

Computational Electromagnetics: Manipulation of the MD-GRAPE Accelerator Board

Senior Comprehensive Research Project



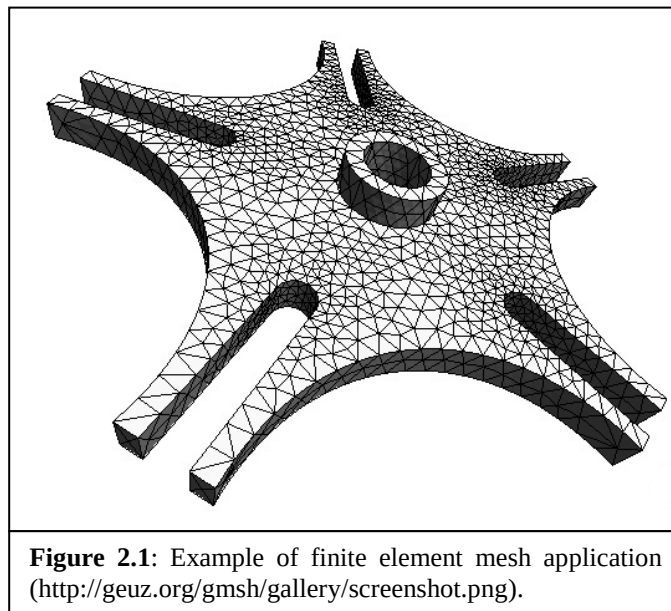
**ALLEGHENY
COLLEGE**
MEADVILLE, PENNSYLVANIA

Adam J. Simbeck
Advisor: Gary Bedrosian
2nd Reader: Jamie Lombardi
Allegheny College
Department of Physics
Meadville, PA
April 24, 2009

Submitted to the Allegheny College Department of Physics to partially
fulfill the requirements for the degree of Bachelor of Science.

Table of Contents

Acknowledgments	2
I. Introduction	3
II. Background Theory: CEM	5
III. Background Theory: FEM Analysis of RHS and Stiffness Matrix A	17
IV. Motivation	22
V. The Project	29
VI. Results: Manipulation of MD_GRAPE	31
VII. Results: Model Visualization	43
VIII. Conclusions & Future Work	48
IX. Appendix	50
X. Resources	72



Acknowledgements

As with any project, the end product is always the result of the efforts of many. To recognize each and every one of these individuals would be impossible, but I will do my best to mention some of the more influential ones.

I would like to begin by thanking my parents, Leo and Linda Simbeck, for their effort, time, patience, energy, and love that was willingly given to me so that I may have the chance at an education in physics at Allegheny College. I only hope that my achievements at this school can begin to repay them for everything that they have already given to me.

Secondly, I would like to thank all the players of the Allegheny College men's varsity basketball team. I have always believed that a successful life is a balanced one, and it was through these teammates that I was able to not only balance school and basketball, but also maintain a steady and healthy work ethic at Allegheny College. Prish the chips.

Next, I must thank all of my fellow physic majors for not only their friendship, but also all of the good times that we have spent together over the past four years. I always knew majoring in physics would not be easy, but I could not even begin to imagine how much more difficult it would have been without such good friends. I offer a similar resounding thank you to all my other friends outside the physics department as well.

Finally, I would like to thank the entire faculty of the Physics Department of Allegheny College, specifically Professor Gary Bedrosian and Professor Jamie Lombardi, for everything that went into the final product of my senior comprehensive project. It is professors like these and classes like theirs that answer the question I have been posed many times over the last four years: "Why the hell are you a physics major?" Again, thank you to everyone and please know that all of you have played a direct part in everything and anything in my life that one could consider a success. Thank you.

I. Introduction

Computational electromagnetics (CEM), a still relatively young discipline, is an area of physics warranting increased research as well as rapid growth because of modern technology's growing demands for greater electromagnetic capabilities and speed. CEM makes use of approximations to Maxwell's equations to model interactions between electromagnetic fields and objects, as well as the surrounding environment. Not only has CEM been a major factor in the transition towards computational models, away from the previous, more costly method of building and testing, but it has also impacted a number of current scientific problems. This computational method for exploring electromagnetic interactions has influenced everything from the optimization of radar cross sections for stealth devices to the design of microwave ovens. The challenge of CEM keeping pace with such abrupt demands in advancement though is twofold: cost and implementation.

The MD-GRAPE (Gravity Pipe) board, which henceforth will be referred to as the GRAPE board, is an accelerator module that can rapidly calculate all of the interatomic forces between millions of particles within a given simulation. The power of the GRAPE board is its ability to efficiently and continuously execute repetitive operations for large data sets: for example, determining the Coulomb or Gravitational force between many pairs of charges. Note that these types of numerical computations could be performed on an ordinary computer, but such methods are rather time consuming. This type of capability becomes especially appealing by realizing that many real-world physical problems can be approximated as an array of particles. Although the accelerator board can technically handle any N-body simulation, its employment has yet to be introduced into CEM. The number of similarities that exist between gravity and

electromagnetics makes adapting such a device to meet CEM calculation requirements rather appealing. Programming a GRAPE board to double as a CEM accelerator would then provide an elegant solution towards inexpensive further implementation of CEM.

II. Background Theory: CEM

Computational electromagnetics seeks to solve Maxwell's equations

$$\begin{aligned}\nabla \cdot \vec{E} &= \rho / \epsilon_0 & \nabla \times \vec{E} &= -\partial \vec{B} / \partial t \\ \nabla \cdot \vec{B} &= 0 & \nabla \times \vec{B} &= \mu_0 \vec{J} + \mu_0 \epsilon_0 \partial \vec{E} / \partial t\end{aligned}\tag{2.1}$$

under the focus of real world problems and applications. Although (2.1) is correct in free space, and probably the form the reader is most familiar with, this set of expressions in the presence of matter is

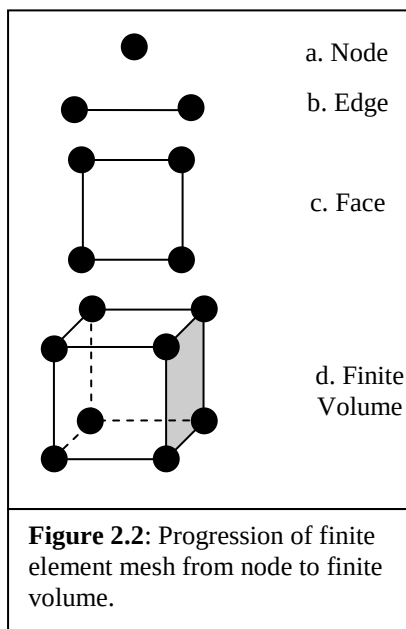
$$\begin{aligned}\nabla \cdot \vec{D} &= \rho & \nabla \times \vec{H} &= \vec{J} + \partial \vec{D} / \partial t \\ \nabla \cdot \vec{B} &= 0 & \nabla \times \vec{E} &= -\partial \vec{B} / \partial t\end{aligned}\tag{2.2}$$

to take into account material properties. The types of problems encountered in CEM, even in their static time-independent form, can be rather complex; not to mention the added difficulty of adapting mathematical solutions to physical reality. It is the purpose of CEM then to develop effective and efficient computational means to address (2.2), which it already has a number of variations for doing.

Within CEM there are a number of different methods for solving Maxwell's equations, depending upon the parameters of the model under inspection. The three major analyses of the electromagnetic equations are the following: finite difference method (FD), finite element method (FEM), and method of moments (MoM). This project will focus on the implementation of the finite element method for use in conjunction with the GRAPE board.

FEM is a standard tool for solving differential equations in a number of disciplines. This scheme seeks to solve an equation by approximating a continuous quantity as a set of quantities at discrete points. It employs unstructured grids, commonly referred to as meshes (Figure 2.1 on Table of Contents page), consisting of various

shapes of varying order and dimension. The mesh itself is built up from lower order dimensional objects until the degree of detail desired is achieved. It begins with simple nodes (Figure 2.2a), which are merely points in space where an unknown function is to be determined. From here, two nodes can be connected to form a one-dimensional edge (Figure 2.2b); edges are connected to make two-dimensional faces (Figure 2.2c); faces are connected to form three-dimensional finite volumes (Figure 2.2d). The nodes, edges, faces, and finite volumes together comprise the finite element mesh. Depending on the type of analysis, any one of these objects may be considered a finite element. Note the development of a mesh also empowers FEM to deal with complex geometries, making it



a favorite among numerous branches of engineering.

The mesh is a very useful conceptualization, but in order to move forward with numerical analysis of (2.2) some type of function will be necessary to describe the elements within it.

One of the first issues of handling Maxwell's equations is the problem of taking a field that is known at a finite set of points and interpolating that field to any other point the region of interest. In order

to make the leap from a simple mesh to a strategy for resolving Maxwell's equations, FEM employs an interpolation scheme of linear combinations of shape functions. Shape functions are merely the nodal expressions employed to interpolate a field from a finite set of points where its value is already known, to all other points in question. For example, consider some region of interest that has already been decomposed into an

appropriate mesh and at each node i , the value of the field f_i is known. Associated with each node then will be a shape function P_i and the total field f at a point x, y, z is then

$$f(x, y, z) = \sum_{i=1}^N f_i P_i(x, y, z) \quad (2.3)$$

The linear combination of shape functions must provide a numerical solution with the following properties: the interpolated field must return the nodal value of the field when evaluated at the node, the interpolated field must be continuous between adjacent finite elements, and the interpolated field must be linear in the nodal values. A necessary but not sufficient requirement that satisfies the requirements is

$$P_i = \begin{cases} 1, & \text{at node } i \\ 0, & \text{all other nodes \& any element not containing node } i \end{cases} \quad (2.4)$$

Equation (2.4) obviously satisfies the first condition for shape functions, but the other two properties are not as apparent. The second property means that the shape functions that describe a finite element must properly reduce to the shape functions of lower dimensional elements at the intersection of adjacent elements. For example, imagine two first order cubic finite elements that share one common face. The shape functions of the cubes at the interface must be such that they reduce to those of a two-dimensional square, which is the finite element at the intersection between these adjacent elements.

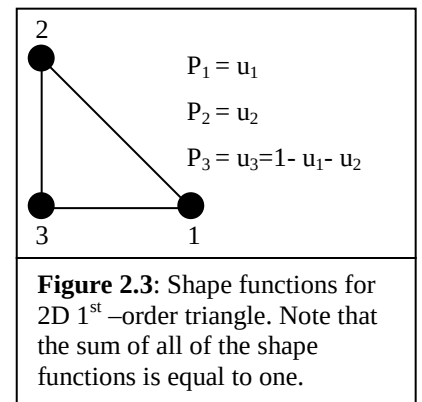
Furthermore the shape functions of the 2D square must in turn reduce to those of a 1D edge, which finally have to reduce to those of the nodes as well. All of this means that in the end the interpolated field found using the shape functions will be continuous between finite elements. The definition given in (2.4), along with the various shape function equations for different elements and orders, ensures this reduction process properly takes place. Finally, (2.4) also ensures that (2.3) is linear in the sense that solutions can be

constructed from either a linear combination of interpolated fields or by using (2.3). For example, if f_1 and f_2 were both interpolated fields whose solutions are known then a linear combination of them f_3 could be written as

$$f_3 = Af_1 + Bf_2 \quad (2.5)$$

where A and B are arbitrary coefficients. It would have also sufficed to apply (2.3) to solve for f_3 and the same value would have been found because of how the shape functions were previously defined in (2.4).

The nodal dependence of (2.3) implies a need to identify which nodes belong to a specific element, on both a local and global scale, via some organized numbering system. Within each element, nodes have both global and local numbers, as well as global and local coordinates. The purpose of the global nodal numbers is for defining the position of nodes relative to every other node in the mesh. Note that the global numbers of the nodes are not unique, rather an arbitrary result of the mesh generation. Likewise, the local nodal numbering scheme defines the position of a node with respect to the other nodes within that element. Local node numbers are only unique to the element they are within. The shape functions themselves are given in terms of the local coordinates u_1, u_2, u_3 (Figure 2.3). Shape functions can be mapped from local to global x, y, z coordinates using the same interpolation technique seen in (2.3). The inverse process, mapping from global to local coordinates, is also possible, but more difficult. Note that all elements of each type (e.g. triangle)



are identical in local coordinate space, while the mapping of local to global coordinates determines the physical shape of each individual element. Finally, mathematical

expressions also exist to define how derivatives and integrals involving shape functions are handled in FEM, but only their known existence, not full detail, is necessary at this point and they shall therefore be introduced as needed.

Furthermore, instead of derivatives, FEM attempts to resolve (2.2) by formulating its matrix equivalent from integral expressions. To better comprehend how to make the transition from the differential form of Maxwell's equations to integral expressions that can be formulated as matrix equations, consider the following example.

Imagine attempting to calculate the static, three-dimensional magnetic field $\vec{H}$ for a linear medium via a scalar potential formulation. Recall the mathematical relationship between the field $\vec{H}$, the magnetic flux density $\vec{B}$, and the magnetization of a material $\vec{M}$

$$\vec{B} = \mu_0 \vec{H} + \mu_0 \vec{M} \quad (2.6)$$

where μ_0 is the magnetic permeability of free space. Note that a linear medium is one in which the magnetization is directly proportional to the magnetic field

$$\vec{M} = \chi_m \vec{H} \quad (2.7)$$

where χ_m is a dimensionless quantity that varies between substances. Now there are actually two possible sources of magnetization: the permanent magnetization $\vec{M}_0$, which is inherent to the medium in question, and induced magnetization $\vec{M}_{ind}$, which is a result of an applied external magnetic field. The total magnetization can then be written as the sum

$$\vec{M} = \vec{M}_{ind} + \vec{M}_0 = \left(\frac{\mu - \mu_0}{\mu_0} \right) \vec{H} + \vec{M}_0 \quad (2.8)$$

where the value μ , known as the magnetic permeability of the material, has the same dimensions as μ_0 and is given by

$$\mu = \mu_0(1 + \chi_m) \quad (2.9)$$

Note (2.8) was formulated by using (2.7) and the definition of χ_m from (2.9) to write $\vec{M}_{ind}$ in terms of $\vec{H}$. When no matter is present to magnetize, as is the case in a vacuum, χ_m vanishes and the permeability is simply μ_0 , hence its name. Although μ is not constant in real materials (it generally decreases as $\vec{H}$ increases), for the purpose of this analysis it will be assumed to be constant for simplification.

Now with the quick background in electrodynamics under the belt, the focus can shift towards actually solving Maxwell's equations. Since the case under consideration is static, the magnetic and electric fields become decoupled and only two of Maxwell's equations need be satisfied

$$\nabla \cdot \vec{B} = 0 \quad (2.10a)$$

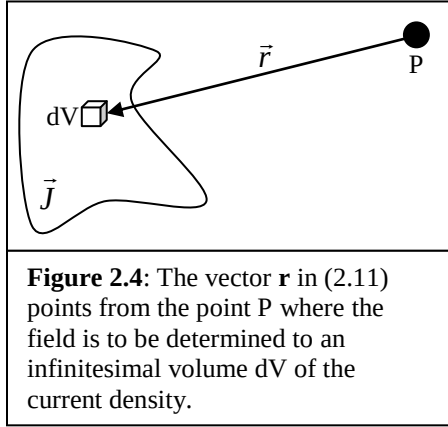
$$\nabla \times \vec{H} = \vec{J} \quad (2.10b)$$

where $\vec{J}$ is the current density. While there are a number of numerical methods for dealing with (2.10a,b), it would be convenient to use one familiar to introductory electromagnetics.

If only the free space solution ($\mu = \mu_0$) were examined, then the current density could be broken into infinitesimal segments and integrated using a form of the Biot-Savart law

$$d\vec{H}_c = \frac{\hat{r} \times \vec{J}}{4\pi r^2} \quad (2.11)$$

Note that in this form the vector $\vec{r}$ actually points from the spot where $\vec{H}$ is to be calculated to the place where the current density exists, the source point (Figure 2.4).



