Numerical Computations For Pde Models Of Rocket Exhaust Flow In Soil

Brian Brennan
University of Central Florida

NUMERICAL COMPUTATIONS FOR PDE MODELS OF ROCKET EXHAUST FLOW IN SOIL

by

BRIAN W. BRENNAN
B.S. Florida State University, 2007

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Mathematics
in the College of Sciences
at the University of Central Florida
Orlando, Florida

Summer Term
2010

Thesis Adviser: Brian E. Moore

ABSTRACT

We study numerical methods for solving the nonlinear porous medium and Navier-Lame problems. When coupled together, these equations model the flow of exhaust through a porous medium, soil, and the effects that the pressure has on the soil in terms of spatial displacement. For the porous medium equation we use the Crank-Nicolson time stepping method with a spectral discretization in space. Since the Navier-Lame equation is a boundary value problem, it is solved using a finite element method where the spatial domain is represented by a triangulation of discrete points. The two problems are coupled by using approximations of solutions to the porous medium equation to define the forcing term in the Navier-Lame equation. The spatial displacement solutions can be used to approximate the strain and stress imposed on the soil. An analysis of these physical properties shows whether or not the material ceases to act as an elastic material and instead behaves like a plastic which will tell us if the soil has failed and a crater has formed. Analytical as well as experimental tests are used to find a good balance for solving the porous medium and Navier-Lame equations both accurately and efficiently.

ACKNOWLEDGMENTS

I would like to show my gratitude to Dr. Brian Moore, whose encouragement, guidance and support over the past year have helped me reach this point. I am very grateful for the countless hours he has spent reviewing many drafts of this thesis.

A large part of this thesis was building a reliable model that can assist with a larger project. So I must also thank the rest of our team, Kristina Kraakmo and Whitney Keith, who have been so supportive over the past year.

I would also like to thank the other members of my thesis committee, Dr. John Cannon and Dr. David Rollins, for their willingness to participate in this important process with me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 PROBLEM HISTORY AND MOTIVATION

Previous missions to the moon, specifically the Apollo 12 and Apollo 15 missions, have demonstrated the potential danger that can arise during landing [14]. Such complications come about from the pressure forced upon the surface by the rocket's exhaust. Inspection of the Surveyor III spacecraft following the Apollo 12 mission showed that while the lunar module landed approximately 200 meters away, the lander's exhaust was still powerful enough to effect the Surveyor III. It was determined that a high-speed sandblast hit the Surveyor III with particles traveling more than 100 m/s casting permanent shadows onto the materials. Microscopic dents were also found on the Surveyor III and it was determined that these were caused by soil particles which were estimated to be traveling between 300 and 2000 m/s [20].

The Apollo 15 landing encountered similar, but more severe problems. It was reported that the crew began seeing the blowing dust as early as 46 m above the surface and that by 18 m, the sandblast was strong enough to inhibit all visibility. The module eventually landed on the edge of a small crater. After a shaky landing, the module eventually found its balance along the edge of the crater but with one leg suspended in space. Fortunately the landing was left with enough stability for the crew to successfully and safely complete their mission.

The Apollo 12 and Apollo 15 landings demonstrated the potential for damage to nearby hardware as well as the lander itself from the blowing material [13]. The new lunar modules that are expected to be used in future lunar or Mars landings will likely be larger than those used in the Apollo missions. Larger modules require more thrust and that means the potential for complications such as those that occurred in the Apollo missions becomes greater. The next lunar module is expected to have ten times the thrust of the landers used in the Apollo missions, which will lead to a higher probability that the new lander will create its own crater when it lands.

In 1966 there was a NASA funded test program which studied the cratering process [23]. This study found that the erosion of the soil begins directly under the jet before quickly spreading. A test showed that for a 600 pound thrust, an initial crater formed of over 40 inches in diameter and 20 inches deep. It was also determined that this initial crater did not change much with time.

Physical properties of the soil, the thrust of the rocket as well as the atmospheric conditions of the environment in which the rocket is landing can impact the results of the landing. Our goal is to build a model that will allow us to accurately predict what combinations of these properties could result in cratering. An accurate model of the pressure due to the rocket's exhaust and how it changes with time is the first step. The pressure results can then be used in a forcing term for a separate model to predict cratering. The Apollo landings as well as a number of small scale tests can provide us with a few sets of data with known results to help test our model.

We consider two ways in which the rocket exhaust may cause the lunar soil to crater. Bearing capacity failure or $\boldsymbol{BCF}$ occurs when the force applied to the surface is too great and creates a depression in the soil. Diffusion-driven flow or $\boldsymbol{DDF}$ occurs when the rocket exhaust causes the soil to break up and shear [15].

Creating a diagram such as Figure (1.1) can help us to determine the proper conditions under which $\boldsymbol{BCF}$ and $\boldsymbol{DDF}$ will occur. If the boundaries between where $\boldsymbol{BCF}$ and $\boldsymbol{DDF}$ occur can be sufficiently determined for particular soil properties, then crater formations can be predicted for a given rocket thrust. This assumes that we have chosen a location where the physical properties of the soil are well known.

We are not only interested in the moon. There has also been a lot of research done as far as studying the atmospheric conditions of Mars and how they effect a possible landing. A study of potential Mars soil erosion due to an imposed pressure is studied in [12]. The true

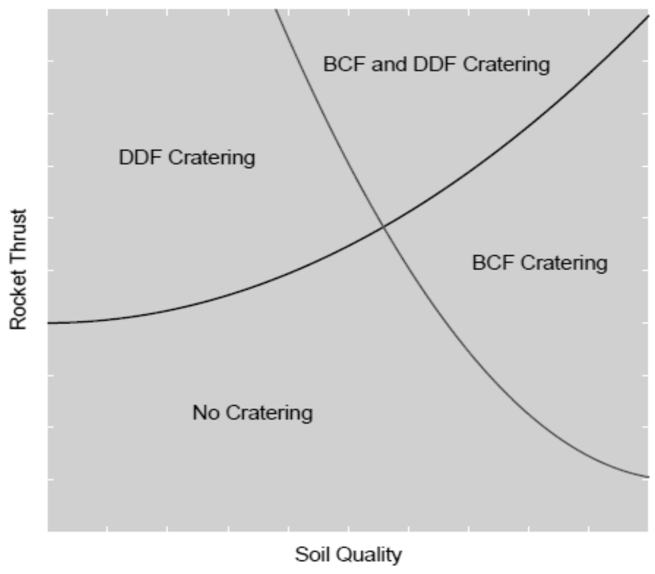

Figure 1.1: An example of the types of results we expect to achieve with the goal of defining the boundaries separating each crater type.

value in our model is that it can be applied to any environment that has been sufficiently examined. Assuming we know the physical properties such as density and porosity of a given surface as well as the thrust of the rocket, then our model can be used.

1.2 THE PROBLEM

The goal is to accurately and efficiently model the application of gas diffusion as a body force and determine if the soil can support such a force without cratering. The source of the body force is the pressure imposed on the soil due to the rocket exhaust during landing. This pressure may result in the displacement on the surrounding soil. With an approximation of the displacement field in hand, additional analysis can be done to determine if the soil can withstand the pressure without cratering. If the pressure imposes enough stress of the soil, then the soil will no longer act like an elastic material. It will instead act as a plastic material which can break down rather than bend and this is when craters are formed.

To model the pressure we start with The Porous Medium Equation [9],

$$\frac{\partial u}{\partial t} = \Delta u^m, \quad m > 1 \tag{1.1}$$

which models gas flow in a porous medium. This is a general, parabolic partial differential equation. In order to use (1.1) to model the pressure in our system, we must incorporate some fluid mechanics as in [3]. Specifically, Darcy's law which states that the rate at which a fluid flows through a permeable substance per unit area is equal to the permeability times the pressure per unit length of flow divided by the viscosity of the fluid. When derived from Darcy's Law the porous medium equation takes the form,

$$\frac{\partial p}{\partial t} = \frac{k}{2\eta\varepsilon}\Delta p^2 \tag{1.2}$$

where p is the pressure, η is the viscosity of the gas, while k and ε are the permeability and porosity of the medium, respectively.

Navier's model for volume displacement is used for modeling the reaction of the soil to the pressure and is defined by,

$$\mu\nabla^2\boldsymbol{u} + (\mu + \lambda)\nabla(\nabla \cdot \boldsymbol{u}) + f = 0, \tag{1.3}$$

where μ and λ are material constants and $\boldsymbol{u}$ is the displacement vector field. For our two dimensional problem we let $\boldsymbol{u} = (u, v)$ where u is the x displacement and v is the y displacement. The function f is the body force of the material and is defined by $f = \rho g + \nabla p$. Here we have the constants g and ρ which are the acceleration due to gravity and the density of the material, respectively. The variable p is the pressure field found by solving the porous medium equation. Our choice of modeling the solution with a static, two dimensional equation means that we can only determine if a crater has formed. The width and depth of the crater as well as the overall displacement of the soil would require a time dependent model in three spatial dimension. Our goal is to use the computationally simpler static problem as a foundation for future work in building a more complete model.

To solve the porous medium equation, we have chosen to use a finite difference method for time. The idea behind a finite difference scheme is to approximate a continuous function by solutions at discrete points. If the difference between these points are small enough, then convergence of the method guarantees that the set of discrete values sufficiently approximates the exact solution. For the porous medium equation, this idea is used to handle the differentiation in time. For spatial discretization, we use a spectral spatial domain to ensure high accuracy and efficiency. This gives us a set of solutions where each is an approximation of the pressure at a particular instant in time. From these pressure approximations we can calculate the forces imposed upon the sand and move on to solving Navier's equation.

For Navier's equation we have chosen a different approach. Here we are implementing a finite element method, which is typical for a boundary value problem. Just as the finite difference method took a continuous function in time and approximated at discrete points, the finite element method takes a continuous function over the entire spatial domain and finds approximations on small subsections of the domain. These small subsections are known as elements, and the approximation on each element is used to build the entire solution. As the size of the elements decreases, the finite element approximation more closely resembles a continuous function and thus the accuracy of the approximation increases.

Calculating an approximation to the solution is not enough. We must also verify that our approximation is sufficiently accurate. It is often the case that the exact solution is not known. Thus a method for approximating the error must also be implemented. Richardson's error estimate is used for approximating the error in our model of the porous medium equation. Residual based, a posteriori error estimates can be effective methods for analyzing the accuracy of a finite element approximation. An analysis of such error estimates for finite element methods in the area of elasticity can be found in [5] and [22].

Both the porous medium equation and Navier's equation have been researched extensively. However it appears that the behavior of Navier's equation with a forcing term derived from Darcy's law has never been explored. The purpose of this paper is to develop and implement an accurate and efficient numerical algorithm for numerical study of the solution behavior.

2 THE POROUS MEDIUM EQUATION

2.1 THE CRANK-NICOLSON METHOD

Approximations of solutions to the porous medium equation are found in this thesis through discrete time stepping and a Spectral approximation of the Laplacian operator. The Crank-Nicolson method is a second order, implicit method derived from averaging the explicit and implicit Euler methods. An implicit method finds a solution to the problem by solving an equation that is dependent on both the current state of the system as well as the later one. The explicit method is simpler in that the new solution is dependent entirely on the former state of the system. This means that for explicit methods, an approximation to the solution at time step t_{k+1} can be found directly from the approximation at time t_k. On the other hand, an implicit method will lead to a system of solved equations.

An arbitrary partial differential equation can be defined as,

$$y_t(t,x) = f(y, \nabla y, \Delta y) = F(y)$$

where $y(t,x)$ and $f(y)$ are discrete vectors. The Crank-Nicolson method comes from discretizing $y(t,x)$ in time while taking the average of the right hand side evaluated at both the current and next time steps. Given that $y_k(t,x)$ is an approximation of $y(t_k,x)$ for each time step, the Crank-Nicolson method is defined by,

$$y_{k+1} = y_k + \frac{h}{2}[F(y_k) + F(y_{k+1})] \tag{2.1}$$

where h is a given time step size such that $t_{k+1} = t_k + h$. The dependency on the $F(y_{k+1})$ term makes (2.1) an implicit method. While implicit methods may take longer and can be more difficult to implement than an explicit method, the benefits of using an implicit method can far outweigh those draw backs for certain types of problems. Equation (1.2) is known as a stiff equation which is a differential equation for which solutions from explicit methods become numerically unstable unless the step size is small. We can see the value in using an implicit method over a less computationally expensive explicit method by finding approximations to the following partial differential equation,

$$\begin{cases} \frac{\partial y}{\partial t} = \Delta y \\ y(0,x) = \sin(x) \\ y(t,x) = 0, \quad x \in \Gamma \end{cases} \tag{2.2}$$

using both an explicit and implicit method. The Crank-Nicolson method is used for the implicit test. The Crank-Nicolson method is of second order which means that the error in the method is proportional to h^2. For the explicit test we would prefer to choose a method that is of the same order as the Crank-Nicolson method. This allows us to assume that any difference found in the numerical results of these methods is due entirely to the fact that the method is implicit or explicit. Heun's method is a second order explicit method that works well here. Huen's method is similar to the Crank-Nicolson method in that it essentially uses the average of the right hand side evaluated at both the current and next time steps. But in order to be an explicit method, the right hand side can not depend on $y_{k+1}(t,x)$. Thus the explicit Euler method is used to approximate $y_{k+1}(t,x)$ in terms of $y_k(t,x)$. Huen's method is defined as follows,

$$\begin{cases} \tilde{y}_{k+1} = y_k + hF(y_k) \\ y_{k+1} \approx y_k + \frac{h}{2}[F(y_k) + F(\tilde{y}_{k+1})] \end{cases} . \tag{2.3}$$

In Figure (2.1) we are comparing the error in the Crank-Nicolson method with that of Heun's method. Here we apply the Heun's and Crank-Nicolson methods to the partial differential equation defined by (2.2). For larger step sizes in time, we clearly see that the explicit Heun's method becomes unstable. However the implicit Crank-Nicolson method

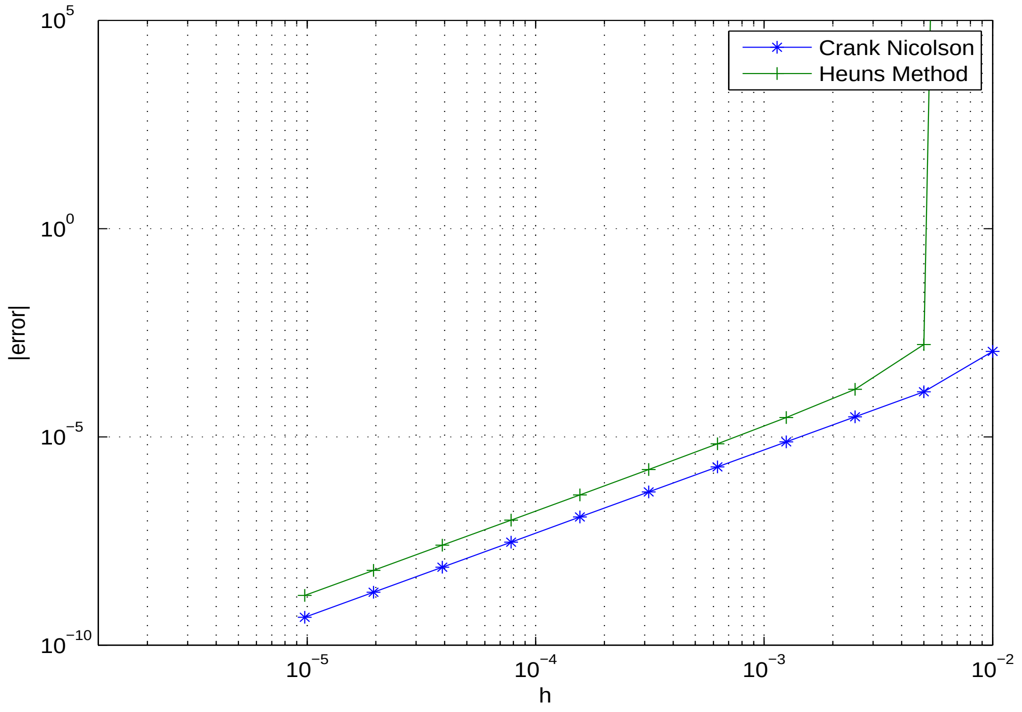

Figure 2.1: The error as functions of the time step h for the Crank-Nicolson and Heun's approximations to (2.2). Heun's method clearly becomes unstable for larger step sizes. This is the key characteristic of a stiff equation which implies that an implicit method is a better choice for this problem.

remains stable as the step size increases. As we decrease the step size, Heun's method becomes stable but the accuracy of the method is still worse than that of the Crank-Nicolson method for equivalent step sizes. This is important because the method runs over the time interval, $t = 0$ up until $t = T$ when T is the total time chosen by the user. The ability to achieve acceptable accuracy while using a larger step size allows us to span the time interval in fewer iterations which improves efficiency. Finding a balance between accuracy and efficiency is crucial in any numerical method.

If we define the discretized pressure by the matrix, $\boldsymbol{p}_k \in \mathbb{R}^{(N+1)\times(N+1)}$, then the vector $p_k \in \mathbb{R}^{(N+1)^2}$ can be used to represent the unknown pressure values in the system of equations. The column vector p_k is constructed by stacking each row of $\boldsymbol{p}_k$ on top of each other. The

Crank-Nicolson discretization of (1.2) is then defined by,

$$p_{k+1} \approx p_k + \frac{h\beta}{2}(\Delta_s p_k^2 + \Delta_s p_{k+1}^2) \tag{2.4}$$

where $\beta = \frac{k}{2\eta\varepsilon}$. An implicit method will result in a system of equations that will need to be solved. But more troublesome than that is the nonlinearity of this particular equation. The $\Delta_s p_{k+1}^2$ term makes it difficult to isolate p_{k+1} in equation (2.4). So we require another method for finding solutions. A root finding iteration such as Newton's method is a good choice for solving such a problem. However, before we can look into ways of finding solutions to (2.4), we must define the Δ_s matrix used to model the Laplacian operator at discrete points. Along with defining the Δ_s matrix, we look into the placement of these discrete nodes and how they effect the accuracy of the approximation in the following section.

2.2 SPECTRAL DIFFERENTIATION MATRICES

The spectral method is a global method. This means that the approximation at any point depends on every discrete point in the domain. This differs from finite difference methods where approximations are made based only on the local neighboring nodes. A global method such as this is of higher order and is potentially far more accurate, but there are problems that can arise with higher order interpolation such as this. Interpolating over equidistant nodes can lead to inaccurate results, especially in higher order interpolations. An example of this can be seen in Figure (2.2) which shows multiple interpolants of the function $f(x) = 1000\exp^{-\frac{x^2}{0.05}}$ using equidistant nodes.

The accuracy of the interpolant can be improved by clustering the nodes near the boundaries using the Chebyshev nodes. Doing so will distribute the error in the interpolant more evenly among all nodes rather than just at the end points.

Given a function $f(x)$ defined on $[-1, 1]$ and N nodes, $x_1, x_2, \ldots, x_N \in [-1, 1]$ then by

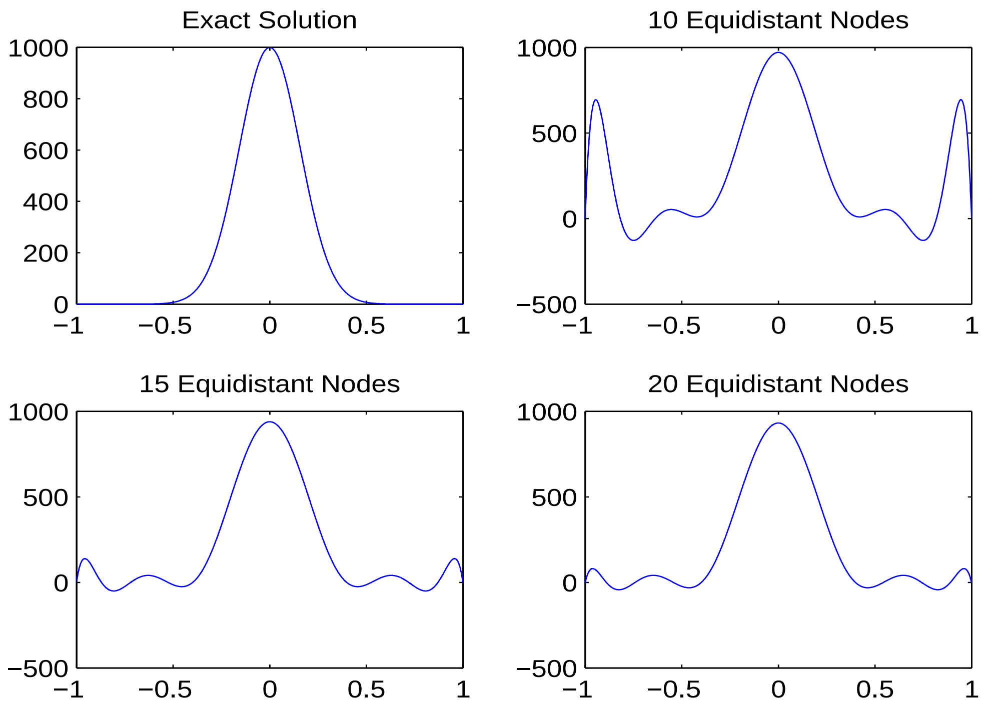

Figure 2.2: This figure shows a graph of the function, $f(x) = 1000\exp^{-\frac{x^2}{0.05}}$ as well as the three interpolants of degree 10, 15 and 20. Each interpolant was found using equidistant nodes.

polynomial interpolation we can approximate $f(x)$ by a unique polynomial, $P_N(x)$ of degree $N-1$ which coincides with $f(x)$ at each node. The error in such a polynomial interpolation is defined by,

$$f(x) - P_N(x) = \frac{f^{(N+1)}(\zeta)}{(N+1)!}\prod_{i=1}^{N}(x-x_i), \quad \zeta \in [-1,1] \tag{2.5}$$

Since we have no control over $f^{(N)}(\zeta)$, the idea is to minimize $\max\limits_{x\in[-1,1]}\left|\prod_{i=1}^{N}(x-x_i)\right|$. This product is a monic polynomial of degree N, meaning is has a leading coefficient of one. In this case, the maximum of the term $\max\limits_{x\in[-1,1]}\left|\prod_{i=1}^{N}(x-x_i)\right|$ is minimized when monic Chebyshev polynomials are used [4]. The Chebyshev polynomials are defined by,

$$T_N(x) = \cos(N \arccos(x)) \tag{2.6}$$

with roots defined by,

$$x_i = \cos(\frac{2i-1}{2N}\pi), \quad i = 0, 1, 2, \dots N \tag{2.7}$$

and these polynomials have the property,

$$\max_{x \in [-1,1]} |T_{N+1}(x)| = \frac{1}{2^N} \tag{2.8}$$

This means that if we chose an interpolating polynomial $P_N(x)$ defined on nodes, x_i, at the roots of $T_{N+1}(x)$, then

$$\max_{x \in [-1,1]} |f(x) - P_N(x)| \leq \frac{1}{2^N (N+1)!} \max_{x \in [-1,1]} |f^{N+1}(x)| \tag{2.9}$$

Due to the $\frac{1}{2^N}$ factor which comes from our choice of the Chebyshev nodes, (2.9) converges to zero much faster than (2.5) when any other set of nodes is used. This clearly makes the Chebyshev nodes far more attractive than standard equidistant nodes. The nodes as defined by (2.7) are spread over the interval $[-1, 1]$. This can easily be changed to any arbitrary interval $[a, b]$ by a the linear transformation,

$$x_i = \frac{1}{2}(a+b) + \frac{1}{2}\cos(\frac{2i-1}{2N}\pi) \tag{2.10}$$

in place of (2.7). In this thesis we let $a = 0$ and $b = 1$ to define our nodes over $x_i \in [0, 1]$. In Figure (2.3) we are again finding interpolants of the function $f(x) = 1000 \exp^{-\frac{x^2}{0.05}}$ but we are now using the Chebyshev nodes. In this case, the interpolants do converge as the degree of the polynomials increase. For this reason, we have chosen the Chebyshev nodes to define the grid on which we build the differential operators used to model the problem.

