Predicting Dust Clouds: Rocket Exhaust Flows in Soil with Triangular Meshes

Group Members: Madison Lytle, Connor Leipelt, Rapha Coutin, Alex Bright

Contents:

Abstract

Here we reproduce the numerical model for soil displacement induced by rocket exhaust flow described in Brain Brennan's 2010 Master's Thesis "Numerical Computations For Pde Models Of Rocket Exhaust Flow In Soil" [1]. To do so, we learn and apply the Crank-Nicolson method, the Chebyshev spectral differences method, and Galerkin Finite Element method in order to reproduce pressure field and displacement field results. MATLAB is used for the numerical implementation. In addition, we briefly explore the potential for non-centroidal node replacement, as included in Brennan's future work.

Project Introduction

Background

In the paper, "Numerical Computations For Pde Models Of Rocket Exhaust Flow In Soil" the primary motivation was to create an accurate model in MATLAB to illustrate the potentially dangerous effects thrusters have when firing on soil, leading to cratering of a surface. These dangerous effects have been observed on both Apollo 12 and Apollo 15 missions. In both missions the lunar modules were able to land, but withstood damage caused by fragments of soil at high velocities, a byproduct of the cratering of the moon's surface [1]. It was noted that the Apollo 15 mission encountered more of these negative effects, in this case dust blowing up from the lunar surface, significantly limiting visibility when landing. Further, the module landed on the edge of a crater created upon landing. With one leg of the module suspended over the edge of the crater, the crew was lucky to have enough stability to complete the mission.

Project Scope and Goals

Our primary goal was to reproduce the results of Brennan's model by learning the methods used in the paper and developing MATLAB code. This meant implementing two main methods that were not taught in class prior to the completion of the project: use of the Crank-Nicolson method to solve for the pressure field and use of Galerikan's method, a finite element method, to solve for the displacement field produced by the rocket. Building off of Brennan's future work, we additionally wanted to explore the potential of non-centroidal node displacement for the FEM used [1]. Brennan mentions that the selection of centroid for successive node replacement was chosen arbitrarily, not for optimality [1].

Implementation and Analysis

Brennan's PDE Model and Nomenclature:

Brennan's model makes use of the following parameters:

- p: pressure
- η: viscosity of the gas
- k: permeability of medium
- ϵ: porosity of medium
- μ, λ: material constants
- $\mathbf{u} = (u, v)$: displacement vector field
- g: gravitational acceleration
- ρ: material density

The base of Brennan's model is Navier's model for volume, given by [1]:

$$\mu\nabla^2 + (\mu + \lambda)\nabla(\nabla\cdot) + f = 0$$

where *f* is the body force term, which can be given by:

$$f = \rho g + \nabla p$$

Note that *p*, the pressure term, is dependent on both time and space. This relationship is captured by the following Porous Medium Equation [1]:

$$\frac{\partial p}{\partial t} = \frac{k}{2\eta\epsilon}\Delta p^2$$

The process of generating pressure solutions, using the Crank-Nicolson scheme, and the process of solving for the displacement field, using Galerikan's method, are detailed in the sections below.

Pressure Solution: Methods and Results

The pressure term used in Navier's model for volume displacement, as discussed in the next section, needs to be calculated using a separate routine that is based on a Porous Medium Equation. Porous Medium Equations, as popularized by Fourier in his Theorie Analytique de la Chaleur in 1822, were originally applied to heat flow problems, but are now used to model an array of dispersive phenomena [2]. The particular form used in Brennan's model, as derived from Darcy's law, is [1]:

$$\frac{\partial p}{\partial t} = \frac{k}{2\eta\epsilon}\Delta p^2$$

This equation is known to be stiff [1],[2]. The Crank-Nicolson method, a well-known finite difference method, was used to solve this equation. This was the method selected by Brennan after running a comparison study on Crank-Nicolson and Heun's method, finding the latter to display instability for large step sizes making it less suitable for stiff problems. In its general form, the method can be expressed by:

$$y_{k+1} \quad = y_k + \tfrac{h}{2}[F(y_k) + F(y_{k+1})]$$

Applied to this specific problem, we obtain the form:

$$p_{k+1} = p_k + \frac{h\beta}{2}[\Delta p_k^2 + \Delta p_{k+1}^2]$$

with $\beta = \dfrac{k}{2\eta\epsilon}$.

