Simulating Elasticity in Two Dimensions

Shea Yonker

Host: Chris Deotte

Thursday, June 2nd, 2016

11:00 AM

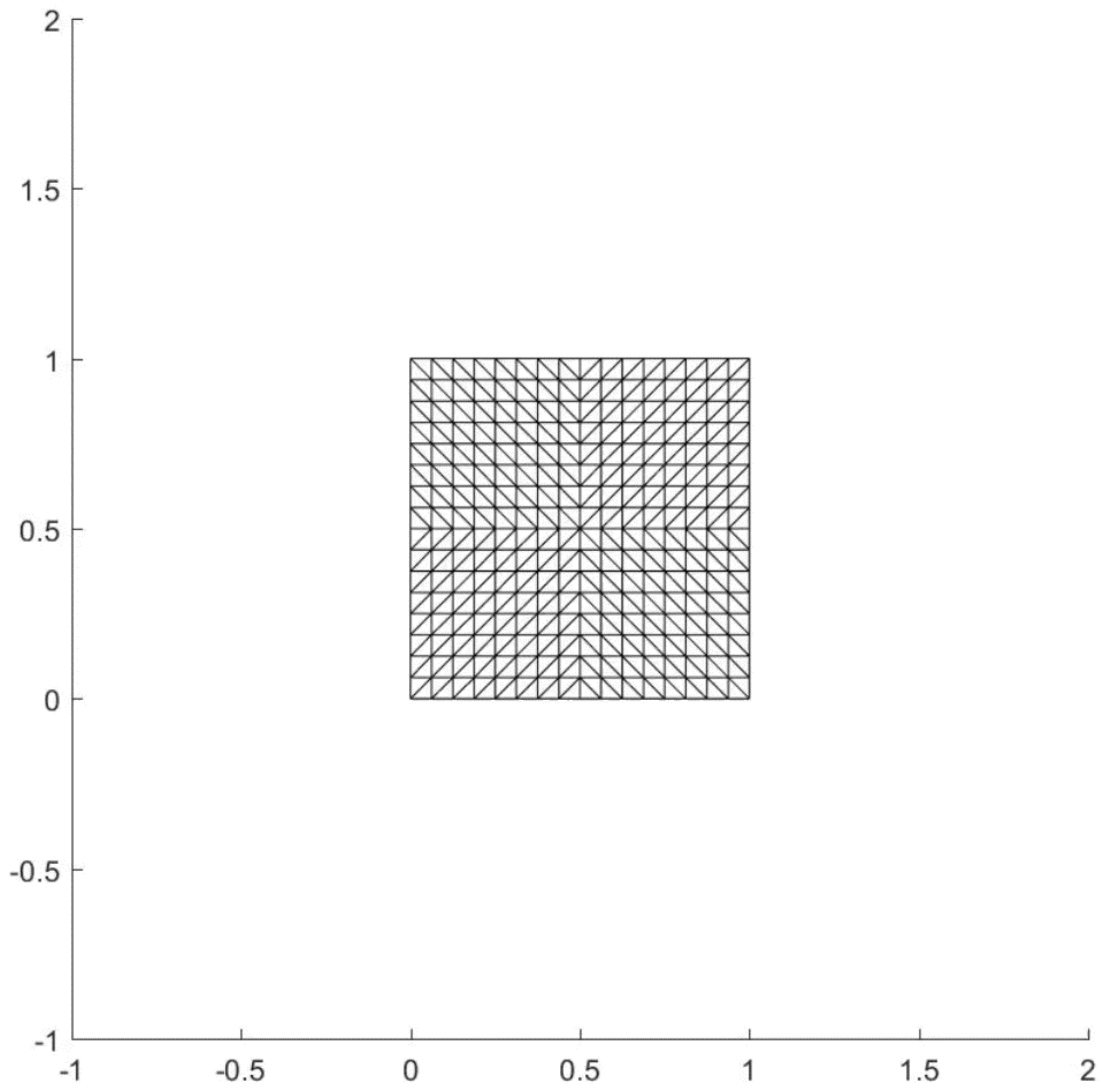
AP&M 2402

Abstract

Accurate simulations of elasticity properties can be constructed by solving second order elliptic boundary value problems which have been approximated using finite elements. This talk will examine the process of converting the given PDE into a weaker form and applying the Galerkin Method. In addition, novel MATLAB programs will be introduced, which will display a visual depiction of an object after force is applied, given a subdivision of the shape into regular or irregular triangles, a Dirichlet boundary condition, and a two dimensional force function.

Our Goal!

To examine the effects of force on an object given a two dimensional force function and a boundary condition

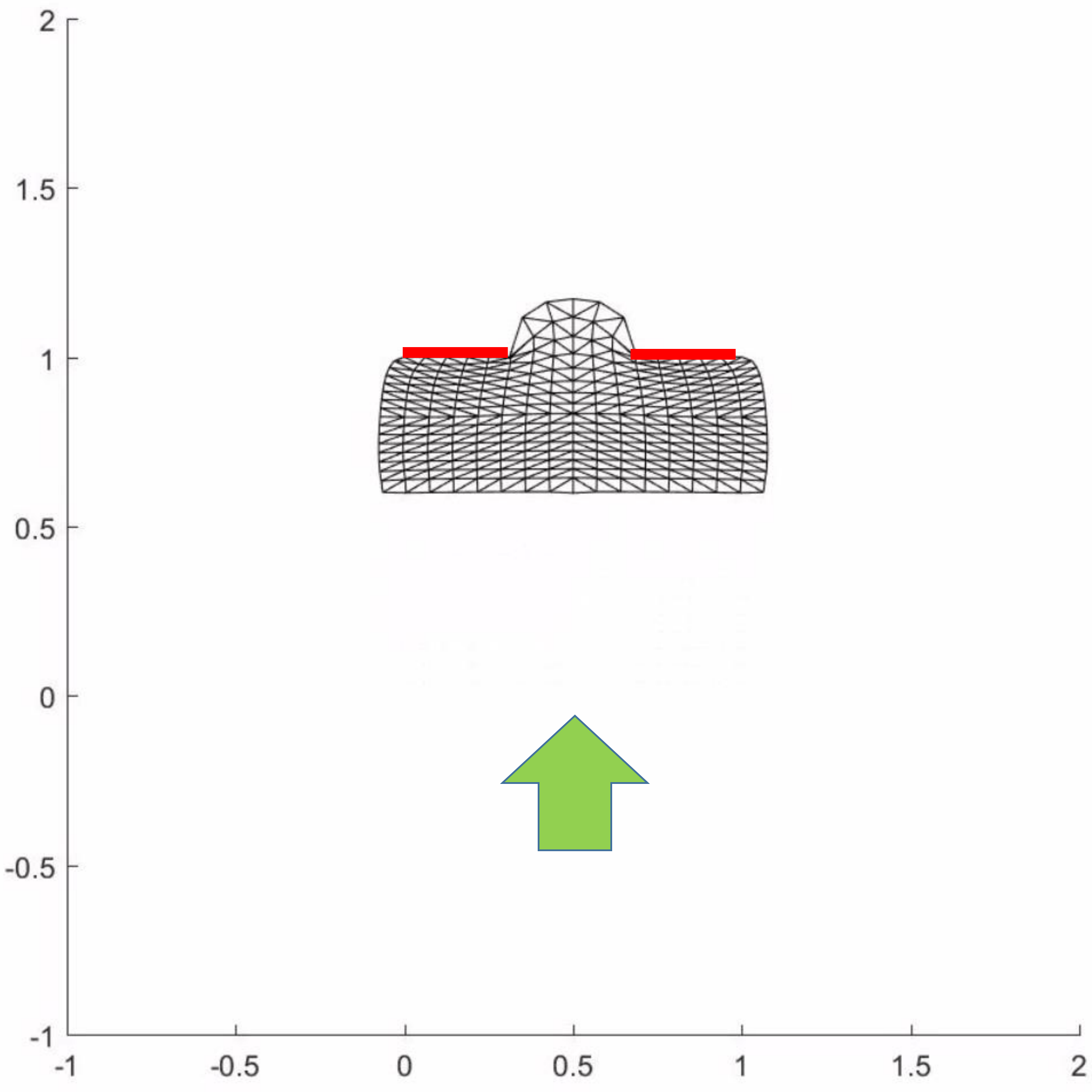


Notice that we have

$f(x,y) = [0;1]$
(when $y = 0$)

&

At $x \leq 1/3$ & $y = 1$
or $x \geq 2/3$ & $y = 1$
The displacement is zero.



Process

1. PDE $\longmapsto Ax = b$

2. Solve $Ax = b$ for x
   (x will give displacement at each point)

3. Add displacement and plot for visual

Our PDE

Is the second order elliptic boundary value problem:

$$- \bigtriangledown \cdot (a(x,y) \bigtriangledown u) + b(x,y) \cdot \bigtriangledown u + c(x,y) u - f(x,y) = 0 \ (in\ \Omega)$$

With boundary conditions

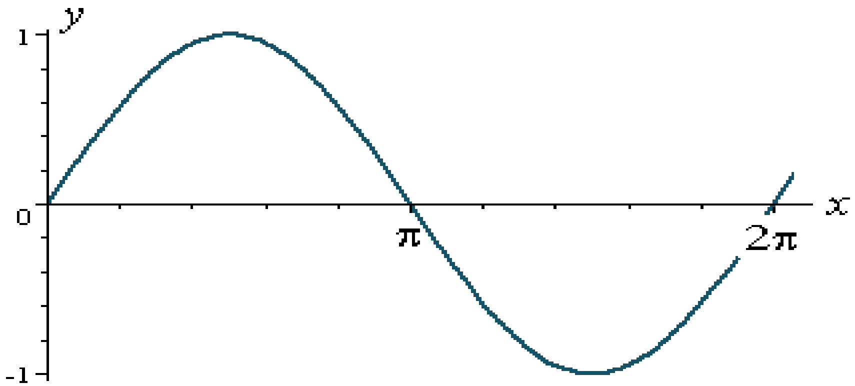
$$(a(x,y) \bigtriangledown u) \cdot n = g_N(x,y) \quad (on\ \partial\Omega_N)$$

$$u = g_D(x,y) \quad (on\ \partial\Omega_D)$$

Constants: $a = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad b = \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix} \quad c \in \mathbf{R}$

Our PDE

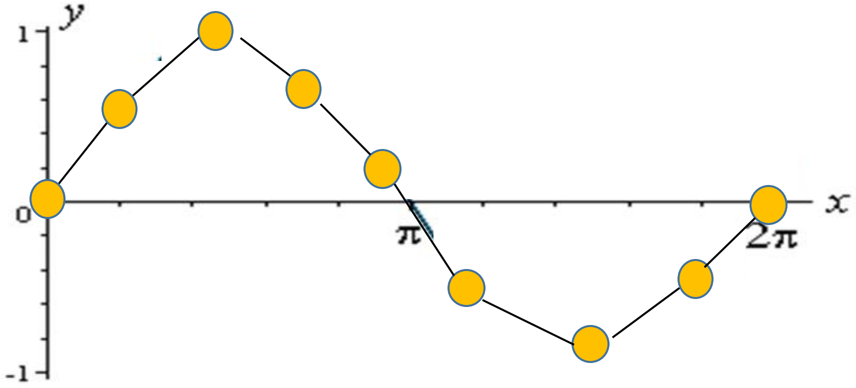
Is not possible to solve for a function $u(x)$ as it would require a solution with infinite degrees of freedom



$$u(x) = \sin(x)$$

A smooth curve with infinite degrees of freedom

So we use a Galerkin Approximation to partition Ω into finite elements



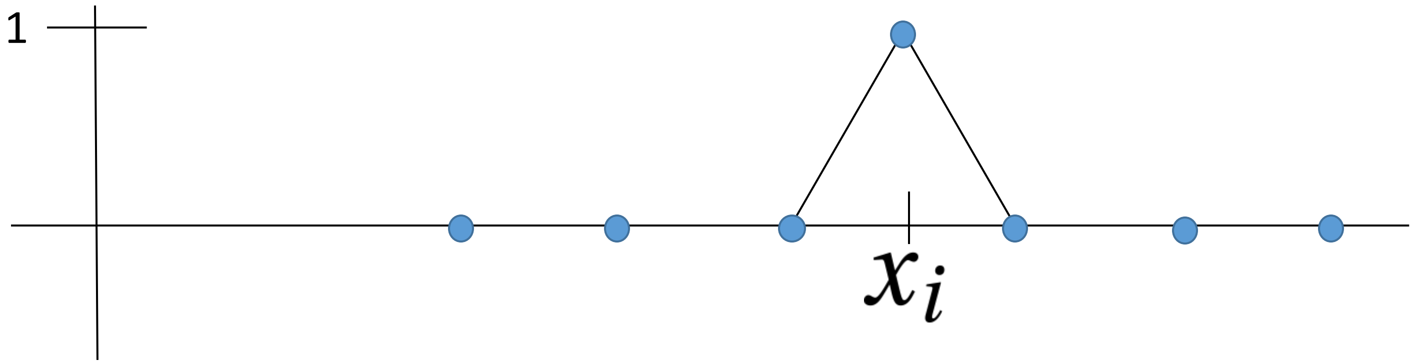
$$u_h(x)$$

Only 9 degrees of freedom

Galerkin Approximation

To do this we use basis functions $\phi_i(x)$ for $i = 1, \ldots, n$ defined as below

$$\phi_i(x) = \begin{cases} 1 & x = x_i \\ 0 & x \neq x_i \end{cases}$$



So $u \approx u_h = \sum_{k=1}^{n} \alpha_k \phi_k$ will be our Galerkin Approximation

For example if $u(x_7) = 0.3$ and $n = 9$, when $\alpha_7 = 0.3$

$$u_h(x_7) = \alpha_1(0) + \alpha_2(0) + \alpha_3(0) + \alpha_4(0) + \alpha_5(0) + \alpha_6(0) + 0.3(1) + \alpha_8(0) + \alpha_9(0) = 0.3$$

However when substituting our new approximation into a portion of our PDE, we can see that we will run into some problems

$$-\nabla \cdot (a(x,y)\nabla u) + b(x,y)\cdot\nabla u + c(x,y)u - f(x,y) = 0 \ (in\ \Omega)$$

$$-\nabla\cdot\nabla u = f$$

$$\nabla\cdot\nabla = \Delta = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

$$-\Delta u_h = f$$

$$-\Delta\sum_{k=1}^{n}\alpha_k\phi_k = f$$

Doesn't make sense

$$-\sum_{k=1}^{n}\alpha_k\Delta\phi_k = f$$

Because of this we need to convert our PDE into its Weak Form

We have

$$-au'' + bu' + cu = f \qquad\qquad u: R \rightarrow R \quad u \in C^2$$

So instead of that u we'll find $u \in H^1$ such that

$$\int_\Omega (-au''v + bu'v + cuv)dx = \int_\Omega (fv)dx$$

for every $v \in H^1(\Omega)$

with $v = 0$ on $\partial\Omega_D$

Now since $u \in H^1$, we can use $u = u_h = \sum_{k=1}^{n} \alpha_k \phi_k$

Simplifying

$$\int_\Omega (-au''v + bu'v + cuv)dx = \int_\Omega (fv)dx$$

Using Green's Identity: $-\int_\Omega (au''v)dx = \int_\Omega (au'v')dx - \int_{\partial\Omega} (au'v)dx$

$$\int_\Omega (au'v' + bu'v + cuv)dx = \int_\Omega (fv)dx + \cancel{\int_{\partial\Omega_N} (au'v)dx}$$

(Is zero by our Neumann boundary condition)

So using $u = u_h = \sum_{k=1}^{n} \alpha_k \phi_k$ for our u_h in finite space S_h, we get…

$$\int_\Omega (a\sum_{j=1}^{n}\alpha_j \nabla\phi_j \cdot \nabla v_i + b\sum_{j=1}^{n}\alpha_j \nabla\phi_j \cdot v_i + c\sum_{j=1}^{n}\alpha_j \phi_j \cdot v_i)\,dx = \int_\Omega (f v_i)\,dx$$

For every $v_i \in S_h$

Or by basis spanning principals, for all $v_i = \{\phi_1, \phi_2, ..., \phi_n\}$

So

$$\int_{\Omega}(a\sum_{j=1}^{n}\alpha_j\nabla\phi_j\cdot\nabla v_i+b\sum_{j=1}^{n}\alpha_j\nabla\phi_j\cdot v_i+c\sum_{j=1}^{n}\alpha_j\phi_j\cdot v_i)dx=\int_{\Omega}(f v_i)dx$$

will give us n equations and n unknowns which we can put in the matrix form of $Au=f$.

Where A is our $n\times n$ stiffness matrix,

u is our vector of unknown displacements,

and f is the vector with contributions from our force function.

For example our first equation looks like

$$\left[\int_\Omega (a\nabla\phi_1\nabla v_1 + b\nabla\phi_1 v_1 + c\phi_1 v_1)\,dx \ldots \int_\Omega (a\nabla\phi_n\nabla v_1 + b\nabla\phi_n v_1 + c\phi_n v_1)\,dx\right]\begin{bmatrix}\alpha_1\\ \vdots \\ \alpha_n\end{bmatrix} = \left[\int_\Omega (f v_1)\,dx\right]$$

The second equation will look exactly like the first except v_2 will be in place of v_1.

So $Au = f$ is

$$
\underbrace{\begin{bmatrix} \int_\Omega (a\nabla\phi_1\nabla v_1 + b\nabla\phi_1 v_1 + c\phi_1 v_1)\,dx & \dots & \int_\Omega (a\nabla\phi_n\nabla v_1 + b\nabla\phi_n v_1 + c\phi_n v_1)\,dx \\ \vdots & & \vdots \\ \vdots & \cdots & \vdots \\ \vdots & & \vdots \\ \int_\Omega (a\nabla\phi_1\nabla v_n + b\nabla\phi_1 v_n + c\phi_1 v_n)\,dx & \dots & \int_\Omega (a\nabla\phi_n\nabla v_n + b\nabla\phi_n v_n + c\phi_n v_n)\,dx \end{bmatrix}}_{\boldsymbol{A}}
\underbrace{\begin{bmatrix} \alpha_1 \\ \vdots \\ \vdots \\ \vdots \\ \alpha_n \end{bmatrix}}_{\boldsymbol{u}}
=
\underbrace{\begin{bmatrix} \int_\Omega (f v_1)\,dx \\ \vdots \\ \vdots \\ \vdots \\ \int_\Omega (f v_n)\,dx \end{bmatrix}}_{\boldsymbol{f}}
$$

And since $v_i = \phi_i$,

We have our matrix equation below

$$\underbrace{\begin{bmatrix} \int_\Omega (a\nabla\phi_1\nabla\phi_1 + b\nabla\phi_1\phi_1 + c\phi_1\phi_1)\,dx & \dots & \int_\Omega (a\nabla\phi_n\nabla\phi_1 + b\nabla\phi_n\phi_1 + c\phi_n\phi_1)\,dx \\ \vdots & & \vdots \\ \vdots & \ddots & \vdots \\ \vdots & & \vdots \\ \int_\Omega (a\nabla\phi_1\nabla\phi_n + b\nabla\phi_1\phi_n + c\phi_1\phi_n)\,dx & \dots & \int_\Omega (a\nabla\phi_n\nabla\phi_n + b\nabla\phi_n\phi_n + c\phi_n\phi_n)\,dx \end{bmatrix}}_{\boldsymbol{A}} \underbrace{\begin{bmatrix} \alpha_1 \\ \vdots \\ \vdots \\ \vdots \\ \alpha_n \end{bmatrix}}_{\boldsymbol{u}} = \underbrace{\begin{bmatrix} \int_\Omega (f\phi_1)\,dx \\ \vdots \\ \vdots \\ \vdots \\ \int_\Omega (f\phi_n)\,dx \end{bmatrix}}_{\boldsymbol{f}}$$

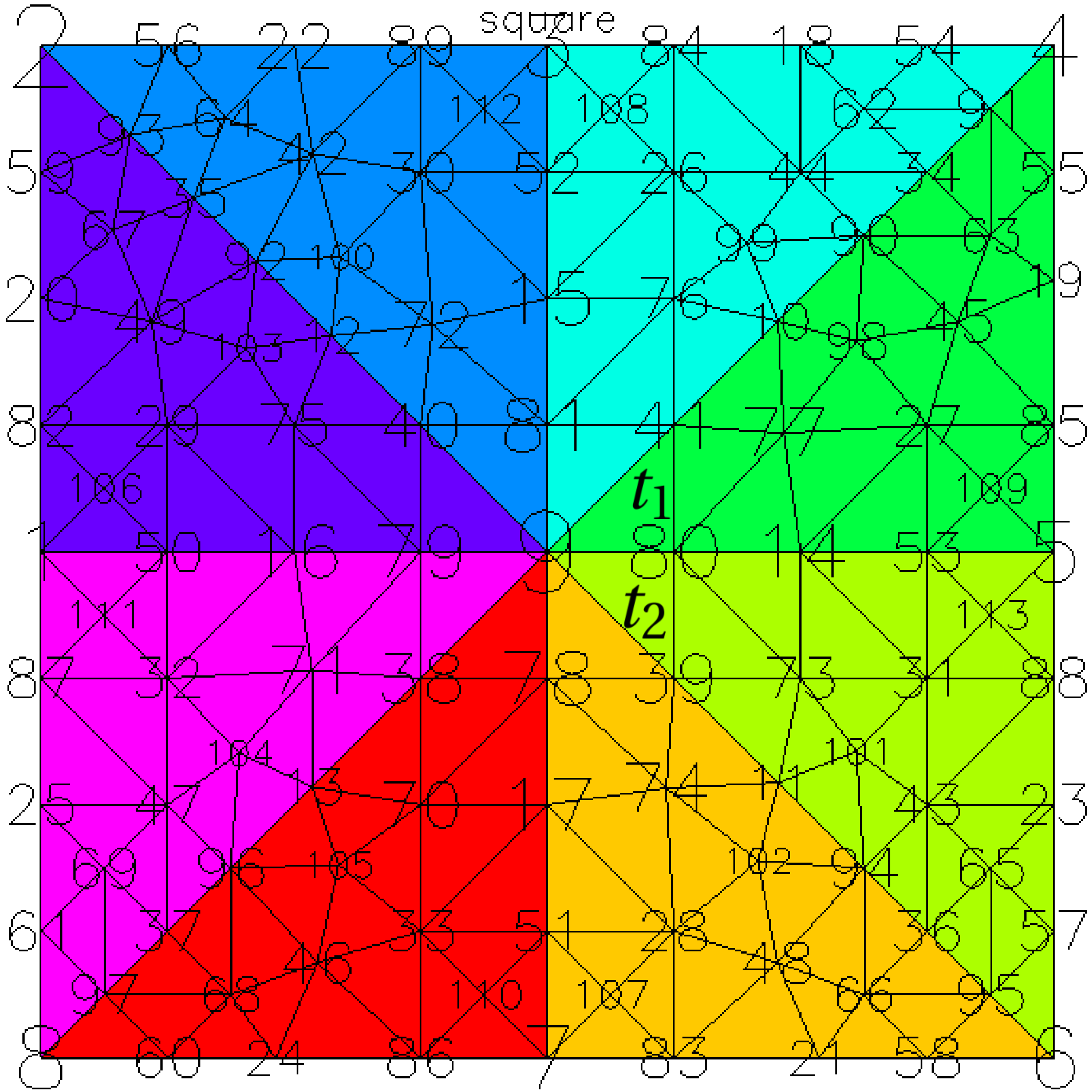
Looking Closer at our Stiffness Matrix A:

$$A_{i,j} = \int_\Omega (a\nabla\phi_j \nabla\phi_i + b\nabla\phi_j\phi_i + c\phi_j\phi_i)\,dx$$

$$= \sum_{t\in\Omega} \int_t (a\nabla\phi_j \nabla\phi_i + b\nabla\phi_j\phi_i + c\phi_j\phi_i)\,dx$$ (sum over all the triangles)

$$= \int_{t_1} (\quad)\,dx + \int_{t_2} (\quad)\,dx + \cdots + \int_{t_m} (\quad)\,dx$$