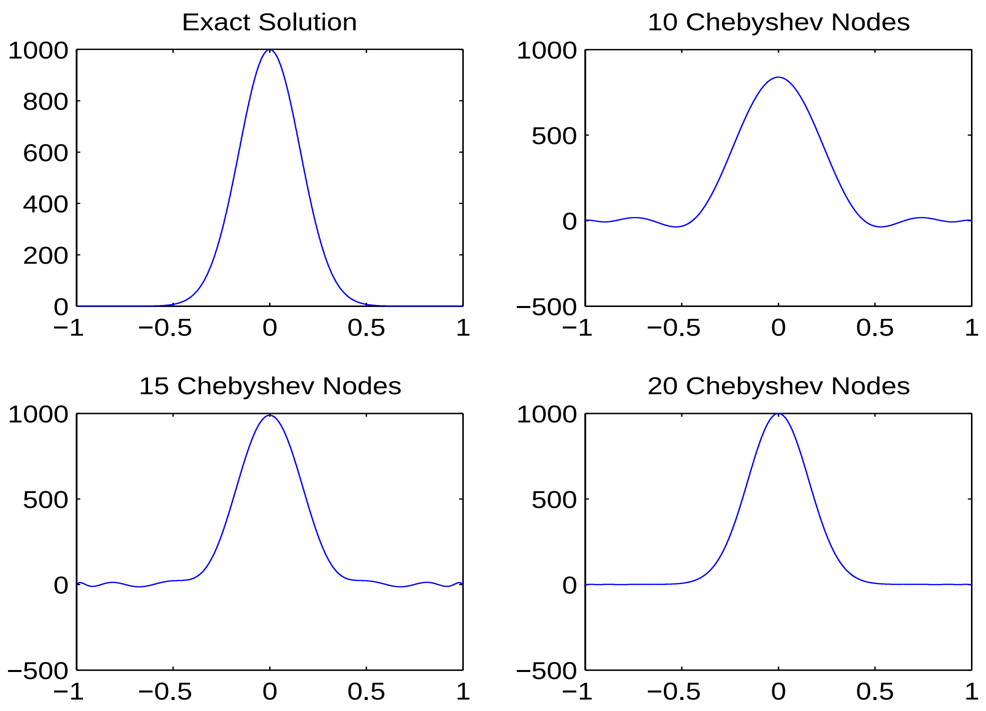

Figure 2.3: This figure shows a graph of the function, $f(x) = 1000\exp^{-\frac{x^2}{0.05}}$ as well as the three interpolants of degree 10, 20 and 30. Each interpolant was found using the clustered Chebyshev nodes. We can see that as the degree of the interpolant increases, the interpolant converges unlike for the equidistant case.

The Laplacian operator is approximated by,

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} \approx \Delta_s p_k \tag{2.11}$$

Here the continuous Laplacian of the pressure function is approximated by the spectral difference operator, Δ_s, acting on the discrete pressure vector at the k^{th} time step. For the two dimensional problem, p_k is originally defined as an $N \times N$ matrix. The idea is to have a matrix that can be multiplied by a column vector to approximate the Laplacian of the corresponding continuous function. When dealing with a problem defined over a single spatial dimension, we simply have to build a matrix to approximate the second derivative in space. For a problem defined over two spatial dimensions we let the rows of the matrix, p_k represent the x dimension and the columns represent the y dimension.

We let $D \in \mathbb{R}^{(N+1)\times(N+1)}$ be the Chebyshev spectral differentiation matrix defined by [21],

$$\begin{cases} (D_N)_{00} &= \frac{2N^2+1}{6}, \\ (D_N)_{NN} &= -\frac{2N^2+1}{6}, \\ (D_N)jj &= \frac{-x_j}{2(1-x_j^2)}, \qquad j = 1, \ldots, N-1, \\ (D_N)ij &= \frac{c_i}{c_j}\frac{(-1)^{i+j}}{x_i - x_j}, \qquad i \neq j, \quad i,j = 0, \ldots, N, \end{cases} \tag{2.12}$$

where

$$c_i = \begin{cases} 2, & i = 0 \quad N, \\ 1, & \text{otherwise.} \end{cases} \tag{2.13}$$

We now have a differential operator of the first derivative that can be used when p_k is a vector corresponding to a function defined on a single spatial variable. Because D operates on a column vector we have restructured the p_k matrix to be a single column vector of length $(N+1)^2$ where each row has been stacked one on top of the other. With this we can build the Δ_s operator for when p_k is a two dimensional surface with the aid of Kronecker products.

If D is a differential matrix for the one dimensional first derivative, then the second partial derivative across each row can be calculated using a block diagonal matrix. This matrix is constructed with D^2 along the diagonal. This new matrix multiplied by our pressure vector of stacked rows, results in D^2 being multiplied by each row as desired. If we let R_i represent the transpose of the i^{th} row of p_k, then we have

$$\frac{\partial^2 p}{\partial x^2} \approx \begin{pmatrix} D^2 & 0 & 0 & 0 & \dots & 0 \\ 0 & D^2 & 0 & 0 & \dots & 0 \\ 0 & 0 & D^2 & 0 & \dots & 0 \\ \vdots & \vdots & & \ddots & & \vdots \\ 0 & 0 & \dots & 0 & D^2 & 0 \\ 0 & 0 & \dots & 0 & 0 & D^2 \end{pmatrix} \begin{pmatrix} R_1 \\ R_2 \\ R_3 \\ \vdots \\ \vdots \\ R_{N+1} \end{pmatrix} = \begin{pmatrix} D^2 \cdot R_1 \\ D^2 \cdot R_2 \\ D^2 \cdot R_3 \\ \vdots \\ \vdots \\ D^2 \cdot R_{N+1} \end{pmatrix} \tag{2.14}$$

A set up such as this can be constructed by simply taking the Kronecker product $I \otimes D^2$ and it is then no surprise that the second partial derivative across each column is the Kronecker product $D^2 \otimes I$. This is an $(N+1)^2 \times (N+1)^2$ block structure matrix composed of diagonal submatrices with the corresponding elements of D^2 along each subdiagonal. This separates the elements of D^2 so that they match up with elements of the p_k vector which correspond to those in the same column of the $\boldsymbol{p}_k$ matrix.

$$\frac{\partial^2 p}{\partial y^2} \approx \begin{pmatrix} d_{00}I & d_{02}I & d_{03}I & \dots & d_{0N}I \\ d_{20}I & d_{22}I & d_{23}I & \dots & d_{2N}I \\ d_{30}I & d_{32}I & d_{33}I & \dots & d_{3N}I \\ \vdots & \vdots & & \ddots & \vdots \\ d_{N0}I & \dots & \dots & \dots & d_{NN}I \end{pmatrix} \begin{pmatrix} R_1 \\ R_2 \\ R_3 \\ \vdots \\ R_{N+1} \end{pmatrix} = \begin{pmatrix} D^2 \cdot R_1 \\ D^2 \cdot R_2 \\ D^2 \cdot R_2 \\ \vdots \\ D^2 \cdot R_{N+1} \end{pmatrix} \tag{2.15}$$

where d_{ij} is the $(i,j)^{th}$ element of D^2. This gives us,

$$\begin{cases} \frac{\partial^2}{\partial x^2} & \approx & I \otimes D^2 \\ \frac{\partial^2}{\partial y^2} & \approx & D^2 \otimes I \\ \Delta_s & = & I \otimes D^2 + D^2 \otimes I. \end{cases} \tag{2.16}$$

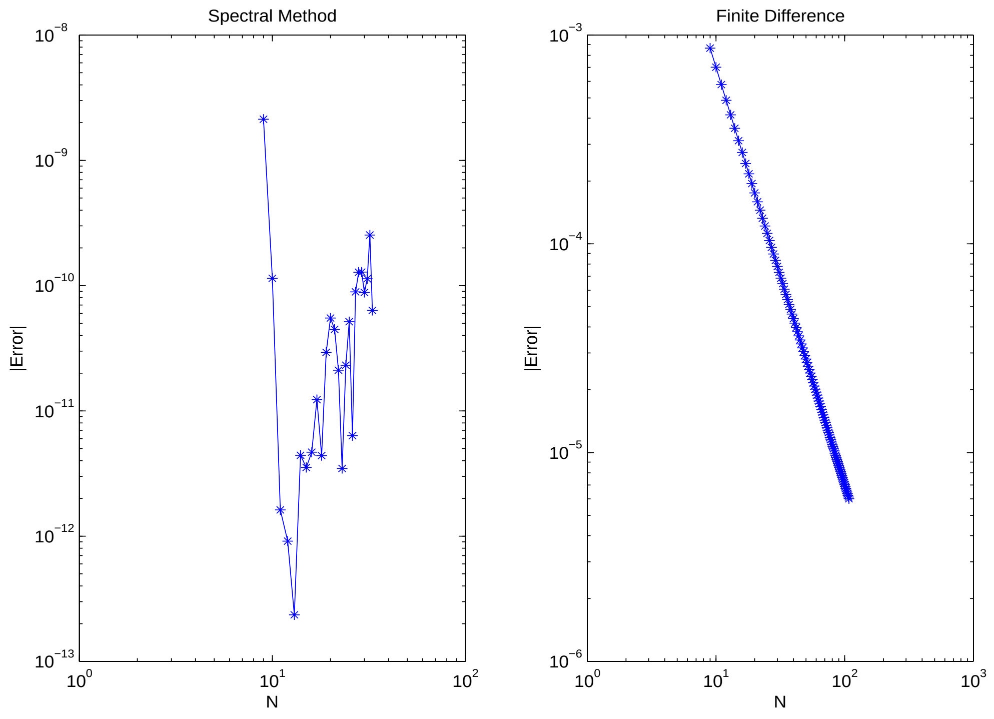

Figure 2.4: The second derivative of $\sin(x)$ is approximated using both Chebyshev and equidistant nodes. The error in each approximation are shown as functions of the number of nodes.

The value in using the Chebyshev nodes along with a spectral differentiation method can be seen in Figure (2.4) where we approximate the second derivative of the function $\sin(x)$. Here the second derivative is approximated using a spectral method at the Chebyshev nodes as well as the second order central difference method at equidistant nodes. We can see that for the spectral case, the error achieves a minimum value of nearly 10^{-13} when N is just over 10. On the other hand, the number of nodes for the equidistant case is taken out to 100 with the error reaching just slightly better than 10^{-5}. Unlike for Chebyshev nodes, the error appears to always decrease as the number of nodes increases for the equidistant case. But more nodes means that there are more equations to solve. Figure (2.4) shows us that choosing Chebyshev nodes allows us to use a relatively small amount of nodes while still achieving very good accuracy. Due to the second order nature of the central difference method, the convergence of the method is proportional to h^2. However it can be shown that the error in the spectral method is proportional to h^N [2].

2.3 NEWTON'S METHOD

Now that we have defined the differential operator in the method, we need a way of solving the nonlinear system of equations defined by (2.4). For all k the current solution, p_k is known while the following solution, p_{k+1} is unknown. Assuming we have a way of generating a sufficiently accurate initial guess for each p_{k+1}, Newton's method can be an efficient option.

Newton's method solves $F(z) = 0$ with the following iteration,

$$z_{i+1} = z_i - \frac{F(z_i)}{F'(z_i)}. \tag{2.17}$$

Newton's method is an iterative scheme that loops until the approximation satisfies,

$$\|z_{i+1} - z_i\|_\infty < \varepsilon \tag{2.18}$$

where ε is some user defined error tolerance. We let the variable z represent p_{k+1} in the loop until the relation (2.18) is satisfied. At that point we set $p_{k+1} = z$ and move on the next time step. To use Newton's method, we rearrange (2.4) and define the function $F(z)$ to be,

$$F(z) = z - [p_k + \frac{\beta h}{2}(\Delta_s p_k^2 + \Delta_s z^2)] \tag{2.19}$$

For the initial guess z_0, we need an explicit method so that an initial guess of p_{k+1} can be found directly and efficiently. Since the Crank-Nicolson method is of second order, we would like to not only chose an explicit method but one that is also of second order to approximate z_0. Huen's method can again be used to find such an initial guess. The function, $F'(z)$ is in fact the Jacobian matrix of $F(z)$ defined by,

$$J_F(z) = I - G'(z) \tag{2.20}$$

where I is the $N^2 \times N^2$ identity matrix and $G(z) = p_k + \frac{\beta h}{2}(\Delta_s p_k^2 + \Delta_s z^2)$.

For each $i = 0, 1, 2, \ldots N^2$,

$$G_i(z) = p_k + \frac{\beta h}{2}\Delta_s p_k^2 + \frac{\beta h}{2}\sum_{j=1}^{N^2} \Delta_s^{ij} z^2 \tag{2.21}$$

Here $\Delta_s^{i,j}$ is the $(i,j)^{th}$ element of the Δ_s matrix. This leads to the definition,

$$G_i'(z) \approx \frac{\partial G_i(z)}{\partial z(j)} = \beta h \Delta_s^{ij} z(j) \tag{2.22}$$

where $z(j)$ is the j^{th} element of the N^2 dimensional column vector z. The complete matrix form of $G'(z)$ is defined by,

$$G'(z) \approx \beta h \cdot \begin{pmatrix} \Delta_s^{0,0} z(1) & \Delta_s^{0,2} z(2) & \ldots & \ldots & \Delta_s^{0,N^2} z(N^2) \\ \Delta_s^{2,0} z(1) & \Delta_s^{2,2} z(2) & \ldots & \ldots & \Delta_s^{2,N^2} z(N^2) \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ \Delta_s^{N^2,0} z(1) & \Delta_s^{N^2,2} z(2) & \ldots & \ldots & \Delta_s^{N^2,N^2} z(N^2) \end{pmatrix} \tag{2.23}$$

Using Heun's method from (2.3) as an initial guess gives us the following iteration for solving the nonlinear system at each time step,

$$\begin{cases} z_0 & = & p_k + \frac{\beta h}{2}[\Delta_s p_k^2 + \Delta_s (p_k + \beta h \Delta_s p_k^2)^2] \\ z_{i+1} & = & z_i - J_F^{-1}(z_i) F(z_i), \quad \text{while}(\|z_{i+1} - z_i|_\infty > TOL) \end{cases} \tag{2.24}$$

When we reach a point where the relation (2.18) is satisfied then we can set p_{k+1} equal to z_{i+1}, move on to the next time step and repeat. We have chosen the error tolerance, TOL, to be 10^{-12} in our code.

The following pseudo code demonstrates how the porous medium equation is solved using Newton's method in MATLAB.

```
$P_0 = \exp^{-(\frac{(x_i-0.5)^2+(y_i-0.5)^2}{\sigma^2})}$;          % Define initial pressure by a Gaussian
for $k = 1 : h : T$ do
    $z_0 = P_{k-1}$;
    Find $z_1$ from Heun's method
    while Error > TOL do
        Solve: $J(z_i)\delta_i = F(z_i)$
        $z_{i+1} = z_i + \delta$;
        Error = $|\delta_i|$;
    end while
    $P_{k+1} = z_{i+1}$;          % Update $P_t$ with the $z_{i+1}$ Newton approximation
end for
```

2.4 ERROR AND CONVERGENCE ANALYSIS

A key step in testing any method is analyzing the error and convergence properties of the approximation. Despite the fact that we are solving a nonlinear equation, we can study the convergence properties of the linear case in order to gain a better understanding of how the Crank-Nicolson method behaves. Doing so can help with determining a step size that will allow us to find a good balance between the accuracy and efficiency of our method. First, we remember that the Crank-Nicolson discretization of the linear porous medium equation is,

$$p_{k+1} = p_k + \frac{\beta h}{2}(\Delta_s p_k + \Delta_s p_{k+1})$$

and after collecting like terms we get the following relation,

$$(I - \frac{\beta h}{2}\Delta_s)p_{k+1} = (I + \frac{\beta h}{2}\Delta_s)p_k$$

Isolating the p_{k+1} term leads to the following sequence,

$$p_{k+1} = (I - \frac{\beta h}{2}\Delta_s)^{-1}(I + \frac{\beta h}{2}\Delta_s)p_k. \tag{2.25}$$

By mathematical induction we can see that for (2.25), $p_k \to 0$ as $k \to \infty$ if and only if,

$$\|(I - \frac{\beta h}{2}\Delta_s)^{-1}(I + \frac{\beta h}{2}\Delta_s)\|_\infty < 1 \tag{2.26}$$

Similarly for Heun's method, (2.3), we have,

$$\begin{aligned} p_{k+1} &= p_k + \frac{\beta h}{2}\Delta_s[p_k + (p_k + \beta h \Delta_s p_k)] \\ &= [I + \frac{\beta h}{2}\Delta_s + \frac{(\beta h)^2}{2}\Delta_s^2]p_k \end{aligned}$$

which implies that Heun's method is stable if,

$$\|I + \beta\frac{h}{2}\Delta_s + \frac{(\beta h)^2}{2}\Delta_s^2\|_\infty < 1 \tag{2.27}$$

From Figure (2.5) we can see that (2.26) is satisfied for any step size h. This is because the Crank-Nicolson method is unconditionally stable. We also see that as h becomes greater than approximately 10^{-4}, the explicit Heun's method no longer satisfies (2.27). This means that Huen's method is only conditionally stable.

