SOFTWARE TOOL METHODOLOGIES ON A GPU FOR FINITE ELEMENT OPTIMIZATION IN MAGNETICS

By

Sivamayam Sivasuthan

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Electrical Engineering—Doctor of Philosophy

2015

ABSTRACT

SOFTWARE TOOL METHODOLOGIES ON A GPU FOR FINITE ELEMENT OPTIMIZATION IN MAGNETICS

By

Sivamayam Sivasuthan

The design of magnetic devices requires optimization coupled with finite element analysis (FEA). This involves a massive computational load and requires a specialized mesh generator. It is therefore not practicable. This thesis therefore presents i) a parameterized iterative mesh generator for two-dimensional and three-dimensional finite element optimization; ii) fast and low memory finite element solvers using a graphics processing unit (GPU). In particular we introduce element by element finite element computations on a GPU with a speedup of 102 while the best competing method gives only 10; and iii) an examination of parallelizing such matrix computations on already parallelized genetic algorithm threads using new GPU architectures. The resulting system is reliable and yields solutions in practicable times with massive speedup. Example inverse optimization problems are presented. These software tools are written in C/C++ and CUDA C/C++. The system is shown to be applicable to the synthesizing of two-dimensional and three-dimensional electromagnetic devices and to non-destructive evaluation (NDE) problems.

Several finite element mesh generators exist in the public domain, some even based on a parametric device description. But for optimization we need a parametrically described mesh dynamically evolving through the iterations without user input. The few that exist are commercial and their methodology is not known. In this thesis the mesh generator that we describe is in open source code with parametric mesh generation that runs nonstop and seamlessly through optimization iterations to convergence without user intervention. Such mesh

generators as do exist are rare, commercial and not easily available to researchers except at great cost and never with the code to modify it to suit needs individual. Besides, the typical mesh generator requires some man-machine interaction to define the mesh points and boundary conditions and does not work for nonstop optimization iterations. We take two regular open source mesh generators, one for two-dimensional systems and the other for three-dimensional systems, and write a script-based interface as open source code to run nonstop for optimization. We then use it to create an NDE system for an army ground vehicle's hull defect characterization and use it equally adaptively for machine design. A simple scheme of averaging neighbor heights gives us a smooth geometry without having to use Bezier curves.

This thesis also points out using a literature survey issues in GPU computation which result in erratic speedup and explain why in some instances GPU solutions are arithmetically a slight improvement on CPU solutions.

This thesis is dedicated to my parents; Sinnathurai Sivamayam and Sivamayam Sivakumary, and my mentors . . .

ACKNOWLEDGMENTS

I would like to address my heartily profound gratitude and appreciation to my advisor Dr. S. Ratnajeevan H. Hoole for taking me on as a research student under his valuable guidance with funding. His advice and insights were the real encouragement to complete this work. It has been an honor to work with him. My sincere thanks also go to the rest of my PhD committee members, Dr. Lalita Udpa, Dr. William Punch and Dr. Nihar Mahapatra, for their suggestions and encouraging comments throughout this work.

Next, I am very grateful to the US Army's Tank Automotive Research and Development Center (TARDEC) for funding our research under Contract Number W911NF-11-D-0001 and W56HZV-07-2-001. I am thankful to the MSU Graduate School and College of Engineering for awarding a summer graduate excellence fellowship thrice, a graduate office fellowship and a graduate teaching assistantship for one semester. This support was instrumental in facilitating the completion of my degree.

I would also like to thank the faculty- and staff- members of the Department of Electrical and Computer Engineering at Michigan State University, especially its College of Engineering, for their support in various forms and their friendship. I have been lucky to share a lab with so many friendly colleagues, namely V. U. Karthik, M. R. Rawashdeh and T. Mathiyalakan. I express my sincere gratitude for their unstinted friendship. My sincere thanks go to all the academic and nonacademic staff members and friends at University of Jaffna, Sri Lanka, where my career as a student in computer science started.

Finally, I am forever indebted to my parents and other members of my family, who have supported and encouraged me through their kindness and affection, so that I could concentrate on my studies. They touched me more deeply than I could have ever expected.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1

Introduction

1.1 Motivation

Existing finite element analysis software for electromagnetic fields provides advanced features for design or analysis problems. However engineering design involves inverse problem solving. That is, once the requirements are given, we have to find the geometrical shape or the material properties, which will satisfy the requirements as closely as possible (if not exactly). This is done basically using optimization algorithms. The deviation of the performance result of the analysis of the present shape from the requirements must be formulated as an error function, which must be minimized to get the optimum solution [1]. Most finite element analysis software packages do not support this. Therefore this must be done using a trial and error process. However a trial and error process alone will be so inefficient that it is not practically possible to obtain a solution. Therefore an expert is required to guide this trial and error process using his experience. However the success is directly dependent on the ability of the human expert and therefore the solution obtained may deviate from the most optimum solution possible (Note that computer based optimization solutions also do not guarantee the global minimum, however they can reach a far better solution than a human expert).

There are some finite element analysis software packages that support optimization. However such software can only be used for the optimization of special classes of problems. A totally new program has to be developed in order to solve a new type of problem. This is an expensive and time consuming task. This forces the designers who do not have the resources to

develop a separate optimization program for their specific need, to use the old trial and error approach with traditional finite element analysis programs. Actually, it is usually not economical to develop a separate program if we have to solve a given problem only once or twice.

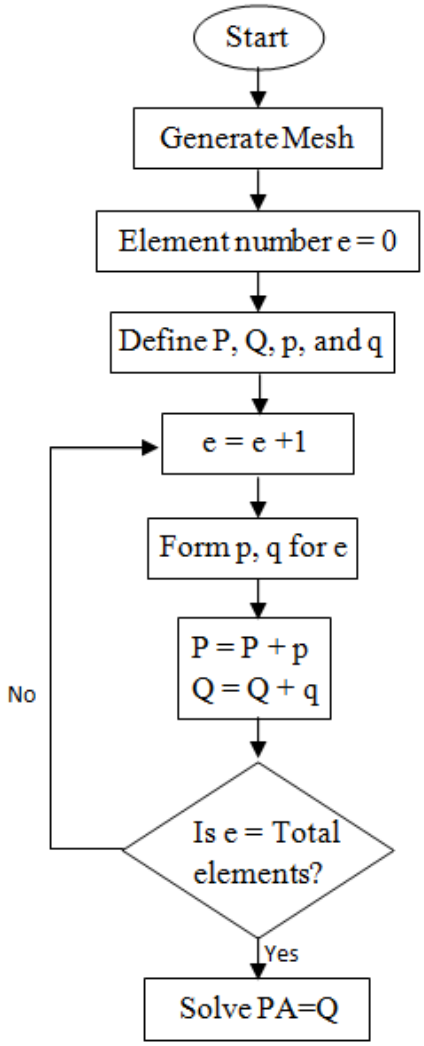

Figure 1.1: Regular finite element analysis

Figure 1.1 shows the regular finite element analysis process. Figure 1.2 shows finite element optimization using the zeroth order genetic algorithm (GA) and gradients based algorithms. If we use GA optimization we can speed-up our solution time through parallelization on a GPU. In the genetic algorithm, the design parameter vector $\{h\}$ is binary encoded in general

[2]. A chromosome is a vector $\{h\}$. Its fitness score f is defined in terms of the object function

$$f = \frac{1}{1+F} \tag{1.1}$$

Though GA is practicable and gives a faster solution when parallelized [3], it is slow in our experience as a single process when compared with the gradient optimization methods. In sequential CPU computing, the fitness value is calculated for each chromosome one by one. When the population is high it takes a very long time to converge. We use GPU computation to overcome this problem [3]. We launched GPU kernels for computing the fitness value. So the fitness value will be calculated simultaneously for each chromosome in the population (Figure 1.2) [4]. Genetic algorithm based finite element analysis in magnetics has been carried out by many researchers [5, 6, 7, 8, 9, 10] over the past 20 year period.

In gradients based optimization, the changes in parameters of device description $\{h\}$ are against the gradient of the object function F because in one-dimensional analogy the minimum point is to the right of locations with negative gradient and to the left of those with positive gradient:

$$\{h\} = \{h\} - \alpha \frac{\partial F}{\partial \{h\}} \tag{1.2}$$

where the amount of change α is determined by a line search [11]. The computation of the gradient ∇F (= $\partial F / \partial \{h\}$) was previously by finite difference, computing F through a finite element solution corresponding to a given $\{h\}$ and then in turn changing each component h_i by an infinitesimal amount and re-computing F to get = $\partial F / \partial h_i \approx \delta F / \delta h_i$. Thus the component of ∇F at each iterative step with n components of $\{h\}$ took n + 1 finite element solutions and then once the direction of change of $\{h\}$, $-\nabla F$, is established several more finite element solutions need to be sought during the line search as α in Equation 1.2 is progressively increased and the

problem iteratively solved until the minimum of F in that direction is identified [11]. Each changed $\{h\}$ means a new geometry and therefore a new mesh. For a seamless iterative process, automatic mesh generators are required that can yield a mesh corresponding to a given $\{h\}$. However with gradient based algorithms the matrix solution may be parallelized whereas with the GA, both matrix solution and optimization may be parallelized through forking within a fork [12].

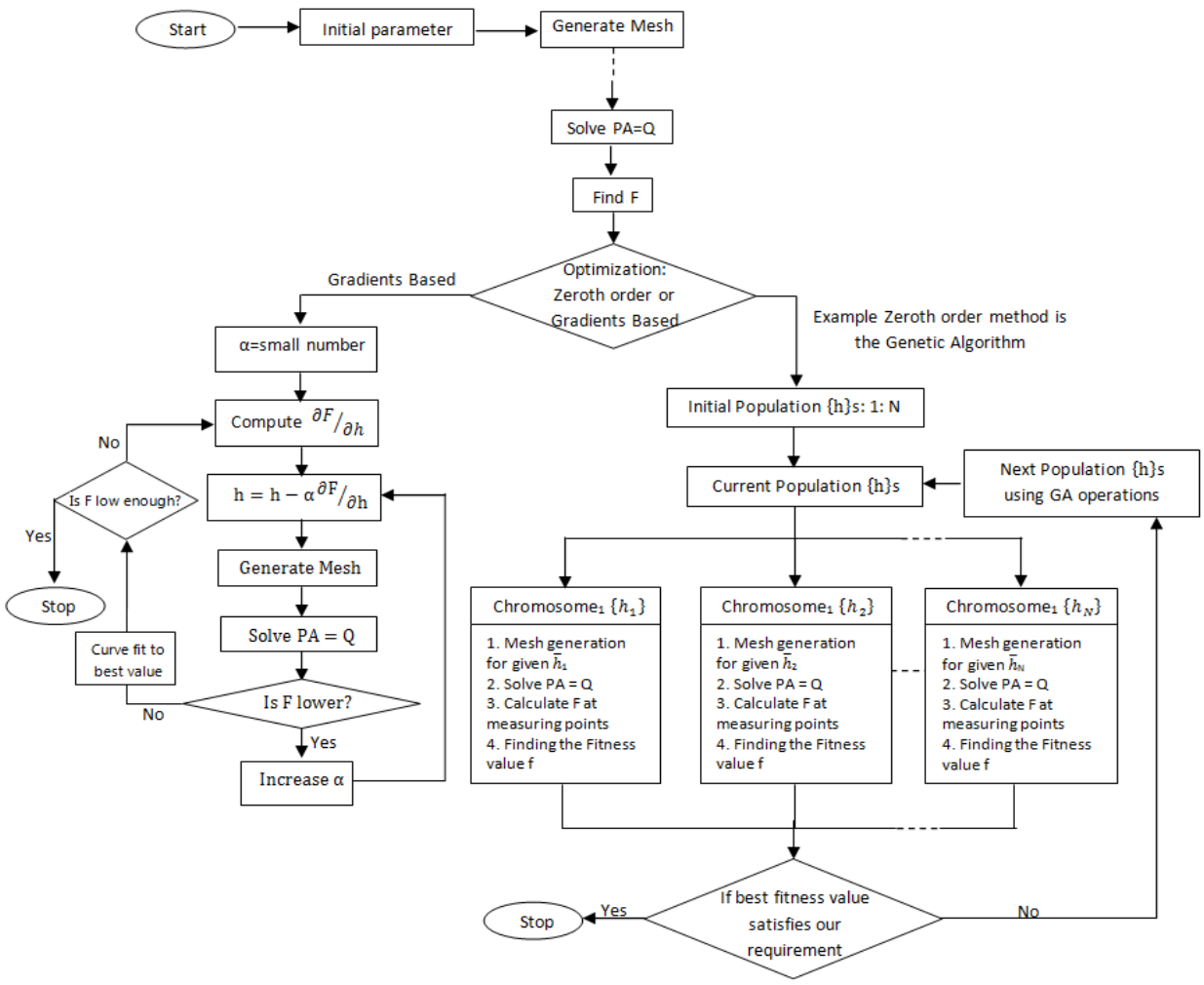

Figure 1.2: FEM with optimization algorithms

Therefore, we have developed general-purpose two-dimensional and three-dimensional finite

element analysis inverse optimization procedure using GA software tools on a GPU for analysis and geometrical shape optimization in design and NDE problems.

1.2 The Finite Element Method

1.2.1 Introduction

The finite element method is a method used for solving partial differential equations numerically. It is widely used to solve electromagnetic problems [13], structural design problems [14] etc. using computers. This method is generally used to find the distribution of a certain field (e.g. magnetic vector potential, electric scalar potential, tension, fluid velocity, etc.) in space governed by a given differential equation called the governing equation.

In order to solve the problem numerically, the solution region is divided into a finite number of elements (which are not needed to be uniform). The potentials are assumed to have a known mathematical variation called the trial function (e.g. linear variation, quadratic variation, etc.) over each individual element. The field is postulated to be interpolations from its values at certain nodes. Note that these elements must be small enough for this assumption to be valid [13]. The problem is then solved to get the potentials at the interpolation nodes of these elements. Then the potential at any given point in the solution space can be found by interpolating these now known potentials using the trial function.

1.2.2 Two-Dimensional Problems

In most two dimensional finite element analyses, the space is divided into a mesh of triangles. Points in these triangles are called the interpolation nodes of the mesh. The variation of

the potential over the triangles is assumed to be defined by a given trial function (most often a first order trial function). The objective is to find the potentials at the nodes of the mesh so that the potential at any given point inside a triangle can be found using the trial function. To do this, we develop one equation per unknown node in the mesh. Then, this set of equations must be solved to find the potentials at each node. Since these equations alone are not sufficient to get a unique solution, some boundary conditions must also be considered.

1.2.3 Trial Function

Since we use only first order trial functions in our software tools, let us consider them in detail. In two-dimensional cartesian coordinates, a first order trial function can be expressed as follows [13, 15]:

$$A = a + bx + cy \tag{1.3}$$

where x and y are cartesian coordinates and a, b and c are constants for the triangle. Triangular coordinates are used in finite element analysis, as these normalized coordinates provide several

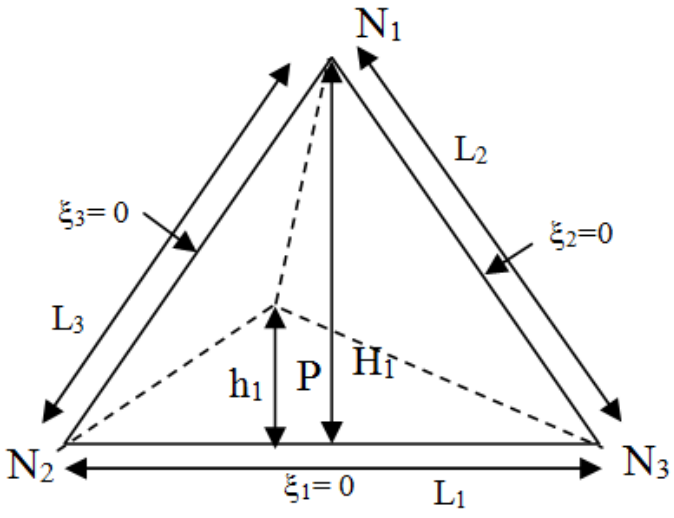

Figure 1.3: Definition of H_1 and h_1 for a triangle

advantages in analyzing properties inside triangles. Triangular coordinates ξ1, ξ2, and ξ3 of a point P inside a triangle are defined as follows (see Figure 1.3).

$$\xi_1 = h_1 \div H_1 \tag{1.4}$$

$$\xi_2 = h_2 \div H_2 \tag{1.5}$$

$$\xi_3 = h_3 \div H_3 \tag{1.6}$$

where H_1 is the shortest distance from N_1 to N_2N_3 and h_1 is the shortest distance from P to N_2N_3, and so on (see Figure 1.3). If S is the area of the triangle,

$$S = \frac{1}{2}H_1L_1 = \frac{1}{2}H_2L_2 = \frac{1}{2}H_3L_3 \tag{1.7}$$

Considering the three triangular areas in the triangle separated by dotted lines, (Figure 1.3)

$$S = \frac{1}{2}h_1L_1 + \frac{1}{2}h_2L_2 + \frac{1}{2}h_3L_3 \tag{1.8}$$

From the definitions of ξ1, ξ2 and ξ3, Equations 1.4, 1.5 & 1.6

$$S = \frac{1}{2}H_1L_1\xi_1 + \frac{1}{2}H_2L_2\xi_2 + \frac{1}{2}H_3L_3\xi_3 \tag{1.9}$$

Using 1.7

$$1 = \xi_1 + \xi_2 + \xi_3 \tag{1.10}$$

Using linear interpolation, the Cartesian coordinates of a point within the triangle can be written as follows [15].

$$x = \xi_1x_1 + \xi_2x_2 + \xi_3\,x_3 \tag{1.11}$$

$$y = \xi_1y_1 + \xi_2y_2 + \xi_3\,y_3 \tag{1.12}$$

These are exact because x and y are linear. Solving 1.10, 1.11 and 1.12,

$$\xi_i = \frac{\begin{vmatrix} 1 & 1 & 1 \\ x & x_{i1} & x_{i2} \\ y & y_{i1} & y_{i2} \end{vmatrix}}{\begin{vmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{vmatrix}} \tag{1.13}$$

Where *i1* = *i* mod 3 + 1, and *i2* = *i1* mod 3 + 1 Equation 1.13 can be re-written as,

$$\xi_i = a_i + b_i x + c_i y \tag{1.14}$$

where

$$a_i = (x_{i1}y_{i2} - x_{i2}y_{i1}) \ /\Delta \tag{1.15}$$

$$b_i = (y_{i1} - y_{i2}) \ /\Delta \tag{1.16}$$

$$c_i = (x_{i2} - x_{i1}) \ /\Delta \tag{1.17}$$

and

$$\Delta = \begin{vmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{vmatrix} \tag{1.18}$$

Using these triangular coordinates, the first order trial function A is expressed as follows

$$A = A_1\xi_1 + A_2\xi_2 + A_3\xi_3 \tag{1.19}$$

where A is the potential at the point (ξ_1, ξ_2, ξ_3) and A_1, A_2 and A_3 are the potentials at the node points 1, 2 and 3 of the triangle.

This can be verified by considering that the triangular coordinates of the three nodes of the triangle are (1,0,0), (0,1,0) and (0,0,1). By substituting these points, one will get A_1, A_2 and

A_3 as the potentials at the three nodes and a linear variation of potentials along any given line inside a triangle. This trial function also provides a continuous variation of potentials from triangle to triangle and a continuous first derivative from triangle to triangle along the tangential direction of the boundary. Equation 1.19 can be re-written as,

$$A = \{A\}^T\{\alpha\} \tag{1.20}$$

where

$$\{A\} = \{A_1\ A_2\ A_3\ \}^T \tag{1.21}$$

and

$$\{\alpha\} = \{\xi_1\ \xi_2\ \xi_3\ \}^T \tag{1.22}$$

From 1.16 and 1.17,

$$\frac{\partial\{\alpha\}}{\partial x} = \frac{[b_1\ b_2\ b_3\ ]^T}{\Delta} \quad = \{b\} \tag{1.23}$$

$$\frac{\partial\{\alpha\}}{\partial y} = \frac{[c_1\ c_2\ c_3\ ]^T}{\Delta} \quad = \{c\} \tag{1.24}$$

From 1.19,

$$\frac{\partial A}{\partial x} = \frac{\partial\{A\}^T\{\alpha\}}{\partial x} \quad = \{A\}^T\frac{\partial\{\alpha\}}{\partial x} \quad = \{A\}^T\{b\} \tag{1.25}$$

$$\frac{\partial A}{\partial y} = \frac{\partial\{A\}^T\{\alpha\}}{\partial y} \quad = \{A\}^T\frac{\partial\{\alpha\}}{\partial y} \quad = \{A\}^T\{c\} \tag{1.26}$$

Let us examine another property of these triangular coordinates, referring to Figure 1.4

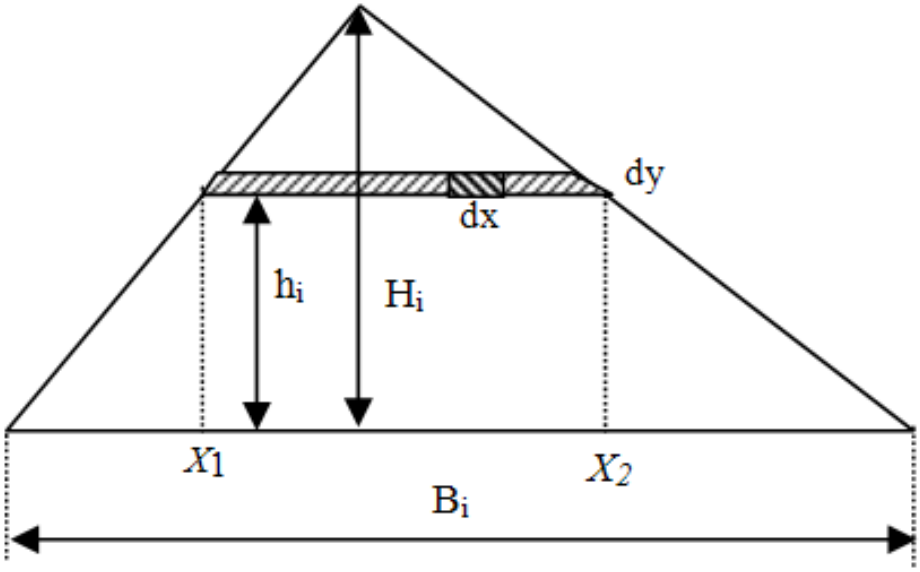

Figure 1.4: Integration of triangular coordinates

$$\int\int_{\Delta} \xi_i dR = \int_{h=0}^{H_i}\int_{x=x_1}^{x_2} \frac{h_i}{H_i} dx dh_i = \int_{h=0}^{H} [x_2 - x_1]\frac{h_i}{H_i} dh_i$$
$$= \int_{h_i=0}^{H_i} \left[\frac{B_i(H_i - h_i)}{H_i}\right] \frac{h_i}{H_i} dh_i = \frac{1}{2}\frac{1}{3} B_i H_i = \frac{1}{3} S \qquad (1.27)$$

where S is the area of the trainable. That is,

$$\int\int_{\Delta} \{\alpha\} dR = \int\int_{\Delta} [\xi_1 \;\; \xi_2 \;\; \xi_3]^T dR = \left[\frac{1}{3}S \;\; \frac{1}{3}S \;\; \frac{1}{3}S\right]^T = T^{0,1} S \qquad (1.28)$$

Where $T^{0,1} = \left[\frac{1}{3} \;\; \frac{1}{3} \;\; \frac{1}{3}\right]$; $T^{0,1}$ is a metric tensor

1.2.4 Solving Magneto-Static Problems

This solution uses first order triangular elements and materials with linear magnetic properties at low frequency for simplicity. The following differential equation (From Maxwell's laws [13] (under static conditions)) governs the solution region.

$$\nabla \times \overline{H} = \overline{J} \tag{1.29}$$

where $\overline{H}$ is the magnetic field intensity and $\overline{J}$ is the current density. Since $\overline{H} = \upsilon \overline{B}$ and $\overline{B} = \nabla \times \bar{A}$ where υ is the reluctivity

$$\nabla \times (\nu \nabla \times \overline{A}) = \overline{J} \tag{1.30}$$

If ν is a constant, since $\nabla \times \nabla \times \overline{A} = \nabla(\nabla.\overline{A}) - \nabla^2\overline{A}$ and $\nabla.\overline{A} = 0$, from 1.28,

$$-\nu \nabla^2 \overline{A} = \overline{J} \tag{1.31}$$

From 1.31, the energy functional can be derived. Since the energy is at its minimum at the stable state, this function should achieve its minimum at the point of the solution. This function is called the Lagrange function:

$$L(\overline{A}) = \int\int_R \left[\frac{\nu}{2}[\nabla \overline{A}]^2 - \overline{J}\,\overline{A}\right] dR \tag{1.32}$$

The solution region has been divided into triangles. Therefore now the total energy can be written as the sum of the energies of each individual triangle.

$$L(\overline{A}) = \sum_{\forall \Delta} \left[\int \int_{\Delta} \left[\frac{\nu}{2} [\nabla \overline{A}]^2 - \overline{J} \ \ \overline{A} \right] dR \right] \tag{1.33}$$

Let us integrate this by parts,

$$\int \int_{\Delta} \left[\frac{\nu}{2} [\nabla \overline{A}]^2 \right] dR = \int \int_{\Delta} \frac{\nu}{2} \left[\left(\frac{\partial \{A\}^T \{\alpha\}}{\partial x} \right)^2 + \left(\frac{\partial \{A\}^T \{\alpha\}}{\partial y} \right)^2 \right] dR$$

$$= \int \int_{\Delta} \frac{\nu \{A\}}{2} \left[\left(\frac{\partial \{\alpha\}}{\partial x} \right)^2 + \left(\frac{\partial \{\alpha\}}{\partial y} \right)^2 \right] \{A\}^T dR = \frac{\nu \{A\}}{2} S[\{b\}\{b\}^T + \{c\}\{c\}^T]\{A\}^T \tag{1.34}$$

From 1.28,

$$J\{A\} \int \int \{\alpha\} dR = J\{A\} S \left[\frac{1}{3} \ \ \frac{1}{3} \ \ \frac{1}{3} \right]^T \tag{1.35}$$

From 1.33, 1,34 and 1.35

$$L\{A\}_{\Delta} = 0.5\{A\}^T [P]^L \{A\} - \{A\}^T \{q\} \tag{1.36}$$

where

$$\{A\} = \{A_1 \ A_2 \ A_3\}^T, \{q\}^T = JS \left[\frac{1}{3} \ \ \frac{1}{3} \ \ \frac{1}{3} \right] /2 \tag{1.37}$$

and

$$[P]^L = \nu S[\{b\}\{b\}^T + \{c\}\{c\}^T] \tag{1.38}$$

From 1.33 and 1.36 we can get

$$L\{A\} = \sum_{\forall \Delta} [0.5\{A\}[P]^L\{A\}^T - \{A\}^T\{q\}] \tag{1.39}$$

To get the solution we have to minimize *L{A}*

$$\frac{\partial L}{\partial \{A\}} = \frac{\partial}{\partial \{A\}} [\{A\}[P]^L\{A\}^T - \{A\}^T\{q\}] \tag{1.40}$$

$$\frac{\partial L}{\partial \{A\}} = \sum_{\forall \Delta} [[P]^L\{A\} - \{q\}] = 0 \tag{1.41}$$

Therefore the final solution can be obtained by solving,

$$\sum_{\forall \Delta} \left([P]^L\{A\} \right) - \sum_{\forall \Delta} \{q\} = 0 \tag{1.42}$$

This can be re-written in the form,

$$\sum_{\forall \Delta} [P]^L\{A\} = \sum_{\forall \Delta} \{q\} \tag{1.43}$$

and the equation solved for {A}.

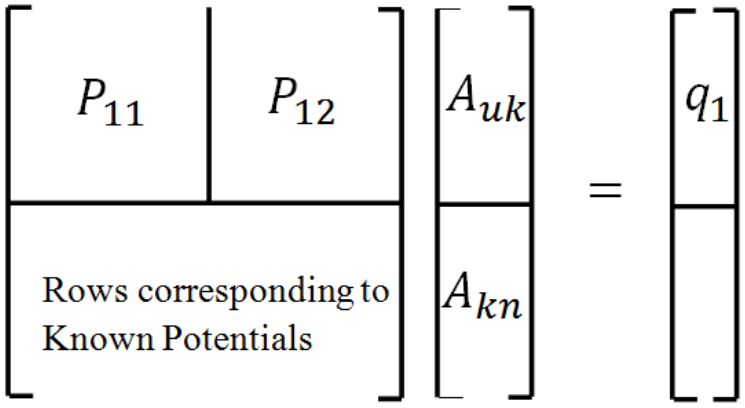

Figure 1.5: Known elements and unknown elements in matrix

1.2.5 Boundary Conditions

We basically use two types of boundary conditions, namely Neumann and Dirichlet [13]. Dirichlet boundary conditions mean that the potential along the boundary is fixed at a given value and Neumann boundary conditions mean the derivative of the unknown potential at the boundary along the normal direction is zero. Dirichlet boundary conditions can be implemented by considering the node points on the boundary to have known values. Neumann boundary conditions are implemented automatically if Dirichlet conditions are not used at a boundary [13]. They are said to be natural. If Neumann boundary conditions are used, it means the potentials at some points along the boundary are known. Therefore the vector A can be broken into two as potentials at nodes with known potentials A_{kn} and potentials at nodes with unknown potentialsA_{uk}. Therefore Equation 1.43 can be written as (see Figure 1.5),

$$[P_{11}]\{A_{uk}\} = \{q_1\} - [P_{12}]\{A_{kn}\} \tag{1.44}$$

This is the matrix equation used by this software to find the final solution of finite element analysis. There is no minimization of L with respect to A_{kn}

1.2.6 Three-Dimensional Problems

Corresponding to the triangle in two dimensions, the tetrahedron is a convenient element to use in three dimensions. For the tetrahedral coordinates ξ_i [13]:

$$\xi_1 = \frac{h_i}{H_i} \tag{1.45}$$

where now h_i is the height of a point from the opposite triangular face of a tetrahedron and H is the height of the vertex opposite that face. First order interpolation,

$$A = A_1\xi_1 + A_2\xi_2 + A_3\xi_3 + A_4\xi_4 \tag{1.46}$$

$$1 = \xi_1 + \xi_2 + \xi_3 + \xi_4 \tag{1.47}$$

$$x = \xi_1 x_1 + \xi_2 x_2 + \xi_3 x_3 + \xi_4 x_4 \tag{1.48}$$

$$y = \xi_1 y_1 + \xi_2 y_2 + \xi_3 y_3 + \xi_4 y_4 \tag{1.49}$$

$$z = \xi_1 z_1 + \xi_2 z_2 + \xi_3 z_3 + \xi_4 z_4 \tag{1.50}$$

Solving the preceding four equations for the four ξ s, we have

$$\xi_i = \frac{1}{6V}(a_i + b_i x + c_i y + d_i z) \tag{1.51}$$

Where,

$$a_i = (-1)^{i1}\left[x_{i1}\left(y_{i2}z_{i3} - y_{i3}z_{i2}\right) + x_{i2}\left(y_{i3}z_{i1} - y_{i1}z_{i3}\right) + x_{i3}\left(y_{i1}z_{i2} - y_{i2}z_{i1}\right)\right] \quad (1.52)$$

$$b_i = (-1)^{i}\left[\left(y_{i2}z_{i3} - y_{i3}z_{i2}\right) + \left(y_{i3}z_{i1} - y_{i1}z_{i3}\right) + \left(y_{i1}z_{i2} - y_{i2}z_{i1}\right)\right] \quad (1.53)$$

$$c_i = (-1)^{i1}\left[\left(x_{i2}y_{i3} - x_{i3}z_{i2}\right) + \left(x_{i3}z_{i1} - x_{i1}z_{i3}\right) + \left(x_{i1}z_{i2} - x_{i2}z_{i1}\right)\right] \quad (1.54)$$

$$d_i = (-1)^{i}\left[\left(x_{i2}y_{i3} - x_{i3}y_{i2}\right) + \left(x_{i3}y_{i1} - x_{i1}y_{i3}\right) + \left(x_{i1}y_{i2} - x_{i2}y_{i1}\right)\right] \quad (1.55)$$

$$6V = a_1 + a_2 + a_3 + a_4 \quad (1.56)$$

where $i, i1, i2, i3\ are$ 1,2,3 and 4 or a cycle permutation of them. A is the potential at the point (x, y, z) and A_1, A_2, A_3 and A_4 are the potentials at the node points 1, 2, 3 and 4 of a tetrahedron [13].

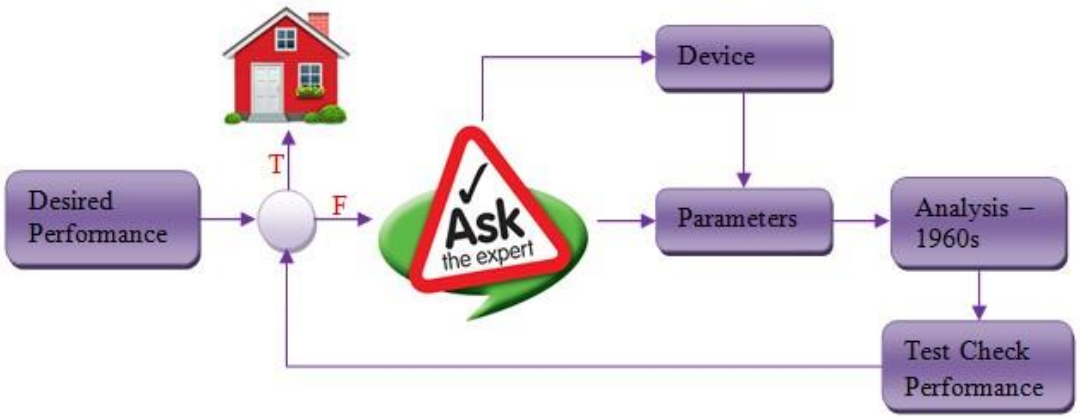

Figure 1.6: Traditional engineering design process

1.3 Inverse Optimization Problems

The inverse problem, the more practically realistic problem, is synthesis. That is, wanting a performance, computing the system description from it. Thus the computational design assignment may be this: compute the size and other descriptions of a motor that can produce so much torque. Figure 1.6 shows the traditional engineering design process. An expert decides which device to use and for that device assigns parameters to use and then checks the performance by making and testing. Finally, an expert has to make changes in parameters if needed. This process repeats until we get the desired performance. In the 1960s, the analysis phase was introduced in place of make and test before checking the performance. Thereafter the expert who makes the entire decision about the design process is replaced by powerful software. Figure 1.7 shows the modern design process using AI techniques, knowledge base etc. to make device selection. Optimization algorithms are used to select parameters in order to get the desired performance. In the modern design process, the analysis phase is replaced with synthesis. The earliest persons to automate this cycle in magnetics were Marrocco and Pironneau in 1978 [16]. In 1976 a parallel work with [16] by Arora and Hang [17] also established finite element optimization in magnetics.

An erratic undulating shape with sharp edges arose when Pironneau [18] optimized a recording-head to achieve a constant magnetic flux density and this was overcome through constraints [19]. Haslinger and Neittaanmaki [20] suggest Bezier curves to keep the shapes smooth with just a few variables to be optimized, while Preis, Magele, and Biro [21] have suggested fourth-order polynomials which when we tried gave us smooth but undulating shapes because of the higher order. Most of the required shape changes can be achieved with linear variations [22]. Figure 1.8 shows that without regularity constraints, sharp corners and jagged

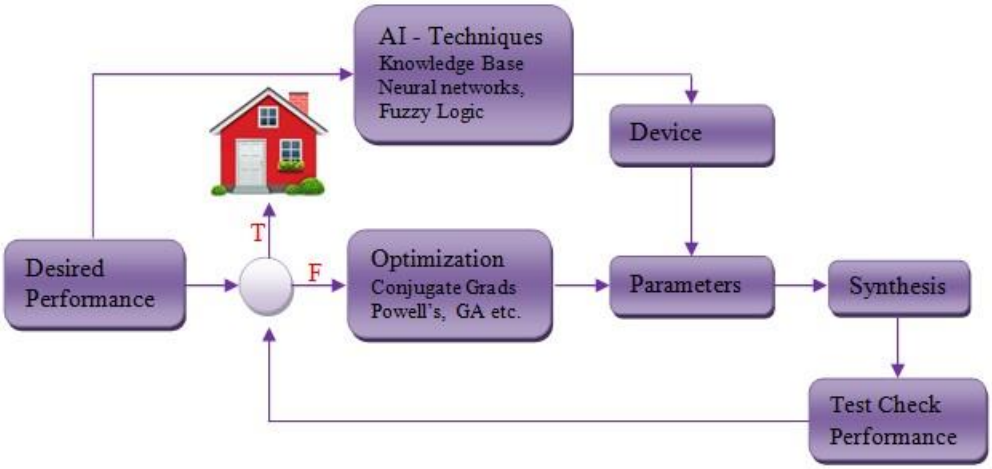

Figure 1.7: Modern design process

contours arise in designing a pole-face for constant vertical flux density. Figure 1.9 shows the shape is smoothened further and there is no sharp corner when B-spline curves are used but the undulation is mathematically correct though not practicable.