For Example

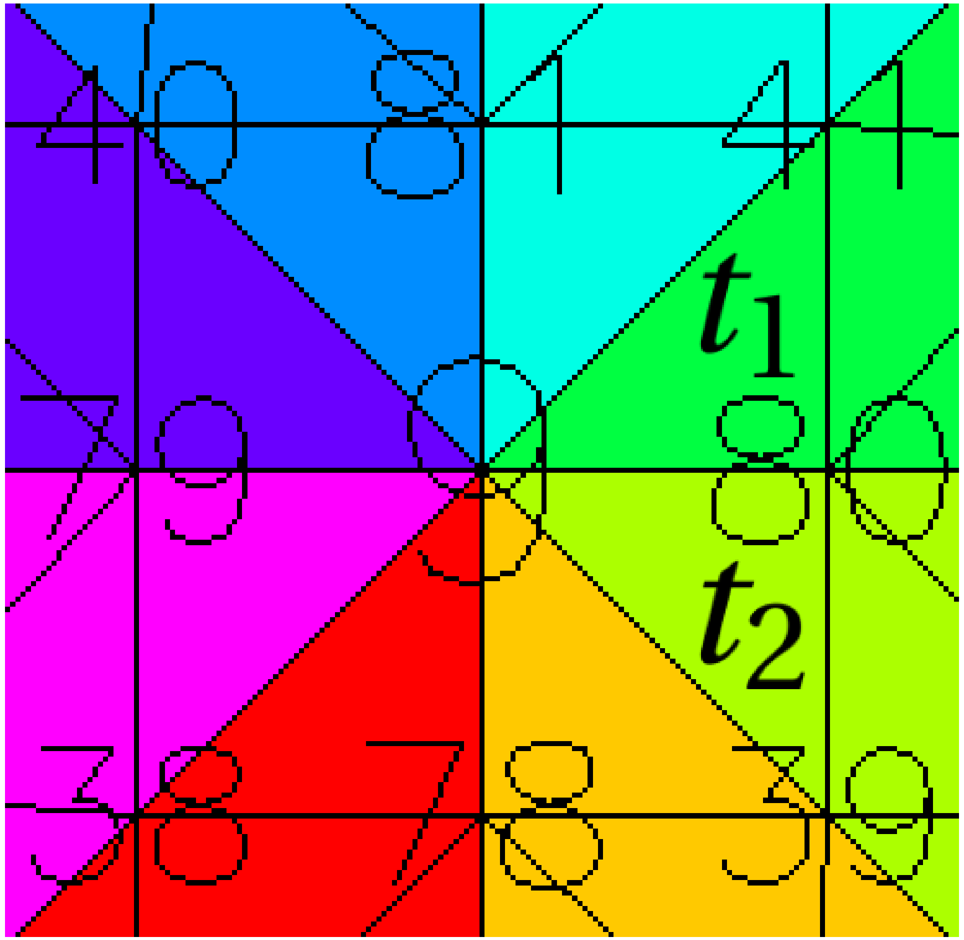


When we look at $\int_{\Omega} (\phi_9 \phi_{80})\, dx$

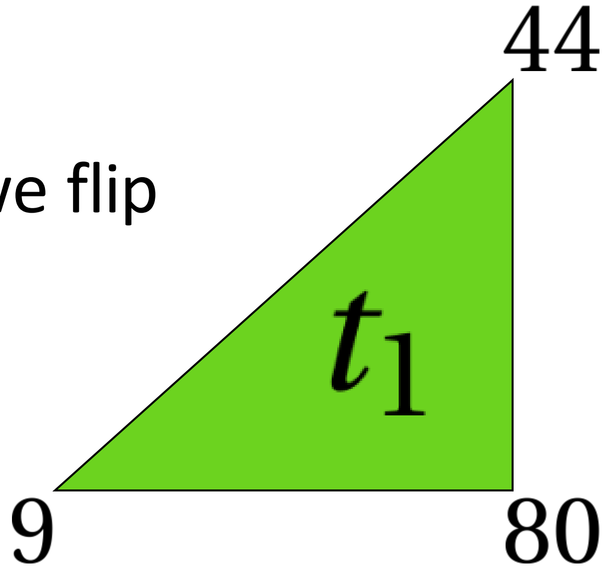
we can take into account that

$$\int_{\Omega} (\phi_9 \phi_{80})\, dx = \int_{t_1} (\phi_9 \phi_{80})\, dx + \int_{t_2} (\phi_9 \phi_{80})\, dx$$

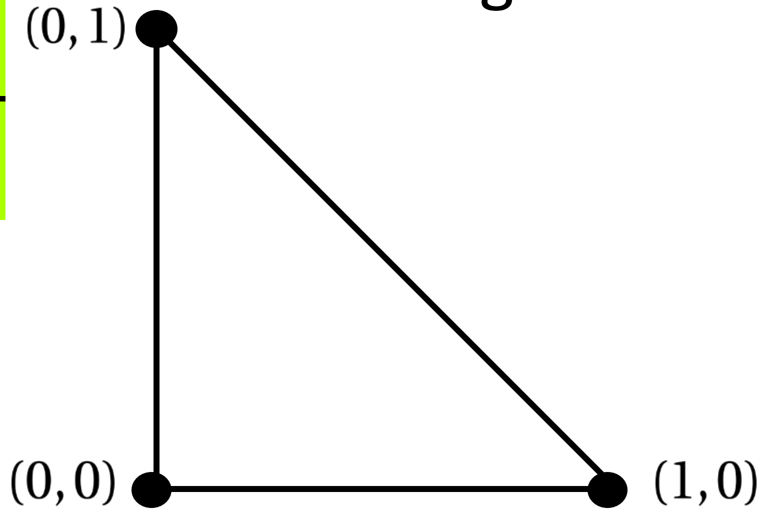
Looking just at $\int_{t_1} (\phi_9 \phi_{80})\, dx$



It is easier if we flip



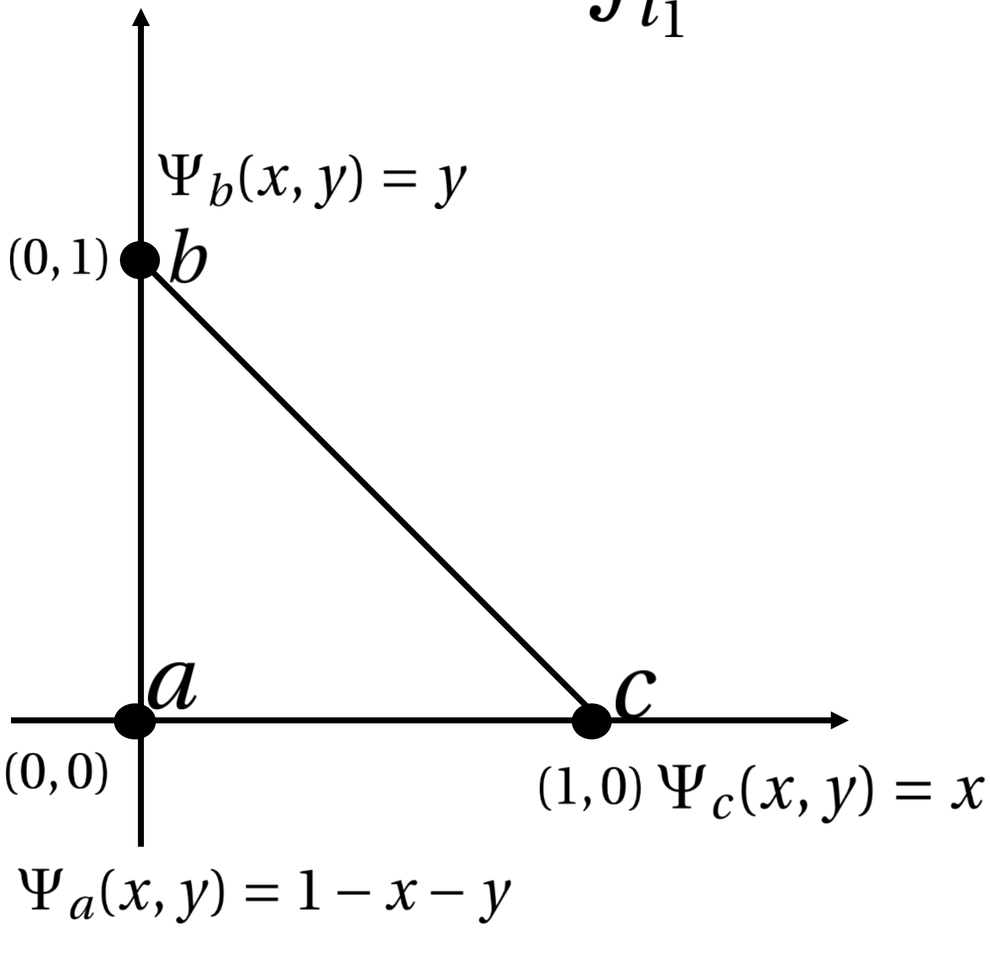
to the triangle at the origin



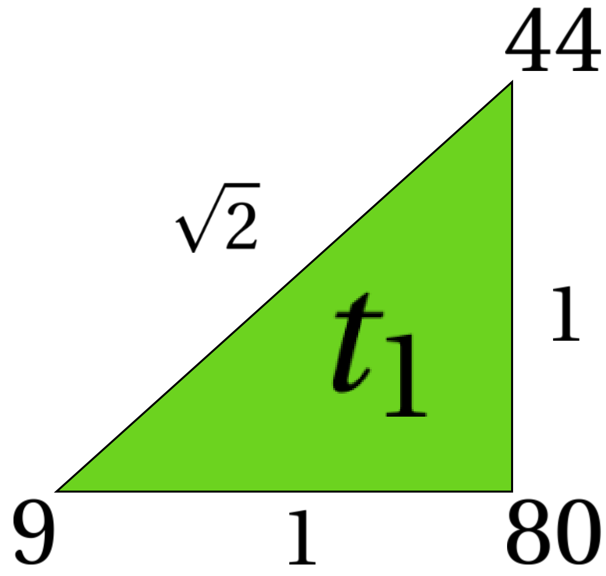
By this method,

$$\int_{t_1} (\phi_9 \phi_{80})\, dA = \int\int (\psi_c \psi_a)\, dA$$

$$= \int_0^1 \int_0^{1-x} x(1-x-y)\, dy\, dx$$

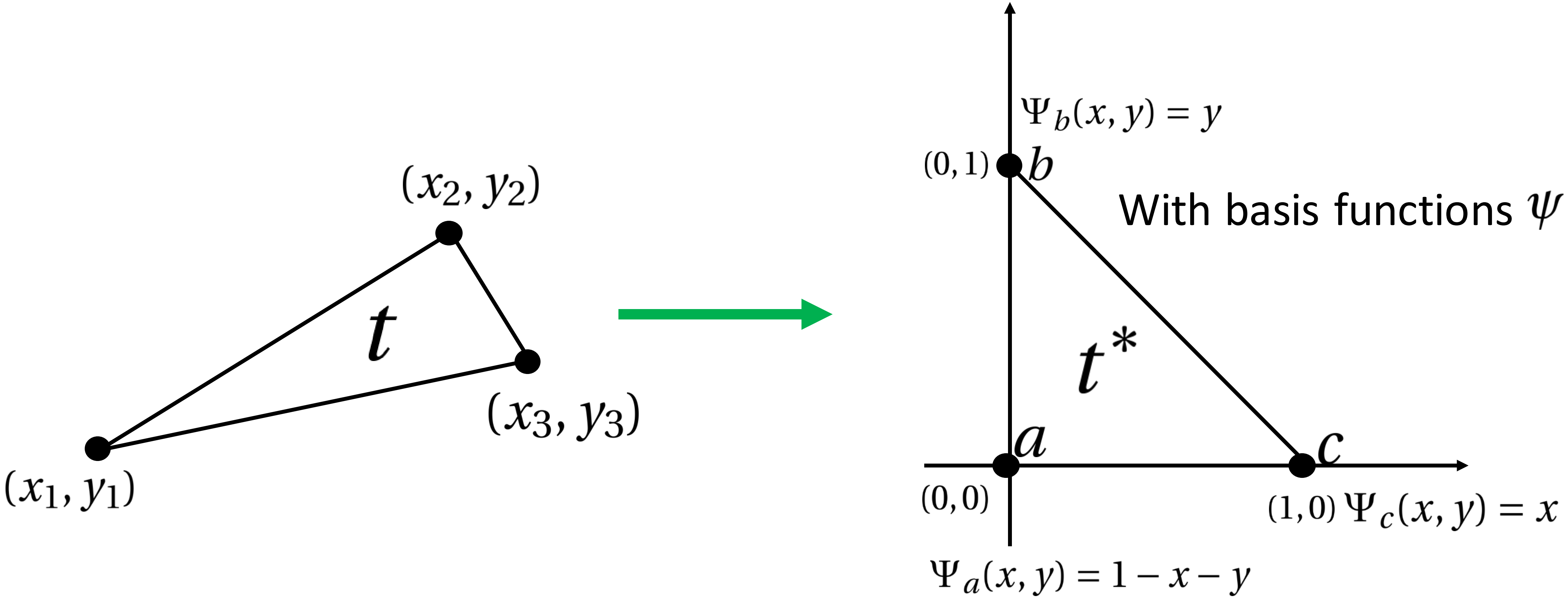


However this assumes that the lengths are 1 on both the sides and $\sqrt{2}$ on the hypotenuse, and therefore that a translation is all that is needed to represent the original triangle as the unit origin triangle.



Because our aim is to accommodate any triangular mesh, we take the time to account for triangles that will need to be transformed.

We will make the transformation of



With basis functions ϕ

Therefore

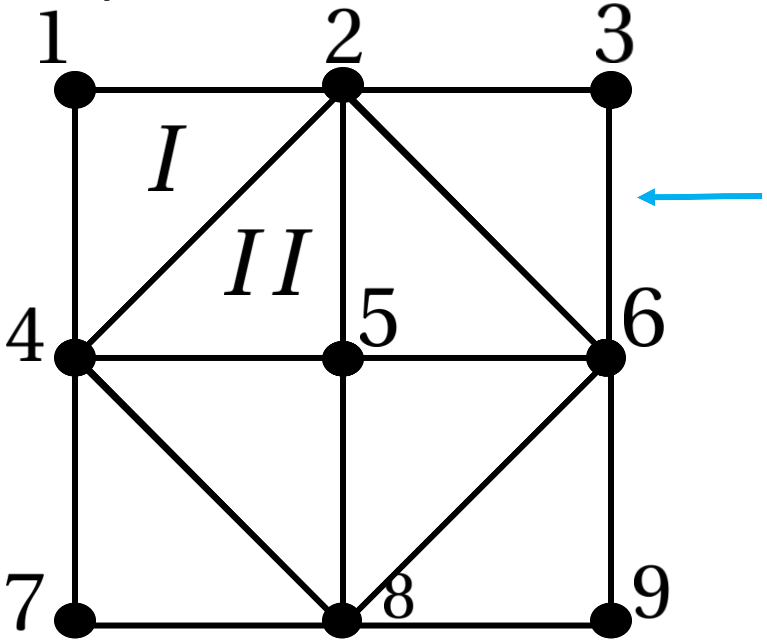
$$T = \begin{bmatrix} x_3 - x_1 & x_2 - x_1 \\ y_3 - y_1 & y_2 - y_1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

will be our transformation matrix

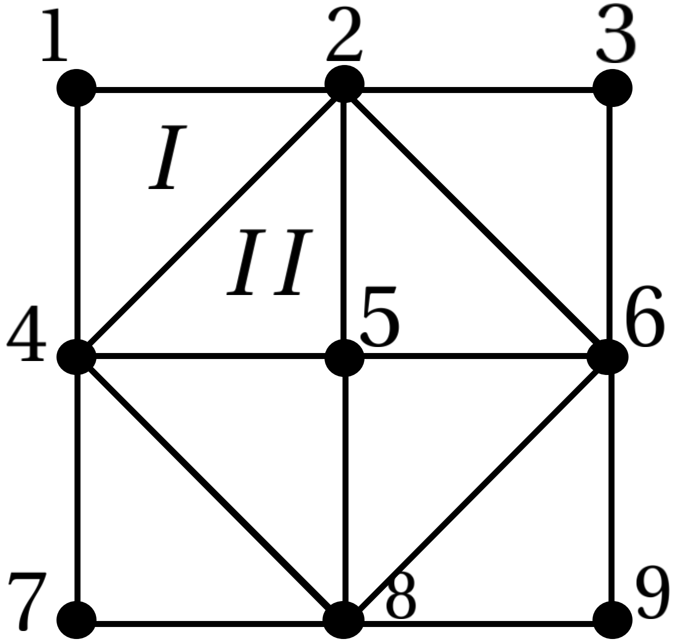
However before we continue, note what is required for one single triangle. In order to make our algorithm create the stiffness matrix in an efficient manner, we will construct it in such a way as to which each triangle will only have to be transformed once.

This will be done by initializing the stiffness matrix to all zeros, and adding in the contributions of each triangle as we iterate through them.

For example:

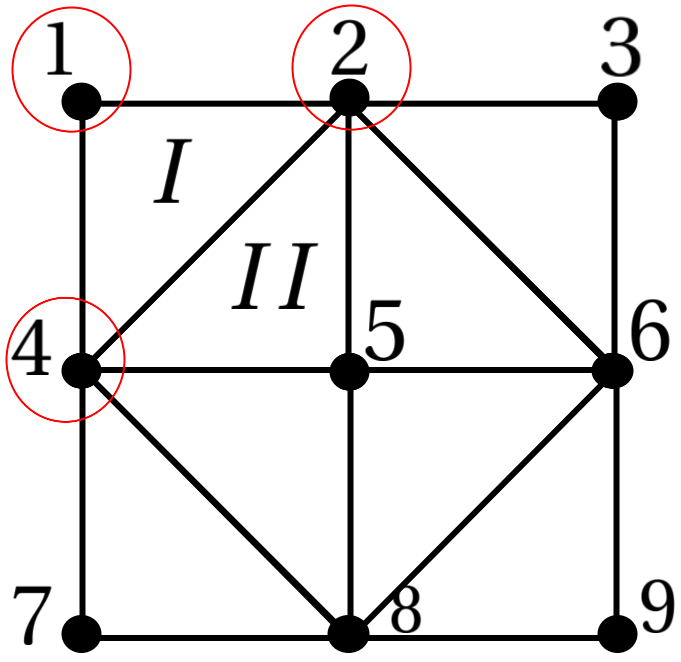


Will have 9 unknowns and therefore the stiffness matrix will start as an $n \times n$ matrix of all zeros.



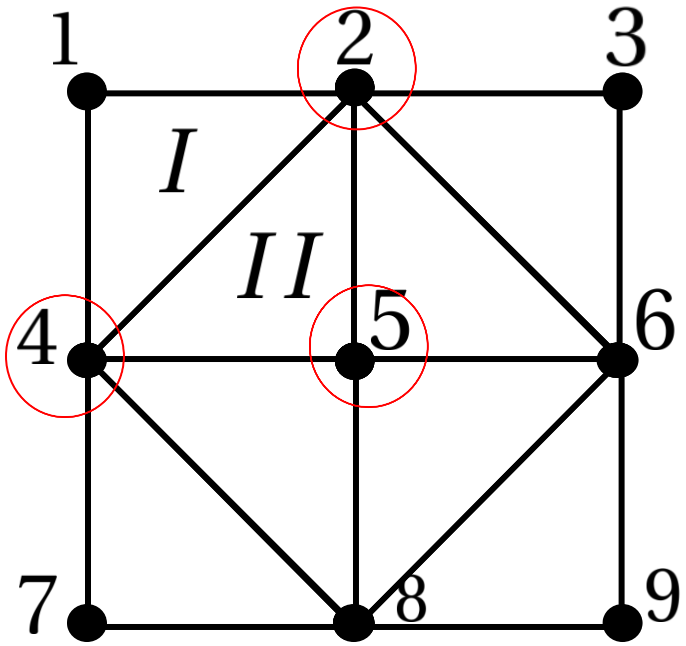
Initially:

$$A = \begin{bmatrix} 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \end{bmatrix}$$



After triangle I:

$$A = \begin{bmatrix} 2 & -1/2 & 0 & -1/2 & 0 & 0 & 0 & 0 & 0 \\ -1/2 & 2 & 0 & -1/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1/2 & -1/2 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$



After triangle II:

$$A = \begin{bmatrix} 2 & -1/2 & 0 & -1/2 & 0 & 0 & 0 & 0 & 0 \\ -1/2 & 4 & 0 & -1 & -1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1/4 & -1 & 0 & 4 & -1/2 & 0 & 0 & 0 & 0 \\ 0 & -1/2 & 0 & -1/2 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

So for each triangle we will cycle through all combinations of i, j vertices. And construct 3×3 sub-matricies δA which we will add into A.

$$\delta A_{i,j} = \int_t (a\nabla\phi_j \nabla\phi_i + b\nabla\phi_j \phi_i + c\phi_j \phi_i) dx$$

This is where our transformation matrix comes in handy.

$$T = \begin{bmatrix} x_3 - x_1 & x_2 - x_1 \\ y_3 - y_1 & y_2 - y_1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad J^{-1} = (T^T)^{-1}$$

Gives us

$$\delta A_{i,j} = \iint_{t^*} [aJ^{-1}\nabla\psi_j J^{-1}\nabla\psi_i + bJ^{-1}\nabla\psi_j \psi_i + c\psi_j \psi_i] |T| dx$$

Now

$$\delta A_{i,j} = \iint_{t^*} [aJ^{-1}\nabla\psi_j J^{-1}\nabla\psi_i + bJ^{-1}\nabla\psi_j\psi_i + c\psi_j\psi_i]|T|dx$$

gives us a function of x, y on which we can use quadrature.

Where

$$\Psi_1(x,y) = 1 - x - y$$

$$\Psi_2(x,y) = y$$

$$\Psi_3(x,y) = x$$

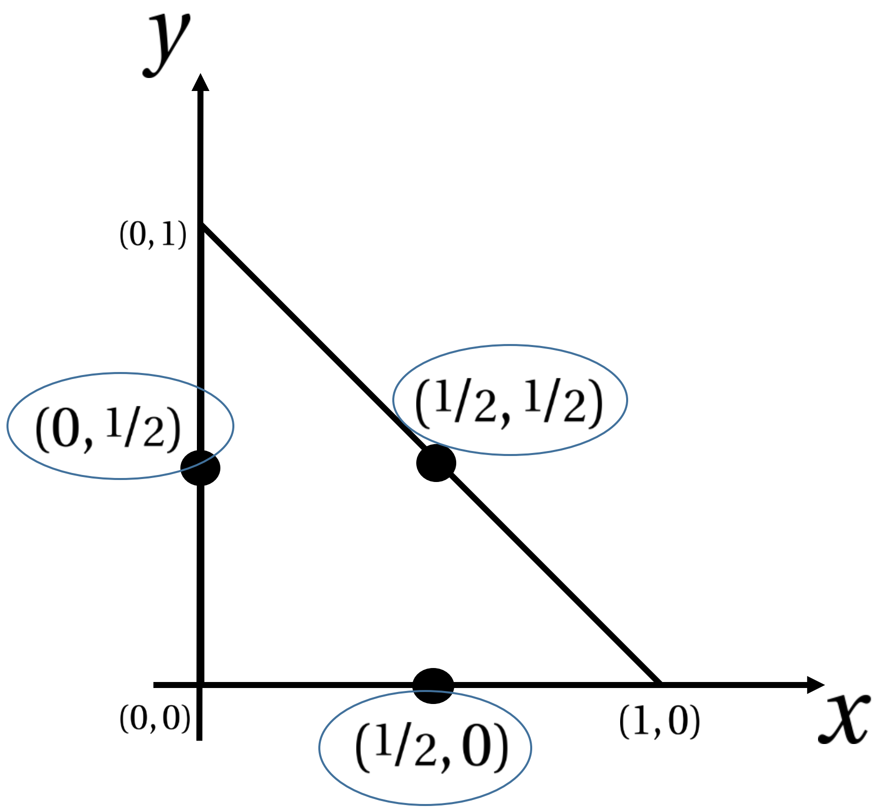
$$\nabla\Psi_1(x,y) = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

$$\nabla\Psi_2(x,y) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\nabla\Psi_3(x,y) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Quadrature

Order of a Quadrature Rule: degree of the lowest degree polynomial that the rule does not integrate exactly



-Therefore we will need a quadrature rule of order at least 3 for a 2D right triangle.