Integrating (2.11) over all space gives the free space

magnetic field $\vec{H}_c$

$$\vec{H}_c = \iiint \frac{\hat{r} \times \vec{J}}{4\pi r^2} dx dy dz \quad (2.12)$$

which is referred to as the “coil field” in CEM. This

allows the free space magnetic flux density $\vec{B}_c$ of the

“coil” to be written as

$$\vec{B}_c = \mu_0 \vec{H}_c \quad (2.13)$$

Note not only does (2.12) still satisfy (2.10b), but the first condition (2.10a) is covered by (2.13).¹ Therefore, expressions satisfying Maxwell’s equations do indeed exist when solving for the free space solution of the magnetic field. Having found $\vec{H}_c$, the next step is to solve for the magnetic field as a whole, not just its free space solution, by introducing a scalar potential.

The full field $\vec{H}$ can be written as a linear combination of its free space solution and a magnetic scalar potential Φ , similar to that of electrostatics,

$$\vec{H} = \vec{H}_c - \nabla \Phi \quad (2.14)$$

while still satisfying (2.10b) by the fact that the curl of the gradient of any function is zero by definition. Physically, the first term of (2.14) serves to pick up the magnetic field produced by a current, whereas the second term picks up the magnetization of any

¹ For the purposes of this analysis this will be taken as true, for the proof is rather lengthy and detailed. The reader is directed towards an introductory book on electromagnetics if interested in the full argument.

materials present. One must not forget about the divergence of the magnetic flux density $\vec{B}$ though. Substituting (2.6) into (2.10a) and plugging in for $\vec{H}$ from (2.14) yields

$$\nabla \cdot \vec{B} = \nabla \cdot (\mu \vec{H} + \mu_0 \vec{M}_0) = \nabla \cdot (\mu \vec{H}_c - \mu \nabla \Phi + \mu_0 \vec{M}_0) = 0 \quad (2.15)$$

which can be arranged to form a second-order partial differential equation for Φ

$$\nabla \cdot (-\mu \nabla \Phi) = \nabla \cdot (-\mu \vec{H}_c - \mu_0 \vec{M}_0) \quad (2.16)$$

The inclusion of the negatives will become clear in a moment. This equation is correct, but remember that in free space the right hand side should equal zero exactly. Obviously though $\vec{H}_c$ cannot be calculated exactly and small errors can upset the free space solution of (2.16). This conflict is easily remedied by adding a form of zero, the divergence of (2.13),

$$\nabla \cdot (-\mu \nabla \Phi) = \nabla \cdot [-(\mu - \mu_0) \vec{H}_c - \mu_0 \vec{M}_0] \quad (2.17)$$

without altering the equation. The right hand side of (2.17) is now always exactly zero in free space, independent of numerical errors in $\vec{H}_c$. There exists yet another issue with the formulation of (2.17) though; it cannot be solved for general shapes, as well as combinations of different media. So a standstill has been reached: an equation exists for solving for the magnetic scalar potential, but a numerical approximation method will be necessary to generate a solution to (2.17). The approximation will have to not only be efficient, but also numerically stable to ensure nontrivial results. A tried and true method that fits the previous description is FEM.

Continuing with the magnetic scalar potential model, the potential itself can be written as a summation of shape functions

$$\Phi(x, y, z) = \sum_{j=1}^N \Phi_j P_j(x, y, z) \quad (2.18)$$

since from (2.3) it is noted that any function can be written as such, provided the nodal values of the function are known and appropriate shape functions are used. In (2.18) P_j are once again the shape functions, Φ_j are the nodal values of the magnetic scalar potential, and the index j runs over all nodes. Use of (2.18) requires that (2.17) somehow be manipulated to give equations for the numerical values of Φ_j . This can be accomplished via Galerkin's method, which is a weighted residual technique that converts continuous operators, such as differentials, into discrete ones.²

Given the setup of the problem at hand, Galerkin's method can be applied directly. Slightly rearranging the differential equation (2.17), multiplying it by an arbitrary weighting function w_i , and integrating over all space yields

$$\iiint w_i \left(\nabla \cdot (-\mu \nabla \Phi) - \nabla \cdot [-(\mu - \mu_0) \vec{H}_c - \mu_0 \vec{M}_0] \right) dx dy dz = 0 \quad (2.19)$$

The trial function for this case is Φ_j since it can be written as a linear combination of weighting functions, which are naturally chosen to be the shape functions P_i . The clever selection of weighting functions to satisfy (2.19) is where the true genius of the Galerkin method is seen. Equation (2.19) now becomes

$$\iiint P_i \left(\nabla \cdot \left(-\mu \sum_{j=1}^N \Phi_j \nabla P_j \right) - \nabla \cdot [-(\mu - \mu_0) \vec{H}_c - \mu_0 \vec{M}_0] \right) dx dy dz = 0 \quad (2.20)$$

which must be satisfied for every value of j from 1 up to N , which is taken as the total number of nodes in the analysis. Note that since Φ_j is assumed to be a known constant at

² Bondeson, Anders, Pär Ingelström, and Thomas Rylander. Computational Electromagnetics. New York, NY: Springer Science+Business Media, Inc., 2005.

the nodal values, the gradient operator can be brought inside the summation and allowed to operate on the shape functions only.

To get (2.20) into its final matrix formulation, recall the following identity from introductory vector analysis

$$\nabla \cdot (g \vec{F}) = g \nabla \cdot \vec{F} + \nabla g \cdot \vec{F} \quad (2.21a)$$

or

$$g \nabla \cdot \vec{F} = -\nabla g \cdot \vec{F} + \nabla \cdot (g \vec{F}) \quad (2.21b)$$

and Gauss's theorem

$$\iiint (\nabla \cdot \vec{F}) dx dy dz = \oint \vec{F} \cdot d\vec{s} \quad (2.22)$$

Applying (2.21b) to (2.20) gives

$$\iiint \left(\nabla P_i \cdot \left(\mu \sum_{j=1}^N \Phi_j \nabla P_j \right) - \nabla P_i \cdot [(\mu - \mu_0) \vec{H}_c + \mu_0 \vec{M}_0] - \nabla \cdot \left\{ P_i \left[\left(\mu \sum_{j=1}^N \Phi_j \nabla P_j \right) - ((\mu - \mu_0) \vec{H}_c + \mu_0 \vec{M}_0) \right] \right\} \right) dx dy dz = 0 \quad (2.23)$$

and applying Gauss's theorem yields

$$\begin{aligned} & \iiint \left(\nabla P_i \cdot \left(\mu \sum_{j=1}^N \Phi_j \nabla P_j \right) - \nabla P_i \cdot [(\mu - \mu_0) \vec{H}_c + \mu_0 \vec{M}_0] \right) dx dy dz - \\ & \oint \left\{ P_i \left[\left(\mu \sum_{j=1}^N \Phi_j \nabla P_j \right) - ((\mu - \mu_0) \vec{H}_c + \mu_0 \vec{M}_0) \right] \right\} \cdot d\vec{s} = 0 \end{aligned} \quad (2.24)$$

which contains the appropriate volume and closed surface integral terms. Equation (2.24) is nothing short of intimidating, but can be reduced by assuming that the model's surface elements are essentially in free space; therefore $\mu = \mu_0$ and $\vec{M}_0 = 0$, which causes part of the closed surface integral in (2.24) to drop off

$$\begin{aligned} & \iiint \left(\nabla P_i \cdot \left(\mu \sum_{j=1}^N \Phi_j \nabla P_j \right) - \nabla P_i \cdot \left[(\mu - \mu_0) \vec{H}_c + \mu_0 \vec{M}_0 \right] \right) dx dy dz - \\ & \oint \left\{ P_i \left[\left(\mu \sum_{j=1}^N \Phi_j \nabla P_j \right) \right] \right\} \cdot d\vec{s} = 0 \end{aligned} \quad (2.25)$$

To further simplify the case at hand, it can be assumed that $\Phi = 0$ on the outer boundary.³

The surface integral then vanishes and (2.25) becomes

$$\iiint \left(\nabla P_i \cdot \left(\mu \sum_{j=1}^N \Phi_j \nabla P_j \right) - \nabla P_i \cdot \left[(\mu - \mu_0) \vec{H}_c + \mu_0 \vec{M}_0 \right] \right) dx dy dz = 0 \quad (2.26)$$

which, after simple rearranging, appears as

$$\begin{aligned} & \iiint \left(\nabla P_i \cdot \left(\mu \sum_{j=1}^N \Phi_j \nabla P_j \right) \right) dx dy dz = \iiint \left(\nabla P_i \cdot \left[(\mu - \mu_0) \vec{H}_c + \mu_0 \vec{M}_0 \right] \right) dx dy dz \\ & \quad (2.27) \end{aligned}$$

Finally, the result (2.27) is the equivalent of a matrix equation for the unknown values of Φ_j , which is identical to the unknown coefficients of Galerkin's method. The rows of the matrix equation are given by the index i , whereas the columns are given by the j index. Also the summation within (2.27) equates to nothing more than matrix multiplication; therefore it can be rewritten as

$$[A][\Phi] = [R] \quad (2.28)$$

where $[A]$ is an $N \times N$ matrix, commonly referred to as the "stiffness matrix," and both $[\Phi]$ and $[R]$ are $N \times 1$ column matrices. Comparing (2.28) to (2.27) shows that the matrix elements of $[A]$ are given by the integral

$$A_{i,j} = \iiint \left(\nabla P_i \cdot (\mu \nabla P_j) \right) dx dy dz \quad (2.29)$$

³ Although for the final testing of this type of project this assumption will not hold, it is sufficient for the work done up to this point.

Not only is it evident that $A_{i,j}$ is symmetric from (2.29), but also that it is extremely sparse since the contribution from the integrand will only not be zero when P_i and P_j belong to nodes sharing at least one common element. This is due to how the shape functions were previously defined (see Section II, p.3). For example, in a typical finite element problem involving a second order mesh, in a given row of matrix $[A]$, only a few dozen column entries are nonzero.⁴ Furthermore, assuming roughly 100,000 nodes in a typical problem, there are less than 10^7 non-zero matrix elements out of all 10^{10} elements within the matrix.⁵ The right-hand side (RHS) of (2.28) is given by

$$R_i = \iiint (\nabla P_i \cdot [(\mu - \mu_0)\vec{H}_c + \mu_0\vec{M}_0]) dx dy dz \quad (2.30)$$

which obviously vanishes in regions of free space, whereas the matrix $[A]$ does not.

Finally, realize that (2.28) is singular in nature. This means that a unique solution to (2.28) cannot be found; rather an infinite amount of solutions exist. More specifically, the presence of the gradient function within (2.28) ensures that the addition of a constant to any solution satisfying the matrix equation, will itself also be a solution. Although (2.28) is what this discussion set out to formulate, developing a program that actually calculates these type of matrix equations is a another issue.

⁴ Bedrosian, G. "Scalar Potential Formulation for Magnetostatic Finite Element." Allegheny College, Meadville, PA. October 2008.

⁵ Bedrosian, Scalar Potential 6

III. Background Theory: FEM Analysis of RHS and Stiffness Matrix A

It has been proposed that (2.27) is equivalent to a matrix equation but now a more in-depth explanation of this concept is necessary in order to understand how to develop an algorithm implementing such a calculation. Before this can be done though, some mathematical features of FEM, specifically Jacobians and integrals, will also need to be explored in greater detail.

The Jacobian is an important concept in FEM for two reasons: coordinate mapping and integration. More specifically, the inverse of the Jacobian is used to map from global coordinates to local coordinates, and the determinant of the Jacobian acts as volume conversion factor for integration in three dimensions. The Jacobian matrix is defined as

$$J = \begin{bmatrix} \frac{\partial x}{\partial u_1} & \frac{\partial x}{\partial u_2} & \frac{\partial x}{\partial u_3} \\ \frac{\partial y}{\partial u_1} & \frac{\partial y}{\partial u_2} & \frac{\partial y}{\partial u_3} \\ \frac{\partial z}{\partial u_1} & \frac{\partial z}{\partial u_2} & \frac{\partial z}{\partial u_3} \end{bmatrix} \quad (3.1)$$

and its inverse appears as

$$J^{-1} = \begin{bmatrix} \frac{\partial u_1}{\partial x} & \frac{\partial u_1}{\partial y} & \frac{\partial u_1}{\partial z} \\ \frac{\partial u_2}{\partial x} & \frac{\partial u_2}{\partial y} & \frac{\partial u_2}{\partial z} \\ \frac{\partial u_3}{\partial x} & \frac{\partial u_3}{\partial y} & \frac{\partial u_3}{\partial z} \end{bmatrix} \quad (3.2)$$

where u_1, u_2, u_3 are the local coordinates and x, y, z are the global coordinates. The components of the Jacobian are generally determined at a given location using the global

coordinates of the local nodes and the partial derivatives of the shape functions; inverting the matrix then gives J^{-1} . For example, consider attempting to calculate the Jacobian of a single first-order triangular finite element (see Figure 2.3), whose global nodal coordinate data is given by Table 3.1.⁶

Table 3.1: Global nodal coordinates for a 1 st -order 2D triangle. Remember that the index j is reserved for designating global nodal values.		
Global Node Number	X_j (m)	Y_j (m)
1	0	0
2	12	6
3	8	8

First, equations need to be developed for mapping from local to global coordinates, which will provide expressions that can be differentiated with respect to the local coordinates u_1, u_2, u_3 . As was mentioned earlier, this mapping procedure is similar to equation (2.3)

$$\begin{aligned}
 x &= \sum_{j=1}^N x_j P_j(u_1, u_2, u_3) \\
 y &= \sum_{j=1}^N y_j P_j(u_1, u_2, u_3) \\
 z &= \sum_{j=1}^N z_j P_j(u_1, u_2, u_3)
 \end{aligned} \tag{3.3}$$

where all three summations are over all nodes N . The data from Table 3.1 and the shape functions defined in Figure 2.3 gives

⁶ Bedrosian, G. “Note on Shape Functions for Finite Element Analysis.” Allegheny College, Meadville, PA. July 2008.

$$\begin{aligned}
x &= \sum_{j=1}^N x_j P_j(u_1, u_2, u_3) = x_1 P_1 + x_2 P_2 + x_3 P_3 = 0u_1 + 12u_2 + 8u_3 = 4(u_2 - 2u_1 + 2) \\
y &= \sum_{j=1}^N y_j P_j(u_1, u_2, u_3) = y_1 P_1 + y_2 P_2 + y_3 P_3 = 0u_1 + 6u_2 + 8u_3 = -2(u_2 + 4u_1 - 4) \\
z &= \sum_{j=1}^N z_j P_j(u_1, u_2, u_3) = z_1 P_1 + z_2 P_2 + z_3 P_3 = 0
\end{aligned}
\tag{3.4}$$

With the above expressions, derivatives with respect to u_1, u_2, u_3 are effortless

$$\begin{aligned}
\frac{\partial x}{\partial u_1} &= \frac{\partial}{\partial u_1} [4(u_2 - 2u_1 + 2)] = -8 \\
\frac{\partial x}{\partial u_2} &= \frac{\partial}{\partial u_2} [4(u_2 - 2u_1 + 2)] = 4 \\
\frac{\partial y}{\partial u_1} &= \frac{\partial}{\partial u_1} [-2(u_2 + 4u_1 - 4)] = -8 \\
\frac{\partial y}{\partial u_2} &= \frac{\partial}{\partial u_2} [-2(u_2 + 4u_1 - 4)] = -2
\end{aligned}
\tag{3.5}$$

and the Jacobian can be easily written as

$$J = \begin{bmatrix} \frac{\partial x}{\partial u_1} & \frac{\partial x}{\partial u_2} \\ \frac{\partial y}{\partial u_1} & \frac{\partial y}{\partial u_2} \end{bmatrix} = \begin{bmatrix} -8 & 4 \\ -8 & -2 \end{bmatrix}
\tag{3.6}$$

Note that although the Jacobian for a first-order triangular element is constant, this will generally not be true for other types and orders of finite elements. From the previous example though, one can readily see how a computer program could be designed to quickly perform such repetitive calculations for elements of any type and any order.