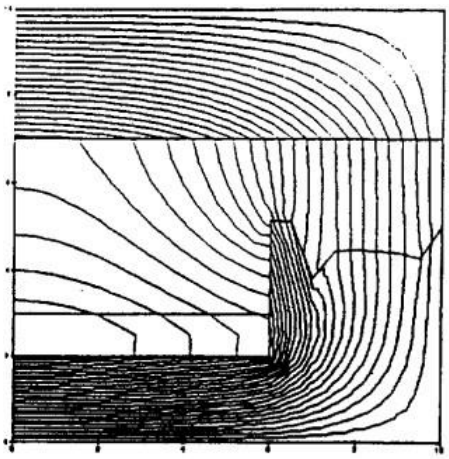

Figure 1.8: Equipotentials of vector potential at the optimum

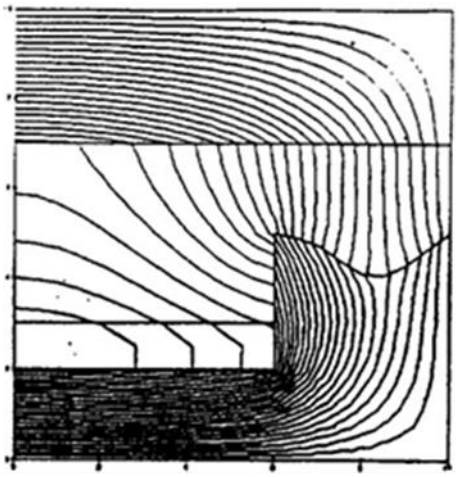

Figure 1.9: Equipotentials of vector potential at the optimum with B-spline interpolation

In optimization problems like NDE of steel plates, besides the detection of cracks, what is also important is their characterization. Characterization is necessary for determining whether any discovered crack demands withdrawal of the part from service [23, 24]. In eddy current

crack identification the response of a part to an eddy current test coil is compared to the response without a crack [4]. When different, the presence of the crack is flagged. But to characterize the defect, the computed response from eddy current analysis with a crack described by parameters is optimized to match measurements with computations. When the two match, the parameters describe the defect [4]. In inverse electromagnetic problem solutions by the finite element method, we require three tools. They are a special mesh generator, efficient matrix equation solvers and optimization algorithms. These are for the 3 major steps of finite element optimization. They are as follows,

1. Preprocessing: The essential operation for optimization is involved with this step. The design for optimization is parameterized before mesh generation. As the geometry defined by parameters is optimized, it changes shape, and a new finite element mesh must be created without stopping the optimization iterations to create a new mesh. Several finite element mesh generators exist in the public domain [25, 26, 27, 28, 29, 30, 31, 32], a few even based on a parametric device description. The required mesh generator must therefore support parameter based mesh generation and be completely automatic once the optimization process begins. That means we must be able to change the physical shape of the problem during run time and generate the mesh without stopping. Such mesh generators as do exist are rare, commercial and not easily available to researchers except at a great cost and never with the code to modify them to suit individual needs. We propose taking a regular open source mesh generator and writing a script-based interface as open source to run nonstop for two and three dimensional optimization problems. In this thesis we are going to develop parametric mesh generators that run nonstop and seamlessly through optimization iterations to convergence.

2. Solution: The biggest load in finite element field computation is in matrix solution. Recently GPU computing has had great success in many very large numerical computations (For example [33, 34, 35, and 36]). GPU-based finite element computation offers massive parallelization. This thesis will investigate speeding up using the GPU in sparse matrix computation. We will also examine the memory needs. For this purpose we will investigate parallel EbE processing by Gauss iterations [24] and preconditioned conjugate gradient [23].

3. Optimization: The parameters need to be optimized in NDE as well as synthesis to make the computed fields match the desired performance. As we discussed in the introduction section, we can use zeroth order optimization methods like the genetic algorithm, bee colony algorithms [37] etc. Moreover, if gradient methods are to be used in the mesh topology the nodal connections need to be held fixed to preserve C^1 continuity of the object function lest the mesh-induced minima are seen by the optimization algorithm as from the physics of the problem

This thesis mainly focuses on the first two steps because several open source optimization algorithms are available on the web (for example, [38]) and a colleague in the group is using GPU computations to parallelize genetic algorithm optimization [4]. However his code will be used for the test problems.

Chapter 2

Parameter Based Unstructured Mesh Generator for Two and Three Dimensional Problems for Seamless Optimization

2.1 Background

Figure 2.1 shows the design cycle for a geometric optimization problem. In the beginning, the initial geometric positions are either selected by the subject expert or in the absence of the expert, randomly selected. In the next step we generate the mesh for the current geometry, measure the object value by a finite element solution and check whether it is minimum or not. If this is a minimum we terminate the loop; otherwise we change the geometric parameters and do the same to procedure again.

Mesh generation is therefore a very important part of finite element analysis. But mesh generators do not support parameter based mesh generation for optimization. For real world inverse problems we need a mesh generator as a library where the design is described by parameters $\bar{h}$ and it takes $\bar{h}$ as input and returns the mesh from iteration to iteration.

Figure 2.2 shows the design cycle for an inverse problem. In the first step the design parameter set $\bar{h}$ is randomly selected (or estimated by a subject expert) and thereupon we generate the corresponding mesh, get the finite element solution and measure the object value (often conveniently defined as a least square difference between design objects desired and

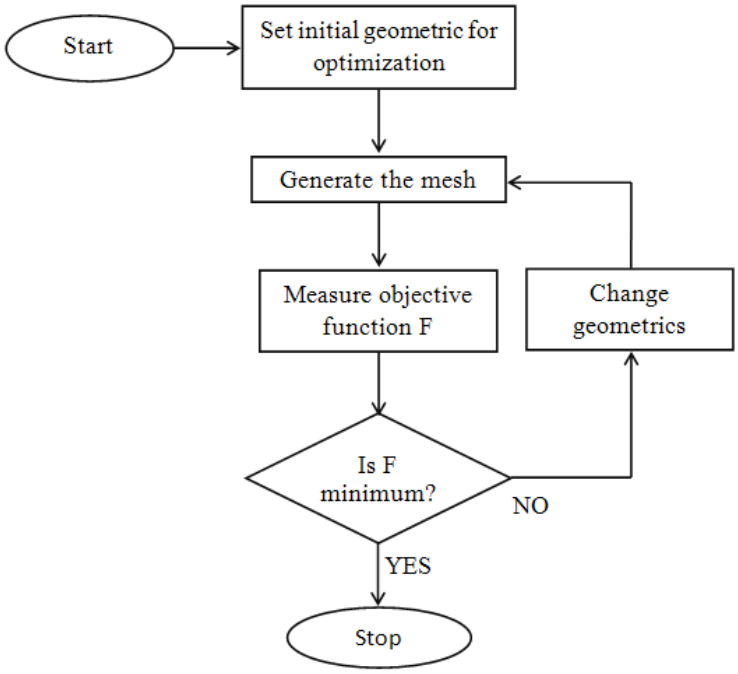

Figure 2.1: Design cycle for a geometric optimization

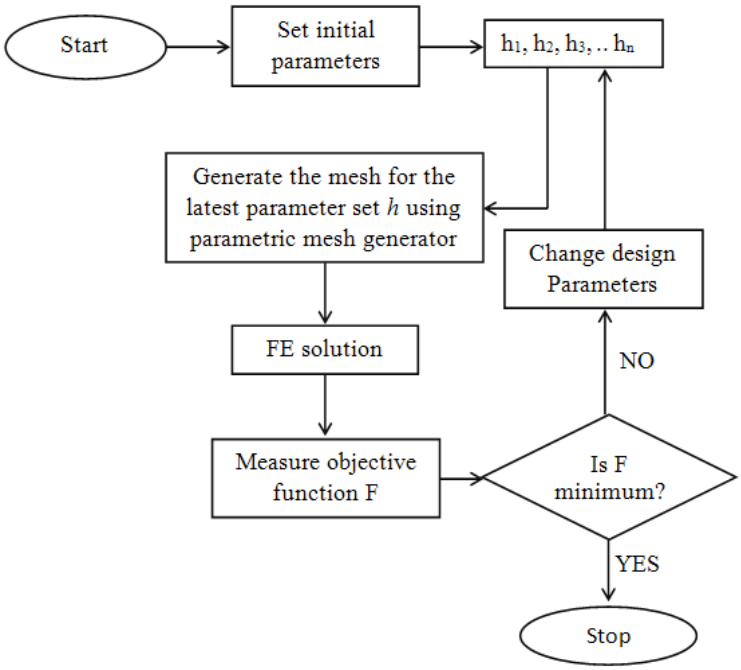

Figure 2.2: Design cycle for an inverse problem

computed) and check whether it is minimum or not. If this is a minimum we terminate the loop; otherwise we change the design parameters and do the same procedure again. This procedure repeats until the object value is acceptably small. For optimization to go on non-stop, the mesh needs to be generated for the new parameters without user intervention. In NDE the only difference is that the object function compares measured values with those computed from presumed values of $\bar{h}$ being sought.

In this section of this thesis we describe the necessity of a parametric mesh generator that runs nonstop and seamlessly through optimization iterations to convergence. Such mesh generators as do exist are rare, commercial and not easily available to researchers except at great cost and never with the code to modify them to suit individual needs. Besides, the typical mesh generator requires some man-machine interaction to define the points and boundary conditions and does not work for nonstop optimization iterations. We will take a regular open source mesh generator and write a script-based interface as open source to run nonstop for optimization.

There are many mesh generators available on the web [25, 26, 27, 28, 29, 30, 31, 32] and in the literature; some packages are open source software and others commercial. But they usually do not support parametric mesh generation. However they do support features we would like in a parametric mesh generator. To summarize some notable mesh generators, Triangle, which we use, generates exact Delaunay triangulations, constrained Delaunay triangulations, conforming Delaunay triangulations, and Voronoi diagrams to yield high quality triangular meshes without large angles as suited to finite element analysis [25]. AUTOMESH2D generates high quality meshes quickly [32]. Cardinal's Advanced Mesh INnovation with Octree [31], CGAL [28], ADMESH [30], and Delaundo [39] are all notable for special features. Indeed there are parametric mesh generators, for example [40]. However it is not publicly available. Another

such mesh generator is, CEDRATs suite Flux whereas parameters are changed, the mesh is generated and the device analyzed to study the effect of parameters on performance [41]. The same approach has been taken in NDE studies [42]. However, the works of [40, 41] are not intended for non-stop optimization. For that CEDRAT uses a script based scheme called GOT-It [43] which passes parameters to the program Flux and gets the results back for the optimization. Their software and information are mainly in the commercial domain. A 'Lightened' version of GOT-It, named FGot, is offered free to students although, but there again, the code is not accessible.

Figure 2.3 shows the performance comparison of triangulation using CPU-Delaunay triangulation (DT) and GPU-DT for different number of points [44]. We can see the gain is nominal compared to finite element solvers which are given in Chapter 5. Therefore it is not worthwhile parallelizing mesh generation

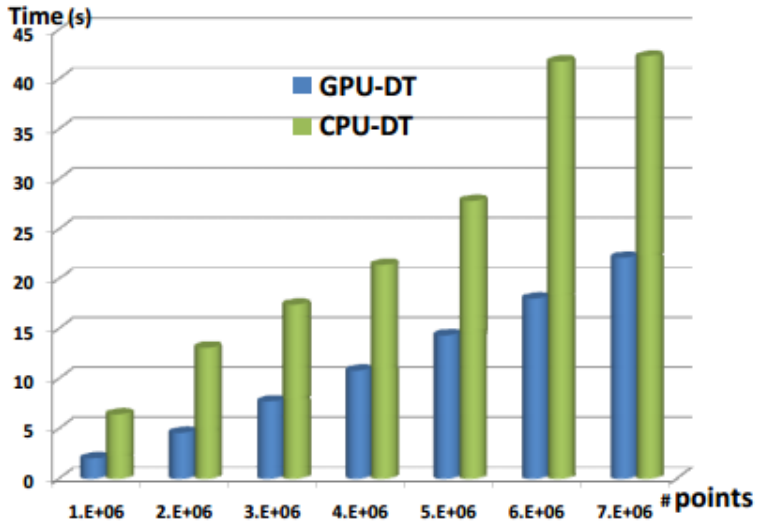

Figure 2.3: Performance comparison of triangulation using CPU-DT and GPU-DT for different number of points

Moreover, if gradient methods are to be used, the mesh topology given by the nodal connections need to be held to preserve C^1 continuity of the object function lest the mesh

induced fictitious minima are seen by the optimization algorithm as from the physics of the

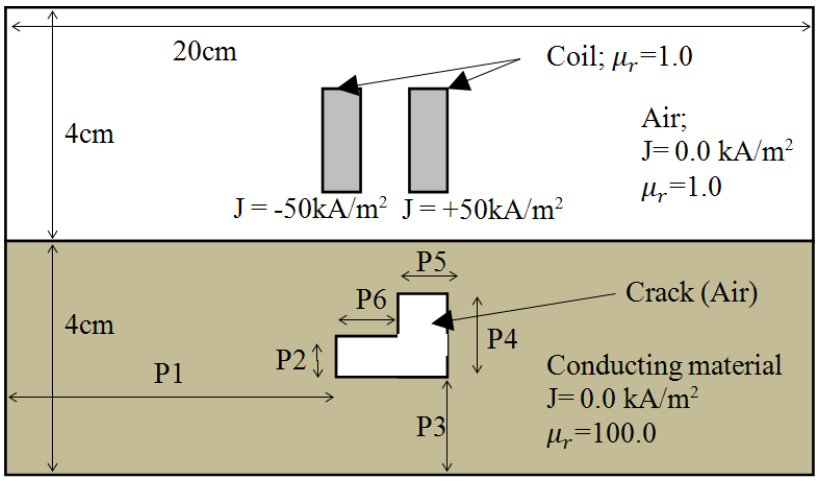

Figure 2.4: Problem specific parametric mesh generators

problem [45]. For these reasons very problem specific mesh generators are constructed by researchers. As an example, when an armored vehicle is targeted by an improvised explosive device, the armor is inspected by an eddy current test probe to characterize the interior damage to determine if the vehicle should be withdrawn from deployment. Figure 2.4 shows a problem specific parametrically described crack in steel excited by an eddy current probe, where P1-P6 are the lengths that represent the position and shape of the crack, J is current density and μr is relative permeability. In this NDE exercise the parameters need to be optimized to make the computed fields match the measurements. The mesh has been constructed for the specific problem. As the parameters change, the mesh topology is fixed, pulling and crunching triangles as shown in Figure 2.5. Such problem specific meshes are a headache because they restrict the geometry, lack flexibility and take time for modifications. Hence the need for general-purpose parametric mesh generators. We can use zeroth order optimization methods for which C^1 continuity is irrelevant, such as the genetic algorithm, bee colony algorithms etc. without pulling and crunching meshes for inverse problems; e.g. in reconstructing cracks to characterize interior

defects, or designing power devices. For non-stop optimization, the commercial code ANSYS offers a gradients-based optimization suite [46], but gives little information on the techniques employed. That is, although these methods are known within the companies, they are rarely published. There are other companies, particularly from structural engineering, that also offer gradients-based optimization. A huge lacuna is how they address the problem of mesh-induced minima. These artificial minima are seen as physics-based object function minima and the code tends to get stuck at these. Other approaches like a mathematical distance function to model the geometry lie in the domain of specialized efforts [47].

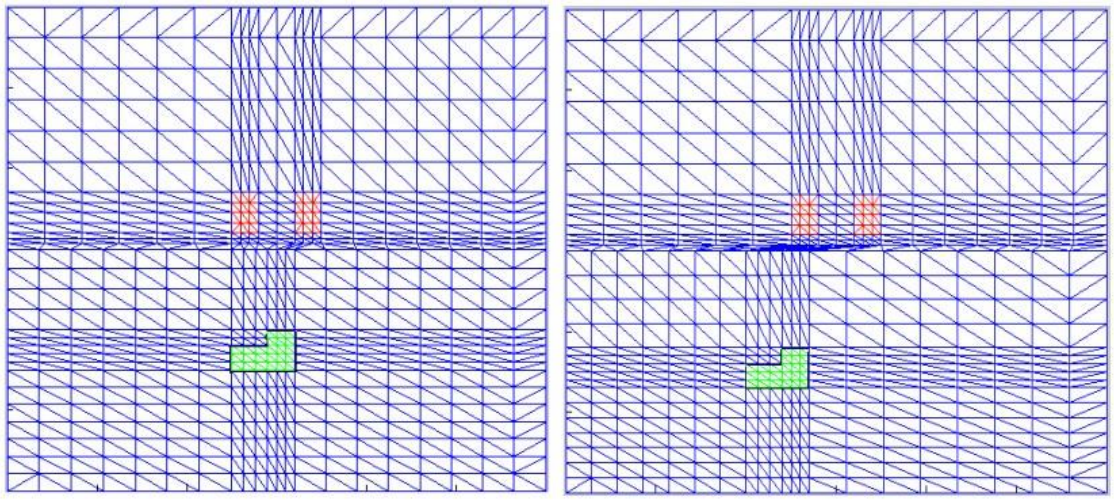

Figure 2.5: Elastically deformed problem specific NDE mesh: As defect moves

Like in 2D there are also many 3D-mesh generators available on the web and in the literature [29, 43, 48, 49]. Some are open source software and others commercial. TetGen [48, 49] is for tetrahedral mesh generation and is more effective than previous methods at removing slivers from and producing Delaunay optimal tessellations [50]. Each of these mesh generators has its own merits but none of these mesh generators supports parametric mesh generation.

We will take the freely available, widely published, nonparametric, open source 3-D

mesh generator TetGen [48, 49] which like all published mesh generators involves user input in the process of mesh generation. Here also we use a script file which uses a parametric description of the system to start the mesh from initial parameters and thereafter runs it seamlessly without stopping as the parameters are updated by the optimization process. Sample 3D meshes are shown in Figures 2.6 and 2.7

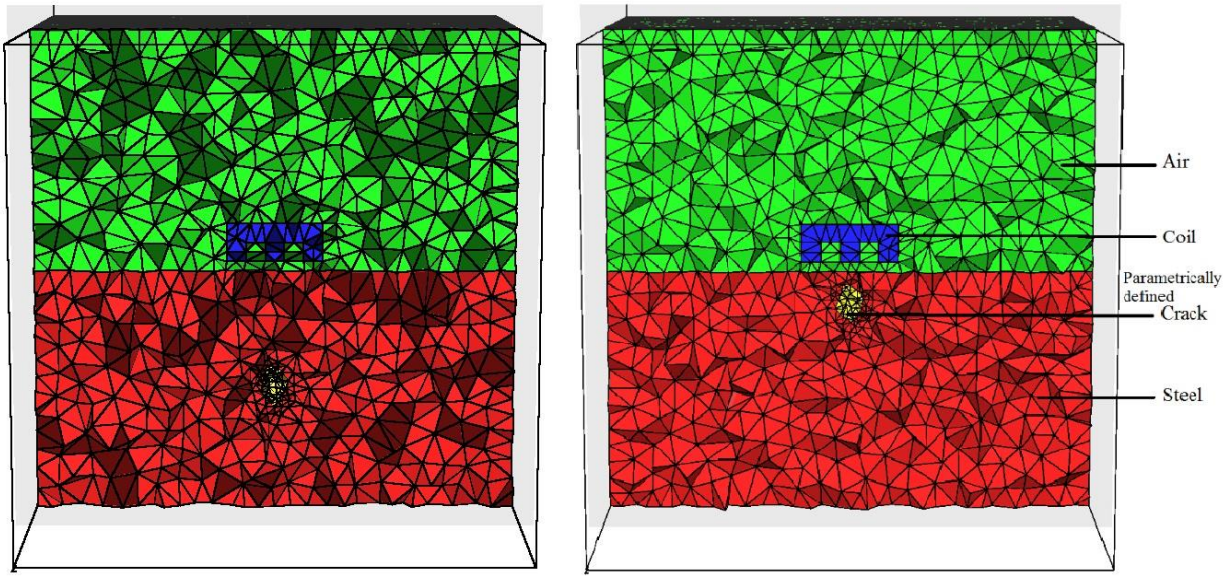

Figure 2.6: 3D mesh for NDE problem: As parameters change

2.2 Mesh Generation

2.2.1 Introduction

The finite element method requires the problem space to be split into a finite number of finer elements. The preferred element shape for two-dimensional problems is the triangle and for three-dimensional problems it is the tetrahedron [13]. This set of triangles/tetrahedrons is called the mesh. If the mesh is finer, it will produce a better result in FEA [13], although it will increase the processing time. If we can have a finer mesh only at the places where we want a more accurate result, then we can reduce the processing time considerably. This is called adaptive

mesh generation [13]. Apart from this, the shape of the triangles/tetrahedrons in the mesh has a great effect on the final solution of finite element analysis. If we have very obtuse angles in the triangles of the mesh, they will introduce considerable errors into the final solution. Therefore, all these facts have to be considered when generating a mesh for FEA problems.

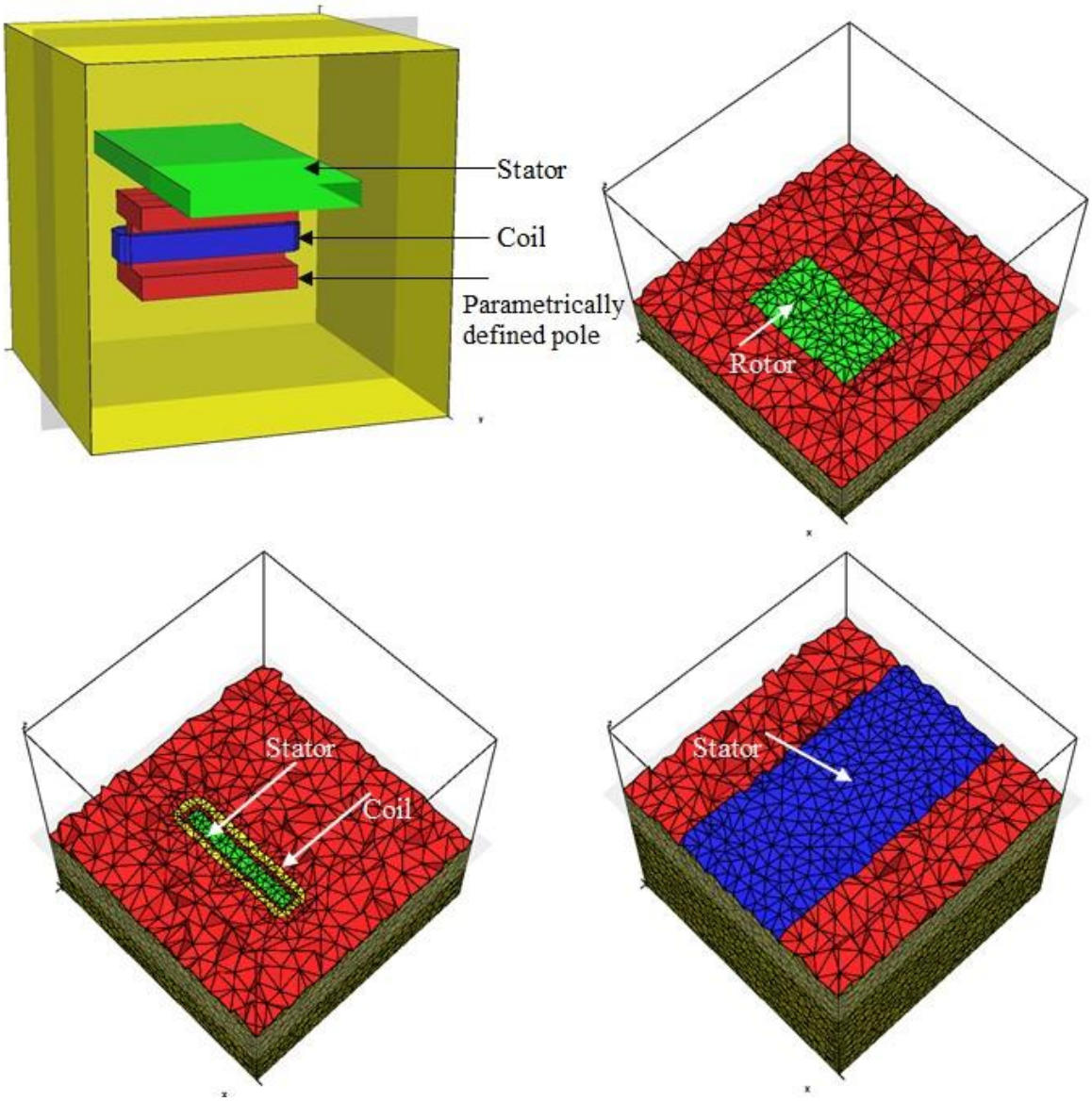

Figure 2.7: 3D Mesh for motor problem

There are many mesh generation algorithms available. Basically we can divide them into

two categories. They are,

1 Algorithms which generate a crude mesh to define the basic geometry and then refine it to get a good quality mesh
2 Algorithms which generate a fine mesh from the beginning,

The Advancing Front Algorithm [51] and Quad-tree Algorithm [52], Delaunay based Algorithms [25] are the examples of the second category. These methods can produce very good quality meshes. Delaunay based algorithms are well-known and commonly used algorithms for quality mesh generations and therefore we use them.

2.2.2 Delaunay Based Methods

A Delaunay based meshing approach is a concept which consists of two tasks:

1 The mesh points are first created by a variety of techniques; e.g. advancing front, octree, or structured methods
2 The Delaunay triangulation is first computed for the boundary without internal points. The mesh points are then inserted incrementally into the triangulation/tetrahedralization and the topology is updated according to the Delaunay definition.

There are many Delaunay triangulation algorithms; the incremental insertion algorithm, the divide and conquer algorithm, the plane-sweep algorithm etc. In this work, we take the freely available, widely published, non-parametric, open source two-dimensional (2D) mesh generator Triangle [25]. These three mentioned algorithms have been implemented in the Triangle [25] mesh generator. We then adapted it for seamless optimization [24].

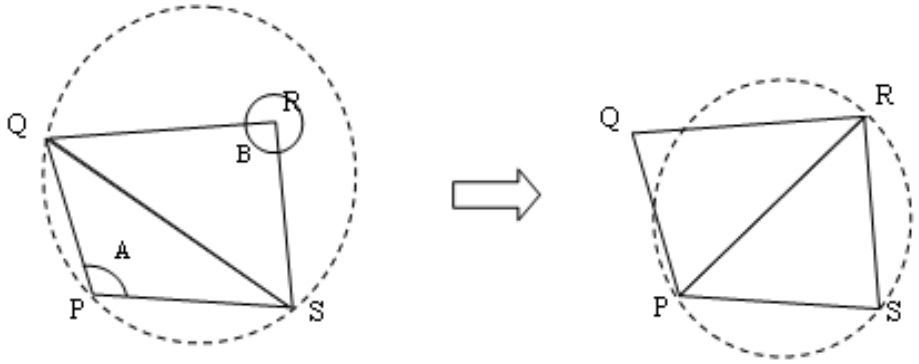

Figure 2.8: Applying Delaunay triangulation

2.2.3 Delaunay Triangulation and Constrained Delaunay Triangulation

2.2.3.1 Introduction

Delaunay triangulation [13] is a technique used to improve the quality of the mesh by simply rearranging the nodal connections that make triangles. This algorithm ensures that there will be no obtuse angles in the mesh other than in the triangles at boundaries. This is done by rearranging the triangles, if the uncommon point of the neighboring triangle lies inside the inscribing circle of one of the triangles, as shown in Figure 2.8. This can be identified by calculating the two angles corresponding to the uncommon points. By the properties of cyclic quadrilaterals, when the sum of these angles is greater than 180°, the triangles must be rearranged.

In Figure 2.8, the triangle QRS lies inside the inscribing circle of the triangle PQS. This can be recognized by summing the two opposite angles, A and B (These are the angles corresponding to the uncommon points for the two triangles, P and R). Since the sum of A and B is greater than 180° the two triangles are rearranged as PQR and PRS as shown in the figure. Now the point of the opposite triangles is not inside the inscribing circles. Constrained Delaunay triangulation is a generalization of the Delaunay triangulation that forces certain required

segments into the triangulation. An example is shown in Figure 2.9. Both triangles are in different regions where each may have different properties. So they cannot be flipped like the previous case

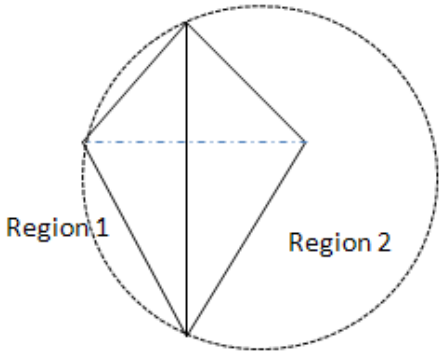

Figure 2.9: Example for constrained Delaunay triangulation

2.2.4 Algorithms for Constructing a Delaunay Triangulation

2.2.4.1 Introduction of Constructing a Delaunay Triangulation

There are many Delaunay triangulation algorithms; for example Divide-and-Conquer [53], Sweepline [54], Incremental insertion [55] etc. As Su and Drysdale [56] found, the divide- and-conquer algorithm is fastest; the second is the sweepline algorithm. The incremental insertion algorithm performs poorly, spending most of its time in point location. Su and Drysdale introduced a better incremental insertion implementation by using bucketing to perform point location, but it still ranks third. A very important development in the divide and conquer algorithm is partitioning the vertices with vertical and horizontal cuttings [53].

2.2.4.2 Divide-and-Conquer Algorithm

The point set *v* is divided into halves until we are left with two or three points in each subset. Then these smaller subsets can be linked with edges or triangles which is called a

Voronoi diagram. Now we have a set of Voronoi diagrams because we have a set of smaller subsets. In the conquer step, we merge the subsets to get the whole Voronoi diagram (see Figure 2.10). The dual of the Voronoi diagram is the mesh [25].

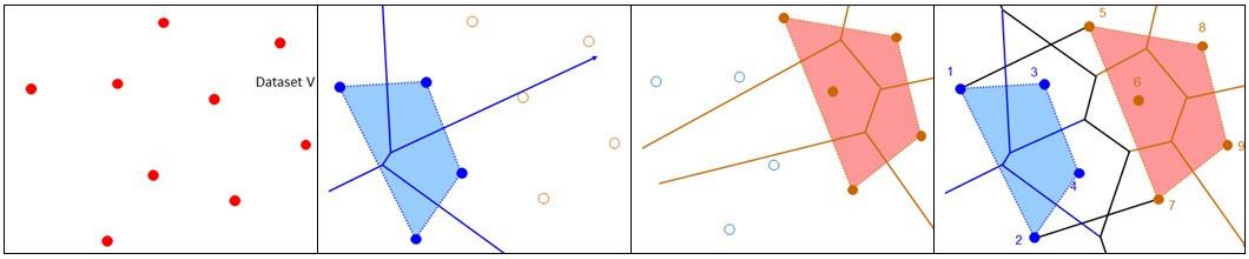

Figure 2.10: Divide and conquer algorithm

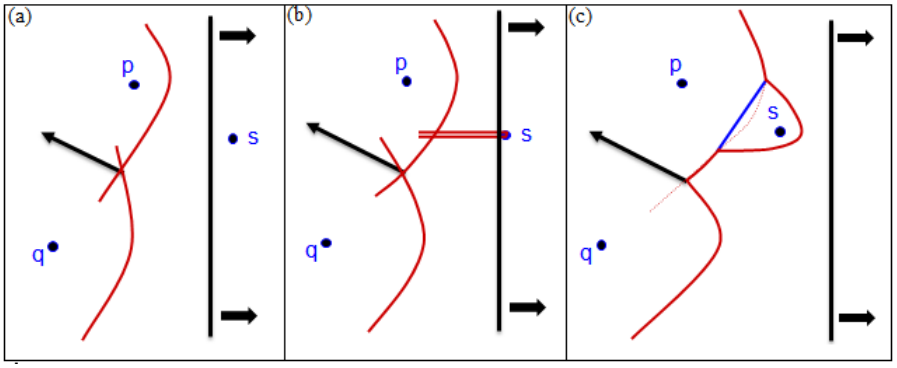

Figure 2.11: Sweep-line algorithm

2.2.4.3 Sweep-line Algorithm

The sweep-line algorithm uses a sweep-line which divides a working area into two sub-areas. This process constructs the Voronoi diagram - the dual graph of Delaunay triangulation (shown in Figure 2.11). This algorithm was introduced by Fortune [54]. Shewchuk [25] presented a successful algorithm for constructing a higher-dimensional Delaunay triangulation. Figure 2.11 explains how the sweep-line algorithm works. There is a vertical line which is called a sweep line in the Figure 2.11. When this line passes a point this algorithm creates a Voronoi diagram with other points which are already passed.

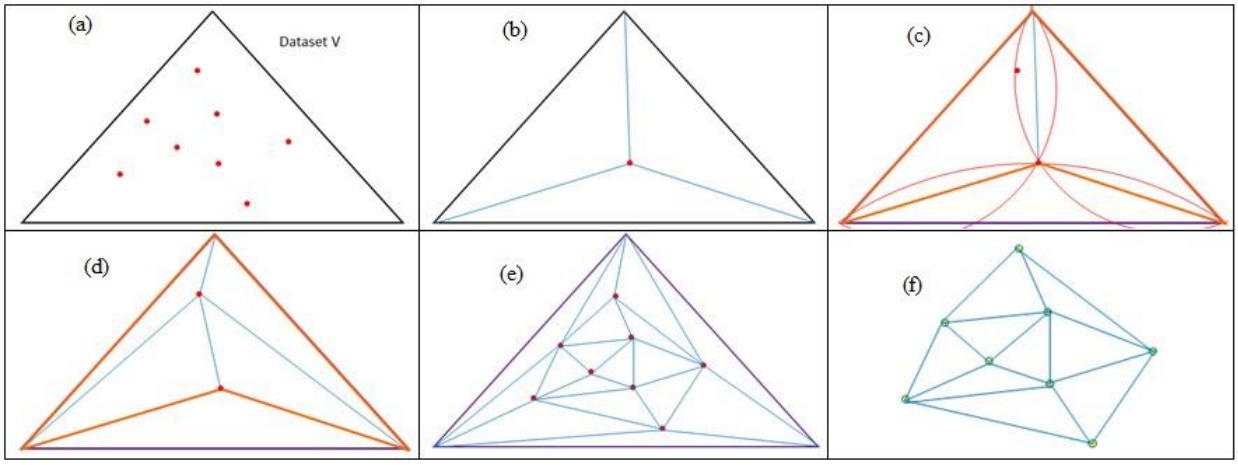

Figure 2.12: Incremental insertion algorithm

2.2.4.4 Incremental Insertion Algorithm

Here we generate a fictitious triangle containing all points of V in its interior. The points are then added one by one. Figure 2.12 (b) shows the mesh after the first point is added. Figure 2.12 (c) shows how to handle the insertion of the second or subsequent point. The idea is to draw circumcircles of a particular triangle where the new point is located and neighboring triangles, select the triangles whose circumcircles cover the new points, remove the interior edges of selected triangles; and finally, a new point is connected with every point of a created polygon. This algorithm always maintains the Delaunay triangulation of the points.

2.2.5 Mesh Refinement

There are many mesh refinement algorithms available. Most of them are based on Rupert's Algorithm [57]. They produce quality meshes with more nodes at regions where there are finer geometrical shapes and fewer nodes at other regions. The basic idea of the algorithm is to maintain a triangulation, making local improvements in order to remove the skinny triangles.

Figure 2.13 shows the basic idea of avoiding skinny angles.