-So we will use the rule of order 3 below.

$$\int_0^1 \int_0^{1-y} f(x,y)\,dx\,dy = \int_0^1 \int_0^{1-x} f(x,y)\,dy\,dx = {}^1\!/_6 f({}^1\!/_2, {}^1\!/_2) + {}^1\!/_6 f({}^1\!/_2, 0) + {}^1\!/_6 f(0, {}^1\!/_2)$$

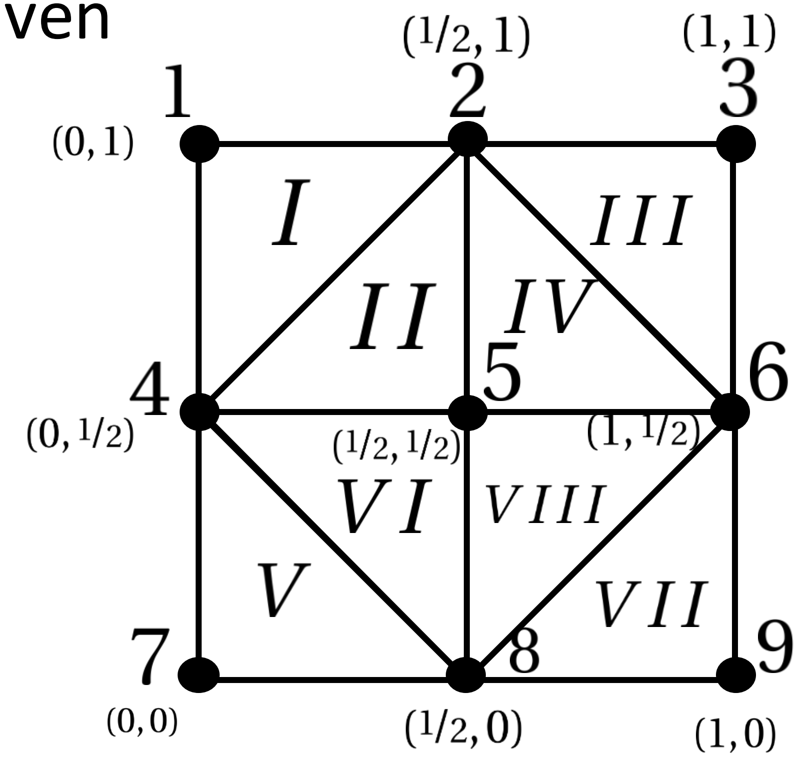
The v and t Our Program Will be Given

$$v$$

$$\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{bmatrix} 0 & 1 \\ 1/2 & 1 \\ 1 & 1 \\ 0 & 1/2 \\ 1/2 & 1/2 \\ 1 & 1/2 \\ 0 & 0 \\ 1/2 & 0 \\ 1 & 0 \end{bmatrix}$$

$$t$$

$$\begin{array}{c} I \\ II \\ III \\ IV \\ V \\ VI \\ VII \\ VIII \end{array} \begin{bmatrix} 1 & 2 & 4 \\ 2 & 4 & 5 \\ 2 & 3 & 6 \\ 2 & 5 & 6 \\ 4 & 7 & 8 \\ 4 & 5 & 8 \\ 6 & 8 & 9 \\ 5 & 6 & 8 \end{bmatrix}$$



Element Program

```
function [ delA,delf ] = element( a,b,c,v1,v2,v3,f )

  %Initialization of return values
  delA = zeros(3);
  delf = zeros(3,1);

  T = [v3(1)-v1(1) v2(1)-v1(1); v3(2)-v1(2) v2(2)-v1(2)];
  Jinv = inv(T');
  DetermT = abs(T(1,1)*T(2,2)-T(1,2)*T(2,1));

  %Basis functions for the unit triangle
  w1 = @(x,y) 1-x-y;
  w2 = @(x,y) y;
  w3 = @(x,y) x;
  w = {w1 w2 w3};

  %Gradient of basis functions
  dw1 = [-1;-1];
  dw2 = [0;1];
  dw3 = [1;0];
  dw = [dw1 dw2 dw3];

  %Matrix to implement quadrature rule of order 3 for a 2D right triangle
  z=[0.5 0 1/6;0 0.5 1/6; 0.5 0.5 1/6];
```

```matlab
%Construction of delA

  for j = 1:3

      for k = 1:3

          %Implementation of quadrature rule
          for m=1:size(z,1)


 delA(j,k)=delA(j,k)+ z(m,3)*((dot(a*(Jinv*dw(:,k)),Jinv*dw(:,j))+...

                    dot(b,Jinv*dw(:,k))*w{j}(z(m,1),z(m,2))+...

                    c*w{k}(z(m,1),z(m,2))*w{j}(z(m,1),z(m,2)))*DetermT);



          end
      end
  end
```

```matlab
%Construction of delf

  for j = 1:3

      %Implementation of quadrature rule
      for m=1:size(z,1)

        %Use of transformation matrix to get (x,y) in t of the current
        %vertex to be able to plug into f(x,y)
        r = T(1,:)*[z(m,1);z(m,2)]+v1(1);
        s = T(2,:)*[z(m,1);z(m,2)]+v1(2);

        delf(j) = delf(j)+ z(m,3)*(f(r,s)*w{j}(z(m,1),z(m,2)))*DetermT;

      end

  end

end
```

Element Assemble Program

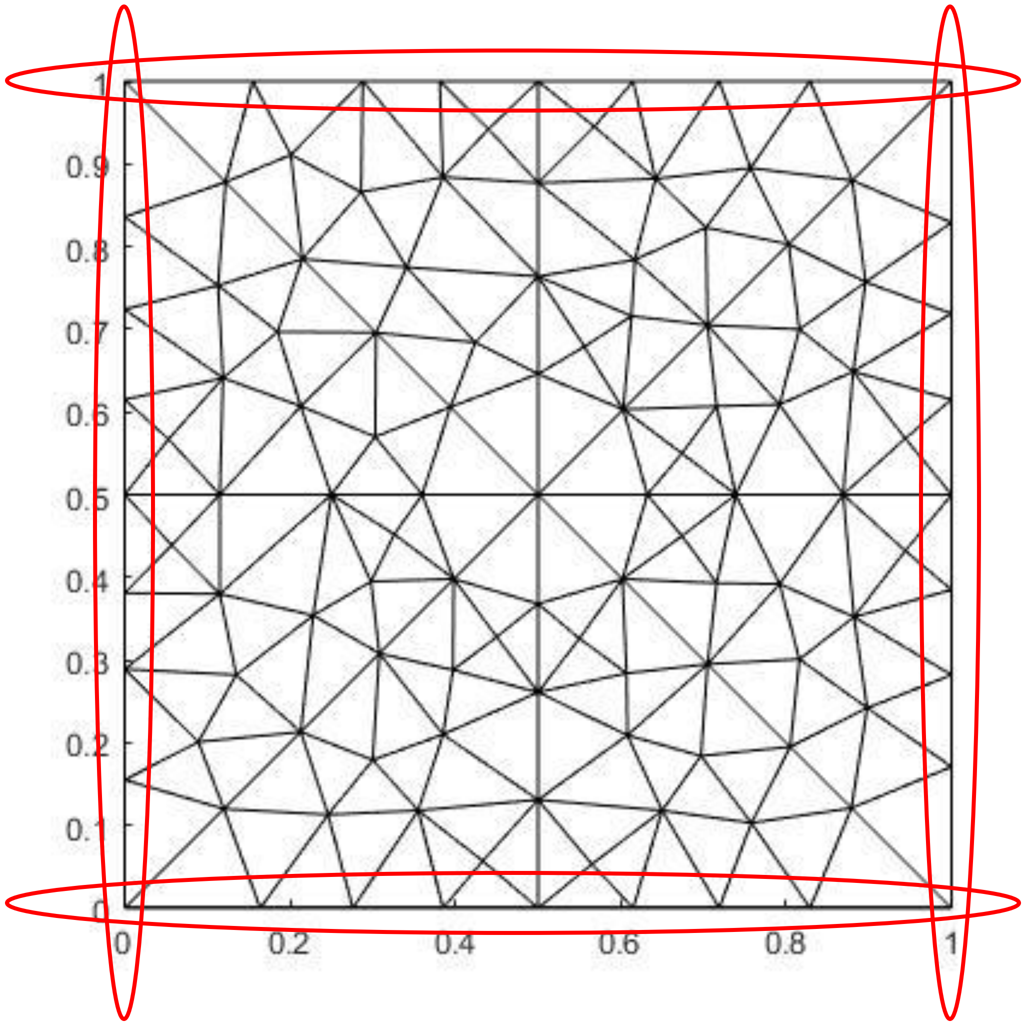
```
function [A F] = elementAssemble( a,b,c,v,t,f )

  A = zeros(size(v,1));
  F = zeros(size(v,1),1);

  for k = 1:size(t,1)

      [delA,delF] = element(a,b,c,v(t(k,1),:),v(t(k,2),:),v(t(k,3),:),f);
      for i = 1:3
          for j=1:3
              A(t(k,i),t(k,j)) = A(t(k,i),t(k,j)) + delA(i,j);
          end
          F(t(k,i))=F(t(k,i)) + delF(i);
      end
  end

end
```

However the A and f that we have constructed have not taken the Dirichlet boundary condition into account. Where the boundary is placed we need to have the displacement be zero.

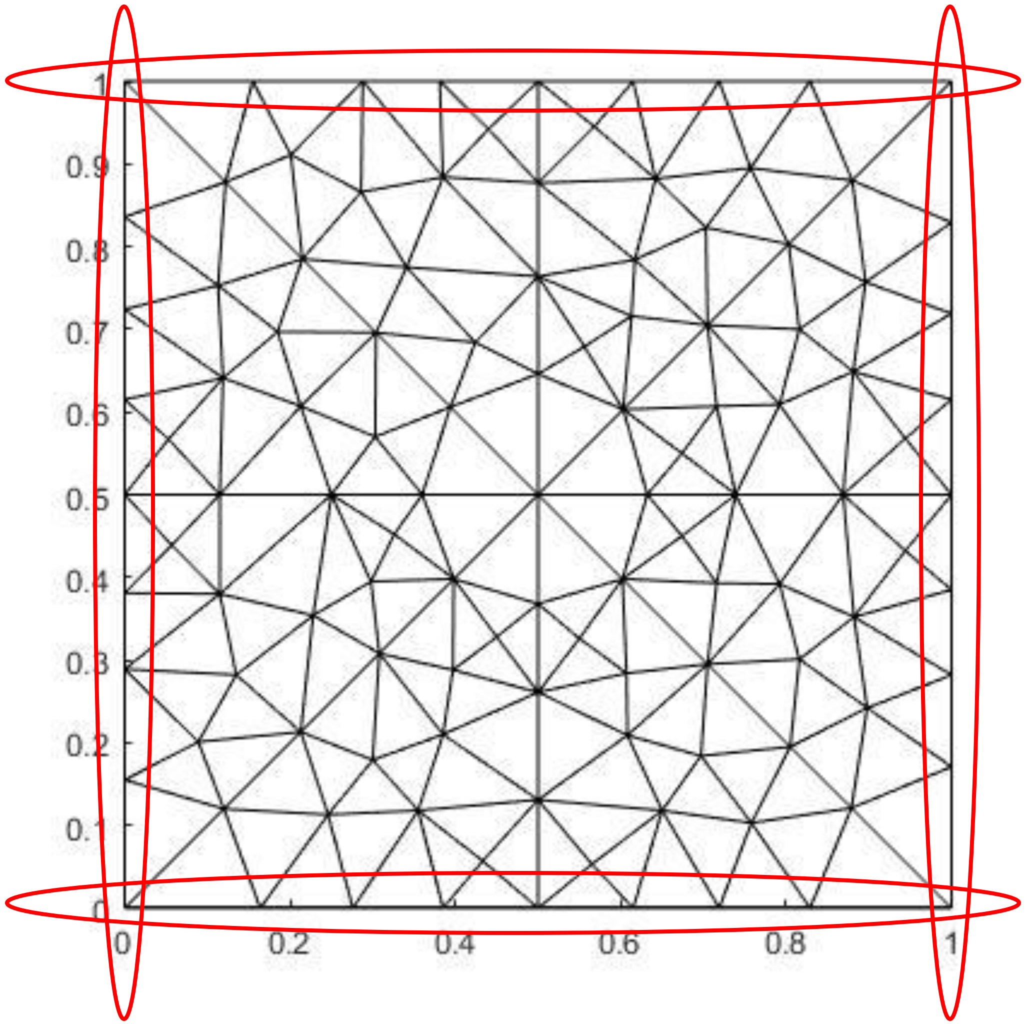
For this reason we need to design another program that will take as input the vertices of the triangles and A and f returned from elementAssemble. It will then remove the boundary conditions creating an A^* and f^*, that we will use to create $u^* = A^* \backslash f^*$. Which finally will be converted to our solution u by adding zeros back in the removed positions.

Boundary Zero Program

```
function [ u ] = getuWithBoundaryZero( A,F,v )

u=zeros(size(A,1),1);
m=1;
B=zeros(size(A,1));
C=zeros(size(A,1),1);

for j=1:size(A,1)
    if (v(j,1)==0) | (v(j,1)==1)
    elseif (v(j,2)==0) | (v(j,2)==1)
    else
        n=1;
        for i=1:size(A,1)
            if (v(i,1)==0) | (v(i,1)==1)
            elseif (v(i,2)==0) | (v(i,2)==1
            else
                B(m,n)=A(j,i);
                n=n+1;
            end
        end
        C(m)=F(j);
        m=m+1;
    end
end
```



```
Areduced=zeros(m-1);
Freduced=zeros(m-1,1);

for j=1:m-1
    for i=1:m-1
        Areduced(j,i)=B(j,i);
    end
    Freduced(j)=C(j);
end

ureduced=Areduced\Freduced;
m=1;
for j=1:size(A,1)
    if (v(j,1)==0) | (v(j,1)==1)
        u(j)=0;
    elseif (v(j,2)==0) | (v(j,2)==1)
        u(j)=0;
    else
        u(j)=ureduced(m);
        m=m+1;
    end
end

end
```



Plot3d Program

```
function [] = plot3d(t,v,u)
  hold on;
  n=size(t,1);
  for i=1:n
    plot3([v(t(i,1),1) v(t(i,2),1)], [v(t(i,1),2) v(t(i,2),2)], [u(t(i,1)) u(t(i,2))],'-k');
    plot3([v(t(i,2),1) v(t(i,3),1)], [v(t(i,2),2) v(t(i,3),2)], [u(t(i,2)) u(t(i,3))],'-k');
    plot3([v(t(i,3),1) v(t(i,1),1)], [v(t(i,3),2) v(t(i,1),2)], [u(t(i,3)) u(t(i,1))],'-k');
  end
  axis square;
end
```

Let's Use It!

Solve $-\triangle u = 1$ on $\Omega = [0,1] \times [0,1] \in \mathbf{R}^2$ with the edges fixed.

First, plugging into our PDE we can see that

$$-\nabla \cdot (a(x,y) \nabla u) + b(x,y) \cdot \nabla u + c(x,y) u - f(x,y) = 0 \ (in\ \Omega)$$

$$a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad b = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad c = 0 \qquad f = 1$$

Calling our Programs

First create a separate function for our force function $f = 1$

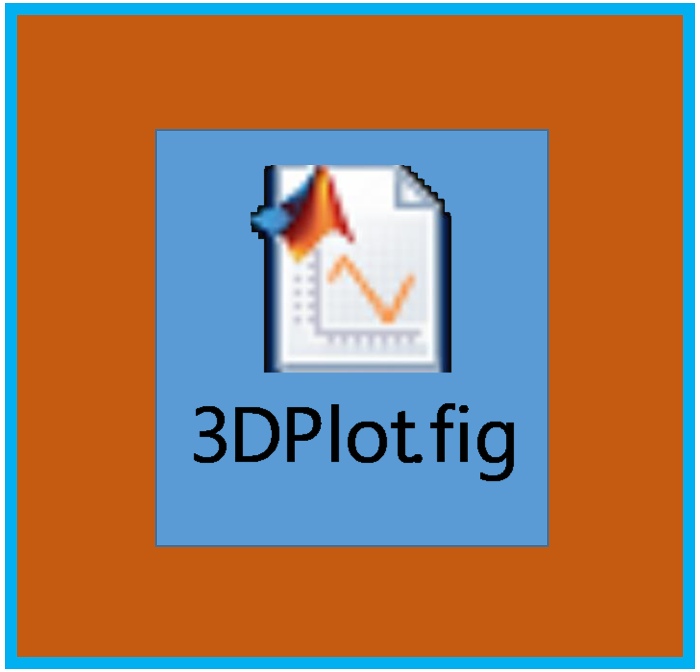
Now call `elementAssemble(a,b,c,v,t,f)` with

```
[A f] = elementAssemble([1 0;0 1],[0;0],0,v,t,@fbasic);
```

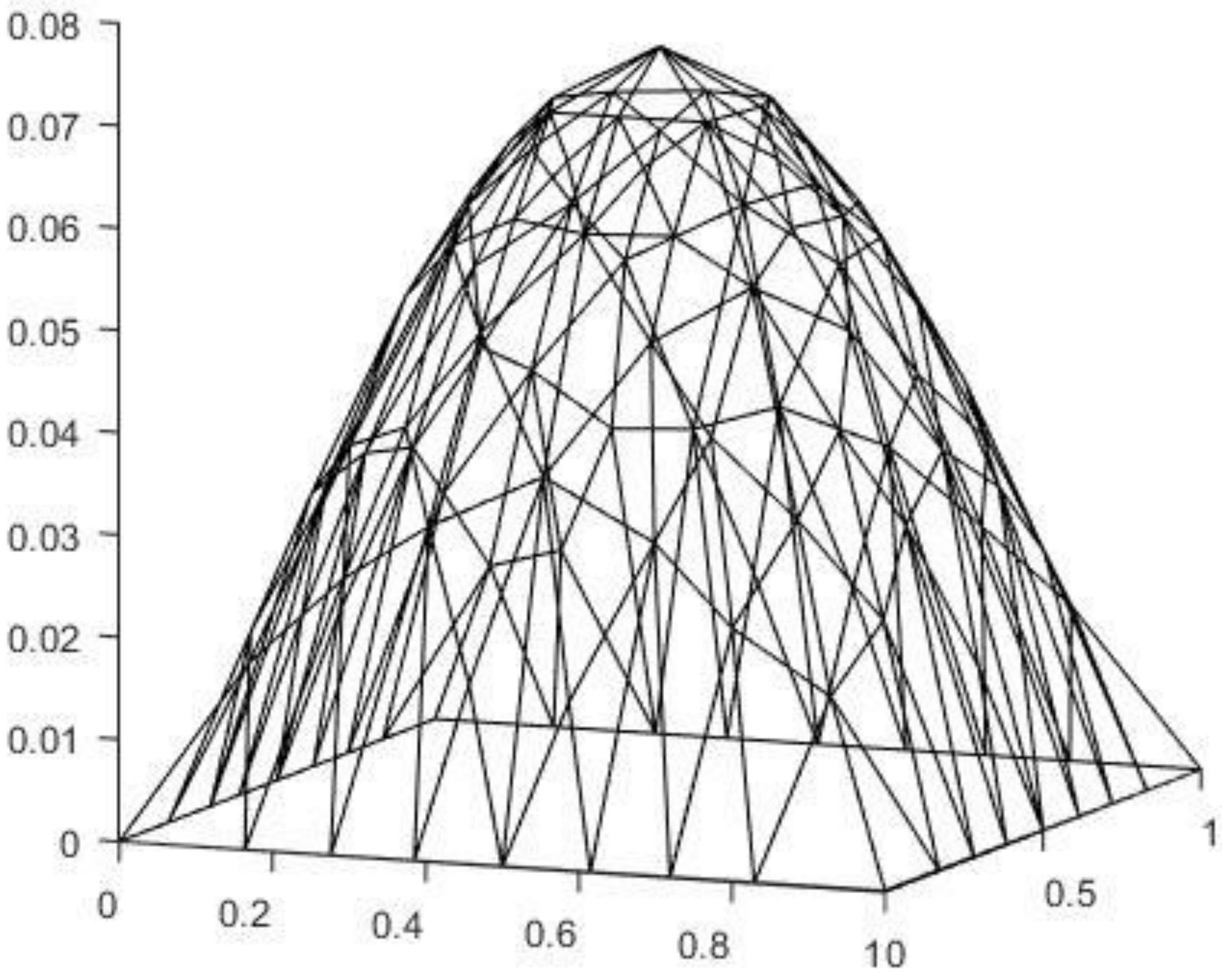
Now call `getuWithBoundaryZero(A,F,v)` with

```
u=getuWithBoundaryZero(A,F,v);
```

Finally call `plot3d(t,v,u)`, and look at the result!



Finally call `plot3d(t,v,u)`, and look at the result!



Elasticity in 2D

We can now use our previous programs to simulate elasticity in two dimensions.

Our new PDE:

$$-2\mu(\nabla \cdot \varepsilon(u)) - \lambda \nabla^2 u = f(x) \text{ in } \Omega$$

$$\sigma(u) \cdot n = g(x) \text{ on } \partial_N \Omega$$

$$u = 0 \text{ on } \partial_D \Omega$$

Where

Linear Stress:

$$\varepsilon(x)_{i,j} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$$

Linear Strain:

$$\sigma(x_R) = \lambda\, trace(E) I + 2\mu E$$

Constants:

$$\mu = \frac{E}{2(1+\nu)} \approx 8.2031 \qquad \lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \approx 10.4403$$

where $E = 21.0$ is Young's Modulus and $\nu = 0.28$ is Poisson Ratio

Process

We follow the same process as before:

- -Strong form
- -Weak form
- -Galerkin Approximation
- -Construction of $Au = f$

However this time we will have $2n$ equations and $2n$ unknowns as each node can move either up/down or left/right. This also causes us to require two basis functions. For the sake of time we will skip this portion as it is a more complicated application of the techniques demonstrated previously.

New Element Function

```
function [ delA,delF ] = elementNew( a,b,c,v1,v2,v3,f,n )

  %Construction of q
  for j = 1:3

      %Implementation of quadrature rule
      for m=1:size(z,1)

        r = T(1,:)*[z(m,1);z(m,2)]+v1(1);
        s = T(2,:)*[z(m,1);z(m,2)]+v1(2);

        fvector = f(r,s);

        delF(j) = delF(j)+ z(m,3)*(fvector(n))*w{j}(z(m,1),z(m,2)))*DetermT;

      end

  end

end
```

This is the only difference between our new program and our old Element function.

New Element Assemble Function

```
function [ A F] = elementAssembleNew( a,b,c,v,t,f,n )

A = zeros(size(v,1));
F = zeros(size(v,1),1);

for k = 1:size(t,1)

    [delA,delF] = elementNew(a,b,c,v(t(k,1),:),v(t(k,2),:),v(t(k,3),:),f,n);
    for i = 1:3
        for j=1:3
            A(t(k,i),t(k,j)) = A(t(k,i),t(k,j)) + delA(i,j);
        end
        F(t(k,i))=F(t(k,i)) + delF(i);
    end
end
end
```

Make A and F Function

```
function [A,F] = makeAandF(v,t,f)

    % lame constants
    lambda = (21.0)/(2*(1+0.28));
    mu = (21.0*0.28)/((1+0.28)*(1-2*(0.28)));

    % initialize A and F to the proper size (n is size of v)
    n = size(v,1);
    A = zeros(2*n);
    F = zeros(2*n,1);

    % construction of A11, A12, A21, and A22

    %A11
        a=[2*(mu)+lambda, 0; 0, mu];
        [A11 F1] = elementAssembleNew(a,[0;0],0,v,t,f,1);

    %A12
        a=[0, lambda; mu, 0];
        [A12 F1] = elementAssembleNew(a,[0;0],0,v,t,f,1);

    %A21
        a=[0, mu; lambda, 0];
        [A21 F2] = elementAssembleNew(a,[0;0],0,v,t,f,2);

    %A22
        a=[mu, 0; 0, 2*(mu)+lambda];
        [A22 F2] = elementAssembleNew(a,[0;0],0,v,t,f,2);
```

Make A and F Function

```
% constructing A with A11
      for i = 1:n
          for j = 1:n
              A(i,j) = A11(i,j);
          end
      end

   % constructing A with A12
      for i = 1:n
          for j = n+1:2*n
              A(i,j) = A12(i,j-n);
          end
      end

   % constructing A with A21
      for i = n+1:2*n
          for j = 1:n
              A(i,j) = A21(i-n,j);
          end
      end

   % constructing A with A22
      for i = n+1:2*n
          for j = n+1:2*n
              A(i,j) = A22(i-n,j-n);
          end
      end

   % constructs F
      for i = 1:n
          F(i) = F1(i);
      end

      for i = n+1:2*n
          F(i) = F2(i-n);
      end

end
```

Improved Boundary Function

```
function [u] = getuWithBoundary(A,F,v,x1,andorX,x2,andorXY,y1,andorY,y2)

    s = size(A,1);
    u=zeros(s,1);
    m=1;
    B=zeros(s);
    C=zeros(s,1);

% if andorXY is an and
if (strcmp(andorXY,'and'))

    % x and, y and
    if ((strcmp(andorX,'and')) & (strcmp(andorY,'and')))
```

Improved Boundary Function

```
for j=1:(s/2)

    if (((v(j,1)<=x2) & (v(j,1)>=x1)) & ((v(j,2)<=y2) & (v(j,2)>=y1)))
    else
        n=1;
        for i=1:s

          if (i<=(s/2))

            if (((v(i,1)<=x2) & (v(i,1)>=x1)) & ((v(i,2)<=y2) & (v(i,2)>=y1)))
            else
                B(m,n)=A(j,i);
                n=n+1;
            end

          else

            if (((v(i-(s/2),1)<=x2) & (v(i-(s/2),1)>=x1)) & ((v(i-(s/2),2)<=y2) & (v(i-(s/2),2)>=y1)))
            else
                B(m,n)=A(j,i);
                n=n+1;
            end

          end

        end

        C(m)=F(j);
        m=m+1;
    end
end
```

Improved Boundary Function

```
for j=((s/2)+1):s
    if (((v(j-(s/2),1)<=x2) & (v(j-(s/2),1)>=x1)) & ((v(j-(s/2),2)<=y2) & (v(j-(s/2),2)>=y1)))
    else
        n=1;
        for i=1:s
        if (i<=(s/2))

            if (((v(i,1)<=x2) & (v(i,1)>=x1)) & ((v(i,2)<=y2) & (v(i,2)>=y1)))
            else
                B(m,n)=A(j,i);
                n=n+1;
            end

        else

            if (((v(i-(s/2),1)<=x2) & (v(i-(s/2),1)>=x1)) & ((v(i-(s/2),2)<=y2) & (v(i-(s/2),2)>=y1)))
            else
                B(m,n)=A(j,i);
                n=n+1;
            end

          end

        end
        C(m)=F(j);
        m=m+1;
    end
end
```