Techniques such as these are quite accurate for the linear problem. However, we are looking to solve the nonlinear porous medium equation, and thus we require a more practical

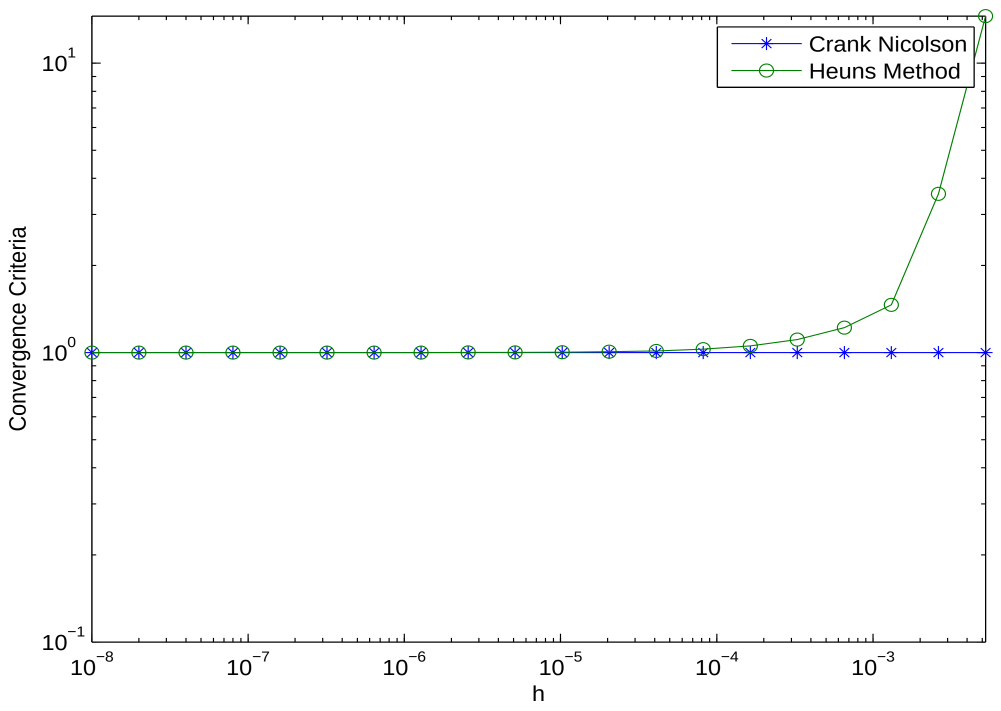

Figure 2.5: The norms of equations (2.26) and (2.27) as functions of the time step h, with $\beta = 1$.

way of analyzing higher order equations. By Taylor's theorem we have that,

$$p = p_k(h) + a_m h^m + O(h^{m+1}) \tag{2.28}$$

where

$$p = \lim_{h \to 0} p_k(h), \tag{2.29}$$

Here p is the exact solution and $p_k(h)$ is the numerical approximation of the solution with a time step size of h. In order to analyze the error in p, we require a second approximation of our solution with a different time step.

$$p = p_k(2h) + 2^m a_m h^m + O(h^{m+1}) \tag{2.30}$$

Subtracting (2.30) from 2^m times (2.28) and neglecting higher order terms gives us,

$$(2^m - 1)p \approx 2^m p_k(h) - p_k(2h) \tag{2.31}$$

which leads to Richardson's error estimate of $p_k(h)$ defined by,

$$p - p_k(h) \approx \frac{p_k(h) - p_k(2h)}{2^m - 1}. \tag{2.32}$$

Since our method is known to be of second order, we let $m = 2$ and now have a method for approximating the error in our numerical solution using a time step size of h,

$$p - p_k(h) \approx \frac{p_k(h) - p_k(2h)}{3}. \tag{2.33}$$

Being of second order means that the error is $O(h^2)$. A log-log plot of error approximations of our method for several values of h should be a straight line of slope two. The average slope between points for the Crank-Nicolson approximation in Figure (2.1) is roughly 2.00012 while the error of the method is tending to 10^{-8} for the reasonable time step size of $h = 10^{-4}$. These facts give us sufficient confidence that the method is being implemented properly.

2.5 POROUS MEDIUM EQUATION IN MATLAB

We have now covered the finite difference and spectral methods in great detail. However, understanding a mathematical method or operation and translating it into a programming language such as MATLAB are two completely separate tasks. Appendix A shows the MATLAB m-file that approximates the pressure as described here in this chapter. In this section however, we will go into detail explaining each step.

The first step is to build the spectral matrix, M, for the first derivative. This is done using the following algorithm for the first derivative approximation as defined in [21],

```
x = ((cos(pi*(0:N)/N)' + ones(N+1,1))/2);

c = [2; ones(N-1,1); 2].*(-1).^(0:N)';

X = repmat(x(1:N+1,1),1,N+1);

dX = X-X';

m = (c*(1./c)')./(dX + (eye(N+1)));

m = m - diag(sum(m'));

M = m;

M(1,:) = 0;

M(N+1,:) = 0;
```

Our problem uses homogeneous Dirichlet boundary conditions, which are enforced by $\mathrm{M}(1,:) = 0$ and $\mathrm{M}(\mathrm{N}+1,:) = 0$. These lines simply set the first and last rows of the spectral matrix to zero which forces the endpoints to be zero in the one dimensional case. Since M is now an approximation of the first derivative, the second derivative is approximated by M^2 or $\mathrm{M} * \mathrm{M}$ in MATLAB. Kronecker products are used to build the two dimensional Laplacian approximation from M. The matrix D in the our code represents the Δ_s operator defined in this thesis and is built by,

```
D = kron(I,D) + kron(D,I);
```

With the spectral matrix D now defined for our problem, we next look to define the initial pressure by a Gaussian.

```
u = zeros(N+1,N+1);
```

```
Sig = 0.1;
for i = 2:N
    for j = 2:N
        u(i,j)= Amp*exp(-(((x(i,1) - 0.5).^2)+(x(j,1) - 0.5).^2)/(Sig^2));
    end
end
P(:,1) = Mat2Vec(u);
```

Here we define an $(\mathrm{N}+1) \times (\mathrm{N}+1)$ matrix u of all zeros and then populate the interior nodes by a Gaussian function. The variable Sig controls the width of the Gaussian curve and can be adjusted by the user. The width of the curve is important as it mimics the size of the nozzle for a given rocket. The amplitude of the initial pressure is set using the variable Amp. This defines the magnitude of the initial thrust from the rocket. Once the initial pressure is defined over the two dimensional domain by the matrix u, it must be translated into a vector. Doing so allows us to multiply by the spectral matrix D to approximate the Laplacian. A short program, Mat2Vec, was written to handle this conversion. Using this program, the line $\mathrm{P}(:,1) = \mathrm{Mat2Vec(u)}$; defines the initial pressure to be the first column of the matrix P.

Next we implement the Crank Nicolson time stepping method using Newton's method at each time step to solve the system of equations that occur as a result of using an implicit method. Newton's method takes a dummy variable, z, updates it through a number of iterations until it approximates the exact solution sufficiently. So before the Crank-Nicolson loop begins, we define the initial value of the z variable to be the initial pressure, $\mathrm{P}(:,1)$. The entire Crank-Nicolson loop is then defined in our MATLAB program as follows.

```
%% Crank Nicolson Method
z = P(:,1);
```

```
k = 2;

for i = h:h:T

    w = z;

    z = w + (h/2)*beta*D*(w.^2 + (w + h*beta*D*w.^2).^2);  % Huen's guess

    e = z - w - (h/2)*beta*D*(w.^2 + z.^2);                % Initial error

    error = norm(e,inf);


    % Newton's Method

    count = 0;

    while (error > 1e-12 && count < 50)

        for j = 1:size(z,1)

            Z(j,:) = z';

        end

        J = eye(size(z,1),size(z,1)) - h*beta*D.*Z;       % Define Jacobian

        z = z - J\(z - w - (h/2)*beta*D*(w.^2 + z.^2));   % Newton iteration


        e = z - w - (h/2)*beta*D*(w.^2 + z.^2);           % Update Error

        error = norm(e,inf);

        count = count+1;

    end

    P(:,k) = z;    % Add next pressure approximation

 k = k+1;

end
```

First we should notice that the code loops from time $i = h$ up to T in step sizes of h. These values are not necessarily integers. The iterator, i, can not be used as the index for updating the pressure matrix. The variable k is used instead and is originally set to a value of two since the first column of P is already used for the initial Gaussian. The Crank-Nicolson method depends on both the current solution as well as the next. Here the current solution is known while the next solution is what we hope to approximate. The variable w is used to hold the value of the current solution. The Newton iteration is then started using Huen's method as an initial guess of the next solution based solely on the current solution, w. The Newton iteration is a conditional while loop which runs as long as the error in our approximation, z, of the next solution is greater than 10^{-12}. A counter is also placed in the code to stop the Newton iteration after fifty loops. This helps to avoid encountering an infinite loop. The key computational task in implementing Newton's method is building the Jacobian matrix, J, which is defined by (2.20). This definition is dependent on the matrix $G'(z)$ defined by (2.23). Reviewing the definition of $G'(z)$ shows us that we simply need to calculate the element by element multiplication between the Δ_s matrix and Z. The matrix Z is simply a matrix having the approximation vector z for every row. Using the Jacobian, J, the Newton approximation z is updated according to equation (2.17). From there we can calculate the error in the new approximation and if need be, continue the loop. Once the error is small enough and the conditions of the while loop are no longer met, the latest approximation vector z is stacked onto the pressure matrix as the k^{th} column.

We now have an approximation for the pressure at every time step. However the purpose of finding the pressure was to use it in the definition of the forcing function for Navier's equation which depends on the gradient of the pressure. Thus we need the first partial derivatives of the pressure in both x and y at every time step. The spectral matrix M is used for this. Again we use Kronecker products to change this one dimensional derivative approximation into an approximation of the two dimensional operator.

```
Px = kron(I,M)*P;
```

```
Py = kron(M,I)*P;
```

Here Px and Py are matrices where each column represents the partial derivative of the pressure at a given time step in the x and y directions respectively. These two matrices are what must be passed on to the finite element method program used to solve Navier's equation.

3 NAVIER'S MODEL

3.1 THE FINITE ELEMENT METHOD

The second equation solved in this thesis is the Navier-Lamé problem defined by (1.3). The Navier-Lamé equation models the displacement field of an elastic medium. To solve this problem we are implementing a finite element method. The finite element method is based on the idea of building a complicated object out of smaller or simpler blocks. In mathematics, this means breaking a complicated problem into smaller more manageable ones. In our case, we are splitting the domain of interest into smaller pieces called elements. The problem then becomes a system of equations built from solutions on each element.

The finite element method differs from the finite difference method used in chapter two in that we look for a solution to the variational or weak form of the problem rather than solve the strong form of the problem [11]. The variational problem involves multiplying by a smooth function and integrating the differential equation over a particular domain of interest. For the finite element method, the domain is broken up into subdomains or finite elements. Here we have chosen triangular elements to represent the domain. Triangles offer a minimal number of nodes required to represent a two dimensional element. This greatly simplifies calculations throughout the method. The solution on each triangle is approximated by a simple, linear polynomial. The choice of linear polynomials simplifies the problem by minimizing the number of nodes required on each element, as well as making the partial derivatives of each function a constant. The solution to the variational problem is then just the sum of the approximations on each of these triangles.

The Galerkin method is used, as in [8], to construct the weak form of the problem which is solved instead of (1.3). The Galerkin method is used on problems which involve solving for an unknown function. In general, the Galerkin method can be applied to any problem of the form,

$$Lu = f \tag{3.1}$$

where L is a linear operator, f is a given function, and u is the unknown function. By selecting a set of linear basis functions $\varphi_1, \varphi_2, \ldots \varphi_{N_n}$, where N_n is the number of nodes, we can approximate the function u by a suitable linear combination,

$$\boldsymbol{u}_h = \sum_{j=1}^{N_n} \varphi_j d_j \tag{3.2}$$

where d_j are the coefficients of the polynomial approximation. Due to the linearity of L, we then have,

$$\sum_{j=1}^{N_n} d_j L\varphi_j = f. \tag{3.3}$$

Solutions to (3.3) can be found using linear functionals $\Phi_1, \Phi_2, \ldots \Phi_{N_n}$ by imposing the condition,

$$\Phi_i\big(\sum_{j=1}^{N_n} d_j L\varphi_j - f\big) = 0 \qquad 1 \leq i \leq N_n \tag{3.4}$$

and again by linearity this becomes,

$$\sum_{j=1}^{N_n} \Phi_i(L\varphi_j) d_j = \Phi_i(f) \qquad 1 \leq i \leq N_n. \tag{3.5}$$

The classical Galerkin method uses the inner product, $\Phi_i(v) = (\varphi_i, v)$. Thus (3.5) becomes,

$$\sum_{j=1}^{N_n} d_j(\varphi_i, L\varphi_j) = (\varphi_i, f). \tag{3.6}$$

Equation (3.6) defines the system of equations that are solved in order to approximate solutions to (1.3). The system (3.6) can then be written in the form,

$$Ad = b \tag{3.7}$$

where A is known as the stiffness matrix, d is the unknown vector and b is called the load vector. The Galerkin method can be applied to a wide variety of problems however in our case, the stiffness matrix A is used to approximate a differential operator and b approximates the forcing term.

For our particular problem, the approximate solution $\boldsymbol{u}_h$ is first found on a square grid broken up into smaller squares of length h. The triangular elements are formed by simply splitting these smaller squares across their diagonal. Such a grid is the simplest form of the Dalaunay triangulation [10] as shown in Figure (3.1).

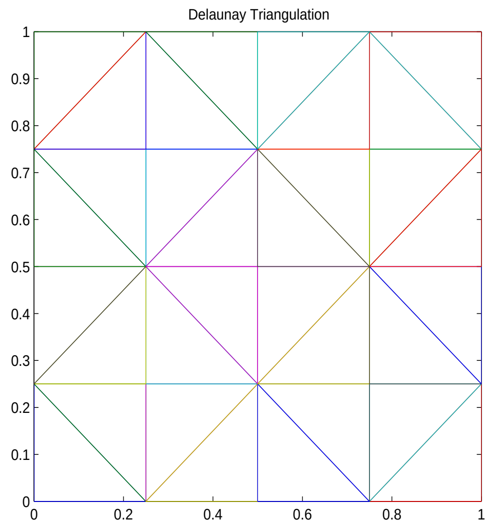

Figure 3.1: An example of the simple uniform mesh via a Delaunay triangulation.

The Dalaunay triangulation comes from the field of computational geometry. The Dalaunay triangulation takes a set of points and connects them in such a way that no point lies inside the circumcircle of any triangle where the circumcircle is the unique circle which passes through all three points of the triangle [17]. An interesting fact about the Dalaunay triangulation is that such an algorithm will construct the edges connecting each node such that the minimum angle of all triangles is maximized. This allows us to avoid thin triangles, which is helpful because it will offer a more uniform mesh that will spread the error around rather than localize it onto particular elements.

Implementation of the finite element method requires three key steps:

- Preprocessing
- FEM Solver
- Postprocessing

Preprocessing involves creating data structures to organize discrete locations of the nodes, as well as operators to define the equation that is to be solved. The finite element method solver will assemble global operators based on local operators on individual elements using the data structures created during preprocessing. Once the global operators are found, the problem becomes solving a simple linear system of equations. In postprocessing the solution is sorted, displayed and analyzed.

Preprocessing

- Let V be the space of piecewise polynomial functions and construct a finite-dimensional subspace $V_h \subset V$ as follows:
  - ⋆ Approximate the boundary Γ with a polygonal curve.
  - ⋆ Make a triangulation of the spatial domain Ω by subdividing Ω into a set $T_h = \{K_1, K_2, \ldots, K_{N_E}\}$ of non-overlapping triangles K_i, such that no corner of any triangle lies on the edge of another triangle,

$$\Omega = \cup_{K \in T_h} K = K_1 \cup K_2 \cdots \cup K_{N_E} \tag{3.8}$$

where N_E is the number of elements.

  - ⋆ Introduce the mesh parameter h as follows: $h = \max_{K \in T_h} diam(K)$, where diam(K) is longest side of K.

  - ⋆ Define $V_h \subset V$ to be the set of functions v_h such that: $V_h = \{v_h : v_h$ is continuous on $\Omega, v_h|_K$ is linear for $K \in T_h, v_h = 0$ on $\Gamma\}$ where $v|_K$ denotes the restriction of v_h to triangle K. Here we have assumed homogeneous Dirichlet boundary conditions.

- Construct the following data structures which define the discrete mesh:

  - ⋆ Define the *Locations* matrix containing the grid coordinates of every node in the mesh.

  - ⋆ Define the *Elements* matrix that matches global nodes on the mesh to local nodes on a given element.

In Figure (3.2) we have a uniform Delaunay triangulation that can help see how the matrices *Locations* and *Elements* are built during preprocessing. The *Locations* matrix is the first two be created as it holds the coordinates of each node which are required for building the Delaunay triangulation. The *Locations* matrix for Figure (3.2) is shown in (3.9) where each row contains the x and y coordinate of a single node.

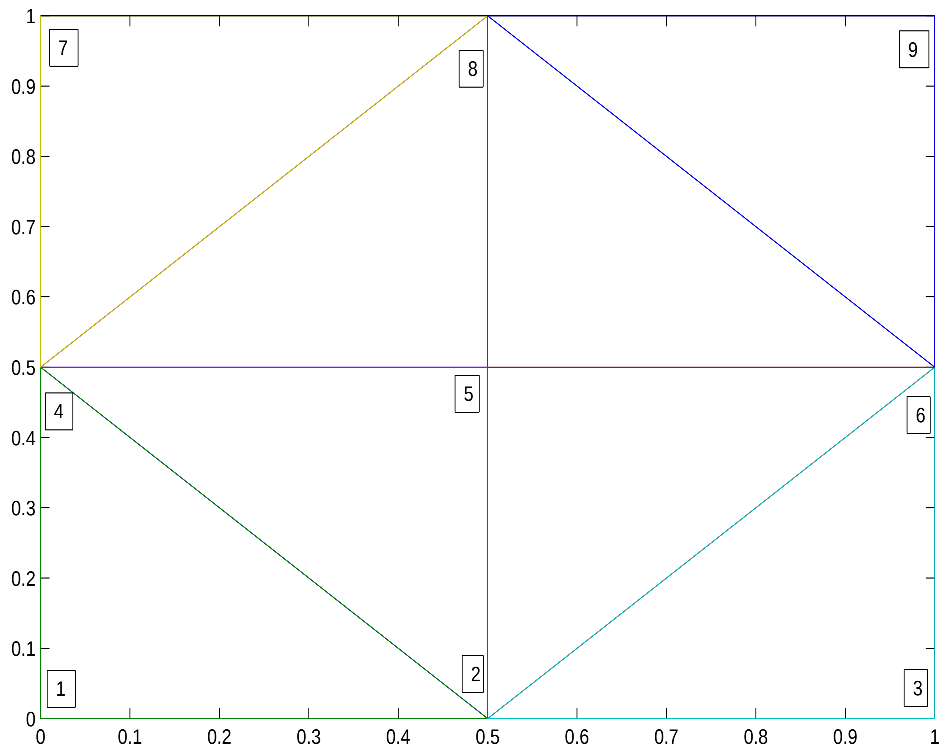

Figure 3.2: The uniform Delaunay triangulation defined from the nodal coordinates in the *Locations* matrix from (3.9).

$$Locations = \begin{bmatrix} 0 & 0 \\ 0.5 & 0 \\ 1 & 0 \\ 0 & 0.5 \\ 0.5 & 0.5 \\ 1 & 0.5 \\ 0 & 1 \\ 0.5 & 1 \\ 1 & 1 \end{bmatrix} \tag{3.9}$$

With the *Locations* matrix defined, we can construct the Delaunay triangulation. In MATLAB this is done by the command,

$$Elements = delaunay(x, y) \tag{3.10}$$

where x and y are the columns of the *Locations* matrix. This command results in the matrix defined by (3.11) where each row contains the three node numbers, as labeled in Figure (3.2), for a particular triangle.

$$Elements = \begin{bmatrix} 4 & 2 & 5 \\ 1 & 2 & 4 \\ 5 & 2 & 6 \\ 6 & 2 & 3 \\ 8 & 4 & 5 \\ 7 & 4 & 8 \\ 5 & 6 & 8 \\ 8 & 6 & 9 \end{bmatrix} \tag{3.11}$$

The node numbers listed in the *Elements* matrix refer to the node's corresponding row in the *Locations* matrix. Using *Locations* and *Elements* together allows us to quickly find the x and y coordinates for the three local nodes of a given triangle. These data structures together form the triangulation of our domain.

FEM Solver

- Input all required data: f, nodes, boundary conditions, physical constants.

- Compute the local stiffness matrices A^K and local load vectors b^K which are defined on individual triangles rather than the entire domain.

- Assemble the global stiffness matrix A and load vector b. These are built using the local stiffness matrices and local load vectors. This is the key idea behind the finite element method. We assemble an approximation of the solution on the entire domain by solving smaller, and likely easier, problems over a finite number of elements.

- Solve the system of equations $Ad = b$.

Postprocessing

- Plot the solution.

- Approximate the error in the finite element solution.

- Analyze the error in the solution under different conditions. Some combinations of physical constants or grid sizes may results in better approximations than others. An analysis of this can help improve the accuracy and efficiency of the code.

3.2 THE WEAK FORM

The first step in implementing any finite element method is deriving a variational, or weak form of the equation. Weak solutions are useful because many differential equations in real world models do not offer sufficiently smooth solutions. Thus the weak form must be used. Even when equations do have differentiable solutions, it is often far easier to prove existence of the weak solution and then show its equivalence to the strong solution.