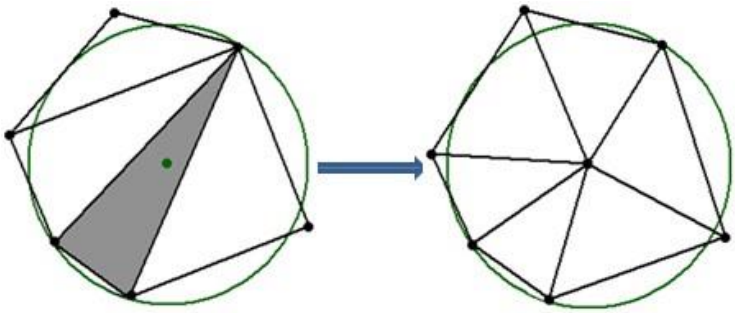

Figure 2.13: Delaunay mesh refinement

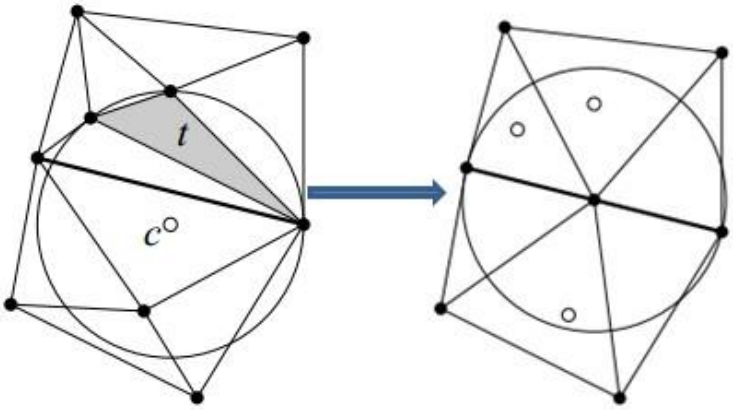

Figure 2.14: Delaunay mesh refinement between two regions

Both triangles are in different regions (shown in Figure 2.14); each may have different properties. So they cannot be refined like in the previous case. Here the algorithm follows the same idea without violating the boundary of separate regions

2.2.6 Three Dimensional Mesh Generation

Since almost every real world problem is three-dimensional, we extend our two-dimensional work to three-dimensional geometric parameterized mesh generation for optimization problems in design and NDE. Mesh generators in electrical engineering commonly use Delaunay

tetrahedralization and constrained Delaunay tetrahedralization for quality meshes. The Incremental Insertion algorithm is a well-known algorithm for tetrahedralization [48]. The worst case runtime of this algorithm is of $O(n^2)$, but the expected runtime for this algorithm is of $O(n \log n)$ [25]. Constrained Delaunay tetrahedralization was first considered by Shewchuk [25]. Gmsh [29] and TetGen [48] are the better-known free, open source 3D mesh generators. TetGen [48] uses a constrained Delaunay refinement algorithm which guarantees termination and good mesh quality. A three-dimensional Delaunay triangulation is called a Delaunay tetrahedralization. Ideas for Delaunay operation, constrained Delaunay triangulations, and mesh refinements are the same but only the dimension is different. 3D objects are usually represented by Piecewise Linear Complexes (PLCs) [48]. The design goal of TetGen is to provide a fast, light and user-friendly meshing tool with parametric input and advanced visualization capabilities. Even though TetGen and Gmsh [29] are great open source mesh generators, from an inspection of the code, it is very hard to use for non-stop optimization problems. For the non-stop optimization that ANSYS offers [46], it gives little information on the techniques employed. CEDRAT uses a script based scheme called GOTIt [58]. FGot, is offered free to students although the code is not accessible and therefore will not permit modification nor work for industry-scale problems [58]. Here TetGen [48] is used as a backend for parameterized meshes for optimization. Therefore we will develop it on our own and make it available as open source.

It is always possible to tetrahedralization a polyhedron if points are not vertices of the polyhedron. Two types of points are used in TetGen:

1 The first type of points are used in creating an initial tetrahedralization of PLC.

2 The second type of points are used in creating quality tetrahedral meshes of PLCs.

The first type of points is mandatory in order to create a valid tetrahedralization. While

the second type of points is optional, they may be necessary in order to improve the mesh quality.

2.3 Parameterized Mesh Generation

Parametric mesh generation is a very important part of finite element optimization problems. In optimization problems, parameters describe the device in terms of materials, currents, and dimension. During optimization, as these parameters are changed to minimize an object function, a new mesh has to be generated and a new finite element solution obtained to re-evaluate the object function. At each iteration of an optimization algorithm, given the variables as input, the mesh is generated without user intervention. Finite element mesh generators exist in the public domain, a few even based on a parametric device description. The typical mesh generator requires some man-machine interaction to define the points and boundary conditions, and does not work for non-stop optimization iterations for which we need a mesh dynamically evolving through the iterations with optimization variables as changing parameters. Such mesh generators as do exist are rare, commercial, and not easily available to researchers except at great cost and never with the code to modify them to suit individual needs.

2.4 New Approach to Parameterized Mesh Generation

We take the freely available, widely published, nonparametric, open source 2-D mesh generator Triangle [25] and 3-D mesh generator TetGen [48, 49] which like all published mesh generators (with the exception of commercially restricted ones whose methodology is not published) involves user input in the process of mesh generation. But for use in optimization we

cannot stop the iterations to make input [24]. To address these problems we use a script file which uses a parametric description of the system to start the mesh from initial parameters and thereafter runs it seamlessly without stopping as the parameters are updated by the optimization process [24]. The script file provides the user input while the code is iteratively running, input that is normally made in the mesh generator being used, but for which the optimization iterations cannot stop [24]. Figure 2.15 explains our approach to parametrized mesh generation. In the first step, the initial input which is described in the following sections is given to our mesh generator. Next, the mesh generator calls our chosen open source mesh generator to generate the mesh. After that, the FEM solver uses the mesh to solve the problem. Next, the optimization algorithm updates the parameters which are accepted by our new mesh generator to generate the new mesh.

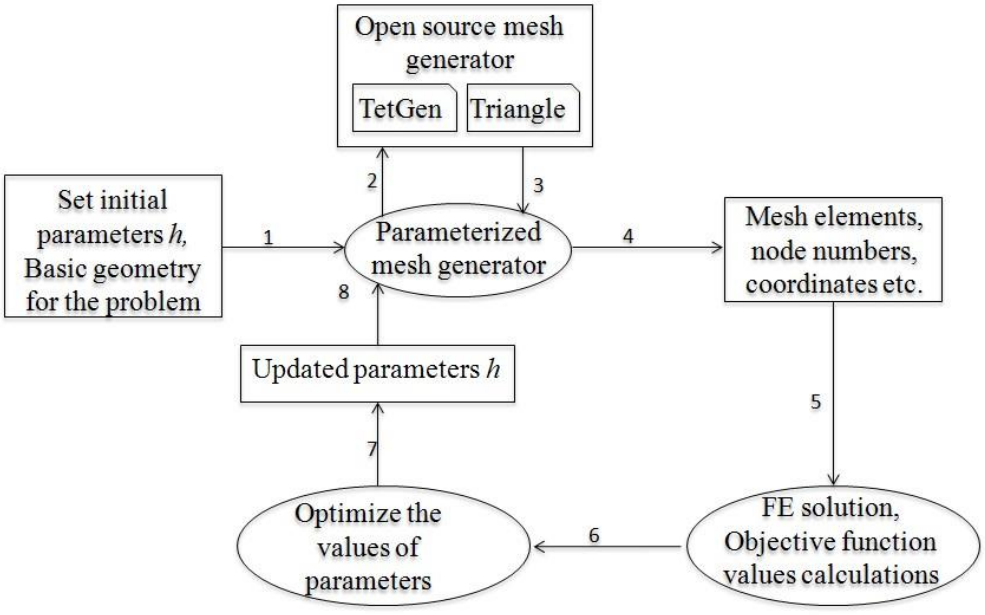

Figure 2.15: My approach to parameterized mesh generation

2.5 Data Structure and User Interface

2.5.1 Data Structure

The data-structure used in these mesh generator software suites contains the following collections of objects:

1. Points list
2. Regions list
3. Properties list
4. Variable points list
5. Measuring points list
6. Segments list
7. Mesh details list (triangles/ tetrahedrons)
8. Holes list
9. Boundary conditions

- Points List: The point collection contains all the points used in the problem definition and solving process. Each point contains the coordinates of the relevant finite element node. For the three-dimensional mesh generator the list has x, y and z coordinates. For the two-dimensional mesh generator, it should have x and y coordinates only.

- Regions List: The region list contains all the regions used in the problem definition. Region here means a material- source combination which has different physical characteristics.

- Properties List: Properties list contains all the properties of each region. Each region has a set of properties.

- Variable points list: The variable points list contains the information about the points to be moved according to the changes of the parameters, during optimization.

- Measuring points list: The measuring points list contains the points in the solution space where we want to find the potentials, flux density, etc.

- Segments list: The segments list contains the edges of the problem model. Each problem may have many segments.

- Mesh details list: The mesh details list contains the triangles/tetrahedrons of the mesh. This collection is empty until the mesh is generated. For a two-dimensional mesh generator each triangle contains references to its three vertex points. For a three-dimensional mesh generator each tetrahedron contains references to its four vertex points.

- Holes list: Holes are a special kind of region where we do not need to generate the mesh. A hole list contains all the holes used in the problem definition.

- Boundary conditions: There are two types of boundary conditions that are usually used in FEA problems. They are
    1. Neumann boundary conditions
    2. Dirichlet boundary conditions

A Dirichlet boundary condition means the potential along the given boundary is fixed and a Neumann boundary condition means the derivative of the potential along the given boundary is fixed, and usually zero. Dirichlet boundary conditions can be implemented by keeping the potential of all the points on the given boundary to be fixed at their given value. The user can select any segment, and define the potential of that segment. If the potential of the segment is set, then all the points which will be added onto that line will get this potential automatically. The boundaries that do not implement Dirichlet conditions will automatically act as Neumann boundaries during the FEA process. This is because it is natural to the finite element formulation [13]. Therefore no special provisions are needed to define Neumann boundaries.

```
# A set of pointsin 2D(* WITHOUT VARIABLE POINTS).
# Number of nodes is 9 number of variables is 5
9 5
# And here are the nine points.
1 0.0 0.0
2 10.0 0.0
3 20.0 0.0
4 10.0 10.0
5 0.0 10.0
6 2.0 2.0
7 4.0 2.0
8 4.0 4.0
9 2.0 4.0
# variable points
# number of points in first draw. Then coordinates
10 20.0 3.0
```

Figure 2.16: Sample input file for mesh generator

2.5.2 User Interface

2.5.2.1 Introduction

A proper user interface is very important for good software. If the user interface is not friendly to use, even if it is very powerful, most users will not be able to use it effectively. Therefore the user interface is carefully designed and used in this software as described in the next subsection.

2.5.2.2 Defining the Geometrical Shape

Since we are providing the code as open source, it is necessary to describe it for other users to re-engineer the code. This software is made to import drawings from the text file format. Figure 2.16 shows the sample input file of our mesh generator adapting Triangle using a script file to run non-stop from iteration to iteration without manual intervention.

1 The mesh generator code does not care about lines which start with #. We can write comments using the # sign.

2 First interpreted row: <a number, a number> - The first number represents the num- ber of nodes in the domain; the second number represents the number of variable points. These variable points are also nodes but their coordinates may vary with the optimization iterations.

3 From the second row to row number 10 (that is, 9+1) in the domain, there are, associated with that row,

<a integer number, a floating point number, a floating point number >

The integer represents the node number. These must be numbered consecutively, start- ing from one. The two floating point numbers represent the coordinates of this node. For the three-dimensional mesh generator, the three floating point numbers represent the coordinates of this node.

4 The next segment of this input file represents variable points. These are also in the same format as the previous.

5 In the third part of the input file we have segment details. The first row of this file has the number of segments. From the second row onwards it has 4 columns. The first column involves the segment numbers which must be numbered consecutively, starting from one. The next two columns are node numbers. Each row represents a segment. The fourth column is a marker. A marker has different integer values; it can be used to define the boundary condition. Here -1 means it is not on a boundary. If we have to define the boundary condition we have to give any positive integer to the marker. Then we can assign the boundary value using these markers. For the three-dimensional mesh

generator, the first row of this segment of this file has the number of faces and a boundary marker. Each face has a list. The first part of the list has the number of polygons in the face, number of holes on the surface and the boundary marker. From the second row onwards, polygons and holes of the surface are defined.

6 The next segment of this file is the definition of boundaries. The first row contains the number of boundary conditions. From the second row onwards the first column represents the numbering; the second column is a marker number which has been already defined in the previous part (the segment part). The third column represents the boundary value for a particular marker.

7 The subsequent segment is a definition of the regions of the problem. Here a different region means different materials so it has different properties. The first row represents the number of regions in the domain. From the second row, each row has five columns. The first column represents the numbering as usual. The second and the third columns represent the coordinates. These coordinates are used to identify the region. The point may be any point in the relevant particular region. The fourth column is an integer which starts from 1. It can be used to assign properties to these regions. The next column is not used here because it is an area constraint coming from Triangle but, as mentioned, it is not used by us.

8 The next segment of this input file represents the number of holes. The hole is a region in which we do not want to generate the mesh. In the first step we define the number of holes in the problem domain. Form the next row, there are three columns. The first column represents the hole number. The second and third columns represent the x and y coordinates of any point within that hole.

9 The segment thereafter is for the measuring point list for object function evaluation. The first part is the number of measuring points where we are going to calculate the solution to get the target solution. Our source code helps users to identify the errors in the input file. It works very efficiently for any shape of problem domain. The sample inputs files are attached in this thesis as Appendices B and C. Users customize their own problem very efficiently as tried out in our lab [4, 3, 24]. This software is easy to use. This software is well supported in any operating system, i.e., Linux/Unix, Windows etc.

2.5.3 Post-processing of Meshing

Once we triangulate/tetrahedralize the problem domain, we have to define the boundaries and boundary values. Upon triangulation/tetrahedralization, we have an element list (node numbers), the properties of regions and point list (for FE solution). We do not need to calculate the solution of known nodes so we have to separate the known and unknown nodes. This step is known as renumbering the nodes which is also to reduce the profile of the matrix [13]. In this process, we

1. Define the boundary (generally using segment numbers in 2D, faces in 3D)
2. Get the boundary values (different boundaries may have different boundary values)
3. Get all nodes which are on boundaries (We used the segment marker list to determine the boundary nodes)
4. Separate boundary elements from non-boundary elements; and separate unknown nodes from known nodes
5. Give the first set of numbers for the unknown nodes and the last set of numbers for the known nodes

6. Renumber the whole point list based on new numbering.
7. Renumber the node entries in the triangle list based on the new numbering system.
8. Get all properties for the particular regions
9. Assign these properties to all corresponding triangles.

Since real world problem size is typically very large, this renumbering process takes a very long time. Algorithm 2.1 describes the regular renumbering process. This algorithm is very inefficient because each node will be searched for in an index array. The order of this algorithm is $O(n^3)$. This step can be improved. In this work the traditional algorithms have been improved based on the merge sort technique.

Algorithm 2.1 Renumbering

```
1: { t old triangle/tetrahedron list (node numbers), n number of elements (triangles/tetrahedrons),
   index- rearranged node numbers(unknowns first; knowns last) and (u renumbered trian-
   gle/tetrahedron list (node numbers}
2: for i = 0 to n − 1 do
3:    for j = 0 to (3or4) − 1 do
4:       temp ← t(i + n ∗ j)
         {t is a one-d array}
5:       for k = 0 to n − 1 do
6:          if temp == index(k) then
7:             u(i + n ∗ j) ← k
8:             break
9:          end if
10:      end for
11:   end for
12: end for
```

2.5.4 Approach to Renumbering

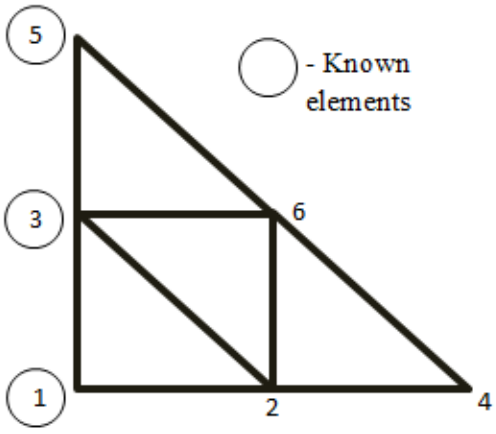

Figure 2.17: Simple example problem

Instead of searching for every element from the whole list, this thesis tracks the node number changes and updates the node numbers of the mesh. Figure 2.17 shows a simple example finite element problem. The boundary elements are circled. Figure 2.18 (a) shows the node numbers which are assigned in the mesh generation process. Figure 2.18 (b) presents the rearranged nodes which are separated based on whether the nodes are of known or un- known values. Figure 2.18 (c) shows the new numbering system. The boundary elements are shown in the gray boxes in Figure 2.18. The corresponding old number list (c) is renumbered in (b). Figure 2.19 shows the new numbering of the nodes. Let us take the array of Figure 2.18 (a) and use a new index which has 1, 2, up to the number of elements. We sort the array of Figure 2.18 (b) using the merge sort algorithm which will be described in the following section. We apply every operation of the sort algorithm to a new index array. The resultant arrays are shown in Figure 2.20. Now if we want to get a renumbered value of a particular node number we can directly access it from the updated index array. For example 1 is replaced by 4, 2 by 1, 3 by 5, and so on.

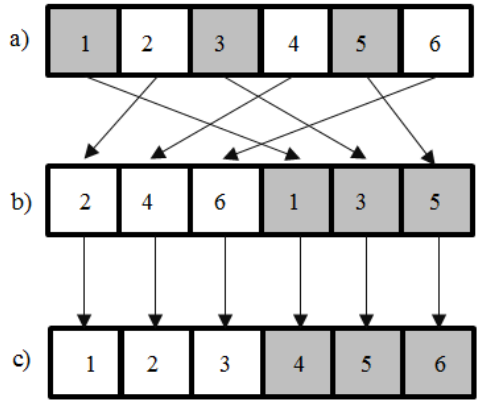

Figure 2.18: Numbering and renumbered nodes

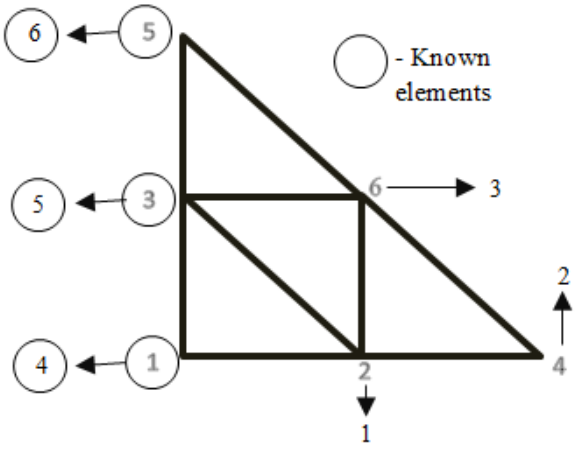

Figure 2.19: Renumbered nodes

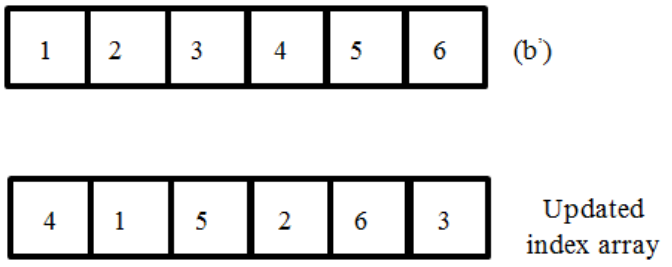

Figure 2.20: Sorted version of (b) and corresponding index changes

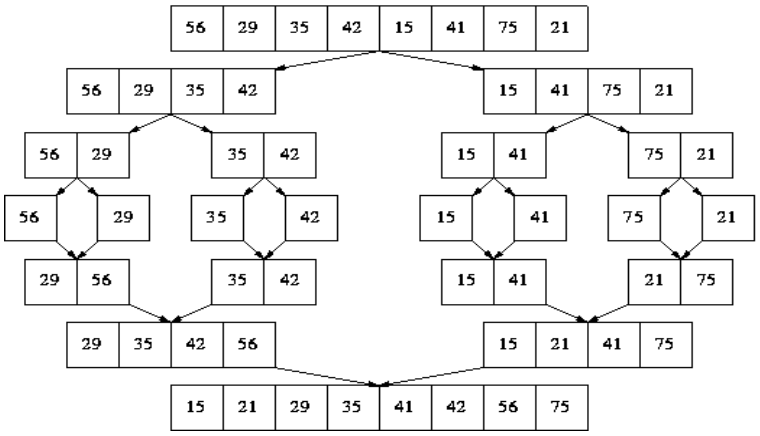

Figure 2.21: Merge Sort

2.5.5 Merge Sort

Sorting is a technique that arranges the elements in a certain order. There are many sorting algorithms such as counting sort [59], bucket sort [59], radix sort [59], merge sort [59], heapsort [59], quicksort [59] etc. Each algorithm has it own advantages. Merge sort is one of the best sort algorithms which has *n log (n*) time complexity for each average, best and worst case time complexities [59]. Another very important reason for selecting merge sort is that this algorithm is easily parallelizable - parallelization on the GPU is the main theme of this thesis. Figure 2.21 [60] describes the merge sort in a graphical way. First the list is split into two pieces and then each is split into two further pieces. This process is repeated until arriving at the singleton list. Then we work our way back up the recursion by merging together the short arrays into larger arrays. Algorithms 2.2 and 2.3 describe merge sort.

Algorithm 2.2 Merge Sort

1: $MergeSort(A, right, left)$
2: **if** $left < right$ **then**
3: $mid \leftarrow floor((left + right)/2)$
4: $MergeSort(A, left, mid)$

Algorithm 2.2 (Cont'd)

```
5:    MergeSort(A, mid + 1, right)
6:    Merge(A, left, mid, right)
7: end if
```

Algorithm 2.3 Merge

```
1: Merge(A, left, mid, right)
2: i ← left, j ← mid + 1, k ← 0,
3: while i ≤ mid and j ≤ right do
4:    if A(i) < A(j) then
5:       B(k) ← A(i)
6:       k ← k + 1
7:       i ← i + 1
8:    else
9:       B(k) ← A(j)
10:      j ← j + 1, k ← k + 1
11:   end if
12: end while
13: while i ≤ mid do
14:   B(k) ← A(i)
15:   k ← k + 1
16:   i ← i + 1
17: end while
18: while j ≤ right do
19:   B(k) ← A(j)
20:   k ← k + 1
21:   j ← j + 1
22: end while
23: k ← k − 1
24: while k ≥ 0 do
```

Algorithm 2.3 (Cont'd)

25: $A(left + k) \leftarrow B(k)$

26: $k \leftarrow k - 1$

27: **end while**

2.5.6 Modified Form of Merge Sort for Renumbering

As we described in section 2.5.4, we track the node number changes using a form of merge sort instead of searching for every element from the whole list. This new algorithm has *n log (n)* time complexity but the traditional method has time complexity of O(n3). We define an array of size given by the number of nodes in the mesh. The array has 0, 1, 2, up to (number of nodes - 1). We applied every operation of the sort algorithm to this newly defined array. Algorithms 2.4 and 2.5 describe the idea underlying what we have used.

Algorithm 2.4 Merge Sort-Modified

1: $MergeSort(A, IdxArray, right, left)$

2: **if** $left < right$ **then**

3: $mid \leftarrow floor((left + right)/2)$

4: $MergeSort(A, IdxArray, left, mid)$

5: $MergeSort(A, IdxArray, mid + 1, right)$

6: $Merge(A, IdxArray, left, mid, right)$

7: **end if**

Algorithm 2.5 Merge-Modified

1: $Merge(A, IdxArray, left, mid, right)$

2: $i \leftarrow left, j \leftarrow mid + 1, k \leftarrow 0,$

3: **while** $i \leq midandj \leq right$ **do**

4: **if** $A(i) < A(j)$ **then**

Algorithm 2. 5(Cont'd)

$B(k) \leftarrow A(i)$

$C(k) \leftarrow IdxArray(i)$

$k \leftarrow k + 1$

$i \leftarrow i + 1$

else

$B(k) \leftarrow A(j)$

$C(k) \leftarrow IdxArray(j)$

$j \leftarrow j + 1$

$k \leftarrow k + 1$

end if

end while

while $i \leq mid$ **do**

$B(k) \leftarrow A(i)$

$C(k) \leftarrow IdxArray(i)$

$k \leftarrow k + 1$

$i \leftarrow i + 1$

end while

while $j \leq right$ **do**

$B(k) \leftarrow A(j)$

$C(k) \leftarrow IdxArray(j)$

$k \leftarrow k + 1$

$j \leftarrow j + 1$

end while

$k \leftarrow k - 1$

while $k \geq 0$ **do**

$A(left + k) \leftarrow B(k)$

$IdxArray(left + k) \leftarrow C(k)$

$k \leftarrow k - 1$

end while

Chapter 3

Low Memory High Speed FEM Solvers Using the GPU

3.1 Introduction

It has been more than 20 years since genetic algorithm (GA) based optimization was first used in finite element optimization [5, 6, 7, 8, 9, 10]. Since GA is practicable and gives a faster solution when parallelized [3], GA has been used for optimization in FE. The object function corresponding to every member $\bar{h}$ of a population has to be computed many times to find the minimum. The many members $\bar{h}$ form the genetic search space. Since $\bar{h}$ consists of dimensions and materials of a particular design being examined for its goodness [3], for those dimensions a mesh is constructed, the finite element problem solved and the object function evaluated (see figure 3.1). The object function itself is computed from a finite element solution involving a matrix equation. Thus we may treat the object function computation as a kernel and launch it on multiple threads, each for a different member of the population. Then within that kernel, we can parallelize the matrix equation solution. In genetic algorithm based finite element optimization [61, 62], several copies of the matrix are held on the GPU and the corresponding solutions attempted. Limited memory is also a very big issue in GPUs [23]. This part of the work mainly focuses on low memory and high speed finite element solvers.

As we already mentioned, the GA based optimization method presents a huge computational load. Powerful PCs are capable today of solving large matrix equations in a few seconds, sometimes using a Graphics Processing Unit (GPU). GPU based finite element

computation offers massive parallelization [63].

The finite element solution of field problems requires the solution of large-sized matrix equations leading to large waiting times [13]. To address these parallel computations were used at one time [64, 65]. But the speedup was limited by the fact that on the shared memory parallel computers, there was a memory bottleneck which typically then allowed 4, 8, 16 or rarely 32 processors with more computing elements meaning exorbitant cost. For an n- processor machine with one processor dedicated to book-keeping on what the other processors were doing, the best speedup was n-1 and always a little less because of communication and waiting issues. Recently the graphics processing unit (GPU) has been shown to speed up the matrix solution part in the finite element solution [34, 66]. This was a major advance in finite element computational efficiency.

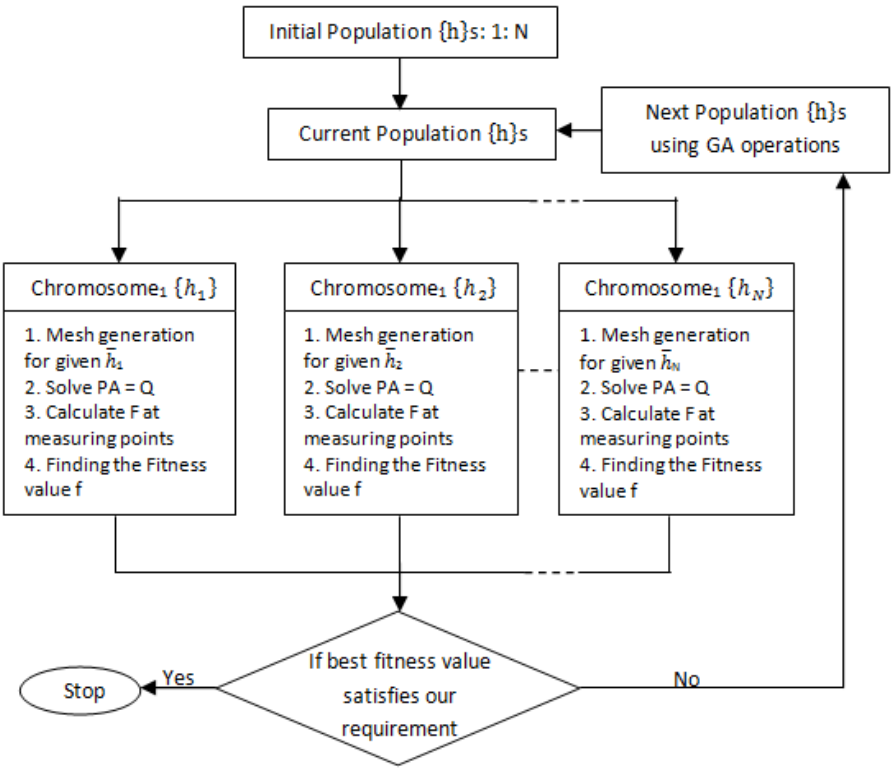

Figure 3.1: Finite element optimization using genetic algorithm

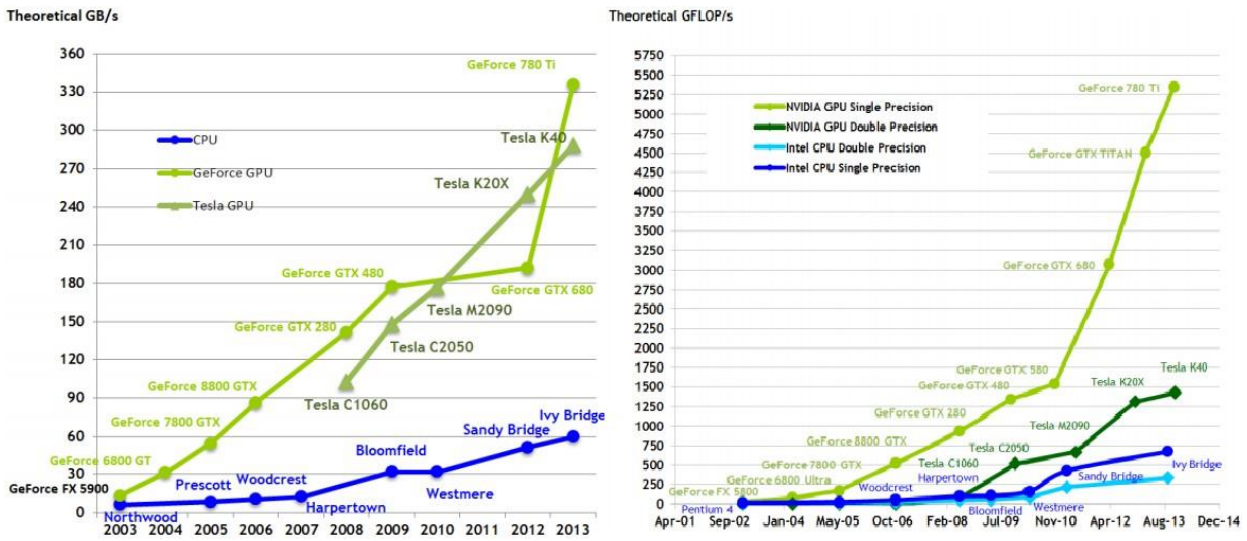

Figure 3.2: Floating-point operations per second and memory bandwidth for the CPU and GPU

Parallelization is the best approach to speeding up as bigger problems are tackled in field computation [13, 64]. However as noted, the need for shared memory between processors was a bottleneck because machines with more than 8 processors were very expensive.

3.2 General Purpose Computing on a Graphics Processing Unit (GPGPU)

The GPU is a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines [67]. In November 2006, Nvidia group introduced CUDA which is a general purpose parallel computing platform and programming model. The GPU has its own memory, up to 24 GB in current configurations [68]. This device (GPU) memory supports a very high data path using a wide data path. The CPU has a few cores which have been used for optimized sequential processes. In contrast, the GPU has thousands of small more efficient cores which have been used for massive parallel processes. The GPU has tremendous

computational horsepower and a very high memory bandwidth (shown in Figure 3.2 [69]).

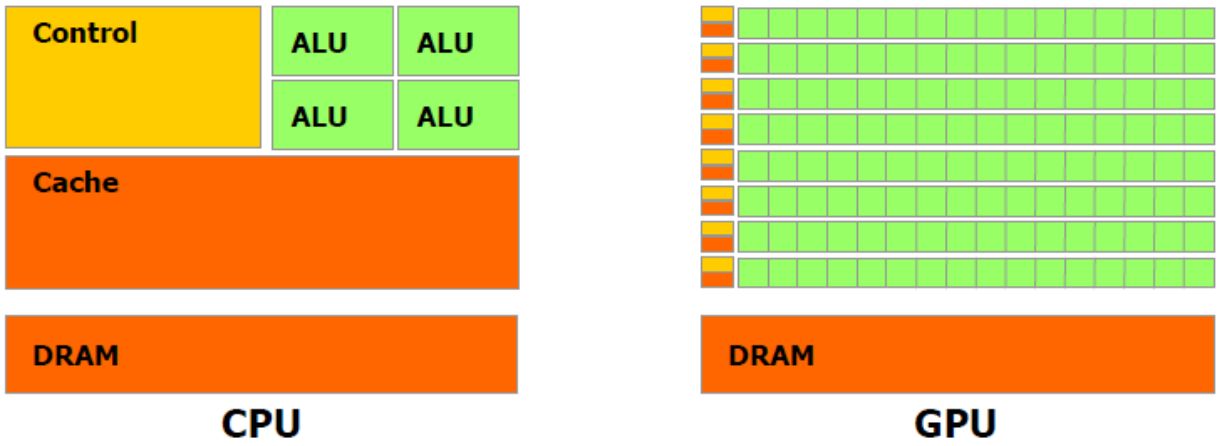

Figure 3.3: The GPU devotes more transistors to data processing

GPUs are used for highly parallel computation. Therefore they are designed using more transistors for data processing (ALUs) rather than data caching and flow control (shown in Figure 3.3 [69]). The GPU is well suited for data-parallel processing. In GPU FE computation, data-parallel processing maps data elements to parallel processing threads.

GPU computing uses the GPU to accelerate the computational speed of very large scientific and engineering problems. Generally a GPU has thousands of cores, For example the Tesla K40 GPU has 2880 cores [68]. Here cores mean a number of computing components. Nowadays there are so many multi-core processors available in the market but the number of cores is very limited in CPUs.

The GPU for general purpose calculations instead of graphics rendering is called general purpose computing on graphics processing unit (GPGPU). The Nvidia GPU CUDA architecture is composed of streaming multiprocessors (SMs), each containing a number of streaming processor cores (SPs) with on-chip memory, and a single instruction unit. All SMs have access

to global memory, the off-chip memory (DRAM), which has a latency of several hundred clock cycles. Thus unlimited speedup is possible unlike with shared memory machines. The until-recent restriction that a kernel launched on parallel threads cannot launch another kernel in further forking from a fork [12] is no serious shortcoming since with multi-processor systems, even though we can fork from a fork, we usually do not have spare processors to assign. However in a recent development CUDA dynamic parallelism has been made available on the SM 3.5 architecture GPU [70] and this is available on PCs now.

In finite element analysis the coefficient matrix is very large when dealing with real world problems; for example in reconstructing cracks in inverse non-destructive evaluation problems and device design problems. We use two different techniques to overcome this memory problem:

1. Use sparse storage schemes to store the coefficient matrix and solve the matrix equation using the GPU [23]

2. Use element by element FEM - performs operations on the local finite element matrix $[P]^{L}$ that corresponds to operations on the global matrix and therefore does not require storage for the larger global matrix($[P]$) [65]

However, the need for shared memory between processors was a bottle neck because machines with more than 8 processors were very expensive. With n processors we could at best get a speedup of n-1. Using the GPU of PCs is a new alternative [71, 72]. The NVIDIA GPU CUDA architecture is composed of streaming multiprocessors (SMs), each containing a number of

streaming processor cores (SPs) with on-chip memory, and a single instruction unit. All SMs have access to global memory, the off-chip memory (DRAM), which has a latency of several hundred clock cycles. Thus unlimited speedup is possible unlike with shared memory machines.

We will use well known storage schemes such as the profile (sky line) [13] storage schemes and the sparse storage scheme [13, 73] (also known as the compressed sparse row scheme). We revive the old element by element finite element solvers from the early 1980s for working on a highly memory limited IBM PC 282 to launch thousands of CUDA threads on the GPU architecture. We will examine different numerical techniques such as conjugate gradients (CG), preconditioned conjugate gradients (PCG), Jacobi, bi-conjugate gradients etc. to get the high speed solution we seek. These ideas are explained below.

3.3 Related Works

To Wu and Heng [71] should go the credit for first exploiting as far back as in 2004 the CUDA architecture in parallelizing FEM computations. They focused their attention on the conjugate gradients matrix solver where the most gains could be made. It took until 2010 for CUDA FEM computations to enter seminal electrical engineering works and that too without reference to Wu and Heng's seminal work [35, 36].

Kiss *et al.* [34] have recently applied EbE processing to solve their finite element equations from a first order tetrahedral mesh using the conjugate gradients algorithms. Their EbE method is different from EbE in references [65, 74]. They implement the bi-conjugate gradient technique [75]. Since the matrix is not formed, they use the diagonal to implement Jacobi preconditioning. Fernandez *et al.* [33] decoupled the solution of a single element from that of the whole mesh (Figure 3.4 [33]), thus exposing parallelism at the element level. Individual element

solutions are then superimposed node-wise using a weighted sum over concurrent nodes. They used Jacobi iterations to calculate the solution of each local matrix in parallel and then couple local solutions using weighted average enforcing continuity. For example node number 1 in Figure 3.4 [33] is replaced by numbers 1 and 6 in new numbering system; number 1 is in element e1 and 6 in e2. Once calculated the local solutions of elements e1 and e2, need to form the global solution using the following formula [33],

$$\varphi_{1(global)} = \varphi_{1(local)} \frac{k_{11}}{k_{11} + k_{66}} + \varphi_{6(local)} \frac{k_{66}}{k_{11} + k_{66}} \tag{3.1}$$

where k_{11} and k_{66} are local matrix elements (shown in Figure 3.4 [33]), $\varphi_{6(local)}$ is the solution of element e2 and $\varphi_{1(local)}$ is the solution of element e1 [33]. Fernandez *et al*. compared different generations of GPUs getting speedups with conjugate gradients of up to 14 times and 111 times for the 8800GT and the GTX480 GPUs respectively, compared to optimized CPU results.