Improved Boundary Function

```
Areduced=zeros(m-1);
Freduced=zeros(m-1,1);

for j=1:m-1
    for i=1:m-1
        Areduced(j,i)=B(j,i);
    end
    Freduced(j)=C(j);
end

ureduced=Areduced\Freduced;
m=1;

for j=1:(s/2)
    if (((v(j,1)<=x2) & (v(j,1)>=x1)) & ((v(j,2)<=y2) & (v(j,2)>=y1)))
        u(j)=0;
    else
        u(j)=ureduced(m);
        m=m+1;
    end
end

for j=((s/2)+1):s
    if (((v(j-(s/2),1)<=x2) & (v(j-(s/2),1)>=x1)) & ((v(j-(s/2),2)<=y2) & (v(j-(s/2),2)>=y1)))
        u(j)=0;
    else
        u(j)=ureduced(m);
        m=m+1;
    end
end
```

Improved Boundary Function

```
function [u] = getuWithBoundary(A,F,v,x1,andorX,x2,andorXY,y1,andorY,y2)

% if andorXY is an and
if (strcmp(andorXY,'and'))

    % x and, y and
    if ((strcmp(andorX,'and')) & (strcmp(andorY,'and')))

        %----------------------------------------------------> if (((v(j,1)<=x2) & (v(j,1)>=x1)) & ((v(j,2)<=y2) & (v(j,2)>=y1)))

    % x and, y or
    elseif ((strcmp(andorX,'and')) & (strcmp(andorY,'or')))

        %----------------------------------------------------> if (((v(j,1)<=x2) & (v(j,1)>=x1)) & ((v(j,2)<=y2) | (v(j,2)>=y1)))

    % x or, y and
    elseif ((strcmp(andorX,'or')) & (strcmp(andorY,'and')))

        %----------------------------------------------------> if (((v(j,1)<=x2) | (v(j,1)>=x1)) & ((v(j,2)<=y2) & (v(j,2)>=y1)))

    % x or, y or
    elseif ((strcmp(andorX,'or')) & (strcmp(andorY,'or')))

        %----------------------------------------------------> if (((v(j,1)<=x2) | (v(j,1)>=x1)) & ((v(j,2)<=y2) | (v(j,2)>=y1)))

    % incorrect input
    else
        disp('Please give the string "and" or the string "or" for the parameters andorX and andorY!');

    end
```

Improved Boundary Function

```
% if andorXY is an or
elseif (strcmp(andorXY,'or'))


    % x and, y and
    if ((strcmp(andorX,'and')) & (strcmp(andorY,'and')))

        %--------------------------------------------------> if (((v(j,1)<=x2) & (v(j,1)>=x1)) | ((v(j,2)<=y2) & (v(j,2)>=y1)))

    % x and, y or
    elseif ((strcmp(andorX,'and')) & (strcmp(andorY,'or')))

        %--------------------------------------------------> if (((v(j,1)<=x2) & (v(j,1)>=x1)) | ((v(j,2)<=y2) | (v(j,2)>=y1)))

    % x or, y and
    elseif ((strcmp(andorX,'or')) & (strcmp(andorY,'and')))

        %--------------------------------------------------> if (((v(j,1)<=x2) | (v(j,1)>=x1)) | ((v(j,2)<=y2) & (v(j,2)>=y1)))

    % x or, y or
    elseif ((strcmp(andorX,'or')) & (strcmp(andorY,'or')))

        %--------------------------------------------------> if (((v(j,1)<=x2) | (v(j,1)>=x1)) | ((v(j,2)<=y2) | (v(j,2)>=y1)))

    % incorrect input
    else
        disp('Please give the string "and" or the string "or" for the parameter andorXY!');

    end

end
end
```

Plot2D Function

```
function [] = plot2dwithu(t,v,u)
  hold on;
  n=size(t,1);
  b=size(v,1);
  for i=1:n
    plot([v(t(i,1),1)+u(t(i,1)) v(t(i,2),1)+u(t(i,2))], [v(t(i,1),2)+u(t(i,1)+b) v(t(i,2),2)+u(t(i,2)+b)], '-k');
    plot([v(t(i,2),1)+u(t(i,2)) v(t(i,3),1)+u(t(i,3))], [v(t(i,2),2)+u(t(i,2)+b) v(t(i,3),2)+u(t(i,3)+b)], '-k');
    plot([v(t(i,3),1)+u(t(i,3)) v(t(i,1),1)+u(t(i,1))], [v(t(i,3),2)+u(t(i,3)+b) v(t(i,1),2)+u(t(i,1)+b)], '-k');
  end
  axis square;
end
```

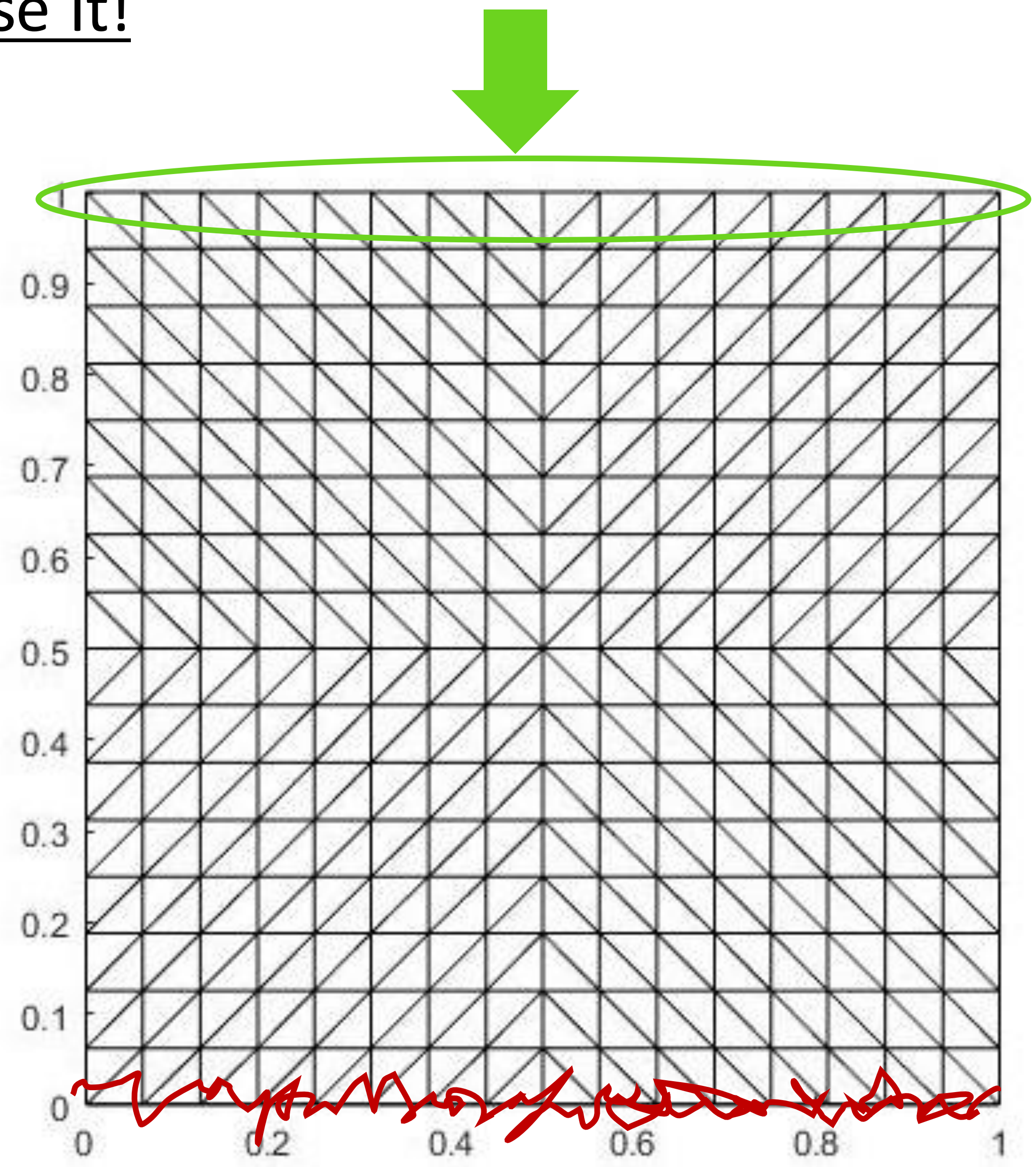
Animation Function

```
function [] = animate(t,v,u,increment,finish)

for i=0:increment:finish
    i/increment
    clf;
    plot2dwithu(t,v,i*u);
    axis([-1 2 -1 2]);
    drawnow;
    saveas(gcf,strcat(strcat('img',num2str(i/increment)),'.jpg'));
end

end
```

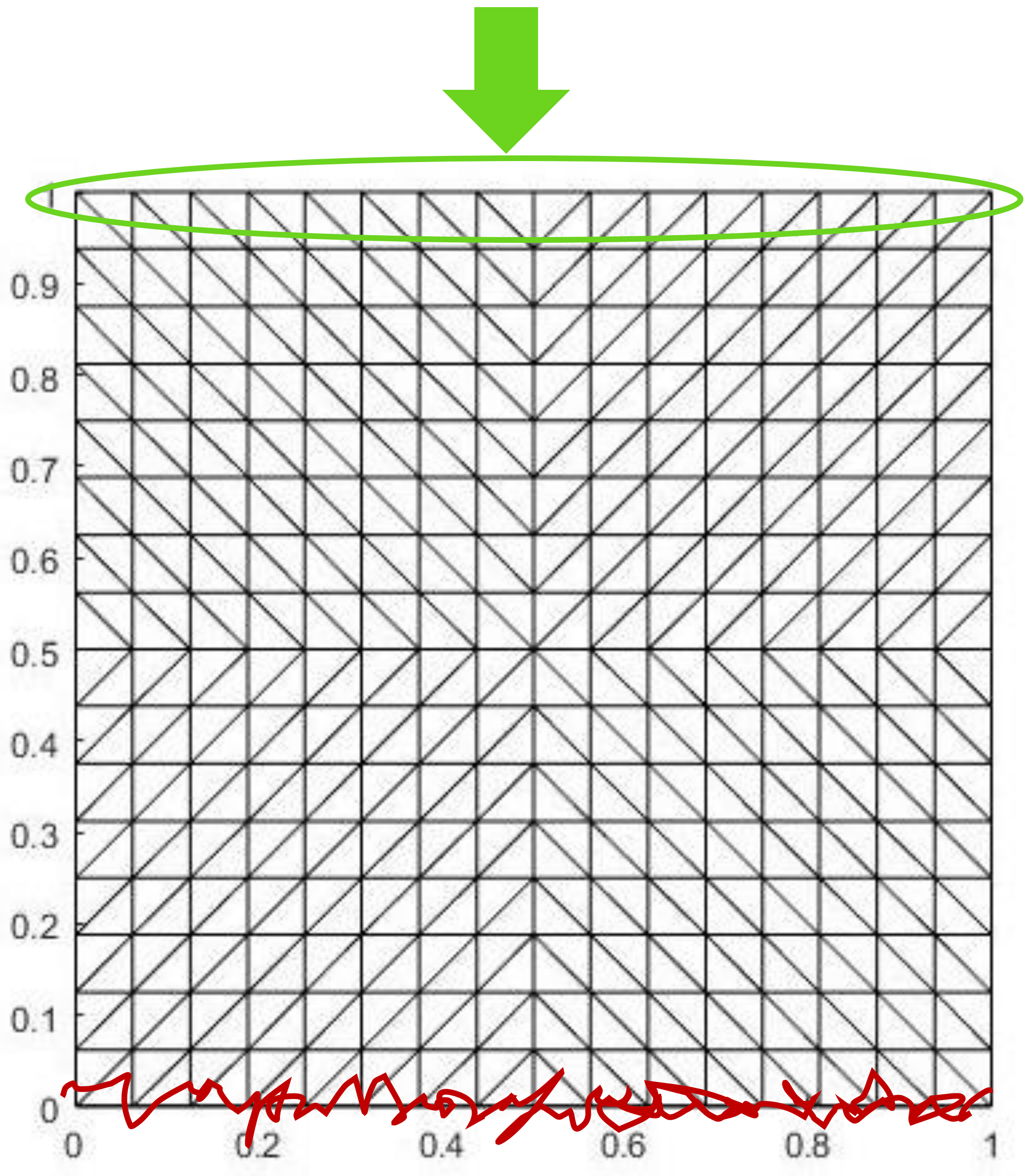
Let’s Use It!

- force function

- Dirichlet boundary condition

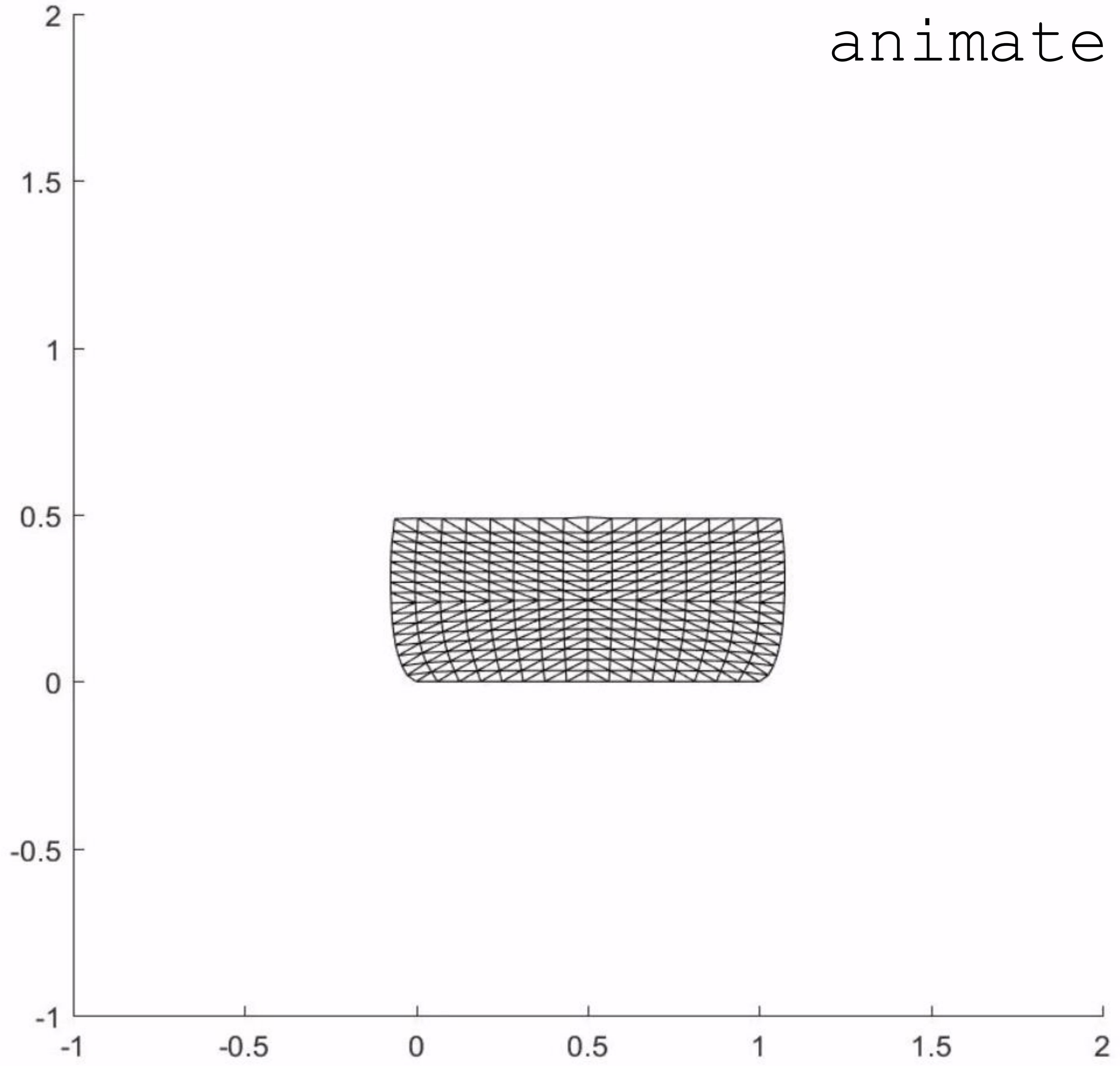


```
function [y] = f1 (x1,x2)

y = zeros(2,1);

    if (x2 >= .95)
         y(1) = 0;
         y(2) = -1;
    end

end
```

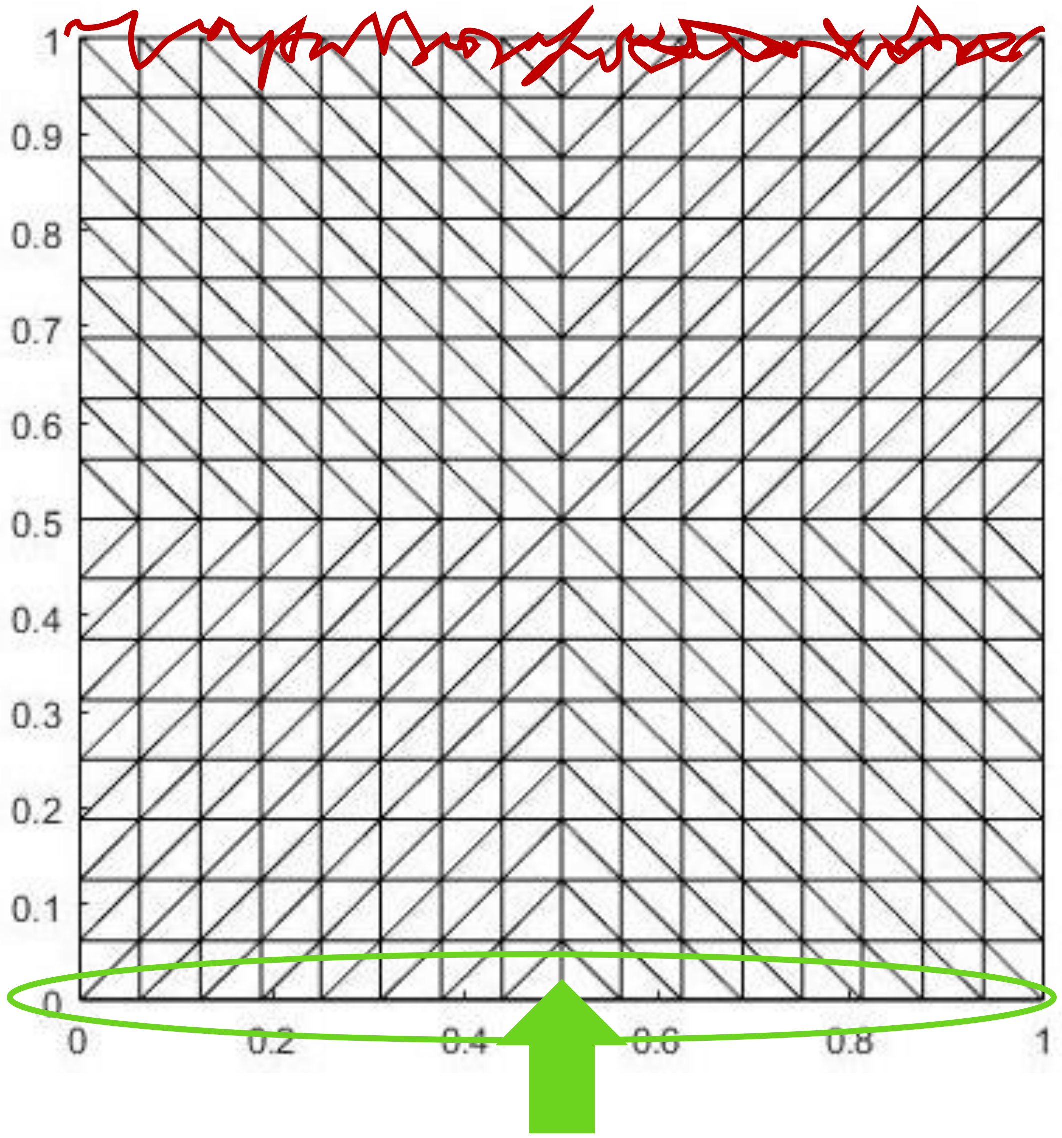



```
u=getuWithBoundary(A,F,v2,0,'and',1,'and',5,'or',0);
```

```
animate(t2,v2,u,25,300)
```

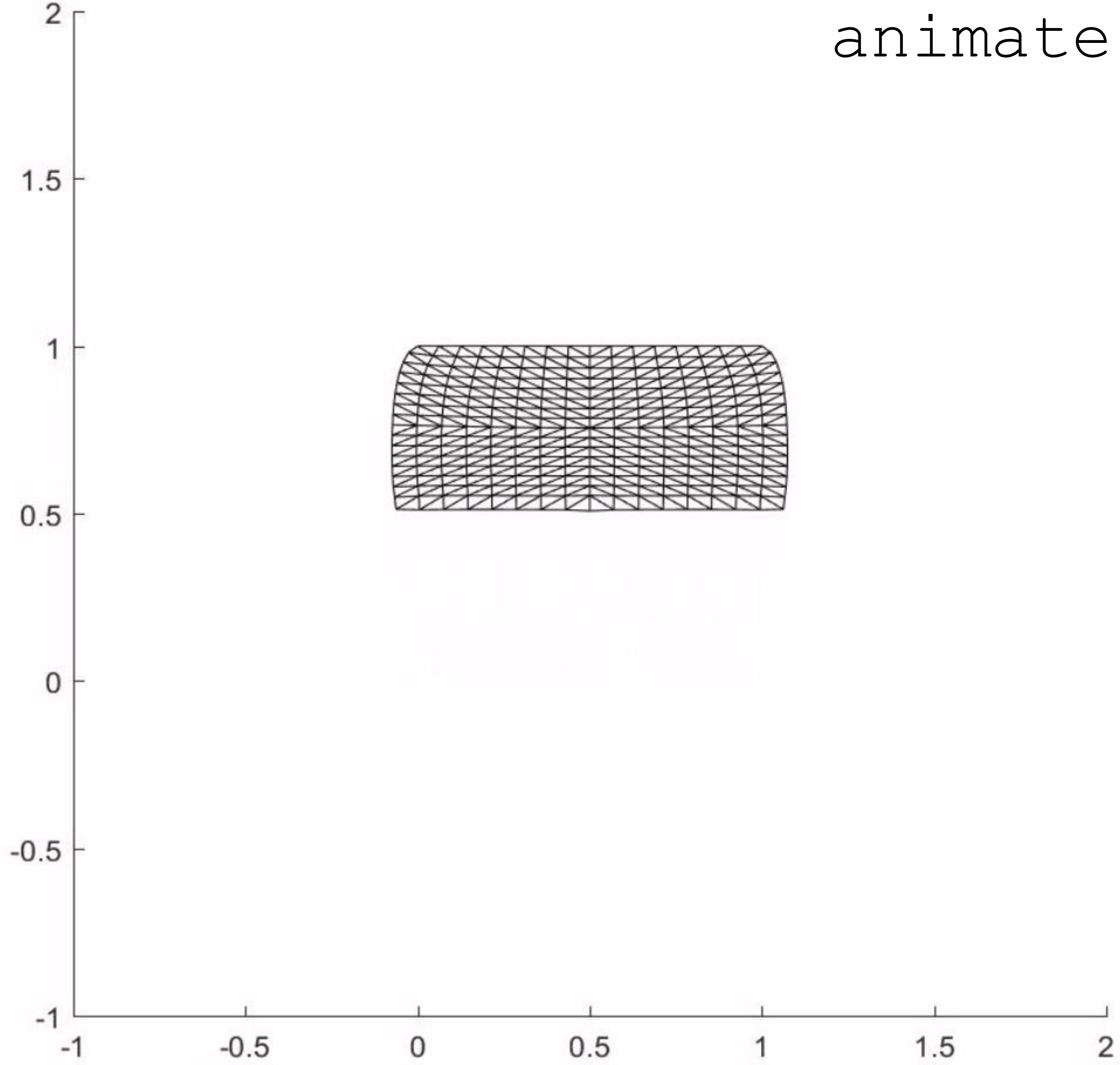


```
function [y] = f2 (x1,x2)

y = zeros(2,1);

    if (x2 <= .05)
        y(1) = 0;
        y(2) = 1;
    end

end
```