The Jacobian, or rather its determinant, is also linked to the other important FEM mathematical feature: integration. Within FEM, integrals are approximated using the technique of Gaussian integration. To get a feel for this process, assume that f is a

quantity one seeks to integrate over a single element and that f can be determined at any location inside this element. To begin, the determinant of the Jacobian is used to transform the integral from global to local coordinates

$$\int_{\text{element}} f(x, y, z) dx dy dz = \int_{\text{element}} f(u_1, u_2, u_3) |J| du_1 du_2 du_3 \quad (3.7)$$

Next, the Gaussian integration technique can be used to approximate (3.7) as a summation over a set of Gaussian integration points

$$\int_{\text{element}} f(u_1, u_2, u_3) |J| du_1 du_2 du_3 \approx \sum_{k=1}^L w_k f[(u_1, u_2, u_3)_k] |J| \quad (3.8)$$

whose locations and weights have been predetermined to integrate up to a given order polynomial exactly.⁷ In (3.8), L is the number of integration points, $(u_1, u_2, u_3)_k$ is the location of integration point k in local coordinates, and w_k is the weight of integration point k . In practice, Gaussian integration points are selected to be within the interior of an element in order to avoid possible singularities at the boundaries. Also, for this project, the physical correspondence of the product of the determinant of the Jacobian $|J|$ and the Gaussian weighting factor w_k can be taken as a local to global volume transformation for an infinitesimal local volume $du_1 du_2 du_3$. Reevaluating the previous statement with (3.8) in mind shows that the sum of all the Gaussian integration weights of an element should be equal to the local volume of that element. Finally, note that the process outlined above can be readily adapted to one or two-dimensional cases, as well as multiple elements.

The understanding of the FEM mathematics of Jacobians and integration now allows for a much clearer interpretation of the RHS and stiffness expressions as matrix equations. Recall the equation for the RHS matrix introduced in Section II

⁷ Bedrosian, [Shape Functions](#) 6

$$R_i = \iiint (\nabla P_i \cdot [(\mu - \mu_0)\vec{H}_c + \mu_0\vec{M}_0]) dx dy dz \quad (3.9)$$

From the above discussion (3.9) may be rewritten as

$$R_i = \sum_{k=1}^L \left\{ (\nabla P_{ik} \cdot [(\mu - \mu_0)\vec{H}_{c,ik} + \mu_0\vec{M}_{0,ik}]) w_{ik} |J_{ik}| \right\} \quad (3.10)$$

where the summation is over all Gaussian integration points k . What equation (3.10) says is that by looping over the L Gaussian integration points associated with each individual node i of the FEM analysis, one can construct the N (total number of nodes) components of the RHS matrix. Likewise, the stiffness matrix equation

$$A_{i,j} = \iiint (\nabla P_i \cdot (\mu \nabla P_j)) dx dy dz \quad (3.11)$$

may be rewritten as

$$A_{i,j} = \sum_{k=1}^L [\nabla P_{ik} \cdot (\mu \nabla P_{jk})] w_{ijk} |J_{ijk}| \quad (3.12)$$

where once again the k index runs over all Gaussian integration points L . Equation (3.12) states that summing over all Gaussian integration points associated with nodes i and j recovers the i^{th} row and j^{th} column of the stiffness matrix A .

Equations (3.10) and (3.12) now appear more suitable for a computer algorithm, especially since the dot products can be broken up into their individual x,y,z components and because the summation naturally lends itself to a do loop in FORTRAN.

Unfortunately though, equations (3.10) and (3.12) do not let on to the complexity involved with actually calculating $[A]$ and $[R]$. In practice, it is necessary to construct both matrices on a local scale first and then use these to build up each matrix's global counterpart. The process of building both of the final, global matrices will be discussed in conjunction with the programming used to derive them, which is outlined in Section VI.

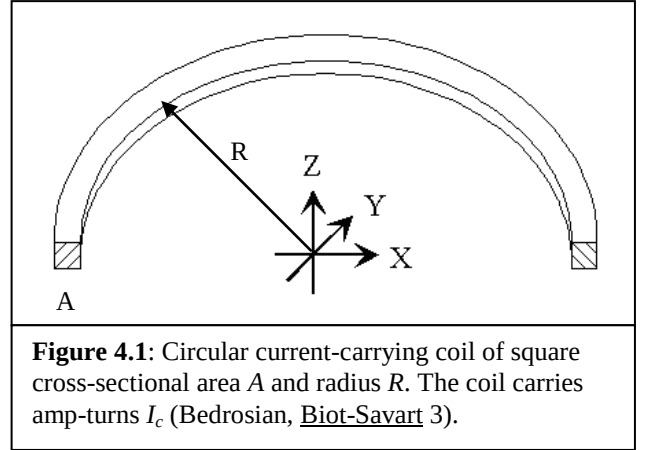
IV. Motivation

The GRAPE accelerator board has the potential to expedite calculations for any N-body problem, but its abilities have yet to be tested for the case of electromagnetism. A simple example though, will showcase not only how the board can be used for basic electromagnetic calculations but also highlight its time saving capabilities.

From the GRAPE board's user manual, the equation used for calculating Coulomb forces appears as

$$\vec{F}_i = q_i \sum_{j=1}^N \frac{q_j}{r_{ij}^3} \vec{r}_{ij} \quad (4.1)^8$$

Note that this formula is nothing more than the Coulomb force acting on particle i due to its interaction with all other particles j a distance r_{ij} away.⁹ The only minor change from the standard expression is the substitution for the unit vector $\hat{r}_{ij} = \vec{r}_{ij}/r_{ij}$. A first guess at manipulating the GRAPE board



would be to simply, yet cleverly, rearrange an electromagnetic equation similar to (4.1). Although this initial idea may sound rather naïve, it will actually prove to be quite beneficial in justifying moving forward with this project.

For this proof of concept, consider a coil containing I_c ampere-turns with an average radius R of 1.00m and cross sectional area A of 0.01m^2 (Figure 4.1). The coil is centered at the origin such that the magnetic field at the center of the coil aligns with the

⁸ MD-GRAPE Board User Manual

⁹ For the descriptions of the functions, parameters, and calls necessary to implement such a calculation the reader is directed to Section XI, Appendix B.

z-axis. Choosing the magnetic flux density at the center of the coil to be 1.00T, I_c can be approximated from

$$B_0 = \frac{\mu_0 I_c}{2R} \Rightarrow I_c = 1.6 \times 10^6 \text{ amp-turns} \quad (4.2)$$

This in turn gives the current density, which is found by dividing the current I_c by the given cross-sectional area of the coil, as $J_c = 1.6 \times 10^8 \text{ amp/m}^2$. Furthermore, the current also immediately gives an analytical result for the coil field $\vec{H}_c$ if the current were concentrated in a single infinitesimally thin ring

$$\vec{H}_c = 800000 \text{ A/m } \hat{z} \quad (4.3)$$

It is then expected that our results from the approximation will be just under the analytic figure (4.3) since the real problem spreads the current out slightly above and below the $z = 0$ plane.

Suppose one wished to determine the coil field $\vec{H}_c$ in free space at any point along the z-axis from zero up to one meter. The starting point for such a problem is the basis of magnetostatics: the Biot-Savart law

$$\vec{H}_c = \iiint \frac{\hat{r} \times \vec{J}}{4\pi r^2} dx dy dz \quad (4.4)$$

which can be rewritten as

$$\vec{H}_c = \iiint \frac{\vec{r} \times \vec{J}}{4\pi r^3} dx dy dz \quad (4.5)$$

by substituting in for the unit vector $\hat{r}$. Note that the vector $\vec{r}$ has the same definition as was previously described (see Figure 2.4). From above, it is already known that the GRAPE board uses a function similar to the form of (4.5) for calculating the Coulomb

force between particles. The problem is that the accelerator board has no means of dealing with the cross product within (4.5). Therefore some clever algebraic manipulations of the Biot-Savart law will be necessary to make use of the GRAPE board.

Switching over to electrostatics, recall the equation for calculating the displacement field $\vec{D}$ from a charge density ρ

$$\vec{D} = \iiint \frac{-\rho \vec{r}}{4\pi r^3} dx dy dz \quad (4.6)$$

which is in a form already directly supported by the GRAPE board. Comparing (4.6) and (4.5) it is seen that by breaking up the cross product of (4.5) a parallel can be made. The Cartesian components of the coil field can be written as

$$H_x = \iiint \frac{r_y J_z - r_z J_y}{4\pi r^3} dx dy dz = \iiint \frac{r_y J_z}{4\pi r^3} dx dy dz - \iiint \frac{r_z J_y}{4\pi r^3} dx dy dz \quad (4.7)$$

$$H_y = \iiint \frac{r_z J_x - r_x J_z}{4\pi r^3} dx dy dz = \iiint \frac{r_z J_x}{4\pi r^3} dx dy dz - \iiint \frac{r_x J_z}{4\pi r^3} dx dy dz \quad (4.8)$$

$$H_z = \iiint \frac{r_x J_y - r_y J_x}{4\pi r^3} dx dy dz = \iiint \frac{r_x J_y}{4\pi r^3} dx dy dz - \iiint \frac{r_y J_x}{4\pi r^3} dx dy dz \quad (4.9)$$

Defining $J_x = -p_1$, $J_y = -p_2$, $J_z = -p_3$; and letting

$$\vec{D}_1 = \iiint \frac{-\rho_1 \vec{r}}{4\pi r^3} dx dy dz \quad (4.10)$$

$$\vec{D}_2 = \iiint \frac{-\rho_2 \vec{r}}{4\pi r^3} dx dy dz \quad (4.11)$$

$$\vec{D}_3 = \iiint \frac{-\rho_3 \vec{r}}{4\pi r^3} dx dy dz \quad (4.12)$$

the components of $\vec{H}_c$ are then elegantly written as

$$H_x = D_{3,y} - D_{2,z} \quad (4.13)$$

$$H_y = D_{1,z} - D_{3,x} \quad (4.14)$$

$$H_z = D_{2,x} - D_{1,y} \quad (4.15)$$

Therefore the game will be to first send the GRAPE board the positions as well as the magnitudes of the currents, which it will interpret as the charges in a Coulomb force calculation. Next, the components of the “ D fields” can be found by passing to the accelerator board another, different set of points where the field is to be evaluated. After the GRAPE board has performed its part of the computation it should be merely a matter of reconstructing the components of $\vec{H}_c$ from $\vec{D}_1, \vec{D}_2, \vec{D}_3$ via (4.13-4.15) and possibly multiplying by an appropriate correction factor. The correction factor for this case should hopefully only be a division by 4π , since this is the only apparent difference between what the GRAPE board will generate and what is desired. As will be seen in moment though, a difference is usually never as simple as its appearance in physics. Theoretically then there exists a means of performing a simple electromagnetic calculation using the GRAPE board.

The process outlined in the previous parts of this section can be easily coded into a FORTRAN program, whose details will be left out since the programming was provided by Professor Gary Bedrosian. Furthermore the programming itself is not within

the scope of this project and therefore unnecessary to specify.¹⁰ To better comprehend how to call the GRAPE board within a program so it will work with CEM see the attached appendix (Section IX, Appendix A), which gives the programming codes used in the proof of motivation calculation.

The main code that was simplified and modified for use in conjunction with the GRAPE board was H3D, a program that determines the three-dimensional magnetic field $\vec{H}$, or in our case $\vec{H}_c$, for arbitrary coil shapes of various types. Multiple runs of the newly vamped H3D code with given sets of currents, current positions, and evaluation points (all provided by Professor Bedrosian) show that the GRAPE board results agree perfectly with those found without the accelerator board's assistance (Table 4.1) Note that Table 4.1 only compares the origin evaluation point for $\vec{H}_c$, but within the program the field at all the evaluation points was double checked. Also, the results of Table 4.1 are both just slightly under the analytic one, exactly as was expected.

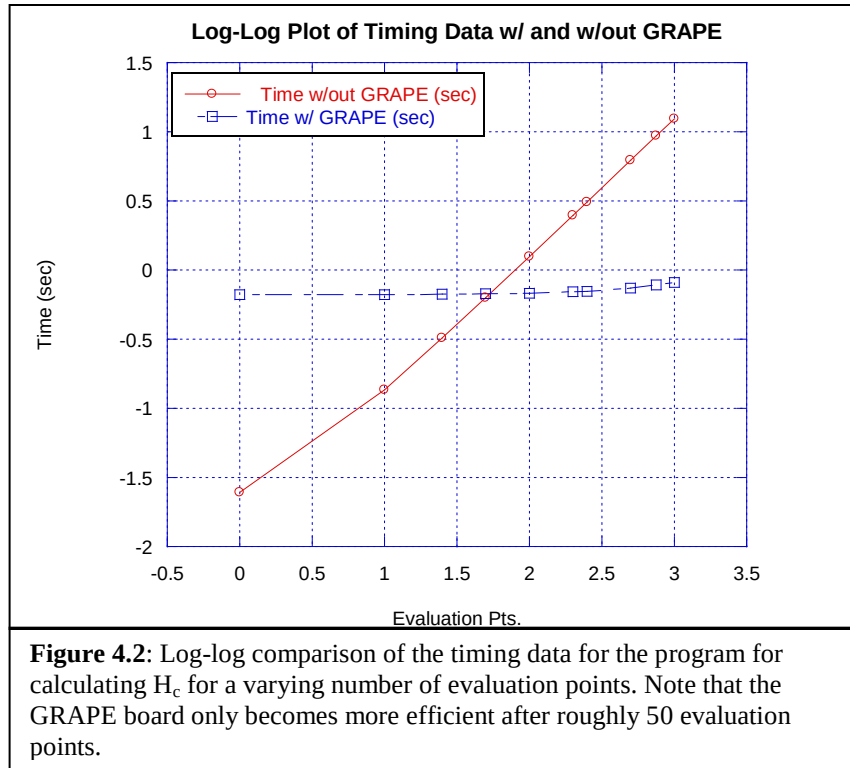
Not only do the calculations satisfactorily agree, but a timing of the two methods for determining $\vec{H}_c$, using standard FORTRAN timing techniques, reveals the true potential of the GRAPE board for CEM. It was found that an overall computation time decrease of a factor of roughly fifteen (Table 4.1) for a relatively small number of repetitions (10^2) is achieved using the accelerator board. These figures must be interpreted cautiously though, since in reality such timing benefits are only achieved if the problem contains a large enough number of evaluation points, greater than 50 to be precise (Figure 4.2). Otherwise the overhead of the board causes it to actually function

¹⁰ If this bothers the reader then he or she is directed towards Section IX, Appendix B where the list of resources gives the names of the programs implemented throughout the project, which Professor Bedrosian will be happy to discuss.

slower than the brute force method. Furthermore, even supplementing the GRAPE board-free calculation with standard FORTRAN optimization techniques failed to bring about a significant difference in the timing data. These figures were not found without complications and setbacks though.

Table 4.1: Comparison of $\vec{H}_c$ calculation at the origin and computational time taken for a sample run of the program. The analytic result is also included.				
	H_x (A/m)	H_y (A/m)	H_z (A/m)	Time (sec)
Without Grape	0.00	0.00	799664.54426	12.41349
With Grape	0.00	0.00	799664.61266	0.82351
Analytic Result for a Single Infinitesimally Thin Ring	0.00	0.00	800000.00000	-

As an interesting note, it was uncovered that the currents used in the calculation were actually too large for the GRAPE board to handle, resulting in what can best be described as an overflow error. The currents all had to be divided by a factor of 10^4 , which subsequently meant that the correction factor of the field from the board had to be multiplied by additional 10^4 to achieve the appropriate results. In the end though, this was only a minor hurdle in the overall procedure, but curious none the less. Such figures are the driving force behind the project outlined in Section V and warrant continued research of manipulating the GRAPE board for the purposes of CEM.



V. The Project

The goal of this senior project is to develop a cost effective method to modify an already existing piece of hardware to perform CEM. Such rapid technology is necessary given the large number of calculations within a standard CEM problem. This in turn will hopefully someday lead to the development of a comprehensive CEM software package, all of which can be run on the existing modified hardware, or even a simple graphics processing unit (GPU).

The similarities between the calculations performed by the GRAPE board and electromagnetics may give a false impression of the complexity of this project though. Note, that switching around a few constants and interpreting data differently will not be enough to achieve the above goals. Rather, an intensive comparison of the CEM algorithms with those of the GRAPE board will be necessary to understand where changes could be made to use such hardware for CEM. The project therefore naturally starts where the accelerator chip can assist CEM.