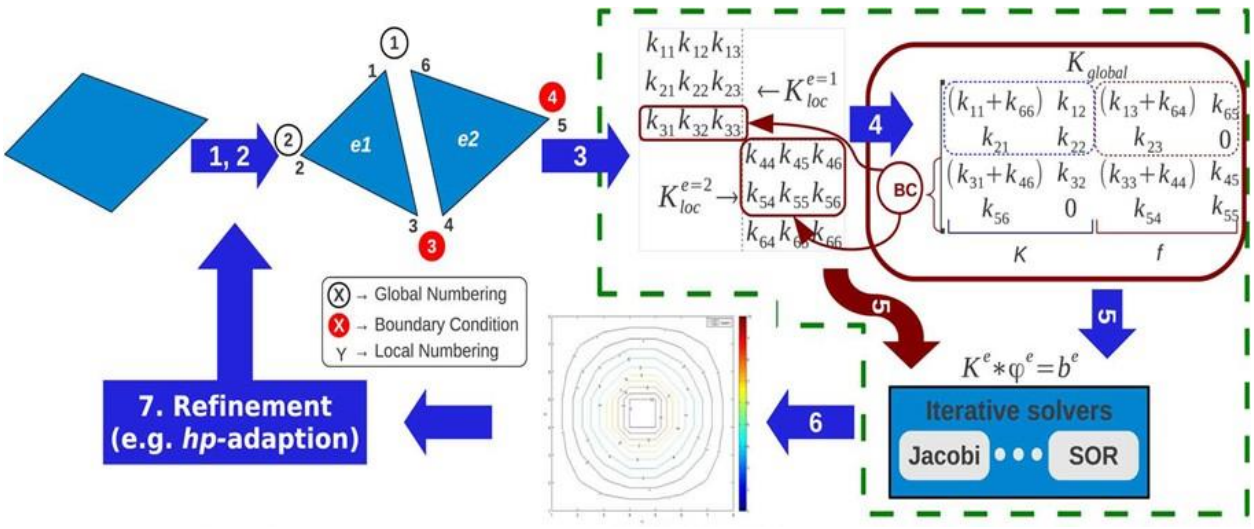

Figure 3.4: Steps in the classic finite element method (FEM) and the proposed changes for the FEM-SES method enclosed within the dashed line

3.4 Element by Element Solvers

3.4.1 Element by Element with Jacobi Algorithm

To overcome the memory limitation of 612 kB on the IBM PC 286 of the early 1980s, Hughes *et al*. [74] introduced EbE processing. The then available memory of 612 kB was not enough to hold even a trivial finite element matrix. What we used to do following Hughes *et al*. [74] was not form the coefficient matrix [P]. Instead, recognizing that [P] is assembled from the small element matrices $[P]^{L}$ (of size 3 × 3 in magnetics with triangular first order elements), according to

$$[P] = \sum_{elements} [P]^{L} \tag{3.2}$$

all the operations meant for [P] we performed multiple times on each $[P]^{L}$. We took an element, computed its $[P]^{L}$, did the operation meant to be done on [P], added the result to that from other elements already dealt with as indicated in equation 3.2, threw away $[P]^{L}$, went to the next element and so on. The non-forming of the coefficient matrix meant that only iterative schemes of solution without decomposing [P] in any way could be done. This restriction therefore excluded the powerful iterative incomplete Cholesky conjugate gradients (ICCG) scheme of solution usually preferred in finite elements work because an incomplete Cholesky factor of [P] is required [13]. In solving $[P]\{\varphi\} = \{Q\}$ the Gauss-Seidel iterations commonly used by power engineers, is an improvement on the older Gauss iterations [13]. In Gauss-Seidel, in each iteration *m + 1* we use the latest available values of the unknowns φ, using equation *i* of $[P]\{\varphi\} = \{Q\}$ to compute φ_i treating only φ_i as the unknown and all the other variables as known and given by their latest values in the iteration cycle:

$$\varphi_i^{m+1} = \frac{1}{P_{ii}}\left\{ Q_i - \sum_{k=1}^{i-1} P_{ik}\varphi_k^{m+1} - \sum_{k=i+1}^{n} P_{ik}\varphi_k^{m} \right\} \tag{3.3}$$

with obvious modifications for $i = 1$ and $i = n$. In this algorithm φ_{i-1} must be computed before φ_i. Here at iteration $m + 1$, computing φ_i in the order i=1 to n, φ is at values of iteration $m + 1$ up to the $(i-1)^{th}$ component of { φ } and at the value of the previous iteration m for values after i. It is therefore necessarily a sequential algorithm.

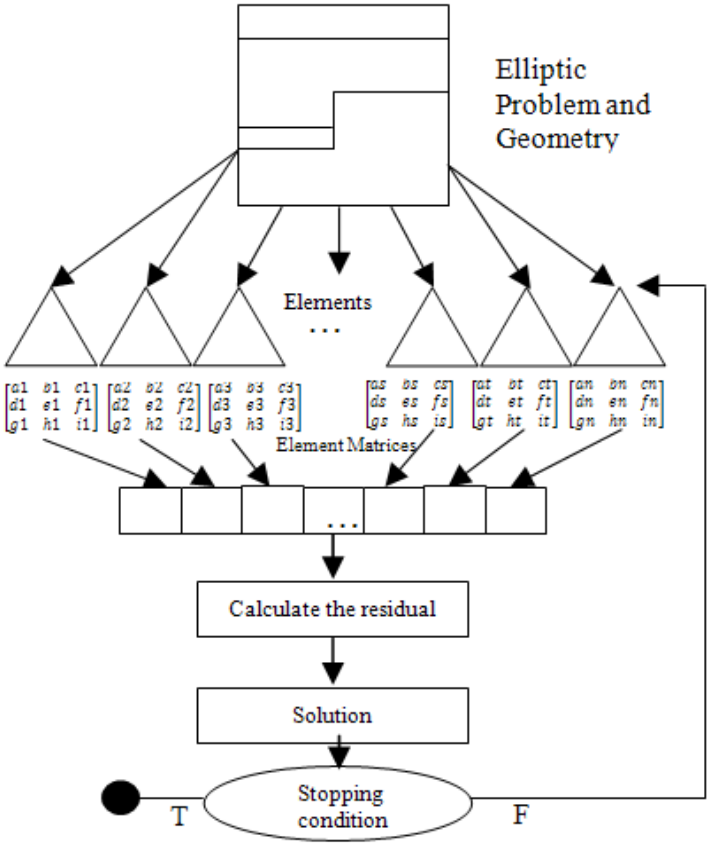

Figure 3.5: Proposed method in flow chart

When EbE processing was developed for the Gauss algorithm [65], in solving $[P]\{\varphi\} = \{Q\}$, the older displaced Gauss iterations [65] use the values of the old iteration m for computing every $\{\varphi_i\}$ in iteration $m + 1$. Therefore the computation of a particular $\{\varphi_i\}$ value is

independent of the computation of all other $\{\varphi_i\}$ values for that iteration and therefore parallelizable:

$$\varphi_i^{m+1} = \frac{1}{P_{ii}}\left\{ Q_i - \sum_{k=1}^{i-1} P_{ik}\varphi_k^m - \sum_{k=i+1}^{n} P_{ik}\varphi_k^m \right\} \tag{3.4}$$

This is inefficient in the context of sequential computations. But in the case of parallelization it is highly efficient as was laid out by Mahinthakumar and Hoole [65] and Carey *et al*. [76] for shared memory systems. Speedups were just below (n-1) where n is the number of processors. If [*D*] is the matrix [*P*] with all off diagonal elements eliminated, then the Gauss iterations yield

$$[D]\{\varphi^{m+1}\} = \{Q\} - [P - D]\{\varphi^m\} \tag{3.5}$$

Our proposed implementation first computes the diagonal vector [*D*] of the unformed global matrix [*P*], the right hand side of the finite element equation $\{Q\}$ and initial solution vector, and saves them as two one dimensional arrays. This computation will be done only once. We can parallelize this computation. In the second step, our algorithm calculates the residual components ri, i = 1 to n, of $\{r\}$ in parallel:

$$r_i = r_i - \sum_{k=1}^{i-1} P_{ik}\varphi_k^m - \sum_{k=i+1}^{n} P_{ik}\varphi_k^m \tag{3.6}$$

Algorithm 3.1 shows the element-by-element Gauss iterations. In that *n* is the total number of unknowns, $\{V\}$ is a vector which contains the vertices of an element, $\{r\}$ is the residual vector, $[P\ ]^L$ is the local coefficient matrix, $\{Q\}$ is global right hand side vector, $\{D\}$ is the diagonal vector of the global matrix [*P*], and $\{\varphi\}$ is the current solution vector. *NC* and *NR* represent the

column number and row number respectively of the unformed global matrix in which the local matrix term $[P]^L(i, j)$ stored as the column vector $[P]^L(i * 3 + j)$.The first step of this algorithm computes the diagonal matrix $[D]$ and solution vector $\{Q\}$.Algorithm 3.2 explains how to compute the diagonal matrix $[D]$ and solution vector $\{Q\}$. This algorithm is common for all following algorithms which are described in this chapter. Moreover, this step is also parallelized. Steps 4 to 19 of the Algorithm 3.1 which compute the residual vector of this process also can be parallelized. A problem is in updating the vector $\{r\}$. Here more than 2 processes may access the same memory location at the same time while updating a particular r_i. This is called the race condition (Figure 3.5). To avoid the race condition we used 'atomic' [72] operations which are predefined functions. The 'arch = sm_20' [72] flag was used to enable FERMI advanced architecture features which support them [72]. Kiss *et al*. used the coloring technique [34] to avoid the race condition and a two numbering system [33] was used by Fernandez *et al.* Both methods take a lot of memory and extra computations. So we avoided these methods.

Algorithm 3.1 Element by Element Jacobi Algorithm

1: Compute diagonal vector ($\{D\}$) and solution vector($\{Q\}$) - parallel
2: **while** $\delta <$ given accuracy **do**
3: $\{r\} \leftarrow \{Q\}$
$\{n$ - number of elements$\}$
$\{\{V\}$ is the set vertices of a element$\}$
4: Compute local coefficient matrix $[P]^L$
{The global matrix [P] is not formed except for its diagonal {D}}
{s - number of nodes per element e.g, triangle =3, tetrahedron = 4}
5: **for** $i = 0$ to $n - 1$ **do**
6: **for** $i = 0$ to $s - 1$ **do**
7: **if** $V(i) < n$ **then**

Algorithm 3.1 (Cont'd)

8: $NR \leftarrow V(i)$

{NR is the row of the unformed global matrix}

9: **for** $j = 0$ to $s - 1$ **do**

10: $NC \leftarrow V(j)$

{NC is the column of the global matrix}

11: **if** $V(i) < n$ **then**

12: **if** $i \neq j$ **then**

13: $r(NR) \leftarrow r(NR) - [P]^L(i * s + j) * \varphi^m(NC)$

{φ is the solution vector}

14: **end if**

15: **end if**

16: **end for**

17: **end if**

18: **end for**

19: **end for**

20: **for** $l = 0$ to n **do**

21: $\varphi^{m+1} = \frac{Q_l - r_l}{D_l}$

22: **end for**

23: $\delta \leftarrow \|\{\varphi\}^{m+1} - \{\varphi\}^{m+}\|_\infty$

24: $\{\varphi\}^m = \{\varphi\}^{m+1}$

25: $m = m + 1$

26: **end while**

Algorithm 3.2 Computing the diagonal vector ($\{D\}$) and the right hand side vector($\{Q\}$)

1: **for** $i = 0$ to $n - 1$ **do**

2: **for** $i = 0$ to $s - 1$ **do**

3: **if** $V(i) < n$ **then**

4: $NR \leftarrow V(i)$

{NR is the row of the unformed global matrix}

5: **for** $j = 0$ to $s - 1$ **do**

Algorithm 3.2 (Cont'd)

$NC \leftarrow V(j)$

{NC is the column of the global matrix}

if $V(j) < n$ **then**

  if $i = j$ **then**

    $D(NR) \leftarrow D(NR) - [P]^L(i * s + j)$

  end if

else

  $Q(NR) \leftarrow Q(NR) - [P]^L(i * s + j) * \varphi(NC)$

end if

end for

end if

end for

end for

3.4.2 Element by Element Conjugate Gradients Algorithm

The linear system equation from the finite element formulation can be solved by a minimization method [77]. Consider a quadratic equation

$$f(\{x\}) = \frac{1}{2}\{x\}^T[A]\{x\} - \{b\}^T\{x\} + \frac{1}{2}\{b\}^T\{b\} \tag{3.7}$$

where $[A]$ is a symmetric matrix, $\{b\}$ and $\{x\}$ are vectors, and $1/2\{b\}^T\{b\}$ is a scalar constant. It is the square of the residual of the equation $[A]\{x\} = \{b\}$. $[A]$ is symmetric and positive-definite for our finite element equation, $f(\{x\})$ is minimized by the solution to $[A]\{x\} = \{b\}$ The iterates $\{x\}^{(i)}$ are updated in each iteration by a multiple $\alpha_{(i)}$ of the search direction vector $\{p\}^{(i)}$ [78]:

$$\{x\}^{(i)} = \{x\}^{(i-1)} + \alpha_i \{p\}^{(i)} \tag{3.8}$$

Corresponding residual $\{r\}^{(i)} = \{b\} - [A]\{x\}^{(i)}$ are updated as $\{r\}^{(i)} = \{r\}^{(i-1)} - \alpha_i [A]\{p\}^{(i)}$ where $\alpha_i = \{r\}^{(i-1)^T}\{r\}^{(i-1)} / \{p\}^{(i)^T}[A]\{p\}^{(i)}$ is chosen so as to minimize *f({x})* along $\{x\}^{(i)} + \alpha_i \{p\}^{(i)}$.

The search directions $\{p\}^{(i)}$ are updated using the residuals $\{r\}^{(i)}$ i.e.,

$$\{p\}^{(i)} = \{r\}^{(i-1)} + \beta_{i-1} \{p\}^{(i-1)} \tag{3.9}$$

where $\beta_{(i)}$is defined by $\{r\}^{(i)^T}\{r\}^{(i)} / \{r\}^{(i-1)}\{r\}^{(i-1)^T}$

The conjugate gradients iterative solver can also be decomposed into an element by element process [79]. Here we have used the element-by-element Jacobi preconditioner [80, 63] because traditional preconditioners cannot be used without the assembled global matrix. In this algorithm we use the Jacobi preconditioner [78] which is weak but offers preconditioning. Algorithm 3.4.8 shows the element by element process using the conjugate gradients algorithm [79]. Steps 6 to 18 of the Algorithm 3.3 compute the residual vector *(v)* – this process also can be parallelized.

Algorithm 3.3 Element by Element Conjugate Gradients Algorithm

1: Compute diagonal vector ($\{D\}$) and solution matrix($\{Q\}$) - parallel
2: $\{z\} = \{Q\}/\{D\}$
3: $\{p\} = \{z\}$
4: **while** $\delta <$ given accuracy **do**
5: $\{r\} \leftarrow \{Q\}$
$\{n$ - number of elements$\}$
{Compute local coefficient matrix $[P]^L$}

Algorithm 3.3 (Cont'd)

{The global matrix [P] is not formed except for its diagonal $\{D\}$}

{s - number of nodes per element e.g, triangle =3, tetrahedron = 4}

for $i = 0$ to n **do**

  for $i = 0$ to s **do**

    if $V(i) < n - 1$ **then**

      $NR \leftarrow V(i)$

      for $j = 0$ to s **do**

        $NC \leftarrow V(j)$

        {NC is the column of the global matrix}

        if $V(i) < n$ **then**

          $v(NR) \leftarrow v(NR) - P^L(i * 3 + j) * p(NC)$

          {NR is the row of the unformed global matrix}

        end if

      end for

    end if

  end for

end for

$\alpha = \{r\}.\{z\}/(\{p\}.\{v\})$

$\{x\} = \{x\} + \alpha \times \{p\}$

$\{r_n\} = \{r\} - \alpha \times \{v\}$

$\{z_n\} = \{r_n\}/\{D\}$

$\beta = (\{z_n\}.\{r_n\})/(\{z\}.\{r\})$

$\{z\} = \{z_n\}$

$\{r\} = \{r_n\}$

check the convergence if necessary

$\{p\} = \{z\} + \beta \times \{p\}$

end while

3.4.3 Element by Element Biconjugate Gradient Algorithm

Kiss *et al.* [34] have recently applied EbE processing to solve their finite element equations from a first order tetrahedral mesh using the bi-conjugate gradients algorithm. Algorithm 3.4 shows their element by element process using the biconjugate gradient algorithm. The CG method is not suitable for a non-symmetric system since the residual vectors cannot be made with short recurrence [81, 82]. The biconjugate gradients method takes another approach, replacing the orthogonal sequence of residuals by two mutually orthogonal sequences, at the price of no longer providing a minimization. In the biconjugate gradients method, we update the two sequences of residuals and search directions for *[A]* and $[A]^T$. Steps 4 to 19 of the Algorithm 3.4 compute the residual vector, steps 31 to 34, update the vectors *{xt}, {r}* and *{rd}* and steps 41 to 47 update the vectors *{d}, {dd}, {q}* and *{qd}.* These processes also can be parallelized. In this process the residuals may be computed using,

$$\{r\}^{(i)} = \{r\}^{(i-1)} + \alpha_i [\mathrm{A}]\{p\}^{(i)} \tag{3.10}$$

$$\{r\}'^{(i)} = \{r\}'^{(i-1)} + \alpha_i [\mathrm{A}]\{p\}'^{(i)} \tag{3.11}$$

The search directions are given by,

$$\{p\}^{(i)} = \{r\}^{(i-1)} + \beta_{i-1}\{p\}^{(i-1)} \tag{3.12}$$

$$\{p\}'^{(i)} = \{r\}'^{(i-1)} + \beta_{i-1}\{p\}'^{(i-1)} \tag{3.13}$$

and the distance to be moved along a direction by,

$$\alpha_i = \{r\}'^{(i-1)^T}\{r\}^{(i-1)} / \{p\}'^{(i-1)^T}[A]\{p\}^{(i)} \tag{3.14}$$

$$\beta_i = \{r\}'^{(i)^T}\{r\}^{(i)} / \{r\}'^{(i-1)^T}\{r\}^{(i-1)} \tag{3.15}$$

Algorithm 3.4 Element by Element Biconjugate Gradient Algorithm

Compute diagonal vector ($\{D\}$) and solution matrix($\{Q\}$) - parallel
$\{D\} \leftarrow 1/\{D\}$
$\{rd\} \leftarrow 1/\{r\}$
$\{d\} \leftarrow \{r\}.\{D\}$
$\{dd\} \leftarrow \{D\}.\{rd\}$
$\{q\} \leftarrow 0$
$\{qd\} \leftarrow 0$
for $i = 0$ to n **do**
    $del \leftarrow del + r(i) \times d(i)$
end for
while $\delta <$ given accuracy **do**
    $\{r\} \leftarrow \{Q\}$
    $\{n$ - number of elements$\}$
    {Compute local coefficient matrix $[P]^L$}
    {The global matrix [P] is not formed except for its diagonal {D}}
    { s - number of nodes per element e.g, triangle = 3, tetrahedron = 4}
    $\{NR$ is the row of the unformed global matrix$\}$
    for $i = 0$ to n **do**
        for $i = 0$ to s **do**
            if $V(i) \leq n$ **then**
                $NR \leftarrow V(i)$
                for $j = 0$ to s **do**
                    $NC \leftarrow V(j)$
                    {NC is the column of the global matrix}
                    if $V(i) \leq n$ **then**
                        $q(NR) \leftarrow q(NR) - P^L(i * 3 + j) * d(NC)$
                        $qd(NR) \leftarrow qd(NR) - P^L(i * 3 + j) * dd(NC)$

Algorithm 3.4 (Cont'd)

```
            end if
          end for
        end if
      end for
    end for
    α ← 0
    for i = 0 to n do
      α ← α + dd(i) × q(i)
    end for
    for i = 0 to n do
      xt(i) ← xt(i) + del/α × d(i)
      r(i) ← r(i) + del/α × q(i)
      rd(i) ← rd(i) + del/α × qd(i)
    end for
    del_d = del
    del = 0
    for i = 0 to n do
      del = del + r(i) × D(i) × rd(i)
    end for
    α = del/del_d
    for i = 0 to n do
      d(i) ← D(i) × r(i) × α × d(i)
      dd(i) ← D(i) × rd(i) × α × qd(i)
      q(i) = 0
      qd(i) = 0
    end for
end while
```

3.4.4 Element by Element with Bi-Conjugate Gradient Stabilized method

The BiCGSTAB iterative method had been used to solve the steady Navier-Stokes equations by Wang *et al.* [83]. Sheu *et al.* [84] used the BiCGSTAB method for solving the monotonic finite element model. The BiCGSTAB method is suitable for non-symmetric systems but can be applied to symmetric systems too [75]. This method avoids the irregular convergence pattern of the conjugate gradient squared method. In the biconjugate gradient method, the residual vector $\{r\}^{(i)}$ can be written as the product of $\{r\}^{(0)}$ and an i^{th} degree polynomial in [A];

$$\{r\}^{(i)} = P_i(A)\{r\}^{(0)} \tag{3.16}$$

This same polynomial satisfies

$$\{r\}'^{(i)} = P_i(A)^T\{r\}'^{(0)} \tag{3.17}$$

So that

$$\rho_i = (\{r\}'^{(i)}.\{r\}^{(i)}) = (\{r\}'^{(0)}P_i^2(A).\{r\}^{(0)}) \tag{3.18}$$

This result suggests that if $P_i(A)$ reduces $\{r\}^{(0)}$ to a smaller vector $\{r\}^{(i)}$, then it might be advantageous to apply this P_i operator twice, and compute $P_i^2(A).\{r\}^{(0)}$ [85]. This algorithm is known as the conjugate gradient squared method. The biconjugate gradient stabilized method (BiCGSTAB) avoids the irregular convergence of the conjugate gradient squared method. The BiCGSTAB computes $q_i(A)P_i(A)\{r\}^{(0)}$instead of $P_i^2(A).\{r\}^{(0)}$[86]. The BiCGSTAB method is computationally expensive per iteration compared to the CG algorithm [87].

Algorithm 3.5 shows the element by element process using bi-conjugate gradient

stabilized algorithm [84, 83]. BiCGSTAB has two stopping tests which are shown in Algorithm 3. 5. Steps 13 to 27 and 36 to 50 of the Algorithm 3.5 compute the residual vectors. These processes also can be parallelized.

Algorithm 3.5 Element by Element with Bi-Conjugate Gradient Stabilized method

1: Compute diagonal matrix ($\{D\}$) and solution matrix($\{Q\}$) - parallel
2: $\{rt\} = \{r\}$
3: **while** $\delta <$ given accuracy **do**
4: $\rho_1 \leftarrow \{r\}.\{rt\}$
5: $i \leftarrow i + 1$
6: **if** $i == 1$ **then**
7: $\{p\} = \{r\}$
8: **else**
9: $\beta = (\rho_1/\rho_2) * (\alpha/\omega)$
10: $\{p\} = \{r\} + \beta * (\{p\} - \omega * \{v\})$
11: **end if**
12: $\{p_h\} = \{p\}/\{D\}$
$\{n$ - number of elements$\}$
$\{V$ is the set vertices of a element$\}$
{Compute local coefficient matrix [PL]}
{The global matrix [P] is not formed except for its diagonal [D]}
{s - number of nodes per element e.g, triangle =3, tetrahedron = 4}
{NC is the column of the global matrix}
$\{NR$ is the row of the unformed global matrix$\}$
13: **for** $i = 0$ to n **do**
14: **for** $i = 0$ to s **do**
15: **if** $V(i) \leq n$ **then**
16: $NR \leftarrow V(i)$
17: **for** $j = 0$ to s **do**
18: $NC \leftarrow V(j)$

Algorithm 3.5 (Cont'd)

19: **if** $V(i) \leq n$ **then**
20: **if** $i \neq j$ **then**
21: $v(NR) \leftarrow v(NR) - P^L(i * s + j) * \varphi^m(NC)$
{φ is the solution vector}
22: **end if**
23: **end if**
24: **end for**
25: **end if**
26: **end for**
27: **end for**
28: $\alpha = \rho_1/(\{rt\}.\{v\})$
29: $\{s\} = \{r\} - \alpha \times \{v\}$
30: $res_s = \|s\|_\infty$
31: **if** $res_s \leq tolerence$ **then**
32: $\{x\} = \{x\} + \alpha \times p_h$
33: break
34: **end if**
35: $\{s_h\} = \{s\}/\{D\}$
{Compute local coefficient matrix [PL]}
{The global matrix [P] is not formed except for its diagonal [D]}
{s - number of nodes per element e.g, triangle =3, tetrahedron = 4}
{NR is the row of the unformed global matrix}
36: **for** $i = 0$ to n **do**
37: **for** $i = 0$ to s **do**
38: **if** $V(i) \leq n$ **then**
39: $NR \leftarrow V(i)$
40: **for** $j = 0$ to s **do**
41: $NC \leftarrow V(j)$
{NC is the column of the global matrix}

Algorithm 3.5 (Cont'd)

42: **if** $V(i) \leq n$ **then**
43: **if** $i \neq j$ **then**
44: $r(NR) \leftarrow r(NR) - P^L(i * s + j) * \varphi^m(NC)$
$\{\varphi$ is the solution vector$\}$
45: **end if**
46: **end if**
47: **end for**
48: **end if**
49: **end for**
50: **end for**
51: $\omega = (\{t\}.\{s\})/(\{t\}.\{t\})$
52: $\{x\} = \{x\} + \alpha \times \{p_h\} \times +\omega \times \{s_h\}$
53: $\{r\} = \{s\} - \omega \times \{s_h\}$
54: $\rho_2 = \rho_1$
55: $\delta = |r|_\infty$
56: **end while**

3.5 Conjugate Gradients Algorithm with Sparse Storage Schemes

3.5.1 Conjugate Gradient Algorithm for Matrix Solution

There are several works, which attempt to solve finite element problems using GPU(s) [66, 88, 89, 90, 91, 92] but they are not based on using the element by element technique as in this thesis. The authors of [92] tested the conjugate gradient (CG), the biconjugate gradient (BiCG), and the biconjugate gradient stabilized (BiCGSTAB) algorithms with popular preconditioning techniques; for example the algebraic multigrid, diagonal, shifted incomplete Cholesky, and shifted incomplete LU methods. The best performance was found for conjugate gradients with preconditioning [92]. The authors of [88] also implemented CG with

preconditioning algorithms. The reported speedup is between 2 and 325. The speedup varies with different applications [88]. The reported speedup for the BiCGSTAB algorithm is on average 8 to 10 times faster [91]. The authors of [89] reported that the speedup obtained with the preconditioned conjugate gradients (PCG) on a GPU, with respect to the CPU implementation of the CG algorithm, is between 8 and 10 (depending on the sparse matrix-vector multiplication used).

The PCG algorithm is considered more efficient for large matrix equations and therefore usually used for solving a symmetric, sparse, positive definite system of linear equations as from finite element analysis. It identifies the residual in successive orthogonal directions and for n equations is guaranteed to converge in no more than n iterations [93]. We use it because it is cheap. Algorithm 3.6 describes the PCG method for the system of linear equations $[P]\{\varphi\} = \{Q\}$; where $[P]$ is a real, positive definite, symmetric matrix from finite element discretization and $\{\varphi\}$ is the initial solution of the system which is improved in each iteration *k.* Preconditioning by the matrix M is used to replace the original system $[P]\{\varphi\} - \{Q\} = 0$ by $M^{-1}([P]\{\varphi\} - \{Q\}) = 0$. The Jacobi preconditioner is one of the simplest forms of preconditioning, in which the preconditioner is chosen to be the diagonal of the matrix P = {diag}(A). In the case of the implementation of this algorithm, we use CUDA C for parallel implementation and C++ for sequential implementation.

Algorithm 3.6 Preconditioned Conjugate Gradient

1: $\{r\}_0 = \{b\} - [A]\{x\}_0$

2: $\{z\}_0 = [M]^{-1}\{r\}_0$

3: $\{p\}_0 = \{z\}_0$

Algorithm 3.6 (Cont'd)

4: $k = 0$

5: **while** stopping condition **do**

6: $\alpha_0 = \frac{\{r\}_k^T \{z\}_k}{\{P\}_k^T [A] \{P\}_k}$

7: $\{x\}_{k+1} = \{x\}_k + \alpha_k \{P\}_k$

8: $\{r\}_{k+1} = \{r\}_k + \alpha_k [A] \{P\}_k$

9: $if \{r\}_{k+1}$ is sufficiently small then exit loop

10: $\{z\}_{k+1} = [M]^{-1} \{r\}_{k+1}$

11: $\beta_k = \frac{\{z\}_{k+1}^T \{r\}_{k+1}}{\{z\}_k^T \{r\}_k}$

12: $\{P\}_{k+1} = \{z\}_{k+1} + \beta_k \{P\}_k$

13: $k = k + 1$

14: **end while**

3.5.2 Matrix Storage Schemes

3.5.2.1 Introduction

In finite element analysis and optimization the coefficient matrix is very large when dealing with real world problems but very sparse [13]. For a symmetric matrix, we need to store only the diagonal and upper or lower triangular part of the matrix. The sparsity property brings storage down to *O(n)* for the finite element method [13]. The elimination of unnecessary multiplications with 0.0 also speeds up computations significantly. The following sections give an overview of matrix storage schemes.

3.5.2.2 Profile Storage

Profile storage is also known as its equivalent skyline storage which reduces the storage requirement for a matrix. The matrix would be stored in three one dimensional floating point

number arrays. Space is allocated for every number to the right of the first non-zero on a row, up to the diagonal term. Therefore renumbering is used first to reduce storage, to band the matrix.

$$\begin{bmatrix} 10 & -3 & 0 & 1 & 0 & 0 \\ -3 & 9 & 6 & 0 & -2 & 0 \\ 0 & 6 & 8 & 7 & 0 & 0 \\ 1 & 0 & 7 & 7 & 5 & 4 \\ 0 & -2 & 0 & 5 & 9 & 13 \\ 0 & 0 & 0 & 4 & 13 & -1 \end{bmatrix} \quad \begin{bmatrix} 10 & & & & & \\ -3 & 9 & & & & \\ 0 & 6 & 8 & & & \\ 1 & 0 & 7 & 7 & & \\ 0 & -2 & 0 & 5 & 9 & \\ 0 & 0 & 0 & 4 & 13 & -1 \end{bmatrix}$$

A B

Figure 3.6: A. Sparse full matrix, B. Sparse lower triangular matrix (because of symmetry)

FC: [1, 1, 2, 1, 2, 4,]

Diag: [1, 3, 5, 9, 13, 16]

V: [10, -3, 9, 6, 8, 1, 0, 7, 7, -2, 0, 5, 9, 4, 13, -1]

Figure 3.7: Data structures for the symmetric profile storage corresponding to Figure 3.6 B

The matrix of Figure 3.6 A is reduced first to its lower triangle part in Figure 3.6 B. It is then stored as the vectors *Diag*, giving the diagonal element location, *FC* giving the first column on a row occupied by a non-zero and *V*, giving the coefficient of the matrix which now has several zeros which are between the first non-zero column and the diagonal. An example matrix and profile storage scheme vectors are shown in Figures 3.6 and 3.7 respectively.

The data structure consists of three one-dimensional arrays: A real type array *V*; the size of this array is equal to the number of non-zero elements plus the number of zeros between two non-zero elements in the array, an integer type array *FC*; the size of this array is equal to the number of non-zero elements in the array, an integer type array *Diag*; the size of this array is n ; where n is the number of rows/columns in the array. When we are dealing with real world

problems, the coefficient matrix is a very large sparse matrix [13] and we can use the profile storage scheme to reduce memory consumption

$$\begin{bmatrix} 1 & -1 & 0 & -3 & 0 \\ -1 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -3 & 0 & 6 & 7 & 0 \\ 0 & 0 & 4 & 0 & -5 \end{bmatrix} \qquad \begin{bmatrix} 1 & -1 & 0 & -3 & 0 \\ & 5 & 0 & 0 & 0 \\ & & 4 & 6 & 4 \\ & & & 7 & 0 \\ & & & & -5 \end{bmatrix}$$

A B

Figure 3.8: A. Sparse full matrix, B. Sparse upper triangular matrix (because of symmetry)

A: [1, -1, -3, 5, 4, 6, 4, 7, -5]
JA: [1, 2, 4, 2, 3, 4, 5, 4, 5]
IA: [1, 4, 5, 8, 9, 10]

Figure 3.9: Data structures for the symmetric profile storage corresponding to Figure 3.8 B

3.5.2.3 Sparse Storage Scheme

The sparse storage scheme is also known as the compressed sparse row (CSR) scheme. The sparse storage scheme is a row-wise (or alternatively column-wise) representation of the nonzero entries in the coefficient matrix of the linear system. For a symmetric matrix, computer memory can be saved by storing only the nonzero entries in each row on and before the main diagonal. The associated column numbers are stored in an integer-valued array JA such that JA(K) is the column number for the coefficient A(K). A mapping vector IA is used to denote the starting location of each row. An example matrix and its sparse storage scheme vectors are shown in Figures 3.8 and 3.9 respectively. The data structure consists of three one-dimensional arrays: A real type array A, contains all the non-zero elements of a matrix. The size of this array

is equal to the number of non-zero elements in the array, an integer type array JA, contains the matrix column indices of the elements of A. The size of this array is equal to the number of non-zero elements in the array. Another integer type array IA contains the index of each row in the arrays A and JA. The size of this array is n +1; where n is the number of rows/columns in the array. When we are dealing with real world problems, the coefficient matrix is a very large sparse matrix [13] and we can use CSR storage scheme to reduce the memory consumption and speedup the computations.

Chapter 4

Test and Validation Problems

4.1 Device design inverse-optimization problem: Design of the Pole Face of an Electrical Motor

4.1.1 Problem Definition

Our mesh generators and solvers will be demonstrated on two examples from design. First let us consider the following sample problem. The objective is to achieve a uniform flux density distribution in the vertical direction in the air gap of a pole face (see Figure 4.1). Since the air gap in turbo-alternators compared to the radius of the machine is small, the shape of the pole face can be approximated by a straight line.