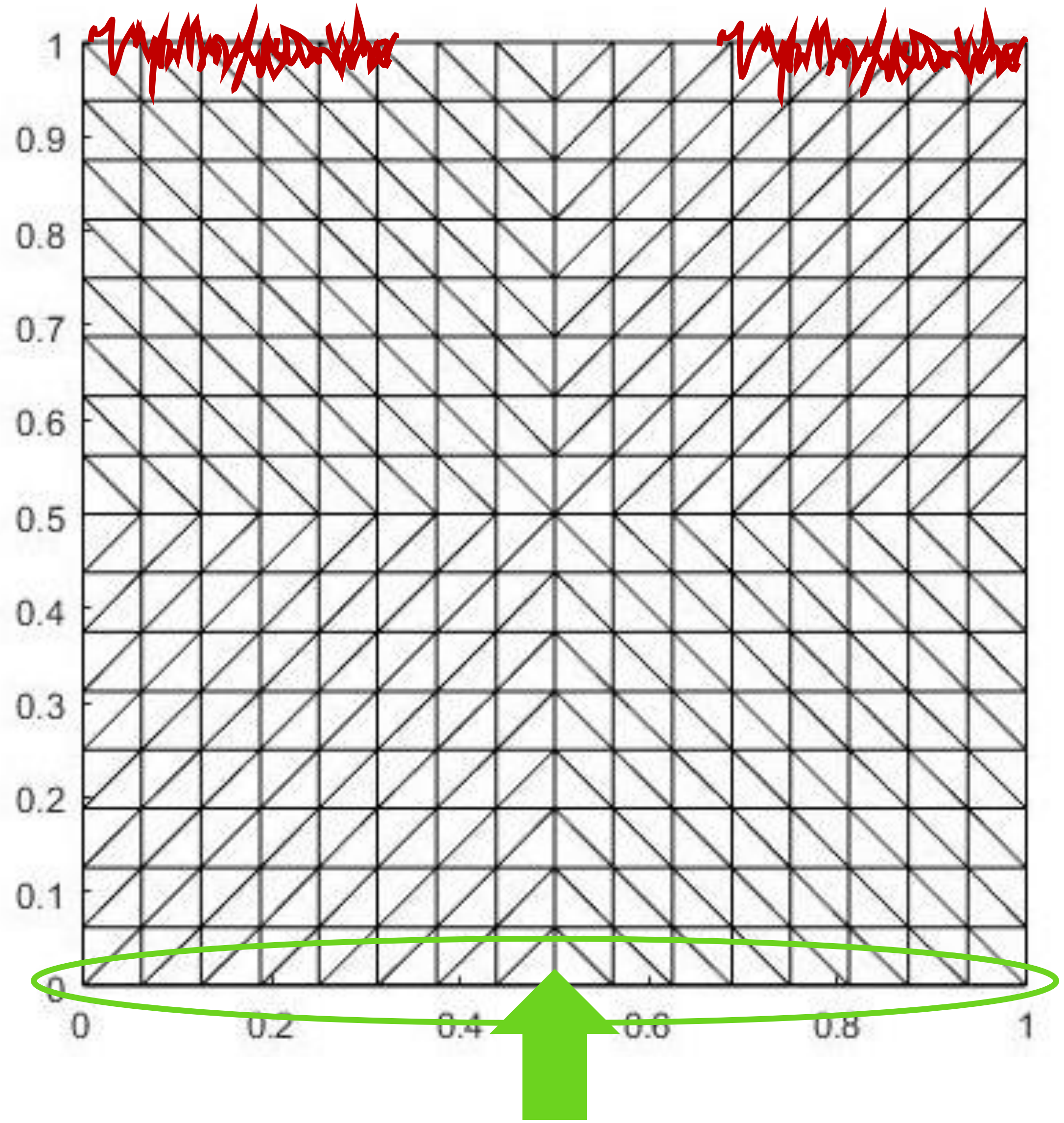



```
u=getuWithBoundary(A,F,v2,0,'and',1,'and',1,'or',-5);
```

```
animate(t2,v2,u,25,300)
```

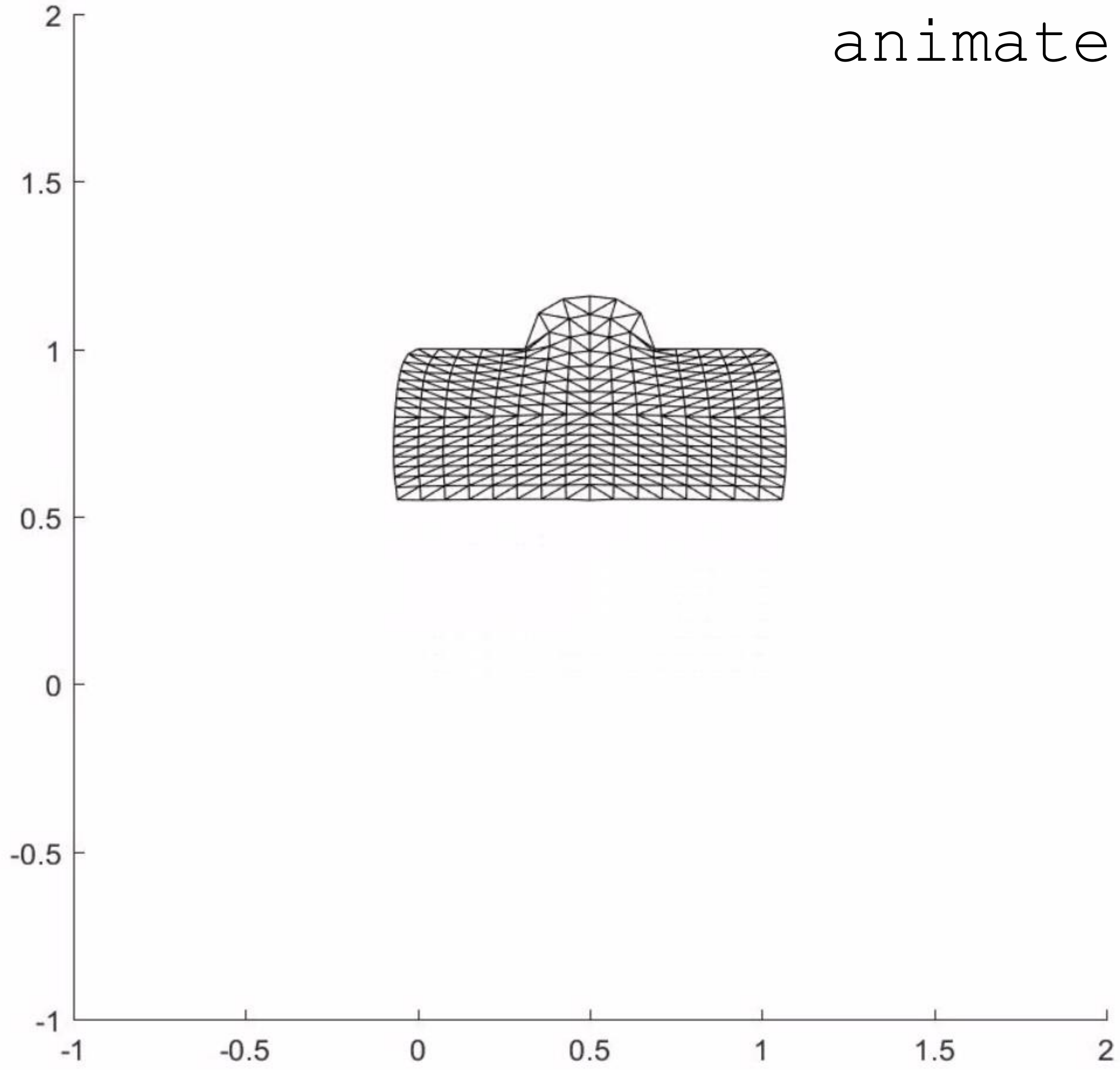


```
function [y] = f3 (x1,x2)

y = zeros(2,1);

    if (x2 <= .05)
        y(1) = 0;
        y(2) = 1;
    end

end
```

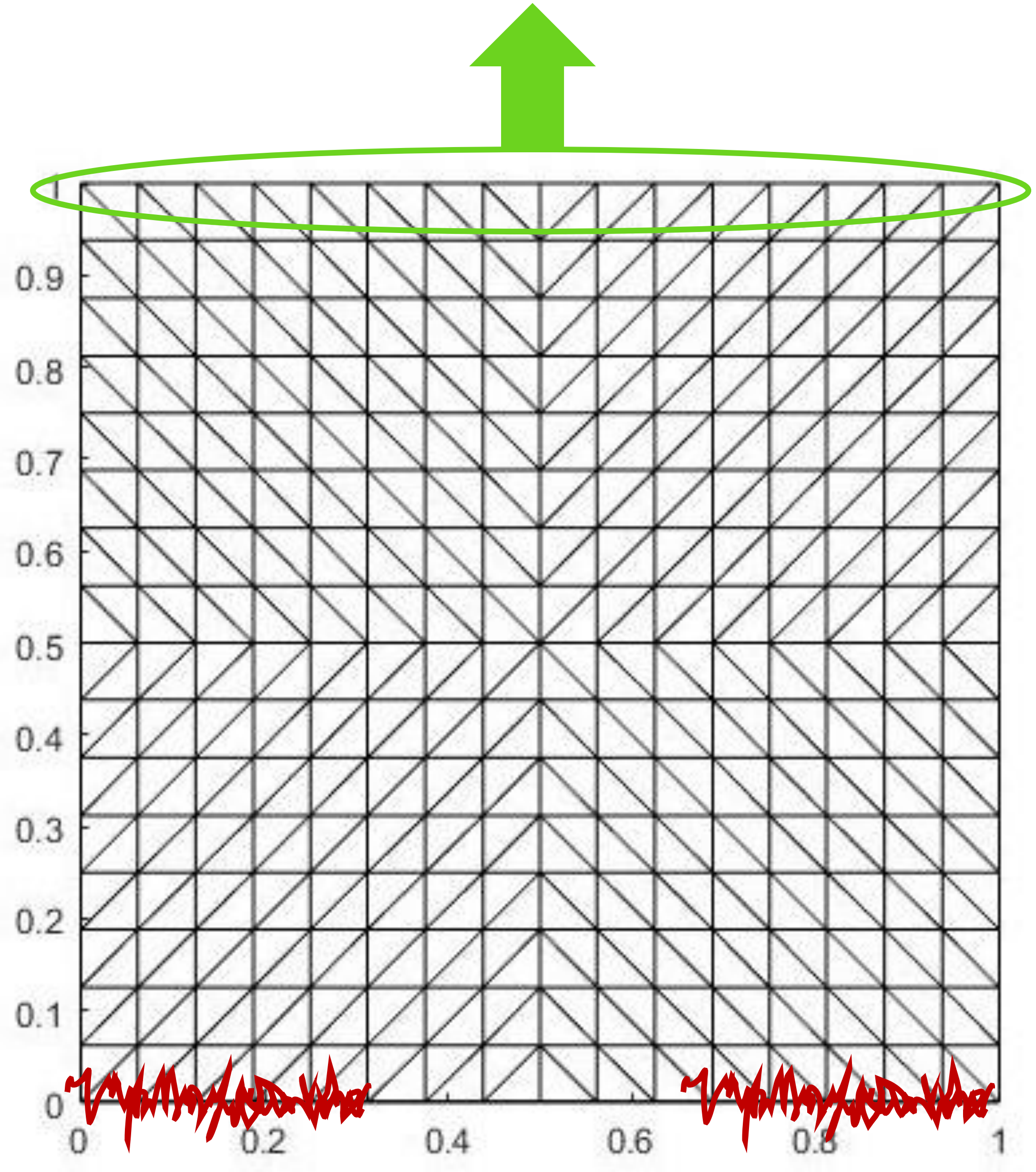



```
u=getuWithBoundary(A,F,v2,2/3,'or',1/3,'and',1,'or',-5);
```

```
animate(t2,v2,u,25,300)
```

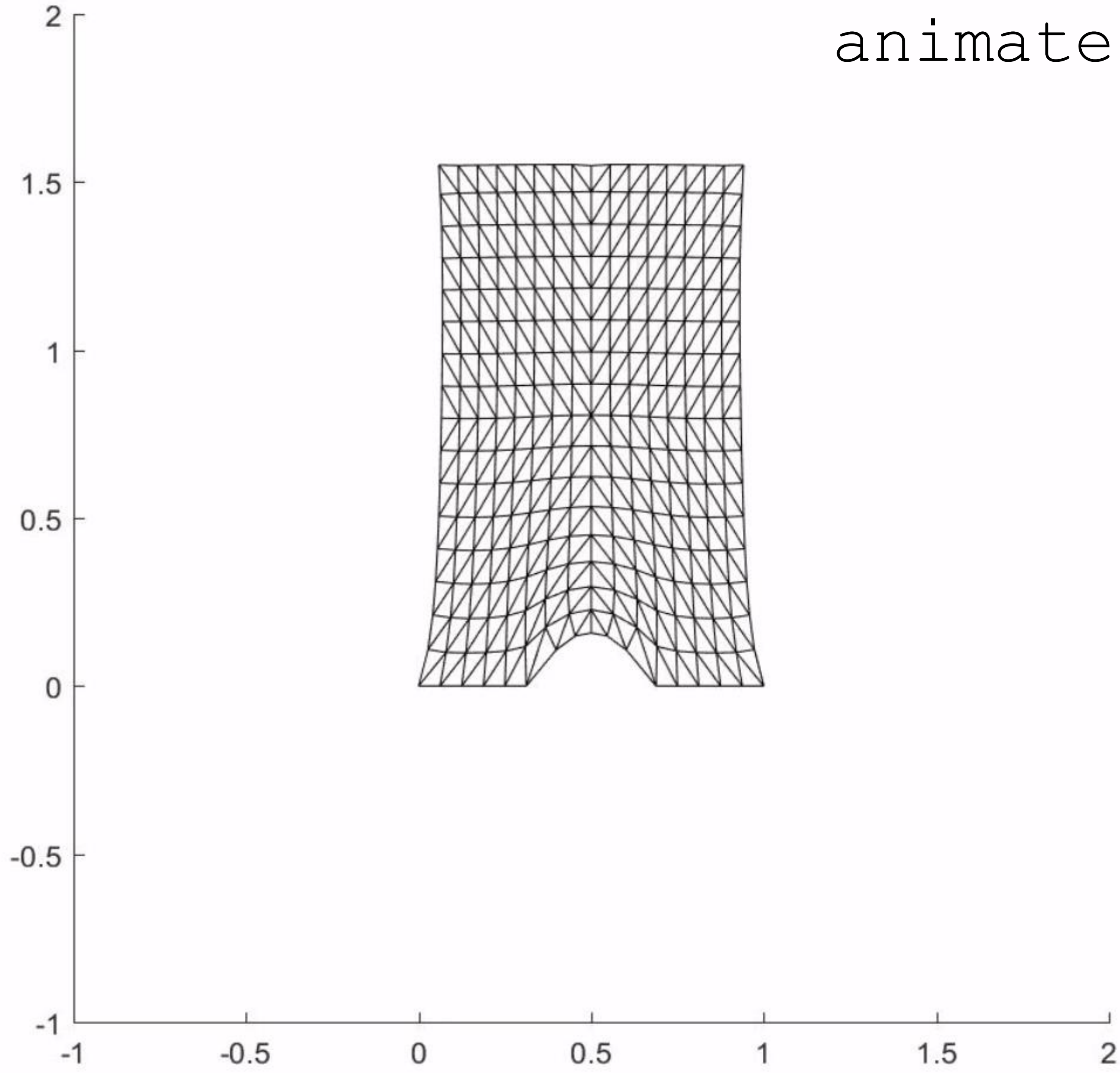


```
function [y] = f4 (x1,x2)

y = zeros(2,1);

    if (x2 >= .95)
        y(1) = 0;
        y(2) = 1;
    end

end
```

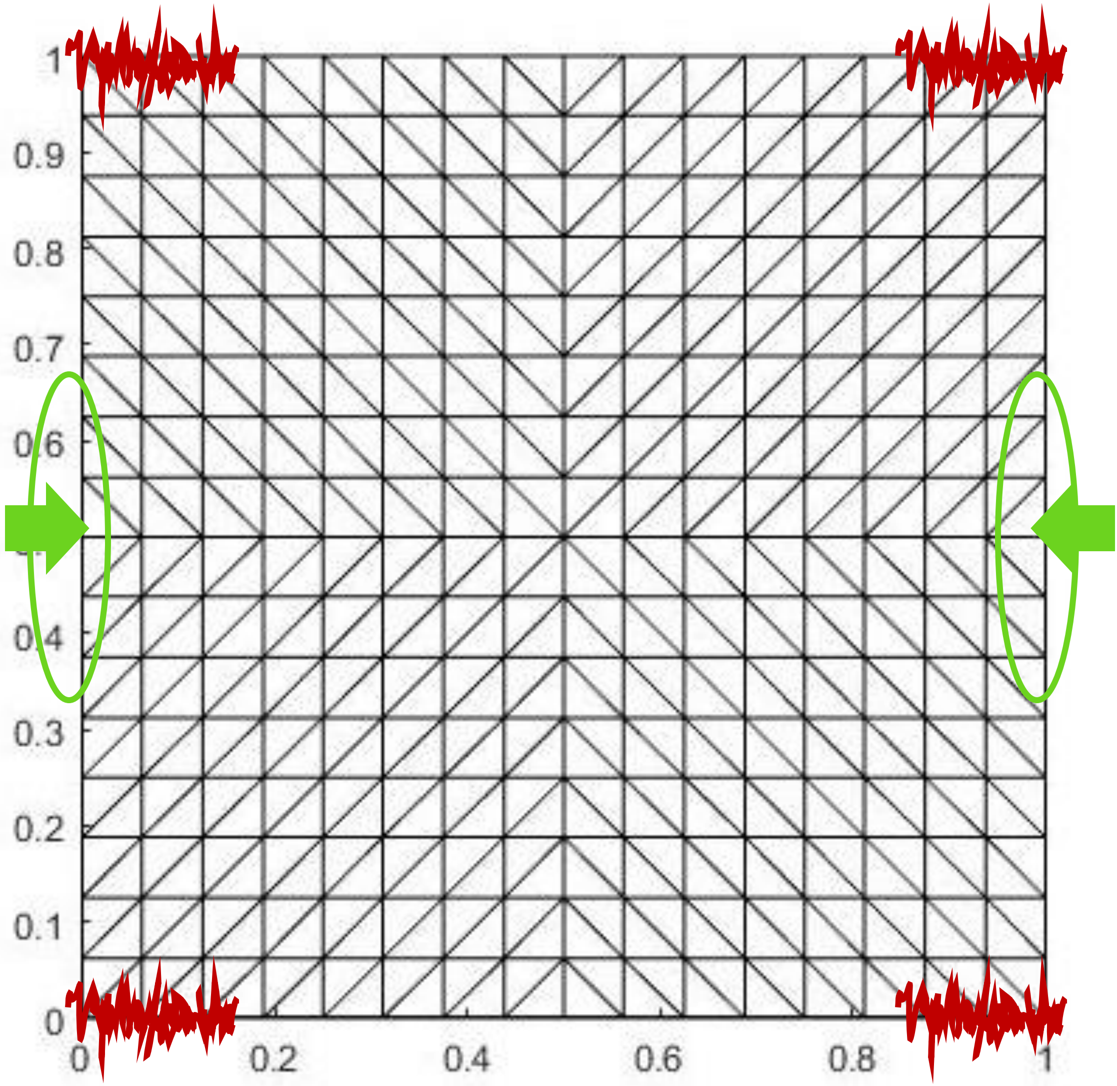



```
u=getuWithBoundary(A,F,v2,2/3,'or',1/3,'and',5,'or',0);
```

```
animate(t2,v2,u,25,300)
```

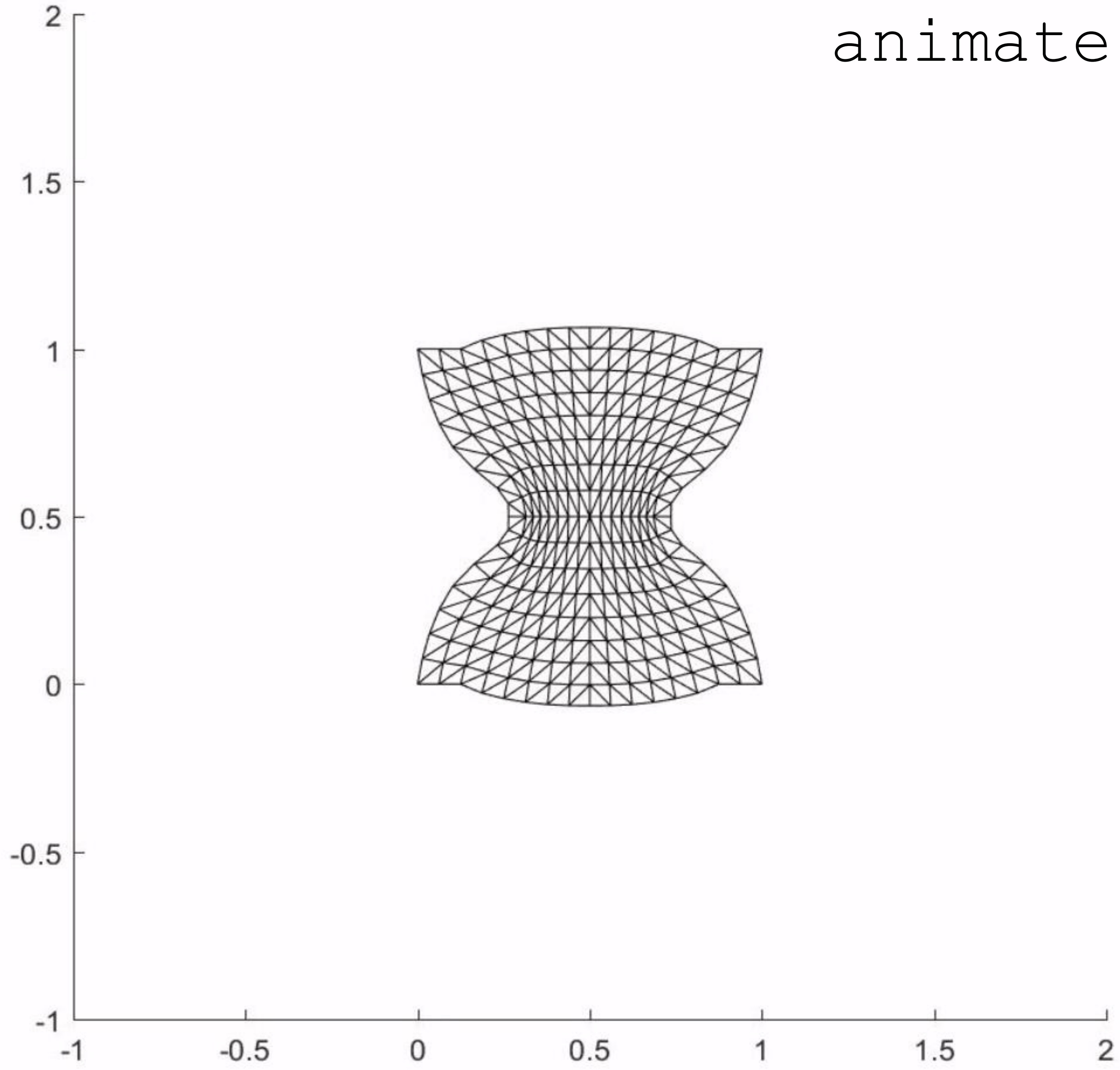


```
function [y] = f5 (x1,x2)

y = zeros(2,1);

   if ((x1 <= .05) & ((x2 <= 2/3) & (x2 >= 1/3))
       y(1) = 1;
       y(2) = 0;
   end

   if ((x1 >= .95) & ((x2 <= 2/3) & (x2 >= 1/3))
       y(1) = -1;
       y(2) = 0;
   end

end
```

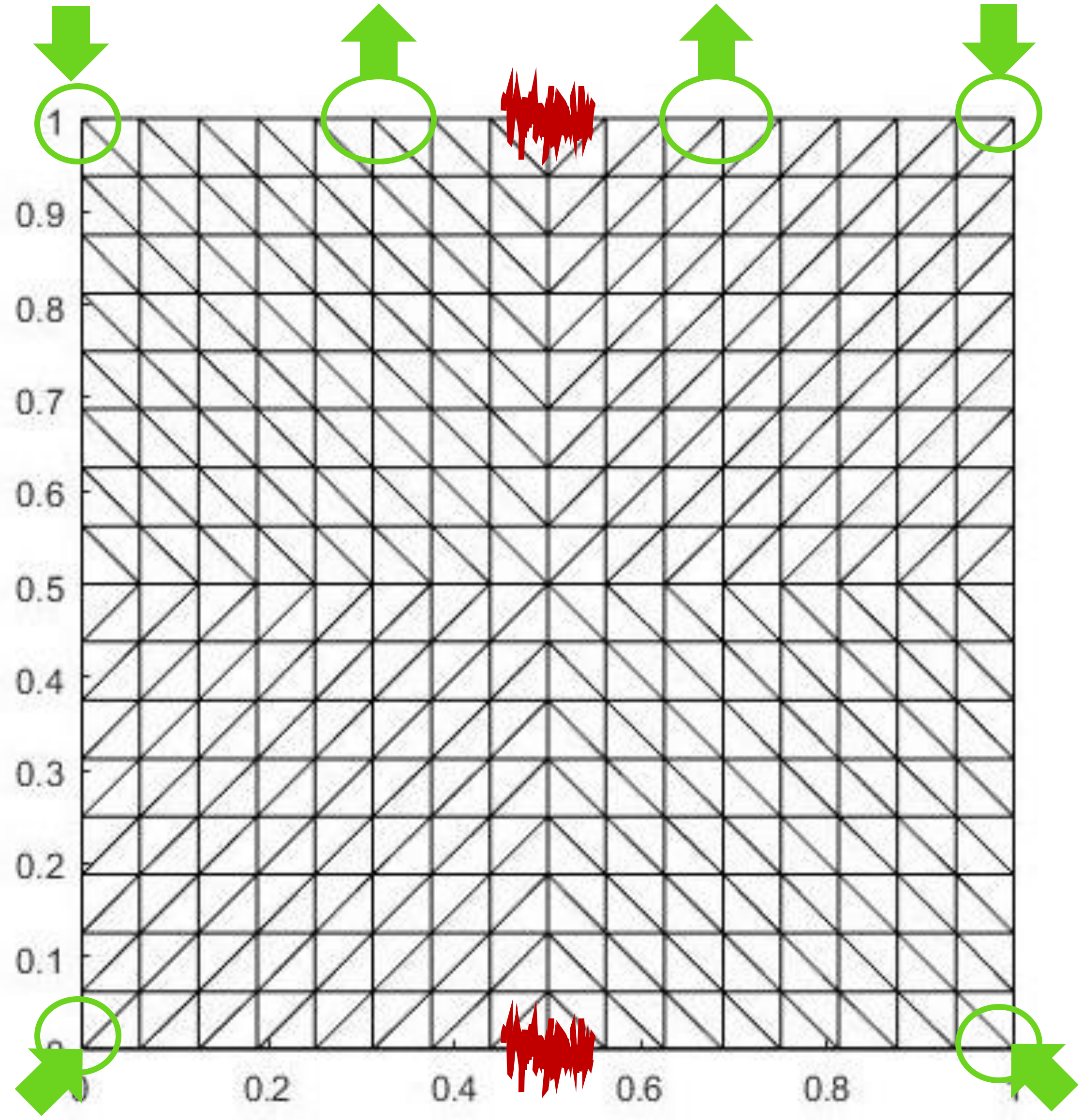



```
u=getuWithBoundary(A,F,v2,1/6,'or',5/6,'and',1,'or',0);
```

```
animate(t2,v2,u,25,500)
```