We start with the strong form of the Navier-Lamé problem,

$$\mu\Delta\boldsymbol{u} + (\mu+\lambda)\nabla(\nabla\cdot\boldsymbol{u}) = \boldsymbol{f} \tag{3.12}$$

where μ and λ are known as the Lamé parameters. The forcing function $\boldsymbol{f}$ is dependent of the pressure and is defined by $\boldsymbol{f} = \rho g + \nabla p$ where ρ is the density of the soil, g is the acceleration due to gravity and p is the pressure. Both μ and λ are constants which depend on $\boldsymbol{E}$, Young's modulus and $\boldsymbol{v}$, Poisson's ratio. Young's modulus is a measure of the stiffness of an elastic material while Poisson's ratio is the ratio of the transverse contraction strain to the longitudinal extension strain. Young's modulus and Poisson's ratio for a given material are readily available in many standard tables. The constants μ and λ are defined by,

$$\lambda = \frac{\boldsymbol{vE}}{(1+\boldsymbol{v})(1-2\boldsymbol{v})}$$
$$\mu = \frac{\boldsymbol{E}}{2(1+\boldsymbol{v})}$$

If we take into account that $\boldsymbol{u} = (u,v)^T$ and $\boldsymbol{f} = (f_1, f_2)^T$, where $f_1 = \rho g + p_x$ and $f_2 = \rho g + p_y$, then (3.12) becomes,

$$\begin{pmatrix} \mu(u_{xx}+u_{yy}) + (\mu+\lambda)(u_{xx}+v_{yx}) \\ \mu(v_{xx}+v_{yy}) + (\mu+\lambda)(v_{yy}+u_{xy}) \end{pmatrix} = -\begin{pmatrix} f_1 \\ f_2 \end{pmatrix}$$

and after a slight rearrangement of terms we have,

$$\begin{pmatrix} (\lambda+2\mu)u_{xx}+\lambda v_{yx}+\mu(u_{yy}+v_{yx}) \\ (\lambda+2\mu)v_{yy}+\lambda u_{xy}+\mu(v_{xx}+u_{xy}) \end{pmatrix} = -\begin{pmatrix} f_1 \\ f_2 \end{pmatrix} \tag{3.13}$$

Factoring out the differential operator $(\frac{\partial}{\partial x}\ \frac{\partial}{\partial y})$ results in,

$$\begin{pmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \end{pmatrix} \cdot \begin{pmatrix} (\lambda+2\mu)u_x+\lambda v_y & \mu u_y+\mu v_x \\ \mu u_y+\mu v_x & (\lambda+2\mu)v_y+\lambda u_x \end{pmatrix} \tag{3.14}$$

If we define the stress tensor $\boldsymbol{\sigma}$ to be

$$\boldsymbol{\sigma} = \begin{pmatrix} (\lambda+2\mu)u_x+\lambda v_y & \mu u_y+\mu v_x \\ \mu u_y+\mu v_x & (\lambda+2\mu)v_y+\lambda u_x \end{pmatrix} \tag{3.15}$$

then (3.14) can be expressed as $\nabla\cdot\boldsymbol{\sigma}_{ij}$. From here we can multiply by a test function, $w \in V$ satisfying the given homogeneous Dirichlet boundary conditions and integrate over the domain, Ω, to get

$$\int_\Omega w_i \nabla\cdot\boldsymbol{\sigma}_{ij}\ d\boldsymbol{x} = -\int_\Omega w_i f_i\ d\boldsymbol{x}. \tag{3.16}$$

By the divergence theorem, $\int_\Omega \nabla\cdot\boldsymbol{\sigma}_{ij}\ d\boldsymbol{x} = \oint_\Gamma \boldsymbol{\sigma}_{ij}\cdot\boldsymbol{n}\ d\boldsymbol{x}$. Using this property, while applying integration by parts [7], results in

$$\int_\Omega w_{(i,j)}\boldsymbol{\sigma}_{ij}\ d\boldsymbol{x} = \int_\Omega w_i f_i\ d\boldsymbol{x} + \sum_{i=1}^{2}\Big(\int_{\Gamma_{h_i}} w_i h_i\ d\boldsymbol{\Gamma}\Big) \tag{3.17}$$

where

$$w_{(i,j)} = \frac{\frac{\partial w_i}{\partial x_j}+\frac{\partial w_j}{\partial x_i}}{2}, \tag{3.18}$$

and $h_i = \boldsymbol{\sigma}_{ij}\boldsymbol{n}_j$ ($\boldsymbol{n}$ is the unit outward normal vector to Γ). We must now introduce the

Hilbert space L_2 with inner product,

$$(w, u) := \int_\Omega wu \; d\boldsymbol{x}.$$

L_2 is the set of square integrable functions defined as,

$$L_2(\Omega) = \{v : v \text{ is defined on } \Omega \text{ and } \int_\Omega v^2 d\boldsymbol{x} < \infty\}.$$

Next, we can define the Sobolev space $H^1(\Omega)$ to be,

$$H^1(\Omega) = \{v : v \text{ and } v' \in L_2(\Omega)\}$$

But the Sobolev space $H^1(\Omega)$ is not quite enough. We also require that our test and basis functions satisfy the boundary conditions. Thus, we define the Sobolev space $H_0^1(\Omega)$ to be,

$$H_0^1(\Omega) = \{v \in H^1(\Omega) : v = 0 \text{ on } \Gamma_\Omega\}$$

Writing the problem in finite element form results in,

$$a(\boldsymbol{w}, \boldsymbol{u}) = (\boldsymbol{w}, \boldsymbol{f}) + (\boldsymbol{w}, \boldsymbol{h}) \tag{3.19}$$

where

$$\begin{aligned} a(\boldsymbol{w}, \boldsymbol{u}) &= \int_\Omega w_{(i,j)} \sigma_{ij} d\boldsymbol{x} \\ (\boldsymbol{w}, \boldsymbol{f}) &= \int_\Omega w_i f_i d\boldsymbol{x} \\ (\boldsymbol{w}, \boldsymbol{h}) &= \sum_{i=1}^{2} \Big(\int_{\Gamma_{h_i}} w_i h_i d\boldsymbol{\Gamma} \Big) \end{aligned}$$

Since we have chosen the space $V_h \subset H_0^1$ such that $\boldsymbol{u}$ and $\boldsymbol{w}$ are zero on the boundary, the third term here is zero. The variational problem now reads,

$$\text{Find } \boldsymbol{u} \in V_h \text{ such that:} \qquad a(\boldsymbol{w}, \boldsymbol{u}) = (\boldsymbol{w}, \boldsymbol{f}), \qquad \forall \boldsymbol{w} \in H_0^1.$$

We should note the fact that the integration by parts has eliminated all second derivatives. This means that solutions of (3.17) might have less continuity than those satisfying (3.12) and this is why it is known as the weak form.

3.3 STIFFNESS MATRIX

This stiffness matrix is a symmetric matrix which is a discretization of the left hand side of the weak form. The global stiffness matrix is defined as,

$$A_{ij} = \int_\Omega w_{(i,j)} \boldsymbol{\sigma}_{ij} \, d\boldsymbol{x} \tag{3.20}$$

In the FEM we use piecewise-polynomial approximations $\boldsymbol{u}_h$ of the exact solution $\boldsymbol{u}$. The approximate solution $\boldsymbol{u}_h$ is defined by,

$$\boldsymbol{u}_h = \sum_{j=1}^{N_n} \varphi_j(x, y) d_j. \tag{3.21}$$

The idea is to determine coefficients d_j such that $\boldsymbol{u}_h$ is an approximation of the exact solution $\boldsymbol{u}$. The coefficients d_j are the values of the solution at the j^{th} mesh node.

Next we need to define the strain vector, $\boldsymbol{\varepsilon}$, for two spatial dimensions to be,

$$\boldsymbol{\varepsilon}(\boldsymbol{u}) = \begin{pmatrix} u_x \\ v_y \\ u_y + v_x \end{pmatrix}. \tag{3.22}$$

Using this definition we can redefine the symmetric stress tensor, $\boldsymbol{\sigma}$, by ignoring everything below the main diagonal and store only the unique terms due to symmetry. This gives the stress vector $\boldsymbol{\sigma} = C\ \boldsymbol{\varepsilon(u)}$ or,

$$\begin{pmatrix} \boldsymbol{\sigma}_{11} \\ \boldsymbol{\sigma}_{22} \\ \boldsymbol{\sigma}_{12} \end{pmatrix} = \begin{pmatrix} \lambda + 2\mu & \lambda & 0 \\ \lambda & \lambda + 2\mu & 0 \\ 0 & 0 & \mu \end{pmatrix} \begin{pmatrix} \boldsymbol{\varepsilon_1}(\boldsymbol{u}) \\ \boldsymbol{\varepsilon_2}(\boldsymbol{u}) \\ \boldsymbol{\varepsilon_3}(\boldsymbol{u}) \end{pmatrix} \tag{3.23}$$

Here C is a matrix which is dependent entirely on the Lamé parameters and the elements of the stress tensor represent forces per unit area. We can now replace the integrand in (3.20) with $\boldsymbol{\varepsilon}(w) : C\boldsymbol{\varepsilon}(\boldsymbol{u})$ where $A : B = \sum_{j=1}^{3} A_j B_j$. Equation (3.20) now becomes,

$$A = \int_\Omega \boldsymbol{\varepsilon}(w) : C\boldsymbol{\varepsilon}(\boldsymbol{u})\ d\boldsymbol{x} \tag{3.24}$$

as defined in [1]. Since $w \in V$ and φ_i form a basis for the subspace $V_h \subset H_0^1$, we can restrict our attention to V_h by replacing the test function w by φ_i. Then by substituting (3.21) into (3.20) we have,

$$A_{i,j} = \sum_{j=1}^{N_n} \Big(\int_\Omega \boldsymbol{\varepsilon}(\varphi_i) : C\boldsymbol{\varepsilon}(\varphi_j) d\boldsymbol{x} \Big) d_j \tag{3.25}$$

The basis functions φ_j are chosen to be the linear hat functions defined by,

$$\varphi_j = a_j + b_j x + c_j y, \quad j = 1, 2, 3 \tag{3.26}$$

where the coefficients a_j, b_j and c_j are determined from the requirement,

$$\varphi_j(N_i) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \tag{3.27}$$

The definition of φ_j in (3.27) will be enforced on each element. This means that for each triangle in the mesh, we just have the three local basis functions $\varphi_1(x,y)$, $\varphi_2(x,y)$, and $\varphi_3(x,y)$. The coefficients, a_j, b_j, and c_j for the three basis functions can be found by solving the system of equations,

$$\begin{pmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{pmatrix} \begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad (3.28)$$

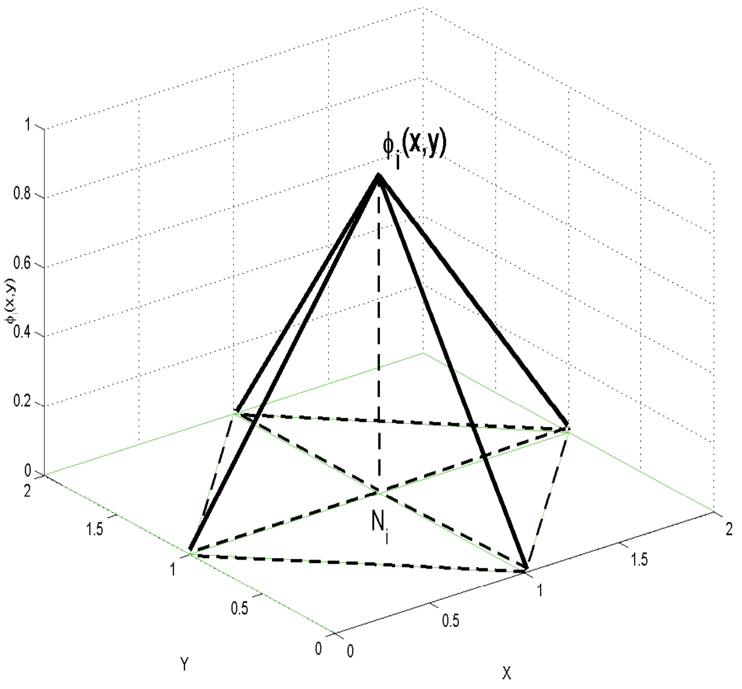

Figure 3.3: Linear hat function $\varphi_i(x,y)$ defined over the four triangular elements surrounding the node, N_i.

In Figure (3.3) we see an example of our particular choice of basis functions. This figure shows a plot of a single basis function, φ_i, over the entire domain. Clearly the only non-zero values occur over elements for which the node N_i is a corner. This is a direct result of equation (3.27). This implies that for any function φ_i, the only elements for which corresponding integrals will be non-zero are those for which N_i is a node. So the values of the stiffness matrix representing node N_i are only dependent on elements for which N_i is a node, and thus all other elements can be ignored. For this reason we now limit ourselves to building a local stiffness matrix for a single triangular element, K. Each time a new local stiffness matrix A^K is found the global stiffness matrix A can be updated. After all elements have been accounted for, the global stiffness matrix will be complete. If we define the global stiffness matrix to be $A(\varphi_i, \varphi_j)$, then for any element K the local stiffness matrix will be $A^K(\varphi_i, \varphi_j)$. If we let N_i, N_j and N_k be the vertices of the triangle K, then an example of a local stiffness matrix is

$$\begin{bmatrix} A^K(\varphi_i,\varphi_i) & A^K(\varphi_i,\varphi_j) & A^K(\varphi_i,\varphi_k) \\ A^K(\varphi_j,\varphi_i) & A^K(\varphi_j,\varphi_j) & A^K(\varphi_j,\varphi_k) \\ A^K(\varphi_k,\varphi_i) & A^K(\varphi_k,\varphi_j) & A^K(\varphi_k,\varphi_k) \end{bmatrix} \tag{3.29}$$

Since $A^K(\varphi_i, \varphi_j) = A^K(\varphi_j, \varphi_i)$, $\forall (i,j) \in [1, N_E]$, the stiffness matrix is symmetric.

The restriction of $\boldsymbol{u}_h$ to a single element K has the form,

$$\begin{cases} u_h^K = \varphi_1(x,y)d_1 + \varphi_2(x,y)d_3 + \varphi_3(x,y)d_5, & (x,y) \in K \\ v_h^K = \varphi_1(x,y)d_2 + \varphi_2(x,y)d_4 + \varphi_3(x,y)d_6, & (x,y) \in K \end{cases} \tag{3.30}$$

Since $\boldsymbol{u}_h^K = (u_h, v_h)$ we can approximate the strain tensor on a single triangle K as,

$$\begin{aligned}\boldsymbol{\varepsilon}(\boldsymbol{u}_h) = R_K\boldsymbol{u}_h^K &= \begin{pmatrix} \varphi_{K_1,x} & 0 & \varphi_{K_2,x} & 0 & \varphi_{K_3,x} & 0 \\ 0 & \varphi_{K_1,y} & 0 & \varphi_{K_2,y} & 0 & \varphi_{K_3,y} \\ \varphi_{K_1,y} & \varphi_{K_1,x} & \varphi_{K_2,y} & \varphi_{K_2,x} & \varphi_{K_3,y} & \varphi_{K_3,x} \end{pmatrix} \cdot \boldsymbol{u}_h^K \\ &= \begin{pmatrix} \varphi_{K_1,x}u_1 + \varphi_{K_2,x}u_2 + \varphi_{K_3,x}u_3 \\ \varphi_{K_1,y}v_1 + \varphi_{K_2,y}v_2 + \varphi_{K_3,y}v_3 \\ \varphi_{K_1,y}u_1 + \varphi_{K_1,x}v_1 + \varphi_{K_2,y}u_2 + \varphi_{K_2,x}v_2 + \varphi_{K_3,y}u_3 + \varphi_{K_3,x}v_3 \end{pmatrix} \\ &= \begin{pmatrix} u_{K,x} \\ v_{K,y} \\ u_{K,y} + v_{K,x} \end{pmatrix}\end{aligned}$$

which is equivalent to (3.22) on a single triangle K. Here $\varphi_{K_i,x}$ and $\varphi_{K_i,y}$ are the partial derivatives of φ at node i of triangle K with respect to x and y respectively. Equation (3.25) can then be approximated locally for a single element by,

$$A^K = \int_K R_K^T C R_K d\boldsymbol{x} \tag{3.31}$$

From the definition of φ_j in equation (3.26), we can see that $\varphi_{k_j,x} = b_j$ and $\varphi_{k_j,y} = c_j$. Since b_j and c_j are constants, the derivatives in the matrix R will simply be constants as well. This plus the fact that C is a constant matrix means that R^TCR can be factored out of the integral. Doing so leaves us with,

$$A^K = R_K^T C R_K \int_K d\boldsymbol{x} \tag{3.32}$$

where $\int_K d\boldsymbol{x}$ is simply the area of element K.

$$A^K = R_K^T C R_K \cdot (\text{ Area of Triangle K }) \tag{3.33}$$

There are multiple ways of calculating the area of the triangle. Since we have used a coarse mesh composed of identical right triangles, this task is trivial. But this is not always the case so other options may become useful. Simply entering the coordinates of each node into the MATLAB function *polyarea* is one option. Taking advantage of the matrices that we have already used in previous calculations of the stiffness matrix operators, another option is

$$\begin{aligned} \text{Area of Triangle K} &= \frac{1}{2}\begin{vmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{vmatrix} \\ &= \frac{1}{2}|T| \end{aligned} \tag{3.34}$$

where (x_i, y_i), $i = 1, 2, 3$ are the coordinates of the nodes for triangle K and T is the matrix defined in the equation. This gives us the equation that is actually implemented during the FEM Solver process of the code for each element in the mesh.

$$A^K = \frac{1}{2}|T|R_k^T C R_k \tag{3.35}$$

The resulting A^K is a 6×6 matrix where each element contributes to a different element of the global stiffness matrix A. The stiffness matrix setup ends with updating the global matrix with the proper elements from A^K for every K.

The following pseudo code demonstrates how the stiffness matrix is created in MATLAB.

```
**for** $k = 1$ to $N_E$ **do**
    Global = 2*Nodes([1,1,2,2,3,3]) - [1,0,1,0,1,0];
    Nodes = *Elements*(k,:);
    x = *Locations*(Nodes,1);
    y = *Locations*(Nodes,2);
    Construct R from x and y
    Construct T from x and y
    $A_k = \frac{1}{2}\det(T) \cdot R^T C R$;                % build local stiffness matrix
    A(Global,Global) = A(Global,Global) + $A_k$;     % Update global stiffness matrix
**end for**
```

3.4 LOAD VECTOR

The right hand side of equation (3.17) handles the forcing function as well as the boundary conditions. For the sake of simplicity of implementation, we have chosen homogeneous Dirichlet boundary conditions. Since the solution $\boldsymbol{u}$ is a displacement field, this choice in boundary conditions means that there is no displacement for particles on the boundary of our domain of interest. This is a reasonable assumption for a sufficiently large domain with a body force that is localized in the center of the domain. This implies that,

$$\sum_{i=1}^{2}\Big(\int_{\Gamma_{h_i}} w_i h_i \, d\boldsymbol{\Gamma}\Big) = 0 \tag{3.36}$$

and that only the first term on the right hand side of equation (3.17) needs to be calculated.

Just as we did for calculating the stiffness matrix, we are replacing the test function w with the linear hat functions φ, to get

$$\int_{\Omega} w_i f_i \, d\boldsymbol{x} = \int_{\Omega} \varphi_i f_i \, d\boldsymbol{x}, \tag{3.37}$$

where $f = \rho g + \nabla p$. Here p is the pressure approximated in chapter 2, ρ is the density of the soil, and g is the acceleration due to gravity. The gradient of the pressure, ∇p, as in,

$$\nabla p = \begin{pmatrix} \frac{\partial p}{\partial x} \\ \frac{\partial p}{\partial y} \end{pmatrix} \tag{3.38}$$

is what ultimately defines the strength of the forcing term. It is logical to expect that for larger initial pressures due to rocket exhaust, we will see larger displacement fields. The vector ∇p is composed of the first derivative of the pressure in the x direction as well as the y direction. These partial derivatives can be found using the Chebyshev spectral differentiation matrix D used in section 2.2 while again making use of the Kronecker product.

$$\begin{aligned} \frac{\partial p}{\partial x} &\approx (D \otimes I) \cdot p_k = P_x \\ \frac{\partial p}{\partial y} &\approx (I \otimes D) \cdot p_k = P_y \end{aligned}$$

It should be noted that these partial derivatives are defined on the Chebyshev nodes used in Chapter 2 and not on those used here for our finite element mesh. The *interp2* function in MATLAB performs the two dimensional interpolation necessary to efficiently define these values on the appropriate grid. The *interp2* function offers the choice of *nearest*, *linear*, *spline*, or *cubic* interpolation. Since the Crank-Nicolson method is of second order, we should use an interpolation option of order at least two. This leaves us with the *spline* or *cubic* options. Since we are not using a uniformly spaced grid, the *spline* option is chosen in our code.

Just as we did for calculating the stiffness matrix, the load vector calculation involves updating the global vector with the contributions from each individual element. The integral

$$\int_K \varphi_i f_i \, d\boldsymbol{x} \tag{3.39}$$

over a particular element K can be approximated using the three dimensional trapezoidal rule,

$$\int_K \varphi(x,y) \cdot f(x,y) d\boldsymbol{x} \approx \frac{\text{(Area of K)}}{3} \sum_{i=1}^{3} \varphi_i(x_i, y_i) \cdot f(x_i, y_i) \tag{3.40}$$

which is simply the average of the function evaluated at each node times the area of the base. By definition $\varphi_i(x_i, y_i) = 1$ for all i. Thus (3.40) simplifies to,

$$\int_K \varphi(x,y) \cdot f(x,y) d\boldsymbol{x} \approx \frac{\text{(Area of K)}}{3} \sum_{i=1}^{3} f(x_i, y_i) \tag{3.41}$$

Since $f(x,y) = \rho g + \nabla P$, the quadrature defined by (3.41) is calculated for both $f_1 = \rho g + P_x$ as well as $f_2 = \rho g + P_y$. These approximations represent the integrals over an entire element. These integrals are approximated over an entire element but what we care about is the contribution of these integrals to the three nodes of the triangle. If we assume an even distribution, we can simply divide these integrals by three and assign that value to each of the corresponding nodes.

We must remember that each node has an x and y displacement value and thus each node takes up two spots in our solution vector. So for a single element we have three nodes and thus a vector of length six,

$$\boldsymbol{u}_h^K = (u_1, v_1, u_2, v_2, u_3, v_3)^T \tag{3.42}$$

For example, the first node will be represented by the first and second elements of our solution vector, the second node takes the third and fourth, the third takes the fifth and six and so on. So we require a way of translating the node numbers into their corresponding solution vector locations. We can see from this pattern that the n^{th} node is represented by the $2n-1$ and $2n$ locations of the solution vector. This means that the values for the x displacement take the odd locations and the y displacement goes in the even locations. If *Nodes* is a vector containing the assigned node numbers, the command

$$\mathit{Global} = 2 * Nodes([1,1,2,2,3,3]) - [1,0,1,0,1,0]; \tag{3.43}$$

in MATLAB will give us the six locations in our solution vector for the three nodes of a particular triangle.