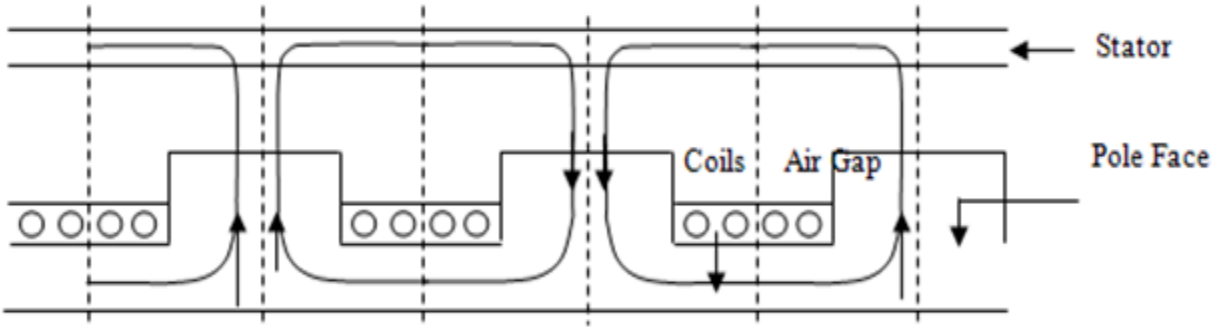

Figure 4.1: Pole face of electrical motor

Figure 4.2 gives the related dimensions, material properties and field excitation values used. The symmetry of the magnetic fields with respect to the pole axis allows the modeling of just half the pole pitch, where the pole axis, which is the line of symmetry, is located at the right boundary to the finite element solution domain. The relative permeability of 20 for the magnetic

circuit is deliberately set this low, so that the leakage flux through air at the left edge of the pole face is larger than for higher and more realistic permeability. Our requirement is to have a uniform vertical flux density of 1 Tesla along measuring points on the stator. That means all our measuring points must have their y direction flux density (x direction derivative of the vector potential) of 1 Tesla. The influence of this leakage flux requires significant correction in the shape of the pole face close to the left edge in order to achieve the desired constant flux density in the air gap. This example is frequently used in the demonstration of electromagnetic optimization methods and it can be considered as a standard demonstration example [13]

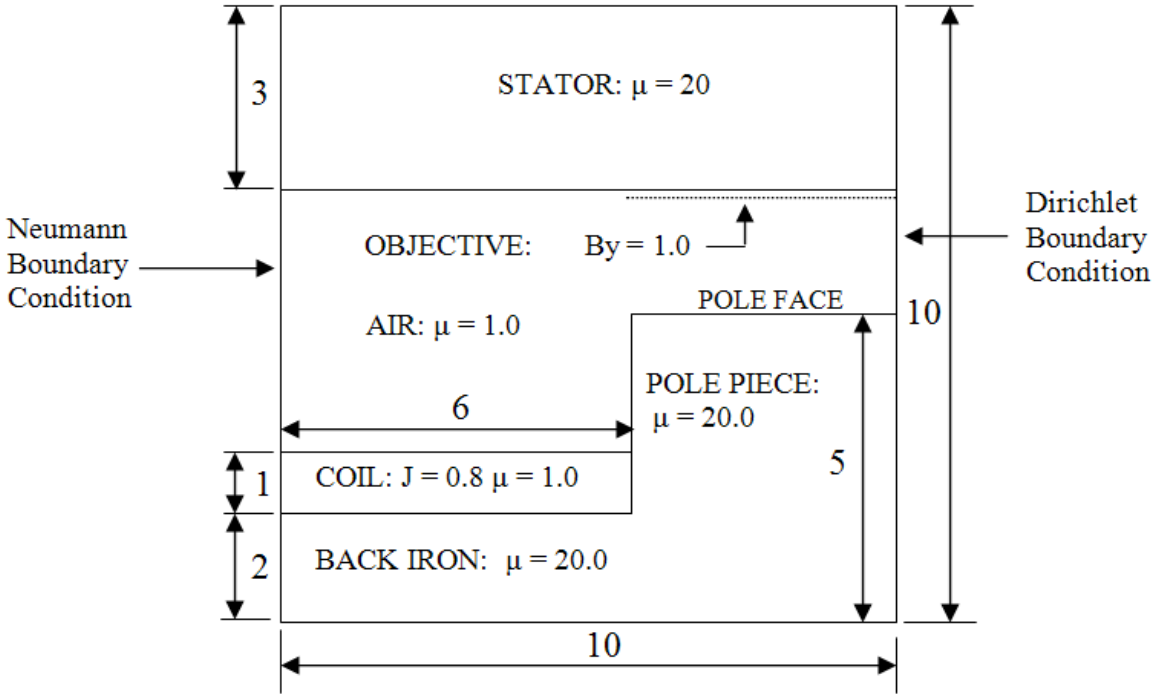

Figure 4.2: Geometry, boundary conditions and the material properties of the sample problem

4.1.2 Problem Model

Figure 4.3 shows how to model the problem using our software. There are 11 fixed points (shown in Figure 4.3), 10 variable heights (h1....h10) to be optimized, 8 measuring points (purple-dots), and 4 materials (stator, air, coil, back ion) in this problem. Figure 4.4 shows the generated mesh. Figure 4.5 shows the equipotential lines of the starting design of the pole face of a motor. Since this is a 2D magneto-static problem, these lines represent the flux lines as well.

As we discussed in Chapters 1 and 3, we use the genetic algorithm for optimization. In genetic algorithm optimization [61, 62], several copies of the matrix are held on the GPU and the corresponding solutions attempted. We have tried different problems using the GA on a GPU [12, 3].

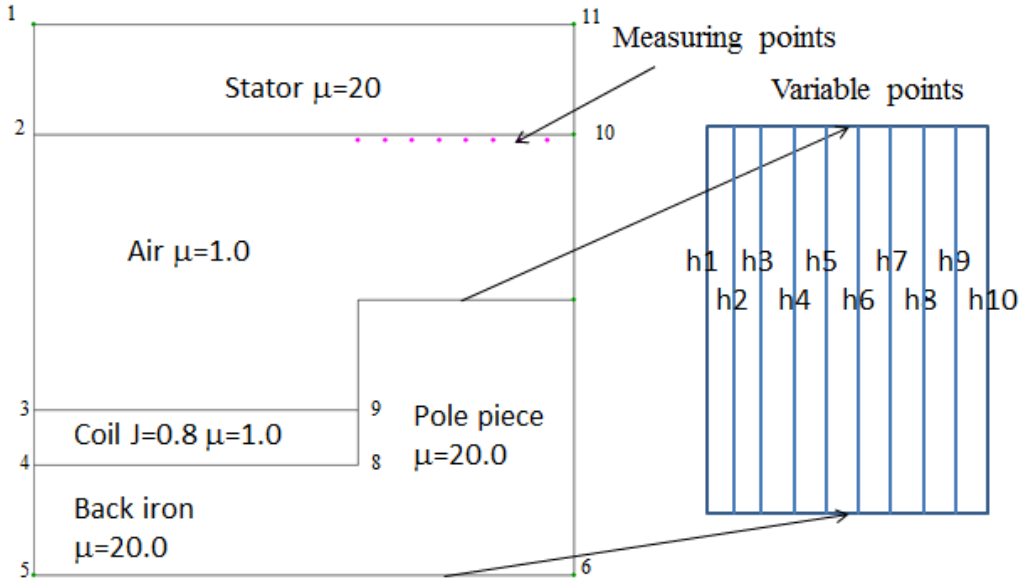

Figure 4.3: Defining the problem using our tool

When we optimize directly this problem by defining an independent parameter for the displacement of each point in the pole face, the shape we get is given in Figure 4.6. This shape is not a manufacturable shape but the solution really gives a very good result (see Table 4.1 and

Figure 4.9). However it is clear that this type of pole face cannot be practically constructed.

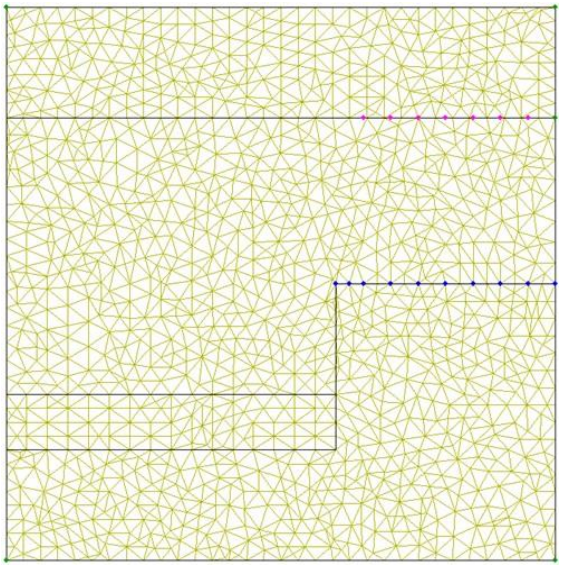

Figure 4.4: Generated mesh using our tool

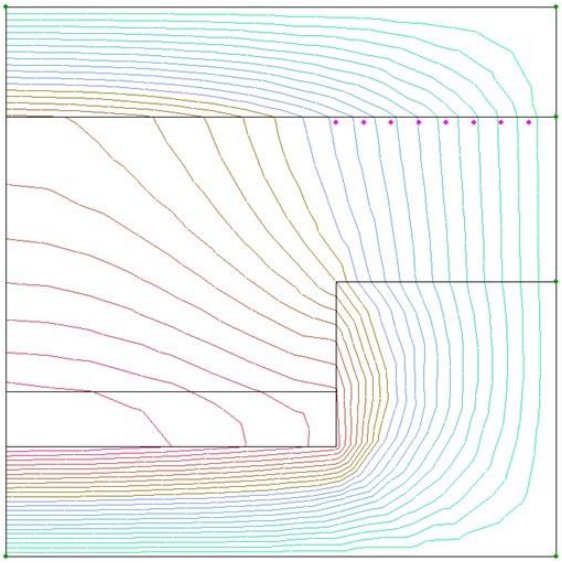

Figure 4.5: Finite element solution

There are two solutions for this problem. We can add some constraint to the solution on force the variable points to be arranged in a curve function of known mathematical form. Both methods had been tested by Wijesinghe [22]. And he claimed that even though the second

method is easy because of the known equation, the result is not as good as the result by the first method.

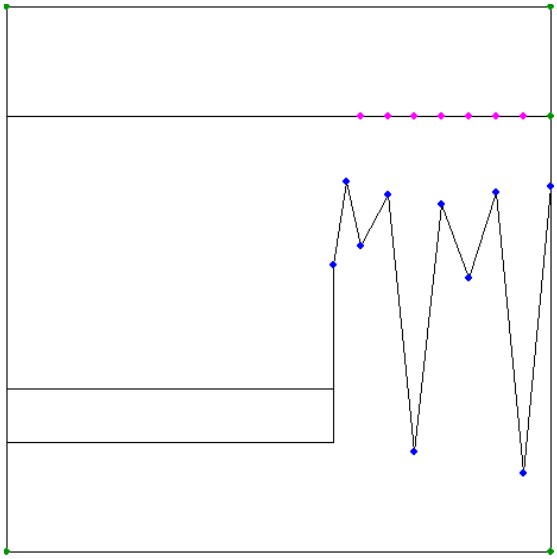

Figure 4.6: Results of the un-constrained optimization of the problem

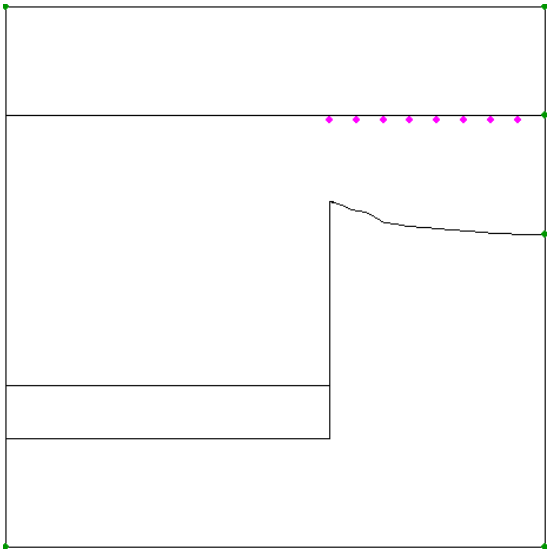

Figure 4.7: Results of the constrained optimization without smoothening

An erratic undulating shape with sharp edges arose when Pironneau [18] optimized a pole face to achieve a constant magnetic flux density and this was overcome through constraints [19].

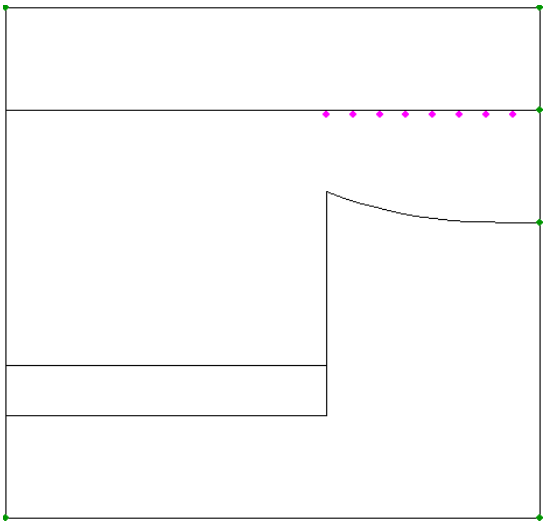

Figure 4.8: Results of the constrained optimization with smoothening

Table 4.1: The flux distribution from the un-constrained optimization

Measuring points	Flux density (in Tesla)
1	1.001
2	1.001
3	1.002
4	0.996
5	1.001
6	1.001
7	0.999
8	0.999

Haslinger and Neittaanmaki [94] suggest Bezier curves to keep the shapes smooth with just a few variables to be optimized, while Preis *et al*. [95] have suggested fourth order polynomials which when we tried gave us smooth but undulating shapes. As such we follow Subramaniam *et al*. [19] and extend their principle, so as to maintain a non-undulating shape by imposing the constraints:

$$h1 \geq h2 \geq h3 \geq h4 \geq h5 \geq h6 \ldots \quad (4.1)$$

to ensure a smooth shape Even this gives a non-smooth shape (Figure 4.7) but we use averaging of neighboring heights which is shown in Figure 4.10, to obtain a very manufacturable shape as demonstrated in Figure 4.8. The final results are given in Table 4.2 and Figure 4.11. Average error percentage of un-constrained optimization is 0.15% while the constrained optimization method has an error percentage of 1.0625%. Even-though un-constrained optimization gives a more accurate solution than constrained optimization, the resulting shape from the un-constrained problem is not practicably manufacturable.

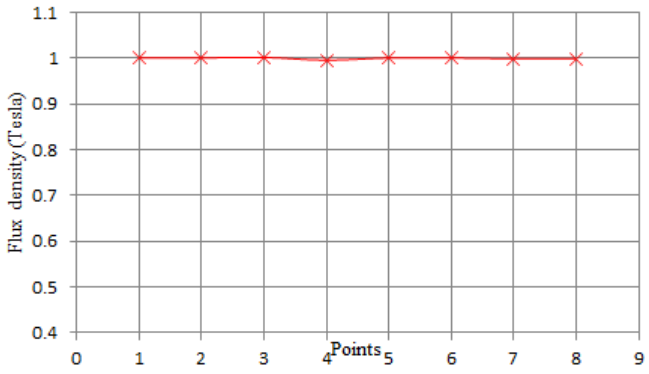

Figure 4.9: The flux distribution from the un-constrained optimization

The average error percentage was calculated using the following formula:

$$E_p = \frac{1}{2}\sum_{i=1}^{n}\frac{|B_{cal} - B_{tar}|}{B_{tar}} \times 100\% \qquad (4.2)$$

where n is the number of measuring points, *Ep* is the error percentage, $B_{cal}$is the calculated flux density and B_{tar} is the target flux density. We have reported solutions for a population size of 100. Figure 4.10 shows the averaging techniques which have been used to get a smooth manufacturable shape. This figure shows a 3-neighbor averaging technique. We have a mask that covers 3 elements. The mask moves the left most elements to the right most element and updates

the middle element with the average of the three values which are covered by the mask. We introduced 2 temporary elements at the front and end to calculate the average of the first and last elements. If we wish to use a 5-neighbor technique, we need to introduce 2 elements at the front and 2 elements at the end to calculate the average of the first and last elements.

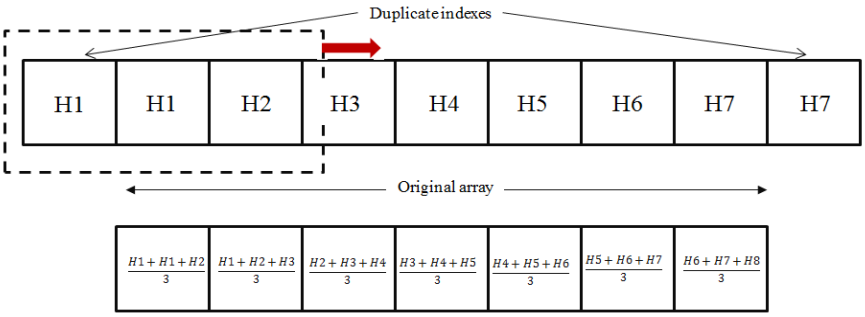

Figure 4.10: Averaging technique for manufacturable shape

Table 4.2: The flux distribution from the constrained optimization

Measuring points	Flux density (in Tesla)
1	0.991
2	1.016
3	1.008
4	1.008
5	0.977
6	0.978
7	0.993
8	0.988

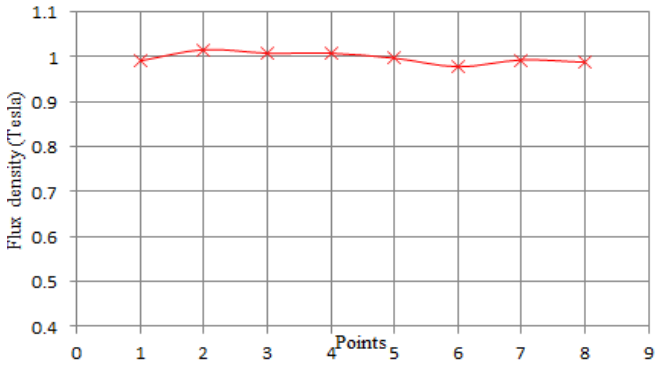

Figure 4.11: The flux distribution from the constrained optimization

4.2 Inverse-optimization for Device Design: Determining the Rotor Contour of a Salient Pole Synchronous Generator

4.2.1 Problem Definition

The second example is about determining the pole face contour of a salient pole synchronous generator to demonstrate the parametrized mesh generator and matrix solution software as applied to constrained optimization. The current density in the excitation coil and the geometric parameters that define the shape of the pole piece have to be predicted in order to achieve a sinusoidal distribution of the airgap flux with a peak value of 1.0 T and reduce the flux leakage while the airgap is constrained to a minimum to prevent the motor from hitting the stator (Figure 4.12).

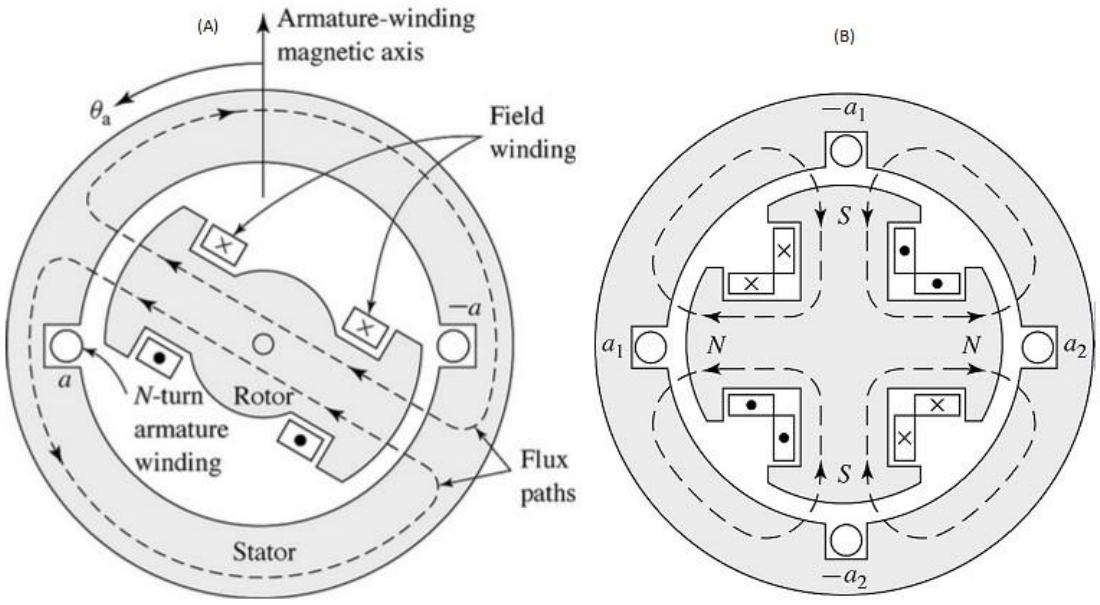

Figure 4.12: A synchronous Generator (A) two pole and (B) four pole

Figure 4.13 gives the related dimensions, material properties and field excitation values used. The symmetry of the magnetic fields with respect to the pole axis allows the modeling of just half the pole pitch, where the pole axis, which is the line of symmetry, is located at the right

boundary to the finite element solution domain. The stator is idealized as a solid steel region without slots, and both stator and rotor are made of linear steel with a relative permeability of 2000. We will optimize the device with constraints of current density $J \leq 2.0$ A/mm^2 which is the limit for copper windings, air gap between stator and rotor $x < 2$ cm and flux go through the points A_1 and $A_2 < 0.3 \times$ flux go through the points A_3 and A_1 which means allowable leakage flux is 30%,

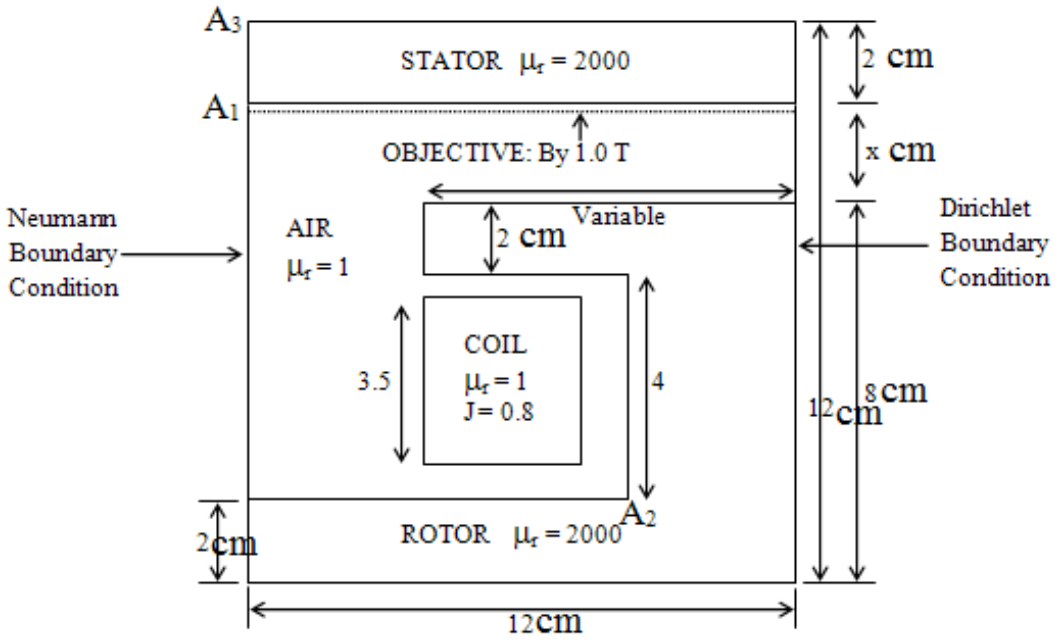

Figure 4.13: Parametrized geometry of salient pole

4.2.2 Problem Model

Figure 4.13 explains the parameters of the problem and how to model this problem. There are 14 fixed points, 16 variable heights (h1....h16), 8 measuring points and 4 materials in this problem (see Figure 4.14). Figure 4.15 shows the corresponding starting mesh for this problem. Figure 4.16 shows the flux lines of this salient pole synchronous Generator at starting. When we

optimize directly this problem by defining an independent parameter for the displacement of each point in the rotor like in the previous example, the shape we get is given in Figure 4.17. This shape is also not a manufactorable shape but the solution really gives very good result in-terms of a sinusoidal distribution of the airgap flux with a peak value of 1 T (Table 4.3 and Figure 4.17).

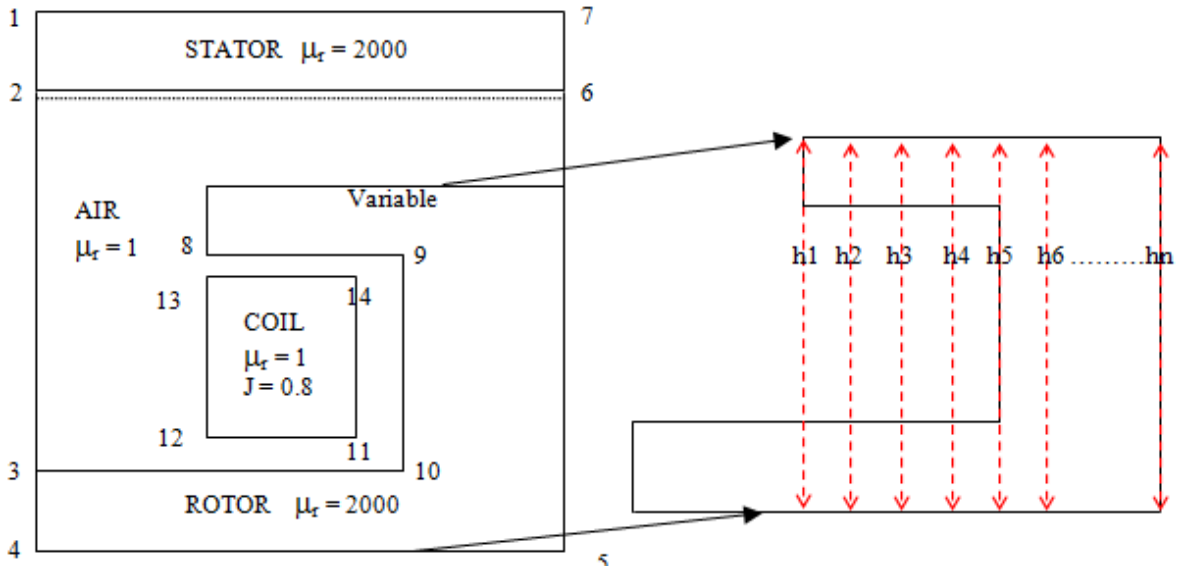

Figure 4.14: Defining the problem

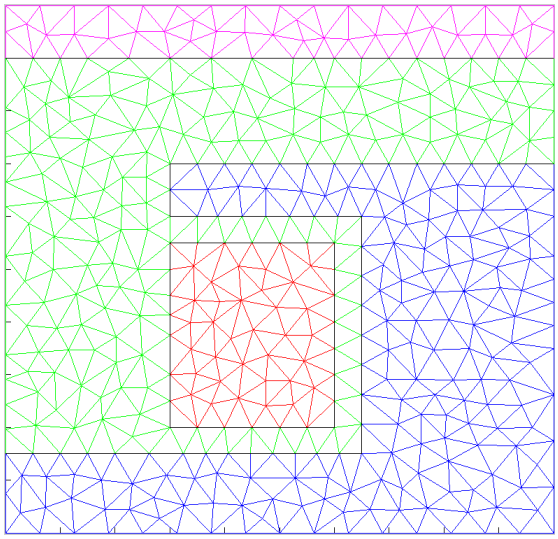

Figure 4.15: Initial mesh

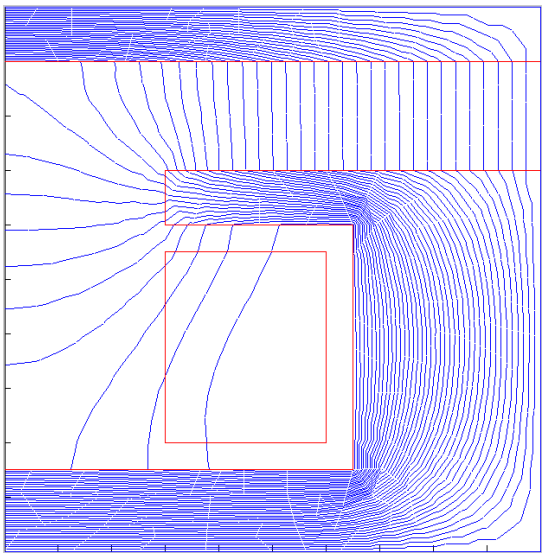

Figure 4.16: Flux line of a salient pole synchronous Generator

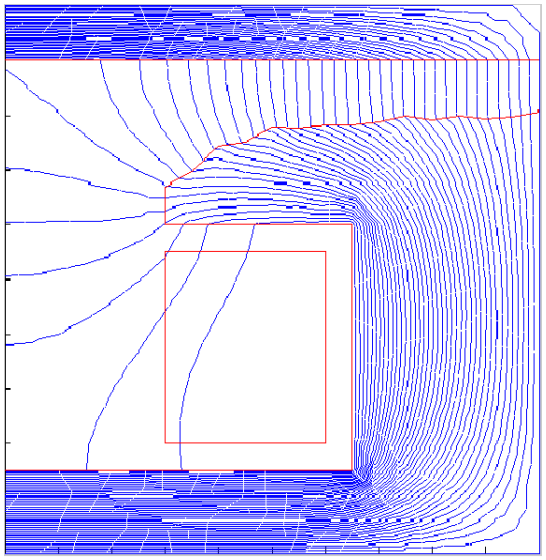

Figure 4.17: Optimized shape without smoothening constrained by rising pole heights from left to right

Figure 4.17 shows the flux lines for the optimum solution with a constraint like in (4.1) but the height of the shaped surface having to go up from left to right. It has sharp corners but is reasonably smooth. We then use an averaging technique to remove sharp bends. We took five neighboring values of a height and calculated the mean for every variable solution with suitable

modification for boundary variables to get Figure 4.20.

Table 4.3: The flux distribution after the optimization without smoothened shape

Measuring points	Flux density (in Tesla)	Target Flux density (in Tesla)
1	0.0202	0.0000
2	0.1869	0.1736
3	0.3721	0.3420
4	0.4987	0.5000
5	0.6321	0.6428
6	0.7453	0.7660
7	0.8769	0.8660
8	0.9215	0.9397
9	0.9709	0.9848
10	1.0091	1.0000

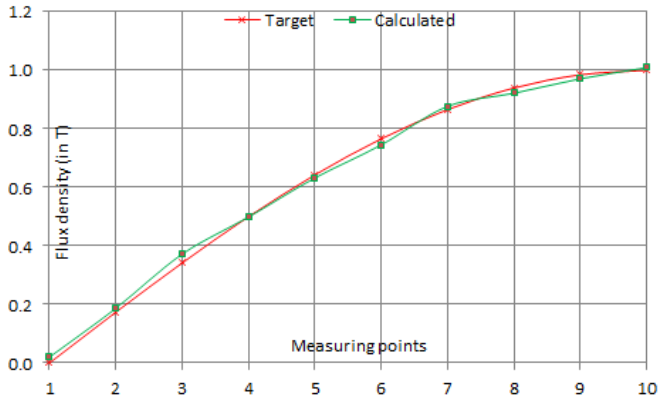

Figure 4.18: The flux distribution after the optimization without smoothened shape

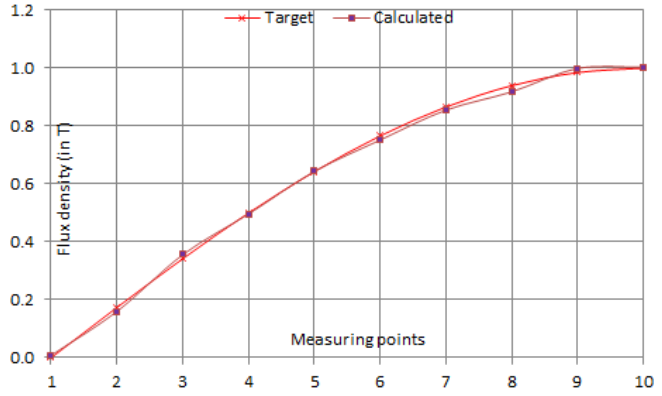

Figure 4.19: The flux distribution after the optimization with smoothened shape

The final results are given in Table 4.4 and Figure 4.19. The average error percentage of un-smoothened optimization is 2.96% while the smoothened optimization method has an error percentage of 2.24%. The average error percentage is calculated using Equation 4.2 (the first point is not included because of the zero denominators in Equation 4.2). Since GA is a stochastic optimization algorithm [96, 97], the optimization value is not always perfect. The optimum value depends on the initial population and search space [96, 97].

Table 4.4: The flux distribution after optimization with smoothened shape

Measuring points	Flux density (in Tesla)	Target Flux density (in Tesla)
1	0.0083	0.0000
2	0.1596	0.1736
3	0.3560	0.3420
4	0.4964	0.5000
5	0.6434	0.6428
6	0.7517	0.7660
7	0.8549	0.8660
8	0.9183	0.9397
9	0.9986	0.9848
10	1.0035	1.0000

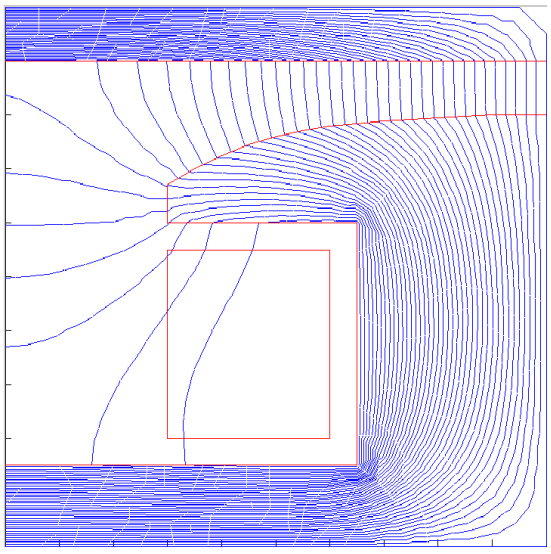

Figure 4.20: Final smoothened shape

4.3 NDE benchmark problem: Characterizing Interior Defects

4.3.1 Problem Definition

As an example, when an armored vehicle is targeted by an improvised explosive device (Figure 4.21), the armor is inspected by an eddy current probe to determine whether there is damage or not. But we wish to characterize the interior damage to determine if the vehicle should be withdrawn from deployment. The same system is also intended for regular rust mitigation maintenance since the US army's estimated loss from corrosion is in the billions of dollars [98, 99, 100]

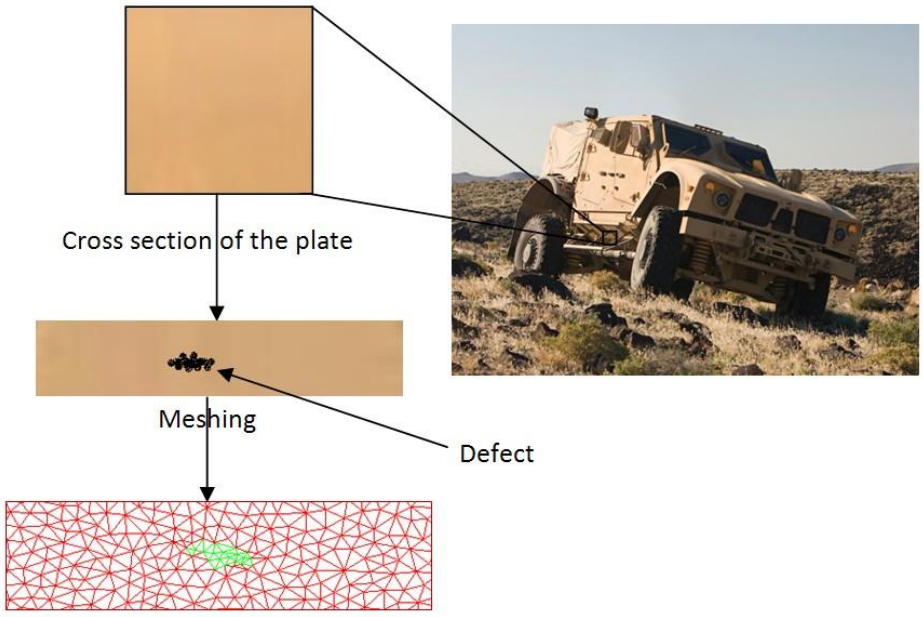

Figure 4.21: Inspection of an army vehicle after improvised explosive device

Figure 4.22 presents crack shapes, both shown through the meshes, to make the computed field match the measured field. The normalized least-square mismatch of nodes from the midpoint between the measured and computed shapes (Figure 2.4) shows a parametrically described crack in steel excited by an eddy current probe. In this NDE exercise the parameters

need to be optimized to make the computed fields match the measurements. An objective function F is defined as the sum of the squares of the difference between computed and desired performance values: at measurement points i,

$$F = \sum_{i} \left(B_{cal}^{i} - B_{mea}^{i} \right) \tag{4.3}$$

where B_{cal}^{i} is the calculated magnetic flux density and B_{mea}^{i} is the measured flux density. By minimizing the objective function F by the optimization method, the characteristics of the defect can be estimated since F is the function of the parameters.

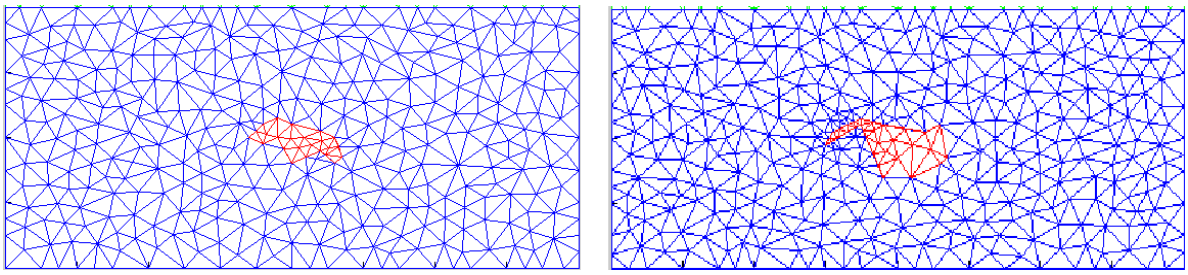

Figure 4.22: Parametrically defined crack in plate from Triangle

4.3.2 Problem Model

Figure 4.23 shows the parameters of the described NDE problem and how to model it to using our tools. There are 14 fixed points, and 6 variable points (for example); the x coordinates are fixed [101]. There are also 4 materials (air, crack, steel plate and coil) in this problem. Figure 4.24 shows the generated mesh for this problem. Different materials are shown in different colors (although in black and white in this printout). Figure 4.25 shows the flux lines from this example. The flux lines are shown in Figure 4.25. Figure 4.26 shows the true defect and constructed defect.