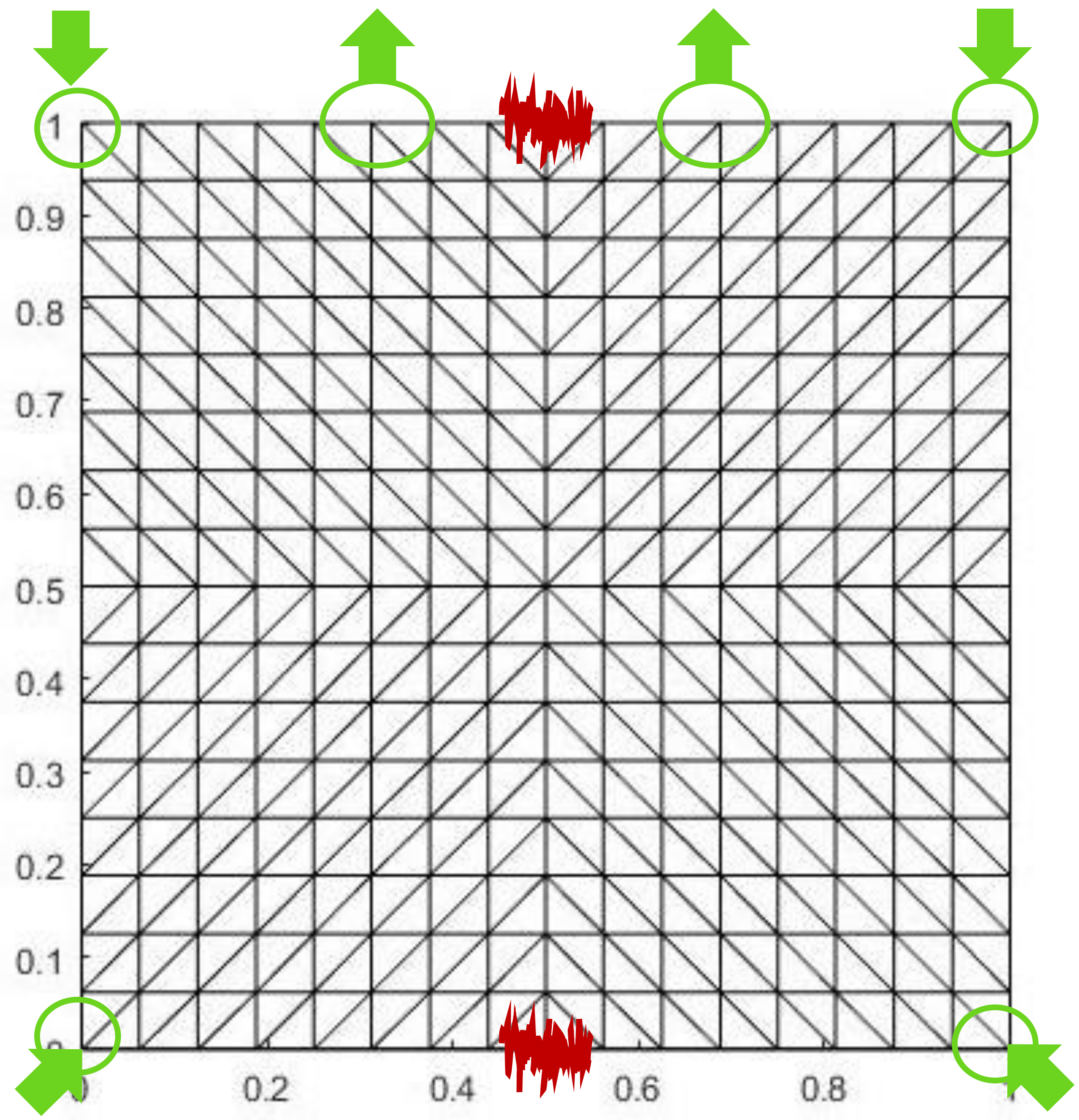


```
function [y] = f6 (x1,x2)

y = zeros(2,1);

   if ((x2 >= .95) & ((x1<=.4) | (x1>=.3)))
       y(1) = 0;
       y(2) = .5;
   end

   if ((x2 >= .95) & ((x1<=.7) | (x1>=.60)))
       y(1) = 0;
       y(2) = .5;
   end

   if ((x2 <= .1) & (x1<=.1))
       y(1) = 1;
       y(2) = 1;
   end
```



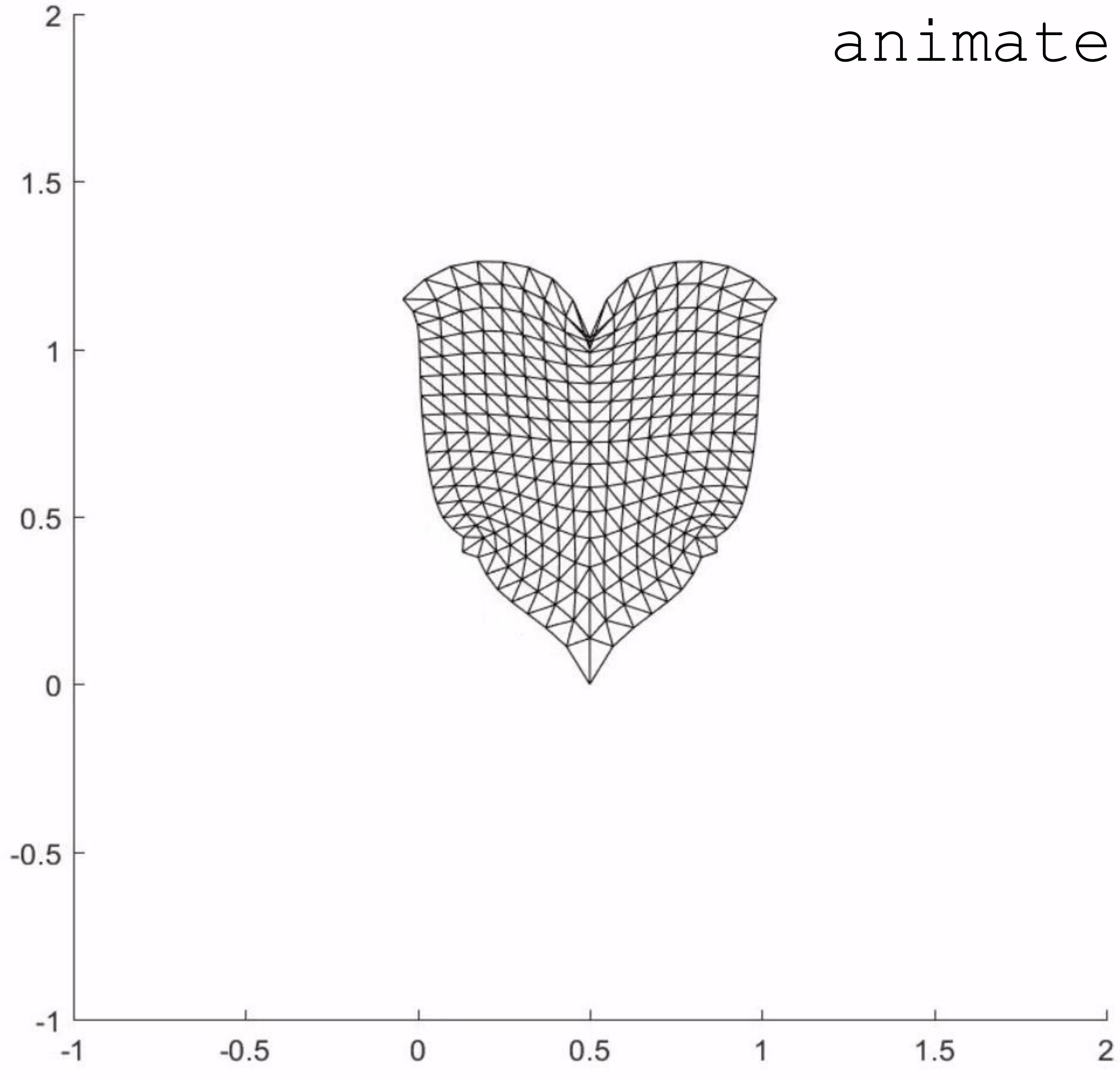

```
u=getuWithBoundary(A,F,v2,.5,'and',.55,'and',.99,'or',.01);
```

```
    if ((x2 <= .1) & (x1>=.9))
        y(1) = -1;
        y(2) = 1;
    end


    if ((x2 >= .95) & (x1<=.05))
        y(1) = 0;
        y(2) = -1.9;
    end

    if ((x2 >= .95) & (x1>=.95))
        y(1) = 0;
        y(2) = -1.9;
    end

end
```

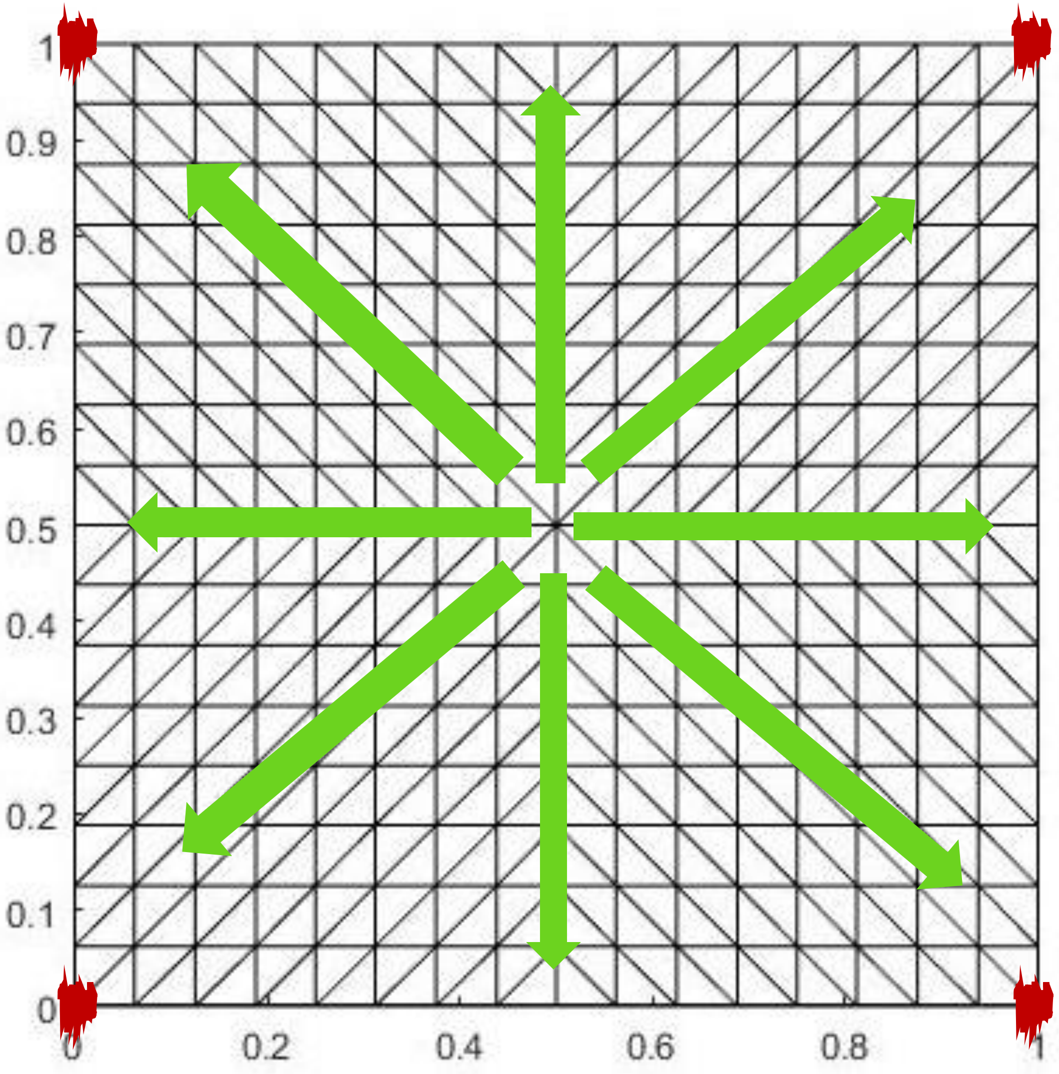



```
u=getuWithBoundary(A,F,v2,.5,'and',.55,'and',.99,'or',.01);
```

```
animate(t2,v2,u,10,250)
```

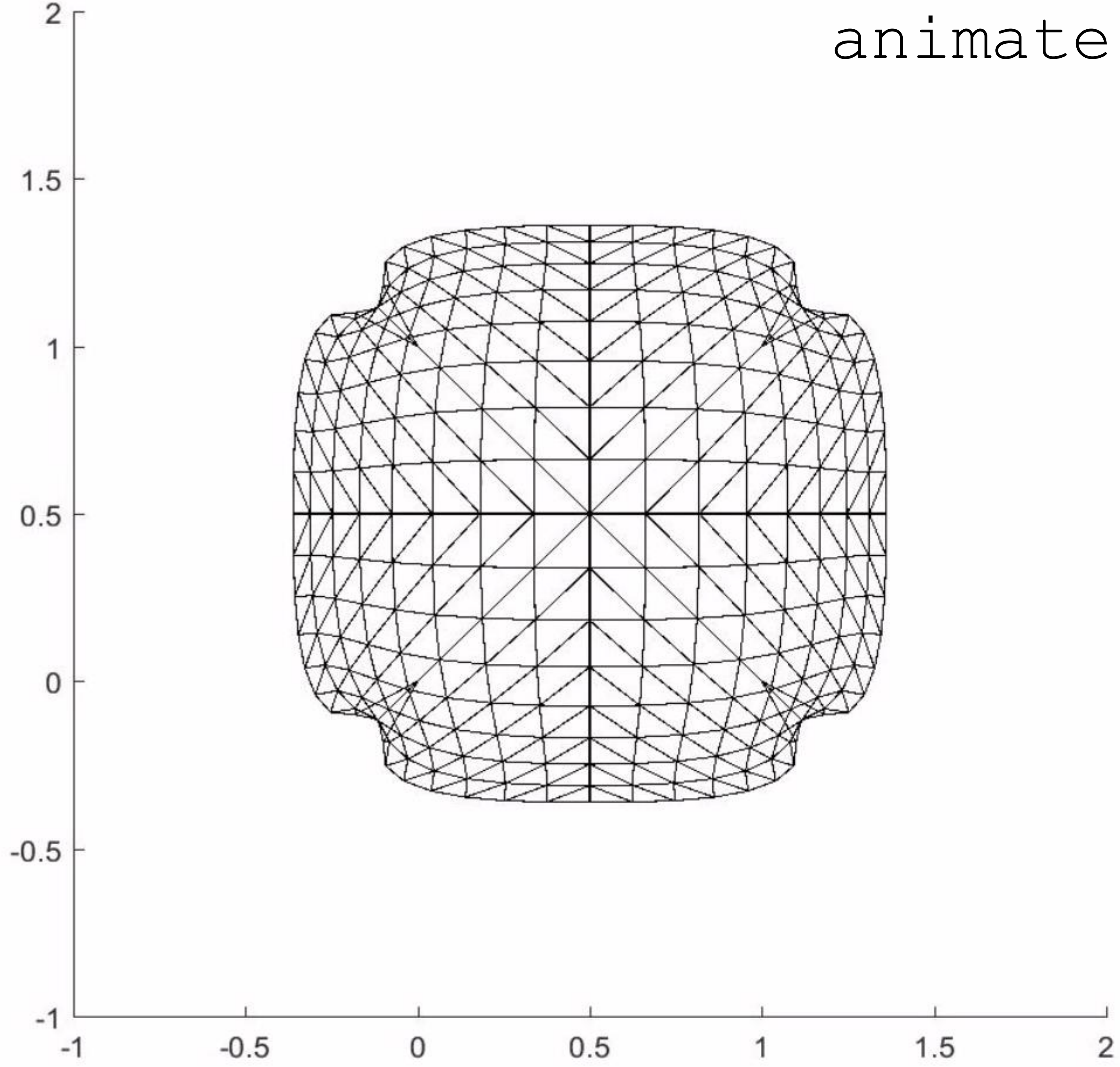


```
function [y] = f7 (x1,x2)

y = zeros(2,1);

   mag = (1/((x1-.5)^(2) + (x2-.5)^(2) + .01));

   y = [mag*(x1-.5);mag*(x2-.5)];

end
```

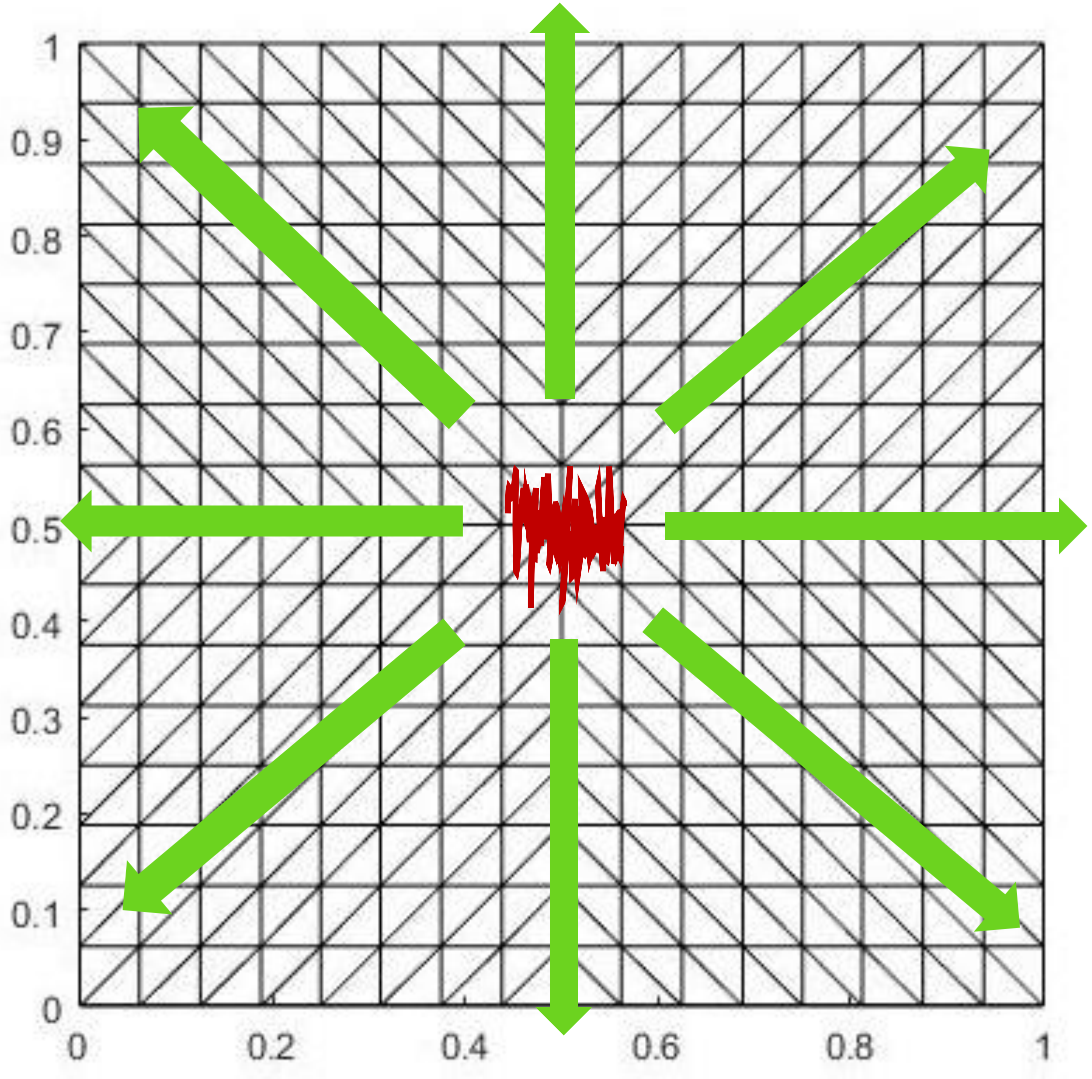



```
u=getuWithBoundary(A,F,v2,1,'or',0,'and',1,'or',0);
```

```
animate(t2,v2,u,10,150)
```

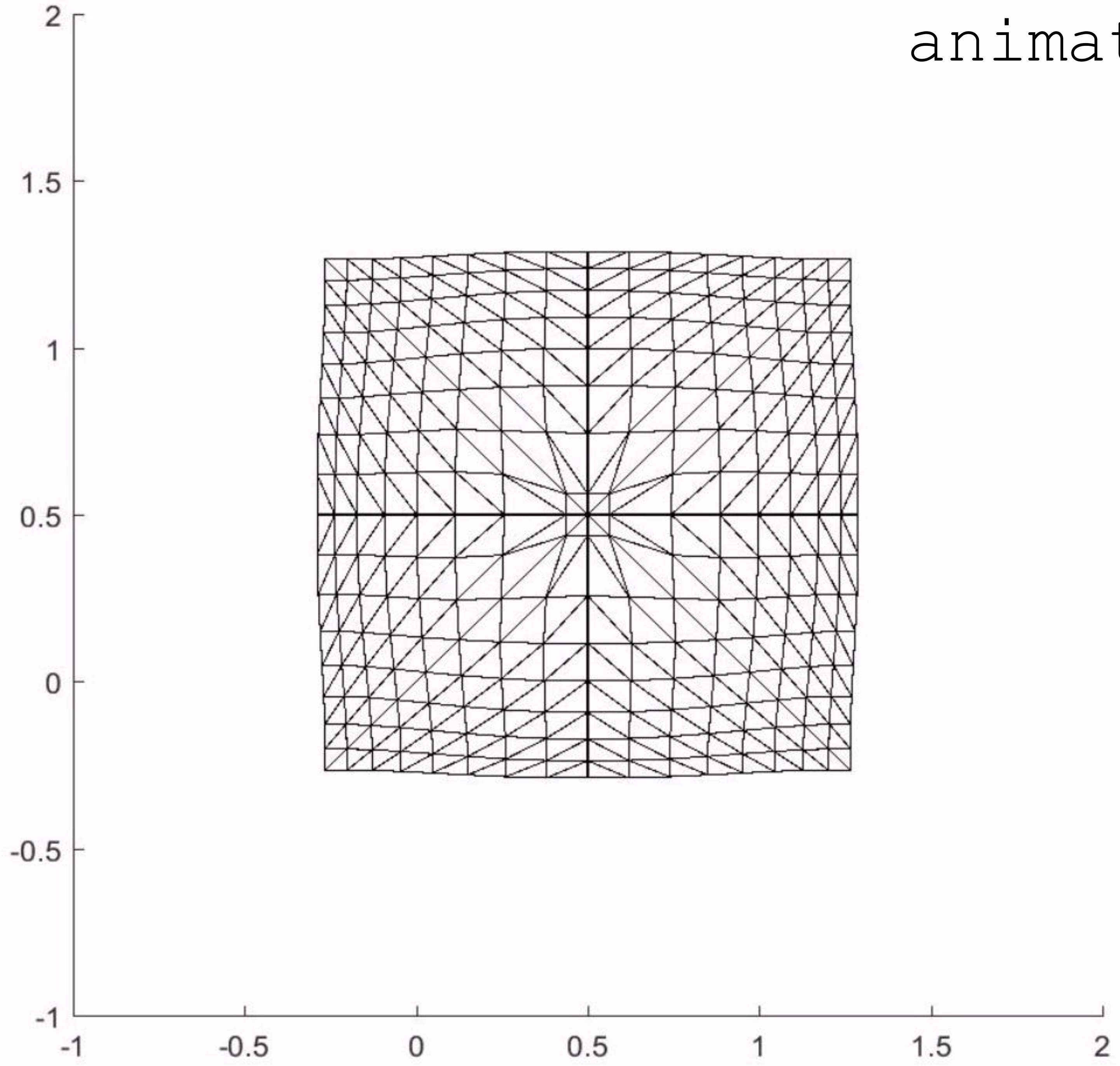


```
function [y] = f8 (x1,x2)

y = zeros(2,1);

    mag = (1/((x1-.5)^(2) + (x2-.5)^(2) + .01));

    y = [mag*(x1-.5);mag*(x2-.5)];

end
```