To account for the nonlinear Δp^2 terms, a Chebyshev spectral method was used. This method produces a differentiation approximation by interpolating between the roots of the Chebyshev polynomials (of the form $T_N(x) = \cos(N \arccos(x))$) across the domain and then differentiating [3]. A general formula is given by Trefethen in the 1994 text *Finite Difference and Spectral Methods for Ordinary and Partial Differential Equations* [4]. Note that this general form is for the case of zero-Dirichlet boundary conditions. Conveniently, these are the boundary conditions used by Brennan. Our differentiation matrix then takes the form:

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

with

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

This matrix can be expanded given us the value of the Laplacian needed, as

$$\frac{\partial^2}{\partial x^2} \approx I$$

$$\frac{\partial^2}{\partial y^2} \approx D^2 \otimes I$$

$$\Delta_s \approx C = I \otimes D^2 + D^2 \otimes I$$

Now we can finally apply the Crank-Nicolson method. As this method is implicit, we recast the problem as a root-finding problem and use Newton's method. This implementation of Newton's method takes the form:

$$z_{i+1} = z_i - F(z)/J(z)$$

with

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

and

$$J(z) = I - h\beta C z$$

Note that the implicit method requires two starting conditions. Following Brennan, we used a 2D Gassian for our first initial condition, given by [1]:

$$P_0 = \exp(-\frac{(x_i - 0.5)^2 + (y_i - 0.5)^2}{\sigma^2})$$

This is meant to generate an approximate diffusive shape that reflects how the highest pressure output of the thruster nozzle will be concentrated in the center of the nozzle. The second initial condition is generated using one step of Heun's method [1].

Displacement Solution: Methods and Results

Finite Element Method:

The general idea of a Finite Element Method is to break the problem down into smaller, simpler problems. In our case, this is seen in practice by both creating a weaker, and thus easier, form of the equation we hope to solve as well as breaking the domain on which we solve the problem down into smaller, more manageable chunks. We will begin with a discussion of the discretization of the domain.

The first step is to discretize the grid as per usual. For one-dimensional domains, this is done as usual. However, for two- and higher-dimensional domains, we discretize each dimension and then create a set of points made of each possible combination from the discretized domains. However, we will only be dealing with the method in two dimensions. This creates the usual rectangular grid of our domain, but the Finite Element Method takes things a step further. The Galerkin method used in this dissertation creates a Delaunay triangulation of the domain [1]. The idea is to create triangles that disjointly sum up to the entire domain which use the discretized points as the nodes, or vertices, of the triangles. What makes Delaunay triangles different from other triangles is that they are chosen from our points such that no other point is inside the circumcircle of any given triangle. This ensures a better spread of the triangles across the domain, and has the added benefit of maximizing the interior angles of each triangle, which prevents the creation of long, thin triangles on our domain. This is preferred, as with more equally angled triangles we get a better approximation of the data within the triangle – longer, skinner triangles have the possibility of covering a border range of domain values, as thus are more likely to have high variation. As a good example, the paper considers the domain Ω = [0,1]×[0,1] and n =3 [1]. As mirrored in the code, we create a Locations matrix consisting of our discretized domain. As seen in the image, we then label each point of our domain with a number one through nine. Finally, an Elements matrix is created, which contains the information about the triangles we have created. This can be done in matlab simply by applying the Locations matrix to the delaunay() function, which will output an Elements matrix consisting of Delaunay triangles.