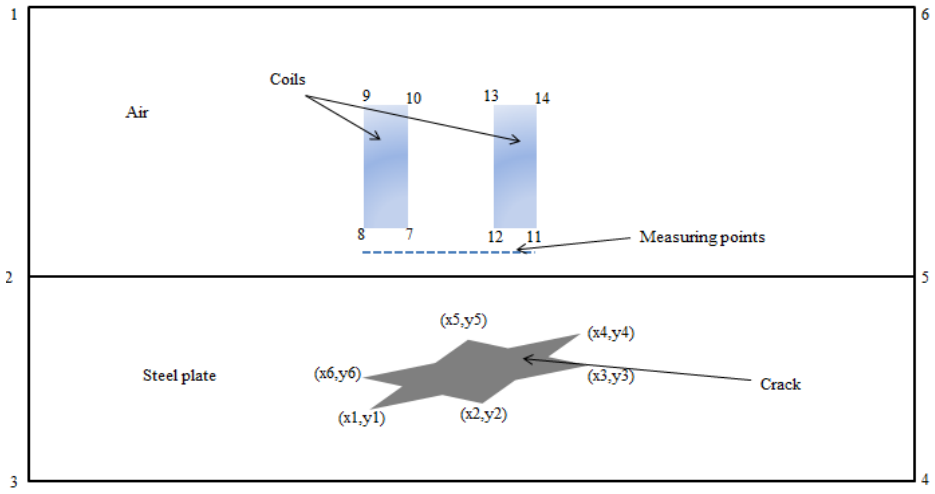

Figure 4.23: Defining the problem

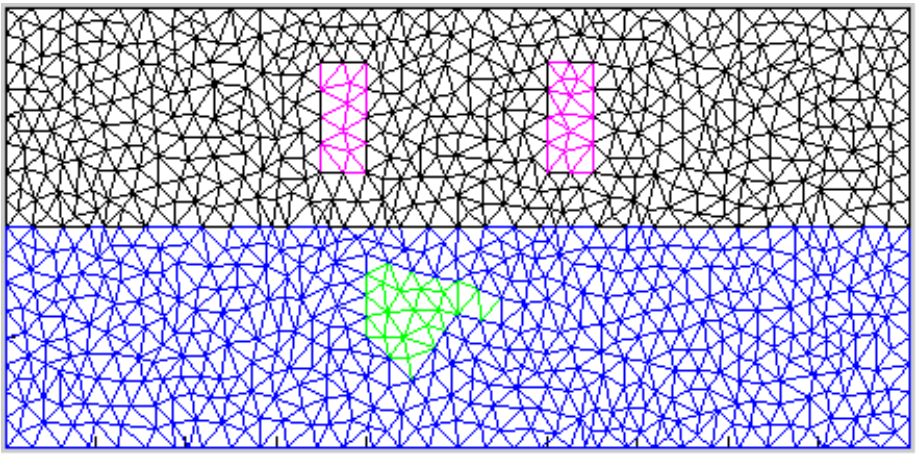

Figure 4.24: Generated Mesh for NDE problem

Table 4.5 shows the variable points (defect coordinates), their x and y coordinates, euclidean distance between the centroid of that the crack and a point (d1 and d2 respectively) and normalized distance between d1 and d2. That means centroid difference between the true profile and the constructed profile, d1 and d2, is calculated using the Equation 4.4. Let us say, (x1, y1) and (x2, y2) are the two points; the euclidean distance between the two points is defined by the following formula:

$$d1 = \sqrt{(x1 - x2)^2 + (y1 - y2)^2} \tag{4.4}$$

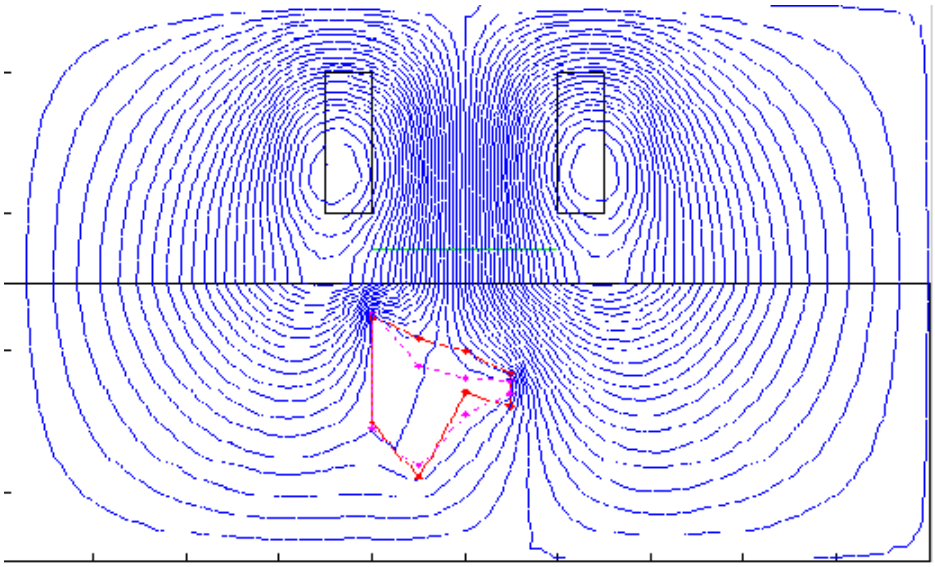

Figure 4.25: Flux line for NDE problem

Table 4.5: The solution of defect characterization

	True profile			Reconstructed profile			Norm error
Variable points	x	y	d1	x	y	d2	$((d1-d2)/d1)^2$
1	8	2.0	1.5892	8	2.24	1.5268	0.001541
2	9	1.2	1.4162	9	2.34	0.5331	0.388815
3	10	2.4	0.5153	10	1.96	0.7544	0.215191
4	11	2.2	1.5348	11	2.15	1.5461	0.000054
5	11	2.7	1.5101	11	2.54	1.5000	0.000044
6	10	3.0	0.6896	10	2.94	0.6497	0.003342
7	9	3.2	0.8400	9	3.42	1.0251	0.048597
8	8	3.5	1.7890	8	3.55	1.8167	0.000240

Figure 4.26: Optimum shape of the reconstructed defect

The normalized least-square match of nodes from the midpoint between the measured and computed shapes (Figure 4.26) was close to 90%. The error in location was 4.65%. A better match would require the use of more parameters. We tried with several population sizes and multiple times of iterations. These results are reported in [102]. This thesis presents the particular solution for a population of 200. Tests were carried out for the different population sizes and different number of iterations and we measured the time taken to compute the solution and tabulated these in Table 4.6. The best fitness score for different population sizes and different number of iterations is reported in Table 4.7 for real and binary representations of GA solutions [102]. We can see that solutions from parameters represented by real numbers are obtained faster than the solutions that are represented by binary numbers [102].

Table 4.6: Real and binary solutions time need to compute

Population Size	30 iterations		40 iterations		50 iterations	
	Time Taken(s)		Time Taken(s)		Time Taken(s)	
	Real	Binary	Real	Binary	Real	Binary
10	274.33	311.97	346.58	408.29	413.16	503.43
20	540.68	661.79	695.28	891.06	866.47	1105.42
30	861.79	1024.65	1095.82	1258.17	1588.32	1749.1
40	1319.36	1431.34	1666.79	1950.36	2024.03	2390.25
50	1583.73	1769.6	1994.77	2302.99	2323.93	2891.94
60	1757.78	2131.94	2199.26	2534.36	2656.53	3303.85

Table 4.7: Real and binary solutions time need to compute

Population Size	30 iterations		40 iterations		50 iterations	
	Best Fitness Score		Best Fitness Score		Best Fitness Score	
	Real	Binary	Real	Binary	Real	Binary
10	0.0123	0.0019	0.0091	0.0020	0.0091	0.0015
20	0.0084	0.0012	0.0084	0.0008	0.0084	0.0008
30	0.0040	0.0009	0.0040	0.0009	0.0040	0.0009
40	0.0105	0.0017	0.0105	0.0017	0.0105	0.0017
50	0.0111	0.0017	0.0068	0.0017	0.0068	0.0017
60	0.0127	0.0017	0.0127	0.0016	0.0127	0.0000

4.4 A Simple Three-dimensional Problem

For testing purposes, we took a small cube which is made of a material with relative permittivity 1. Another inner cube with a relative permittivity of 1 and charge density of 1C/m3 is inside the outer cube (shown in Figure 4.27). The outer surface's potential is zero (boundary condition). The measuring points (along a line) are shown in Figure 4.27. Then we solved the Poisson equation to calculate the potential (ϕ) using FEM:

$$\varepsilon \nabla^2 \phi = -\rho \tag{4.5}$$

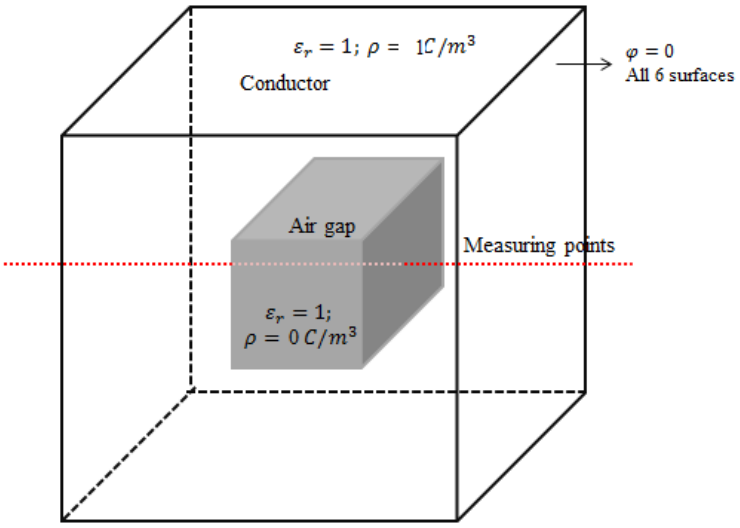

Figure 4.27: Square conductor problem

There are 16 fixed points, 2 materials (air and material) and 100 measuring points (along the line) in this problem. Since this is not an optimization problem, there are no variable points. Figure 4.28 shows the generated mesh for this problem. Different materials are given in different colors; light green for material 1 and red for material 2.

Figure 4.29 shows the potential at the measuring points. We may recognize that the potential within the inner cube is constant because the inner cube has a constant uniform charge.

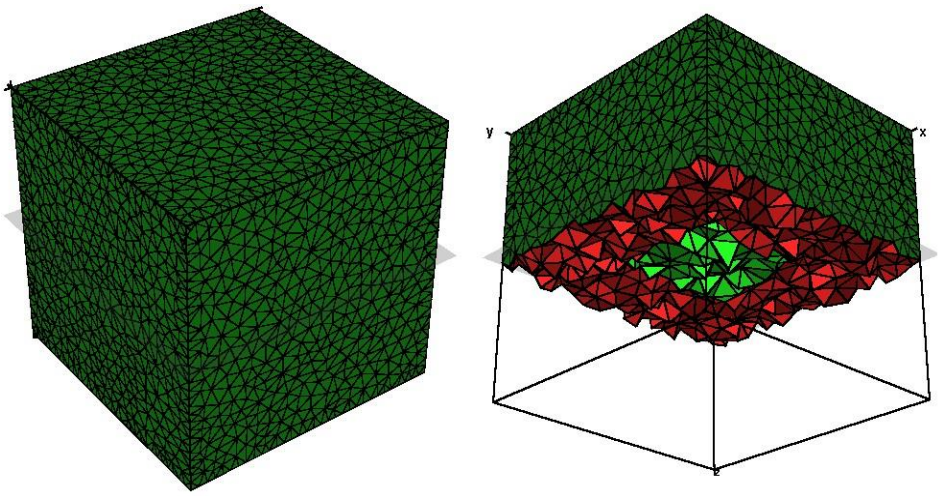

Figure 4.28: Mesh for square conductor problem

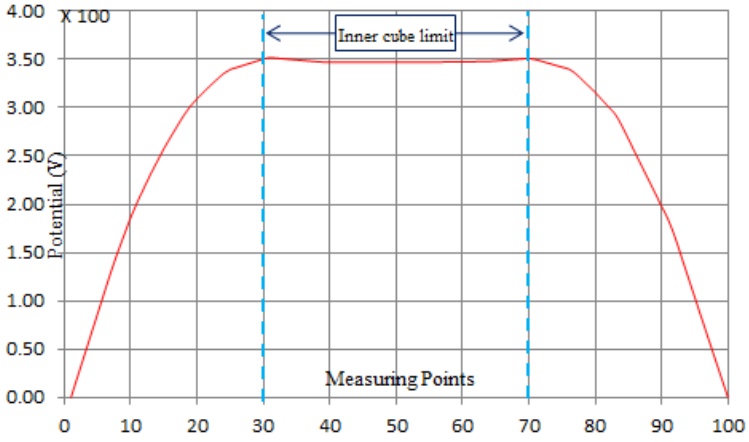

Figure 4.29: Potential at measuring points

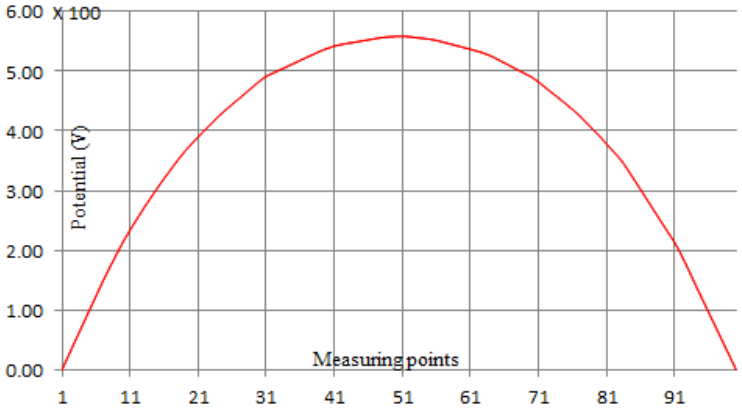

Figure 4.30: Potential at measuring points for a simple cube

Table 4.8 shows the numerical solution for this problem. The first column gives the measuring point number; the second and third columns give the electric potential value and tetrahedron number in which the particular measuring point exists, respectively. Figure 4.30 shows the potential at the measuring points for a simple cube problem (a simple cube with the same dimension as above, charge density is $1C/m^3$- no inner outer cubes and no change in permittivity).

Table 4.8: Potentials at measuring point

Measuring points	ϕ(in V)	Element#	Measuring points	ϕ(in V)	Element#
1	0.0000	19962	51	3.4708	10058
2	0.2159	13483	52	3.4711	12276
3	0.4318	13483	53	3.4714	12276
4	0.6477	13483	54	3.4717	12276
5	0.8636	13483	55	3.4721	12276
6	1.0795	13483	56	3.4727	7335
7	1.2954	13483	57	3.4736	7335
8	1.4938	982	58	3.4745	7335
9	1.6796	982	59	3.4755	7335
10	1.8616	12016	60	3.4763	1321
11	2.0228	18763	61	3.4767	1321
12	2.1673	18763	62	3.4772	18264
13	2.3118	18763	63	3.4777	18264
14	2.4446	5702	64	3.4793	15140
15	2.5760	5702	65	3.4844	15136
16	2.6948	21367	66	3.4897	15136
17	2.8076	21367	67	3.4950	15136
18	2.9204	21367	68	3.5003	15136
19	3.0161	21366	69	3.5056	15136
20	3.0922	18694	70	3.5109	15136
21	3.1629	15809	71	3.4988	15448
22	3.2293	5610	72	3.4792	15448
23	3.2924	5610	73	3.4596	15448
24	3.3532	14667	74	3.4400	15448
25	3.3950	25126	75	3.4205	15448
26	3.4169	25126	76	3.4009	15448
27	3.4388	25126	77	3.3562	15446
28	3.4607	25126	78	3.2909	15446
29	3.4826	25126	79	3.2257	15446
30	3.5045	25126	80	3.1591	8028

Table 4.8: Potentials at measuring points (Cont'd)

Measuring points	ϕ(in V)	Element#	Measuring points	ϕ(in V)	Element#
31	3.5181	23548	81	3.0841	19985
32	3.5122	23548	82	3.0092	19985
33	3.5064	23548	83	2.9343	19985
34	3.5005	23548	84	2.8152	18420
35	3.4947	23548	85	2.6758	18420
36	3.4888	23548	86	2.5364	18420
37	3.4830	23548	87	2.3971	18422
38	3.4772	23548	88	2.2578	18422
39	3.4723	22059	89	2.1185	18422
40	3.4719	22059	90	1.9794	8298
41	3.4712	19315	91	1.8409	8298
42	3.4708	19049	92	1.6655	17398
43	3.4707	19049	93	1.4562	17398
44	3.4708	4482	94	1.2470	17398
45	3.4707	4482	95	1.0383	14912
46	3.4705	4482	96	0.8303	14912
47	3.4703	4482	97	0.6222	14912
48	3.4701	4482	98	0.4142	14912
49	3.4704	17128	99	0.2072	13685
50	3.4706	5560	100	0.0001	13685

This is a simple problem. We will test the mesh generator on the more complex system of the next section. These results are verified with the mesh generator Gmsh [29] with Matlab functions. This simple problem verifies that our parameterized mesh generator and FEM solver work perfectly.

4.5 NDE benchmark problem in 3D: Characterizing Interior Defects

As a more complex example for testing our parameterized mesh generator, this is a three dimensional version of example 3 (Section 4.3). We wish characterize the interior damage to a land vehicle hull to determine if the vehicle should be withdrawn from deployment. Figure 4.31 shows the initial shape of the crack (in the lower half of the Figure 4.31), the 'E'shaped coil frame, air (upper half of the Figure 4.31) and steel plate (lower half of the Figure 4.31).

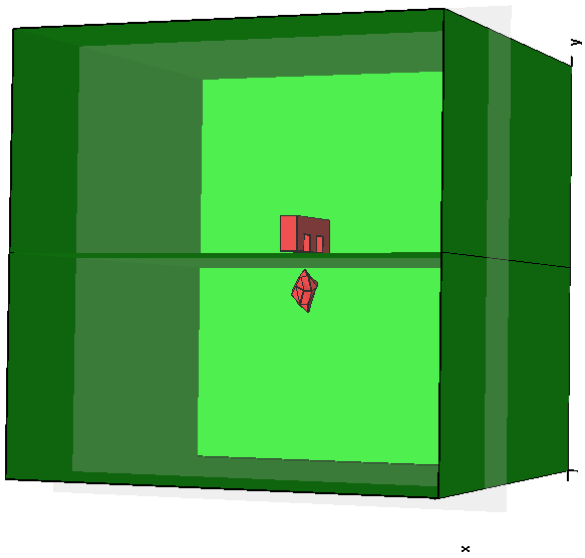

Figure 4.31: Three-dimensional NDE problem

Figure 4.32 shows how to define a crack whose outline coordinates and location coordinates are changing as the optimization algorithm runs. We define two surfaces; the outer coordinates of both surfaces are the same. We made some constraints on the y coordinates to ensure that this is truly a volume without the surfaces crossing each other. In Figure 4.32 top, there are two surfaces. Each has a few variables shown in green, while the outer common coordinates are given in blue. In Figure 4.32 the lower part of the figure shows the top view of the top surface which has 13 variables in addition to its outer common coordinates. Similarly the bottom surface also has variable points.

Figure 4.33 shows the generated mesh for this problem. Different materials are shown in different colors; green - air, red - steel, blue - coil and yellow - crack (made by 2 surfaces) although this printout is black and white. For this test problem we took 5 variable points on the upper surface, 8 variable points on the common interface and 4 variable points on the lower surface. There are a total of 17 × 3 variables associated with the 17 variable points. Here we made another assumption that the points are varying in the y direction only; the x and z

coordinates are fixed. Table 4.9 shows the true and characterized profile coordinates and the normalized distance between results.

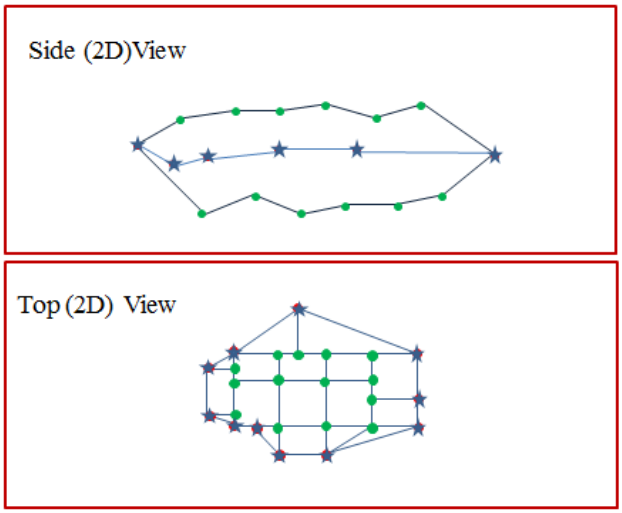

Figure 4.32: Defining variable: top: side view, bottom: side view

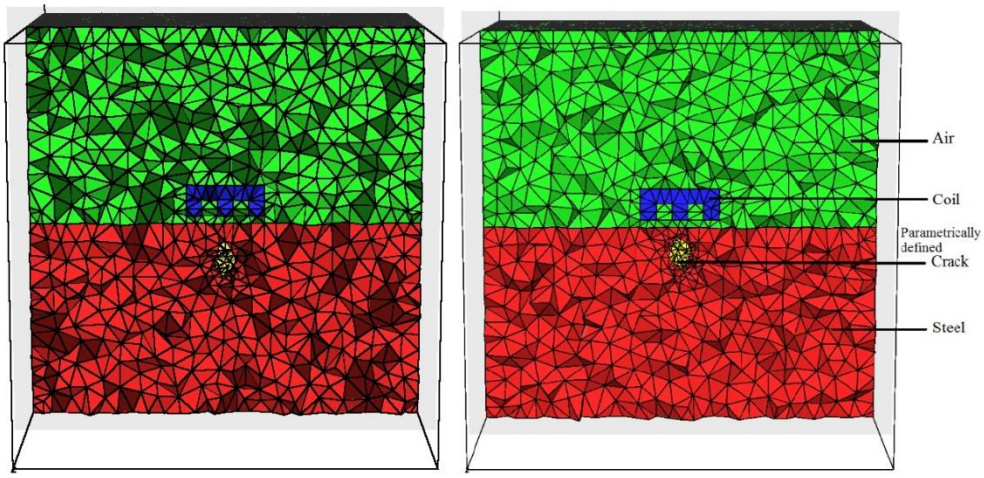

Figure 4.33: Three-dimensional mesh for NDE problem: As parameters change

The last column of Table 4.9 shows the norm error (the formula is also given in the Table 4.9). Average error is 2.5%. Average error is calculated using the following formula:

$$AE = \frac{1}{n} \sum_{variables} \frac{|z_t - z_{re}|}{z_t} \times 100\% \tag{4.6}$$

where n is the number of variables, z_t is the target value and z_{re} is the reconstructed value. This thesis presents the particular solution for a population of 200. This demonstrates the validity of our 3D optimization tools.

Table 4.9: The solution of 3D defect characterization

	True profile			Reconstructed profile			Norm error
#	x	y	z	x	y	z	$\lvert z_t - z_{re} \rvert / z_t$
1	52.853	44.000	50.927	52.853	42.628	50.927	0.0312
2	51.763	42.000	52.427	51.763	43.892	52.427	0.0451
3	48.237	45.000	52.427	48.237	43.682	52.427	0.0293
4	47.147	42.000	50.927	47.147	41.873	50.927	0.0030
5	48.237	45.000	47.573	48.237	44.670	47.573	0.0073
6	50.000	44.000	47.000	50.000	41.893	47.000	0.0479
7	51.763	43.000	47.573	51.763	44.735	47.573	0.0403
8	52.853	43.000	49.073	52.853	42.444	49.073	0.0129
9	51.427	46.000	50.464	51.427	44.239	50.464	0.0383
10	50.000	46.000	51.500	50.000	47.193	51.500	0.0259
11	48.573	47.000	50.464	48.573	46.317	50.464	0.0145
12	50.882	46.500	48.786	50.882	46.483	48.786	0.0004
13	50.000	48.000	50.000	50.000	47.219	50.000	0.0163
14	51.427	40.000	50.464	51.427	37.999	50.464	0.0500
15	48.573	40.000	50.464	48.573	39.835	50.464	0.0041
16	49.118	40.000	48.786	49.118	40.022	48.786	0.0005
17	50.882	40.000	48.786	50.882	38.213	48.786	0.0447
18	50.000	38.000	49.000	50.000	36.903	49.000	0.0289

Chapter 5

Results and Analysis

5.1 Memory Limitation

Recently, Graphics Processing Unit (GPU) computing has had great success in many very large numerical computations. Software developers, researchers, and scientists have been using the GPU for speeding up their computations. Applications taking advantage of this new technology have ranged from quantum chemistry [66] and molecular dynamics [103, 104] to fluid dynamics [105, 106] and cloth simulation [107]. In this work we discuss the often undiscussed GPU memory limitation in finite element optimization. In GPU computing the memory of the Nvidia GPU is limited. This part of this thesis assesses the memory limits in terms of matrix size in light of the various ways to store a large matrix in order to overcome these limits.

We took a 4 cm^2 square conductor with current density = 1 A/mm2 and relative permeability 1. Then we solved the Poisson equation to calculate the magnetic vector potential (A) using FEM

$$\frac{1}{\mu}\nabla^2\bar{A} = -\bar{J} \tag{5.1}$$

In this experiment we defined different number of progressively refined triangles; i.e. 288, 768, 1408 etc. Therefore the final solution can be obtained from the equation, $[P]\ \{A\} = \{Q\}$; where $[P]$ is the matrix and $\{A\}$ and $\{Q\}$ are vectors. In this experiment we used both

storage schemes to reduce the storage capacity. Matrix [*P*] is a symmetric positive definite sparse matrix; each row has approximately 3 to 5 elements in a symmetric half [13], because of the first order mesh.

Table 5.1 shows the total number of matrix elements and storage with matrix size for different storage schemes. Clearly, we can store very large matrices using the profile or sparse storage scheme because in FEM, the matrix [*P*] is a symmetric sparse matrix [13]. According to our result, memory-wise, the sparse storage scheme is much better than the profile storage scheme as to be expected because of fill-in during decomposition with the latter. Although well known, we repeat this investigation to obtain memory limits with CUDA. In inverse problems where many equations need to be solved, this is limiting. Figure 5.1 (A) shows the memory required for the matrix versus matrix elements. Figure 5.1 (B) shows the memory required for profile and sparse storage schemes.

Table 5.1: Number of elements (NE) and storage (in MB) with matrix size for different storage schemes

Matrix Size	NE	Regular	NE(Profile)	Profile	NE(Sparse)	Sparse
100	10500	0.040054	1161	0.004429	1709	0.006519
400	18000	0.068665	10819	0.041271	4421	0.016865
900	814500	3.107071	33329	0.127140	9521	0.036320
1600	2568000	9.796143	75239	0.287014	18421	0.070271
2500	6262500	23.889542	142549	0.543781	29801	0.113682
6000	36030000	137.443543	404459	1.542889	71681	0.273441
8000	64040000	244.293213	697679	2.661434	94221	0.359425
10000	100050000	381.660461	1070099	4.082104	118021	0.450214

Using curve fitting projection we determined the maximum size of the problem that may be attempted within the 4 GB memory limit of the GetForce GTX 970 GPU card that we worked with. Table 5.2 shows the matrix size and corresponding memory size that we need to store the variables. From these calculations we can say that the sparse storage scheme can be used for very large problems of size 50,000K × 50,000K. It takes much lower memory than the profile storage

scheme as remarked. Using the sparse storage scheme we can solve bigger than a 50,000K × 50,000K matrix size before running into memory limits (4 GB limit). With the 24 GB Kepler K80 GPU cards [68] now available, we can solve problems much bigger than 50,000K × 50,000K matrix

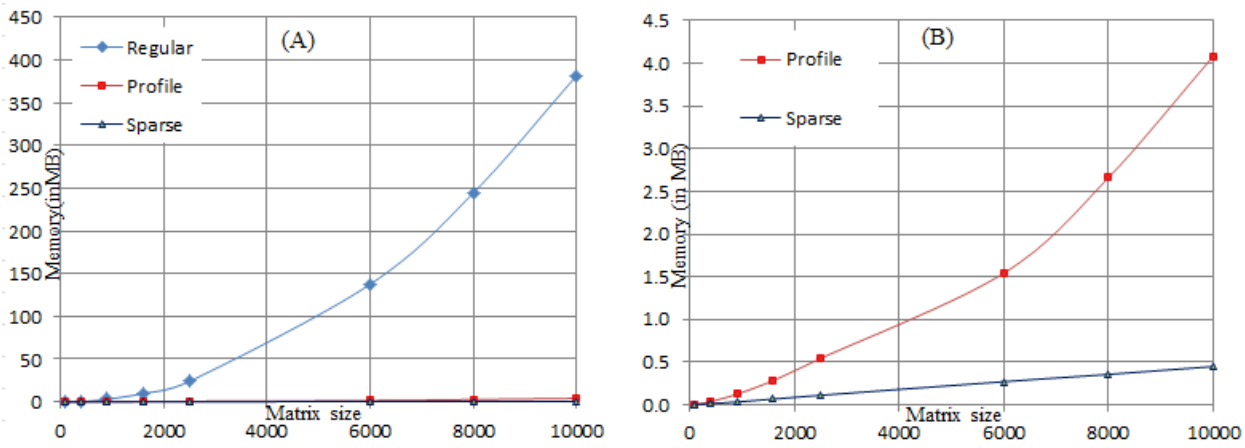

Figure 5.1: Memory vs matrix size

Table 5.2: Projected Memory (in MB) Needs

Size	Regular	Profile	Sparse
20K	1525.2	14.73	0.90
30K	3430.5	32.13	1.35
50K	9526.5	87.06	2.26
100K	38097.0	341.79	4.52
500K	952240.0	8417.70	22.58
1000K	3808900.0	33608.00	45.17
5000K	95220000.0	838940.00	225.85
10000K	380880000.0	3355100.00	451.70
50000K	9522000000.0	83865000.00	2258.50
100000K	38088000000.0	335450000.00	4517.00

Our conjugate gradients method with Jacobi preconditioning gives the results shown in Figure 5.2. We stopped our experiment with size of 10000 × 10000 because if we have a population of 5000 in GA, theoretically we can go up to 10000 × 10000 with sparse storage schemes because of the memory limits as discussed above. If we want a larger population, the problem size should

be decreased. Even-though the sparse storage scheme reduces the memory consumption, we can not solve problems larger than 10000 × 10000 with larger populations than 50. But with the new GPU cards like Kepler K80 [68] that came out as this thesis was completed, memory will not be a problem

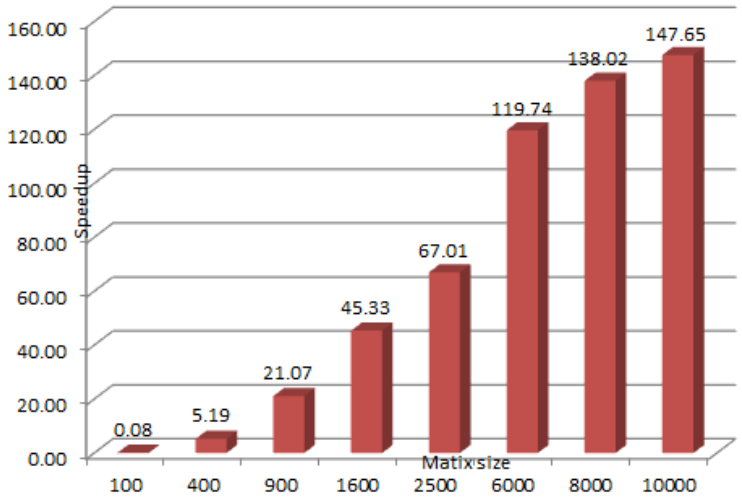

Figure 5.2: Speed-up versus matrix size: Jacobi preconditioned conjugate gradients algorithm

5.2 Element-by-Element Solvers

Again we solved the Poisson equation for the magnetic vector potential A with progressively refined meshes using the finite element method for the Poisson equation for a test problem from magnetics. We defined different number of progressively refined tetrahedrons to obtain results for different matrix sizes. Run time statistics obtained for different mesh sizes is given in Table 5.3 which compares the CPU and GPU calculation times for different sizes of problems for the conjugate gradient element by element (CGEbE) algorithm. This table gives the speedup which is defined as CPU calculation time/GPU calculation time and the speedup per iteration which is defined as CPU calculation time for one iteration/CPU calculation time per iteration. It shows a high speedup of 51.23 for 2,828,782 unknowns which is equivalent to a

matrix size of 2, 828, 782 × 2, 828, 782 in regular first order finite element analysis.

Table 5.3: The CGEbE solution

		CPU		GPU			
Elements		Itns	Time	Itns	Time	Speedup (S)	S per itn
26595	3072	51	1.15	66	0.33	3.4848	4.5098
32682	3843	55	1.63	69	0.33	5.0938	6.3903
45328	5440	60	2.39	69	0.33	7.2424	8.3288
106587	13312	79	7.38	90	0.45	16.4000	18.6835
210024	27852	119	22.34	126	0.73	30.6027	32.4029
273008	36704	130	36.02	134	0.91	39.5824	40.8003
399928	54672	148	54.32	151	1.33	40.8421	41.6700
978905	137639	206	189.24	201	7.11	26.6160	25.9700
1940736	279340	294	524.26	271	25.19	20.8122	19.1841
2569403	372149	329	953.45	310	41.27	23.1027	21.7685
3830869	559546	413	1542.75	346	75.06	20.5536	17.2192
9521059	1413142	645	11120.06	514	308.25	36.0748	28.7480
18879664	2828782	903	46348.39	733	904.54	51.2397	41.5933

This number of iterations depends on both number of unknown elements and total number of elements (see the algorithms in Chapter 3). That is why both iterations and speedup with number of unknown elements and total number of elements are plotted. Figure 5.3 shows the number of iterations vs number of unknowns. Figure 5.4 shows the number of elements vs number of unknowns. In both graphs, the number of iterations increases with problem size. We can see that the number of iterations increases with problem size for both CPU and GPU. The reason for the difference between number of iterations in the GPU and in the CPU is as explained later in this section.

The Figure 5.5 shows the speedup vs number of unknowns for the CGEbE. Figure 5.6 shows the speedups vs number of elements for the CGEbE. The speedups vary with problem size. We can see an erratic nature in the speedups in Figures 5.5 and 5.6. The causes of the erratic speedup need to be investigated [108].

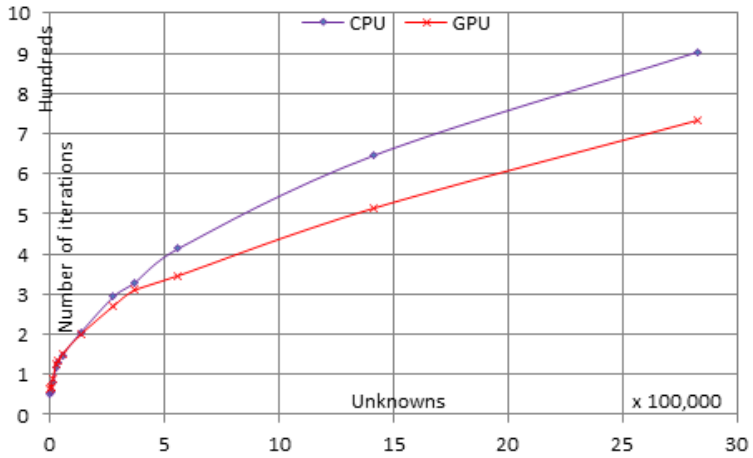

Figure 5.3: Number of iterations vs number of unknowns for CGEbE

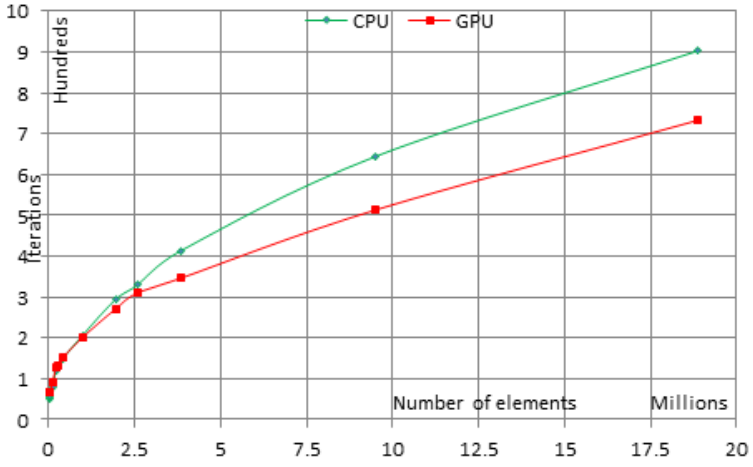

Figure 5.4: Number of iterations vs number of elements for CGEbE

Since we have been working with the regular finite element method for the Poisson equations for magnetics, the bi-conjugate gradient method [34] also works as the conjugate gradient method because $[A] = [A]^T$ [34, 77]. In the situation where the convection effect when dealing with temperature problems is significant, finite element matrix equations take antisymmetric form [83]. Therefore for a generalized finite element package, antisymmetric solvers also should be included with the package.