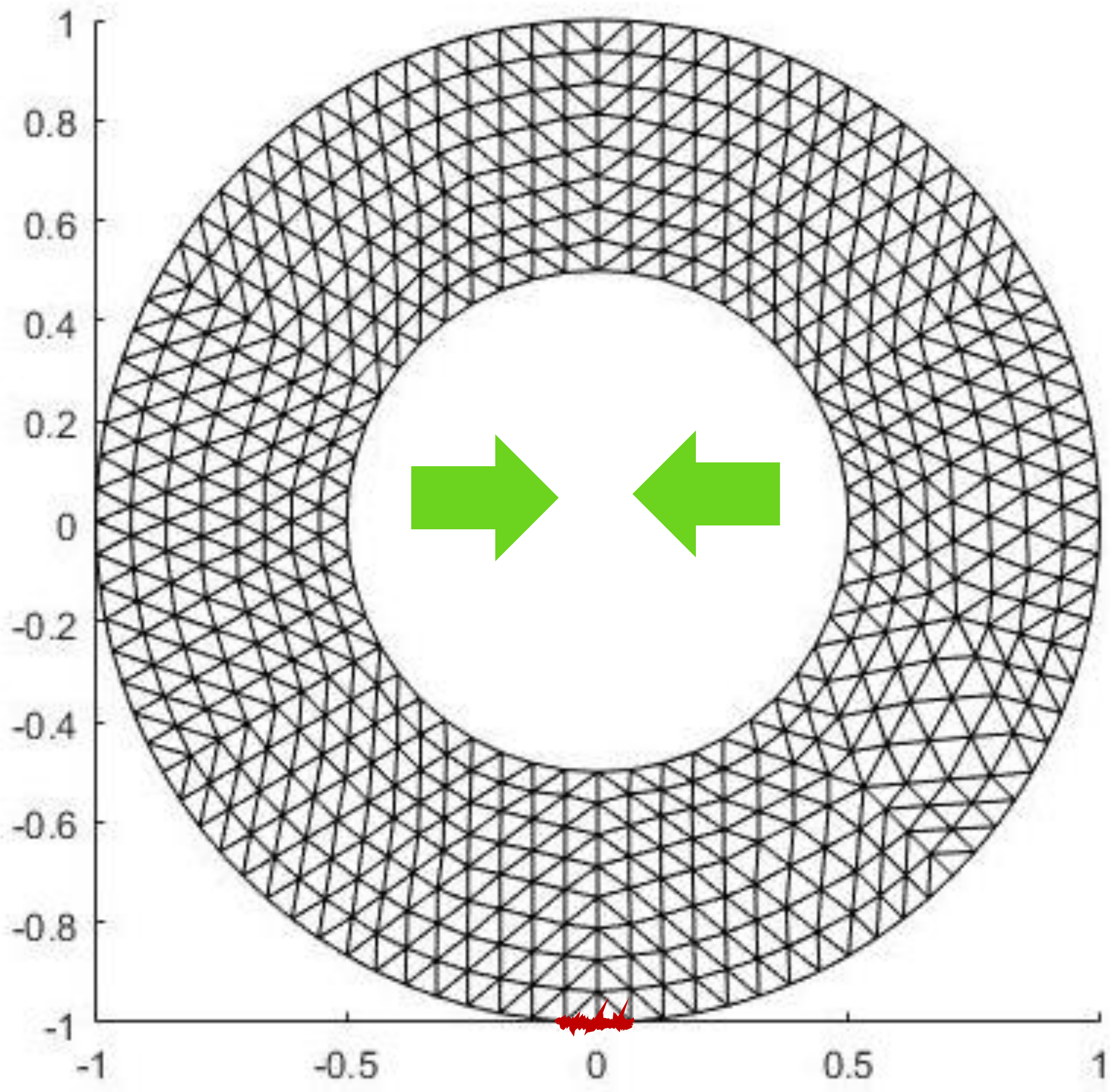



```
u=getuWithBoundary(A,F,v2,.4,'and',.6,'and',.4,'and',.6);
```

animate(t2,v2,u,5,60)

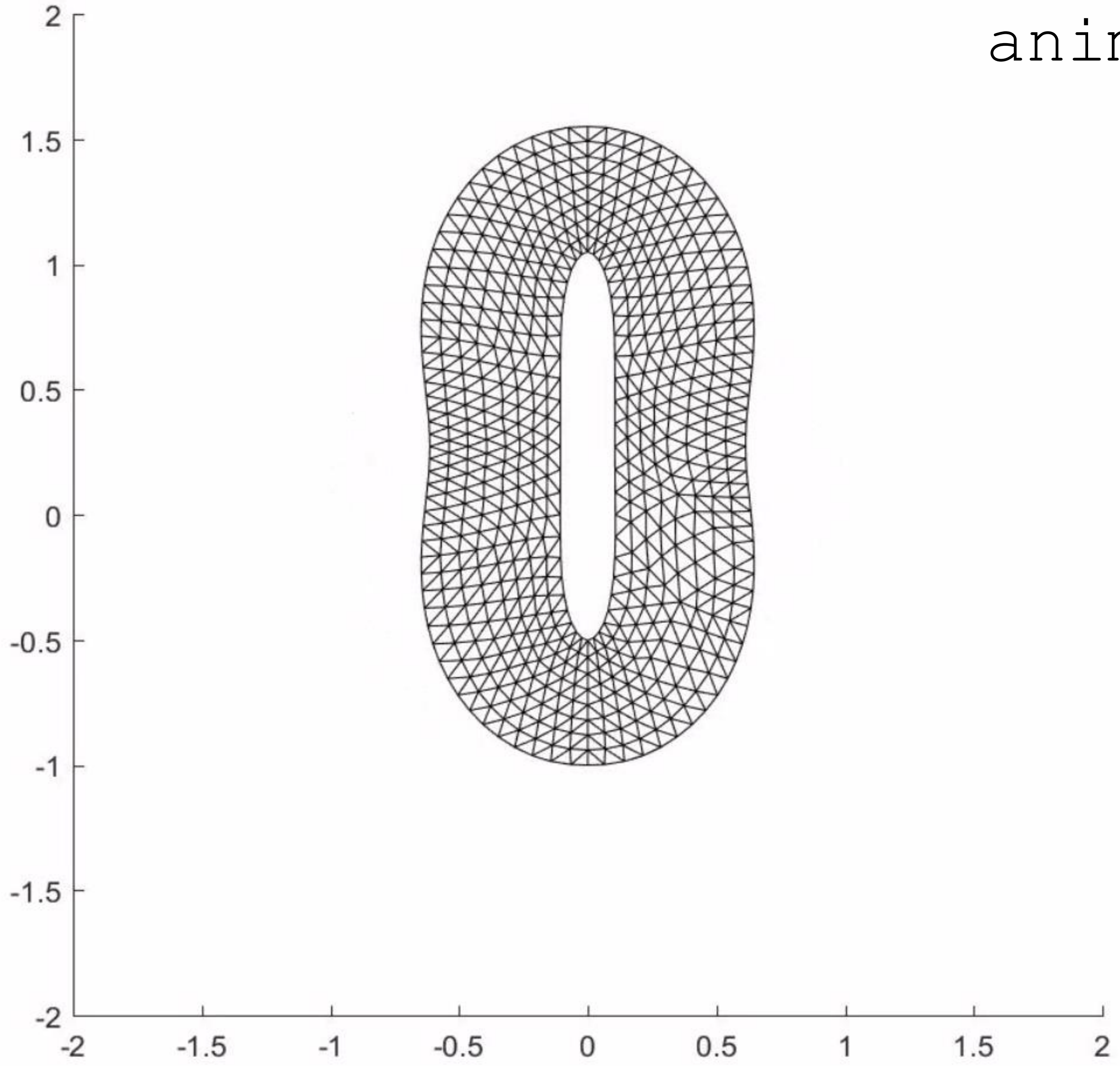


```
function [y] = f9 (x1,x2)

y = zeros(2,1);

   y = [-(x1);0];

end
```

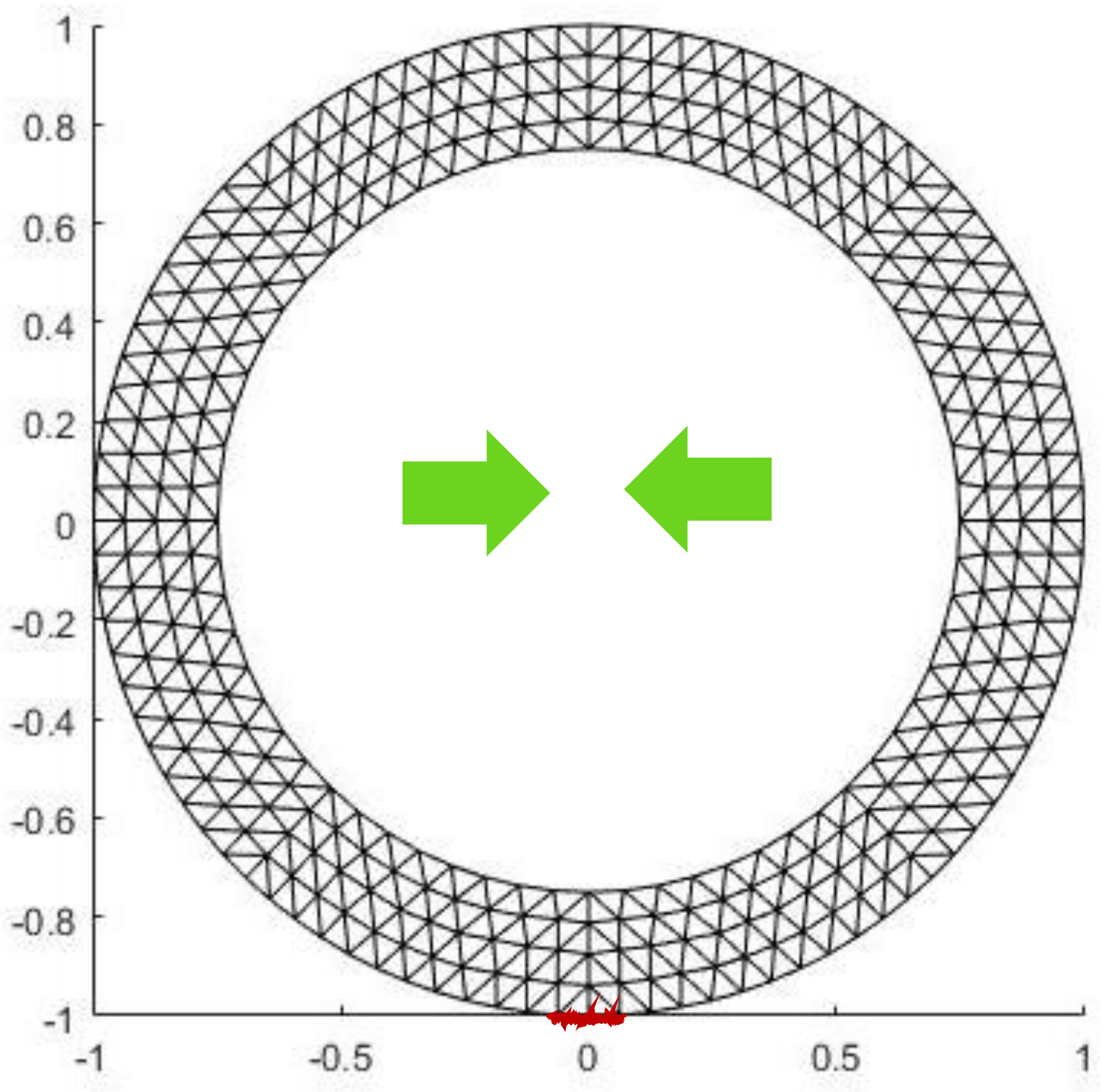



```
u=getuWithBoundary(A,F,v,-1,'and',1,'and',-1,'and',-.995);
```

```
animate(t,v,u,.5,8)
```

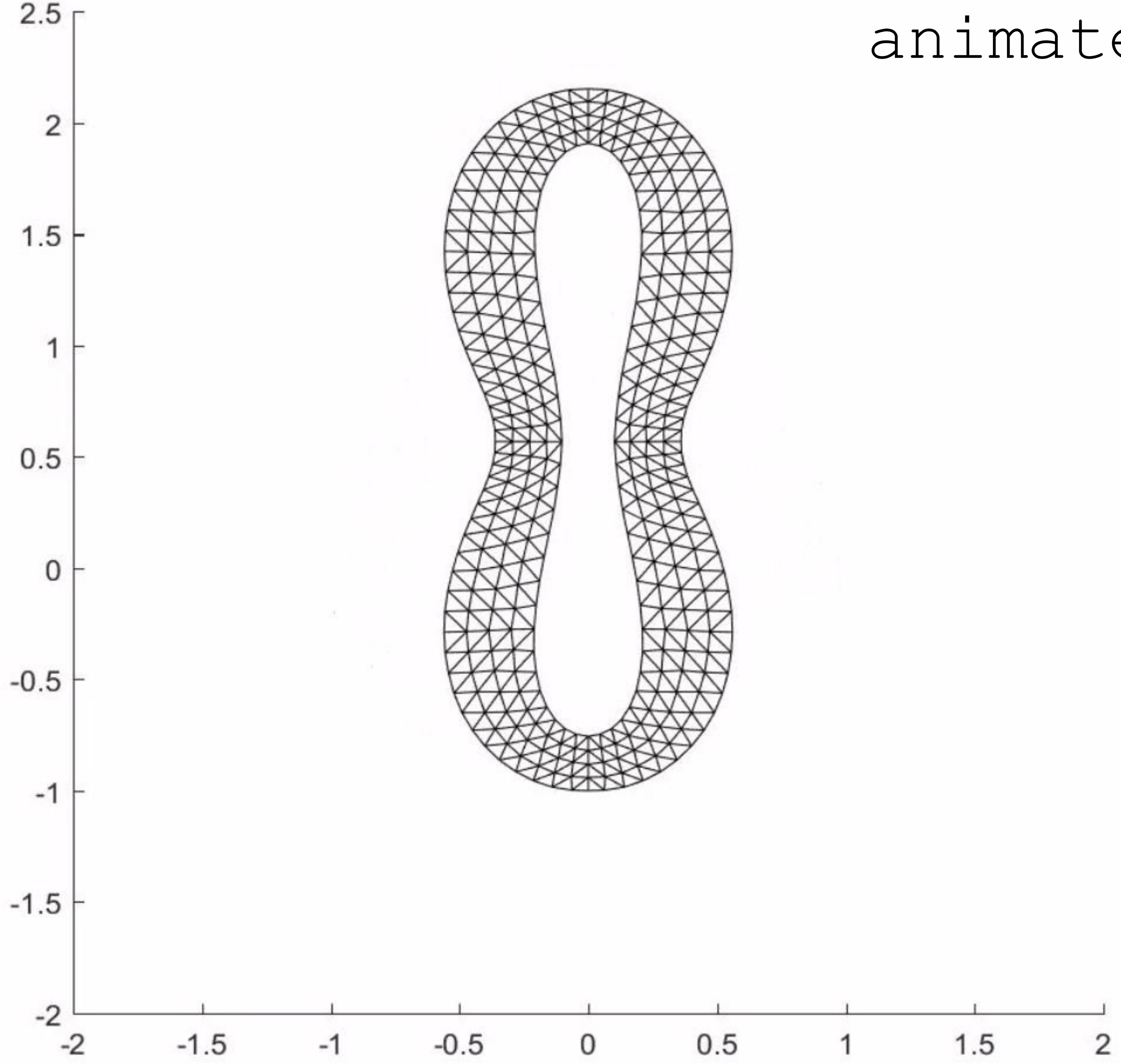


```
function [y] = f10 (x1,x2)

y = zeros(2,1);

    y = [-(x1);0];

end
```

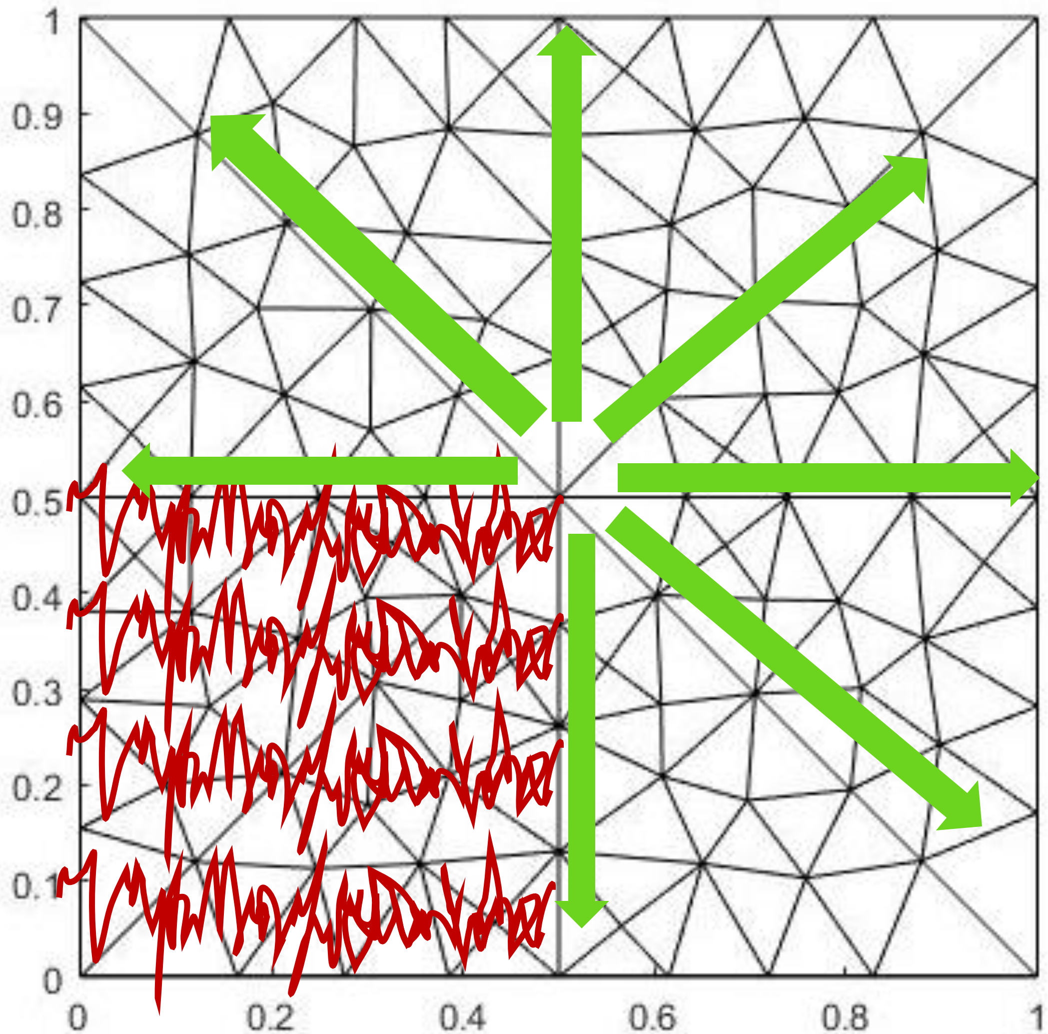



```
u=getuWithBoundary(A,F,v,-1,'and',1,'and',-1,'and',-.995);
```

animate(t,v,u,.25,2.5)

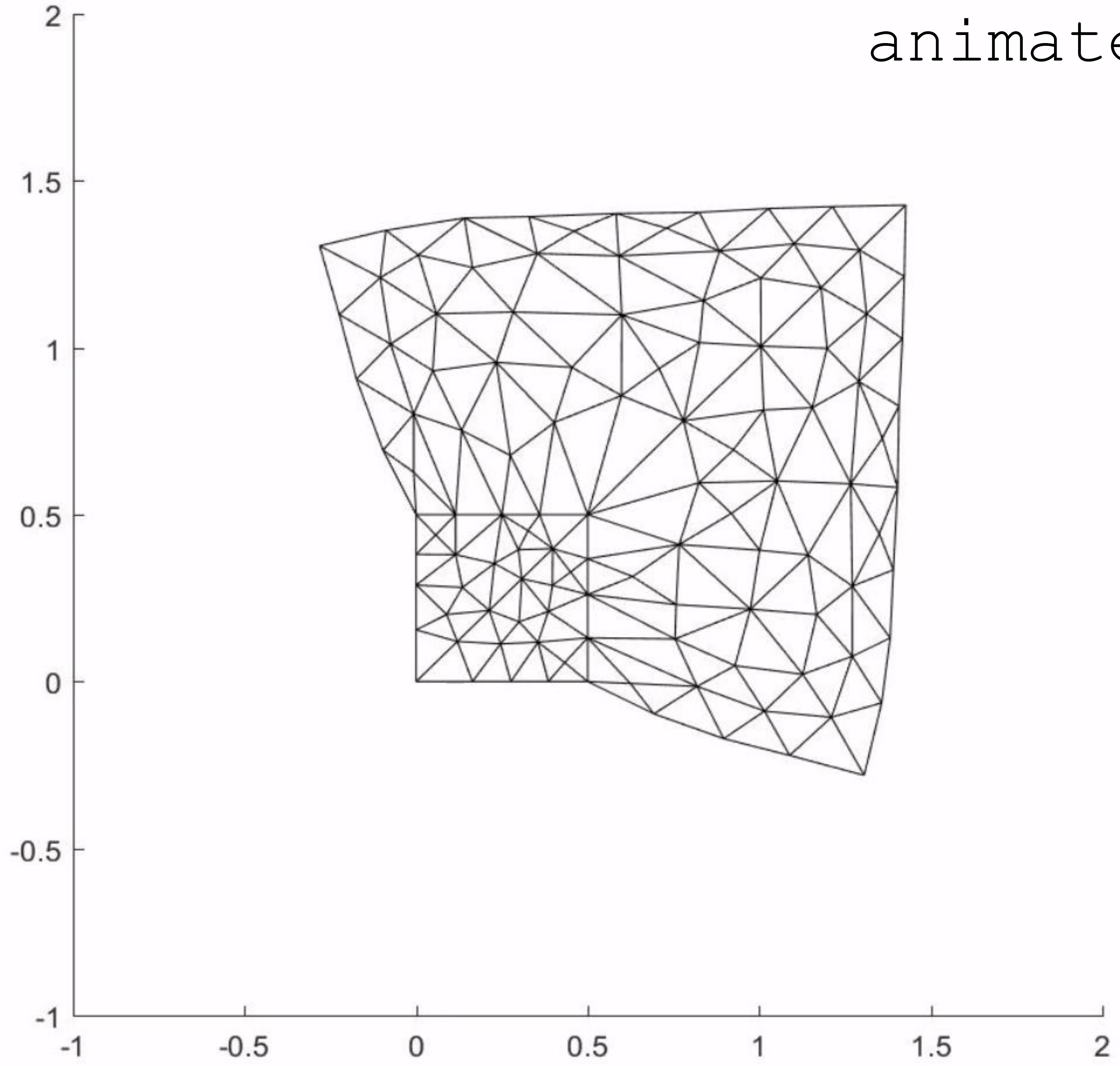


```
function [y] = f11 (x1,x2)

y = zeros(2,1);

    mag = (1/(((x1-.5)^(2) + (x2-.5)^(2)) + .01));

    y = [mag*(x1-.5);mag*(x2-.5)];

end
```