$$Locations = \begin{bmatrix} 0 & 0 \\ 0.5 & 0 \\ 1 & 0 \\ 0 & 0.5 \\ 0.5 & 0.5 \\ 1 & 0.5 \\ 0 & 1 \\ 0.5 & 1 \\ 1 & 1 \end{bmatrix} \qquad Elements = \begin{bmatrix} 4 & 2 & 5 \\ 1 & 2 & 4 \\ 5 & 2 & 6 \\ 6 & 2 & 3 \\ 8 & 4 & 5 \\ 7 & 4 & 8 \\ 5 & 6 & 8 \\ 8 & 6 & 9 \end{bmatrix}$$

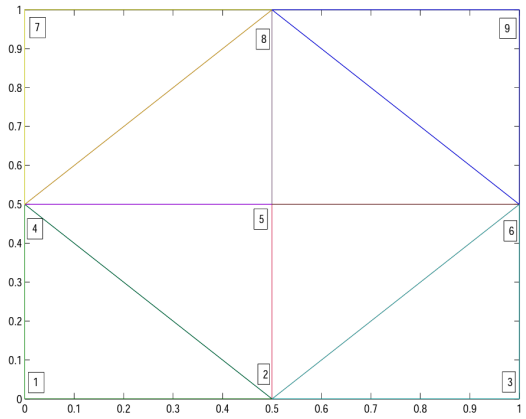


Fig. 1: Equations (3.9), (3.11), and Fig 3.2 from Brennan 2010 [1]

Here the columns each represent a dimension and each row is a point in our domain, creating $n^2 = 9$ equally spaced points. The Elements matrix is also created, where the columns of each row correspond to the nodes, or vertices, of the triangle in question. For example, the top left triangle has vertices (0, 1), (0, 0.5), and (0.5, 1), so it corresponds to the 6th row of our Elements matrix, which are the nodes 7, 4, and 8.

In addition to the Delaunay triangulation of the domain, the Finite Element Method also calls for creating a weak form of the problem. We begin with the strong Navier-Lamé problem:

$$\mu\Delta\boldsymbol{u} + (\mu + \lambda)\nabla(\nabla\bullet\boldsymbol{u}) = \boldsymbol{f}$$

Multiplying the differential by a smooth test function and integration over the domain, we are able to achieve a smooth form of the problem, which looks like

$$a(\boldsymbol{w},\boldsymbol{u}) = (\boldsymbol{w},\boldsymbol{f})$$

where $(\boldsymbol{w},\boldsymbol{u})$ is the L2 inner product over the Hilbert space. The smooth test function Brennan chooses for this problem is the Linear Hat function $\phi_{i,j}$, which is defined on each node within each triangle. For each triangle, we have three functions. Each function is the simple linear p1 function that is equal to 1 at the node of choice and equal to 0 at the other two nodes, and then linearly interpolating between those three points.

Now that we have done the preprocessing for the problem, we can create our global stiffness matrix A and global load vector b to solve $A\boldsymbol{u} = b$ and get our approximate solution to the problem. To assemble these, we first compute their local versions over each triangle on the domain and then sum to get the solution over the entire domain. The local stiffness matrix, A^K, is computed as follows [1]:

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

where R_K is the matrix representation of an operator which constructs the local strain vector on each triangle and C is a constant matrix created from constants relating to the soil type we are dealing with. This is then adjoined to the other local stiffness matrices further along the diagonal

of the larger global stiffness matrix A. We then also need to create the global load vector from each local load vector, b_K, which are calculated as follows [1]:

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

Here, we approximate the integral over each individual triangle by computing the value of the forcing at each node on the triangle and distributing that equally across the triangle itself.

Exploration of Nodal Adaptive Refinement

We coded up a version of Fig 3.2 from Brennan [1] with the Delaunay triangulation, as shown in Fig. 2 below.

This grid refinement is also a solution to the requirements of Delaunay triangulation, as the circumcircles of all triangles do not include any other nodes [1].
This means that there are multiple solutions to our grid requirement creating a degenerate case. It is important to note that since they both meet the criterion, they both maximize the interior angles and will efficiently split up our domain into better digestible pieces.