Table 5.4 shows the CPU and GPU calculation times for the different sizes of problems for

the biconjugate gradient stabilized element by element (BiCGSTABEbE) method. Even- though it gives higher speedup than the CGEbE method, this BiCGSTABEbE method takes a long time to converge. We can see the differences in Tables 5.3 and 5.4. The BiCGSTABEbE method shows a higher speedup of 80.1364 for 54,674 unknowns while the speedup is only 20.55 for 2,828,782 unknowns. So BiCGSTABEbE method can be used to solve small problems (smaller than 54,674 unknowns problems).

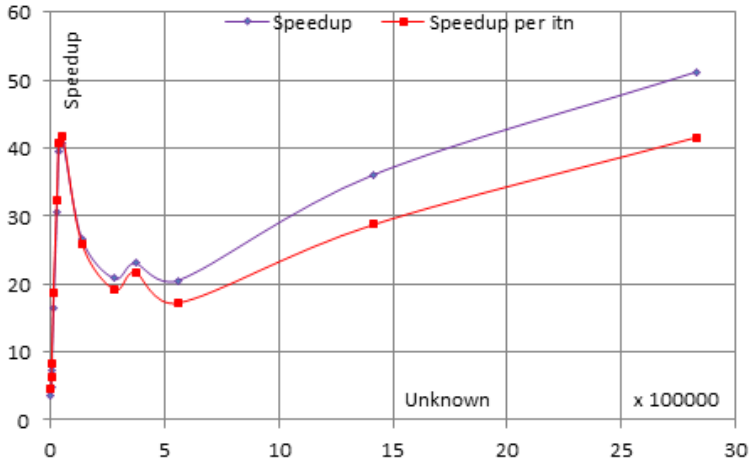

Figure 5.5: Speedup vs number of unknowns for CGEbE

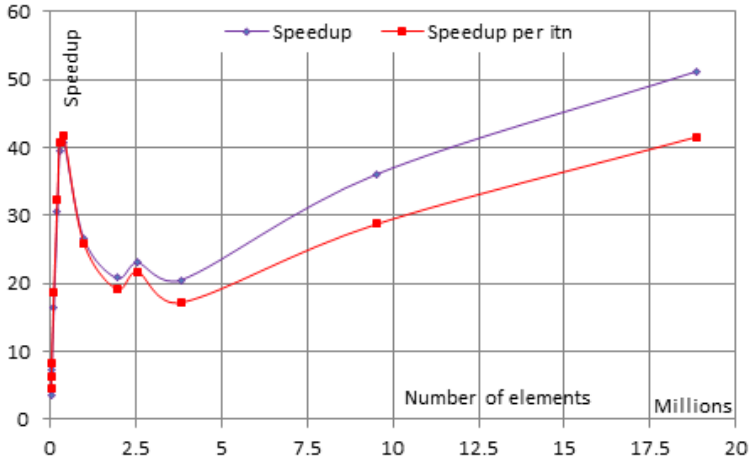

Figure 5.6: Speedup vs number of elements for CGEbE

The Figure 5.7 shows the number of iterations vs number of unknowns for BiCGSTABEbE. Figure 5.8 shows the number of elements vs number of unknowns for BiCGSTABEbE. In both graphs, the number of iterations increases with problem size. This is as to be expected for a well-conditioned problem, such as this.

Table 5.4: The BiCGSTABEbE solution

		CPU		GPU			
Elements		Itns	Time	Itns	Time	Speedup (S)	S per itn
26595	3072	71	3.44	69	0.34	10.1176	9.8326
32682	3843	72	4.27	72	0.34	12.5588	12.5588
45328	5440	83	6.84	72	0.35	19.5429	16.9528
106587	13312	107	21.24	96	0.50	42.4800	38.1129
210024	27852	143	56.68	129	0.88	64.4091	58.1033
273008	36704	145	74.81	134	1.08	69.2685	64.0137
399928	54672	162	123.41	146	1.54	80.1364	72.2217
978905	137639	219	421.69	198	6.41	65.7863	59.4780
1940736	279340	263	1051.64	237	21.24	49.5122	44.6175
2569403	372149	361	1785.89	260	33.43	53.4218	38.4755
3830869	559546	371	5158.10	375	82.78	62.3109	62.9828
9521059	1413142	466	11147.04	569	366.80	30.3900	37.1071

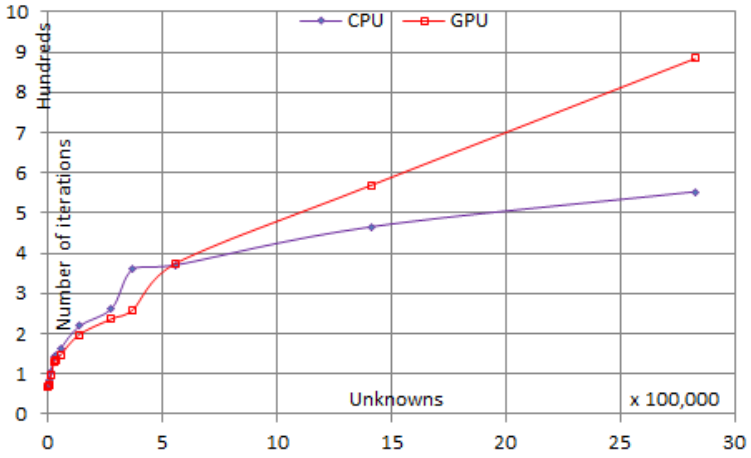

Figure 5.7: Number of unknowns vs number of iterations for BiCGSTABEbE

The Figure 5.9 shows the speedups vs number of unknowns for BiCGSTABEbE. Figure 5.10 shows the speedup vs number of elements for BiCGSTABEbE. The speedups vary with problem size. The speedup is decreasing after 54,674 unknowns in contrast to the speedup increasing for CGEbE after 54,674 unknowns. Again the causes of the erratic speedup need to be investigated [108].

Table 5.4 compares the CPU and GPU calculation times for different sizes of problems for CGEbE algorithm. This table gives speedup and speedup per iteration. It shows high speedup of 102.12 for 54,672 unknowns which is equivalent to a matrix size of 54,672 × 54,672 in regular first order finite element analysis.

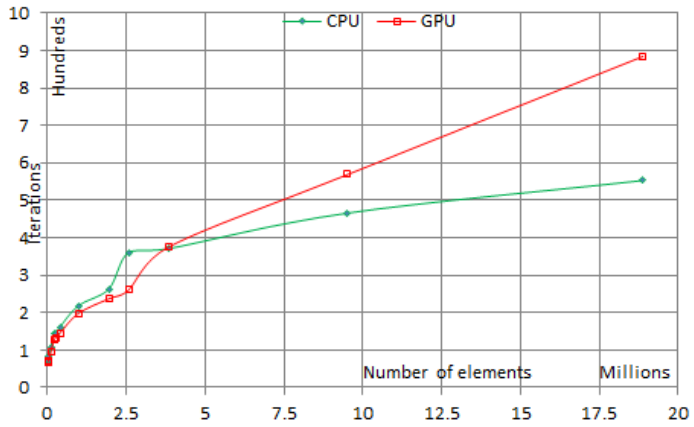

Figure 5.8: Number of elements vs number of iterations for EbEBiCGSTAB

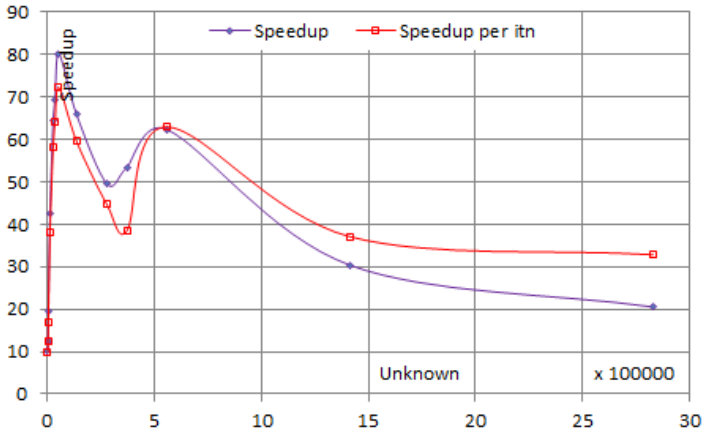

Figure 5.9: Speedup vs number of unknowns for BiCGSTABEbE

Figure 5.11 shows the number of iterations vs number of unknowns for Jacobi EbE. Figure 5.12 shows the number of iterations vs number of elements for Jacobi EbE. In both graphs, the number of iterations increases with problem size as to be expected. We can see that the number of iterations increases with problem size for both CPU and GPU

The Figure 5.13 shows the speedups vs number of unknowns for Jocobi EbE. Figure 5.14 shows the speedups vs number of elements for Jacobi EbE. The speedup is decreasing after 372,149 elements (see Table 5.5).

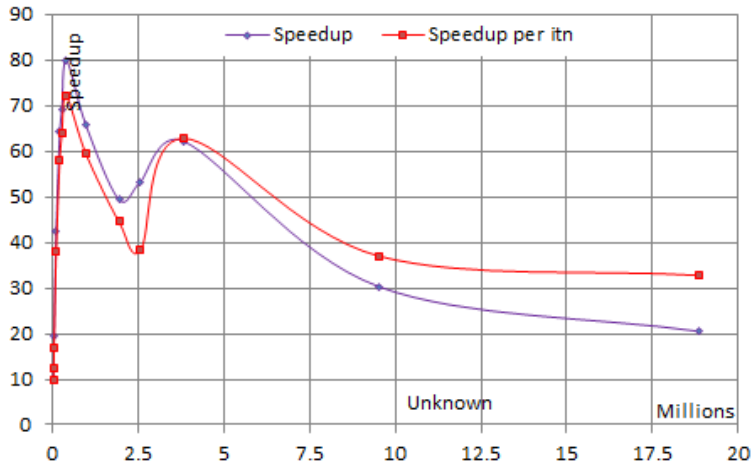

Figure 5.10: Speedup vs number of elements for BiCGSTABEbE

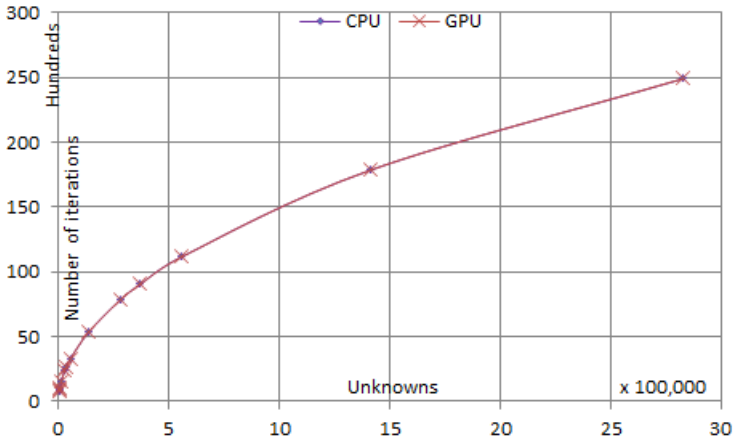

Figure 5.11: Number of unknowns vs number of iterations for JacobiCG

Figures 5.15 and 5.16 show the convergence of the CGEbE algorithm for CPU and GPU respectively. We can see that both convergence patterns are almost the same and that the convergence rate is very high in both figures.

Figures 5.17 and 5.18 show the convergence of the BiCGSTABEbE algorithm for CPU and GPU respectively. We can see that both convergence patterns are slightly different and the convergence rate is high in both figures

Table 5.5: The Jacobi solution

		CPU		GPU			
Elements		Itns	Time	Itns	Time	Speedup (S)	S per itn
26595	3072	806	17.24	807	0.70	24.6286	24.6591
32682	3843	889	23.58	889	0.78	30.2308	30.2308
45328	5440	1019	37.01	1018	0.95	38.9579	38.9197
106587	13312	1557	134.42	1557	1.96	68.5816	68.5816
210024	27852	2371	409.10	2374	4.67	87.6017	87.7126
273008	36704	2712	619.42	2713	6.52	95.0031	95.0381
399928	54672	3320	1129.47	3314	11.06	102.1221	101.9375
978905	137639	5376	6219.27	5374	66.84	93.0471	93.0125
1940736	279340	7826	19026.61	7819	269.75	70.5342	70.4711
2569403	372149	9070	31069.48	9063	452.96	68.5921	68.5392
3830869	559546	11185	56743.83	11176	930.97	60.9513	60.9023
9521059	1413142	17853	217808.09	17856	4364.02	49.9100	49.9184
18879664	2828782	24904	488881.06	24908	12852.42	38.0381	38.0441

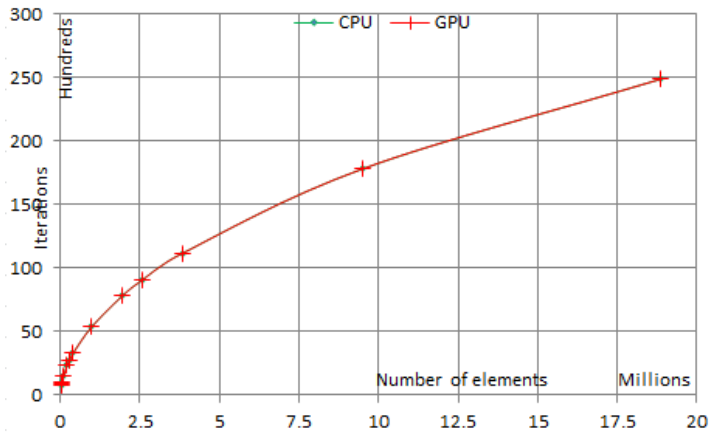

Figure 5.12: Number of elements vs number of iterations for JacobiCG

Figures 5.19 and 5.20 show the convergence of the Jacobi EbE algorithm for CPU and GPU respectively. We can see that both CPU and GPU convergence rates are very slow compared to CGEbE and BiCGSTABEbE. But convergence patterns in CPU and GPU are the same and the convergence pattern is smooth. Here we can see there is not that much difference between number of iterations between GPU and CPU compared to CGEbE and BiCGSTABEbE algorithms. In CGEbE and BiCGSTABEbE methods many statements including element by element process are parallelized in GPU but in the Jacobi method only one statement with element by element executes in the GPU. This is the possible reason for this difference in the number of iterations in CPU and GPU

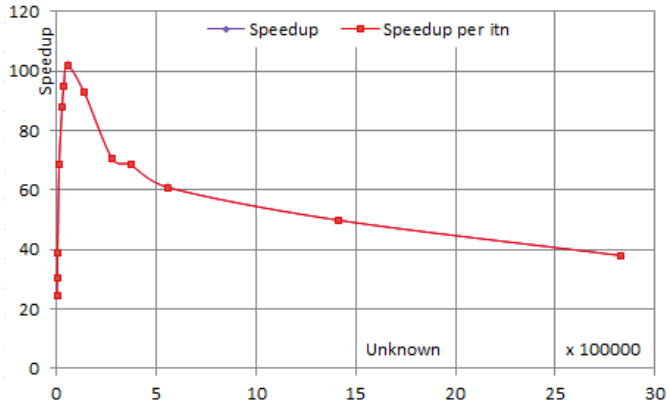

Figure 5.13: Speedup vs number of unknowns for Jacobi EbE

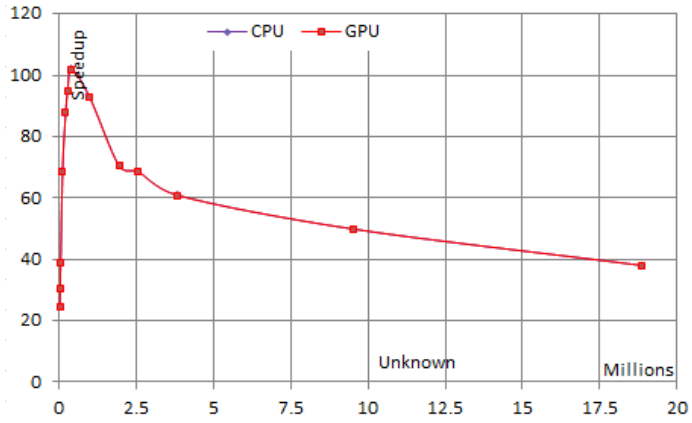

Figure 5.14: Speedup vs number of elements for Jacobi EbE

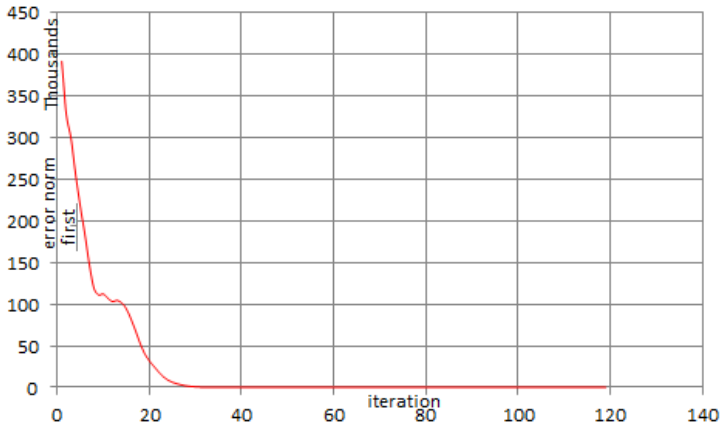

Figure 5.15: Convergence rate of CG in CPU

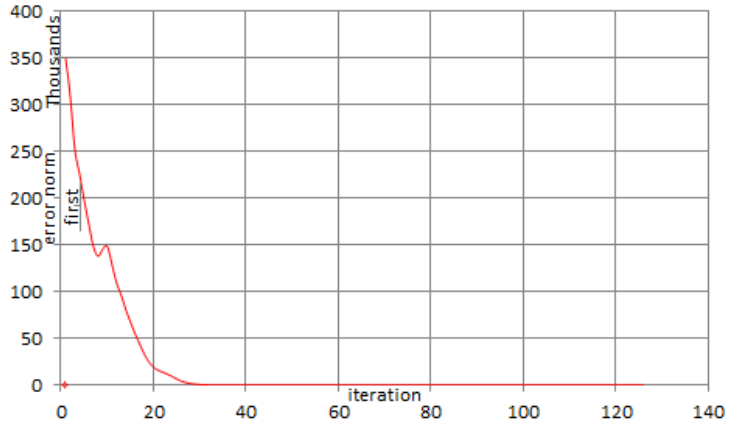

Figure 5.16: Convergence rate of CG in GPU

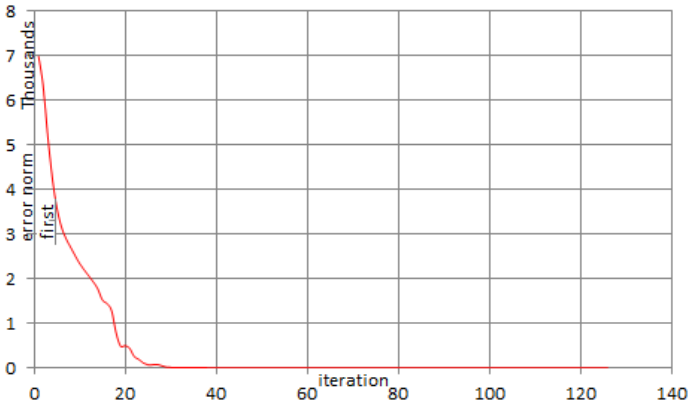

Figure 5.17: Convergence rate of BiCGSTABEbE in CPU

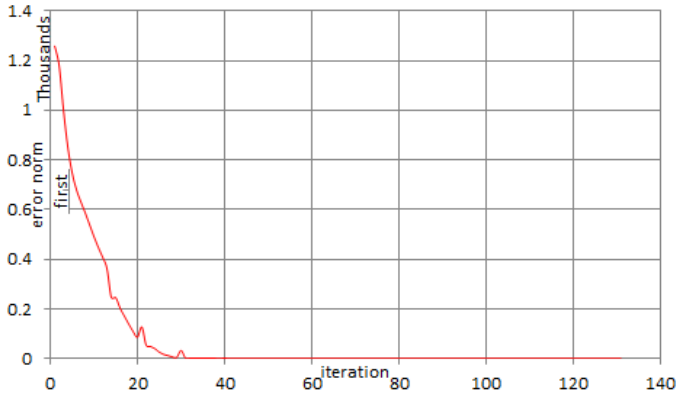

Figure 5.18: Convergence rate of BiCGSTABEbE in GPU

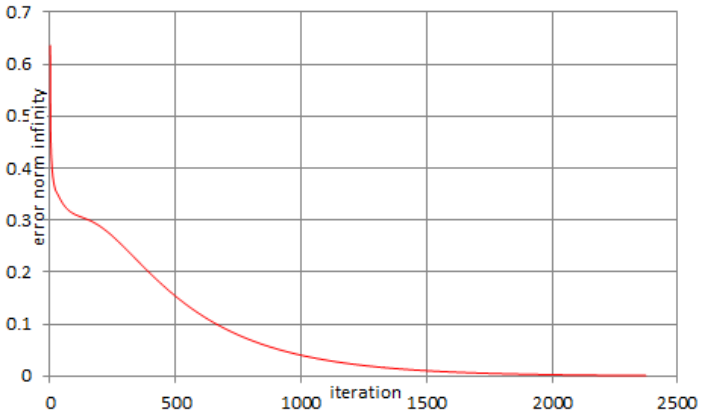

Figure 5.19: Convergence rate of Jacobi in CPU

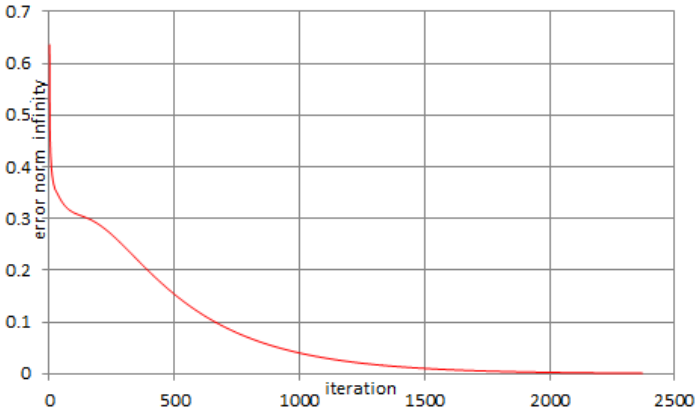

Figure 5.20: Convergence rate of Jacobi in GPU

Figure 5.21 compares the speedup of Jacobi EbE, CGEbE and BiCGSTABEbE methods. In terms of speedup, the Jacobi CG method gives a higher speedup for small problems (less than 372,149 unknowns), the second higher speedup is for BiCGSTABEbE and then CGEbE. But convergence time is much less for CGEbE, followed by BiCGSTABEbE and then Jacobi EbE. For large problems (bigger than 372,149 unknowns, see Tables 5.3, 5.4, 5.5), CGEbE gives a higher speedup, followed by Jacobi EbE and then BiCGSTABEbE. But convergence time is much less for CGEbE, second less for BiCGSATBEbE and then Jacobi EbE (see Tables 5.3, 5.4, 5.5).

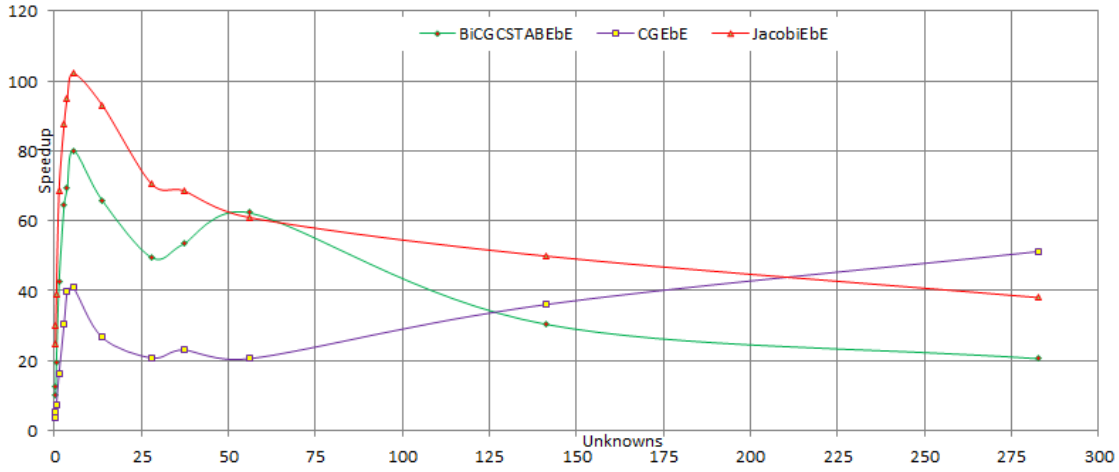

Figure 5.21: Speedup comparison between CGEbE, BiCGSTABEbE and Jacobi EbE algorithms

Figure 5.22 graphically summarizes Kiss *et al.*'s speedups from [109] which are lower than those by the same authors, Kiss *et al.*, in [34] even though both use the same hardware. Kiss *et al.* [109] got 10.01 as their maximum speed up for computations on a single GPU (corresponding to our results which shows a maximum speedup of 102.12). In terms of speedup we got better results while at the same time they got a higher convergence rate than ours with an astoundingly fast rate in terms of iterations (e.g., 79 iterations for a 91,000 X 91,000 matrix on a CPU) but the results of Fernandez *et al.* [33] are more comparable to our Jacobi EbE. Kiss *et al*

[109]'s speedup peaks at 1,339,434 unknowns and decreases with matrix size thereafter. There is an erratic up and down speedup that needs to be investigated [108].

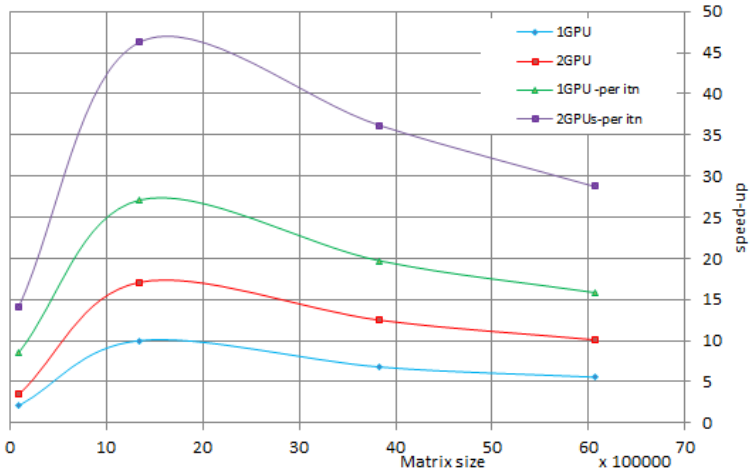

Figure 5.22: Speed-up versus matrix size of Kiss *et al*.

Table 5.6: Speedup ratio between single and double

Matrix size	Jacobi	Gauss-Seidel	GMRES(35)	BiCGSTAB
2000	1.9148	1.9197	1.9773	1.9712
4000	1.5568	1.8194	1.9854	1.9976
8000	2.3471	1.9148	1.9255	1.8784
12000	1.9851	1.8095	2.0099	1.9157
16000	2.2012	1.9276	1.8528	1.9545
20000	2.0449	1.8841	1.8819	1.8850

Table 5.6 shows the speedup ratio of the single and double precision GPU implementation of the direct method for linear systems [110]. Theoretically for memory bound algorithms, double precision work on the GPU has been shown to take twice as much time than single precision arithmetic [111]. However, there are papers where this is not so [108, 110] (see Table 5.6). Communications is a factor but the exact nature is still not known as pointed out in our paper [108]. Devon Yablonski analyzes numerical accuracy issues that are found in many scientific

GPU applications using floating point computation [112]. As he puts it, two widely held myths about floating-point on GPUs are that the CPU's answer is more precise than its GPU version and that computations on the GPU are unavoidably different from the same computations on a CPU [112]. He appears to have dispelled both myths by studying specific applications.

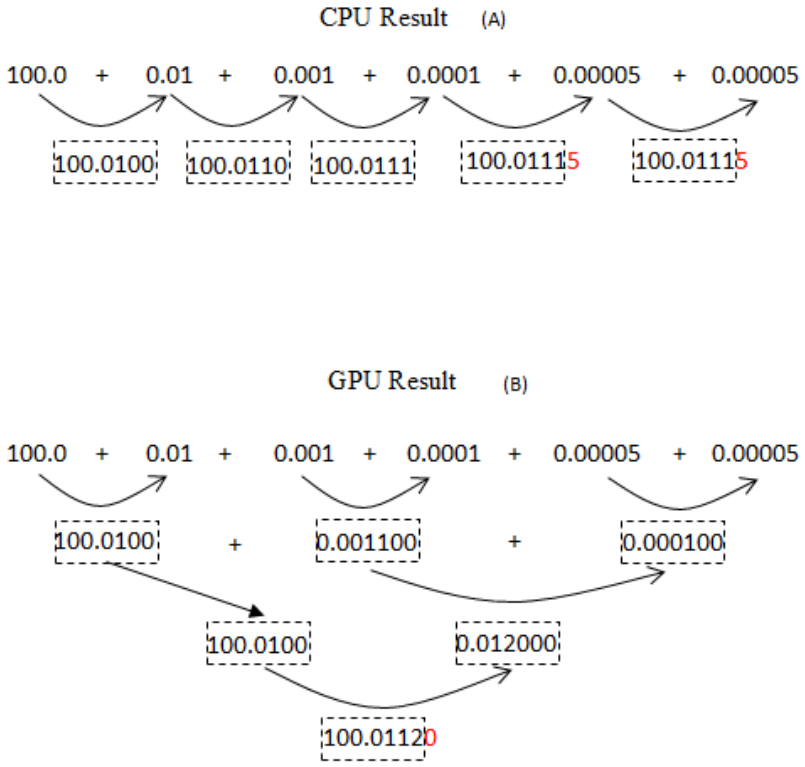

Figure 5.23: Serial addition losing precision. Numbers surrounded by a box represent the actual result floating point value with 7 digits

Accumulating values serially (Figure 5.23 A) will sometimes result in a large value that each successive small value is added to, resulting in diminished accuracy of the results. The reduction style of computation (Figure 5.23 B) avoids the issue in accumulating floating-point values in a way that is similar to binning [112]. Binning describes collecting and computing small groups of values and then computing the final result by combining each result. There is sometimes an erratic up and down speedup [113] like in Figures 5.5, 5.6, 5.9, 5.10, 5.13, 5.14. The causes need

to be investigated as we have pointed out [108]. In a study we did of GPU computation for finite element optimization by the genetic algorithm [3], the speedup showed an unexpected erratic up and down trajectory [108]. This result is seen in other works too such as of Krawezik and Pool [113] as shown in Figure 5.24 and Kiss *et al*. [34] as shown in Figure 5.22. In the absence of an explanation we carry on but a real understanding of the method to obtain the best speedup, requires some investigation.

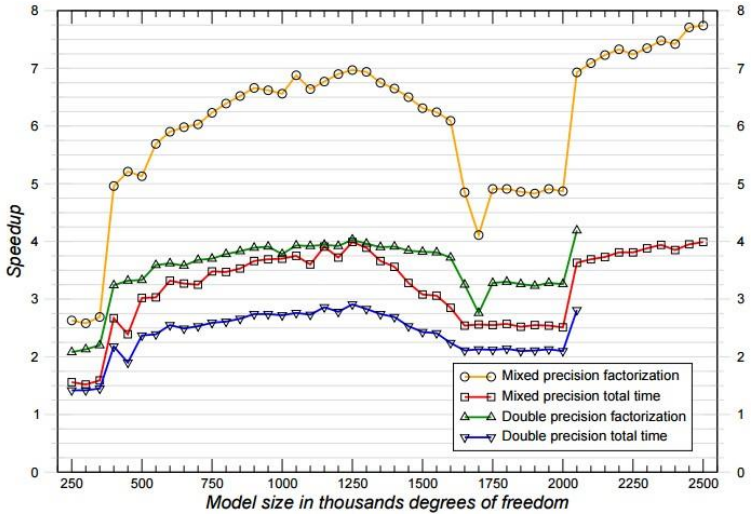

Figure 5.24: Erratic behavior of gain for various methods

While expert programmers have programmed their problems in CUDA C to reap the benefits of speedup, it is to be noted that the compiler is very difficult to work with as pointed out by us [108]. Error messages are still difficult to use in debugging. When memory is violated in one function, the program crashes in another without a proper error message. Debugging is therefore more difficult in CUDA C, particularly because we cannot print intermediate outputs directly from the GPU

Chapter 6

Conclusion and Future Works

The first part of this thesis describes a successful script-based, parameterized mesh generator library for design and NDE developed in C for seamless finite element optimization. Unlike other such systems written in inaccessible code and designed for use with specific software, this system is provided as open-source code [114, 115] so that it may be modified by anyone and used with his or her own finite element (FE) optimization system. This system with the finite element analysis and optimization on the GPU makes for quick NDE assessments and other finite element optimization problems in the field [3]. The modified open source codes Triangle and TetGen we have developed for optimization are CPU-based because mesh generation takes little time compared to finite element matrix solution and optimization. As such the massive effort to port them to a GPU is not justified. Moreover they are too complex for porting to the GPU. A large team of engineers can translate that too to the GPU but we do not think the effort is warranted by the gains to be had. The mesh generator has been demonstrated through a successful NDE system for testing army ground vehicle hulls with damage from corrosion or IEDs and some associated optimization problems.

In the second contribution of this thesis, we have presented our GPU-based EbE matrix solution routine that is very fast and takes little memory. It has been specifically developed for finite element optimization. Several finite element solutions are done on parallel GPU threads for use where memory is critical and solution times long. The GPU-parallelized pre- conditioned conjugate gradient (PCG) algorithm with sparse-stored matrix formation is the best way for

PCG. With 24 GB GPU cards now available, EbE processing is not necessary for most practical, single, forward problems. This element by element (EbE) method, how- ever, becomes a must for genetic algorithm optimization where, whether in NDE or device design, several genetic algorithm threads are launched in parallel and memory capabilities are challenged. The EbE Jacobi iterations give better speed-up than the conjugate gradients element by element (CGEbE) for which incomplete cholesky conjugate gradient (ICCG) is not possible because the matrix is never formed in EbE processing.

In our work we use the genetic algorithm where the object function corresponding to every member $\bar{h}$ of a population has to be computed many times to find the minimum. The many members $\bar{h}$ form the genetic search space. Since $\bar{h}$ consists of dimensions and materials of a particular design being examined for its goodness [116], for those dimensions a mesh is constructed, the finite element problem solved and the object function evaluated. The object function itself is computed from a finite element solution involving a matrix equation. Thus we may treat the object function computation as a kernel and launch it on multiple threads, each for a different member of the population. Then within that kernel, we can parallelize the matrix equation solution at a speedup which we shall refer to a *SP*, which depends on problem size. Alternatively, we may do the object function evaluation for each member of the population in sequence and in that process parallelize the matrix computations. Let the population number be *n*. Say the object function evaluation for each member of the population takes $t_0 + t_m$ in time where t_m is the time for the matrix solution and t_0 is the time for other operations. Therefore if we parallelize the operations for the different members of the population, the time for evaluating all the object functions corresponding to the entire population would still be, neglecting coordination time,

$$t = t_0 + t_m \tag{6.1}$$

since these are done simultaneously. Here we have assumed that the work for each member of the population is being done in parallel, and that the time for combining results and other communications is negligible.

On the other hand, if we parallelized the matrix computation, the evaluation of the object function has to be in sequence since we cannot have forking from a parallelized kernel. The total time would then be the number of members in the population multiplied by the time for computing the object function for each member of the population

$$t = n\left[t_0 + \frac{t_m}{SP}\right] \tag{6.2}$$

where SP is the matrix equation solution speedup, tm is the time for the matrix solution, n is population number and t_0 is the time for other operations. A decision on which of the processes is to be parallelized would depend on considerations like this. However we have not seen such considerations in the literature [12]. Further in a recent development CUDA Dynamic Parallelism has been made available on the SM 3.5 architecture GPU [72], we can parallelize both the genetic algorithm and the FE calculations. Suggested future work includes,

- Develop the GPU based parametrized mesh generator for two and three dimensions.
- Develop dynamic parallelization based optimization tools.
- Develop a user friendly PC-based optimization tools for portable NDE use in the field.

