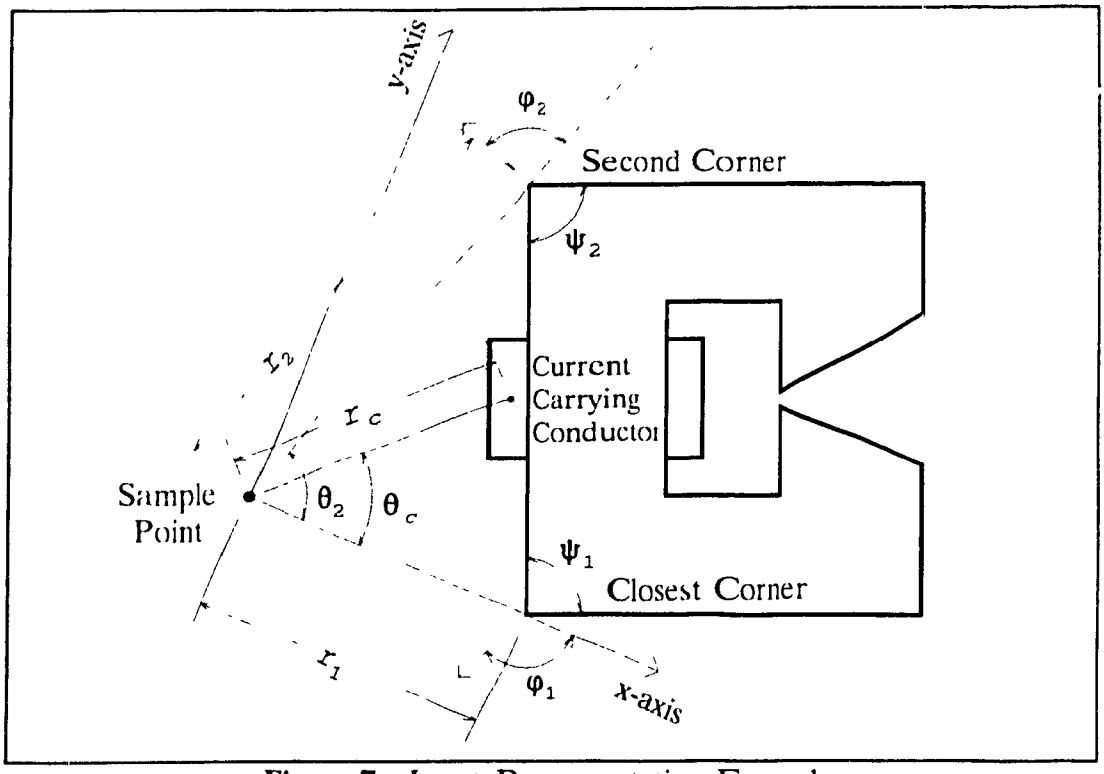

Figure 7 - Input Representation Example

All the inputs are scaled appropriately so that they are either in the interval [0, 1] or [-1, 1]. The inputs are:

Input 1: φ_1/π The orientation (see below) of Corner 1 w.r.t the local x-axis. If necessary the whole geometry is flipped about the local x-axis to make this angle positive.

Input 2: $1 - \psi_1/\pi$ The sharpness of Corner 1.

Input 3: φ_2/π The orientation of Corner 2 w.r.t. the local x-axis.

Input 4: $1 - \psi_2/\pi$ The sharpness of Corner 2.

Input 5: $S(r_2/r_1)$ The squashed distance (see below) to Corner 2 in the (scaled) local coordinate system.

Input 6: θ_2/π The angle from the local x-axis to Corner 2.

Input 7: $S(r_c/r_1)$ The distance to the center of gravity of the nearest current carrying conductor in the (scaled) local coordinate system.

Input 8: θ_c/π The angle from the local x-axis to the center of gravity of the nearest current carrying conductor.

The orientation of a corner is the direction the corner "points". More precisely, the corner has the same orientation as a vector on a line *bisecting* the corner angle, and which points outward from the corner (notice the vectors attached to the corners in Figure 7). The squashing function $S()$ warps the distance so that it is always in the interval [0, 1]. It is defined as:

$$S(x) = \frac{1}{1 + \exp(1-x)} \qquad (11)$$

It should be pointed out that some knowledge about meshing was used in choosing this representation. Specifically, corners are used as the basis for the encoding because it is known that corners are important features of the meshing process (because of the singularity in the solution at corners). The location of the current carrying conductors is also an important consideration for human experts.

A brief comparison between this representation (which will be referred to as the "invariant representation") and the radial sampling representation may be instructive at this point. In fact a numerical comparison is not available since the tested versions of each scheme used a different output representation. However a comparison based on which local features are represented in each scheme is possible. The invariant representation is able to encode the two nearest corners in near perfect accuracy. This includes the line segments attached to these corners. The radial sampling varies considerably in the amount of detail represented. In Figure 6 only a single line segment is encoded, and even the length of this segment is rather vague. In the gap this representation could encode the two faces, but could miss the presence of the

gap (i.e. it would only see a "dent" in the iron) unless a "ray" happened to pass through the gap. The invariant representation could also miss the gap, if the two closest corners were on the same face. What tips the scales in favour of the invariant representation is two factors. The first is the dependence of the mesh density on *corners*. The fact that the radial sampling scheme leaves corners rather vague is therefore a major failing of this scheme. The second factor is the number of inputs that each representation requires. The last section of this chapter explains that the quality of the interpolation between examples can be expected to decrease as the number of inputs increases.

Neither of these representations distinguish between the inside or the outside of a material boundary. This could be another (binary) input, but instead the problem is subdivided, as described in the next sub-section.

3.4.4 Separating Air and Iron

The rules that an expert uses to determine the mesh density in iron are not the same as the rules used when meshing air. For example, the iron at an outside corner in iron is meshed differently from the air at an outside corner in air. Therefore, to improve the performance of the network, the meshing problem is divided into two sub-problems, one problem being the meshing of iron, the other being the meshing of air (current carrying conductors are included as air). Two separate training sessions produce different networks for each region. Both networks have the same topology and initial weights, but they have different final weights (and therefore different responses) by virtue of the fact that they are trained on different training examples. This separation between iron and air also simplifies the representation, since it means that any boundaries near a point are always the same, either from air to iron for the network used to mesh air, or vice versa for the other network.

In principle this sub-division of a neural network implementation can be based on any

binary (or even discrete) input to the network, although there is a point where the number of networks becomes unmanageable.

3.5 Guidelines for Input and Output Representation

In the first section of this chapter the question was raised: should representations for the input (or output) favour simple encoding in many nodes or complex encoding in few nodes. The results presented in the previous sections indicate that the answer to this question is different for the input and the output. For the input, a complex encoding in as few nodes as possible gave the best results. The output, on the other hand, gave much better results when it was divided between several nodes. The following subsections explain why these rules should apply in the general case to other applications.

3.5.1 Theoretical Justification

In retrospect the results just stated are logical, given the equivalence between a neural network and a modelling function. First of all the results for the input representation are discussed. When viewed as a modelling function, the property of generalization is equivalent to interpolation between examples. The first implication of this was already mentioned earlier in this chapter, that similar inputs should correspond to similar outputs. There is, however another implication: the training examples must populate the input space *densely* enough so that the interpolation will not break down between examples. As the number of dimensions of the input space is increased (by increasing the number of input nodes) the density of examples *decreases*. This can be seen by imagining that the examples are distributed on a regular grid in the input space; even with only eight inputs it takes 6561 examples to populate a grid with only three vertices in each dimension (of course, in many cases the inputs are correlated in some way, so the actual dimensionality of the input points

is smaller than the number of inputs, but how obvious this correlation is also depends on the representation). Fewer examples per dimension means interpolation is more risky. This is the reason, then, that fewer input nodes improve the performance of the neural network: fewer nodes mean the input space is more densely populated (for the same number of training examples).

The opposite is true for the output. If a single output node is taken in isolation, then interpolation is simplest if the node takes on only two values, instead of a continuous range of values. The binary output node has only to tune itself to *recognize* certain features of the output. Also, a simple output encoding typically has an error tolerance that is much larger than the error tolerance of continuous output nodes. In addition, training is faster, since the larger error coupled with more outputs means that the gradient descent step size at each iteration is larger. This explains why a simpler encoding scheme with many outputs improves the performance of the neural network.

3.5.2 Rules for Representation

The reasoning presented in the previous subsection is far from rigorous, but it does give some insight into how input and output representations should be chosen. The result is the following set of rules for input and output representation.

This first rule applies equally to input and output representations:

1) The input and output should be encoded so that similar inputs correspond to similar outputs. Here "similar" can be taken as a Euclidean distance measure, for example. Expressed another way, if a collection of outputs are all in the same neighbourhood in the output space, then the inputs which give rise to these outputs should also be in the same neighbourhood in the input space. The inverse is also

true, i.e., if a collection of inputs are all in the same neighbourhood then the responses of the network to these inputs should be similar.

This rule applies only to the input representation:

2) When encoding the input, preference should be given to increasing the complexity of the encoding in favour of a reduced number of input nodes, providing the encoding still respects the first rule.