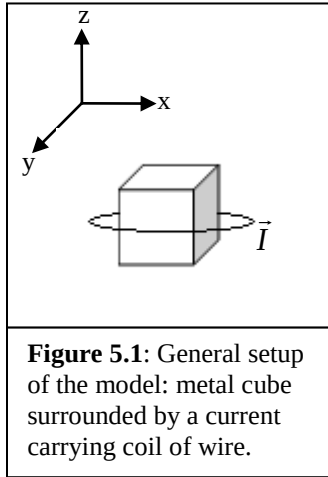
The three basic building blocks to any CEM problem involve pre-processing, the solving of matrices, and finally post-processing. Pre-processing involves building a simple model from which one can generate a matrix equation of the form

$$[A][x] = [R] \quad (5.1)$$

which has already been seen in Section II. Once such a matrix has been constructed, the second part of CEM works to solve these types of equations via computational methods. In the final step, post-processing, the numerical calculations derived from the algorithms are analyzed, which also usually involves some type of three-dimensional graphics display. All three fundamental steps of the basic CEM problem are rather

computationally intensive, especially when the repetition of such processes approaches the millions, and naturally lend themselves to a computational acceleration method.

More specifically, this project will focus on the pre-processing step for a general



magnetostatic case in order to determine the necessary modifications for use of the GRAPE board. Also, as a side project, some efforts will be put towards generating model visualizations with programs such as Mathematica and MatLab. The model will consist of a cube of an idealized ferromagnetic material, which will be called “iron” for short, surrounded by a wire coil (Figure 5.1), similar to the example

that was investigated in the previous section. The project will not only include the application of FEM in the development of the model, but also generating matrix equations identical to (5.1), all of which must be adapted for compatibility with the GRAPE board. All of the basic CEM programming code will be provided by Professor Gary Bedrosian, for this is not the purpose of this research; the current methods and algorithms of CEM to increase efficiency or reduce time requirements are already tried and true. The project has great potential for the CEM community, and may possibly one day be tailored to other basic electromagnetic problems (i.e. electrostatics or induction). Finally, there is great promise with this project, given the known connections between the GRAPE board’s current utilization and electromagnetics, but the apparent simplicity should not cloud the obvious complexity and hopefully eventual elegance.

VI. Results: Manipulation of MD_GRAPE¹¹

Besides manipulating the GRAPE board to perform basic electromagnetic problems, as presented in Section IV, this project also sought to implement the accelerator board for the construction of the matrix equation

$$[A][\Phi] = [R] \quad (6.1)$$

Although the work did not reach its full potential do to time constraints, important details about employing the GRAPE board to assist with the generation of both the stiffness matrix and RHS matrix were uncovered. The following outlines the steps that were taken to reach the current progress of the project.

The first step in any FEM analysis of a problem is to break the area of consideration into an appropriate finite element mesh. Recall that the model in question for the project is a cube of iron within a cube of air (see Figure 5.1). By making a simple alteration to the MakeFEM.f program, a $10 \times 10 \times 10$ mesh of first-order three-dimensional hexahedron (brick) finite elements of both air and iron could be generated. The MakeFEM.f program functions by building up finite elements from nodes. In other words, once the program establishes all the nodes it looks to connect them to create edges, which are in turn connected to generate faces, which are finally joined to define hexahedron finite elements. The major complication with MakeFEM.f is ensuring that the nodes/edges/faces are connected in the appropriate manner and direction. For instance, the first-order hexahedron elements that the model has been broken down into contain eight nodes, twelve edges, and six faces apiece. The MakeFEM.f program must be able to determine not only which eight nodes go into making this brick, but also how those nodes

¹¹ This section outlines the excerpt from the MakeFEM.f programming code included in Section IX, Appendix A.

need to be oriented so that the edges and faces all fit together properly.¹² Once the model has been decomposed into the correct mesh, the construction of the RHS and stiffness matrices can begin.

Beginning with the RHS matrix, as formulated in Section III

$$R_i = \sum_{k=1}^L \left\{ \left(\nabla P_{ik} \cdot \left[(\mu - \mu_0) \vec{H}_{c,ik} + \mu_0 \vec{M}_{0,ik} \right] \right) w_{ik} |J_{ik}| \right\} \quad (6.2)$$

one can see that the first key component in constructing the matrix is the Gaussian integration points and weights. Before moving too far into the process though, it should be noted that computing time/storage and programming effort can both be cut down on by taking advantage of the extreme symmetry present within the mesh. For instance, since the Gaussian integration points are given in terms of local coordinates, which will be the same from one element to the next (i.e. they are not unique between elements) they only need to be found once for one element. Also, since the brick elements are all first order cubes, it is expected that the weights associated with each integration point should be equal and, as discussed in Section III, should sum to the local volume of the cube. The Gaussian integration points and weights were found using the program GAUSS3D by selecting an initial order and number of integration points to be passed to the subroutine. Both points and weights were stored in one-dimensional arrays to be used later in the program.

After the Gaussian integration points were found, it was necessary to determine the shape functions, as well as their derivatives with respect to the local coordinates u_1, u_2, u_3 . This can again be done for only a single element because of the

¹² How one can turn the finite element mesh data from numbers on a computer into an actual visualization will be discussed in Section VII.

same line of reasoning presented in the previous paragraph. By passing the element type (number of faces), number of nodes per element, and already determined Gaussian integration points to the program SHAPE3D, the shape functions and their local coordinate derivatives were calculated. The shape functions and their respective derivatives were stored in two-dimensional arrays to be used later on in the program.

Next, it was necessary to distinguish the elements of iron from those of air, but this step was already performed in the MakeFEM.f program via a “flagging” system. Knowledge of which nodes were in air, which were in iron, and which lied on the interface between the two then allowed for storage of the global node numbers and their corresponding global coordinates, which were also generated by MakeFEM.f, on an element by element basis. A simple counter introduced at this stage will also allow for determining the number of iron and air elements. The global air/iron nodes were stored in two-dimensional arrays for future use in the program. When constructing the RHS matrix, distinguishing between iron and air elements is crucial because the RHS vanishes in free space, which in our case is simply taken as air (see Section II).

With the coordinates of the iron nodes and the local derivatives of the shape functions known, the subroutine DVJC3D within the program SHAPE3D was employed to calculate the derivatives of the shape functions with respect to the global x, y, z coordinates. The purpose of this part of the program is nothing more than to convert the derivatives from local coordinates to global coordinates so that they can be used in breaking up the gradient of the shape function present within (6.2). There will be a global shape function derivative associated with each node in an element, which are themselves associated with a Gaussian integration point. DVJC3D was also used to find

the determinate of the Jacobian corresponding to each element. Note that the high symmetry of the mesh should result in the determinant of the Jacobian for all elements being equal. Also, at this point one should double check to make sure that the sum of the product of the Jacobians and the integration weights of each iron element is equal to the global volume of that element. Furthermore, the sum of all the elements' volumes should be equal to the total, global volume of the iron cube (see Section III).

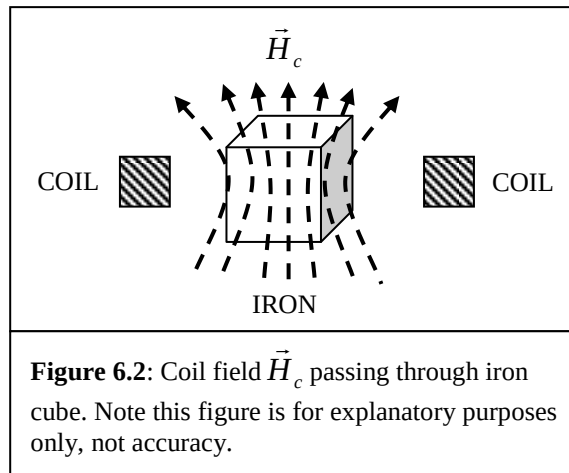
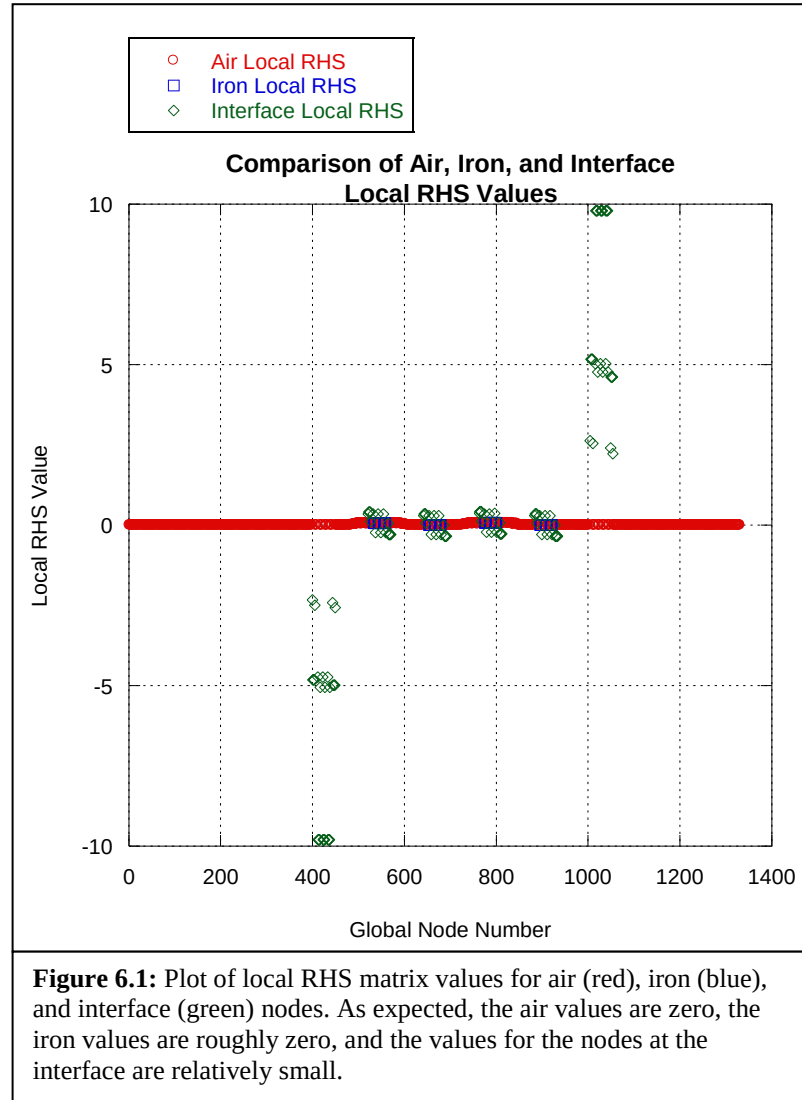
Within the above steps, it is also convenient to convert the Gaussian integration points from local to global coordinates and store them. This process is done in a manner similar to equation (3.3), which was introduced in the example from Section III. Whereas previous calculations have been done on a single element basis, the coordinate mapping had to be performed for all coordinates of all the nodes and stored as such.

Writing the global coordinates of the Gaussian integration points of the iron elements into a file then makes them accessible for calculation of the coil field $\vec{H}_c$ via the GRAPE board. This computation is identical to that which was performed in proving the motivation for this project (see Section IV). The only adjustment to the program that implements the GRAPE board to determine $\vec{H}_c$ was the file being read in. Once the coil field has been calculated, one is finally ready to take a first crack at constructing the RHS matrix.

A quick review of equation (6.2) shows that solving for [R] requires nothing more than summing over all the Gaussian integration points associated with each node. In FORTRAN, the equivalent of a summation is simply a do loop. The major dilemma though, is ensuring that each pass of the do loop introduces the correct information for the particular node the program happens to be summing over at that time. In other words,

the program must perform the RHS calculation for a particular node with the shape function derivatives, coil field components, integration weights, and Jacobian determinant specific to that one node. Luckily, the storage methods described above make such a task merely a matter of proper programming. A plot of all the RHS values, which distinguishes between nodes in air, nodes in iron, and nodes lying on the air/iron interface, is given in Figure 6.1. Note that the values in Figure 6.1 are for the case where $\vec{M}_0 = 0$ and $\mu = 1000\mu_0$. Again a flagging system was used to determine whether a certain node was in air, in iron, or on the interface between the two. For air, Figure 6.1 is easy to understand because here $\mu = \mu_0$ and $\vec{M}_0 = 0$, which causes (6.2) to go to zero. For nodes in iron, the values in Figure (6.1) are expected because of the gradient of the shape function present within (6.2). More specifically, the gradient of the shape function associated with a node will always points towards that particular node. Connecting this with the fact that in 3D any node is tied to eight elements, one can begin to visualize these vectors all approximately cancelling one another out, resulting in an RHS of approximately zero. Finally from Figure (6.1), note that the RHS for nodes at the interface is negative at the bottom of the cube and positive at the top, but all of the values are relatively small. This is expected because the RHS can also be viewed as a measure of the number of field lines penetrating the iron cube (Figure 6.2). From Figure 6.2 it is then evident that the highest values, in terms of magnitude, of the RHS for the interface nodes should be on the top and bottom of the cube. Furthermore, the magnitudes of the RHS for nodes at the interface are identical because of the model's symmetry and their signs are a result of the iron dipoles' response to the coil field; similar to a dielectric medium's response to an external electric field applied in the vertical direction. It will be

seen shortly though that the components of the [R] are not fully complete with the formulation given in (6.2).



The benefit of going through the above analysis with such detail is that the explanation of how to generate the stiffness matrix $[A]$ has now already been mostly introduced. Recall the formulation of the stiffness matrix from Section III

$$A_{i,j} = \sum_{k=i}^L [\nabla P_{ik} \cdot (\mu \nabla P_{jk})] w_{ijk} |J_{ijk}| \quad (6.3)$$

The derivatives of the shape functions with respect to the global coordinates, integration weights, and determinants of the Jacobians have all already been found and stored when the RHS matrix calculation was performed. It then seems relatively straightforward to calculate $[A]$ using a program that does just exactly what equation (6.3) says: sum over all the integration points associated with the nodes i and j , and in fact this is exactly how $[A]$ was found on a local level. Globally though, the sparse nature of $[A]$ makes it inefficient to store the components of the stiffness matrix that are just zero. To add even more to the complexity, the issue of the singularity of the matrix equation as a whole also still needs to be treated. Therefore what is sought is a method to distinguish between which elements of the stiffness matrix are necessarily zero and need not be stored, and which are nonzero and need be stored. If a row in $[A]$ corresponds to node i and a column corresponds to node j , that matrix element is necessarily zero if i and j share no finite elements. At the same time, the technique must also be able to somehow remove the singular nature of (6.1). Both of these conditions can actually be satisfied with the concepts of boundary conditions and degrees of freedom (dof).