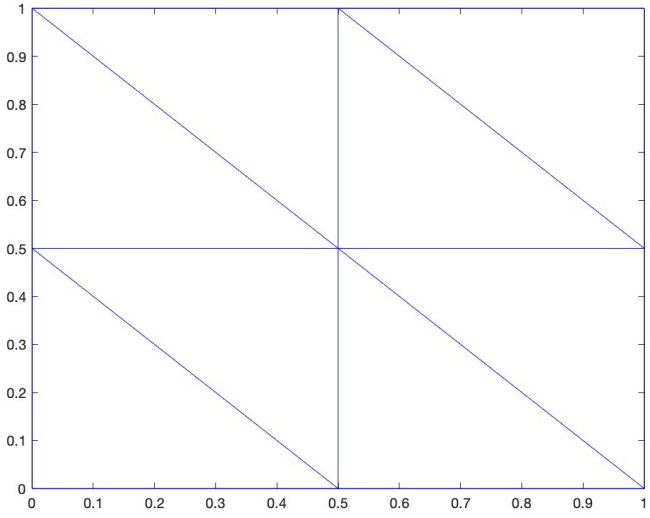


Fig. 2: 9 Point Triangular Mesh

Another option is to create an adaptive grid refinement [1]. This change in domain refinement would occur after every iteration of the finite element method. To decide which triangles warrant a refinement, an error analysis is necessary. We can run an error analysis by calculating the residual of our approximation. The residual is the difference between the right and left sides of the strong equation when our approximation solution is used. If we let $A(\mathbf{u}_h) = b$ be our finite element equation and $\hat{A}(\mathbf{u}) = \hat{b}$ represent the strong equation defined by (3.12) where [1]:

$$\hat{A} = \mu\nabla^2 + (\mu + \lambda)\nabla\nabla^T$$
$$\hat{b} = -f$$

We can then say the residual is:

$$R(u_h) = |\hat{A}(\mathbf{u}_h) - \hat{b}|$$

The new error estimate is dependent on the residual and the size of the grid spacing, h, which we define as the largest diameter among all elements in the mesh. We now have our error to be defined as:

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

where S_c is the stability factor and C_i is an interpolation constant. We can do one last refinement by saying that the error is proportional to $h^2 R(\mathbf{u}_h)$ since S_c and C_i are constants. This means:

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

So we can use $h^2 R(\mathbf{u}_h)$ to estimate the error! [1] The result of the above equation is a column vector containing the error approximation on each node of our mesh. The error for each triangle is found by summing the three errors for the corresponding nodes.

Now that we've created a way to analyze the error of our grid after each step in our finite element method, we can set a tolerance criterion and if any of the triangles on our grid obtains and error larger than our criterion, a new node would be place at the circumcenter creating a new set of refined triangles.

An example of this refinement applied to Fig. 2 is shown in Fig. 3. This refinement assumed that all triangles in our grid did not meet the tolerance criterion resulting in eight new nodes.

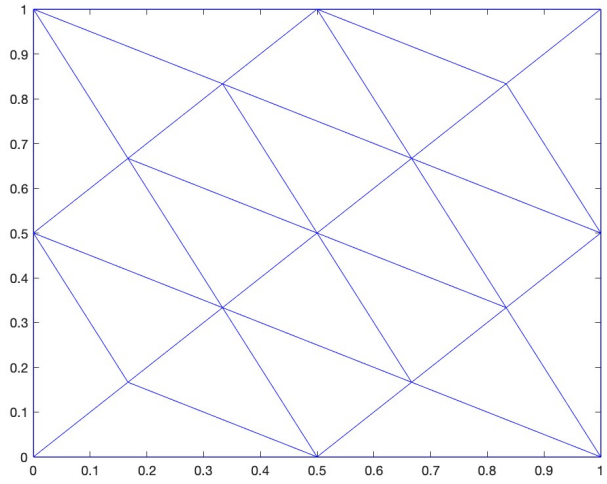


Fig. 3: One Step of Adaptive Node Refinement Applied to Fig. 2

Results and Conclusions

Pressure Solution Results:

The pressure results were tested against Brennan's for the same parameter sets. Both even and odd parities of interior points were tested (8 and 9 points) to compare against Brennan's paper with the selections of h = 0.001, T = 20, and β = 0.000001 at time steps 0, 6000, 13000, and 20000. General agreement in shape and magnitude was found. for all cases. Example plots can be found in the Appendix. Fig. 5 in the Appendix gives Brennan's pressure solutions with 9 grid points at time steps 0 and 13000. Fig. 6 gives the outputs from our code with the same parameters.

Displacement Field Results:

The equation was solved for an experiment that was performed with 100 Newtons of initial pressure, where the solution is taken after a single time step of 0.001 seconds. We discretize the axes into 50 points, for a total of 2,500 points on the domain.

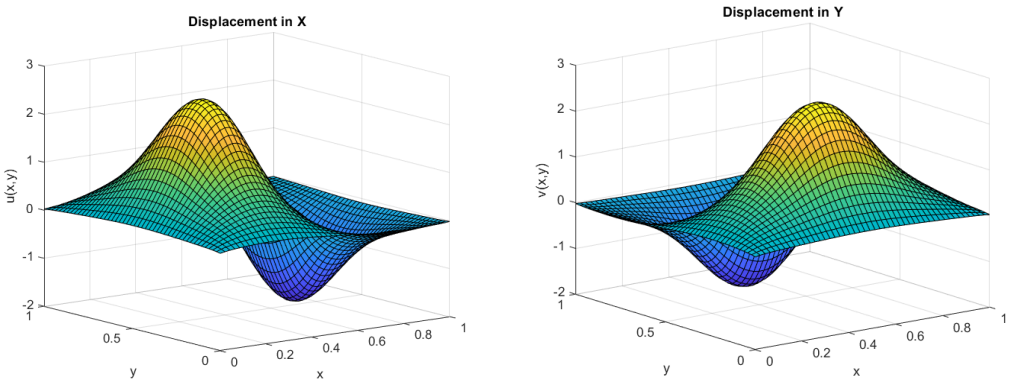


Fig. 4: Displacement Results

Pictured above are our matlab figures graphing the solutions to the two-dimensional Navier-Lamé problem, where the two dimensions are displacement on the x and y axes. The point of initial pressure from the rocket is the center of the domain. Near the center of the domain is where we see the highest absolute values of displacement, which is what we'd expect. The closer the soil is to the point of initial pressure, the more it is affected by the pressure. Additionally, we see that the two figures are the same, simply rotated by 90 degrees. This is also expected, as this creates a circular displacement field when graphed on the xy-plane. Thus, we note that with non-zero, rotationally symmetric displacement we see evidence of cratering occurring.

Potential Future Works

For the pressure model, the computational efficiency must be improved for significant grid refinement. A resolution of 9 interior points was used for the pressure model before being fed into the displacement code. Increases beyond 30 points resulted in unrealistic computation times.

Another possible addition to the project would be to use a higher order test function in the finite element method. This would improve the accuracy and efficiency of our approximation and allow us to have a more precise solution. We could also implement the nodal refinement as the paper did to create a more accurate representation of the model.

Works Cited

[1] Brian Brennan. “Numerical Computations For Pde Models Of Rocket Exhaust Flow In Soil”. In: 2010.

[2] Juan Luis Vazquez. The Porous Medium Equation: Mathematical Theory. Oxford University Press, Oct. 2006. isbn: 9780198569039. doi: 10.1093/acprof:oso/9780198569039.001.0001. url: https://doi.org/10.1093/acprof: oso/9780198569039.001.0001.