Finally this rule applies only to the output representation:

3) When encoding the output, preference should be given to increasing the number of output nodes in favour of a reduction in the complexity, with the optimum being many binary output nodes.

3.6 Summary

This chapter describes the representation used for the input and output of the neural network for the mesh generation application. Also presented is a robust encoding scheme for continuous output values in general, and some general rules for representing input and output in other neural network applications.

Chapter 4
Generating Ideal Meshes

As mentioned in the previous chapter, the neural network acquires its knowledge from training examples. In this application the training examples are derived from ideal meshes of representative magnetic devices. This chapter explains how these ideal meshes are generated.

4.1 Computing Ideal Element Sizes

An ideal mesh is defined here to be a mesh in which the error is distributed uniformly throughout the mesh. The error is the difference between the solution computed on the mesh and the exact (theoretical) solution. The error is uniformly distributed if the total error of each element (measured using an appropriate norm, and possibly normalized by area) is approximately equal throughout the mesh.

As mentioned in the literature survey, most of the research in the field has been in the context of adaptive solvers, and has centered around the formulation of an *a posteriori error measure*. The a posteriori error measure determines the error of a solution computed with a trial mesh, but without knowledge of the exact solution [Babuska, 1986]. This error measure can be used in an adaptive solver to determine where the mesh should be refined to improve the solution. Since this error measure is based only on the inexact solution, reliability can be difficult to obtain.

In the application discussed here, however, the exact solution *is* known, or at least a very close approximation to it can be computed. The difference is due to the fact that the neural network only needs the ideal mesh for the training examples. Once the network is trained, the mesh density for a magnetic device is computed based only on the input geometry and material properties. Therefore the computational cost of the solution is irrelevant.

The algorithm presented here is based on the availability of the exact solution. The algorithm also takes advantage of the fact that the neural network only requires the mesh *density* at specified points, and not an actual mesh of the geometric domain. The next section presents the details of this algorithm.

4.1.1 The Sizing Algorithm

The algorithm used to compute the sizes is given below in outline form. The remainder of this section clarifies the flow of the algorithm, and discusses the simplifications and assumptions implicit in the algorithm.

1) Compute the "exact" solution to the magnetic device.

2) Pick a value for the desired error level in the mesh. This error level is as measured with respect to the exact solution, and the elements will be sized so they all have this same error.

3) At each point compute the ideal element size as follows:

   3.1) Pick a starting guess for the size that an element should have at this point.

   3.2) Generate a trial element of this size centered at this point (i.e. compute the coordinates of the vertices of the element).

   3.3) Compute the error that this element would have if it were included as part of a mesh (see below).

   3.4) Compare the resulting error to the desired error.

   3.5) Increase the size if the error is less than the desired error and decrease the size if the error is greater than the desired error.

3.6) Repeat steps (3.2) to (3.5) until the error converges to the desired error.

4.1.2 Explaining the Sizing Algorithm

In step (1) the "exact" solution can itself be computed using finite elements, as long as an expert ensures that the mesh will generate a good solution. Recall that the cost of computation is not a factor at this stage, since this step is only required to train the neural network. Therefore the mesh used to generate the "exact" solution can over-discretize the magnetic device, and a high order solver can be used to maximize the accuracy of the computed solution.

Step (2) requires selecting a value for the global error. This number will determine how large the elements actually are, although the *relative* sizes of the elements remain approximately the same for different errors. Also since this is an *absolute* measure of the error its value cannot be set in advance; i.e., the global error depends on the solution to the magnetic device. In practice this value is chosen to generate reasonable element sizes.

Step (3) applies the same procedure to every sample point. The algorithm is basically a root finding algorithm applied to the element error as a function of size. This root finding algorithm determines the size of the element which results in the desired error specified in step (2). The details of the root finding algorithm were taken from Section 9.3 of [Press, 1988]. Note that computing the sizes in this manner implies that the element sizes are obtained *independently*. In fact the trial elements never form part of an actual mesh. This will be clarified in the following.

Step (3.1) sets the starting guess for the root finding algorithm.

Step (3.2) generates an element centered at the current sample point and of the specified size. Since the orientation of the element is not specified, and to make

things as symmetric as possible, the element shape was taken to be a hexagon. So this step actually computes the coordinates of the six vertices of a hexagon of the specified size centered at the current sample point.

Step (3.3) computes the error of the trial element. The mathematical details behind this step are given in Appendix I. In essence, the error is computed as follows:

3.3.1) Perform a least squares fit of the (linear) trial element to the exact solution.

3.3.2) Integrate (over the domain of the trial element) the square of the difference between the linear solution on the trial element and the "exact" solution.

3.3.3) The trial element error is obtained by scaling this result by the area of the trial element.

Step (3.3.1) hides the key assumption behind this approach to computing the ideal mesh. Here it is assumed that the least squares fit of the trial element to the exact solution is equivalent to using this element in a trial mesh (notwithstanding its unusual shape), and then computing a trial solution to the magnetic device using this mesh. Although this assumption may introduce errors in the sizes computed for the "ideal" elements, these errors are negligible when compared to, for example, the neural network output error (even at the completion of the training phase). This method has the advantage of being simple, consistent and reliable. Simple in that no solutions on trial meshes are required, consistent in that the sizes of the elements do not vary arbitrarily, but rather are determined precisely by the "exact" solution, and reliable in that this method will not accidentally over-size an element (as can happen in the adaptive case).

Step (3.3.2) computes the total squared error over the element. This is scaled in step (3.3.3) by the area of the element to give the mean squared error of the element. During the development of this algorithm, several variations on computing the error

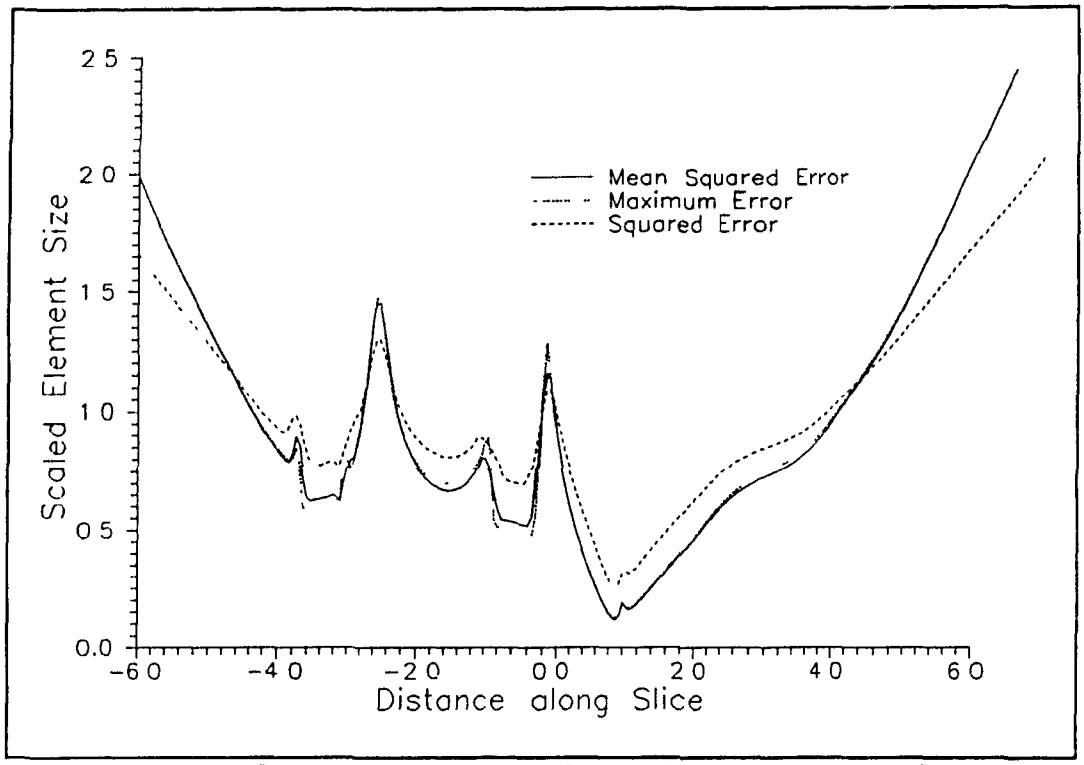

Figure 8 - Comparison of Error Norms along a Horizontal Slice

were compared (see Figure 8). The maximum error is obtained by taking the maximum difference between the solution on the trial element and the "exact" solution. The squared error is not scaled by area. Without scaling by area, the squared error gives skewed results since the algorithm determines element sizes such that each element contributes the same amount to the total error, irrespective of size. This has the effect of concentrating the error in the finely meshed regions, which is precisely where the highest accuracy is usually required.

The maximum error does not need to be scaled since in this case the algorithm determines element sizes such that the solution is bounded everywhere by the specified global error. In fact the maximum and mean squared methods give practically identical values for the sizes of the "ideal" elements (within a scale factor). In this case the mean squared error is cheaper to compute (the algorithm is already doing a least squares fit, so the squared error comes for free), therefore it is the one

used in this algorithm.

Finally steps (3.4) to (3.6) complete the steps in the root finding procedure.