Just like gravitational potential energy, the value of the magnetic potential Φ is only dependent upon differences; therefore one is free to define a boundary condition: the $\Phi = 0$ location. This freedom can be exploited to assist in construction of both the global RHS and global stiffness matrices. The basic strategy is as follows: select a $\Phi = 0$ plane,

use this plane to determine the dof for each and every node, and finally use the dof to decide which components of $[R]$ and $[A]$ need to be kept for the global matrix equation. The purpose of the dof is four-fold: it defines which values of $[\Phi]$ are already known, it determines which components of $[A]$ and $[\Phi]$ can be moved to the opposite side of (6.1) and added to their corresponding $[R]$ components, it will assist in figuring out which elements of $[A]$ are zero, and finally it details how to map matrix components from a local to global level. To better understand the use of boundary conditions and dof, consider the following example where $[A]$ is only a 3x3 matrix defined as ¹³

$$[A] = \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix} \quad (6.4)$$

which is both symmetric and singular ($\det[A] = 0$). Also let $[R]$ for this example appear as

$$[R] = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \quad (6.5)$$

Following the strategy outlined above, the first step is to define a solution, or boundary condition, to one of the components of $[\Phi]$ in the matrix equation

$$[A][\Phi] = [R] \quad (6.6)$$

which will be chosen to be $\Phi_2 = 0$. As will be seen in a moment, this designation will effectively remove the singularity of (6.6) and reduce the matrix equation so that it can be solved for the other components of $[\Phi]$. To distinguish Φ_2 as the assumed solution, its dof is set to zero whereas the dof of Φ_1 and Φ_3 are set to one and two, respectively. The

¹³ The credit for this simple example goes to an email provided by Professor Gary Bedrosian.

dof designation details how the matrix equation can be reduced. For instance, since Φ_2 is already known, we only need worry about the first and last equations when attempting to solve (6.6). Bringing the known Φ_2 components to other side of these equations yields

$$\begin{aligned} 2\Phi_1 - \Phi_3 &= -1 - (-\Phi_2) = -1 - 0 = -1 \\ -2\Phi_1 + 2\Phi_3 &= 1 - (-\Phi_2) = -1 - 0 = 1 \end{aligned} \quad (6.7)$$

which can be written as the reduced matrix equation

$$\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} \Phi_1 \\ \Phi_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \quad (6.8)$$

whose solutions are $\Phi_1 = -1/3$ and $\Phi_3 = +1/3$. Therefore the final solution of $[\Phi]$ is

$$[\Phi] = \begin{bmatrix} -1/3 \\ 0 \\ 1/3 \end{bmatrix} \quad (6.9)$$

So a quick review: the boundary condition was defined for some component of $[\Phi]$, which, after setting the dof, removed any singularities and determined the components of $[\Phi]$ that could be added to the RHS, essentially reducing the matrix equation. Finally, notice the pattern between the dof and the index of the component of Φ : the two quantities are identical before the boundary condition, but they are not afterwards. Rather they differ by the number of elements for which a solution was defined for, i.e. the number of elements with $dof = 0$. This misalignment highlights that although computer storage space can be saved by using the above method, it becomes more difficult to keep track of the positions of matrix elements when storing and retrieving information. Although the programming becomes a little more difficult, the benefit of reduced storage capacity makes it a worthwhile tradeoff. Also, recognize that the number of elements with $dof = 0$ corresponds to the drop in the size of both the $[A]$

and $[R]$ matrices. In other words, by designating one component of $[\Phi]$ as having $dof = 0$ (Φ_2), the stiffness matrix dropped from a 3×3 to a 2×2 matrix, and likewise the RHS matrix was reduced from a 3×1 to a 2×1 matrix. It is important to fully understand the concept of the boundary conditions and dof because they both will be crucial for not only the remainder of this section, but also interpreting the programming code associated with it.

For this project, the $z = 0$ plane was defined as the $\Phi = 0$ plane, due to the symmetry of the test problem, and the dof were then determined from this designation. For nodes below the $z = 0$ plane, the dof simply corresponded to the global node number of that particular node. For nodes lying on the $z = 0$ plane, the dof was set to zero, since the value of Φ was already defined there. Lastly, for nodes above the $z = 0$ plane, the dof ending up being a plane's worth of nodes below any given global node number. Note that just as above, the discrepancy between the dof and global node number after the $\Phi = 0$ plane is equal to the number of nodes for which $\Phi = 0$, $dof = 0$ was defined. Using these dof the appropriate elements of $[A]$ were removed from the matrix equation and added to their corresponding $[R]$ components, which were also removed from the matrix equation. Note these elements of the matrix equation were not discarded; rather they were taken out in order to reduce the matrix equation to the point where it could be solved for the other values of Φ . Finally, the dof were utilized in the removal of the sparseness of the global stiffness matrix by only storing the nonzero components of $[A]$. A graph of the values of the global RHS matrix, featuring both the components that were removed and those that remained, is given in Figures 6.3.

Unfortunately, generation of the global $[A]$ and $[R]$ matrices is as far as this project was able to progress due to time constraints. One success though, was the discovery that the slowest part of the RHS calculation, determination of the coil field $\vec{H}_c$, could be sped up by utilizing the GRAPE board (Table 6.1). As for the stiffness matrix though, it is still uncertain where and how the GRAPE board could be implemented. The best hypothesis at this point is that a table within the accelerator board would need to be programmed in order to perform computations associated with the stiffness matrix; specifically storing the reduced, global matrix and recovering the full, global matrix. Building a table on the GRAPE board is a concept with enough information to warrant the writing of a whole new paper though, and it will therefore not be introduced here.

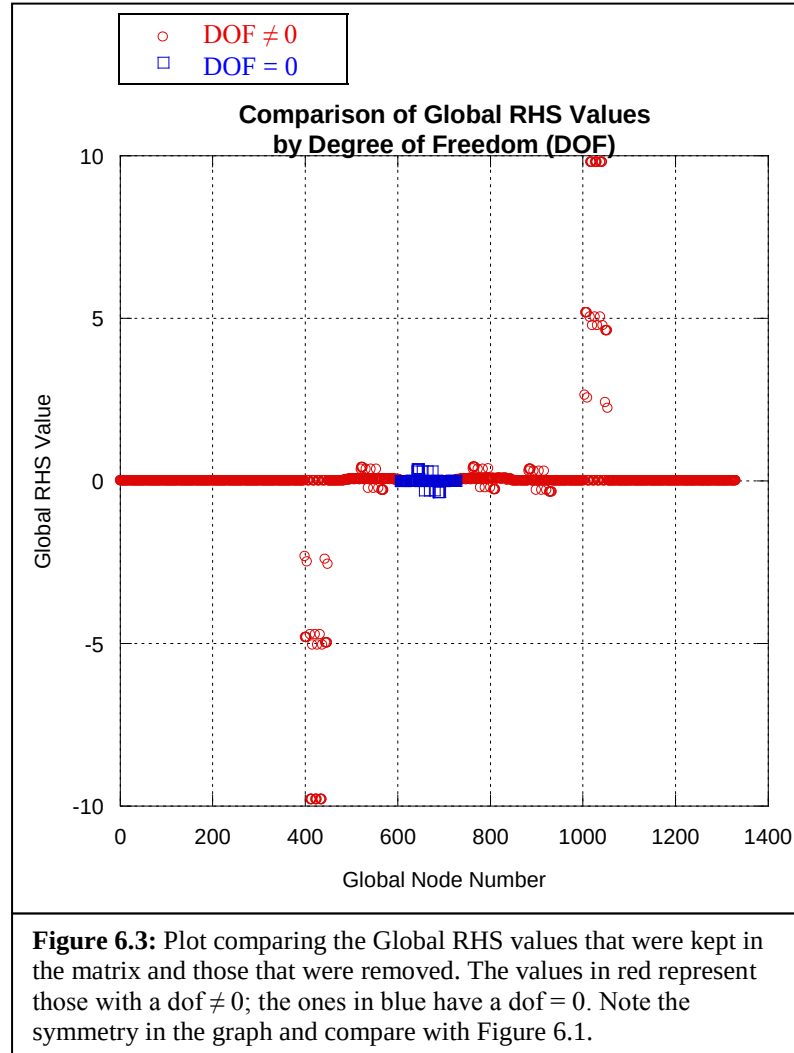


Table 6.1: Timing data for all the processes that went into the construction of the local RHS matrix. Note that although the calculation of the free coil field using the GRAPE board is still the slowest part, this process would be even slower had the accelerator board never been introduced.

Process	Time (sec)
Determination of Gaussian Integration Points	9.05991×10^{-6}
Use of the Subroutine SHAPE3D	3.6954×10^{-5}
Coordinate Mapping of Gaussian Integration Points	1.50204×10^{-4}
Use of the Subroutine DVJC3D	4.91381×10^{-4}
Coil Field Calculation w/ GRAPE	7.92670×10^{-2}
Local RHS Construction	1.23024×10^{-4}
Total Time	0.12662

VII. Results: Model Visualization

Besides the efforts put towards generating matrix equations, as outlined in the previous section, some work was also spent developing a method to plot the FEM mesh associated with the model, depicted in Figure 5.1. Such a mesh should consist of a $10 \times 10 \times 10$ cube composed of first order three-dimensional hexahedral (brick) elements of both iron and air. A number of programs, from MatLab to ones available for free on the internet, were considered, but it was eventually discovered that Mathematica was capable of handling such visualizations. Furthermore, the use of Mathematica maintains the spirit of the project: adapting what is already available at Allegheny College for the purposes of CEM. Although there is still a minor concern over the lag time Mathematica presents for visualizations of this magnitude, the program was a relatively simple solution to the problem of model visualization.

The basic procedure for creating plots in Mathematica for the FEM mesh was 3 fold: import the node coordinate data, plot the node coordinate data, and lastly form three-dimensional hexahedron (brick) finite elements plots from the nodes. Note that the mesh itself was generated by using the MakeFEM.f program provided by Professor Gary Bedrosian. A call to the MakeFEM.f program returns the global nodal coordinate data of both the air and iron elements present within the model. The node coordinate data could then be simply written into a file on the computer directly from MakeFEM.f in an appropriate format and then read into Mathematica as a table using the *Import* function. For example, if the name of the file containing the information about the global location of the nodes was “NodeData,” then the call in Mathematica to read in this information would appear as

Import ["NodeData.txt", "Table"] (7.1)

The only other condition, as seen from the previous statement, is that the file being read (NodeData) be saved as a plain text file in order for Mathematica to properly interpret it.

Once the node coordinate information has been brought into Mathematica, it can actually be directly plotted using the *ListPointPlot3D* function. This command will interpret the previous list of node data as groups of x, y, and z coordinates and plot them as such. For example, if the previously imported node data were stored as the variable "Nodes" the plot call would be as simple as

ListPointPlot3D [Nodes] (7.2)

While there are also a number of other features available to manipulate such plots, the basic command is given in (7.2). Plots of the iron nodes only and the iron and air nodes together are featured in Figure 7.1. Note that distinguishing between which nodes are purely air, which are purely iron, and which are on the air/iron interface was done via a "flagging" technique within MakeFEM.f.

The last step to create a finite element mesh is connecting the previous figures' nodes to form the cubic elements of the model, which can be done using Mathematica's *Cuboid* and *Graphics3D* functions. The basic form of the *Cuboid* function is as follows

Cuboid [{ x_{min} , y_{min} , z_{min} }, { x_{max} , y_{max} , z_{max} }] (7.3)

where the two sets of coordinates specify opposite corners of the cube to be constructed. Since all of the cubes within the mesh follow a regular pattern, replacing x_{min} , y_{min} , and z_{min} with coordinate data for a single corner of each element and defining the corner opposite the first will be sufficient for model visualization. Combining the *Cuboid* function with the *Graphics3D* function then allows for plotting of the FEM mesh. To

understand how to use these two functions together, consider a cube with one corner at the origin and side lengths of 1 (arbitrary units). The corner opposite the one at the origin will then have Cartesian coordinates of (1,1,1). If this cube's origin coordinate data were read into Mathematica and stored as "Corner" then the cube could be plotted using the following

Graphics3D [Corner/. {x_, y_, z_}:-> , **Cuboid** [{x, y, z}, {x + 1, y + 1, z + 1}]] (7.4)¹⁴

Note that the beginning part of the previous command

Corner/. {x_, y_, z_}:-> (7.5)

is merely a means of temporarily replacing the x, y, and z variables in the *Cuboid* function with the coordinate values from Corner. In the actual plots of the air and iron elements, the corner data was found by manipulating the output of the MakeFEM.f program. The plots of the nodes and mesh can finally be combined by defining the graphics outputs as variables and using the *Show* function. Plots of the mesh, both nodes and cubic elements, appear in Figure 7.2.

¹⁴ Special thanks goes to the Wolfram Mathematica Corporation, whose booth at the March meeting of the American Physical Society assisted in locating, understanding, and implementing such functions for visualization generation.

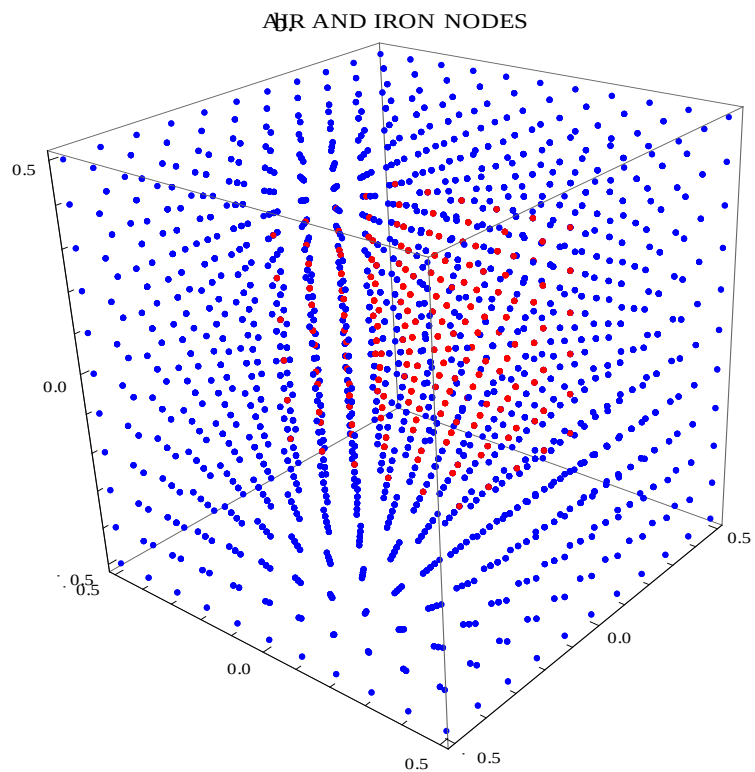
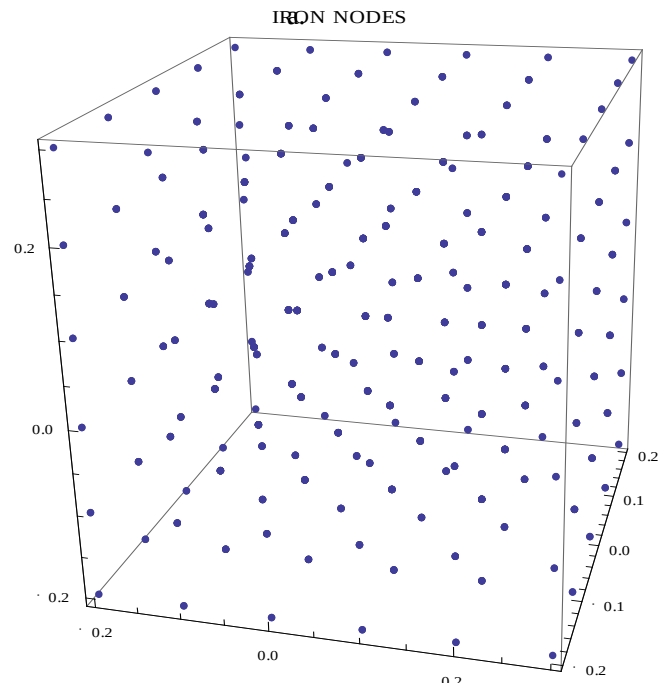
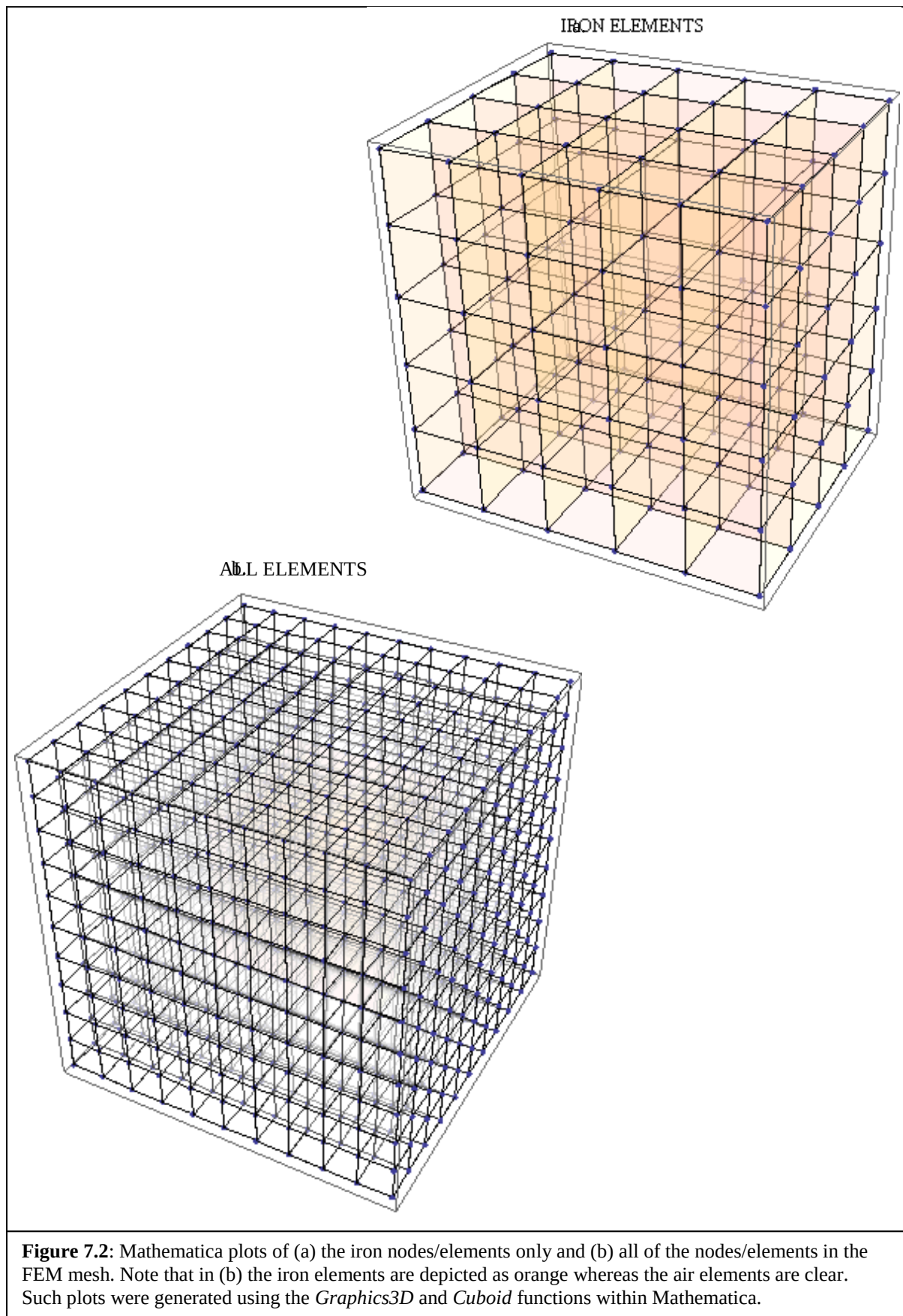


Figure 7.1: Mathematica plots of (a) the iron nodes only and (b) all of the nodes in the FEM mesh. Note that in (b) the iron nodes are depicted as red whereas the air nodes are blue. Such plots were generated using the *ListPointPlot3D* function within Mathematica.