[3] Randall J. LeVeque. Finite Difference Methods for Ordinary and Partial Differential Equations. Society for Industrial and Applied Mathematics, 2007. doi: 10.1137/1.9780898717839. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898717839. url: https://epubs.siam.org/doi/abs/10.1137/1.9780898717839

[4] Lloyd N. Trefethen, *Finite Difference and Spectral Methods for Ordinary and Partial Differential Equations*, unpublished text, 1996, available at http://people.maths.ox.ac.uk/trefethen/pdetext.html

[5] Wikipedia contributors. Derivation of the Navier–Stokes equations — Wikipedia, The Free Encyclopedia. [Online; accessed 7-June-2023]. 2023. url: https: //en.wikipedia.org/w/index.php?title=Derivation_of_ the_Navier%E2%80%93Stokes_equations&oldid=1151019798.

Appendix

Pressure Results Figure:

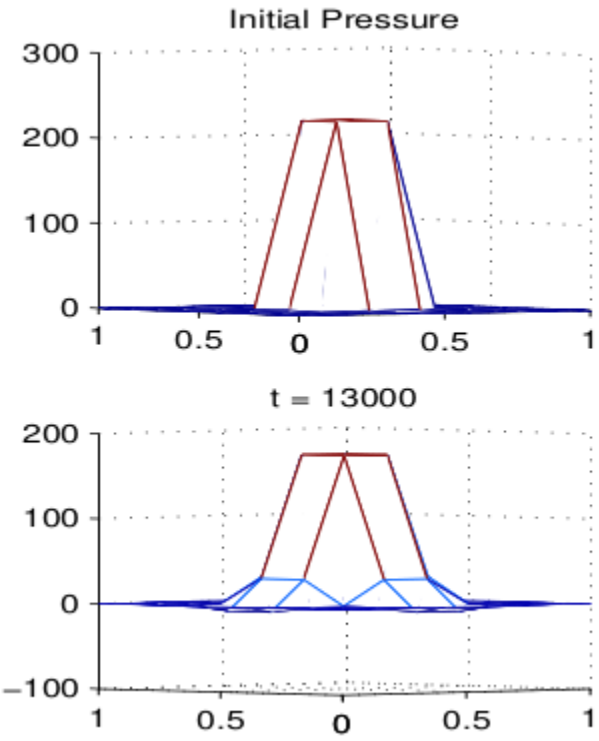
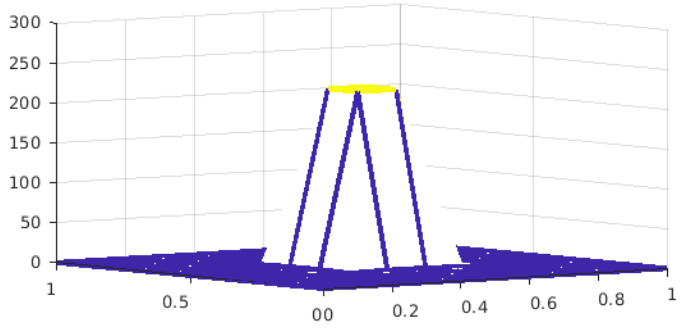


Fig 5: Brennan's Pressure Solutions at Time Steps 0 and 13000, Fig 4.2 [1]