4.1.3 Special Case: Material Boundaries

The algorithm outlined in the previous subsection needs to be modified in the special case of material boundaries. In step (3.2) the trial element cannot be allowed to cross material boundaries, since the solution is often discontinuous across material boundaries. Instead, in this case the trial element is *clipped* against material boundaries. To avoid distorting the element size when a large part of the element is clipped away, the size of the ideal trial element is measured by the square root of its area.

4.1.4 Scaling Properties of the Error Criterion

Figure 9 shows the scaling properties of the error criterion. The data is taken from the C-core device (see Chapter 5), by evaluating the error at different points. Interestingly enough, the relationship between the error and trial element area is almost exactly linear. This means that there is a certain degree of freedom allowed in choosing the desired error (in step (2) of the algorithm). Because of the linear relationship, the effect of choosing a larger or smaller desired error can be approximated by simply multiplying all the element sizes by a single scale factor. Even more important, it means that the same network can be used to compute the mesh density for different final errors.

For example, instead of supplying an absolute error level (which can be difficult if the magnitude of the solution is unknown in the first place!) the user of this package may want to specify the total number of elements in the final mesh. The scale factor required to do this can be determined by first integrating (over the domain of the

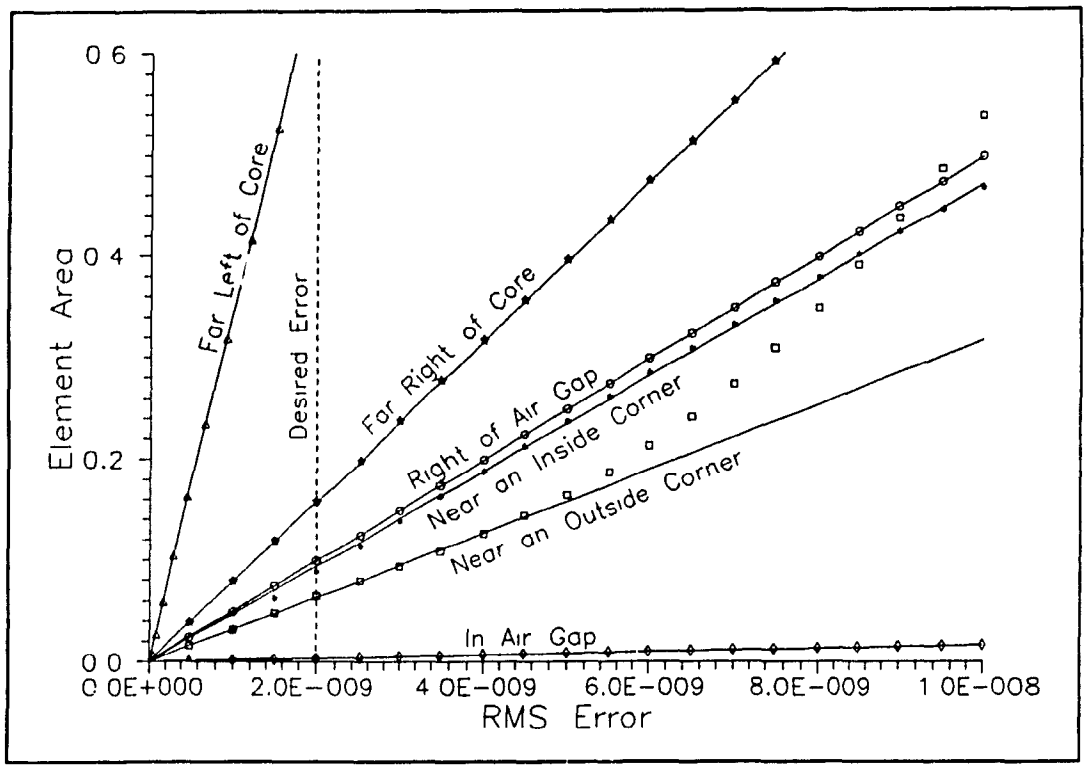

Figure 9 - Scaling Properties of the Error Criterion

problem) the inverse of the mesh density squared (the number of elements per unit area). The scale factor is found by dividing this result by the specified number of elements.

The linear relationship also makes the network response meaningful even if the magnitude of the solution for a given device is very different from the magnitude of the solution of the device used to generate the training examples. It may happen that the desired error specified for the example device is meaningless in terms of the solution of the given device (recall that the desired error is an *absolute* number). However, because of the linear relationship, the output of the neural network still prescribes element sizes which are appropriate for the device. In effect, the neural network is learning the relative error, even if it is the absolute error that is used to train the network.

4.2 Encoding Element Size to Train the Network

Once the ideal element size has been determined, the encoding process is straightforward. The encoding scheme which is used to represent the output was presented in chapter 3. However before each ideal size value is encoded it must be scaled by the geometric scale factor used to scale the corresponding input (this scale factor is different for each point). This step is necessary to maintain consistency with the scaling property of the mesh density function, as presented in Chapter 3. One possible problem with this step is the case when a point falls on a corner, since in this case the scale factor is infinite. How this case is treated leads to artifacts, which is the subject of the next section.

4.3 Artifacts of the Encoding Process

As mentioned previously, forcing the target sizes into bins produces certain artifacts when the bin representation is converted back to sizes. The main artifact is due to the squashing that takes place when a size value is larger than the value of the largest bin. These extra-large values correspond to inputs with a very small geometric scale factor, and compromise typically 0.1% of all the size values in a uniformly sampled input domain. Since the current input encoding scales by the distance to the nearest corner, the effect of squashing is to reduce the size of the elements near corners. Since corners are usually meshed very densely anyway, this artifact is not detrimental to the performance of the system. The second artifact is simply the error introduced by the discretization process. This error is inversely proportional to the number of output nodes. However since the required resolution of the mesh density is not large, this error is sufficiently small with only ten output nodes (5% discretization error). Also the smoothing which takes place in the decoding process reduces this error considerably.

4.4 Summary

This chapter explains in detail the algorithm used to generate the training examples for the mesh discretization application. Chapter 5 describes how this algorithm was used to generate examples based on two different devices.

Chapter 5
Simulations

The results presented in the previous chapters were found through a series of software experiments. These results include the representation of the input and output, and the network topology. The software experiments involved training a simulated neural network and evaluating the performance of the trained network. This chapter describes how these simulations were set up, as well as some additional results.

5.1 Objectives of the Simulations

The main objective of the simulations was to determine the configuration of the neural network. Specifically, the simulations allowed the evaluation and comparison of different input and output representations, and of different network topologies. Strictly speaking, these two objectives satisfy the requirements of the meshing system. However the possible *methods* used to train the network also required evaluation, so additional objectives were defined: to determine which training algorithm gives the best results (gradient descent, or conjugate gradient), and to determine values for the training parameters in the case of the backpropagation algorithm (the momentum parameter and the step-size parameter).

The following sections describe how these objectives were achieved, starting with the training objectives. Along the way additional details are included to clarify the simulation results.

5.2 Training

The mathematical backround of the backpropagation algorithm was presented in Chapter 2. However, the algorithm leaves several details unspecified, these are the length of the training session, the values for the learning parameters, how the initial

weights are set, and the order of presentation of the examples. The following paragraphs tackle each of these details in turn.

5.2.1 Error Criterion and Termination Condition

During the training process, the performance of the network can be monitored using the root mean square (rms) of the network output error with respect to the training examples. This is cheap to compute since the squared error is computed at each step as part of the backpropagation algorithm. In the application described here, there is also a more significant error: the *decoded* output error. This is the rms of the difference between the decoded output of the neural network and the target size values before encoding. Monitoring this error gives a indication of the performance of the neural network on the training examples.

However monitoring the error with respect to the training examples is not sufficient to gauge a more significant ability: the ability to generalize. Therefore to properly monitor the network during training it is necessary to measure the error of a set of test examples which are distinct from the training examples. In this case a small complication arises if the test examples come from a different magnetic device. In this case the element size can differ by a scale factor. To account for this, this scale factor must be taken into account. This is accomplished by minimizing the mean squared error with respect to the scale factor. The result is:

$$\begin{aligned} \langle e \rangle &= \langle (t - ky)^2 \rangle \\ &= \langle t^2 \rangle - 2k\langle ty \rangle + k^2\langle y^2 \rangle \\ \frac{\partial \langle e \rangle}{\partial k} &= -2\langle ty \rangle + 2k\langle y^2 \rangle = 0 \\ \therefore k &= \frac{\langle ty \rangle}{\langle y^2 \rangle} \\ \therefore \langle e \rangle &= \langle t^2 \rangle - \langle ty \rangle^2 / \langle y^2 \rangle \end{aligned} \qquad (1)$$

where $\langle\rangle$ denotes expectation value or, more precisely, the ensemble average (over

the examples), e is the error, y is the decoded output of the neural network, and t is the target value.

As training progresses, the error of the training set always shows an average decrease. However the error of the test set at first decreases with the training error, but after a certain point it will stop decreasing and eventually start to increase. This phenomena is known as *overtraining* [Hecht-Nielsen, 1990]. For optimum performance of the neural network in the general case, it is important to stop the training as soon as the test set error stops decreasing. After this point the neural network is starting to pick up on features particular to the training set, and which do not hold in general. The simulations were run to 500,000 iterations, with the best weights on the test examples saved along the way.