VIII. Conclusions & Future Work

Although this project did not reach its full potential, it was a step in the right direction for the implementation of CEM via the GRAPE board and should be considered a success. The work showed that not only could the accelerator board be used for basic electromagnetic analysis, but also the potential of fully adapting the GRAPE board for the purposes of CEM. Future work for this study would include continuing the FEM analysis of the model to see where and how the GRAPE board could be introduced into the $[A]$ calculation. Specifically, as was speculated above, it would most likely involve defining a table within the GRAPE board's library to perform calculations similar to those used in the generation of the stiffness matrices.

Also, it will be necessary to devise a scheme for displaying the values of $[A]$ in an effective manner. The matrix is very large (1331×1331) so it makes no sense to just show all 1,771,561 elements in numeric format because it will be very difficult to understand. One solution could be to use Mathematica to turn $[A]$ into a picture instead of a matrix of numbers. Specifically, the *ArrayPlot* function could plot each component of the matrix as a different color region in the matrix depending on its magnitude. This would highlight the sparseness of the stiffness matrix and clarify how the matrix reduction process takes place. Furthermore, the steps to use *ArrayPlot* would be very similar to those already seen with the discussion of model visualization generation (see Section VII).

Beyond the findings, the project was a beneficial and challenging learning experience for an undergraduate in physics. It was enjoyable to work on a topic outside the neat confines of a textbook problem and had the potential to become something

tangible in the real world. In conclusion, the project should be considered a success and hopefully one day another undergraduate will continue and further the research of manipulating the GRAPE board for CEM.

IX. Appendix

A. FORTRAN Code

The following is the modified H3D code used in the proof of concept for manipulating the GRAPE board for CEM:

```
C*****
)
C*
*)
C* Module name : GrapeTestTime
*)
C*
*)
C* Module description : Adapted from the HcScan module, the following
*)
C* program was developed as a proof of concept for the ability to
*)
C* to manipulate the MD-GRAPE accelerator board for the purposes of
*)
C* of basic CEM. The program itself was written and adapted for use
*)
C* in companion with the GRAPE board by Professor Gary Bedrosian,
*)
C* Allegheny College. The timing sequences and multiple calculation
*)
C* loops were added by Adam Simbeck, Allegheny College, as motivation
*)
C* for his senior project, which is under the advisory of Gary
*)
C* Bedrosian.
*)
C*
*)
C*
*)
C* Author(s) : Gary Bedrosian
*)
C* User/Editor: Adam Simbeck
*)
C*
*)
C* Organization(s) : Gary Bedrosian                Adam J. Simbeck
*)
C*                  P.O. Box 451                    Box 2210
*)
C*                  Chautauqua, NY 14722            Meadville, PA 16335
*)
C*                  716-753-2061                     814-336-9675
*)
C*
*)
```

```

C* Version : 1.1
*)
C*
*)
C* Date : 04-Nov-08
*)
C*
*)
C* Revisions : 05-Nov-08 for adding MDGRAPE-3 skeleton
*)
C*           20-Nov-08 for timing comparsion
*)
C*
*)
C* Copyright (C), 2008 by Gary Bedrosian
*)
C*
*)
C*****
)
C*
*)
C* subroutine hcscan(x,y,z,hx,hy,hz)
*)
C* x,y,z - real*8 coordinates of evaluation point (input)
*)
C* hx,hy,hz - real*8 components of magnetic field (output)
*)
C* t1,t2,t3,t4 - real*8 times of calculations (output)
*)
C* calculates H from the defined Gaussian integration points
*)
C*
*)
C*****
)
C*
*)
C* subroutine hcmdgrape(ierr)
*)
C* calculates H from the defined Gaussian integration points and
*)
C* saves the results in the common data storage arrays
*)
C* returns an integer error code, = 0 means no error
*)
C*
*)
C*****
)
C*
*)
C* subroutine rdgau(filnam)
*)
C* filnam - character*(*) file name (input)
*)

```

```

C* read the saved Gaussian integration data from a file
*)
C*
*)
C*****
)
C*
*)
C* subroutine rdeval(filnam)
*)
C* filnam - character*(*) file name (input)
*)
C* read the evaluation points from a file
*)
C*
*)
C*****
)
C*
    program grapetesttime
    real*8 x,y,z,hx,hy,hz,t1,t2
    call rdgau('JGauss.dat')
    call rdeval('evalpts.dat')
    call hcmdgrape(ierr)
    end

C*
C*****
)

    block data zscanz
    include 'hcscan.inc'
    data ngpts /0/
    data nepts /0/
    end

C*
C*****
)
C*
    subroutine rdgau(filnam)
C common to both, only run once
C reads Gaussian integration data
    include 'hcscan.inc'
    character*(*) filnam
    integer g
    open(unit=23,file=filnam,status='OLD')
C read the actual number of points from the file
    read(23,*) ngpts
    if (ngpts.gt.maxgs) then
        write(*,*) 'Too many Gaussian points =',ngpts,maxgs
        stop
    end if
C for each point read 2 lines
    do 1 g=1,ngpts
C the 1st has the location
        read(23,*) xg(g),yg(g),zg(g)
C the 2nd has the x,y,z components of current density * dvolume

```

```

        read(23,*) xi(g),yi(g),zi(g)
        xi(g) = xi(g)/10000.0d0
        yi(g) = yi(g)/10000.0d0
        zi(g) = zi(g)/10000.0d0
1      end do
      close(23)
      return
    end

C*
C*****
)
C*
      subroutine rdeval(filnam)
C common to both, only run once
C reads evaluation points from a file
      include 'hcscan.inc'
      character*(*) filnam
      integer e,n,repeat
      parameter(repeat=100)
      real*8 xo,yo,zo,hxo,hyo,hzo
      real*8 hxn(maxes),hyn(maxes),hzn(maxes)
      real*8 t1,t2,rtc
      open(unit=23,file=filnam,status='OLD')
C read the actual number of points from the file
      read(23,*) nepts
      if (nepts.gt.maxes) then
        write(*,*) 'Too many evaluation points =',nepts,maxes
        stop
      end if
C for each point read 1 line
      do 1 e=1,nepts
C the location of the evaluation points
        read(23,*) xe(e),ye(e),ze(e)
1      end do
C time and loop through calculation of H w/out mdgrape
      t1 = rtc()
      do 3 n=1,repeat
        do 4 e=1,nepts
          xo = xe(e)
          yo = ye(e)
          zo = ze(e)
          call hcscan(xo,yo,zo,hxo,hyo,hzo)
          hxn(e) = hxo
          hyn(e) = hyo
          hzn(e) = hzo
C
          open(unit=12,file='NoGrapeTime.dat',status='NEW')
C
          write(12,*) hxo,hyo,hzo
4        end do
3      end do
      t2 = rtc()
      close(23)
      print*, 'W/OUT GRAPE BOARD H=',hxn(1),hyn(1),hzn(1)
      print*, 'TIME W/OUT GRAPE BOARD(SEC)=',t2-t1
      return
    end

C*

```

```

C*****
)
C*
    subroutine hcscan(x,y,z,hx,hy,hz)
    include 'hcscan.inc'
    real*8 x,y,z,hx,hy,hz
    integer g,n
    real*8 rx,ry,rz
    real*8 r2,r3
    real*8 cx,cy,cz
    real*8 jx,jy,jz
    hx = 0.0d0
    hy = 0.0d0
    hz = 0.0d0
C loop though all Gaussian points of the coil
    do 1 g=1,ngpts
C find the vector from the evaluation point to the Gauss point
        rx = xg(g)-x
        ry = yg(g)-y
        rz = zg(g)-z
        r2 = rx*rx+ry*ry+rz*rz
        r3 = 1.0d0/(r2*sqrt(r2))
        jx = xi(g)
        jy = yi(g)
        jz = zi(g)
C cross product in numerator of Biot-Savart formula
        cx = ry*jz-rz*jy
        cy = rz*jx-rx*jz
        cz = rx*jy-ry*jx
C divide by distance cubed and add to total
        hx = hx+r3*cx
        hy = hy+r3*cy
        hz = hz+r3*cz
1    end do
C save dividing by 4 pi to end so only do it once
    hx = hx/(twopi+twopi)*10000.0d0
    hy = hy/(twopi+twopi)*10000.0d0
    hz = hz/(twopi+twopi)*10000.0d0
    return
    end
C*
C*****
)
C*
*)
C* the rest of this module deals with the MDGRAPE-3 board
*)
C* adapted from Fortran code provided by Prof. Lombardi
*)
C* 05-Nov-08 note: not tested yet
*)
C*
*)
C*****
)
C*
    subroutine hcmdgrape(ierr)

```

```

        include 'hscan.inc'
        integer ierr,n
c      include 'sphu.h'
        include 'm3_unit.h'
C factor is an overall multiplier for the final result
C 05-Nov-08 this is a guess for now, will set when tested
        real*8 factor,repeat
        parameter(repeat=100)
        parameter (factor=-(10000.0d0/repeat)/(twopi+twopi))
        real*8 xyzmin,xyzmax,range
        real*8 t3,t4,rtc
        integer g,e,mswait1,waited,maxwait
        character*(18) gravtable
C be optimistic!
        ierr = 0
C first thing, find the limits of the problem
        xyzmin = xg(1)
        xyzmax = xyzmin
        do 1 g=1,ngpts
            xyzg(1,g) = xg(g)
            xyzg(2,g) = yg(g)
            xyzg(3,g) = zg(g)
            xyzmin = min(xyzmin,xg(g),yg(g),zg(g))
            xyzmax = max(xyzmax,xg(g),yg(g),zg(g))
1        end do
        do 2 e=1,nepts
            xyze(1,e) = xe(e)
            xyze(2,e) = ye(e)
            xyze(3,e) = ze(e)
            xyzmin = min(xyzmin,xe(e),ye(e),ze(e))
            xyzmax = max(xyzmax,xe(e),ye(e),ze(e))
            hxe(e) = 0.0d0
            hye(e) = 0.0d0
            hze(e) = 0.0d0
2        end do
C push the limits out a little
        range = (xyzmax-xyzmin)*1.0d-5
        xyzmax = xyzmax+range
        xyzmin = xyzmin-range
C initialize the board
C gravtable = ? get the name and data from Prof. Lombardi
        gravtable = 'grav.tblmd3'
C milliseconds to wait if board is busy
        mswait1 = 100
C max time to wait in ms = #minutes * 60 sec/min * 1000 ms/s
        maxwait = 10*60*1000
        waited = 0
3        m3unit = m3_allocate_unit(gravtable,m3_force,xyzmin,xyzmax,0)
        if (m3unit.eq.0) then
C suspends the program for a short time
            call sleepqq(mswait1)
            waited = waited+mswait1
C if it waited too long = error condition
            if (waited.gt.maxwait) then
                ierr = 1
                return
            end if

```

```

C otherwise, try again
    go to 3
end if
C do the Biot-Savart
C set the positions of the currents -- same for all components
    call m3_set_positions(m3unit,xyzg,ngpts)
C time and loop through calculations with mdgrape
    t3 = rtc()
    do 8 n=1,repeat
C Jx terms
        call m3_set_charges(m3unit,xi,ngpts)
        call m3_calculate_forces(m3unit,xyze,nepts,field)
        do 4 e=1,nepts
            hye(e) = hye(e)+field(3,e)
            hze(e) = hze(e)-field(2,e)
4        end do
C Jy terms
        call m3_set_charges(m3unit,yi,ngpts)
        call m3_calculate_forces(m3unit,xyze,nepts,field)
        do 5 e=1,nepts
            hze(e) = hze(e)+field(1,e)
            hxe(e) = hxe(e)-field(3,e)
5        end do
C Jz terms
        call m3_set_charges(m3unit,zi,ngpts)
        call m3_calculate_forces(m3unit,xyze,nepts,field)
        do 6 e=1,nepts
            hxe(e) = hxe(e)+field(2,e)
            hye(e) = hye(e)-field(1,e)
6        end do
8    end do
C release the board
    call m3_free_unit(m3unit)
C if we have to do any overall scaling of H, do it here...
    if (factor.ne.1.0d0) then
        do 7 e=1,nepts
            hxe(e) = factor*hxe(e)
            hye(e) = factor*hye(e)
            hze(e) = factor*hze(e)
C        open(unit=12,file='MDGData2.out',status='NEW')
C        write(12,*) hxe(e),hye(e),hze(e)
7        end do
    end if
    t4 = rtc()
    print*, 'GRAPE BOARD: H(1)=', hxe(1),hye(1),hze(1)
    print*, 'GRAPE BOARD TIME(SEC)=', t4-t3
C H is complete
    return
end

C*
C*****
)
C*
C*
C*****
)
C*

```

```

C*****
)
C*
*)
C* Module name : HcScan.inc
*)
C*
*)
C* Module description : Include file for HcScan module - coil field
*)
C*
*)
C*
*)
C* Author(s) : Gary Bedrosian
*)
C*
*)
C* Organization : Gary Bedrosian
*)
C*                P.O. Box 451
*)
C*                Chautauqua, NY 14722
*)
C*                716-753-2061
*)
C*
*)
C* Version : 1.1
*)
C*
*)
C* Date : 04-Nov-08
*)
C*
*)
C* Revisions : 05-Nov-08 added storage for evaluation arrays
*)
C*
*)
C* Copyright (C), 2008 by Gary Bedrosian
*)
C*
*)
C*****
)
C
C maxgs - maximum stored Gaussian integration locations
C ngpts - actual stored Gaussian integration locations
C xg,yg,zg - locations of Gaussian integration locations
C xi,yi,zi - currents at the Gaussian integration locations
C maxes - maximum stored evaluation locations
C nepts - actual stored evaluation locations
C xe,ye,ze - evaluation locations for evaluating Hc
C mdunit - unit ID for MDGRAPE-3 board
C hxe,hye,hze - values of Hc at the evaluation locations
C xyzg,xyze,field - 2D arrays for the MDGRAPE-3 board functions