The following pseudocode demonstrates how the stiffness matrix is created in MATLAB.

```
for $k = 1$ to $N_E$ do
    Nodes = Elements(k,:);
    Global = 2*Nodes([1,1,2,2,3,3]) - [1,0,1,0,1,0];
    x = Locations(Nodes,1);
    y = Locations(Nodes,2);
    area = polyarea(x,y);

    $bK_x = \frac{1}{3}\sum_{i=1}^{3} P_x(Nodes) + \rho g$;
    b(Global([1,3,5])) = b(Global([1,3,5])) + $bK_x \cdot area/3$;

    $bK_y = \frac{1}{3}\sum_{i=1}^{3} P_y(Nodes) + \rho g$;
    b(Global([2,4,6])) = b(Global([2,4,6])) + $bK_y \cdot area/3$;
end for
```

3.5 ERROR ANALYSIS

The Richardson extrapolation technique used in section **2.4** to analyze and approximate the error in the finite difference method can also be applied to the finite element method so long as we limit ourselves to a uniform mesh.

The *a priori* interpolation error estimate is again of the form,

$$\boldsymbol{u} - \boldsymbol{u}_h^m = C_{m+1}h^{m+1} + O(h^{m+2}) \tag{3.44}$$

This applies to a polynomial approximation of degree m with a mesh spacing h. Just as before we require two approximations for our estimate, one for grid spacing h and a second for a grid spacing of $2h$. So along with (3.44), we need

$$\boldsymbol{u} - \boldsymbol{u}_{2h}^m = C_{m+1}(2h)^{m+1} + O(h^{m+2}) \tag{3.45}$$

where $m = 1$ due to our choice of linear basis functions. Setting m equal to one and subtracting (3.45) from (3.44) leads to,

$$\boldsymbol{u}_{2h} - \boldsymbol{u}_h = -3C_2h^2 \tag{3.46}$$

or

$$C_2h^2 = \frac{\boldsymbol{u}_h - \boldsymbol{u}_{2h}}{3} \tag{3.47}$$

after neglecting higher order terms. From (3.44) we have that C_2h^2 is an approximation of $\boldsymbol{u} - \boldsymbol{u}_h$. Thus, we have an estimate of the discretization error of the uniform mesh solution as

$$\boldsymbol{u} - \boldsymbol{u}_h = \frac{\boldsymbol{u}_h - \boldsymbol{u}_{2h}}{3} \tag{3.48}$$

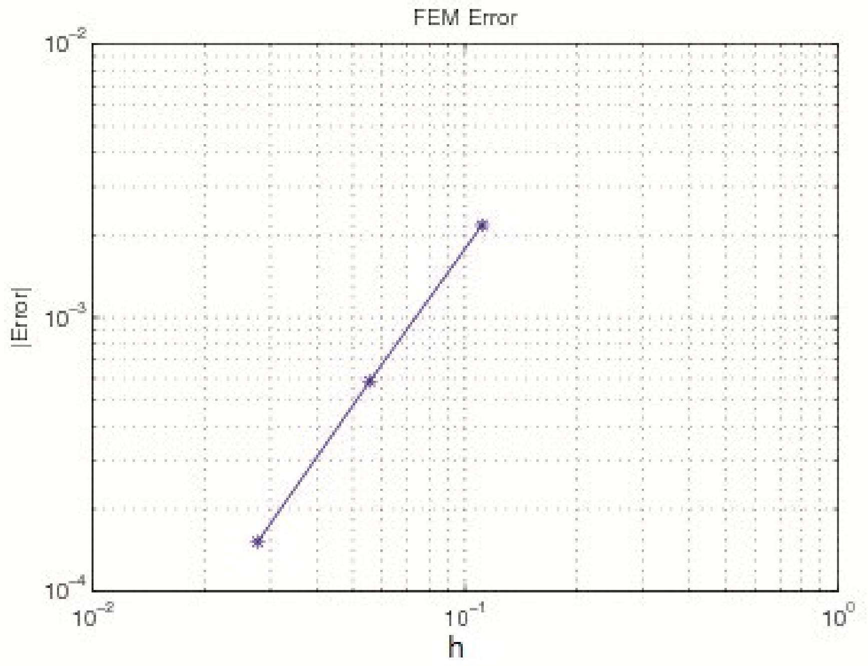

Figure 3.4: A log-log plot of the error as a function of the grid size h for the finite element method. The slope of the line is 1.951. As the grid size h decreases, this value appears to converge to exactly 2.

The equation $\boldsymbol{u} - \boldsymbol{u}_h = C_2 h^2$ also tells us something about the behavior of the error. If we create a log-log plot of the error approximation for multiple mesh sizes, we would expect to again see a line of slope 2.

Unfortunately equation (3.48) is not very practical for approximating the error in a finite element approximation, because uniform meshes such as the one shown in (3.1) are very rarely used. When some nodes no longer lie on a uniform mesh, a new method for approximating the error is required. Rather than approximating the error through a comparison of solution approximations from different grid sizes, we can calculate the residual of our approximation. The residual is the difference between the right and left sides of the strong equation when our approximate solution is used. If we let $A(\boldsymbol{u}_h) = b$ be our finite element equation as before, then $\hat{A}(\boldsymbol{u}) = \hat{b}$ represents the strong equation defined by (3.12) where

$$\begin{cases} \hat{A} = \mu\nabla^2 + (\mu + \lambda)\nabla\nabla^T \\ \hat{b} = -f \end{cases} . \tag{3.49}$$

The residual is then defined to be,

$$R(\boldsymbol{u}_h) = |\hat{A}(\boldsymbol{u}_h) - \hat{b}| \tag{3.50}$$

and the new error estimate will be dependent on this residual as well as the size of the grid spacing, h, which is again the largest diameter among all elements in the mesh. The new error estimate is defined by [6],

$$\|e\| \leq S_c C_i \|h^2 R(\boldsymbol{u}_h)\|_\infty \tag{3.51}$$

where S_c is the stability factor and C_i is an interpolation constant. The stability constant S_c is problem dependent while the interpolation constant C_i does not rely on the particular problem. C_i depends entirely on properties such as the shape of the elements, the order of the basis functions, the norms used in approximating the error and so on. Since S_c and C_i are constants, we can say that the error is proportional to $h^2 R(\boldsymbol{u}_h)$. This means that

$$\|e\| \propto \|h^2 R(\boldsymbol{u}_h)\|_\infty \tag{3.52}$$

and thus, $h^2 R(\boldsymbol{u}_h)$ can be used to estimate the error. The result of this equation is a column vector that contains the error approximation on each node in the mesh. A measure of the error on a particular element can be found by summing the errors on the three nodes that make up that triangle.

3.6 ADAPTIVE REFINEMENT

Now that we have a way of measuring the error in an approximation found on a non-uniform grid, we can use more complicated meshes. Or rather, we can adjust the mesh in an attempt to minimize the error in the approximation. We begin by finding a solution on the uniform mesh and approximating the error on each element using (3.52). The error on each element is then compared to a given error tolerance, Tol, chosen by the user. Any element K that does not meet the requirement,

$$ERROR_K \leq Tol \tag{3.53}$$

requires refinement. Here $ERROR_K$ is the error on element K defined by summing the error of the three nodes on triangle K. As stated before, smaller elements will more closely resemble a continuous function and thus will decrease the error. However, a refinement of each and every element is inefficient. Thus the idea is to only refine particular elements. So by refinement, we mean that new nodes will be added to the mesh such that all elements that do not satisfy (3.53) are split into multiple smaller triangles. Here we have chosen to add a single node to the element. This node will be located at the centroid of the triangle. If (x_1, y_1), (x_2, y_2) and (x_3, y_3) are the corners of a given triangle, then the centroid where the refinement node is placed is defined by,

$$Centroid = \Big(\frac{x_1 + x_2 + x_3}{3}, \frac{y_1 + y_2 + y_3}{3}\Big)$$

The centroid is added to the mesh through the *Locations* matrix. Thus any element can be refined by adding a new row to *Locations* which contains the x and y coordinates of that particular element's centroid. This can be repeated as many times as needed. For every element that does not satisfy (3.53), a new node is added to the mesh which will split the element into smaller ones. This is done in the code by adding a new row containing the x and y coordinates of the centroid to the *Locations* matrix. Once *Locations* has been updated with a new node for all elements not satisfying (3.53), the Delaunay triangulation can be rebuilt with the new set of nodes. A simple example of this is shown in Figure (3.5).

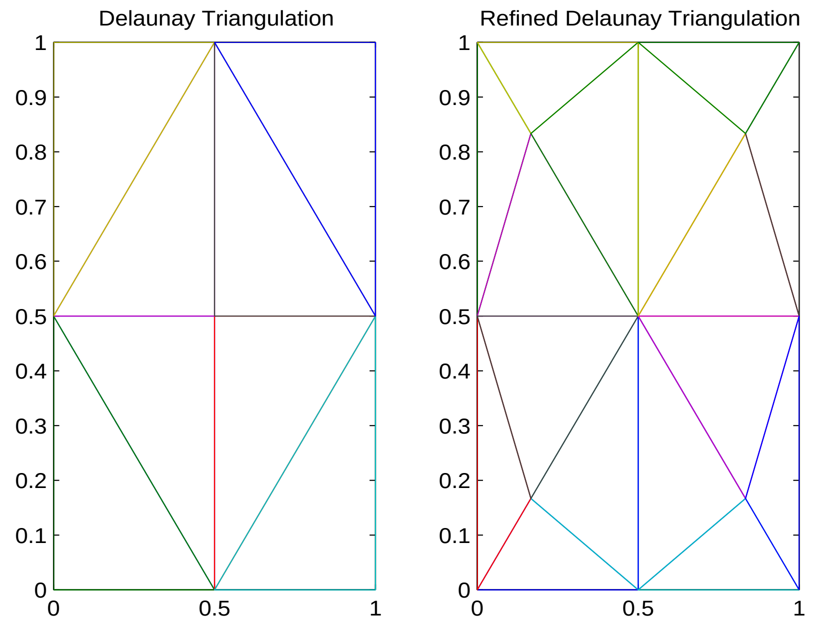

Figure 3.5: On the left is a uniform Delaunay triangulation for a 3×3 mesh. When a new node is been placed at the centroid of the 4 corner elements the Delaunay triangulation is then applied to this new list of nodes to generate the figure on the right.

Unfortunately there is no guarantee that all elements of the mesh will satisfy (3.53) once this refinement is made. A single refinement can only do so much in terms of decreasing the error on any given element. But we are not limited to just a single refinement. We may use a conditional *While* loop to continue refining the mesh until each and every element does satisfy (3.53). Figure (3.6) demonstrates the progression of this process from a uniform initial mesh through three refinements.

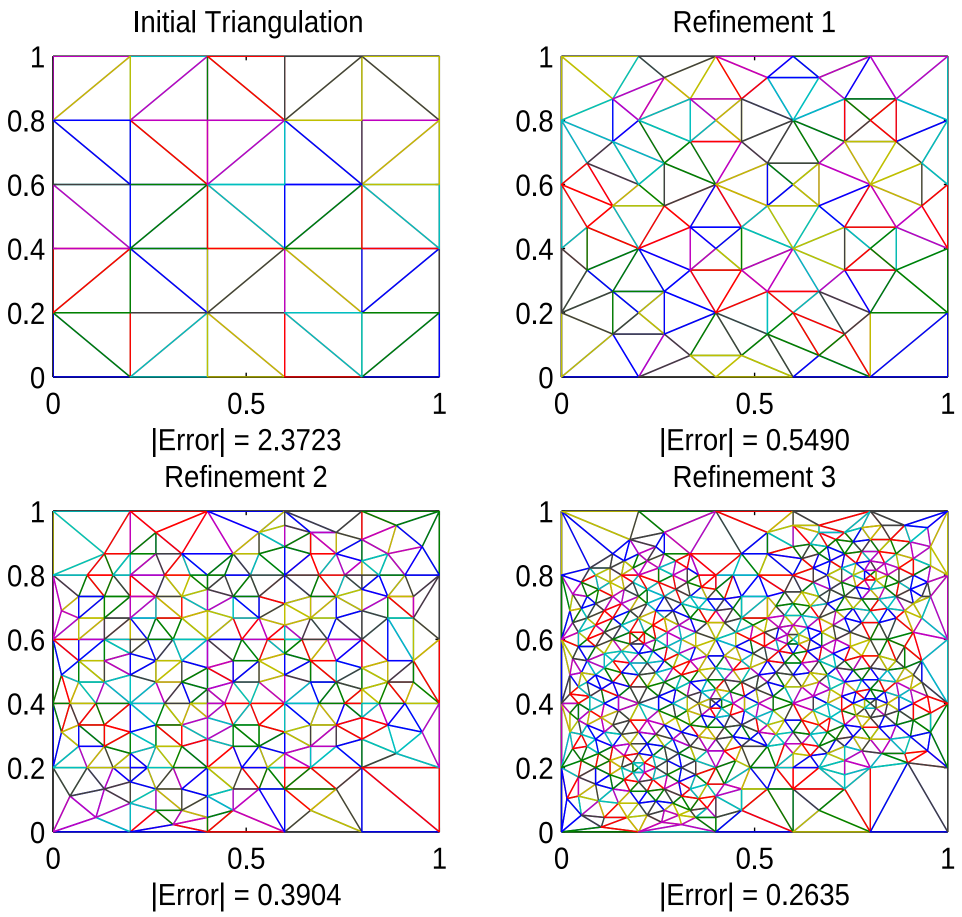

Figure 3.6: The initial uniform mesh along with the first three refinements with the error norm of the approximation found on each mesh as defined by (3.52). This approximation was calculated with μ, λ and ρ all set to 1, $\beta = 0.01$ and $g = 9.81$. The magnitude of the initial pressure is set to 1.

As we continue refining the mesh, the effect it has on the error decreases, making the error more difficult to improve. The dependency on the grid spacing h in equation (3.52) partially explains this. As the elements get smaller and smaller, the potential for decreasing h becomes less. However with each refinement, the size of the data structures *Elements* and *Locations* as well as the stiffness matrix and load vector grow greatly. This means that the computational cost of refinement can be great. This should be kept in mind when defining the tolerance.

In Table (1) we can find some valuable information on how to optimize the results of the

Table 1: On the left is a table of errors on uniform meshes of N^2 nodes for $N = 10$ to 30. On the right is a table for the error in the first four refinements of the same initial $N = 10$ uniform mesh.

N	Nodes	Error
10	100	2.4058
15	225	0.5946
20	400	0.3572
25	625	0.1370
30	900	0.1124

Refinement	Nodes	Error
Initial Mesh	100	2.4058
1	255	0.2747
2	643	0.1981
3	1708	0.1639
4	4802	0.1370

adaptive refinement method proposed in this thesis. In both tables we start with finding the error on a uniform mesh of $N = 10$, or 100 nodes. From there we use two methods for improving the accuracy of the finite element approximation. First, we simply increase the value of N which in turn decreases the uniform grid spacing h which in turn increases the number of nodes. Such a method is called *h-refinement* and is simply a refinement of each and every element in the mesh by adding more nodes and decreasing the uniform grid spacing h. This is the simplest form of mesh refinement. The second option is to use the adaptive refinement algorithm suggested in this section which will only refine elements of insufficient accuracy. At first it would seem from Table (1) that the refinement algorithm is an efficient option. After four refinements we have a mesh of $4,802$ nodes with an error of 0.1370. This error is nearly identical to that of the first method when $N = 25$ and only 625 nodes are used. But a closer look shows that during the first refinement we have achieved a massive improvement in the error. The error after a single adaptive refinement drops to 0.2747 while using only 255 nodes. A side test was ran on a uniform mesh using the *h-refinement* method and we did not achieve a similar error of 0.2670 until we let $N = 22$ giving us 484 nodes. Fewer nodes means we have fewer equations to solve and thus our program can run more efficiently while achieving similar accuracy. Only after this first refinement does the algorithm become more effective than simply increasing the value of N. This can be seen more clearly in Figure (3.7) where the errors from both tables are plotted with respect to the number of nodes of the mesh they were found on.

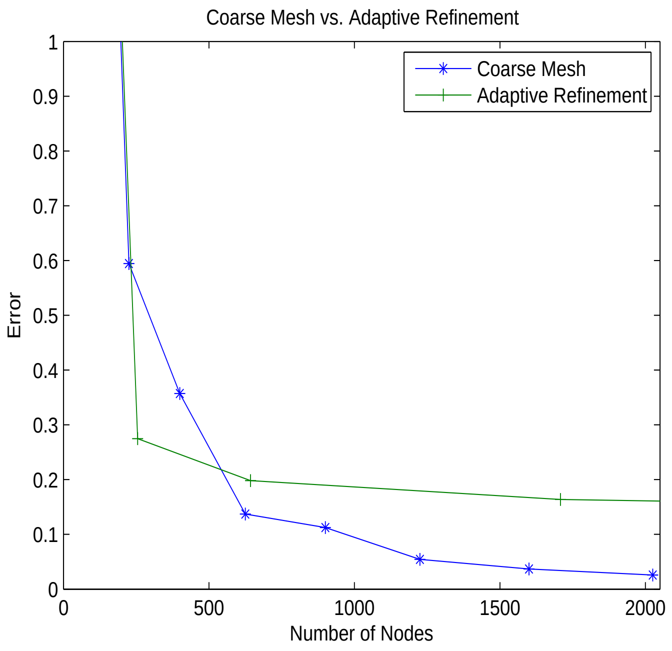

Figure 3.7: Errors from a coarse mesh of N^2 nodes for $N = 10$ to 45 compared to the error from the same initial 100 node coarse mesh along with that of the first 4 refinement meshes. For each of these tests we set μ, λ, and ρ equal to one, $\beta = 0.00001$ and let the refinement tolerance be 0.02

An analysis of Figures (3.6) and (3.7) shows us that the error estimate defined by equation (3.52) is not sufficient. The current error estimate uses a global value for the grid spacing h which is a constant defined as the single largest edge of any element on the entire mesh. There will likely be some elements of a mesh that do not require nearly as many refinements as others. This leads to the problem we see here where the longest edge in the mesh differs greatly from the average edge length. In Figure (3.6) the longest edge in the three refinements shown is found in the lower left corner. We can see that this length is fairly accurate for the first refinement but becomes exceedingly worse for refinements two and three. To solve this problem we turn our attention from the error on each node toward the error on each element. Doing so allows us to find the longest edge on any given elements and use that as the value of h for the local error estimate. Doing so leads to the error estimate for a single element K,

$$\|e_K\| \leq \|h_K^2 R(\boldsymbol{u}_h^K)\|_\infty \tag{3.54}$$

where h_K is now the longest edge on element K rather than a global constant and $R(\boldsymbol{u}_h^K)$ is the sum of the residual on the three nodes of element K.

Table 2: Error results in the FEM using the local error estimate.

Refinement	Number of Elements	Average Local Error
Initial Mesh	578	0.139931
1	1682	0.056900
2	4566	0.021569
3	10616	0.009638

We can see from Table (2) that by using the local error estimate defined by equation (3.54) we achieve a more steady decline in the error than that found in Table (1). Despite the fact this new method of approximating the error on each element will potentially decrease the number of refinements needed on some elements, Table (2) shows that the number of elements still grows at an alarming rate. So we are still limited to the number of refinements or our choice in the error tolerance. However, we can now see that results of sufficient accuracy can be efficiently reached in a reasonable number of refinements.

In Figure (3.8) we have applied the estimate defined by equation (3.54) using local grid spacing values h_K which lead to better approximations of the error for each refinement than those shown in Figure (3.6). The error on the initial mesh was improved greatly by using a smaller value for β which improves the accuracy of the pressure approximation.

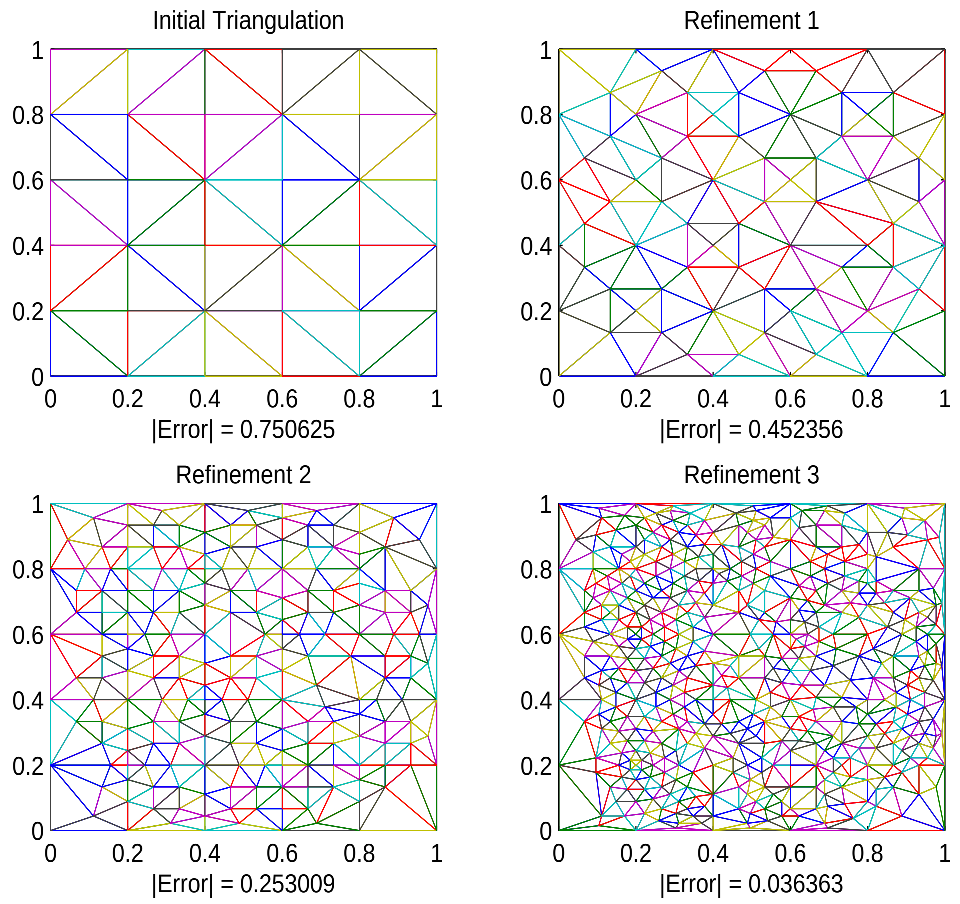

Figure 3.8: The initial uniform mesh along with the first three refinements with the error norm of the approximation found on each mesh as defined by (3.52). This approximation was calculated with μ, λ and ρ all set to 1, $\beta = 0.00001$ and $g = 9.81$. The magnitude of the initial pressure is set to 1.