APPENDICES

Appendix A: Publications Raised from This Research

Journals

1. S. Sivasuthan, V. U. Karthik, A. Rahunanthan, P. Jayakumar, Ravi Thyagarajan, Lalita Udpa and S.R.H. Hoole; "Addressing Memory and Speed Problems in Nondestructive Defect Characterization: Element-by-Element Processing on a GPU," *Journal of Nondestructive Evaluation*, vol. 34(9), 2015

2. S. Ratnajeevan H. Hoole, Sivamayam Sivasuthan, Victor U. Karthik and Paul R.P. Hoole "Flip-teaching Engineering Optimization, Electromagnetic Product Design and Nondestructive Evaluation in a Semesters Course," *Computer Applications in Engineering Education Journal(CAEE)*, vol. 23, pp. 374-382, 2015

3. Victor U. Karthik, Sivamayam Sivasuthan Arunasalam Rahunanthan, Ravi S. Thyagarajan, Paramsothy Jayakumar, Lalita Udpa and S. Ratnajeevan H. Hoole, "Faster, more accurate, parallelized inversion for shape optimization in electroheat problems on a graphics processing unit (GPU) with the real-coded genetic algorithm," *COMPEL International Journal of Computations and Mathematics in Electrical*, vol. 34, no. 1, pp. 344-356, Jan. 2015.

4. S. Sivasuthan, V. U. Karthik, A. Rahunanthan, P. Jayakumar, Ravi Thyagarajan, Lalita Udpa and S.R.H. Hoole; "A Script-based, Parameterized Mesh Generator for Machine Design and Army Ground Vehicle Crack Characterization on a GPU," *IETE Technical Review*, Vol. 32(2), pp. 94-103, 2015.

5. S. Ratnajeevan H. Hoole , Victor U. Karthik, S. Sivasuthan, A. Rahunanthan, R. Thyagarajan , P. Jayakumar, "Finite Elements, Design Optimization, and Non-destructive Evaluation: A Review in Magnetics, and Future Directions in GPU-based, Element- by-Element Coupled Optimization and NDE," *International Journal of Applied Electromagnetics and Mechanics*, vol. 47(3), pp. 607-627, 2015

6. S. Ratnajeevan H. Hoole, Sivamayam Sivasuthan Victor U. Karthik, Arunasalam Rahunanthan, Ravi S. Thyagarajan and Paramsothy Jayakumar, "Electromagnetic Device Optimization: The Forking of Already Parallelized Threads on Graphics Processing Units," *Applied Computational Electromagnetics Society (ACES) Journal*, vol. 29(9), pp. 677-684, 2014

7. Sivamayam Sivasuthan, Victor U. Karthik, Arunasalam Rahunanthan, Ravi S. Thyagarajan, Paramsothy Jayakumar and S. Ratnajeevan H. Hoole, "GPU Computations for Finite Element Optimization: Some Issues to be Addressed," *Revue roumaine des sciences techniques-Serie Electrotechnique et Energetique* (accepted)In press Vol. 60, No. 3, 2015.

Peer-reviewed Conference Papers

1. S. Sivasuthan, P. Jayakumar, R. S. Thyagarajan, and S. R. H. Hoole, "A Parameterized 3D Mesh Generator for Optimization in NDE and Shape Design on a GPU*," The 31st International Review of Progress in Applied Computational Electromagnetics*, 22-26 March 2015, in Williamsburg, VA

2. S.R.H. Hoole, S. Sivasuthan, "GPU Computations for Finite Element Optimization: A Review of the Methodology and Problems for Study," *ISFEE Conference 2014* (Accepted) November 28-29 2014, Bucharest, Romania

3. S. Sivasuthan, V. U. Karthik, A. Rahunanthan, P. Jayakumar, Ravi Thyagarajan, Lalita Udpa and S.R.H. Hoole; "GPU Computation: Why Element by Element Conjugate Gradients," *16th Biennial IEEE Conference on Electromagnetic Field Computation (CEFC)*, May 25-28 2014, Annecy, France.

4. S. Sivasuthan, V. U. Karthik, A. Rahunanthan, P. Jayakumar, Ravi Thyagarajan, Lalita Udpa and S.R.H. Hoole, "A Script-based, Parameterized Mesh Generator Li- brary for CoupledGradient Design and NDE," *16th Biennial IEEE Conference on Electromagnetic Field Computation (CEFC)*, May 25-28 2014, Annecy, France

5. Sivamayam Sivasuthan , "The General Purpose Parameter Based Two Dimensional Mesh Generator," *In proceedings of 2014 American Society For Engineering Education, NCS Conference*, April 4 and 5, 2014, Oakland University, Rochester, MI.

6. S. Sivasuthan Victor U. Karthik, Arunasalam Rahunanthan, Ravi S. Thyagarajan, Paramsothy Jayakumar, and S. Ratnajeevan H. Hoole, "The Finite Element Method in Optimization: The Forking of Already Parallelized Threads on Graphics Processing Units to Realize Speedup," *The 30th International Review of Progress in Applied Computational Electromagnetics*, March 23th-27th, 2014 in Jacksonville, Florida

7. S. Sivasuthan, Victor U. Karthik and S. Ratnajeevan Hoole, "CUDA Memory Limitation in Finite Element Optimization to Reconstruct Cracks," in 40th Annual

Review of Progress in Quantitative Nondestructive Evaluation, edited by Dale E. Chimenti, Leonard J. Bond, and Donald O. Thompson, *AIP Conference Proceedings 1581,* 1967- 1974 , American Institute of Physics, Melville, NY.

8. Victor U. Karthik, S. Sivasuthan and S. Ratnajeevan Hoole, "Parallel Implementation of the Genetic Algorithm on NVIDIA GPU Architecture for Synthesis and Inversion," in 40th Annual Review of Progress in Quantitative Nondestructive Evaluation, edited by Dale E. Chimenti, Leonard J. Bond, and Donald O. Thompson, *AIP Conference Proceedings 1581,* 1991-1998 , American Institute of Physics, Melville, NY.

9. S. Sivasuthan V. U. Karthik, P. R. P. Hoole, and S. R. H. Hoole, "The Finite Element Method in Electrical Engineering Optimization: Parallelization on Graphics Processing Units Realizing High Speedup Without Memory Limits," *ICPAM-LAE satellite conference,* in Port Moresby, Papua New Guinea, 2013.

10. Victor U. Karthik, S. Sivasuthan, A. Rahunanthan, P. Jayakumar, R. Thyagarajan, S. Ratnajeevan Hoole, "Finite Element Optimization for Nondestructive Evaluation on a Graphics Processing Unit for Ground Vehicle Hull Inspection*," In proceedings of NDIA Ground Vehicle Systems Engineering and Technology Symposium*, Troy, MI, August 2013.

11. S. Sivasuthan and S. R. H. Hoole. "Software Tools for Inverse Problem Solution," *Inverse problem symposium*, East Lansing, 2012 (Digest)

Appendix B: Sample Input File 2D

```
#A set of pointsin 2D(* WITHOUT VARIABLE POINTS).
# Number of nodes is 9 number of variables is 5
9   5
#And here are the nine points.
1 0.0  0.0
2     10. 0 0. 0
3     20.0  0. 0
4     10. 0 10. 0
5     0. 0  10. 0
6     2. 0  2. 0
7     4. 0  2. 0
8     4. 0  4. 0
9     2.0   4.0
# variable points
# number of points in first draw. Then coordinates
10  20.0 1.85
11 18 1.90
12    15.5 2.10
13    13 2.40
14    10  3
#segments, 1st line --number of segments following lines are segments (node
numbers, each
segment has two node numbers and a marker to identify the boundary elements.
#number of segments
15
#segments (two nodes) and a marker
1     3     10    1
2     4     5     2
3     6     7     -1
4     7     8     -1
5     1     2     -1
6     2     3     -1
7     5     1     -1
8     14    4     -1
9     10    11    -1
10    11    12    -1
11    12    13    -1
12    13    14    -1
13    2     14    -1
14    8     9     -1
15    9     6     -1
#segment markers used to identify the boundaries and set boundary
#conditions. do not give 0 or 1 to segment marker because already
#fixed as a default, 3rd column is boundary condition value
2
#asdf
```

```
1 2 0
2 3 0
#regions, number of regions x y   coordinates of region, regional
#attribute  (for whole mesh), Area constraint that will not be used
#we can leave one region without any assignments we have to assign for
#this case 0 0 0 0 but we can give properties to this region

2
1 1 0.5 1 0.1
2   12  3   2   0.9
#
#properties of regions, first number of properties then property
#values
2
1 1 1.32
2   1.90  9.312
# holes, number of holes x y   coordinates of the hole
1
1 3 3
#type
0
#measuring points
10
#point coordinates
1 1.1 1.10
2   2.1 1.19
3   3.1 1.18
4   3.3  1.17
5   3.6  1.16
6   3. 9   1.15
7   4 .1 1.14
8   4. 4   1.13
9   4. 9   1.12
10  5.1 1.1
```

Appendix C: Sample Input File 3D

```
#3DMesh Input File
#Number of  points <--> Number of variable points
36    22
#13   3     0     1
1     0     0     0
2     100   0     0
3     100   0     100
4     0     0     100
5     0     50    0
6     100   50    0
7     100   50    100
8     0     50    100
9     0     100   0
10    100   100   0
11    100   100   100
12    0     100   100

#coil
13    40.0 52.0 48.0
14    44.0 52.0 48.0
15    48.0 52.0 48.0
16    52.0 52.0 48.0
17    56.0 52.0 48.0
18    60.0 52.0 48.0
19    40.0 52.0 52.0
20    44.0 52.0 52.0
21    48.0 52.0 52.0
22    52.0 52.0 52.0
23    56.0 52.0 52.0
24    60.0 52.0 52.0
25    44.0 56.0 48.0
26    48.0 56.0 48.0
27    52.0 56.0 48.0
28    56.0 56.0 48.0
29    44.0 56.0 52.0
30    48.0 56.0 52.0
31    52.0 56.0 52.0
32    56.0 56.0 52.0
33    40.0 60.0 48.0
34    60.0 60.0 48.0
35    60.0 60.0 52.0
36    40.0 60.0 52.0
#variables
37    52.85316955 44    50.92705098
38    51.76335576 42    52.42705098
39    50    43    53
```

```
40    48.23664424 45    52.42705098
41    47.14683045 42    50.92705098
42    47.14683045 43    49.07294902
43    48.23664424 45    47.57294902
44    50    44    47

45    51.76335576       43    47.57294902
46    52.85316955       43    49.07294902
47    51.42658477       46    50.46352549
48    50    46    51.5
49    48.57341523       47    50.46352549
50    49.11832212       46    48.78647451
51    50.88167788       46.5 48.78647451
52    50    48    50
53    51.42658477       40    50.46352549
54    50    40    51.5
55    48.57341523       40    50.46352549
56    49.11832212       40    48.78647451
57    50.88167788       40    48.78647451
58    50    38    49


#number of  faces,   boundary markers
53    1
#  no  of  polygons> no  of holes, boundary marker
1     0     3
4     1     2     3     4
1     0     3
4     5     6     7     8
1     0     3
4     9     10    11    12
1     0     3
4     1     2     6     5
1     0     3
4     5     6     10    9
1     0     3
4     2     3     7     6
1     0     3
4     6     7     11    10
1     0     3
4     1     4     8     5
1     0     3
4     5     8     12    9
1     0     3
4     8     7     11    12
1     0     3
4     4     3     7     8
1     0     -1
12    13    14    25    26    15    16    27    28 17 18    34    33
```

```
1     0     -1
12    19    20    29    30    21    22    31    32    23    24    35    36
1     0     -1
4     13    19    36    33
1     0     -1
4     18    24    35    34
1     0     -1
4     33    34    35    36
1     0     -1
4     13    14    20    19
1     0     -1
4     14    20    29    25
1     0     -1
4     25    26    30    29
1     0     -1
4     15    21    30    26
1     0     -1
4     15    16    22    21
1     0     -1
4     16    22    31    27


1     0     -1
4     27    28    32    31
1     0     -1
4     17    23    32    28
1     0     -1
4
#new
1     17

0     18

-1    24    23
3     37    38    47
1     0     -1
4     38    39    48    47
1     0     -1
4     39    40    49    48
1     0     -1
3     40    41    49
1     0     -1
4     41    42    50    49
1     0     -1
3     42    43    50
1     0     -1
4     43    44    51    50
1     0     -1
3     44    45    51
```

```
1     0     -1
4     45    46    47    51
1     0     -1
3     46    37    47
1     0     -1
4     47    48    52    51
1     0     -1
3     48    49    52
1     0     -1
3     49    50    52
1     0     -1
3     50    51    52
1     0     -1
3     37    38    53
1     0     -1
4     38    39    54    53
1     0     -1
4     39    40    55    54
1     0     -1
3     40    41    55
1     0     -1
4     41    42    56    55
1     0     -1
3     42    43    56
1     0     -1
4     43    44    57    56
1     0     -1
3     44    45    57
1     0     -1
4     45    46    53    57
1     0     -1
3     46    37    53
1     0     -1
4     53    54    58    57
1     0     -1
3     54    55    58
1     0     -1
3     55    56    58
1     0     -1


3     56    57    58

#bounday conditions
1
#conditions
1     3     0

# 2 regions
```

```
3
1     10    10    10    1     0.1
2     46    50    50    2     0.01
3     -1    -1    -1    3     1.2

#number of properties
2
1     1.90 2.20
2     2.213.30
3     3.214.30

#number of holes
0

#measuring points
5
1     52.85316955 51    50.92705098
2     51.76335576 51    52.42705098
3     50    51    53
4     48.23664424 51    52.42705098
5     47.14683045 51    50.92705098
#mesh area constraint
10
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] M. Yan, S. Udpa, S. Mandayam, Y. Sun, P. Sacks, and W. Lord, "Solution of inverse problems in electromagnetic NDE using finite element methods," *IEEE Transactions on Magnetics*, vol. 34, no. 5, pp. 2924–2927, 1998.

[2] M. Melanie, "An Introduction to Genetic Algorithms," *Computers Mathematics with Applications*, vol. 32, p. 133, 1996.

[3] V. U. Karthik, S. Sivasuthan, A. Rahunanthan, R. S. Thyagarajan, P. Jayakumar, L. Udpa, and S. R. H. Hoole, "Faster, More Accurate Parallelized Inversion for Shape Optimization in Electroheat Problems on a Graphics Processing Unit (GPU) with the Real-Coded Genetic Algorithm," *COMPEL*, vol. 34, no. 1, pp. 344–356, 2015.

[4] V. U. Karthik, S. Sivasuthan, and S. R. H. Hoole, "Parallel implementation of the genetic algorithm on NVIDIA GPU architecture for synthesis and inversion," in *AIP Conf. Proc.*, pp. 1991–1998, 2014.

[5] G. F. Uler, O. A. Mohammed, and C. Koh, "Utilizing genetic algorithms for the optimal design of electromagnetic devices," *IEEE Transactions on Magnetics*, vol. 30, no. 6, pp. 4296–4298, 1994.

[6] J. Kim, H. B. Lee, H. K. Jung, and S. Y. Hahn, "Optimal design technique for waveguide device," *IEEE Transactions on Magnetics*, vol. 32, no. 3 (2), pp. 1250–1253, 1996.

[7] S. Dong-Joon, C. Dong-Hyeok, C. Jang-Sung, J. Hyun-Kyo, and C. Tae-Kyoung, "Efficiency optimization of interior permanent magnet synchronous motor using genetic algorithms," *IEEE Transactions on Magnetics*, vol. 33, pp. 1880–1883, Mar. 1997.

[8] M. Khanzadeh, M. Malekshahi, and A. Rahmati, "Optimization of loss in orthogonal bend waveguide: Genetic Algorithm simulation," *Alexandria Engineering Journal*, vol. 52, no. 3, pp. 525–530, 2013.

[9] L. Saludjian, J. Coulomb, and A. Izabelle, "Genetic algorithm and Taylor development of the finite element solution for shape optimization of electromagnetic devices," *IEEE Transactions on Magnetics*, vol. 34, no. 5, 1998.

[10] H. Enomoto, K. Harada, Y. Ishihara, T. Todaka, and K. Hirata, "Optimal design of linear oscillatory actuator using genetic algorithm," *IEEE Trans. on Mag.*, vol. 34, no. 5, 1998.

[11] G. N. Vanderplaats, *Numerical Optimization Techniques for Engineering Design: With Applications (Mcgraw Hill Series in Mech. Engineering)*. Mcgraw-Hill College, 1984.

[12] S. R. H. Hoole, S. Sivasuthan, V. U. Karthik, A. Rahunanthan, R. S. Thyagarajan, and P. Jayakumar, "Electromagnetic Device Optimization: The Forking of Already Parallelized Threads on Graphics Processing Units," *ACES Journal*, vol. 29, no. 9, pp. 677–684, 2014.

[13] S. R. H. Hoole, *Computer-aided Analysis and Design of Electromagnetic Devices*. New York: Elsevier, 1989.

[14] E. Ellobody, R. Feng, and B. Young, "Design Examples of Metal Tubular Connections," *Finite Element Analysis and Design of Metal Structures*, pp. 182–205, 2014.

[15] M. V. K. Chari and P. P. Silvester, *Finite Elements in Electrical and Magnetic Field Problems (Wiley Series in Numerical Methods in Engineering).* New York: John Wiley & Sons Inc, 1980.

[16] A. Marrocco and O. Pironneau, "Optimum design with lagrangian finite elements: De- sign of an electromagnet," *Computer Methods in Applied Mechanics and Engineering*, vol. 15, no. 3, pp. 277–308, 1978.

[17] J. S. Arora and E. J. Haug, "Efficient optimal design of structures by generalized steepest descent programming," *International Journal for Numerical Methods in Engineering*, vol. 10, no. 4, pp. 747–766, 1976.

[18] O. Pironneau, *Optimal Shape Design for Elliptic Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984.

[19] S. Subramaniam, A. Arkadan, and S. Ratnajeevan Hoole, "Optimization of a magnetic pole face using linear constraints to avoid jagged contours," *IEEE Transactions on Magnetics*, vol. 30, no. 5, pp. 3455–3458, 1994.

[20] J. Haslinger and P. Neittaanmaki, *Finite Element Approximation for Optimal Shape, Material and Topology Design*, 2nd Edition. Wiley, 1996.

[21] K. Preis, C. Magele, and O. Biro, "FEM and evolution strategies in the optimal design of electromagnetic devices," *IEEE Tran. on Magnetics*, vol. 26, no. 5, pp. 2181– 2183, 1990.

[22] K. Wijesinghe, "The First General Purpose Optimization Software for Electromagnetic Device Optimization," *Annual Transaction of IESL*, pp. 141–149, 2003.

[23] S. Sivasuthan, V. U. Karthik, and S. R. H. Hoole, "CUDA memory limitation in finite element optimization to reconstruct cracks," *in AIP Conf. Proc*. 1581, pp. 1967–1974, 2014.

[24] S. Sivasuthan, V. U. Karthik, A. Rahunanthan, P. Jayakumar, R. Thyagarajan, L. Udpa, and S. Hoole, "A Script-Based, Parameterized Finite Element Mesh for Design and NDE on a GPU*," IETE Technical Review*, vol. 32, pp. 94–103, Mar. 2015.

[25] J. R. Shewchuk, "Applied Computational Geometry towards Geometric Engineering," *Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator*, vol. 1148, pp. 203–222, 1996.

[26] J. R. Shewchuk, "*Delaunay refinement algorithms for triangular mesh generation," Computational Geometry*, vol. 22, pp. 21–74, May 2002.

[27] J. Schoberl, "NETGEN An advancing front 2D/3D-mesh generator based on abstract rules," *Computing and Visualization in Science*, vol. 1, pp. 41–52, July 1997.

[28] The CGAL Project, "CGAL User and Reference Manual," *CGAL Editorial Board 4.2*, 2013.

[29] C. Geuzaine and J. Remacle, "Gmsh: A 3-D finite element mesh generator with built- in pre and post-processing facilities," *International Journal for Numerical Methods in Engineering*, pp. 1–24, 2009.

[30] C. J. Conroy, E. J. Kubatko, and D. W. West, "ADMESH: An advanced, automatic unstructured mesh generator for shallow water models," *Ocean Dynamics*, vol. 62, pp. 1503–1517, Nov. 2012.

[31] T. Chen, J. Johnson, and W. D. Robert, *A Vector Level Control Function for Generalized Octree Mesh Generation*. Vienna: Springer Vienna, 1995.

[32] X. W. Ma, G. Q. Zhao, and L. Sun, "AUTOMESH2D/3D: robust automatic mesh generator for metal forming simulation," *Materials Research Innovations*, vol. 15, pp. s482–s486, Feb. 2011.

[33] D. M. Fernandez, M. M. Dehnavi, W. J. Gross, and D. Giannacopoulos, "Alternate Parallel Processing Approach for FEM," *IEEE Trans. on Mag.*, vol. 48, pp. 399–402, Feb. 2012.

[34] I. Kiss, S. Gyimothy, Z. Badics, and J. Pavo, "Parallel realization of the element-by-

element FEM technique by CUDA," *IEEE Trans. on Magnetics*, vol. 48, no. 2, pp. 507–510, 2012.

[35] T. Okimura, T. Sasayama, N. Takahashi, and S. Ikuno, "Parallelization of Finite Element Analysis of Nonlinear Magnetic Fields Using GPU," *IEEE Transactions on Magnetics*, vol. 49, pp. 1557–1560, May 2013.

[36] A. Kakay, E. Westphal, and R. Hertel, "Speedup of FEM micromagnetic simulations with graphical processing units," *IEEE Trans. on Magnetics*, vol. 46, pp. 2303–2306, 2010.

[37] W. Khamsen, A. Aurasopon, and W. Sa-ngiamvibool, "Power Factor Improvement and Voltage Harmonics Reduction in Pulse Width Modulation AC Chopper Using Bee Colony Optimization," *IETE Technical Review*, vol. 30, no. 3, p. 173, 2013.

[38] K. Deb, "COIN Lab," http://www.egr.msu.edu/˜kdeb/COIN.shtml.

[39] J. Muller, "On triangles and flow," Special Section on Software Agents in Electronic, 1996.

[40] S. Niu, Y. Zhao, S. L. Ho, and W. N. Fu, "A parameterized mesh technique for finite element magnetic field computation and its application to optimal designs of lectromagnetic devices," in *IEEE Trans. on Magnetics*, vol. 47, pp. 2943–2946, 2011.

[41] "FLUX 10 2D and 3D Applications New features." http://www.jaewoo.com/material/magazinefolder/jwnews/0710/10_New_Features.pdf. [online accessed 2013-08-01].

[42] J. Xin, N. Lei, L. Udpa, and S. S. Udpa, "Nondestructive Inspection Using Rotating Magnetic Field Eddy-Current Probe," *IEEE Transactions on Magnetics*, vol. 47, pp. 1070–1073, May 2011.

[43] "Forge." http://forge-mage.g2elab.grenoble-inp.fr/project/got. [online accessed 14-07-26].

[44] K. Vishnukanthan and K. Markus, "Parallel finite element mesh generator using mul- tiple GPUs," in *14th International Conferennce on Computing in Civil and Building Engineering*, pp. 1–8, Publishing House ASV, 2012.

[45] S. R. H. Hoole, K. Weeber, and S. Subramaniam, "Fictitious minima of object functions, finite element meshes, and edge elements in electromagnetic device synthesis," *IEEE Transactions on Magnetics*, vol. 27, no. 6, pp. 5214–5216, 1991.

[46] A. Vaidya, S. H. Yu, J. St. Ville, D. T. Nguyen, and S. D. Rajan, "Multiphysics CAD-

Based Design Optimization," *Mechanics Based Design of Structures and Machines*, vol. 34, pp. 157–180, July 2006.

[47] C. Talischi, G. H. Paulino, A. Pereira, and I. F. M. Menezes, "PolyTop: a Matlab implementation of a general topology optimization framework using unstructured polygonal finite element meshes," *Structural and Multidisciplinary Optimization*, vol. 45, pp. 329–357, Jan. 2012.

[48] H. Si, "TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator," *ACM Transactions on Mathematical Software*, vol. 41, pp. 1–36, Feb. 2015.

[49] H. Si, "TetGen, a quality tetrahedral mesh generator and three-dimensional Delaunay triangulator, 2007," URL http://tetgen. berlios.de, 2006.

[50] B. Joe, "Construction of Three-Dimensional Improved-Quality Triangulations Using Local Transformations," *SIAM Journal on Scientific Computing*, vol. 16, pp. 1292– 1307, Nov. 1995.

[51] D. J. Mavriplis, "An Advancing Front Delaunay Triangulation Algorithm Designed for Robustness," *Journal of Computational Physics*, vol. 117, pp. 90–101, Mar. 1995.

[52] R. Schneiders, "Algorithms for Quadrilateral and Hexahedral Mesh Generation," in *Proceedings of the VKI Lecture series on Computational Fluid Dynamic*, pp. 2000– 2004, 2000.

[53] D. T. Lee and B. J. Schachter, "Two algorithms for constructing a Delaunay triangulation," *International Journal of Computer & Information Sciences*, vol. 9, pp. 219–242, 1980.

[54] S. Fortune, "A sweepline algorithm for Voronoi diagrams," *Algorithmica*, vol. 2, pp. 153–174, 1987.

[55] C. L. Lawson, *Software for C1 Surface Interpolation. Mathematical Software III*. Academic p ed., 1977.

[56] P. Su and D. R. L. Scot, "A comparison of sequential Delaunay triangulation algorithms," *Computational Geometry*, vol. 7, pp. 361–385, Apr. 1997.

[57] J. Ruppert, "A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation," *Journal of Algorithms*, vol. 18, pp. 548–585, May 1995.

[58] "CEDRAT." http://www.cedrat.com/en/software/got-it.html. [online accessed 2014-04-21].

[59] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2011.

[60] "Merge Sort." http://webdocs.cs.ualberta.ca/~holte/T26/merge-sort.html. [online accessed 2015-02-11].

[61] D. Robilliard, V. Marion-Poty, and C. Fonlupt, "Genetic programming on graphics processing units," *Genetic Programming and Evolvable Machines*, vol. 10, pp. 447– 471, Oct. 2009.

[62] M. L. Wong and T. T. Wong, "Implementation of Parallel Genetic Algorithms on Graphics Processing Units," *Intelligent and Evolutionary Systems*, vol. 187, pp. 197– 216, 2009.

[63] S. Sivasuthan, V. U. Karthik, A. Rahunanthan, P. Jayakumar, R. S. Thyagarajan, L. Udpa, and S. R. H. Hoole, "Addressing Memory and Speed Problems in Nondestructive Defect Characterization: Element-by-Element Processing on a GPU*," Journal of Nondestructive Evaluation*, vol. 34, no. 2, 2015.

[64] S. R. H. Hoole, "Optimal design, inverse problems and parallel computers," *IEEE Transactions on Magnetics*, vol. 27, no. 5, pp. 4146–4149, 1991.

[65] G. Mahinthakumar and S. R. H. Hoole, "A parallel conjugate gradients algorithm for finite element analysis of electromagnetic fields," *Journal of Applied Physics*, vol. 67, no. 9, p. 5818, 1990.

[66] C. Cecka, A. J. Lew, and E. Darve, "Assembly of finite element methods on graph- ics processors," *International Journal for Numerical Methods in Engineering*, vol. 85, pp. 640–669, Feb. 2011.

[67] "GPU." http://www.nvidia.com/object/gpu.html. [online accessed 2014-07-26].

[68] "NVIDIA Tesla." http://www.nvidia.com/object/tesla-servers.html. [accessed 2015-07-26].

[69] "NVIDIA programming guide."http://docs.nvidia.com/cuda/ cuda-c-programming-guide/index.html\#axzz3gSPnvaGC. [online accessed 2013-08-01]

[70] "Tuning CUDA Applications for Kepler." http://www.nvidia.com/object/ cudaget.html. [online accessed 2014-07-26].

[71] W. Wu and P. A. Heng, "A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting," *in Computer Animation and Virtual World*s, vol. 15, pp. 219–227, 2004.

[72] "NVIDIA CUDA Toolkit Release Notes." http://docs.nvidia .com/cuda/ cuda-toolkit-release-notes/index.html. [online accessed 2013-08-01].

[73] D. R. Kincaid, J. R. Respess, D. M. Young, and R. R. Grimes, "Algorithm 586: IT- PACK 2C: A FORTRAN Package for Solving Large Sparse Linear Systems by Adaptive Accelerated Iterative Methods*," ACM Transactions on Mathematical Software*, vol. 8, pp. 302–322, Sept. 1982.

[74] T. J. Hughes, I. Levit, and J. Winget, "An element-by-element solution algorithm for problems of structural and solid mechanics," *Computer Methods in Applied Mechanics and Engineering*, vol. 36, pp. 241–254, Feb. 1983.

[75] Y. Saad, Iterative Methods for Sparse Linear Systems, vol. 3. 2003.

[76] G. F. Carey, E. Barragy, R. McLay, and M. Sharma, "Element-by-element vector and parallel computations," *Communications in Applied Numerical Methods*, vol. 4, pp. 299–307, May 1988.

[77] C. Heusser "Conjugate gradient-type algorithms for a finite-element discretization of the Stokes equations," *Journal of Computational and Applied Mathematics*, vol. 39(1), pp. 23–37, Feb. 1992.

[78] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. 1994.

[79] J. Erhel, A. Traynard, and M. Vidrascu, "An element-by-element preconditioned conjugate gradient method implemented on a vector computer," *Parallel Computing*, vol. 17, pp. 1051–1065, Nov. 1991.

[80] A. Wathen, "An analysis of some element-by-element techniques," *Computer Methods in Applied Mechanics and Engineering*, vol. 74, pp. 271–287, Sept. 1989.

[81] V. Faber and T. Manteuffel, "Necessary and sufficient conditions for the existence of a conjugate gradient method," *SIAM J. Numer. Anal.*, vol. 21, pp. 352–362, 1984.

[82] V. V. Voevodin, “The problem of non-self-adjoint generalization of the conjugate gradient method is closed,” *USSR Comput. Maths. Math. Phys*., vol. 23, pp. 143–144, 1983.

[83] M. M. Wang and T. W. Sheu, “An element-by-element BICGSTAB iterative method for three-dimensional steady Navier-Stokes equations*,” Journal of Computational and Applied Mathematics*, vol. 79, pp. 147–165, Mar. 1997.

[84] T. Sheu, C. Fang, and S. Tsai, “Application of an element-by-element BiCGSTAB iterative solver to a monotonic finite element model,” *Computers & Mathematics with Applications*, vol. 37, pp. 57–70, Feb. 1999.

[85] P. Sonneveld, “CGS, A Fast Lanczos-Type Solver for Nonsymmetric Linear systems,” *SIAM Journal on Scientific and Statistical Computing*, vol. 10, no. 1, pp. 36–52, 1989.

[86] H. A. van der Vorst, “Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems,” *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 2, pp. 631–644, 1992.

[87] H. A. van der Vorst, *Iterative Krylov Methods for Large Linear Systems*, vol. 13. Cambridge: Cambridge University Press, 2003.

[88] A. F. P. de Camargos, V. C. Silva, J.-m. Guichon, and G. Munier, “Efficient Parallel Preconditioned Conjugate Gradient Solver on GPU for FE Modeling of Electromagnetic Fields in Highly Dissipative Media,” *IEEE Transactions on Magnetics*, vol. 50, pp. 569–572, Feb. 2014.

[89] R. Helfenstein and J. Koko, “Parallel preconditioned conjugate gradient algorithm on GPU,” in *J. of Computational and Applied Mathematics*, vol. 236, pp. 3584–3590, 2012.

[90] M. Ament, G. Knittel, D. Weiskopf, and W. Straber, “A parallel preconditioned conjugate gradient solver for the Poisson problem on a multi-GPU platform,” in *Proc. of the 18th Euromicro Conf. on Parallel, Distributed and Net.-Based Processing* , pp. 583–592, 2010.

[91] Z. Ning and X. Wang, “A parallel Preconditioned Bi-Conjugate Gradient Stabilized solver for the Poisson problem,” *J. of Computers (Finland),* vol. 7, no. 12, pp. 3088–3095, 2012.

[92] A. F. P. de Camargos and V. C. Silva, “Performance Analysis of Multi-GPU Implementations of Krylov-Subspace Methods Applied to FEA of Electromagnetic Phenomena,” *IEEE Transactions on Magnetics*, vol. 51, no. 3, pp. 1–4, 2015.

[93] C. A. J. Fletcher, *Computational Techniques for Fluid Dynamics 1*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

[94] J. Haslinger and P. Neittaanmaki, *Finite Element Approximation for Optimal Shape, Material and Topology Design*, 2nd Edition. Wiley, 1997.

[95] K. Preis, C. Magele, and O. Biro, "FEM and evolution strategies in the optimal design of electromagnetic devices," IEEE Trans. on Magnetics, vol. 26, no. 5, pp. 2181–2183, 1990.

[96] D. M. Rocke and Z. Michalewicz, "Genetic Algorithms + Data Structures = Evolution Programs," *Journal of the American Statistical Association*, vol. 95, p. 347, Mar. 2000.

[97] J. H. Holland, "Genetic Algorithms and the Optimal Allocation of Trials," *SIAM Journal on Computing*, vol. 2, no. 2, pp. 88–105, 1973.

[98] E. F. Herzberg, D. A. Forman, N. T. O'Meara, and J. C. Tran, "The annual cost of corrosion for army ground vehicles," 2009.

[99] BDM Federal Inc, "Corrosion Detection Technologies," 1998.

[100] E. Herzberg, "The Annual Cost of Corrosion for DOD," *in DoD Corrosion Conference*, pp. 1–11, 2009.

[101] V. U. Karthik, "Reconstructing and Classifying Damage in a 2D Steel Plate Using Non-Destructive Evaluation (NDE) Methods," *in ASEE-NCS, (Oakland), ASEE*, 2014.

[102] V. U. Karthik, Shape optimization using finite element analysis in eddy current testing and electro-thermal coupled problems. PhD Thesis, Michigan State University, 2015.

[103] L. Vogt, R. Olivares-Amaya, S. Kermes, Y. Shao, C. Amador-Bedolla, and A. Aspuru-Guzik, "Accelerating Resolution-of-the-Identity Second-Order Møller-Plesset Quantum Chemistry Calculations with Graphical Processing Units," *Journal of Physical Chemistry A*, vol. 112, no. 10, pp. 2049–2057, 2008.

[104] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, pp. 5342–5359, May 2008.

[105] D. J. Hardy, J. E. Stone, and K. Schulten, "Multilevel summation of electrostatic potentials using graphics processing units," *Parallel Computing*, vol. 35, no. 3, pp. 164–177, 2009.

[106] E. Elsen, P. LeGresley, and E. Darve, "Large calculation of the flow over a hypersonic vehicle using a GPU," *Journal of Comput. Physics*, vol. 227, pp. 10148–10161, Dec. 2008.

[107] D. Goddeke, S. H. Buijssen, H. Wobker, and S. Turek, "GPU acceleration of an unmodified parallel finite element Navier-Stokes solver," in *2009 International Conference on High Performance Computing & Simulation*, pp. 12–21, IEEE, June 2009.

[108] S. Sivasuthan, V. U. Karthik, R. Arunasalam, R. S. Thyagarajan, P. Jayakumar, and S. R. H. Hoole, "GPU Computations for Finite Element Optimization: Some Issues to be Addressed," Revue roumaine des sciences techniques-Serie electrotechnique et energetique, vol. 60(3), no. (accepted- in press), 2015.

[109] I. Kiss, Z. Badics, S. Gyimothy, and J. Pavo, "High locality and increased intra-node parallelism for solving finite element models on GPUs by novel element-by-element implementation," in *2012 IEEE Conf. on High Perf. Extreme Computing, HPEC 2012*.

[110] B. OANCEA, T. ANDREI, and R. M. DRAGOESCU, "Improving the performance of the linear systems solvers using cuda," in *Challenges for the Knowledge Society*, pp. 2036–2045, 2012.

[111] "CUDA Tutorials." http://www.nvidia.com/content/PDF/sc_2010/CUDA_Tutorial/SC10_Accelerating_GPU_Computation_Through_Mixed-Precision_Methods.pdf. [online accessed 2014-07-26].

[112] D. Yablonski, Numerical accuracy differences in CPU and GPGPU codes. MS Thesis-Northeastern University, 2011.

[113] G. Krawezik and G. Poole, "Accelerating the ANSYS Direct Sparse Solver with GPUs," in Symp. on App. Accelerators in High Perf. Computing, (Urbana- Champaign, IL), 2009.

[114] "2DMesh." www.egr.msu.edu/~hoole/FE2DMesh. [online accessed 2014-05-25].

[115] "3DMesh." www.egr.msu.edu/~hoole/FE3DMesh. [online accessed 2015-07-26].

[116] E. Laithwaite, "The goodness of a machine," *Electronics and Power*, vol. 11, no. 3, pp. 101–103, 1965.