```

```

C
    implicit none
    integer maxgs,maxes
    parameter (maxgs=10000,maxes=10000)
    integer ngpts,nepts,m3unit
    real*8 twopi
    parameter (twopi=6.283185307179586476925287d0)
    real*8 xg(maxgs),yg(maxgs),zg(maxgs)
    real*8 xi(maxgs),yi(maxgs),zi(maxgs)
    real*8 xe(maxes),ye(maxes),ze(maxes)
    real*8 hxe(maxes),hye(maxes),hze(maxes)
    real*8 xyzg(3,maxgs),xyze(3,maxes),field(3,maxes)

C
    common /xscanx/ xg,yg,zg,xi,yi,zi,ngpts
    common /yscany/ xe,ye,ze,hxe,hye,hze,nepts
    common /xyzxyz/ xyzg,xyze,field,m3unit

```

The following is an excerpt from the modified MakeFEM.f code that generates the RHS and stiffness matrices:

```

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C for nodes:
C maxnod - maximum number of nodes
C nodes - actual number of nodes
C xn,yn,zn - node location
C renum - renumbering array
C ndflag - flag for general use
C
C for edges:
C maxedg - maximum number of edges
C edges - actual number of edges
C edgn0,edgn1,edgmd1,edgmd2 - nodes on edge
C enodes - number of nodes on edge
C enlist - list of nodes on edge (hard-coded for 3rd order max)
C edflag - flag for general use
C
C for faces:
C maxfac - maximum number of faces
C faces - actual number of faces
C facedg - edge numbers of the face
C edgdir - direction of the edge (true = forward)
C factyp - 3 = triangle, 4 = quad (also the number of edges)
C facell - elem on inside of face
C facel2 - elem on outside of face
C midfac - mid-face node number (0 = none)
C fnodes - number of nodes on face
C fnlist - list of nodes on face (hard-coded for 3rd order max)
C fcflag - flag for general use
C
C
C for elems:
C maxele - maximum number of elems
C elems - maximum number of elems

```

```

C elefac - face numbers of the elem
C facdir - direction of the face (true = normal outward)
C eletyp - 4 = tetrahedron, 5 = wedge, -5 = pyramid, 6 = brick
C lnodes - number of nodes in elem
C lnlist - list of nodes in elem (hard-coded for 3rd order max)
C elflag - flag for general use
C ironel - 2D array to store elements of iron
C
CCCCC
C ironodes - 2D array to store iron element nodes
C airnodes - 2D array to store air element nodes
C ironcoors- 2D array to store the coord of the iron element nodes
C ixx,iyy,izz - temporary arrays to store iron coord on
C                loccal node number basis
C aircoord - 2D array to store the coord of the air element nodes
C
C for gaus3d:
C maxgpts - maximum number of gaussian integration points
C totgpts - total number of gaussian integration points
C ug1,ug2,ug3 - arrays of isoparametric Gaussian integration points
C wg - array of gaussing integration weights
C
C for shap3d: (2D arrays to store data of 1 cube)
C ulder,u2der,u3der - 2D arrays to store derivatives
C P - 2D array to store shape function
C noper - number of nodes per element (8)
C dndu1,dndu2,dndu3 - arrays of derivatives wrt isoparametric
coordinates
C NP - temporary shape function storage array
C
C for dvjc3d:
C xnode,ynode,znode - arrays containing nodal coordinates of elements
C detjac - determinant of the Jacobian of transformation
C dndx,dndy,dndz - arrays of shape fucntion partial derivatives
C dxder,dyder,dzder - arrays to hold the derivatives wrt x,y,z
C                for each gaussian integration point
C
C xg,yg,zg - 2D arrays to store global coordinates of the gaussian
integration points
C
C for rhs matrix:
C rhs - 1D array to store elements of the right hand side matrix
C mu - permeability of iron (set at 1000mu0)
C mu0 - permeability of free space
C
C for stiffness matrix A:
C Alocal - 2D array to store components of A matrix on a local nodal
basis
C eps,neps - small interval about zero used to define phi=0 plane and
flag nodes
C dof - array to store degree of freedom of nodes, which is used in
sorting nodes
C                on the basis of boudary conditions and building the global
stiffness matrix
C idof,jdof - temporary variables used in sorting the Alocal components
C ndof - maximum number of degrees of freedom; really just a counter
C nmax - maximum size of the global stiffness matrix A

```

```

C nzmax - maximum number of elements in a row that could be zero
C diag - array to hold diagonal components of stiffness matrix
C upper - array to hold only upper portion of A matrix since it is
symmetric
C colidx -
C add_to_Aglobal - subroutine to store Alocal into Aglobal matrix
C initAglobal - initial all components of global A matrix and dbl
check
C
C           to make sure matrix size is large enough
C
C for timing:
C rtc - variable used to determine computing time of various units of
the program
C
CCCCC
C
C additional data:
C mtitle - title of the mesh
C mdate - date the mesh was created
C mtime - time the mesh was created
C order - order of the mesh (1,2,3)
C errcod - error number stored (various values)
C
C max sizes based on a cubic array of bricks, up to 3rd order
C           integer maxnod,maxedg,maxfac,maxele
C           parameter (maxele=12500)
C           parameter (maxfac=3*maxele)
C           parameter (maxedg=3*maxele)
C           parameter (maxnod=maxele+2*maxedg)
C
C node data
C           integer nodes
C           integer renum(maxnod),ndflag(maxnod)
C           real*8 xn(maxnod),yn(maxnod),zn(maxnod)
C           real*8 ironcoord(maxnod,3),aircoord(maxnod,3)
C
C edge data
C           integer edges
C           integer edgn0(maxedg),edgn1(maxedg),edgmd1(maxedg),edgmd2(maxedg)
C           integer enodes(maxedg),enlist(4,maxedg),edflag(maxedg)
C
C face data
C           integer faces
C           integer facedg(4,maxfac)
C           logical edgdir(4,maxfac)
C           integer factyp(maxfac),facel1(maxfac),facel2(maxfac)
C           integer midnod(maxfac)
C           integer fnodes(maxfac)
C           integer fnlist(12,maxfac)
C           integer fcflag(maxfac)
C
C elem data
C           integer elems
C           integer elefac(6,maxele)
C           logical facdir(6,maxele)
C           integer eletyp(maxele)

```

```

integer lnodes(maxele)
integer lnlist(32,maxele)
integer elflag(maxele)
integer ironodes(maxele,8)
integer airnodes(maxele,8)
real*8 ix(8),iyy(8),izz(8)

C
C guas3d data
integer maxgpts,totgpts
parameter(maxgpts = 64)
real*8 ug1(maxgpts),ug2(maxgpts),ug3(maxgpts)
real*8 wg(maxgpts)

C
C shap3d data
integer nodper
parameter(nodper = 8)
real*8 P(maxgpts,nodper)
real*8 u1der(maxgpts,nodper),u2der(maxgpts,nodper),
+          u3der(maxgpts,nodper)
real*8 dndu1(nodper),dndu2(nodper),dndu3(nodper)
real*8 NP(nodper)

C
C dvjc3d data
real*8 xnode(nodper),ynode(nodper),znode(nodper)
real*8 detjac
real*8 dndx(nodper),dndy(nodper),dndz(nodper)
real*8 dxder(maxgpts,nodper),dyder(maxgpts,nodper),
+          dzder(maxgpts,nodper)

C
C rhs data
real*8 rhs(maxnod)

C
C stiffness matrix A data
integer idof,jdof,ndof
real*8 Alocal(nodper,nodper)
integer nzmax,nmax
parameter(nzmax=26,nmax=1331)
real*8 upper(nzmax,nmax)
real*8 diag(nmax)
real*8 colidx(nzmax,nmax)
integer dof(maxnod)
real*8 eps,neps
real*8 xmax,xmin,ymax,ymin,zmax,zmin

C
C additional data
integer order,errcod,iflag
real*8 xxg(100,maxgpts),y yg(100,maxgpts),zzg(100,maxgpts)
real*8 rtc
character*80 mtitle
character*9 mdate
character*8 mtime

```

```

C
C define the functions of the module for calling
C the following have already been defined in the program FEMData.f
itself
C they have therefore been commented out so as to not double declare
the variables
    integer addnod
    integer addedg
    integer addfac
    integer addele
    integer cnedgs
    integer cefacs
    logical fwd2ix

C store in common blocks
    common /femnnn/ nodes,edges,faces,elems,order,errcod,
&
    mtitle,mdate,mtime
    common /femnod/ xn,yn,zn,renum,ndflag
    common /femedg/ edgn0,edgn1,edgmd1,edgmd2,enodes,enlist,edflag
    common /femfac/ facedg,edgdir,factyp,facell,facel2,midnod,
&
    fnodes,fnlist,fcflag
C ADDED NEW ARRAYS TO COMMON STATEMENT
    common /femele/ elefac,facdir,eletyp,lnodes,lnlist,elflag,
&
    ironodes,airnodes,ironcoord,aircoord,P,
&
    ulder,u2der,u3der,dndu1,dndu2,dndu3,NP
&
    dxder,dyder,dzder,rhs,intAmatrix
C*
C*****
)
C*
    real*8 u1,u2,u3,mu,munot,fourpi
    real*8 x,y,z,hx,hy,hz
    real*8 t1,t2,t3,t4,t5,t6,t7,t8,t9
    real*8 t10,t11,t12,t13,t14,t15,t16
    real*8 coordtime,dvjc3dtime
    integer etype,gorder,gpts
    parameter(etype = 6)
    include 'femdata.inc'
    include 'hcscan.inc'

C THE FOLLOWING HAS BEEN ADDED TO THE ORIGINAL PROGRAM
C SET ORDER AND NUMBER OF GAUSSIAN INTEGRATION POINTS
    gorder = 2
    gpts = 20
    t1 = rtc()
C DETERMINES THE GUASSIAN INTEGRATION POINTS
    t3 = rtc()
    call gaus3d(etype,nodper,gorder,gpts,ug1,ug2,ug3,wg)
    t4 = rtc()
    print*,'TIME TO DETERMINE GUASS PTS:',t4-t3

C OPEN FILE TO DBL CHECK VARIOUS PORTIONS OF DATA
C    open(unit=25,file='chk.dat',status='NEW')
C OPEN FILE TO STORE GLOBAL COORDINATES OF GUASSIAN INT PTS
C AND PASS THEM TO THE GRAPE BOARD
    open(unit=23,file='gpts.dat',status='NEW')
C FILE TO HOLD OPPOSITE CORNER NODE DATA FOR MATHEMATICA VISUAL

```

```

C      open(unit=27,file='ICorn.dat',status='NEW')
C      open(unit=28,file='ACorn.dat',status='NEW')
C DO LOOP TO DETERMINE AND STORE THE SHAPE FUNCTIONS AND
C DERIVATIVES WRT U1,U2,U3 FOR ONE ELEMENT ONLY!!!

      t5 = rtc()
      do 1 i=1, gpts
        u1 = ug1(i)
        u2 = ug2(i)
        u3 = ug3(i)
C      write(25,*) ug1(i),ug2(i),ug3(i),wg(i)
      call shap3d(etype,nodper,u1,u2,u3,NP,dndu1,dndu2,dndu3)
      do 2 lnn=1, nodper
        P(i,lnn)      = NP(lnn)
        ulder(i,lnn) = dndu1(lnn)
        u2der(i,lnn) = dndu2(lnn)
        u3der(i,lnn) = dndu3(lnn)
C      write(25,*) P(i,lnn),ulder(i,lnn),u2der(i,lnn),
C      +          u3der(i,lnn)
2      continue
1      continue
      t6 = rtc()
      print*,'SHAP3D TIME(SEC):',t6-t5
C COUNTER TO DETERMINE NUMBER OF IRON ELEMENTS
      count = 0
C DO LOOPS TO DETERMINE IRON ELEMENTS, NODES, AND DERIVATIVES
      do el=1,elems
        write(iounit,*) elflag(el),eletyp(el)
C DIFFERENTIATE BTW IRON AND AIR ELEMENT NODES
        if (elflag(el).eq.1) then
C COUNTER TO DETERMINE TOTAL NUMBER OF IRON ELEMENTS
          count = count + 1
          do 6 lnn=1, nodper
C 2D ARRAY TO STORE THE IRON ELEMENT'S GLOBAL NODE NUMBERS
            nn = lnlist(lnn,el)
            ironodes(el,lnn) = nn
C      write(25,*) ironodes(el,lnn)
C 2D ARRAY TO STORE THE COORD OF THE IRON ELEMENT NODES
C ON A GLOBAL NODE NUMBER BASIS
            ironcoord(nn,1) = xn(nn)
            ironcoord(nn,2) = yn(nn)
            ironcoord(nn,3) = zn(nn)
C TEMPORARY VARIABLE TO STORE ELEMENT COORDINATES
C ON A LOCAL NODE NUMBER BASIS
            ixx(lnn) = xn(nn)
            iyy(lnn) = yn(nn)
            izz(lnn) = zn(nn)
C IF STATEMENT TO DETERMINE OPP CORNER NODE COORD DATA
C TO BE USED IN MATHEMATICA VISUALIZATIONS
C      if (lnn.eq.1) then
C      write(27,100) nn,xn(nn),yn(nn),zn(nn)
C100      format(i4,2x,1pe10.2,2x,1pe10.2,2x,1pe10.2)
C      end if
C      write(25,*) ironcoord(nn,1),ironcoord(nn,2),
C      +          ironcoord(nn,3)
6      continue
C DO LOOP TO MAP FROM LOCAL TO GLOBAL COORDINATES FOR THE

```

```

C GAUSSIAN INTEGRATION POINTS
      t7 = 0
      t7 = rtc()
      do 10,i=1, gpts
C INITIALIZE ARRAYS WHICH WILL HOLD COORDINATES
          xxg(el,i) = 0
          yyg(el,i) = 0
          zzg(el,i) = 0
C CALC COORDINATES USING IRON COORD AND SHAPE FUNX
          do 9, lnn=1, nodper
              xxg(el,i) = xxg(el,i) + ixx(lnn) * P(i,lnn)
              yyg(el,i) = yyg(el,i) + iyy(lnn) * P(i,lnn)
              zzg(el,i) = zzg(el,i) + izz(lnn) * P(i,lnn)
          9          continue
C              write(25,*) xxg(el,i),yyg(el,i),zzg(el,i)
10          continue
          t8 = 0
          t8 = rtc()
          coordtime = coordtime + (t8-t7)
C KEEP IRONCORD DATA BUT PASS DATA TO DVJC3D WITH XNODE,YNODE,ZNODE
          t9 = 0
          t9 = rtc()
          do 3 lnn=1, nodper
              nn = lnlist(lnn,el)
              xnode(lnn) = ironcoord(nn,1)
              ynode(lnn) = ironcoord(nn,2)
              znode(lnn) = ironcoord(nn,3)
          3          continue
C GET BACK THE DERIVATIVES WRT U1,U2,U3 FOR DVJC3D
          do 4 i=1, gpts
              do 5 lnn=1, nodper
                  dndu1(lnn) = ulder(i,lnn)
                  dndu2(lnn) = u2der(i,lnn)
                  dndu3(lnn) = u3der(i,lnn)
C              write(25,*) dndu1(lnn),dndu2(lnn),dndu3(lnn)
          5          continue
C DETERMINE DERIVATIVES WRT X,Y,Z AND STORE
                  call dvjc3d(nodper,xnode,ynode,znode,dndu1,dndu2,
+                      dndu3,detjac,dndx,dndy,dndz)
C                      write(25,*) detjac
                  do 8 lnn=1, nodper
                      dxder(i,lnn) = dndx(lnn)
                      dyder(i,lnn) = dndy(lnn)
                      dzder(i,lnn) = dndz(lnn)
C                      write(25,*) dndx(lnn),dndy(lnn),dndz(lnn)
          8                      continue
          4          continue
          t10 = 0
          t10 = rtc()
          dvjc3dtime = dvjc3dtime + (t10-t9)
          else if (elflag(el).eq.0) then
              do 7 lnn=1, nodper
C 2D ARRAY TO STORE THE AIR ELEMENT'S GLOBAL NODE NUMBERS
                  nn = lnlist(lnn,el)
                  airnodes(el,lnn) = nn
C 2D ARRAY TO STORE THE COORD OF THE AIR ELEMENT NODES
                  aircoord(nn,1) = xn(nn)