3.7 FINITE ELEMENT METHOD IN MATLAB

Just as we did for the porous medium equation, we now go into detail explaining how the finite element method is implemented in MATLAB for Navier's equation. The FEM function should require the user input values for μ, λ, ρ, N, β, and the error tolerance for the adaptive refinement method.

The first step is to define our mesh and create the data structures, *Locations* and *Ele-*

ments. Since our initial uniform mesh is just a triangulation of $(N-1)^2$ squares cut along the diagonal as in Figure (3.1), the number of elements in the initial mesh is defined by

```
E = 2*(N-1)^2;
```

Next we define the nodes on the mesh and build a grid with the commands,

```
H = 1/(N-1);

X = 0:H:1;

Y = X;

[x0,y0] = meshgrid(X,Y);
```

Here we use the value H to be the uniform grid spacing defined by the number of nodes. Using this we can define the uniform grid using MATLAB's *meshgrid* command. This function takes two vectors and returns two matrices that will allow all combinations of x and y to be considered. For example, if we let both x and y be the vectors $[0, 1, 2, 3, 4, 5]$, then $[x0, y0] = meshgrid(x, y)$ produces:

```
x0 =


     0     1     2     3     4     5
     0     1     2     3     4     5
     0     1     2     3     4     5
     0     1     2     3     4     5
     0     1     2     3     4     5
     0     1     2     3     4     5


y0 =
     0     0     0     0     0     0

     1     1     1     1     1     1

     2     2     2     2     2     2

     3     3     3     3     3     3

     4     4     4     4     4     4

     5     5     5     5     5     5
```

The matrices x0 and y0 can now be used to plot a three dimensional function at all combinations of $(x, y) \in 0, 1, 2, 3, 4, 5$. With the coordinates of each node now well defined by the vectors X and Y, we can build the *Locations* matrix. While X and Y are vectors, *Locations* holds the values of nodes on a two dimensional grid. A double *FOR* loop can be used here to account for all combinations of x and y.

```
Locations = zeros(n^2,2);

k = 1;

for i = 1:N

    for j = 1:N

        Locations(k,1) = X(1,j);

        Locations(k,2) = Y(1,i);

        k = k+1;

    end

end
```

The *Locations* matrix is similar to the *meshgrid* command in that they both represent a grid of all combinations of x and y. However *meshgrid* returns two square matrices while *Locations* stores the x and y values in the first and second column respectively. This makes it easier to keep track of which node we are using. For example, in *Locations* the k^{th} node is simply stored in the k^{th} row. Next, we can use the *Locations* matrix to build *Elements* using the *delaunay* command in MATLAB.

```
Elements = delaunay(Locations(:,1),Locations(:,2));
```

This results in the Delaunay triangulation which defines the elements in our mesh. The final step before we can begin the finite element algorithm is defining which rows in the *Locations* matrix represent boundary nodes since these must be handled separately.

```
numBC = 4*N-4;
DirBC = zeros(1,2*numBC);
k = 1;
for i = 1:N^2
    if Locations(i,1)*Locations(i,2)==0||Locations(i,1)==1||Locations(i,2)==1
        DirBC(1,k) = 2*i-1;
        DirBC(1,k+1) = 2*i;
        k = k+2;
    end
end
```

The number of boundary nodes, $numBC$, on a square grid is defined by $numBC = 4N - 4$. Here we subtract by four in order to avoid counting the corners twice. The vector $DirBC$ is of length $2 * numBC$ since each node has an x and y component. If either component of the k^{th} row in *Locations* is zero or one, then the k^{th} row represents a boundary node. But again since each node has two components, we add two elements to the $DirBC$ vector each time a boundary node is found. The $DirBC$ vector will be used later to remove these values from the global stiffness matrix and load vector as they are not needed for approximating the solution on the interior nodes.

Next we can build the stiffness matrix which starts off as simply a matrix of all zeros using the command,

```
A = zeros(2*N^2,2*N^2);
```

From here we loop over every element and find the local stiffness matrix for each of them. The global stiffness matrix, A, is then updated at locations corresponding to the u and v solutions of the nodes on that element. As mentioned before, these are found with the commands

```
Global = 2*N(k,[1,1,2,2,3,3]) - [1,0,1,0,1,0];
Nodes = Elements(k,1:3);
x = Locations(Nodes,1);
y = Locations(Nodes,2);
```

Since we have chosen our basis functions to be $\varphi(x,y) = ax + by + c$, the gradient of φ is defined by $\nabla\varphi = (a,b)^T$. For each element there we have φ_1, φ_2, and φ_3. to account for each of the three nodes. The gradient of each of the three basis functions can be found simultaneously as follows,

$$\begin{pmatrix} \nabla\varphi_1 \\ \nabla\varphi_2 \\ \nabla\varphi_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{pmatrix}^{-1} \cdot \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{3.55}$$

In MATLAB this is handled with the command,

```
PhiGrad = [1,1,1;x(1),x(2),x(3);y(1),y(2),y(3)]\[zeros(1,2);eye(2)];
```

But we must also make sure that the basis functions satisfy the homogeneous Dirichlet boundary conditions. So we then set PhiGrad equal to zero for any node where the x or y coordinate is zero or one. This is done by,

```
for i = 1:3
   if (x(i)*y(i) == 0 || x(i) == 1 || y(i) == 1)
      PhiGrad(i,:) = 0;
   end
end
```

Equation (3.35) which defines the local stiffness matrix is built around the following two matrices

$$\begin{aligned} R_K &= \begin{pmatrix} \varphi_{K_1,x} & 0 & \varphi_{K_2,x} & 0 & \varphi_{K_3,x} & 0 \\ 0 & \varphi_{K_1,y} & 0 & \varphi_{K_2,y} & 0 & \varphi_{K_3,y} \\ \varphi_{K_1,y} & \varphi_{K_1,x} & \varphi_{K_2,y} & \varphi_{K_2,x} & \varphi_{K_3,y} & \varphi_{K_3,x} \end{pmatrix} \\ C &= \begin{pmatrix} \lambda+2\mu & \lambda & 0 \\ \lambda & \lambda+2\mu & 0 \\ 0 & 0 & \mu \end{pmatrix} \end{aligned}$$

The matrix C is built entirely of the input values μ and λ as follows,

```
C = mu*[2,0,0;0,2,0;0,0,1] + lambda*[1,1,0;1,1,0;0,0,0];
```

The matrix R^K, on the other hand, is a bit more difficult. We can notice that the odd elements of rows one and three are composed of the partial derivatives of φ_i with respect to x and y respectively. The even elements of rows two and three are composed of the partial derivatives of φ_i with respect to y and x respectively. All other elements are zero. In MATLAB the R matrix is built as follows,

```
R = zeros(3,6);
R([1,3],[1,3,5]) = PhiGrad';
```

```
R([3,2],[2,4,6]) = PhiGrad';
```

With R and C now defined, we can find the local stiffness matrix as defined by equation (3.35).

```
Ak = det([1,1,1;x(1),x(2),x(3);y(1),y(2),y(3)])/2*R'*C*R;
```

and the global stiffness matrix A is then updated using the Ak and $Global$ matrices

```
A(Global,Global) = A(Global,Global) + Ak;
```

This process is repeated for every single element in the mesh. When all elements have been accounted for, we can use the $DirBC$ matrix to remove all boundary nodes from the global stiffness matrix. In MATLAB an element of a matrix can be deleted by simply setting it equal to [ ]. The command

```
A(DirBC,:) = [];
A(:,DirBC) = [];
```

removes all rows and columns of A which represent a boundary node. The global stiffness matrix A is now complete.

Next we need to build the load vector b, which is again done by finding the local load vector on each triangle. Since the load vector is dependent on the pressure, we must call the program defined in Chapter **2**.

```
[XX,P,px,py] = Pressure(9,h,T,beta);
[xI,yI] = meshgrid(XX,XX);
```

Here XX is a vector containing the Chebyshev nodes, P is the pressure approximation, px is the partial derivative of P with respect to x, and py is the partial derivative of P with respect to y. On the second line we use XX to define the grid on which the pressure is defined. This grid is used to interpolate the partial derivatives of P from the Chebyshev nodes onto the FEM nodes.

```
Px = interp2(xI,yI,px,x0,y0,'spline');
Py = interp2(xI,yI,py,x0,y0,'spline');
```

The *interp2* function interpolates the *px* and *py* matrices from the $[xI, yI]$ Chebyshev grid onto the $[x0, y0]$ FEM grid using a cubic spline option.

Equation (3.41) with $f(x, y) = \rho g + \nabla p$, shows us that other than the partial derivatives of the pressure just found, we only need the area of the triangle to find the load vector. Again we loop over every element and construct *Global*, *Nodes*, x and y by

```
Global = 2*N(k,[1,1,2,2,3,3]) - [1,0,1,0,1,0];
Nodes = Elements(k,1:3);
x = Locations(nodes,1);
y = Locations(nodes,2);
area = polyarea{x,y};
```

MATLAB has a built in function, *polyarea*, that can be used to calculate the area of any closed polygon if given the coordinates of each node. Using this function we define the area of triangle K.

The local load vector is of length six, where the odd elements represent the x displacement and the event elements represent the y displacement on each of the three nodes.

```
DPx = (Px(Elements(k,1),1)+Px(Elements(k,2),1)+Px(Elements(k,3),1))/3+rho*g;
b(Global([1,3,5]),1) = b(Global([1,3,5]),1) + DPx*area/3;


DPy = (Py(Elements(k,1),1)+Py(Elements(k,2),1)+Py(Elements(k,3),1))/3+rho*g;
b(Global([2,4,6]),1) = b(Global([2,4,6]),1) + DPy*area/3;
```

The values DPx and DPy take the average of Px and Py respectively, and add them to ρg in order to approximate $f(x, y)$ on the element. The global load vector b is then updated by multiplying DPx and DPy by the area as in equation (3.41) and splitting that value into third to distribute it evenly over each node.

The last step in building the load vector, as it was for the stiffness matrix, is to remove all boundary nodes. This is done with the command

```
b(DirBC) = [];
```

With the stiffness matrix and load vector now defined, we have a system of the form

$$AU = b \tag{3.56}$$

where U = (u,v). MATLAB has a number of methods for solving such systems. We can use the fact that stiffness matrix is symmetric and positive definite to optimize this process by setting the following conditions

```
opts.SYM = true;
opts.POSDEF = true;
```

and then using the *linsolve* function to get

```
U = linsolve(A,b,opts);
```

The solution vector U contains both $u(x, y)$ and $v(x, y)$. The odd elements of U make up u and the even elements of U make up v. These can be separated by the commands

```
u = U(1:2:size(U,1)-1,1);
v = U(2:2:size(U,1),1);
```

This entire process must be repeated for every time step. Nothing changes in the implementation other than that the pressure derivatives are now matrices with each column representing a separate time step. We then loop through every column of the pressure derivative and perform this entire calculation with each of those defining a new load vector.

4 EXPERIMENTAL RESULTS

4.1 PRESSURE RESULTS

The first goal for our results from experimenting with the pressure is to determine an optimal number of nodes, N, to define the discrete spatial domain. This value determines the number of nodes in both the x and y directions upon which the spectral difference operators are defined.

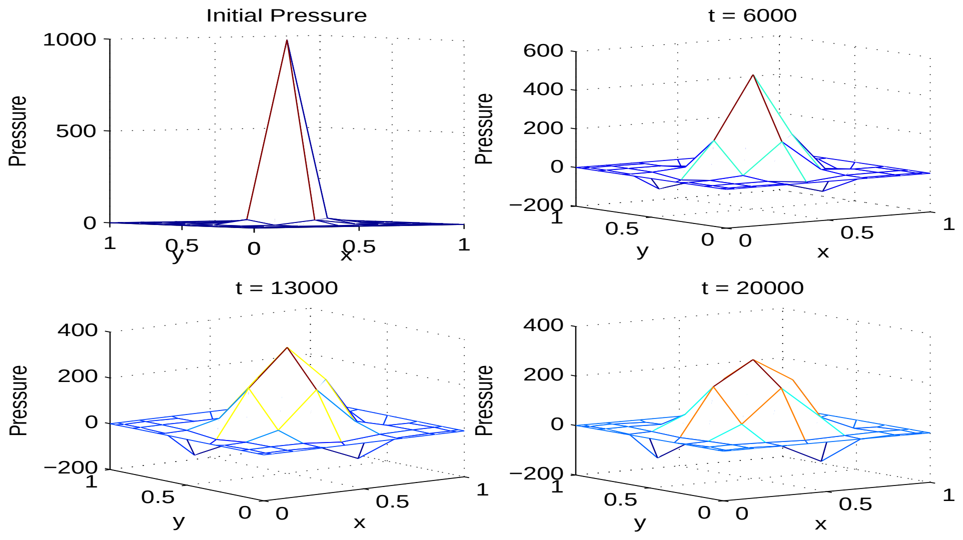

Figure 4.1: Plots of the initial pressure as well as the pressure at time steps t = 6000, 12000, and 20000. For this test we let $N = 8$,$h = 0.001$, $T = 20$, and $\beta = 0.000001$.

Figure (4.1) shows a key problem with our approximation. We can see that as time goes on, the pressure becomes negative toward the center of all four sides of the square grid. This is not physically accurate in that the pressure should always be positive. Small cases of negative values could be ignored since the interpolating polynomial could be above or below the exact solution between nodes and this could be a cause for negative values. But we are getting results that are far less than zero. In Figure (4.1) the values reach nearly negative one hundred after $20,000$ time steps and continues to get worse with every time step.

The cause of this is our choice in a square grid which do not allow for boundary conditions that uniformly comply with the diffusion of the initial pressure. The nodes in the center of each of the four sides of the boundary are closer to the source than those near the corners of the boundary. However we are enforcing the same value onto each of these nodes. In Figure (4.2) we see that using $N = 9$ gives us an initial pressure that while still defined by a Gaussian, more closely resembles a $3D$ rectangular structure and thus matches our square grid more closely. This helps to minimize the issue with negative pressure values.

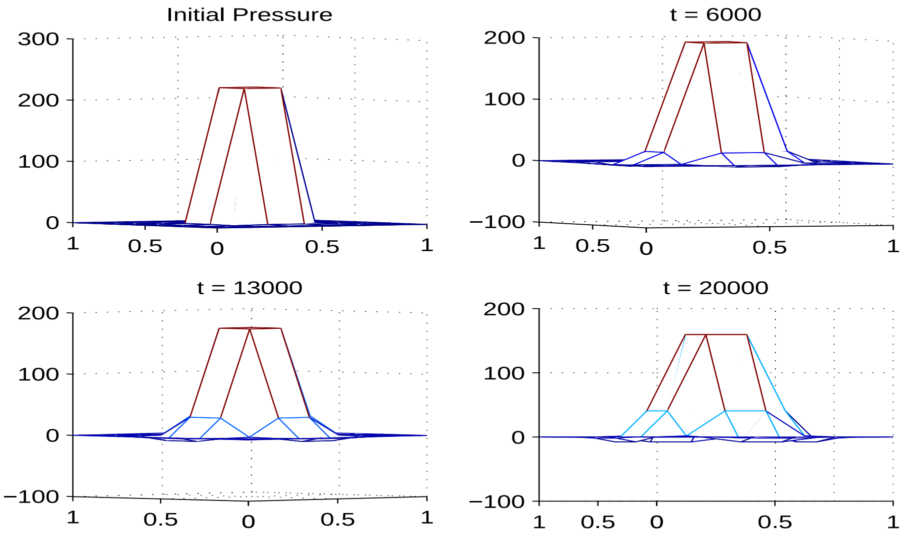

Figure 4.2: Plots of the initial pressure as well as the pressure at time steps t = 6000, 12000, and 20000. For this test we let $N = 9$,$h = 0.001$, $T = 20$, and $\beta = 0.000001$.

In Table (3) we continue to set $\beta = 0.000001$, $T = 20$, and use the value $N = 9$ in an effort to find a good time step size. A comparison between the error and the runtime allows us to select a value of h that gives a good balance between accuracy and efficiency.

We can see from Table (3) that while decreasing the step size h does lower the error, it also has a big impact of the runtime. The step sizes are being cut in half for each of these tests yet the increase in runtime does not follow a similar linear progression. The run time appears to be growing exponentially as the step size decreases. When we let $h = 0.0016$, the

Table 3: Error and runtime results for the pressure approximation with $N = 9$, $\beta = 0.000001$ and a total time of $T = 20$ seconds. The slope of the log-log error plot is 2.000004.

Time Step	Error	Runtime (seconds)
h = 0.05	$0.5543 \cdot 10^{-6}$	0.488338
h = 0.025	$0.1386 \cdot 10^{-6}$	1.071605
h = 0.0125	$0.0346 \cdot 10^{-6}$	2.373442
h = 0.0063	$0.0087 \cdot 10^{-6}$	5.760332
h = 0.0031	$0.0022 \cdot 10^{-6}$	16.616368
h = 0.0016	$0.0005 \cdot 10^{-6}$	54.148479
h = 0.0008	$0.0001 \cdot 10^{-6}$	183.297758

error is approximately $5 \cdot 10^{-10}$ with a run time of under one minute. While the accuracy of the approximation continues to improve, the run time jumps to over three minutes for $h = 0.0008$. It would seem from Table (3) that a step size of about $h = 0.002$ gives us the best balance of accuracy and efficiency.

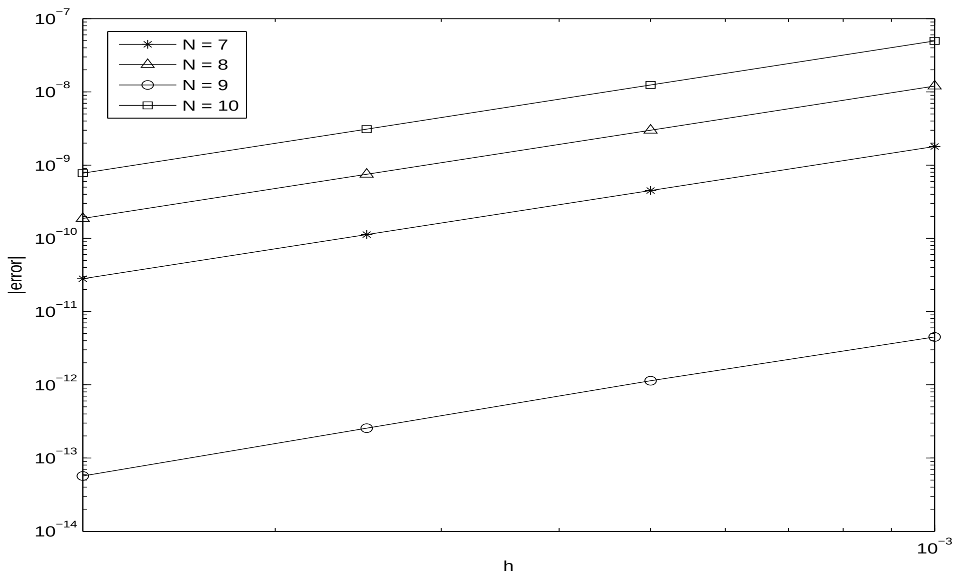

Figure 4.3: Plots of the error at different time steps for four different values of N. We have let $\beta = 0.000001$ and used an initial amplitude of 1000 for all four plots. We can see that using $N = 9$ gives the best results.

Now that we have settled on a good time step size, our next goal is to find a value of N that gives us the best results. Figure (4.3) plots the error as a function of the time step with four different values of N. In all cases we see that the plots have a slope of approximately two as expected. We have used $\beta = 0.000001$ and an amplitude of 1000 for the initial pressure on each of these four plots. It is no surprise that the best result is $N = 9$, an odd number. Since the Chebyshev nodes are defined on x_i for $i \in [0, N]$, $N = 9$ actually gives us ten nodes. Meaning we will have Gaussian which resembles that shown in Figure (4.2) which provided better results than the odd values from Figure (4.1).