5.2.2 Training Parameters

The backpropagation algorithm has two free parameters: α, the momentum parameter, and β, the step-size parameter. The step-size determines how large each step is relative to the gradient of the error. The momentum parameter allows the algorithm to escape local minima in some cases. The value of α was taken from the literature ([Caudill, 1989] and [Rumelhart, 1986]), and was set to 0.9 for practically all the simulations. A few informal simulations with different values of α hinted that there are no significant improvements to be had for small deviations from this value, and larger variations only make the convergence worse (slower convergence to a larger final error).

Much more effort was taken into choosing an appropriate value for β, the step-size parameter. This was motivated especially by simulations with convergence properties as shown in Figure 10. This graph shows the error for two learning sessions. Each learning session started with the same initial weights, and used the same examples. The only difference between them was the step-size: the learning session that

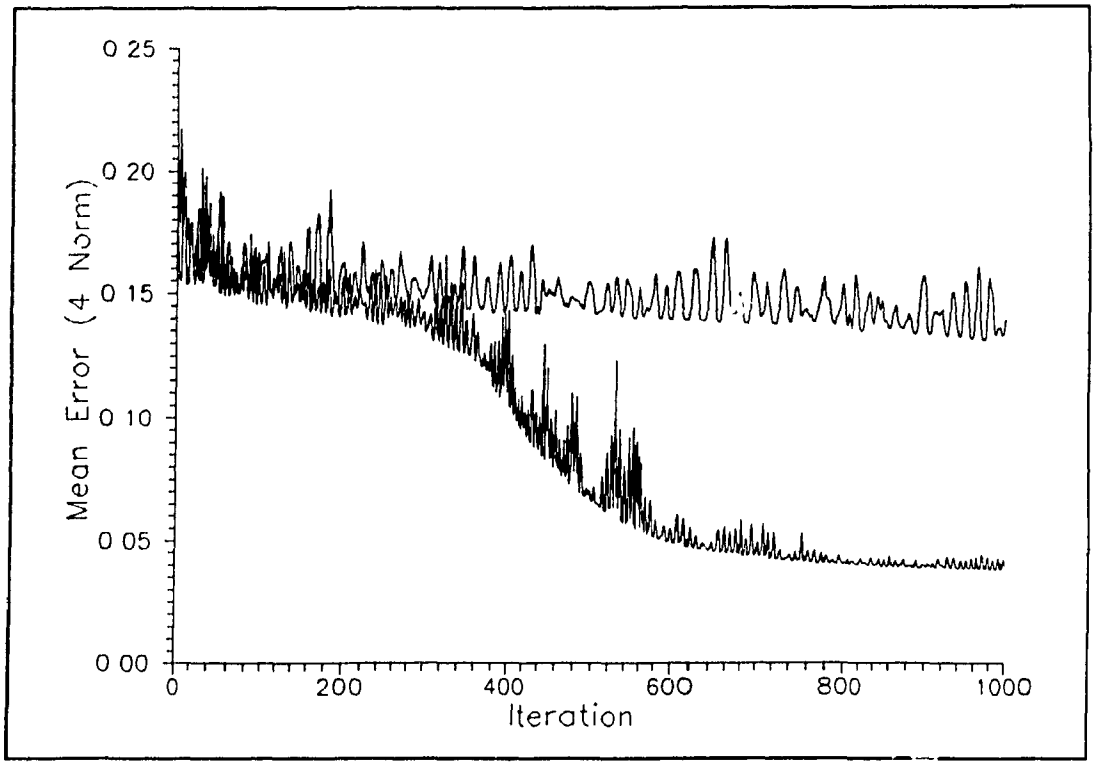

Figure 10 - Convergence of Two Identical Networks

converged to the lower error used a step-size that was effectively half that of the other session.

The actual step-size taken at any iteration is actually the product of β with the decaying average of the gradient of the error w.r.t. the weights. If this gradient is slowly varying, the step-size is:

$$\Delta w^{(k)} = \beta \sum_{j=0}^{\infty} \alpha^{j} \frac{\partial E^{(k-j)}}{\partial w} \sim \frac{\beta}{1-\alpha} \frac{\partial E^{(k)}}{\partial w} \qquad (2)$$

where the superscript refers to the training iteration (and not the layer). The literature [Caudill, 1989] states that β should between zero and one. In fact it is the product $\beta/(1-\alpha)$ that should be less than one. Choosing too large a step-size results in a phenomena known as network paralysis. Instead of converging to a low error state, the network reaches a state where all the outputs are saturated at either

one or zero. In this state the error is large, but learning depends on the *gradient* of the error, which in this case is small (because the outputs are saturated). Therefore the weights remain virtually constant, and the network response never improves. To avoid this the step-size must be reduced, but again the optimum value is unknown. The training session that produced the final network used $\beta=0.03$, so that the product $\beta/(1-\alpha)=0.3$.

It is training behaviour like that shown in Figure 10 that complicates the evaluation of different learning experiments. When one learning session results in a network with improved performance, it is not clear whether this is due to a superior input representation, for example, or simply a lucky choice for one of the learning parameters.

5.2.3 Initial Weights

The literature suggests small random weights. Some informal experiments revealed that random weights, uniformly distributed between $-\frac{1}{2}$ and $\frac{1}{2}$ are as good a choice as any. Note that the weights cannot be initialized to zero, because in this case backpropagation of the error will stop at the last hidden layer, and weights on connections between previous layers will never be modified. Most of the experiments used the same initial weights (if the topologies were the same) to make evaluations of the performance of different networks as consistent as possible. A few experiments were tried using the same learning parameters and training examples, but starting with different weights, to see how much influence this had on the final performance. The performance of the trained networks was similar in this case, but it is still a possibility that this could also have a significant effect on network performance.

5.2.4 Order of Presentation of the Training Examples

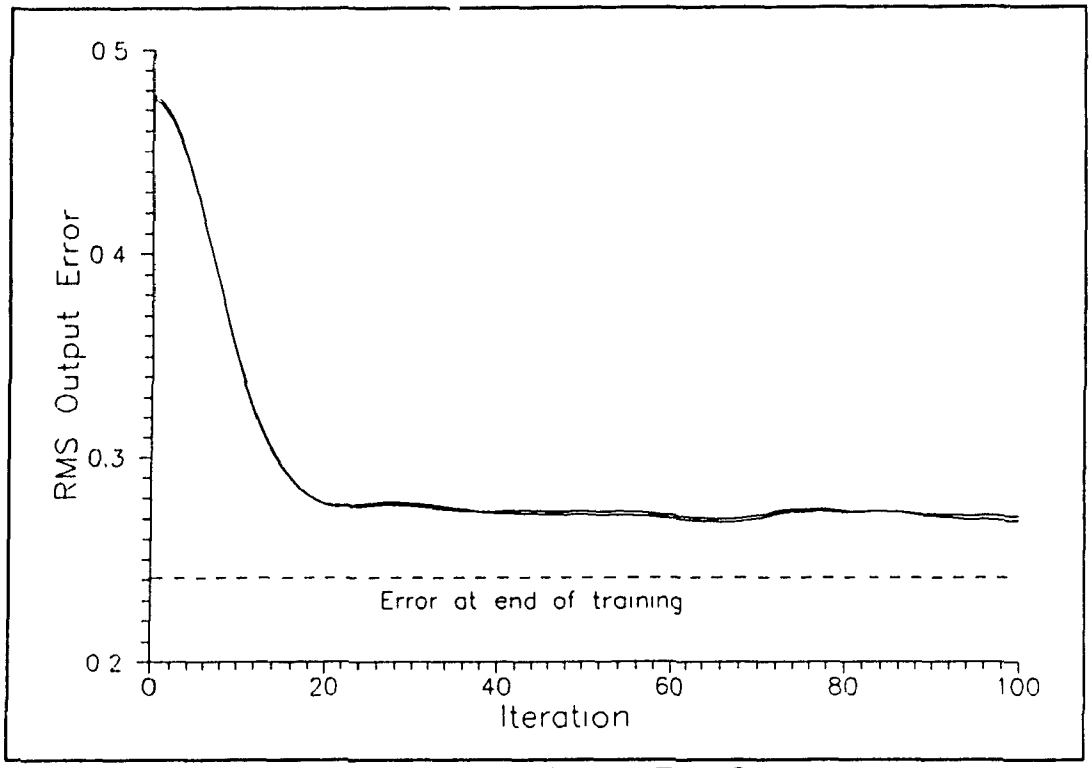

Figure 11 - Learning During First Century

Figure 11 shows the error (before decoding) for the first 100 iterations of a learning session. Note that a large part of the training appears to occurs during the presentation of the first 20 examples. It is reasonable to assume, then, that *which* examples come first will have a significant impact on the final result. An effort was made to slow down learning (by reducing β) so that more examples would influence the initial learning, but then the learning failed altogether. In the end the order of presentation was left random. Trying different random orders did not seem to have too much of an influence, although this does not mean that a carefully selected order would not improve learning.

There is another interpretation for the graph in Figure 11. The rms error of the *decoded* output does not show such a sharp decrease. Therefore it appears that the

rms error of the test examples remains relatively constant during the training session (after the first 20 iterations); it is the rms error of the decoded output which decreases. So why is the rms error being minimized in the first place? That is a good question! (but beyond the scope of this thesis).