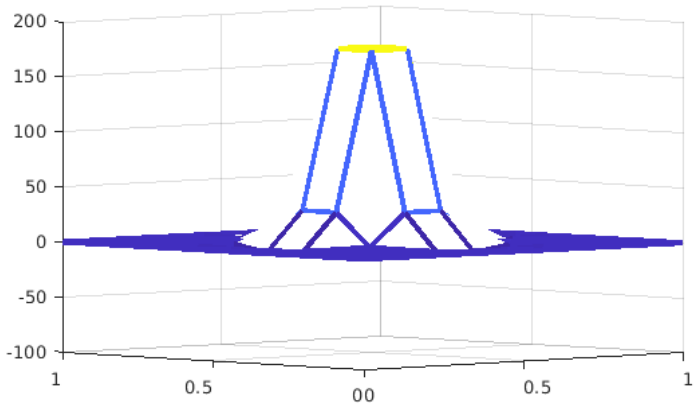


Fig 6: Our Pressure Solutions at Time Steps 0 and 13000

Pressure Solution Code:

```
function [XX,P,px,py] = Pressure(N,h,T,beta,plot_flag)

%% Params
% N = 9;
% h = 0.001;
Amp = 1000;
% T = 20;
% beta = 0.000001;

t = 1; % for plotting (fig 4.2)

%% Set-Up

% Set up Chebyshev difference
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;

% Dirchlet BCs:
M(1,:) = 0;
M(N+1,:) = 0;

% Construct FDM mat
h2 = x(2)-x(1);
I = eye(N+1);
```

```
% [d,x_out] = cheb(N);
% d(1,:) = 0;
% d(N+1,:) = 0;
% d = d^2;
D = kron(I,M^2) + kron(M^2,I);


% ICs:
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) = reshape(u, [(N+1)^2,1]);%Mat2Vec(u) % *** use reshape instead

%% 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

%% Get Partials
px = kron(I,M)*P;
py = kron(M,I)*P;
XX = X;
```

```
%% Plotting
P = reshape(P(:,t), [(N+1), (N+1)]);

if plot_flag
    figure()
    mesh(linspace(0,1,N+1),linspace(0,1,N+1),P)
end

%% Function
function [D,x] = cheb(N)
    x = cos(pi*(0:N)/N)';
    c = [2; ones(N-1,1); 2].*(-1).^(0:N)';
    X = repmat(x,1,N+1);
    dX = X-X';
    D  = (c*(1./c)')./(dX+(eye(N+1)));
    D  = D - diag(sum(D'));
end

end
```

Initial Delaunay Grid Refinement Demo

```
clear vars; close all; format long;
% Math 452 Project Displacement Code
% Using Navier-Lame equation model for displacement
%% Preprocessing Piece
% Parameters
num_x_inputs = 3;
column_length = num_x_inputs^2;
region_points = linspace(0,1,num_x_inputs);
dimension_num = 2;
Locations_matrix = zeros(column_length,dimension_num);
for j = 1:column_length
   i = ceil(j/num_x_inputs);
   Locations_matrix(j,:) = [region_points(j - 3*(i-1)),region_points(i)];
end
x_inputs = Locations_matrix(:,1);
y_inputs = Locations_matrix(:,2);
Elements = delaunay(x_inputs,y_inputs);
% Since we are working in an equidistant shape, there are multiple
% solutions
% The solution from the paper has the Elements come out to Element_2 below:
% Elements_2 = [4,2,5;1,2,4;5,2,6;6,2,3;8,4,5;7,4,8;5,6,8;8,6,9];
% We will use the Elements that we've founded
triplot(Elements,x_inputs,y_inputs)
% triplot(Elements_2,x_inputs,y_inputs)
```