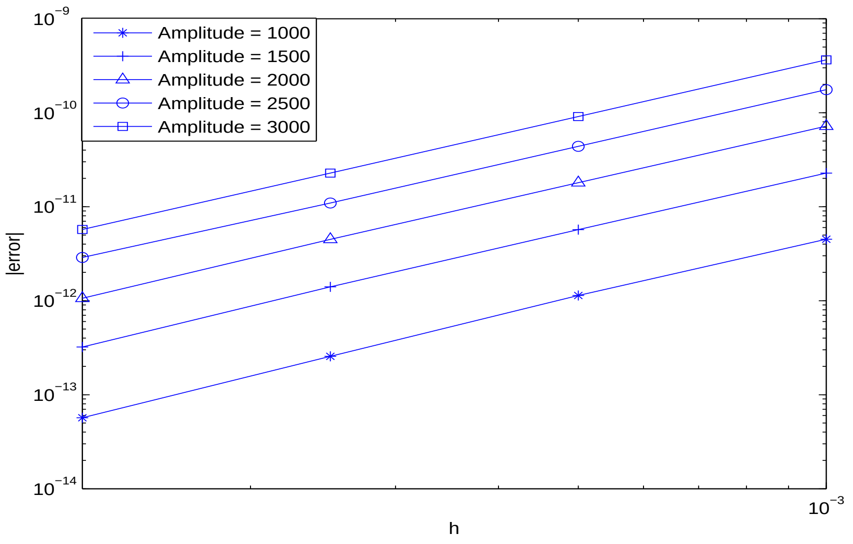

Figure 4.4: Plots of the error at different time steps for five different initial amplitudes of the pressure Gaussian. We have let $\beta = 0.000001$ and used $N = 9$ for all five plots. We can see that as the amplitude increases, the error gets worse.

Unlike the step size or the number of nodes, the amplitude of the initial pressure is something that will likely change for each test. The amplitude represents the magnitude of the initial pressure imposed by the rocket. Tests should be run for many different values of the amplitude. Figure (4.4) again shows that we have a slope of two when the error is plotted as a function of the time step. We also notice that as the amplitude increases the method become less accurate. But even for an amplitude of 3000 N/m^2 we have an error of no worse that approximately 10^{-10}.

If we remember that the solutions of the finite element method represent the displacement of the soil due to the applied pressure, we can expect a couple of key things. First, we would expect that by applying greater pressure to the system we are in fact increasing the forcing term in Navier's equation and thus the displacement should be greater. A more powerful rocket should have a greater effect in terms of the displacement of the soil. Next, we should expect that soil particles on opposite sides of the pressure source would be displaced in equal but opposite directions.

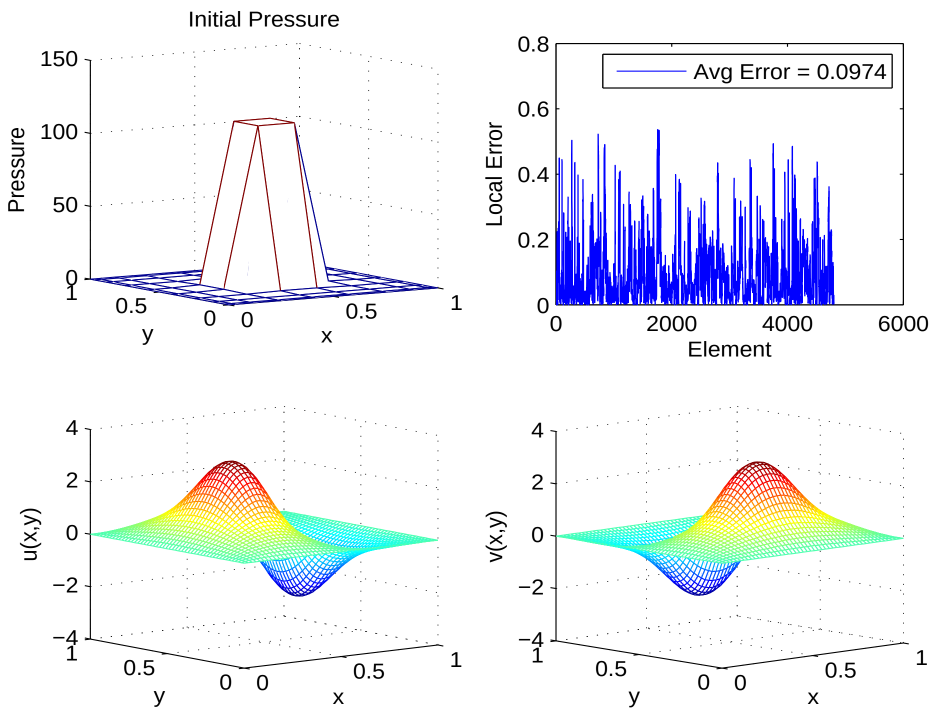

Figure 4.5: The four plots shown here are the initial pressure, the error in the FEM approximation on each element, and the mesh plots of u(x,y) and v(x,y) which represent the x and y displacement. For this test we let $N = 50$, $\mu = 1$, $\lambda = 1$, $\rho = 1$, and $\beta = 0.000001$.

In Figure (4.5) we see an initial pressure of at most 100 Newtons, as well as the $u(x, y)$ and $v(x, y)$ solutions after a single time step of 0.001 seconds. Here the local error plotted on the z axis is the error on a given element. This is found by summing the three values in the vector $h^2 \cdot R(\boldsymbol{u}_h)$ which correspond to the nodes of the given triangle. Such a pressure results in a maximum displacement in the x direction of approximately ± 2 meters. The maximum displacement in either direction is near the center of the grid and the displacement decreases as we look to particles farther and farther away from the central pressure source. The y displacement is nearly identical to the x displacement due to the initial pressure being defined by a uniform Gaussian. The only difference between the x and y displacement is that the plot is rotated by 90 degrees, as expected.

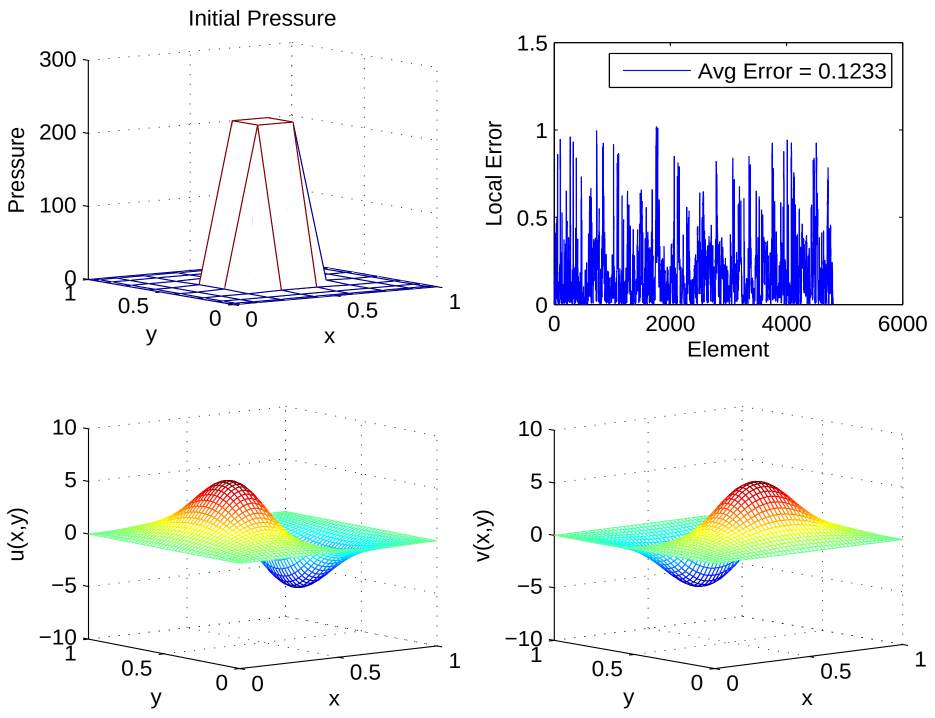

Figure 4.6: The four plots shown here are the initial pressure, the error in the FEM approximation on each element, and the mesh plots of u(x,y) and v(x,y) which represent the x and y displacement. For this test we let $N = 50$, $\mu = 1$, $\lambda = 1$, $\rho = 1$, and $\beta = 0.000001$.

In Figure (4.6) we again see a plot of the initial pressure and the x displacement caused by this applied pressure after 0.001 seconds. However here we have set the initial pressure such that it reaches a maximum of 200 Newtons, ten times that used in Figure (4.5). As one would expect, the increase in pressure has resulted in an increase in displacement. The soil particles have now been displaced by over ± 5 meters.

The value of the Lamé constants μ and λ, the number of nodes, N, as well as the density of the soil, ρ, play a large role in our problem. This is true for not only the physical properties of the results but also the accuracy and stability of the finite element approximation. Figure (4.7) can be used as a basis for comparison of different combinations of values for these constants. Here we used a relatively simple case where $\mu = 1$, $\lambda = 1$, $\rho = 1$, and $\beta = 0.0001$. For this combination of parameter values, $N = 18$ is the largest value where three refinements are possible. This is due to the fact that we are using the student version of MATLAB which greatly limits the maximum dimensions of matrices. While there are spikes in the error at a number of elements in the mesh, the general trend of the refinement process is a sufficiently steady decline in the average error across the entire mesh. After just three refinements, the approximation is estimated to be less than one percent off from the exact solution.

In Figure (4.8) we have decreased the value of N to be 12 and left all other values exactly the same as those used to create Figure (4.7). As is the case with most numerical methods, a decrease in the number of nodes on the mesh has decreased the accuracy of the approximation. However we still see a steady decline in the error after each refinement. Changing the number of nodes in the initial mesh seems to have little effect on the stability of the approximation, but clearly it has a large impact on the accuracy.

In Figure (4.9) we have gone back to using $N = 18$ but have switched the value of μ to 100. We can see that the average error over the initial mesh as well as all three refinements shown is remarkably similar to those found for $\mu = 1$ in Figure (4.7). The spikes in error occur on different elements, but the overall error is none the less unchanged. Changes to the value of μ seem to have little to no effect on the accuracy or stability of the approximation.

In Figure (4.10) we run a similar test where we switch the value of μ back to one while

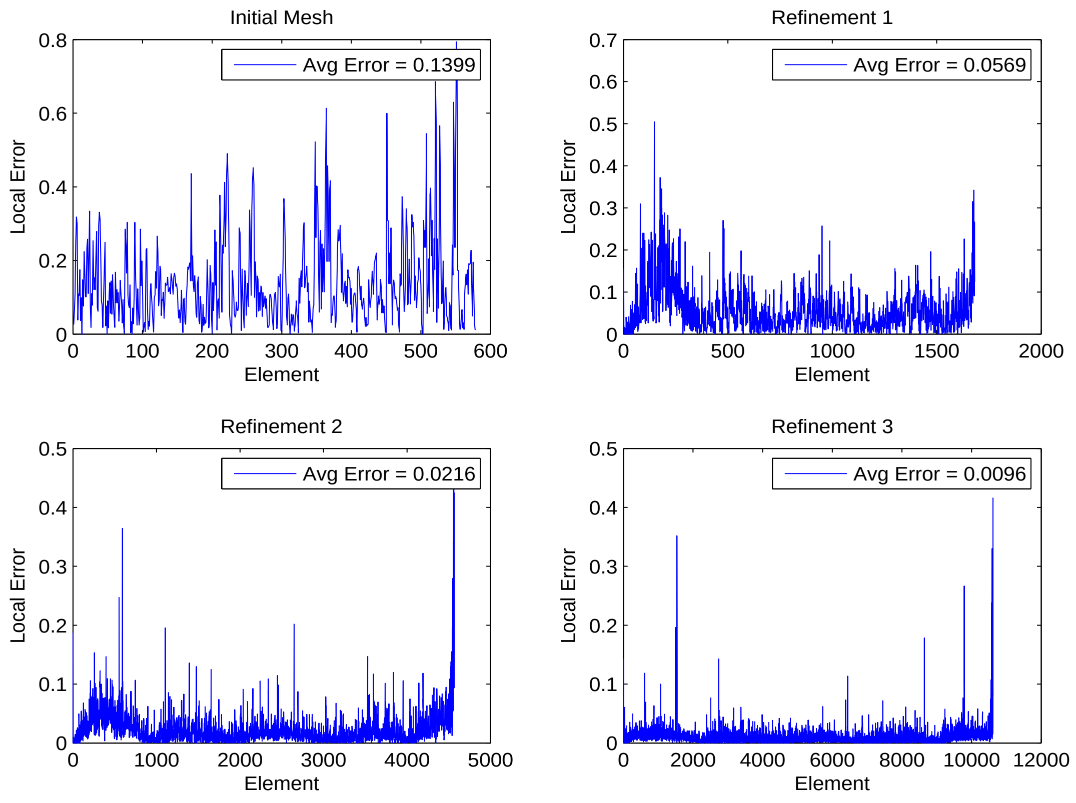

Figure 4.7: Local error on each element for an initial mesh with $N = 18$, $\mu = 1$, $\lambda = 1$, $\rho = 1$, and $\beta = 0.0001$ as well as the first three refinement of this mesh.

setting λ equal to 100. However unlike for the previous test, we see that there is in fact a very noticeable change in the average error when compared to that shown in Figure (4.7). The differences in the effects on the error from changing μ versus changing λ can be seen more clearly in Table (4). It can be helpful to remember that our problem is defined in equation (3.12) to be,

$$\mu\Delta\boldsymbol{u} + (\mu + \lambda)\nabla(\nabla \cdot \boldsymbol{u}) = \boldsymbol{f} \tag{4.1}$$

The constant μ appears in the coefficient of both terms while λ only appears in the term containing the gradient of the divergence of the displacement solution. The λ constant occurs

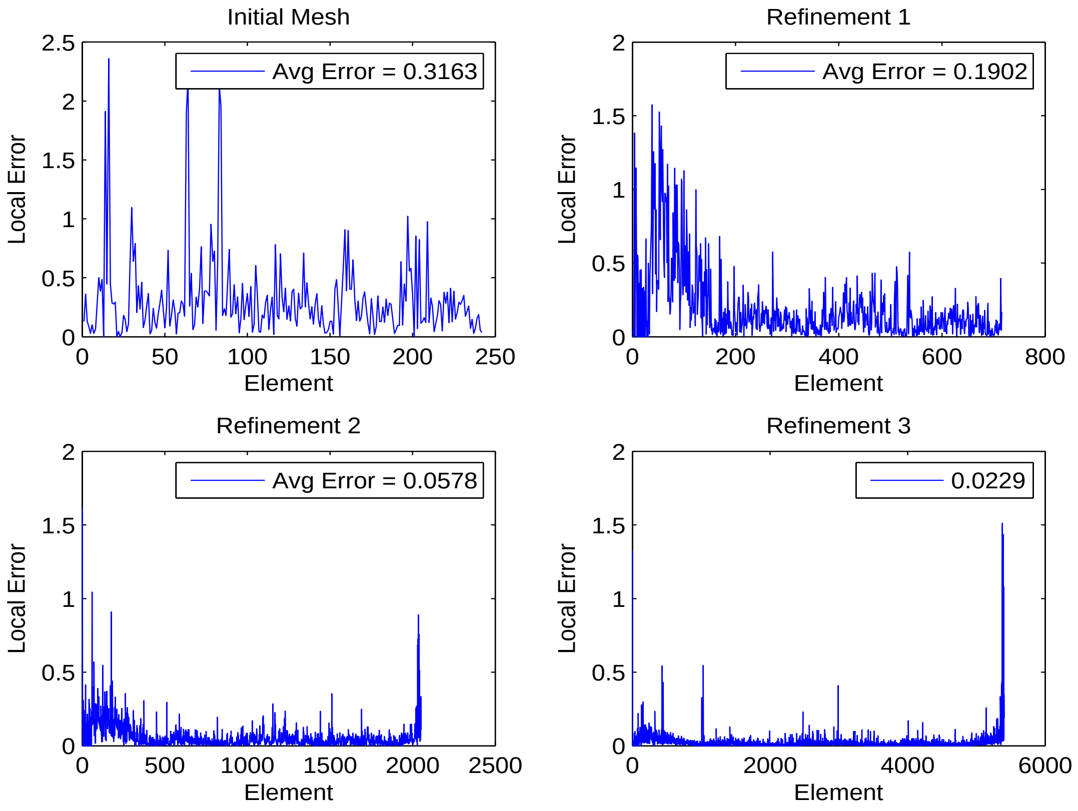

Figure 4.8: Local error on each element for an initial mesh with $N = 12$, $\mu = 1$, $\lambda = 1$, $\rho = 1$, and $\beta = 0.0001$ as well as the first three refinement of this mesh.

together with μ where the summation of the two make up the coefficient of the second term of the equation. We tested the two cases, $(\mu, \lambda) = (100{,}1)$ and $(\mu, \lambda) = (1{,}100)$ where the coefficient $(\mu + \lambda)$ is the same for each. This means that the value of the coefficient $(\mu + \lambda)$, and thus the value of λ, alone can not be the reason for this difference. Rather, it must be the ratio of the two coefficients μ and $(\mu + \lambda)$ that results in this difference in accuracy.

The conclusion from the previous tests that the ratio of the two coefficients in equation (3.12) rather than the values of the constants μ and λ themselves is what results in changes in accuracy is reinforced in Figure (4.11). Here we have changed both μ and λ to equal 100 rather than just one. The ratio of the two coefficients in equation (3.12) is again close to that used to generate Figure (4.7) and as expected, the average error over each refinement

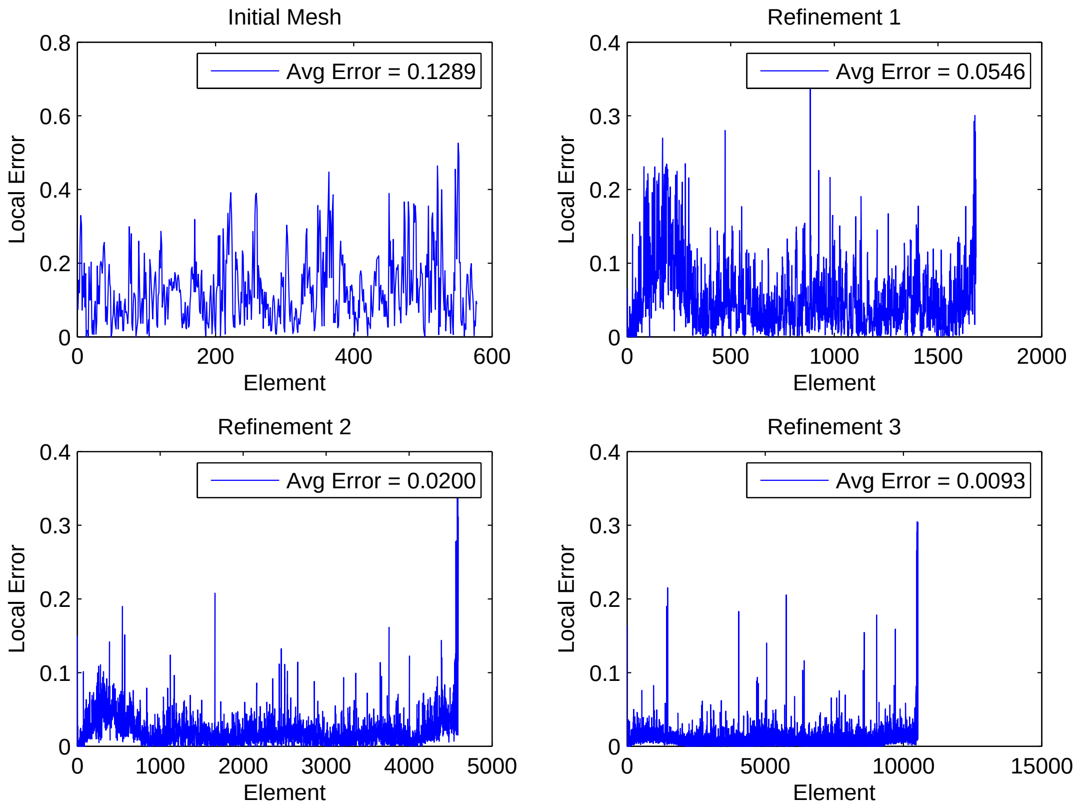

Figure 4.9: Local error on each element for an initial mesh with $N = 18$, $\mu = 100$, $\lambda = 1$, $\rho = 1$, and $\beta = 0.0001$ as well as the first three refinement of this mesh. Notice that the change in μ has had little effect on the error when compared to Figure (4.7).

mesh is nearly identical to those found in Figure (4.7).

So far we have seen how changes in the constants μ and λ effect accuracy, but the stability of the approximation was never an issue. Changes to the value of ρ however can cause problems with the convergence and efficiency of the approximation which greatly effects the stability of the method. In Figure (4.12) we see that with μ and λ again set being set to one but letting ρ equal 10, we have a major loss in accuracy over each mesh. A consequence of this inflation in error is that more elements will require refinement in each refinement iteration. This leads to more nodes being added to the mesh and hence more equations needing to be solved. The runtime of the program increases greatly and the MATLAB

Table 4: Error results in the finite element approximation for $(\mu, \lambda) = (100{,}1)$ and $(\mu, \lambda) = (1{,}100)$.