5.3 Training Examples

The generation of training examples from a representative magnetic device and its "exact" solution was described in Chapter 4. This section describes how the magnetic devices which formed the basis for the examples were chosen, as well as some of the details of the example generation phase.

5.3.1 Selection of Representative Magnetic Devices

The representative magnetic devices which form the basis for the training examples were chosen with the help of an expert in magnetic device analysis: Prof J. S. McFee. Based on his suggestions the two devices shown in Figures 12 and 13 were created. The first, a C-core inductor, is one of the most common magnetic devices, yet captures many of the basic features of these devices. On the suggestion of Prof D. A. Lowther, the air gap of this device, which usually has a constant width, is *bevelled* so that the magnetic field in the gap is non-uniform. This increases the complexity of the distribution of the element sizes in the gap region. In addition, it increases the sharpness of the corners at the smallest part of the gap, further complicating the element density distribution because of the resulting singularities in the solution. The device in Figure 12 is an E-core inductor. The right gap is bevelled similar to that of the C-core, but in the opposite direction. The left gap has the two poles shifted, again to increase the non-uniformity of the field in the gap. This pole structure, with the two poles not quite aligned, is also important because this feature is found in many magnetic devices, for example motors. The properties of both devices are

modelled using linear materials.

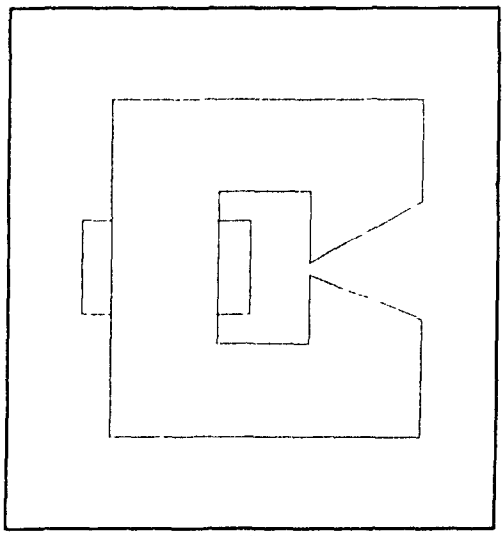

Figure 12 - C-Core Geometry

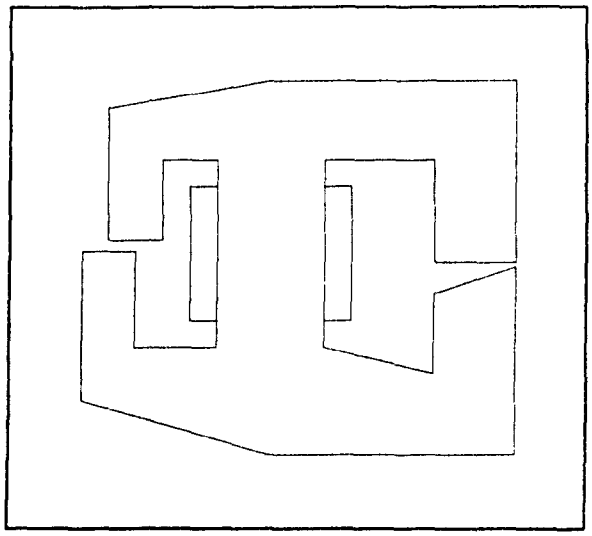

Figure 13 - E-Core Geometry

During most of the training sessions, the examples derived from the E-core were used to train the network, while the examples from the C-core were used as test examples to monitor the performance of the network. Training sessions with the roles reversed resulted in comparable performance.

5.3.2 Solving for the "Exact" Solution

The "exact" solution to these devices was generated using a finite element solver. Several steps were taken to ensure that the solutions were sufficiently accurate. First of all the device domain was *over-discretized* (using more elements than strictly necessary) and in critical areas (e.g. the air gap) an expert touched up the final mesh. The solutions were then computed with fourth order elements, using a standard conjugate gradient solver. Finally the resulting solutions were inspected visually to verify that the results were consistent with the expected solution.

5.3.3 Generation of the Examples

From the "exact" solution, examples are generated using the algorithm presented in Chapter 4. This step requires specifying the location of the trial elements in the domain of the device, as well as the desired error for the trial elements.

The trial elements were positioned at the vertices of a regular grid covering the domain of the device. Because of the earlier network topology, this grid was triangular instead of cartesian. Although a carefully chosen, non-uniform distribution may have improved the speed of training, this possible gain did not out-weigh the difficulty of choosing the points. In fact with a uniform distribution of points, the distribution of element sizes was also roughly uniform (see Figure 3 in Chapter 3)

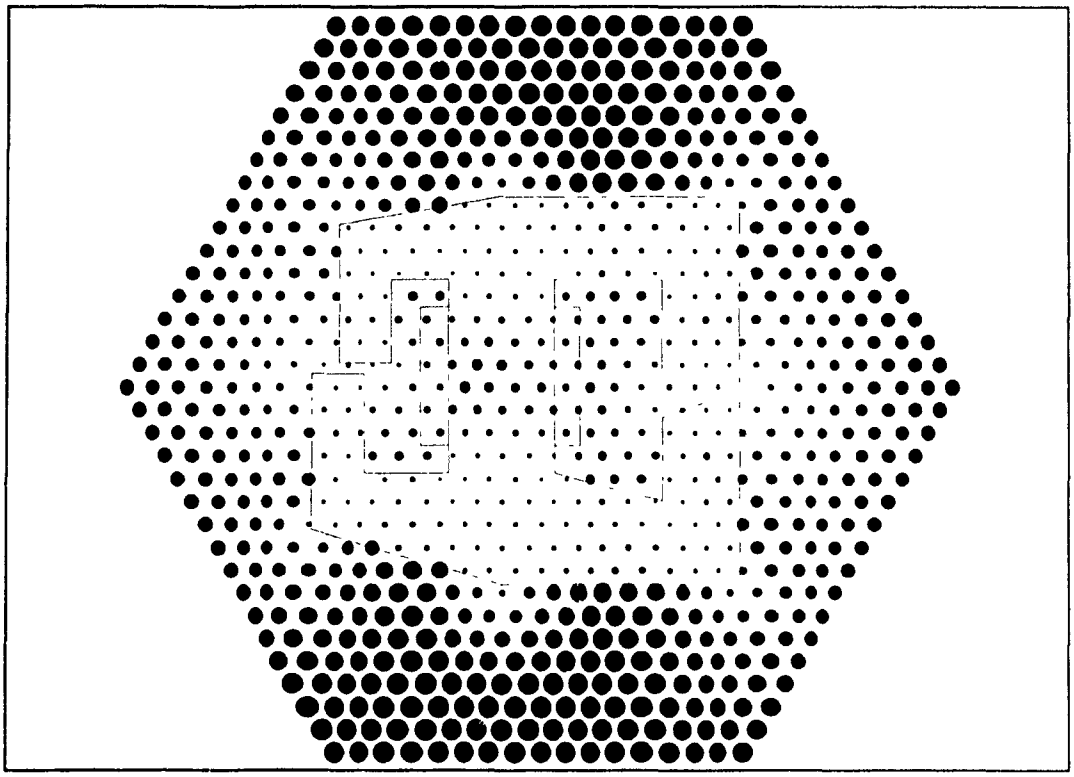

Figure 14 - Target Element Sizes for the E-Core

The desired error of the elements was chosen by first evaluating the element error at a few critical points for different size trial elements. The desired error was then

chosen to give reasonable sizes for the elements. In fact since the relationship between desired size and element area is approximately linear, this step is not critical.