Displacement Solution Code:

```
clearvars; clc;
n = 50;
h = 0.001;
T = 20;
beta = 0.000001;
mu = 1;
lambda =1;
rho = 1;
g = 9.81;

%% Preprocessing
E = 2*(n-1)^2;
H = 1/(n-1);
X = 0:H:1;
Y = X;
[x0,y0] = meshgrid(X,Y);

subpoints = linspace(0,1,n);
Locations = zeros(n^2,2);

for i = 1:n
for j = 1:n
Locations(j+(i-1)*n,2) = subpoints(i);
Locations(j+(i-1)*n,1) = subpoints(j);
end
End

x = Locations(:,1);
y = Locations(:,2);

Elements = delaunay(x,y);
Ne = length(Elements);

triplot(Elements,x,y);

%% FEM SOLVER
%% Stiffness Matrix
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

A = zeros(2*n^2,2*n^2);

for k = 1:Ne
  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);
  C = mu*[2,0,0;0,2,0;0,0,1] + lambda*[1,1,0;1,1,0;0,0,0];
  PhiGrad = [1,1,1;x(1),x(2),x(3);y(1),y(2),y(3)]\[zeros(1,2);eye(2)];

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


% Construct R_k
R = zeros(3,6);
R([1,3],[1,3,5]) = PhiGrad';
R([3,2],[2,4,6]) = PhiGrad';

Ak = det([1,1,1;x(1),x(2),x(3);y(1),y(2),y(3)])/2*R'*C*R;
A(Global,Global) = A(Global,Global) + Ak;
end

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

%% Create b Vector
[XX,P,px,py] = Pressure(9,h,T,beta,true);
[xI,yI] = meshgrid(XX,XX);

% reformat px and py
px = reshape(px(:,end),[10 10]);
py = reshape(py(:,end),[10 10]);

Px = interp2(xI(1,1:10),yI(1:10,1),px,x0,y0,'spline');
Py = interp2(xI(1,1:10),yI(1:10,1),py,x0,y0,'spline');
```

```
Px = reshape(Px,[n^2 1]);
Py = reshape(Py,[n^2 1]);
b = zeros(2*n^2,1);

for k = 1:Ne
  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);

  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;
end

b(DirBC) = [];
%%

opts.SYM = true;
opts.POSDEF = true;
U = linsolve(A,b,opts);
u = U(1:2:size(U,1)-1,1);
v = U(2:2:size(U,1),1);

%% Plotting
size = sqrt(size(v,1));
plotu = reshape(u,[size size]);
plotv = reshape(v,[size size]);
figure(3)
surf(linspace(0,1,size),linspace(0,1,size),plotu);
xlabel('x');ylabel('y');zlabel("v(x,y)");title("Displacement in Y");
figure(4)
surf(linspace(0,1,size),linspace(0,1,size),plotv);
xlabel('x');ylabel('y');zlabel("u(x,y)");title("Displacement in X");
```

Nodal Mesh Refinement

```
clear vars; close all; format long;
% 9 point grid refinement example
% Using Locations from M452 Proj_Displacement Code
% Compute a new refinement of the grid agreeing with the centroid
% constraint
```

```
%NEW VERSION
Locations_matrix =  [               0                   0;
                    0.500000000000000                   0;
                    1.000000000000000                   0;
                                    0   0.500000000000000;
                    0.500000000000000   0.500000000000000;
                    1.000000000000000   0.500000000000000;
                                    0   1.000000000000000;
                    0.500000000000000   1.000000000000000;
                    1.000000000000000   1.000000000000000];

   Elements =      [1     2     4;
                    2     5     4;
                    4     5     7;
                    5     6     8;
                    7     5     8;
                    2     3     5;
                    8     6     9;
                    5     3     6];
% Elements from paper
% Elements = [4,2,5;1,2,4;5,2,6;6,2,3;8,4,5;7,4,8;5,6,8;8,6,9];
x_vals = [];
y_vals = [];
for i = 1:length(Elements(:,1))
   triang = Elements(i,:);
   for j = 1:length(Elements(1,:))
       x_val = Locations_matrix(triang(j),1);
       y_val = Locations_matrix(triang(j),2);
       x_vals = [x_vals x_val];
       y_vals = [y_vals y_val];
   end
end
x_vals = reshape(x_vals, 3, length(Elements(:,1)));
y_vals = reshape(y_vals, 3, length(Elements(:,1)));
new_x_vals = sum(x_vals)./3;
new_y_vals = sum(y_vals)./3;
new_points = [new_x_vals; new_y_vals]';
all_points = [Locations_matrix; new_points];
x_inputs = all_points(:,1);
y_inputs = all_points(:,2);
New_Elements = delaunay(x_inputs,y_inputs);
% triplot(Elements,x_inputs,y_inputs)
% hold on;
triplot(New_Elements,x_inputs,y_inputs)
```