```

```

        aircoord(nn,2) = yn(nn)
        aircoord(nn,3) = zn(nn)
C SET RHS EQUAL TO ZERO FOR AN AIR ELEMENT
        rhs(nn) = 0
C IF STATEMENT TO DETERMINE NODE CORNER COORD DATA
C FOR MATHEMATICA VISUALIZATIONS
C        if (lcn.eq.1) then
C            write(28,170) nn,xn(nn),yn(nn),zn(nn)
C170        format(i4,2x,1pe10.2,2x,1pe10.2,2x,1pe10.2)
C        end if
7        continue
    end if
        write(iounit,*) (elefac(i,el),facdir(i,el),i=1,6)
    end do
    print*, 'COORD MAPPING TIME(SEC):', coordtime
    print*, 'DVJC3D TIME(SEC):', dvjc3dtime

C TOTAL NUMBER OF GUASSIAN INTEGRATION POINTS
    totgpts = gpts*count
    write(23,*) totgpts
C DO LOOP TO STORE GLOBAL GAUSS INT PT COORD TO BE USED W/ GRAPE
    do 13 el=1, elems
        if (elflag(el).eq.1) then
            do 14 i=1,gpts
                write(23,*) xxg(el,i),yyg(el,i),zzg(el,i)
14            continue
            end if
13        continue

C CALL GRAPE TO DETERMINE HC FIELD
C KEEP SAME CURRENT PTS
        t11 = rtc()
        call rdgau('JGauss.dat')
C CHANGE EVAL PTS TO THE GLOBAL GAUSS INT PTS
        call rdeval('gpts.dat')
        call hcscan(x,y,z,hx,hy,hz)
        call hcmdgrape(ierr)
        t12 = rtc()
        print*, 'TIME IMPLEMENTING GRAPE(SEC):', t12-t11

C DEFINE VARVIABLES NEEDED FOR RHS CALC
        fourpi = twopi + twopi
        munot = fourpi/(10000000)
        mu = 1000*munot

C INITIALIZE MAX X,Y,Z VALUES
        xmax = 0
        xmin = 0
        ymax = 0
        ymin = 0
        zmax = 0
        zmin = 0
C FIND MAX X,Y,Z VALUES FOR DIAGONAL OF SQUARE
        do i=1, nodes
            xmax = max(xn(i),xmax)
            xmin = min(xn(i),xmin)
            ymax = max(yn(i),ymax)

```

```

        ymin = min(yn(i),ymin)
        zmax = max(zn(i),zmax)
        zmin = min(zn(i),zmin)
    end do
C DEFINE SMALL INTERVAL AROUND ZERO TO ACCOUNT
C FOR ROUND OFF ERROR WHEN ASSIGNING DOF VALUES
    eps = 1d-10*sqrt((xmax-xmin)**2+
+ (ymax-ymin)**2+(zmax-zmin)**2)
    neps = -eps

C DO LOOP TO FLAG NODES BASED ON BOUNDARY CONDITIONS OF PHI
C NOTE DOF IS SIMPLY THE NODE NUMBER IF NOT ON Z=0 PLANE
    ndof = 0
    do i=1, nodes
C NEUMANN NODES NOT ON Z=0 PLANE
        if (zn(i).gt.eps) then
            ndof = ndof + 1
            dof(i) = ndof
C NEUMANN NODES NOT ON Z=0 PLANE
        else if(zn(i).lt.neps) then
            ndof = ndof + 1
            dof(i) = ndof
C DIRICHLET NODES ON Z=0 PLANE
        else
            dof(i) = 0
        end if
    end do

C INITIALIZE ALL COMPONENTS OF GLOBAL STIFFNESS MATRIX
    call initAglobal(ndof)

C CONSTRUCT RHS MATRIX, FIRST INITIALIZE ALL PARTS OF ARRAY
    t13 = rtc()
    do 19 i=1, nodes
        rhs(i) = 0
19    continue
C CONSTRUCT RHS MATRIX, NOTE INHERENT MAGNETIZATION IS ZERO
    do 15 el=1,elems
        if (elflag(el).eq.1) then
            do 17 i=1, gpts
                do 16 lnn=1, nodper
                    nn = lnlist(lnn,el)
                    rhs(nn) = rhs(nn)+wg(i)*detjac*(mu-munot)*
+ (hxe(i)*dxder(i,lnn)+hye(i)*
+ dyder(i,lnn)+hze(i)*dzder(i,lnn))
16                continue
17            continue
        end if
15    continue
    t14 = rtc()
    print*, 'TIME TO CONSTRUCT RHS(SEC):', t14-t13
C CONSTRUCT LOCAL STIFFNESS MATRIX ALOCAL
C FIRST INITIALIZE ALL PARTS OF ARRAY
    t15 = rtc()
    do 34 i=1,nodper
        do 35 el=1,elems
            Alocal(el,i) = 0

```

```

35      continue
34      continue
C NEXT CONSTRUCT LOCAL COMPONENTS OF STIFFNESS MATRIX
      do 36 el=1,elems
C          Alocal(j,i) = 0
          do 37 g=1, gpts
              do 38 i=1, nodper
                  do 39 j=i, nodper      !ONLY DO DIAGONAL AND UPPER HALF
FOR EFF
                      Alocal(j,i) = Alocal(j,i)+mu*wg(g)*detjac*
                      +          (dxder(g,i)*dxder(g,j)+
                      +          dyder(g,i)*dyder(g,j)+
                      +          dzder(g,i)*dzder(g,j))
39          continue
38          continue
37          continue
C DO LOOP TO CONSTRUCT GLOBAL RHS MATRIX
      do 56 i=1,nodper
          idof = lnlist(i,el)
          do 57 j=1,nodper
              jdof = lnlist(j,el)
              if (dof(idof).eq.0) then
                  if (dof(jdof).eq.0) then
                      go to 57
C IF BOTH DOF=0 THEN STAYS THEN DO NOTHING
                      else
C MOVE A LOCAL COMPONENT TO OTHER SIDE SINCE IT IS KNOWN
C WOULD ALSO ADD PHI VALUE BUT ZERO IN OUR CASE
                          rhs(jdof) = rhs(jdof)-Alocal(j,i)
                      end if
                      else if (dof(jdof).eq.0) then
C MOVE A LOCAL COMPONENT TO OTHER SIDE SINCE IT IS KNOWN
C WOULD ALSO ADD PHI VALUE BUT ZERO IN OUR CASE
                          rhs(idof) = rhs(idof)-Alocal(j,i)
                      else
C NOW ADD NONZERO DOF COMPONENTS TO GLOBAL STIFFNESS MATRIX
C                          call add_to_Aglobal(idof,jdof,Alocal(j,i))
                      end if
57          continue
56          continue
36          continue
          t16 = rtc()
          print*, 'TIME TO CONSTRUCT LOCAL STIFFNESS MATRIX
          + A(SEC):', t16-t15

C DO LOOP TO CHECK THE VALUES OF THE GLOBAL RHS MATRIX
C      open(unit=35,file='GlobalRHS2.dat',status='NEW')
C      open(unit=36,file='RemovedGRHS2.dat',status='NEW')
C      write(35,221)
C      write(36,221)
C221  format('Node',3x,'GLOBAL RHS')
C      do 333 n=1, nodes
C          if (dof(n).eq.0) then
C              write(36,222) n,rhs(n)
C          else if (dof(n).ne.0) then
C              write(35,222) n,rhs(n)
C          end if

```

```

C222      format(i4,2x,1pe10.2)
C333      continue

C FILE/DO LOOP TO STORE/CHECK ONE ELEMENTS LOCAL STIFFNESS MATRIX A
C      open(unit=29,file='Alocal.dat',status='NEW')
C      do 44 i=1,nodper
C          do 45 j=1,nodper
C              write(29,170) Alocal(j,i)
C170      format(1x,1pe10.2)
C45      continue
C44      continue

      t2 = rtc()
      print*, 'TOTAL TIME(SEC):', t2-t1

C IS A NODE IN AIR, IRON, OR ON BORDER? USE FLAGS TO TELL
C INITIALIZE NDFLAG FOR ALL NODES TO NEGATIVE ONE
      do 31 i=1,nodes
          ndflag(i) = -1
31      continue

      do 32 el=1, elems
          if(elflag(el).eq.0) then
              iflag = 0
          else if (elflag(el).eq.1) then
              iflag = 2
          end if
          do 33 lnn=1, nodper
              nn = lnlist(lnn,el)
              if (ndflag(nn).eq.-1) then
                  ndflag(nn) = iflag
              else if (ndflag(nn).ne.iflag) then
                  ndflag(nn) = 1
              end if
33      continue
32      continue

C OPEN AND WRITE INTO FILE TO HOLD RHS AND STIFFNESS DATA
C      open(unit=24,file='ARHS.dat',status='NEW')
C      write(24,70)
C70      format(1x,'Node',6x,'RHS',3x,'A/I/INT')
C      do 30 nn=1,nodes
C AIR NODES
C          if (ndflag(nn).eq.0) then
C              write(24,40) nn, rhs(nn)
C40      format(i4,2x,1pe10.2,2x,'AIR')
C IRON NODES
C          else if (ndflag(nn).eq.2) then
C              write(24,50) nn, rhs(nn)
C50      format(i4,2x,1pe10.2,2x,'IRON')
C NODES ON AIR IRON BORDER
C          else if (ndflag(nn).eq.1) then
C              write(24,60) nn, rhs(nn)
C60      format(i4,2x,1pe10.2,2x,'INTERFACE')
C          end if
C30      continue

```

```

C      close(24)
C      close(23)
C      close(25)
C      close(27)
C      close(28)
C      close(29)
C      close(35)
C      close(36)
C      return
C      end

C*
C*****
)
C*

C SUBROUTINE TO DBL CHECK SIZE OF GLOBAL STIFFNESS MATRIX A
C AND TO SET ALL A COMPONENTS TO ZERO
      subroutine initAglobal(ndof)
      include 'femdata.inc'
      integer i,j

C DBL CHECK TO MAKE SURE MATRIX IS BIG ENOUGH
      if (ndof.gt.nmax) then
        print*, 'ERROR MATRX TOO SMALL'
        stop
      end if

C INITIALIZE ALL COMPONENTS OF GLOBAL STIFFNES MATRIX
      do i=1, ndof
C SINCE A IS SYMMETRIC ONLY NEED TO CONSTRUCT/STORE
        diag(ndof) = 0
        do j=1, nzmax
          upper(j,i) = 0
          colidx(j,i) = 0
        end do
      end do
      return
      end

C*
C*****
*)
C*

C SUBROUTINE TO CONSTRUCT GLOBAL STIFFNESS MATRIX A
      subroutine add_to_Aglobal(i,j,localA)
      include 'femdata.inc'
      integer i,j
      real*8 localA

C DBL CHECK ROW COLUMN INDICIES TO ENSURE CORRECT VALUES
      if (i.eq.0.or.j.eq.0) then
        print*, 'ERROR I/J =0 !'
        stop
      else if (i.gt.ndof.or.j.gt.ndof) then
        print*, 'ERROR I/J > DOF!'

```

```

        stop
    end if

C CONSTRUCT AND STORE DIAGONAL PARTS OF A MATRIX
C THAT ARE NOT ZERO
    if (i.eq.j) then
        diag(i) = diag(i) + localA
        return
    end if

C CONSTRUCT AND STORE OFF-DIAGONAL TERMS OF A MATRIX
C THAT ARE NOT ZERO; ONLY NEED UPPER PART BC A IS SYMMETRIC
    if (i.ne.j) then
        irow = min(i,j)
        jcol = max(i,j)
        do j=1, nzmax
            if (colidx(j,irow).eq.jcol) then
                upper(j,irow) = upper(j,irow) + localA
                return
            else if (colidx(j,irow).eq.0) then
                upper(j,irow) = localA
                colidx(j,irow) = jcol
                return
            end if
        end do
    end if
    return
end

C*
C*****
)

```

B. MD-GRAPE Accelerator Board

Some of the basic and more useful programming calls to implement the GRAPE board for a Coulomb force calculation are featured in Table B.1. Also a description of the necessary parameters is included in Table B.2.

Table B.1: Description of useful GRAPE board calls with their necessary parameters.		
Function	Parameters	Description
m3_allocate_unit	fname,mode,xmin,xmax,int,dum	Allocates appropriate GRAPE board unit for calculation
m3_set_positions	unit,xj,nj	Sets positions of particles
m3_set_charges	unit,qj,nj	Sets charges of particles
m3_calculate_forces	unit,xi,ni,force	Calculates forces between particles
m3_free_unit	unit	Releases board unit when calculation is finished

Table B.2: Type and description of parameters from Table B.1.		
Parameter	Type (Input: I/Output: O)	Description
fname	Character (I)	File name of a table function
mode	Integer (I)	Force or potential mode
xmin	Real (I)	Minimum x value in simulation
xmax	Real (I)	Maximum x value in simulation
dum	Integer (I)	Dummy variable
unit	Integer (I)	Used to set return value from allocation
xj	Real (I)	2D array to hold x,y,z coordinates of j-th particle
nj	Integer (I)	Number of j-particles
qj	Real (I)	Array to hold charge of j-th particle
xi	Real (I)	2D array to hold x,y,z coordinates of i-th particle
ni	Integer (I)	Number of i-particles
force	Real (O)	2D array to hold x,y,z components of calculated force on i-th particle

X. Resources

- Bedrosian, G. "Implementation of the Biot-Savart Calculation on the MDGRAPE-3 Board." Allegheny College, Meadville, PA. November 2008.
- Bedrosian, G. "Note on Shape Functions for Finite Element Analysis." Allegheny College, Meadville, PA. July 2008.
- Bedrosian, G. "Scalar Potential Formulation for Magnetostatic Finite Element." Allegheny College, Meadville, PA. October 2008.
- Bedrosian, G. "Shape Functions and Integration Formulas for Three-Dimensional Finite Element Analysis." International Journal For Numerical Methods in Engineering Vol. 35 (1992): 95-108.
- Bondeson, Anders, Pär Ingelström, and Thomas Rylander. Computational Electromagnetics. New York, NY: Springer Science+Business Media, Inc., 2005.
- Chari, Dr. M. V. K., et al. Finite Elements in Electrical and Magnetic Field Problems. Eds. M. V. K. Chari and P. P. Silvester. New York, NY: John Wiley and Sons, Ltd., 1984.
- Griffiths, David J. Introduction to Electromagnetics. 3rd ed. Upper Saddle River, NJ: Prentice Hall, Inc., 1999.
- Kong, J. A. Ed. Progress in Electromagnetics Research: Computational Electromagnetics and Supercomputer Architecture. Cambridge, MA: EMW Publishing, 1993.
- Leestma, Sanford and Larry Nyhoff. Fortran 77 for Engineers and Scientists. 4th ed. Upper Saddle River, NJ: Prentice Hall, Inc., 1996.
- User Manuals for the MD-GRAPe board and various Fortran programming techniques.
- Various emails between Professor Gary Bedrosian and Professor Jamie Lombardi, Allegheny College.
- Wikipedia: The Free Encyclopedia. "Galerkin Method." Wikipedia.org 23 July 2008. 4 November 2008 <http://en.wikipedia.org/wiki/Galerkin_method>.

Various FORTRAN programs provided by Professor Gary Bedrosian, Allegheny College, and Professor Jamie Lombardi, Allegheny College, for the following:

Program Name	Description
SHAPE1D	1D FEM shape/derivative functions
SHAPE2D	2D FEM shape/derivative functions
SHAPE3D	3D FEM shape/derivative functions
GAUSS3D	FEM Gaussian integration points
findnode	Finds local coordinates for nodes in elements, also finds local nodes of element sides
H3D	Module for calculating 3D Biot-Savart Law
B3D	Computes 3D nonlinear magnetostatic field for shielded magnets
B3D.INC	Include file for B3D with common data
HcScan.f	Determines coil field from stored points (w/ and w/out GRAPE board)
HcScan.inc	Include file for HcScan.f with common data (w/ and w/out GRAPE board)
MakeFEM.f	Makes a FE grid using FEMData.f module; the grid is a 3D cube of iron inside a cube of air
FEMData.f	Simple FORTRAN module for data storage of MakeFEM.f program
femdata.inc	Include file for FEMData.f file
IOUtil.f	FORTRAN module for IO utilities
ScanTest	Program to exercise HcScan.f (w/ and w/out GRAPE board)
rtc.c	Program to time routines with a program