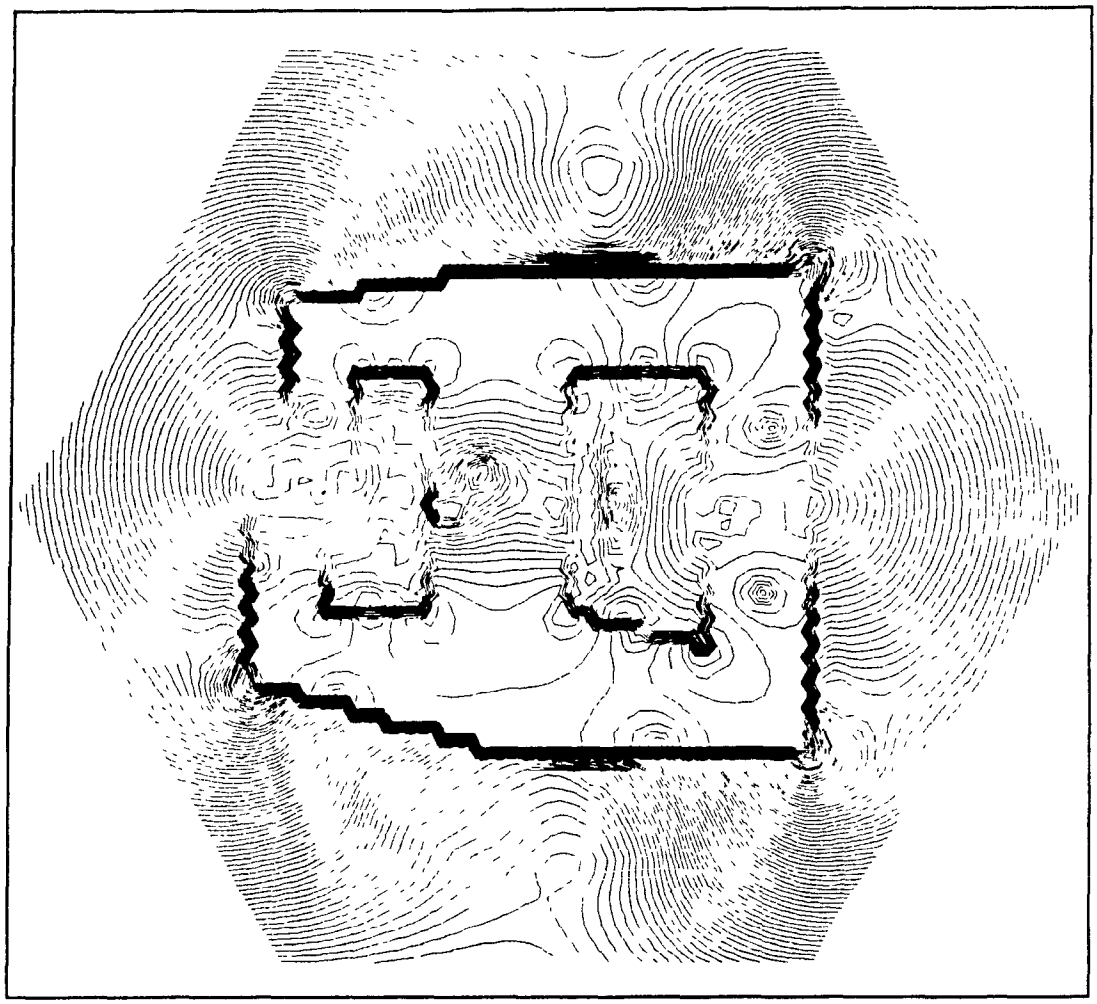

Figure 15 - Contour Plot of the Target Element Sizes for the E-Core

For the E-core training examples, the density of the grid was chosen to give 3997 examples. This many training examples are necessary since the input values of the examples should sample the input space as densely as possible. For the test examples from the C-core, the grid was chosen to give only 547 examples, since complete coverage is not necessary to evaluate the performance of the network. Figure 14 shows the target element sizes for the E-core device. The area of each spot is

proportional to the size an element should have there. Figure 15 shows a contour plot of the ideal element sizes. There are a total of 75 lines, so a variation of 10% is equivalent to 7.5 lines.

5.4 The Neural Network System

5.4.1 Network Topology

The network topology was chosen based on only very approximate guesswork. The analysis in [Lippmann, 1987] holds only for threshold networks. More or less arbitrarily the first hidden layer was chosen to have 24 nodes, and the second hidden layer 18 nodes. Some experimentation was done to determine the optimum number of layers.

Table 2 - Error of Networks with 1, 2, and 3 Hidden Layers

Number of Hidden Layers	Number of Nodes in each Hidden Layer	RMS Decoded Error of Training Examples	RMS Decoded Error of Test Examples
1	24	7.0%	13.5%
1	45	6.4%	11.6%
2	20, 15	5.5%	13.3%
2	24, 18	5.0%	9.9%
3	24, 18, 10	5.1%	12.9%

Table 2 shows the error for the different simulations. Surprisingly the rms error increased with the addition of another hidden layer of 10 nodes. This can be accounted for by the fact that three layers gives the neural network too many degrees of freedom, so it can easily model the training examples, but in between the examples the interpolation is not very smooth. Conversely a single hidden layer of 24 nodes

performs significantly worse than the two layer network. To be fair the single layer network should have the same number of *weights* as the two layer network, which in this case would mean 45 nodes in the single hidden layer. This network, while better than the one with only 24 nodes, still has a significantly larger error than the two hidden layer network. These results give a ballpark estimate of the number of layers required by the system to learn the mapping.

5.4.2 Performance

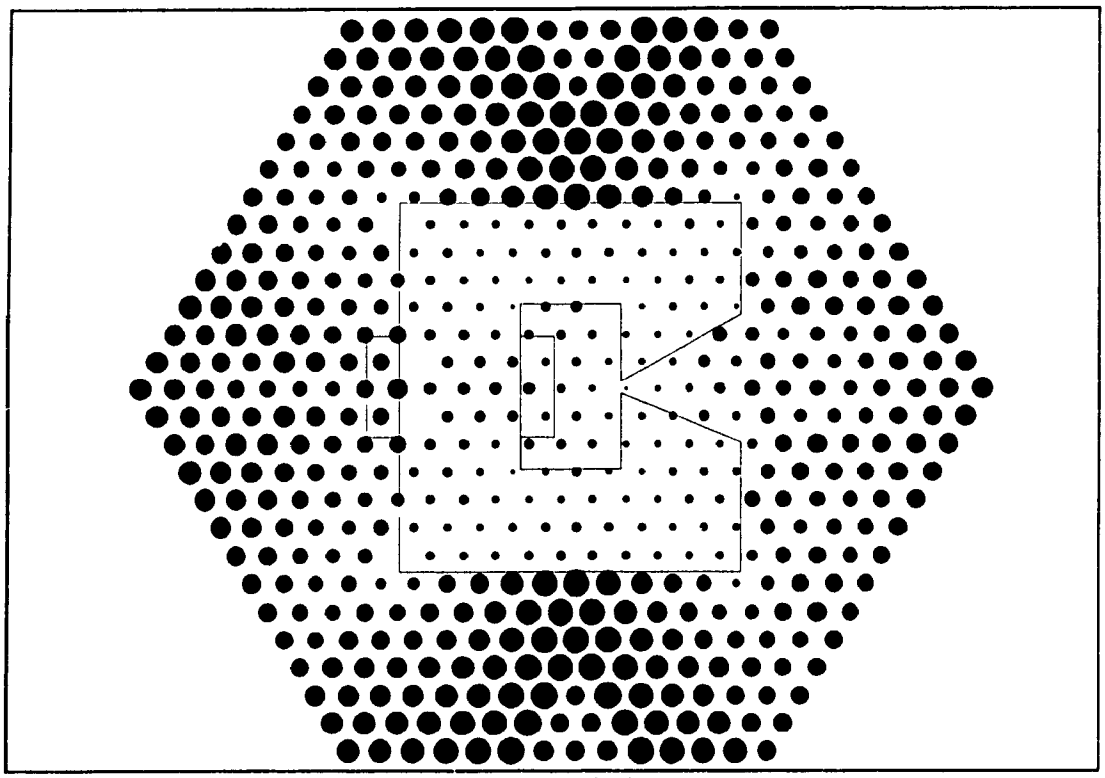

Figure 16 - Element Sizes for the C-Core from Neural Network

The performance can be judged in two ways. First of all the decoded output of the neural network can be compared with ideal element sizes. Figure 16 shows the sizes recomended by the neural network for the C-core, using the same format as Figure 14.