Refinement	$(\mu, \lambda) = (100{,}1)$	$(\mu, \lambda) = (1{,}100)$
Initial Mesh	0.1289	0.3186
1	0.0546	0.1404
2	0.0200	0.0241
3	0.0093	0.0150

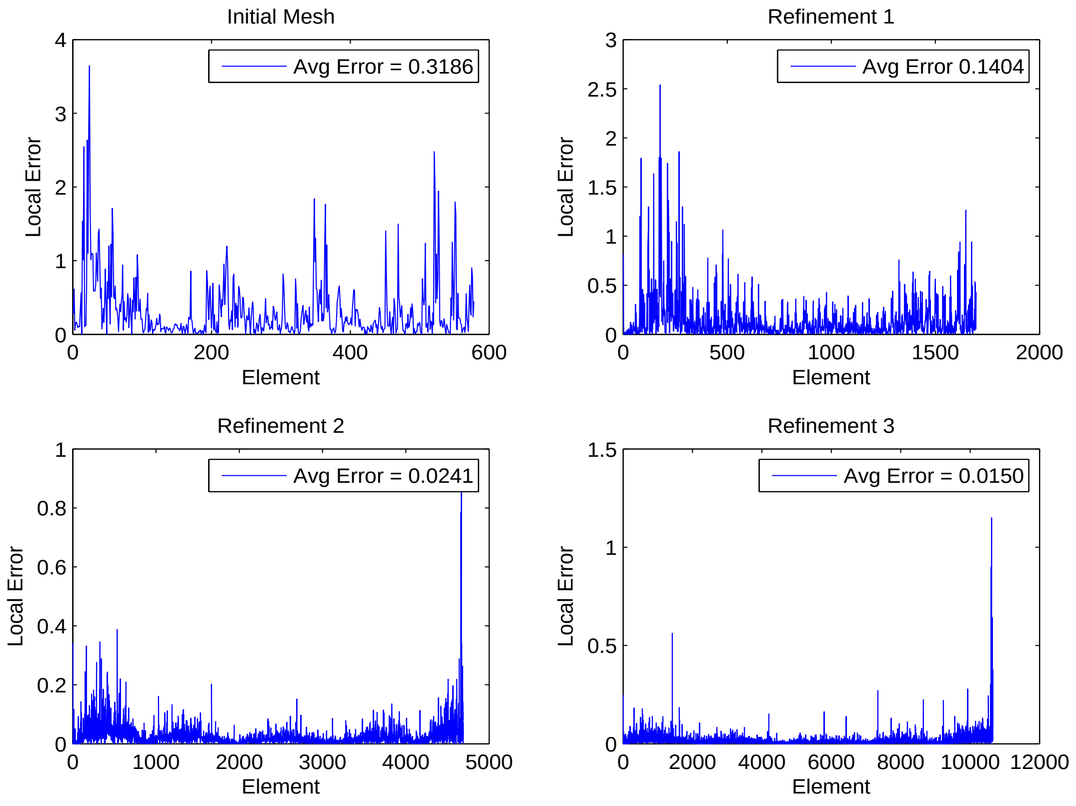

Figure 4.10: Local error on each element for an initial mesh with $N = 18$, $\mu = 1$, $\lambda = 100$, $\rho = 1$, and $\beta = 0.0001$ as well as the first three refinement of this mesh. Notice that the change in the value of λ has had significant effects on the error when compared to Figure (4.7).

limitation on matrix dimensions is reached much earlier, hence only two refinements were possible for this test.

Table (5) shows results for multiple combinations of (μ, λ), all of which confirm the

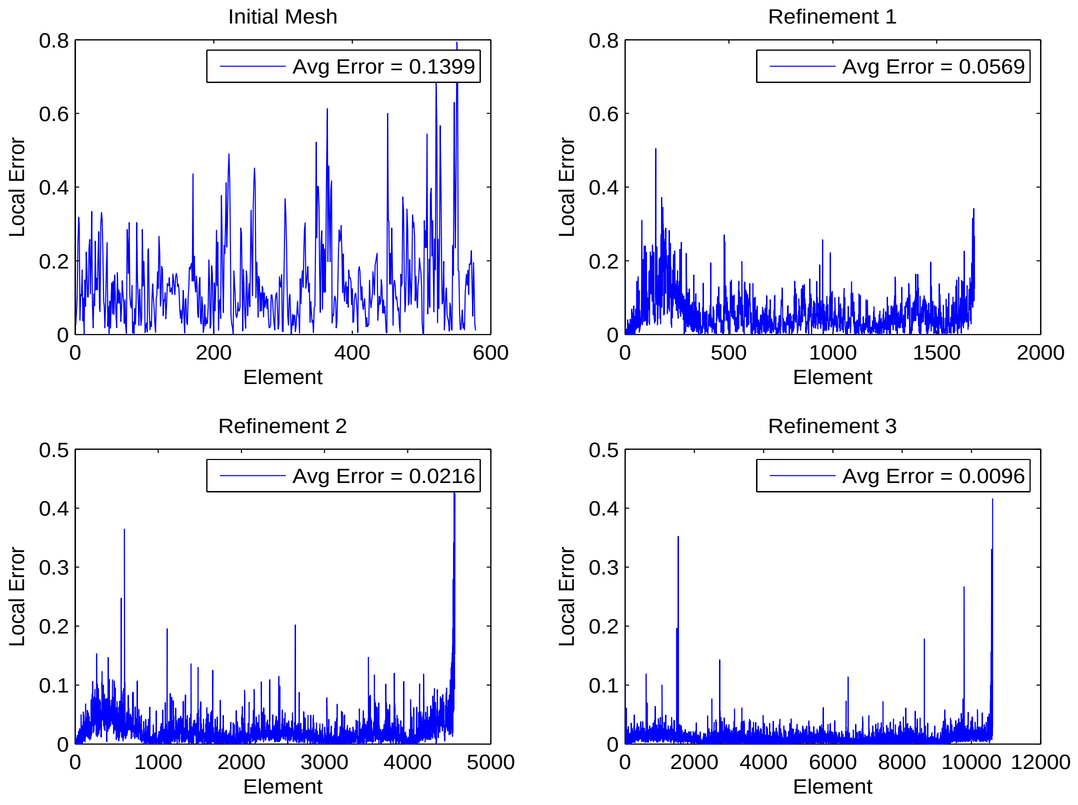

Figure 4.11: Local error on each element for an initial mesh with $N = 18$, $\mu = 100$, $\lambda = 100$, $\rho = 1$, and $\beta = 0.0001$ as well as the first three refinement of this mesh. Notice that with a balance between μ and λ, we achieve an error identical to that in Figure (4.7).

previous suggestions that the ratio of the two coefficients μ and $(\mu + \lambda)$ rather than the individual values of μ and λ themselves are what effect the accuracy of the method. We can see that in any case where $\mu = \lambda$, the error at any particular refinement is identical to any other pair for which $\mu = \lambda$. When μ is much larger than λ we see that there are changes in the error, but they are minimal. Only in the case where λ is much greater than μ do we encounter large increases in the error. This is the case where the $(\mu + \lambda)$ coefficient is in turn much larger than μ.

Based on our understanding of the properties of the lunar soil near the actual lunar landing, these constants were estimated to be $(\mu, \lambda) = (21.3, 49.6)$. This is the case where

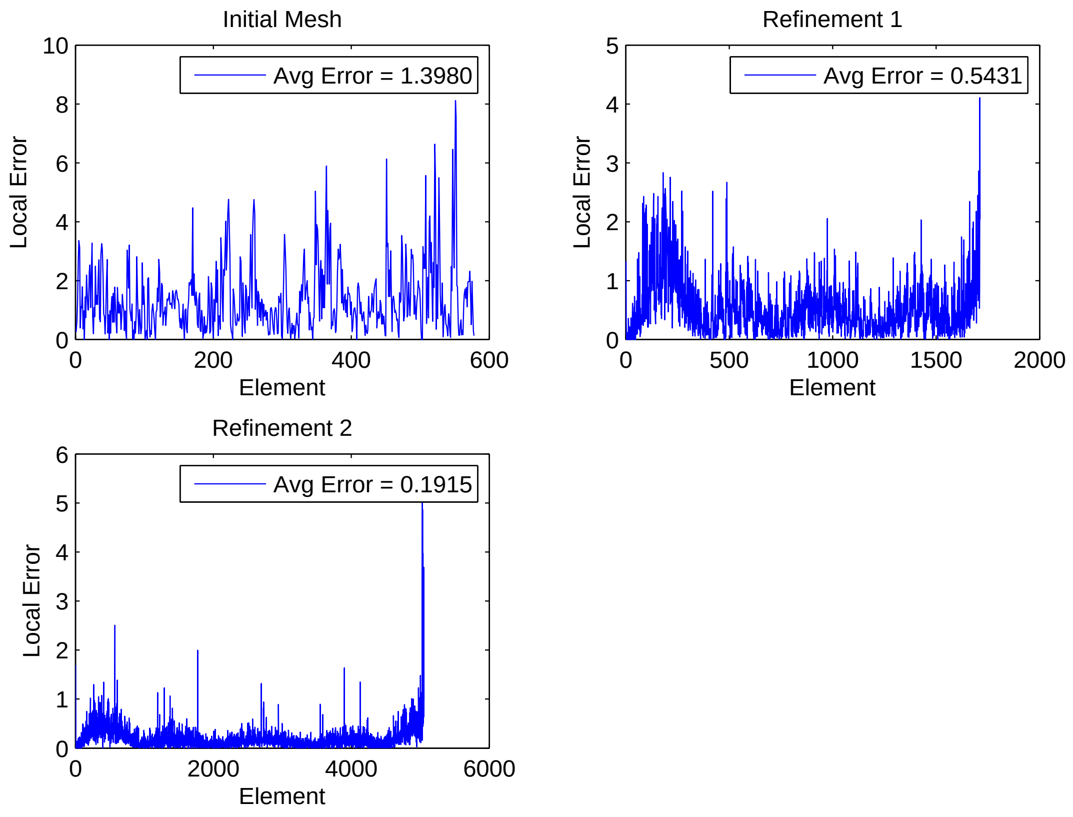

Figure 4.12: Local error on each element for an initial mesh with $N = 18$, $\mu = 1$, $\lambda = 1$, $\rho = 10$, and $\beta = 0.0001$. The error in the initial mesh as well as those in the first two refinements are far worse that those in Figure (4.7). Due to the matrix size limitations of MATLAB, only two refinements were possible under these conditions.

$\lambda > \mu$, which is not ideal. But as we can see from Table (5), λ is not so much greater than μ that we find unacceptable results. In just three refinements we still achieve an average error of nearly 0.01 which was the desired tolerance for the test. Another example is from laboratory tests at Kennedy Space Center where lunar soil with the properties $(\mu, \lambda) = (5.32, 12.4)$ was used in a small scale demonstration. While the values here are different from those in the lunar landing, the ratio of $(\mu + \lambda)$ to μ is nearly the same as the first case. Thus the error results are identical out to four decimal places.

We should remember that every error estimate shown in this section has been found using

Table 5: Average error over all elements in the finite element approximation for various combinations of (μ, λ) with an initial, uniform mesh defined by $N = 18$ and the values $\beta = 0.000001$ and $h = 0.002$.

Refinement	(25,1)	(1,25)	(25,25)	(50,1)	(1,50)	(50,50)	(21.3,49.6)	(5.32,12.4)
Initial Mesh	0.1289	0.2802	0.1399	0.1289	0.3026	0.1399	0.1620	0.1620
1	0.0545	0.0976	0.0569	0.0545	0.1183	0.0569	0.0583	0.0583
2	0.0199	0.0256	0.02156	0.0199	0.0254	0.0215	0.0215	0.0215
3	0.0091	0.0127	0.0096	0.0092	0.0141	0.0096	0.0103	0.0103

(3.52) which is an upper bound on the error, but not the least upper bound. In all cases we can say that the error is no worse than the values shown.

5 CONCLUSIONS AND FUTURE WORK

5.1 CONCLUSIONS

The purpose of this thesis is to propose and analyze numerical methods for solving the porous medium and Navier equations in an attempt to accurately and efficiently model the cratering effect due to the flow of rocket exhaust through a porous medium. Well known methods were used to approximate solutions of both equations. The Crank-Nicolson method, Newton's method, spectral differentiation and even the finite element method are all trusted methods which have been studied in great detail. The idea then becomes how to implement these methods in such a way that we accurately model our particular problem.

A number of key conclusions were made to support our choice and implementation of the Crank-Nicolson method for approximating solutions of the porous medium equation,

- ***Simplicity*** **:** The Crank-Nicolson method is simple to implement.
- ***Accuracy*** **:** Table (3) as well as Figures (4.3) and (4.4) show that the method can achieve results with very good accuracy. Results with error as low as 10^{-13} were found in Figure (4.3).
- ***Efficiency*** **:** Table (3) shows that even for a step size as small as 0.0008 seconds, we can run up to a total time of $T = 20$ ($\approx 25,000$ iterations) in just a few minutes.
- ***Stability*** **:** After countless tests with a variety of different input values, convergence of the method has not appeared to be an issue.

Similar conclusions were made to support our implementation of the finite element method for approximating solutions of Navier's equation,

- ***Simplicity*** **:** Making use of the adaptive refinement method allows us to begin with a uniform mesh which is the easiest to set up. The refinement process creates the new meshes automatically and improves the approximation for us.

- ***Accuracy* :** After three refinements, we found that our approximations were typically around one percent off from the actual solution. If the full version of MATLAB had been used, we surely would have been able to make at least one more refinement and improve on this already acceptable approximation.

However, as is with any numerical approximations, there were some complications with the method. For the porous medium equation we have chosen to use the Chebyshev nodes over a square grid to define our domain. The initial pressure solution is chosen to be defined as a Gaussian which leads to the problem we encountered previously involving negative values around the boundary. The Gaussian function in two dimensions produces a uniform curve which somewhat resembles a cone. This does not fit well with our choice of a square grid in that the boundary nodes are not of equal distance from the pressure source. The corner nodes are farther away than the rest. Polar coordinates would be an option for avoiding this problem.

Another issue with the method used for the porous medium equation is that we plan to use these solutions as forcing terms for the finite element approximation of Navier's equation which is defined on a completely different grid. Interpolating back and forth between these grids may introduce more error into the approximation than is necessary. Using the finite element method to approximate solutions to the porous medium equation could be another option which would avoid both issues mentioned here.

Contributions to the Field

- Writing code to implement the Crank-Nicolson time stepping method with spectral differentiation in space to approximate solutions of the porous medium equation as well as a second program which implements the finite element method to approximate solutions to Navier's equation. These programs are a vital building block for any future work done to explore the formation of cratering as studied in this thesis.

- The coupling of Darcy's law with Navier's equation has, to the best of our knowledge, never been studied before. Darcy's law is used frequently for modeling the flow of a gas or fluid through a porous medium and Navier's equation is used regularly in the field of elasticity. But our work is the first to use solutions of Darcy's equation in the forcing term for Navier's equation to predict cratering.

5.2 FUTURE WORK

The program built during this project was written in MATLAB and all tests were run using the student version. This lead to the key complication of the finite element method which was keeping the size of the stiffness matrix within MATLAB's matrix dimension limitations. Running the code in the full version of MATLAB to get more than just three refinements would yield better results than we have been able to achieve so far. Another option would be to rewrite this program in a high performance language such as C^{++} or FORTRAN. This would allow us to run in parallel and improve efficiency. However these languages do not offer MATLAB's massive library of mathematical operations that greatly simplify the process.

For the porous medium equation it has become clear that a change to polar coordinates should fix the issue with the boundary nodes not being uniformly distributed about the central pressure source.

There are also improvements to be made in the algorithm itself. For one, we can improve the accuracy and efficiency of our approximation by using higher order test functions in the finite element method. Our current method uses the centroid of the element for no reason other than it minimizes the number of nodes added for any single element refinement. A more rigorous study on how the location of refinement nodes effects the error would be useful. We can see in Figure (3.5) that by refining one element you in fact also make a slight refinement to all neighboring elements. So the current process of refining each and every element without taking this into account is not ideal. But it is incredibly inefficient to approximate the error after every single refinement. Some work should be done to balance this out in an attempt to build a smarter algorithm.

The focus of this particular thesis has been building accurate mathematical models for the flow of rocket exhaust through a porous medium. With that completed, we hope to turn our attention to investigating the effects of the exhaust on different qualities of soil. The primary goal is to determine what conditions will cause the soil to fail due to a given pressure load. Assuming we have already found the displacement solution $\boldsymbol{u} = (u, v)^T$ with specific Lamé constants μ and λ then we can build the strain tensor,

$$\boldsymbol{E} = \frac{1}{2}(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T) \tag{5.1}$$

and the stress field,

$$\boldsymbol{T} = \lambda(trace\boldsymbol{E})\boldsymbol{I} + 2\mu\boldsymbol{E} \tag{5.2}$$

where $trace\boldsymbol{E}$ is the trace of the strain tensor $\boldsymbol{E}$, and $\boldsymbol{I}$ is the identity matrix. Now that the stress field $\boldsymbol{T}$ is defined, we can determine if the soil is capable of supporting the pressure load without failing or cratering. The limit for which the soil can support a given load is determined by,

$$\frac{\tau}{\sigma} < \tan\varphi \tag{5.3}$$

where φ is the internal friction coefficient of the soil, σ is any on-diagonal element of $\boldsymbol{T}$, and τ is any off-diagonal element of $\boldsymbol{T}$ in the same row as σ. If (5.3) is violated for any (σ, τ) pair then the soil fails and a crater will form.

Finding what combinations of ρ, μ and λ cause approximations from the finite element method to violate (5.3) is the primary goal for future work on this project. The atmospheric conditions which define the gravitational constant g as well as the magnitude of the rocket exhaust will also play a large part in determining such results.

References

[1] J. Alberty, C. Carstensen, S. A. Funken, and R. Klose, *MATLAB Implementation of the Finite Element Method in Elasticity*, Computing 69, 239-263 (2002)

[2] John P. Boyd, *Chebyshev and Fourier Spectral Methods*, Dover, 25-27 (2001)

[3] G.H. Bruce, D.W. Peaceman, H.H. Rachford, and J.D., *Calculations of unsteady-state gas flow through porous media*, Transactions of the American Institute of Mining and Metallugical Engineers., 5, 79-92 (1953)

[4] Richard L. Burden, and J. Douglas Faires, *Numerical Analysis*, Brooks/Cole, California, 503-512 (2005)

[5] Carsten Carstensen, and Georg Dolzman, *A Posteriori Error Estimates for Mixed FEM in Elasticity*, Numer. Math 81: 187-209, Springer-Verlag (1998)

[6] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson, *Introduction to Adaptive Methods for Differential Equations*, Acta Numerica pp. 001-054 (1995)

[7] Thomas J.R. Hughes, *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Dover (2000)

[8] David Kincaid, and Ward Cheney, *Numerical Analysis: Mathematics of Scientific Computing*, Brooks/Cole, California (2002)

[9] Dongjin Kim, and Wlodek Proskurowski, *An Efficient Approach for Solving a Class of Nonlinear 2D Parabolic PDEs*, Journal of Applied Mathematics and Decision Sciences (2009)

[10] Dani Lischinski, *Incremental Delaunay Triangulation*, Academic Press, Inc. (1993)

[11] G.R. Liu, and S.S. Quek, *The Finite Element Method: A Practical Course*, Butterworth-Heinemann (2003)

[12] Manish Mehta, Nilton O. Renno, Peter G. Huseman, Douglas S. Gulick and Aline Cotel, *The Interaction Dynamics of Pulsed Retro-Rocket Plumes with the Ground During Spacecraft Decent on Mars*, International Planetary Probe Workshop, NASA Report, Pasadena (2006)

[13] Phillip T. Metzger, Christopher D. Immer, Carly M. Danahue, Bruce T. Vu, Robert C. Latta, III, and Mathew Deyo-Svendsen, *Jet-Induced Cratering on a Granular Surface with Application to Lunar Spaceports*, Journal of Aerospace Engineering, (2009)

[14] Phillip T. Metzger, Christopher D. Immer, John E. Lane, and Sandra Clemets, *Cratering and Blowing Soil by Rocket Engines During Lunar Landings*, 6th International Conference on Case Histories in Geotechnical Engineering, (2008)

[15] Phillip T. Metzger, Robert C. Latta, III, Jason M. Schuler, and Christopher D. Immer, *Craters Formed in Granular Beds by Impinging Jets of Gas*, Powders and Grains 2009: Proceedings of the 6th International Conference on Micromechanics of Granular Media, Volume 1145, 767-770 (2009)

[16] Serdal Pamuk, *On the Solution of the Porous Media Equation by Decomposition Method: A Review*, Physics Letters A, 184-188 (2005)

[17] Per-Olof Persson, and Gilbert Strang, *A Simple Mesh Generator in MATLAB*, SIAM Rev. Volume 46, Issue 2, 329-345 (2004)

[18] E. Ramm, E. Rank, R. Rannacher, K. Schweizerhof, E. Stein, W. Wendland, G. Wittum, P. Wriggers, and W. Wunderlich, *Error-Controlled Adaptive Finite Elements in Solid Mechanics*, John Wiley & Sons (2003)

[19] Michael E. Rose, *Numerical Methods for Flows Through Porous Media. I*, Mathematics of Computation, Volume 40, Number 162, 435-467 (1983)

[20] Ronald F. Scott, and Hon-Yim Ko, *Transient Rocket-Engine Gas Flow in Soil*, AIAA Journal, 6, 258-264 (1968).

[21] Lloyd N. Trefethen, *Spectral Methods in MATLAB*, SIAM, 51-59 (2000)

[22] R. Verfurth, *A Review of A Posteriori Error Estimation Techniques for Elasticity Problems*, Comput. Methods Appli. Mech. Engrg. 176: 419-440 (1999)

[23] Ran Zhou, and Pei-Feng Hsu, *Numerical Simulations of Soil Erosion by Landing Rocket Exhaust Plume*, 25th AIAA Applied Aerodynamics Conference, Miami, FL (2007)