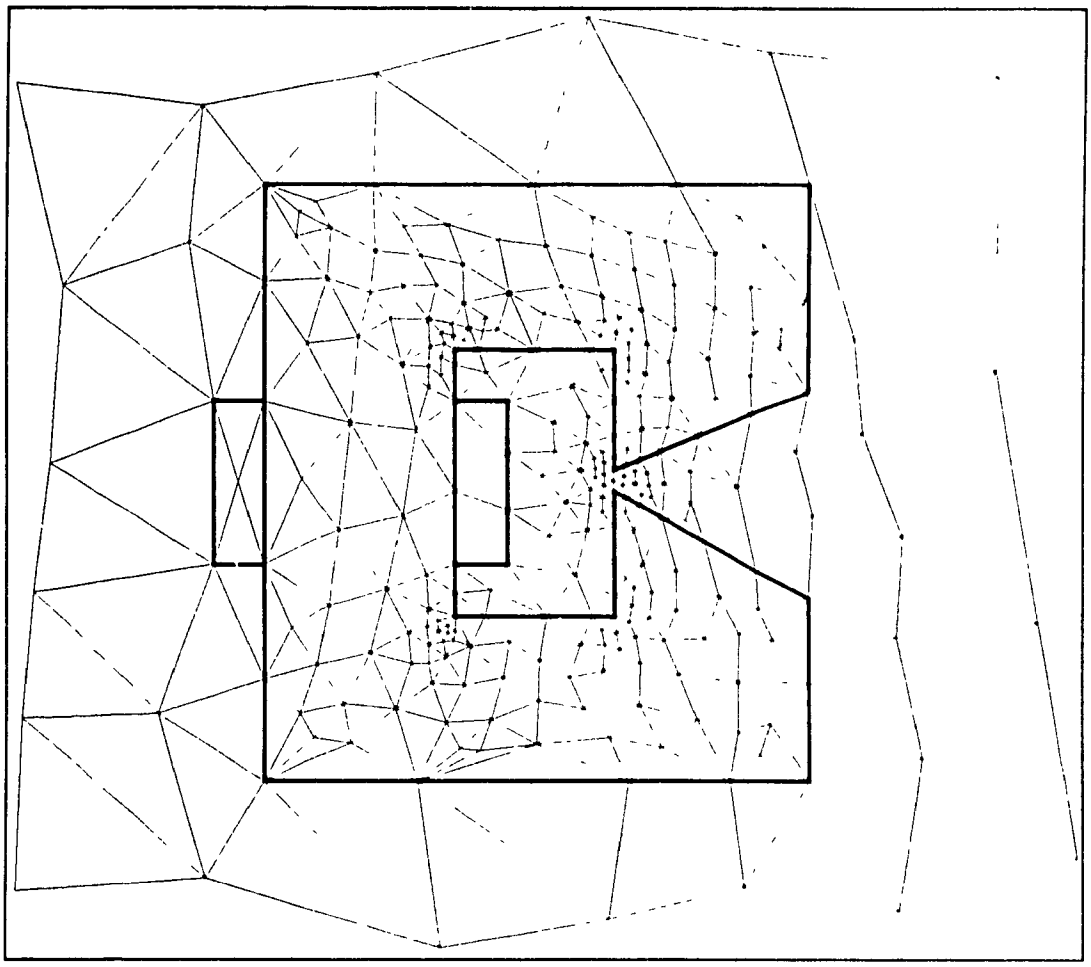

Figure 17 - Mesh for C-Core based on Neural Network

And secondly the resulting mesh can be judged on its own merit. Figure 17 shows the final mesh for the C-core device. The density for this mesh was computed using the weights obtained by training the network on the E-core device. The mesh was then generated using the TRIA2D mesh generator of [Hétu, 1990], which uses density information to direct node placement. (The fact that this mesh is not Delaunay is the fault of the mesh generator, and is not related to the mesh density). According to experts, this mesh is close to what they would produce if asked to mesh the device themselves. There is no doubt that the network is contributing valuable information to the meshing process. If this mesh is not optimal enough, then a few adaptive steps would finish the process with a significant saving in total computational cost

5.5 Summary

The purpose of this chapter is to clarify the procedures used to obtain the results which are presented in earlier chapters. This chapter should also highlight the lack of rigour in the neural network field at the present time. As much effort as possible was put into the software experiments so that the conclusions drawn from them had a reasonable chance of being valid. The system thus far still has more an aura of a collection of patches than of a logically evolved structure. However, in a sense, this is engineering and not science!

Chapter 6
Conclusion and Future Work

This chapter outlines the future work spawned by this mesh generation application, as well as noting what loose ends need to be tied up to complete the system. The summary briefly states the initial premise of the thesis and the original work accomplished in pursuing this premise.

6.1 Future Work

A great deal of work remains to be done in order to make this application useable. Because of the nature of neural networks, the basis in theory may never be completely rigorous, but more experimentation should clear up many issues.

6.1.1 Error Criterion

The error criterion described in Chapter 4 should be verified. To do this, a mesh of a device should be generated using the ideal density information. The solution on this mesh should then be compared to the "exact" solution by integrating for the mean squared error of each element in the mesh. If the scheme for determining ideal element size is working correctly, then each element should have the same mean squared error (i.e., the variance of this quantity should be small).

6.1.2 Input Representation

The representation for the input is currently a bit too arbitrary, and as it stands has a few problems. The first problem has to do with the scaling of the input and output (by distance to nearest corner). When a sample point is on a corner, then the scale factor becomes infinite. The second problem is the way material boundaries are treated. Because different networks are used on either side of the boundary, the

mesh density as computed by the system is *discontinuous* across the boundary. Some sort of smoothing would correct this, and could also make sure the mesh was graded properly. The third problem is more fundamental, and has to do with the fact that the representation assumes that the geometry can be approximated locally by a couple of line segments. This assumption is violated in the more general case where, for example, a curved surface is approximated with many short line segments. Finally it is still an open question as to how local the representation can be and still contain enough information to compute the mesh density.

In more general terms, the input representation should be evaluated for redundancy. This can be accomplished by training the network using an input representation which omits one input. In this way each input can be tested in turn to see how "necessary" it is.

6.1.3 Output Representation

The output representation is fairly sound, and in concept is probably near optimal for this network architecture. The number of "bins" was chosen arbitrarily, and this could be refined through further experimentation. More generally, the nature of the output makes it natural to examine *competitive* networks, where the outputs compete for the right to respond to a given input.

6.1.4 Training

The training method used in the learning process is unsatisfactory. While the performance of the network is acceptable, it seems likely that there exist more global minima in the error surface, and especially minima that would perform better at generalizing. The are several approaches to finding these minima.

1) First of all, in any approach a much larger number of examples is required.

The examples should come from many, very different, devices. Examples could include a C-core inductor, a stepping motor, a transformer, an actuator.

2) Different training algorithms could be tested. One possibility that seems especially attractive is *simulated annealing*. As mentioned in Chapter 2, this method can be slow to find a network with an acceptable error. However this method could be accelerated by using it in conjunction with the conjugate gradient method. In this hybrid scheme, the weight vector from a simulated annealing step would be refined to the nearest local minima in the error surface. This would help offset the primary disadvantage of the simulated annealing algorithm, which is its very slow convergence. Also the conjugate gradient method is very fast, so the additional cost would be minor.

Another alternative would use an explicit, nearest neighbour, interpolation scheme. The main disadvantage of this approach is the requirement that all the training examples be kept around to construct the local interpolants on demand. The advantage is that there is no training involved at all. One difficulty is in determining which of the nearest neighbours should be used to construct the local interpolant.

6.1.5 Evaluation

There is little doubt that the neural network approach is computationally more efficient that an adaptive system (its only competitor). However explicit computational cost and resource usage should be calculated for both systems. This would help convince sceptics of the utility of this approach.

6.1.6 Steps to a Working System

Finally several steps are necessary to turn the network into a complete system. A mesh generator working from density has to be adapted to use the output of the

neural network. Also the system should be extended to handle different materials and different types of problems (e.g., time harmonic, rotationally symmetric). This could most easily be accomplished by using a different network (essentially only different weights) for each type of problem.

Another major step is extending the system to handle non-linear materials. This problem may be too difficult for the neural network to handle directly (since knowing where the iron saturates is an essential part of the problem). However one approach would use a solution to the device computed on a very crude mesh (and therefore relatively cheap to compute). This crude solution could then be one of the inputs to the network. In fact this is similar to the expert who traces approximate flux lines to help determine where saturation occurs.

6.2 Summary

This thesis started by examining what contribution neural networks could make to automating engineering design and analysis, in particular finite elements. The idea is to emulate, and not surpass, the ability of human experts applying themselves to a certain task. The result is a working system that successfully applies neural networks to a previously unsolved problem in mesh generation.

References

[Babuska, 1986]

I. Babuska, O. C. Zienkiewicz, J. P. Gago, E. R. de A. Oliveira, Editors, *Accuracy Estimates and Adaptive Refinements in Finite Element Computations*, John Wiley, New York, 1986.

[Baehmann, 1987]

P. L. Baehmann, K. R. Grice, M. S. Shephard, S. L. Wittchen, M. A. Yerry, "Robust, Geometrically Based, Automatic Two-Dimesional Mesh Generation," *International Journal for Numerical Methods in Engineering* **24**, No. 6, pp. 1043-78, 1987

[Caudill, 1989]

M. Caudill, "Neural Networks Primer" (series), *AI EXPERT*, Part 1, December 1987, pp. 46-52; Part 2, February 1988, pp. 55-61; Part 3, June 1988, pp. 53-59; Part 4, August 1988, pp. 61-67; Part 5, November 1988, pp. 57-65; Part 6, February 1989, pp. 61-67; Part 7, May 1989, pp. 51-58; Part 8, August 1989, pp. 61-67.

[Carpenter, 1988]

G. A. Carpenter and S. Grossberg, "The ART of Adaptive Pattern Recognition by a Self-Organizing Neural Network," *Computer* **21**, March 1988, pp. 77-88.

[Fang, 1990]

M. Fang and G. Hausler, "Class of transforms invariant under shift, rotation and scaling," *Applied Optics* **29**, No. 5, pp. 704-8, February, 1990.

[Farmer, 1988]

J. D. Farmer and J. J. Sidorowich, "Exploiting Chaos to Predict the Future and Reduce Noise," in Y. C. Lee (ed.), *Evolution, Learning and Cognition*, World Scientific, New Jersey, 1988, pp. 277-330.

[Foley, 1990]
J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, "Computer Graphics," 2nd Edition, Addison-Wesley, Reading, Massachusetts, 1990.

[Fujita, 1988]
M. Fujita and M. Yamana, "Two-Dimensional Automatically Adaptive Finite-Element Mesh Generation," *IEEE Trans. on Magnetics*, **MAG-24**, No. 1, pp. 303-6, 1988.

[Fukushima, 1982]
K. Fukushima and S. Miyake, "Neocognitron: A New Algorithm for Pattern Recognition Tolerant of Deformations and Shifts in Position," *Pattern Recognition* **15**, No. 6, 1982, pp. 455-69.

[Gallant, 1988]
S. I. Gallant, "Connectionist Expert Systems," *Communications of the ACM* **31**, No. 2, February 1988, pp. 152-69.

[Grossberg, 1983]
S. Grossberg, *Studies of Mind and Brain*, Reidel, Boston, 1982.

[Hecht-Nielsen, 1987]
R. Hecht-Nielsen, "Neurocomputer Applications," *AFIPS Conference Proceedings* **56**, 1987, pp. 239-44.

[Hecht-Nielsen, 1990]
R. Hecht-Nielsen, *Neurocomputing*, Addison-Wesley, Reading, Massachusetts, 1990.

[Hétu, 1990]
J. F. Hétu, D. Pelletier, "Adaptive Remeshing for Incompressible Viscous Flows," AIAA Paper 90-1604, *AIAA 21st Fluid Dynamics, Plasma Dynamics and Lasers Conference*, June 18-20, 1990 Seattle, Washington.

[Hinton, 1990]
G. E. Hinton and S. Becker, "An Unsupervised Learning Procedure that Discovers Surfaces in Random-dot Stereograms," to appear in *IJCNN*, Washington D.C., January 1990.

[Jin, 1990]
H. Jin, and N. E. Wiberg, "2-Dimensional Mesh Generation, Adaptive Remeshing, and Refinement," *International Journal for Numerical Methods in Engineering* **29**, No. 7, pp. 1501-26, 1990.

[Kohonen, 1984]
T. Kohonen, *Self-Organization and Associative Memory*, Springer-Verlag, 1984.

[Lapedes, 1988]
A. Lapedes and R. Farber, "How Neural Nets Work," in Y. C. Lee (ed.), *Evolution, Learning and Cognition*, World Scientific, New Jersey, 1988, pp 331-46.

[Lejeune, 1989]
C. Lejeune and Y. Sheng, "Invariant Pattern Recognition using Back-Propagation Neural Network and Fourier-Mellin Filters," *Canadian Conference on Electrical and Computer Engineering*, Montreal, Quebec, September 1989, pp. 417-19.

[Linsker, 1988]
R. Linsker, "Self-Organization in a Perceptual Network," *Computer* **21**, March 1988, pp. 105-17.

[Lippmann, 1987]
R. P. Lippmann, "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine*, April 1987, pp. 4-22.

[Lowther, 1986]
D. A. Lowther, P. P. Silvester, *Computer-Aided Design in Magnetics*, Springer-

Verlag (New-York), 1986.

[Montana, 1989]
D. J. Montana and L. Davis, "Training Feedforward Neural Nets Using Genetic Algorithms," *IJCAI '89*, pp. 762-7.

[Press, 1988]
W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, *Numerical Recipies in C, The Art of Scientific Computing*, Cambridge University Press, New York, 1988.

[Reichert, 1990]
K. Reichert, J. Skoczylas, T. Tarnhuvud, "Automatic Mesh Generation Based on Expert-System-Methods," (private communication) 1990.

[Rumelhart, 1986]
D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning Internal Representations by Error Propagation," in D. E. Rumelhart, J. L. McClelland and the PDP research group (Eds.), *Parallel Distributed Processing: Exploarations in the Microstructure of Cognition, Vol 1: Foundations*, MIT Press, Cambridge, 1987, pp. 318-62.

[Sejnowski, 1987]
T. J. Sejnowski and C. R. Rosenberg, "Parallel Networks that Learn to Pronounce English Text," *Complex Systems* **1**, 1987, pp. 145-68.

[Sompolinsky, 1988]
H. Sompolinsky, "Statistical Mechanics of Neural Networks," *Physics Today*, December 1988, pp. 70-80.

[Soucek, 1988]
B. Soucek and M. Soucek, *Neural and Massively Parallel Computers*, John Wiley & Sons, New York, 1988.

[Swaine, 1989]
M. Swaine, "Programming Paradigms" (column), *Dr. Dobb's Journal*, No. 153, July 1989, pp. 100-110; No. 154, August 1989, pp. 134-8; No. 155, September 1989, pp. 114-18; No. 156, October 1989, pp.112-21.

[Webb, 1988]
J. P. Webb, *Finite Elements in Electromagnetics*, class notes (304-647), pp. TR53-TR70, 1988.

[Widrow, 1988a]
B. Widrow, R. G. Winter and R. A. Baxter, "Layered Neural Nets for Pattern Recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing* **36**, No. 7, July 1988, pp. 1109-18.

[Widrow, 1988b]
B. Widrow and R. Winter, "Neural Nets for Adaptive Filtering and Adaptive Pattern Recognition," *Computer* **21**, March 1988, pp. 25-39.

Appendix I.
Mathematics for Element Error Computation

i. Least Squares Fit

The least squares fit of a trail element to a solution is accomlished by minimizing the squared error. The error is the difference between the "exact" solution, and the linear solution on the element. The minimization is done with respect to three parameters: the values of the solution on the three vertices of the linear trial element.

The "exact" solution in this case is in fact only a close approximation which itself has been computed on a mesh. To avoid confusion in the following derivation, the elements from the solution will be refered to as "solution elements", and the element being fitted to the solution as the "trial element".

The linear solution on the trial element is represented by:

$$\tilde{A}_z = a_1 + a_2 x + a_3 y = \mathbf{a}^T \mathbf{x} \qquad (1)$$

where $\mathbf{x} = (1 \; x \; y)^T$ and $\mathbf{a} = (a_1 \; a_2 \; a_3)^T$. The squared error of the trial element is found by integrating over the trial element:

$$E = \int_S (A_z - \tilde{A}_z)^2 \, dS \qquad (2)$$

The minimum of the squared error is found by differentiating w.r.t. the each of the parameters and equating the results to zero:

$$\frac{\partial}{\partial a_1} \int_S (A_z - \tilde{A}_z)^2 \, dS = -2 \int_S (A_z - \tilde{A}_z) \mathbf{x} \, dS = 0 \qquad (3)$$

Solving for the parameters gives:

$$\begin{aligned} \int_S A_z \mathbf{x} \, dS &= \int_S \tilde{A}_z \mathbf{x} \, dS \\ &= \int_S (\mathbf{a}^T \mathbf{x}) \mathbf{x} \, dS \\ &= \int_S (\mathbf{x}\mathbf{x}^T) \, dS \, \mathbf{a} \end{aligned} \qquad (4)$$

This equation is a linear system of three variables, and can be solved for the

parameters of the linear element.

ii. Squared Error

The expression for the squared error can be simplified as follows:

$$\begin{aligned} E &= \int_S A_z^2 - 2A_z\tilde{A}_z + \tilde{A}_z^2\, dS \\ &= \int_S A_z^2\, dS - \mathbf{a}^T \int_S A_z \mathbf{x}\, dS \end{aligned} \tag{5}$$

In the program to compute ideal sizes, the integration is performed by taking each solution element and clipping it against the trial element. The clipping algorithm used is the Sutherland-Hodgman algorithm which is described in Section 3.14 of [Foley, 1990]. Once the integral quantities are computed, the parameters are calculated by solving equation (4), after which the squared error is readily computed using equation (5).

The mathematical methods necessary to perform these integrations is presented in [Webb, 1988]. The FORTRAN code used to implement parts of these computations was donated by J. S. McFee.

Appendix II.
Geometric Input Files for C-Core and E-Core

```
; Geometric input file for C-Core.

; Domain
-5.0 -5.0 ; x_min, y_min
 4.7  5.0 ; x_max, y_max

; Core
1          ; material type (1 = iron, 2 = copper)
12         ; number of vertices (a closed figure is assumed)
-3.1        3.5 ;    (x, y) coordinates of vertices
-3.1       -3.4
 3.2       -3.4
 3.2       -1.0
 0.95      -0.1
 0.95      -1.5
-0.9       -1.5
-0.9        1.6
 0.95       1.6
 0.95       0.15
 3.2        1.4
 3.2        3.5

; Left Coil
2
4
-3.1        1.0
-3.7        1.0
-3.7       -0.9
-3.1       -0.9

; Right Coil
2
4
-0.3        1.0
-0.9        1.0
-0.9       -0.9
-0.3       -0.9

0              ; zero indicates no more objects
```

```
; Geometric input file for E-Core.

; Domain
 -8.0       -9.8      ; x_min, y_min
 11.0       10.0      ; x_max, y_max

; Core
1           ; material type (1 = iron, 2 = copper)
22          ; number of vertices (a closed figure is assumed)
 9.0   7.0 ; (x, y) coordinates of vertices
 0.0   7.0
-6.0   6.0
-6.0   1.0
-4.0   1.0
-4.0   4.0
-2.0   4.0
-2.0  -3.0
-5.0  -3.0
-5.0   0.6
-7.0   0.6
-7.0  -5.0
 0.0  -7.0
 9.0  -7.0
 9.0   0.0
 6.0  -1.0
 6.0  -4.0
 2.0  -3.0
 2.0   4.0
 6.0   4.0
 6.0   0.2
 9.0   0.2

; Left Coil
2
4
-2.0   3.0
-3.0   3.0
-3.0  -2.0
-2.0  -2.0

; Right Coil
2
4
 3.0   3.0
 2.0   3.0
 2.0  -2.0
 3.0  -2.0

0           ; zero indicates no more objects
```