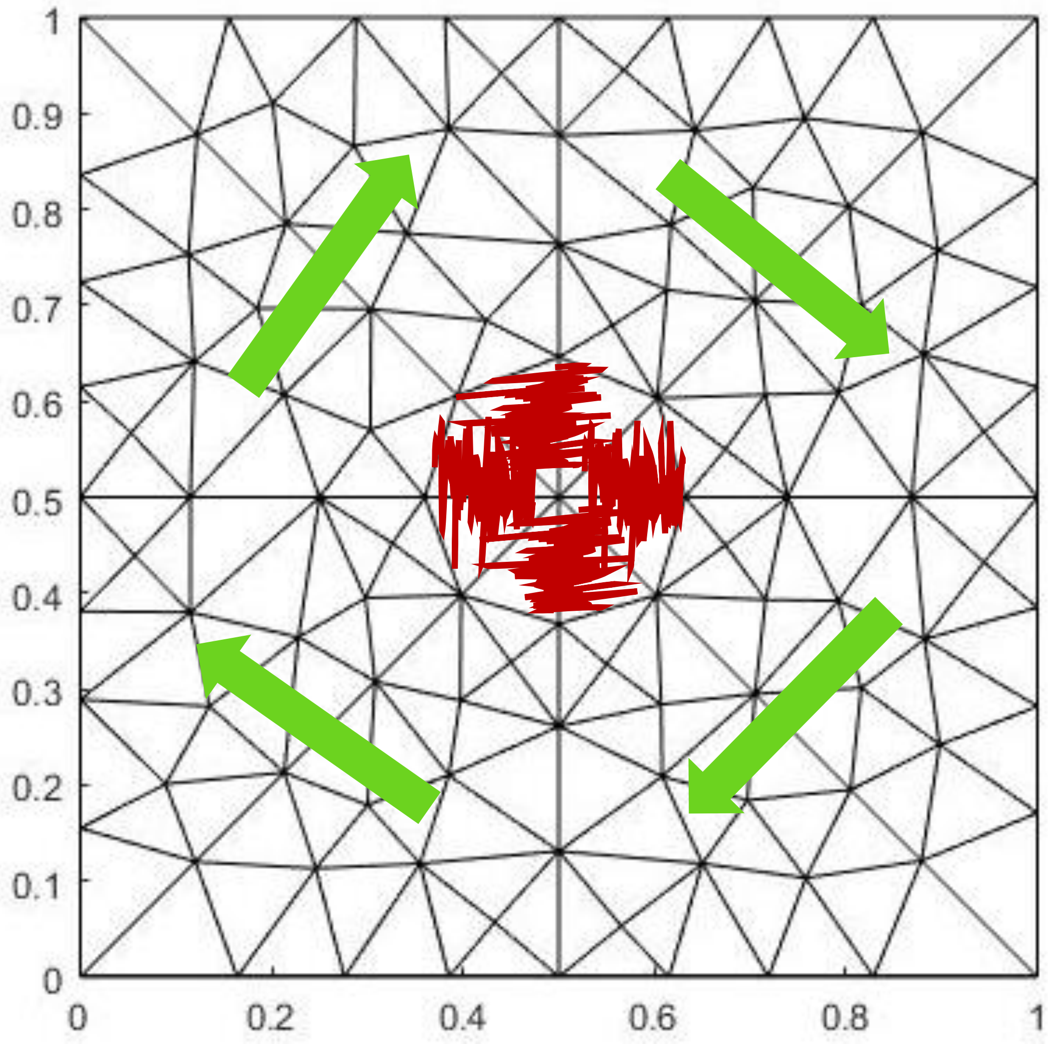



```
u=getuWithBoundary(A,F,v2,0,'and',.5,'and',0,'and',.5);
```

animate(t2,v2,u,5,100)

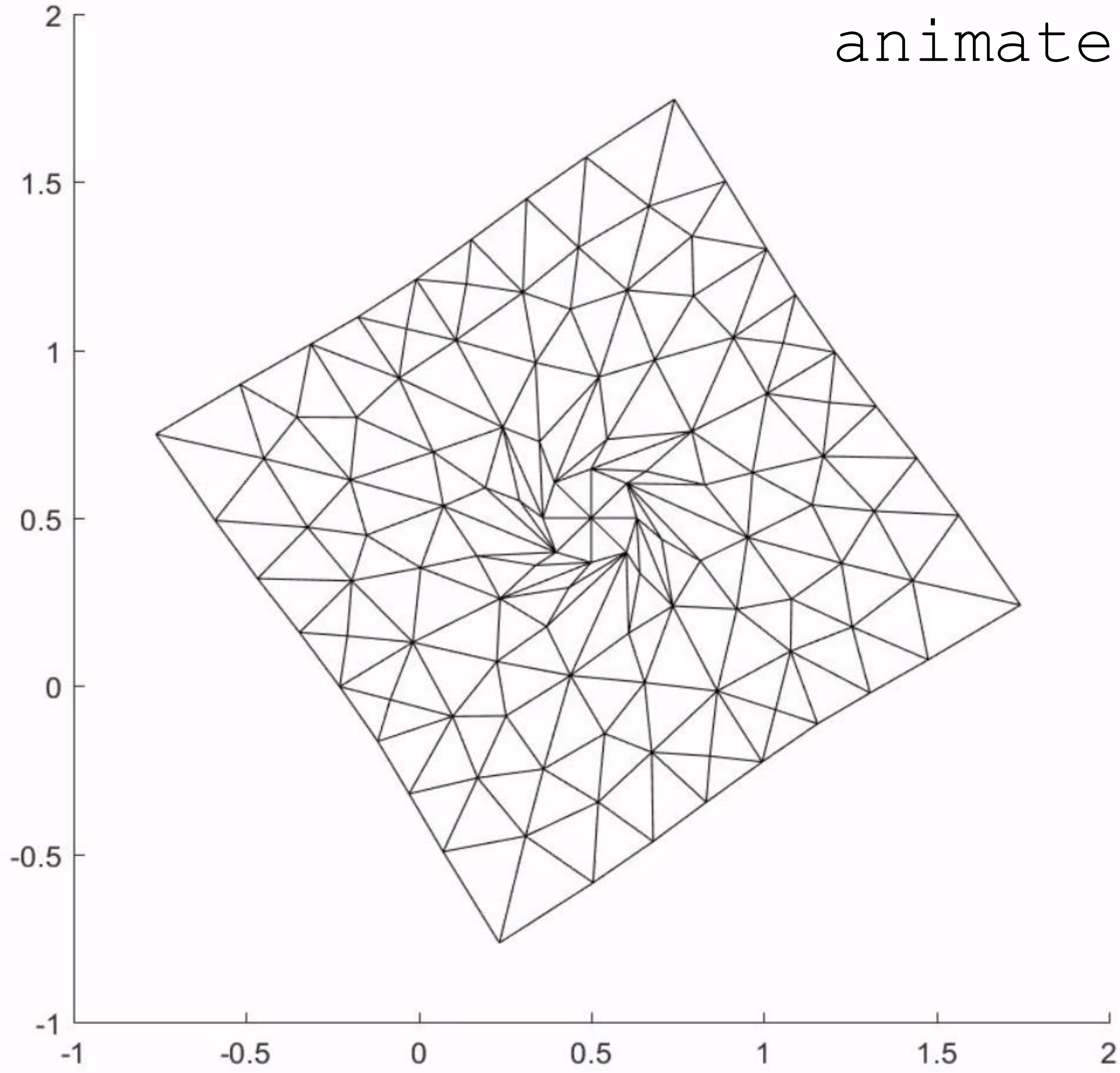


```
function [y] = f12 (x1,x2)

y = zeros(2,1);

    if ((x1-.5 >= 0) & (x2-.5 >=0))
        y(1) = (x2-.5);
        y(2) = -(x1-.5);
    end

    if ((x1-.5 >= 0) & (x2-.5 <=0))
        y(1) = (x2-.5);
        y(2) = -(x1-.5);
    end

    if ((x1-.5 <= 0) & (x2-.5 <=0))
        y(1) = (x2-.5);
        y(2) = -(x1-.5);
    end

    if ((x1-.5 <= 0) & (x2-.5 >=0))
        y(1) = (x2-.5);
        y(2) = -(x1-.5);
    end

end
```

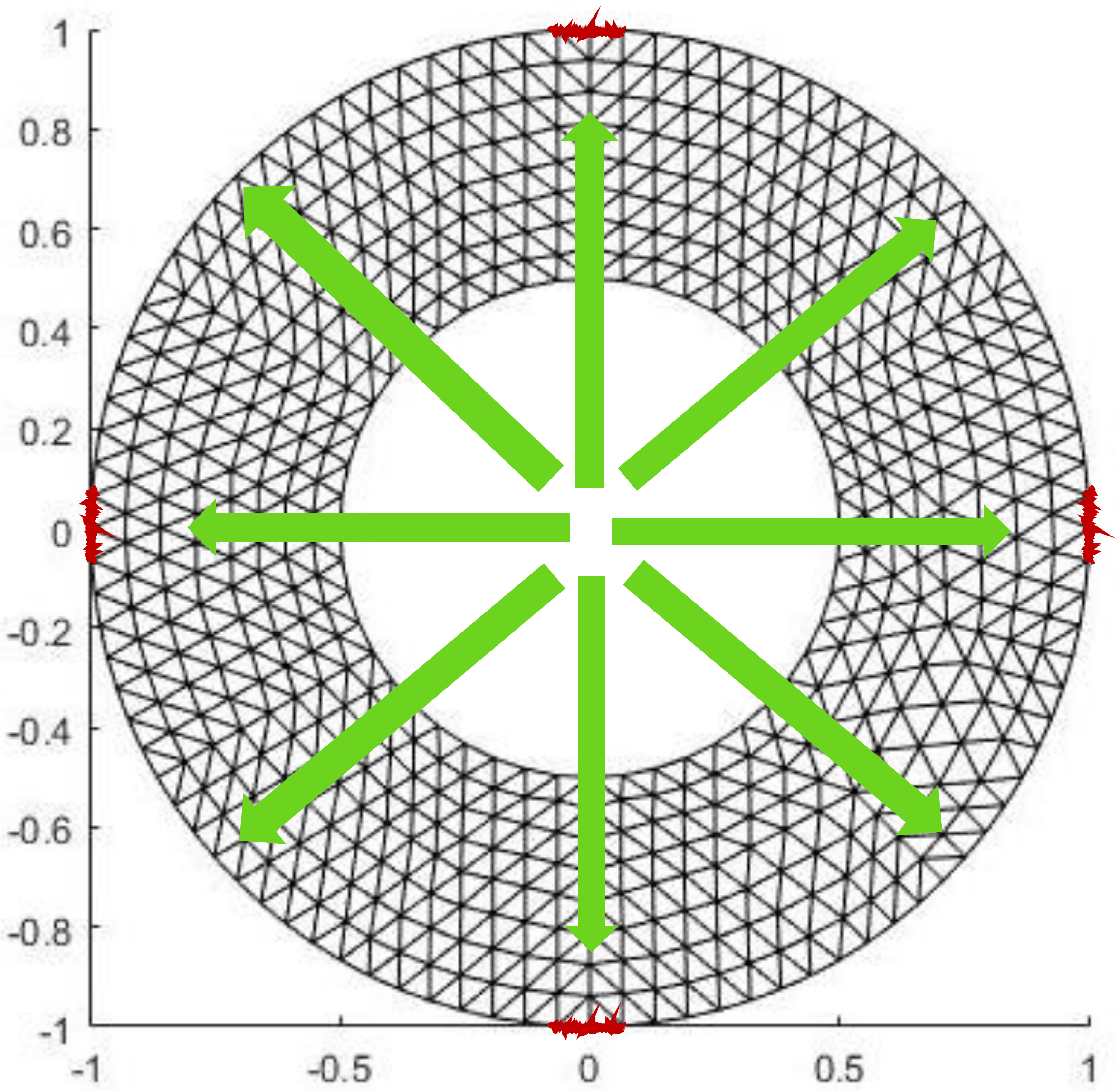



```
u=getuWithBoundary(A,F,v2,.35,'and',.66,'and',.35,'and',.66);
```

`animate(t2,v2,u,2.5,35)`

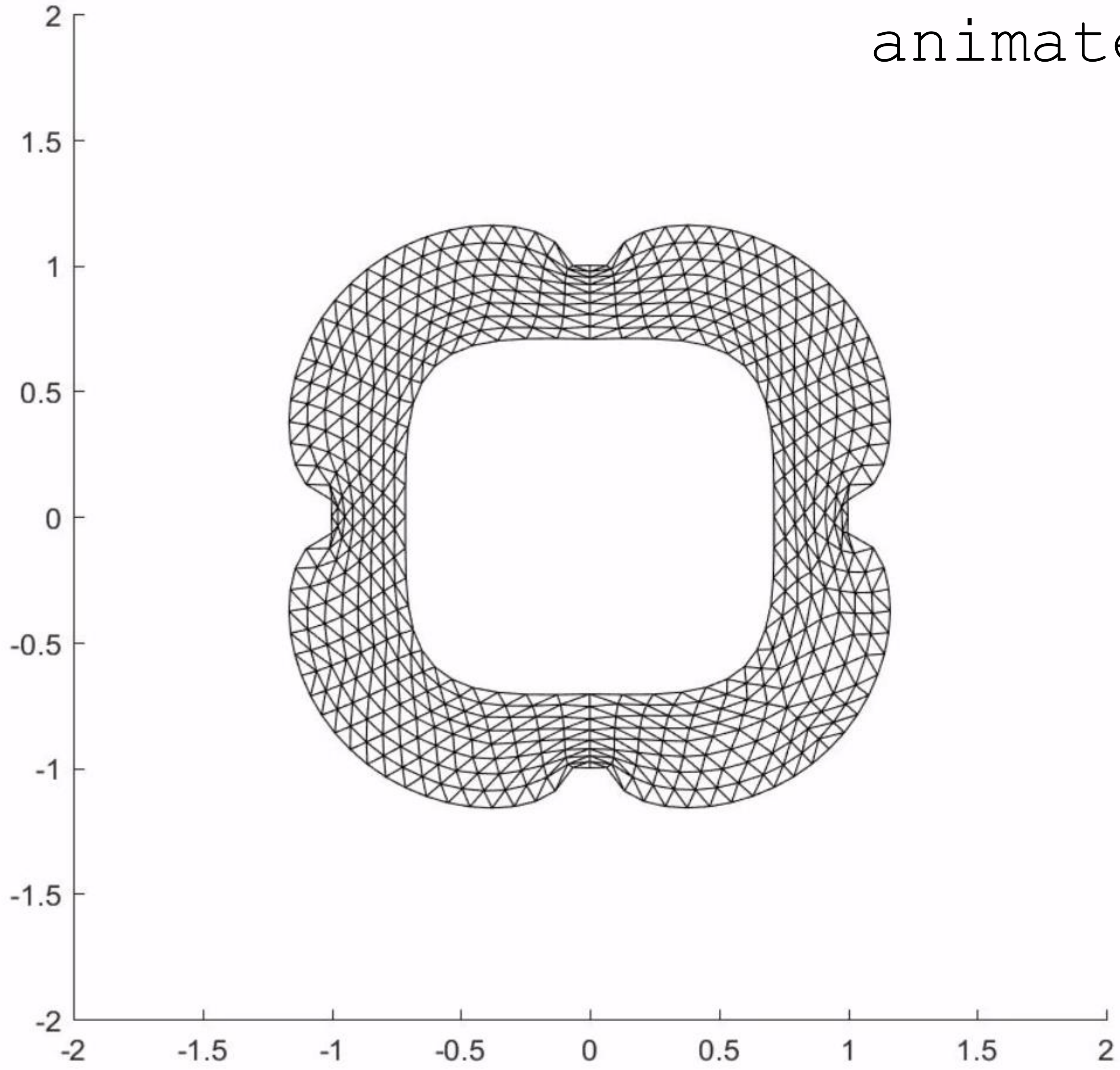


```
function [y] = f13 (x1,x2)

y = zeros(2,1);

    mag = (1/(((x1)^(2) + (x2)^(2)) ));

    y = [mag*(x1);mag*(x2)];

end
```

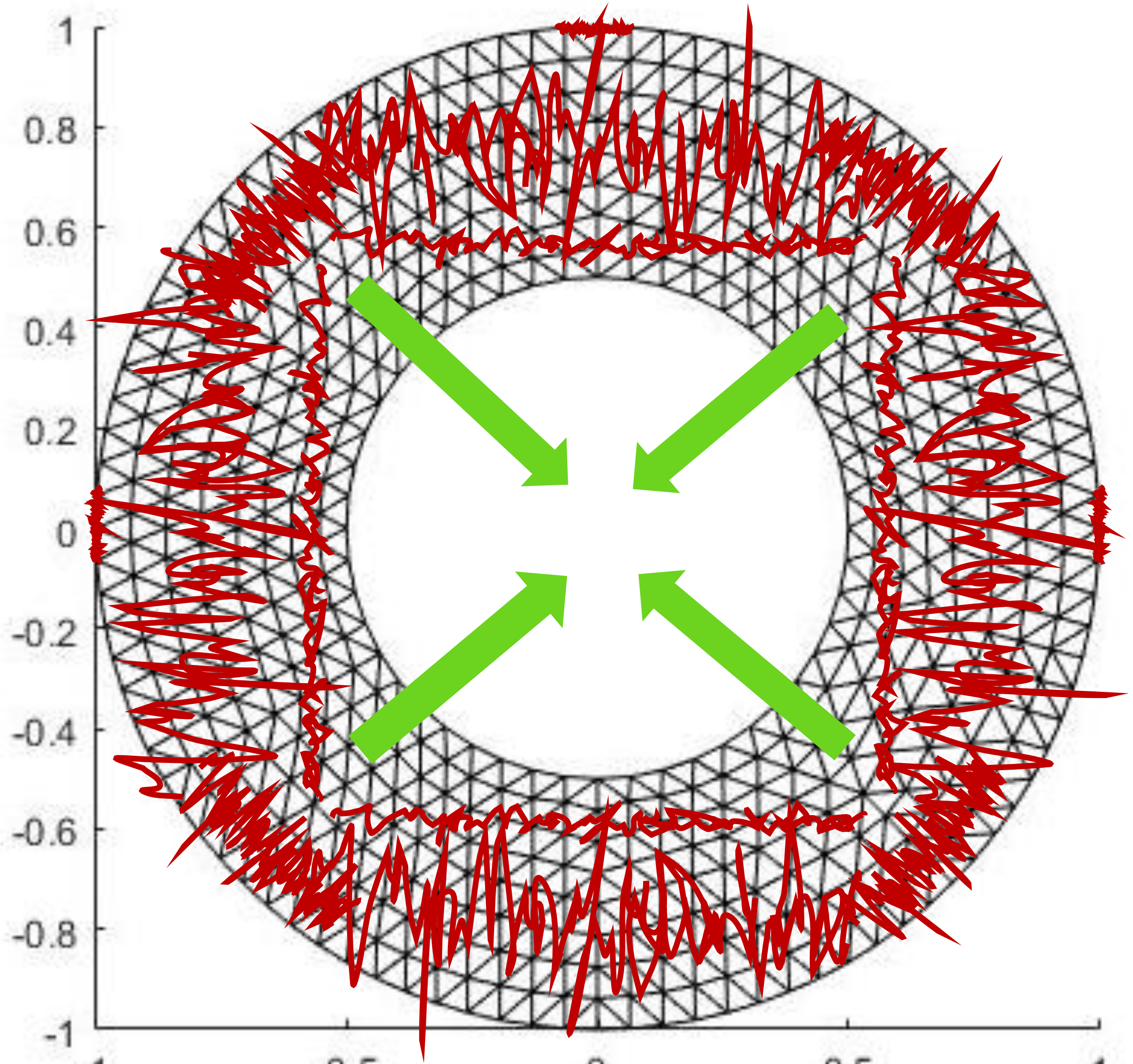



```
u=getuWithBoundary(A,F,v,.995,'or',-.995,'or',.995,'or',-.995);
```

```
animate(t,v,u,1.25,25)
```

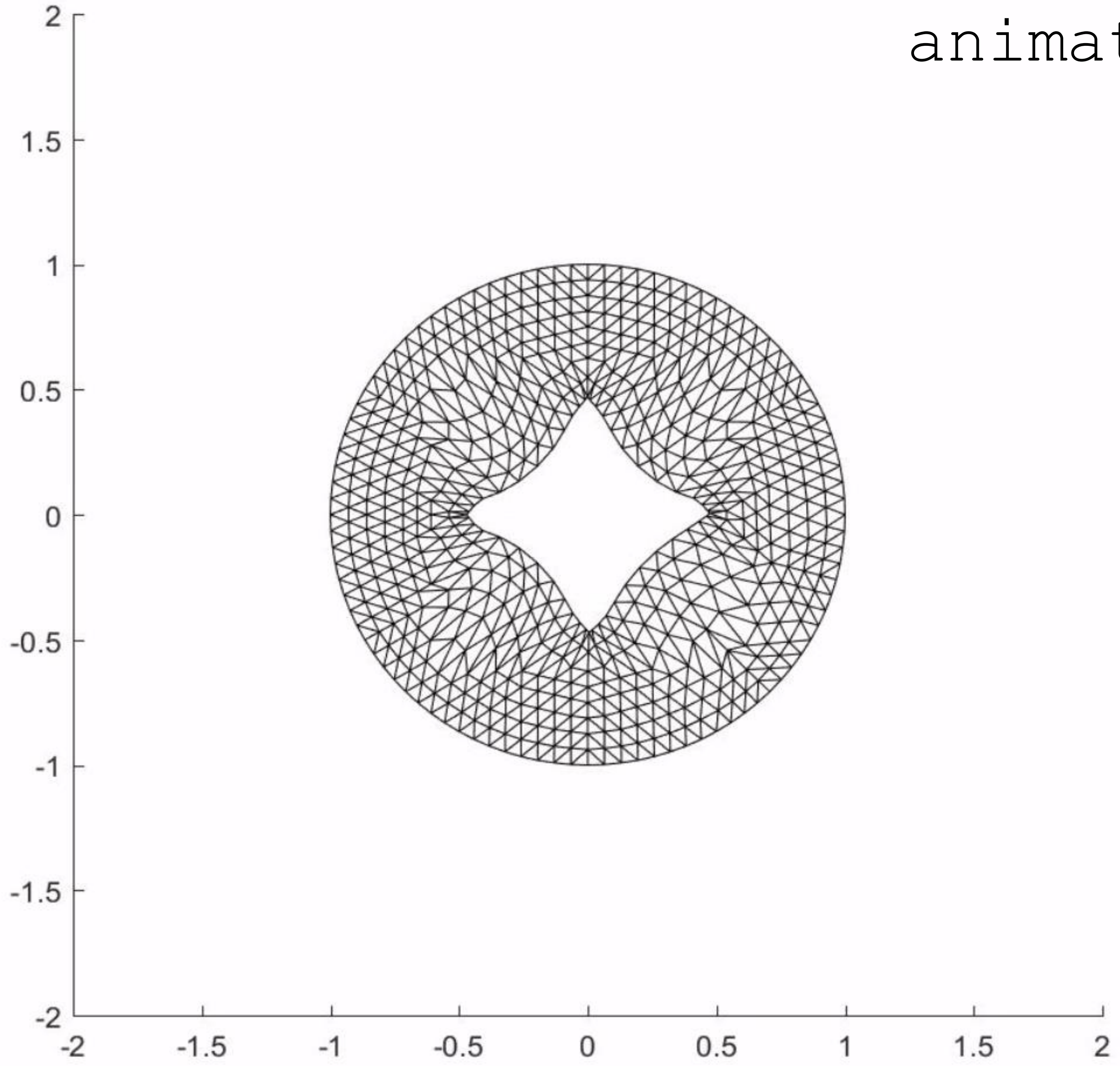


```
function [y] = f14 (x1,x2)

y = zeros(2,1);

        y(1) = -(sign(x1));
        y(2) = -(sign(x2));

end
```

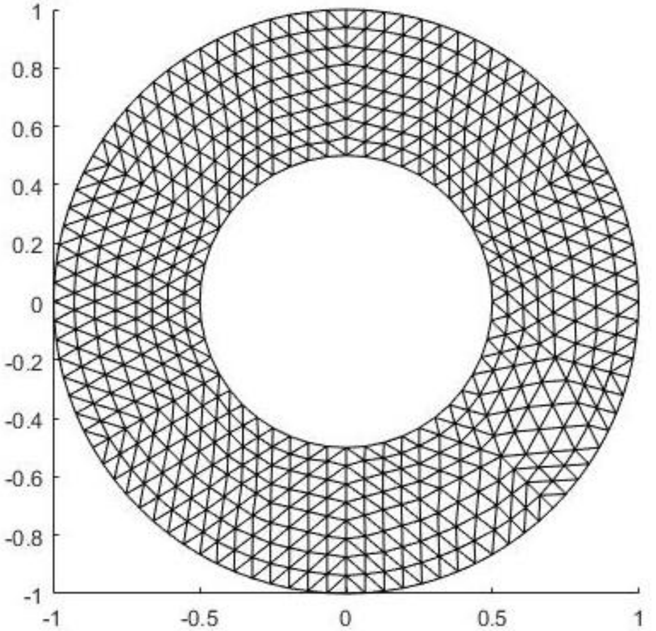


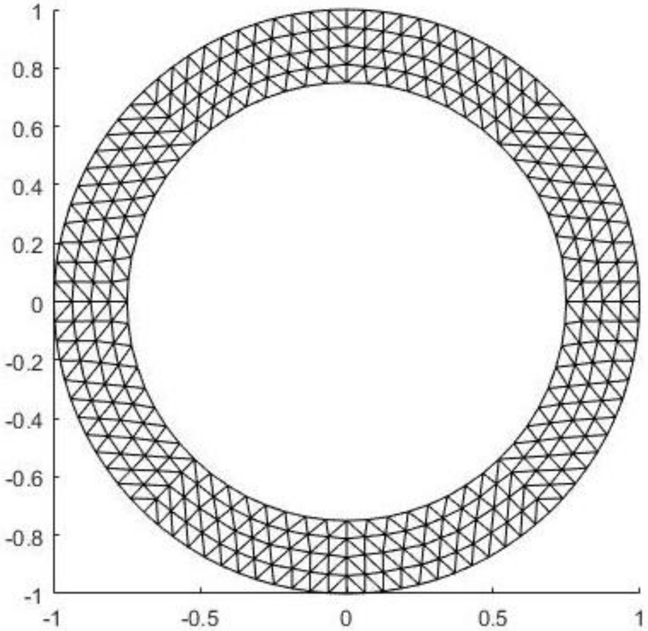

```
u=getuWithBoundary2(A,F,v,.501,'or',-.501,'or',.501,'or',-.501);
```

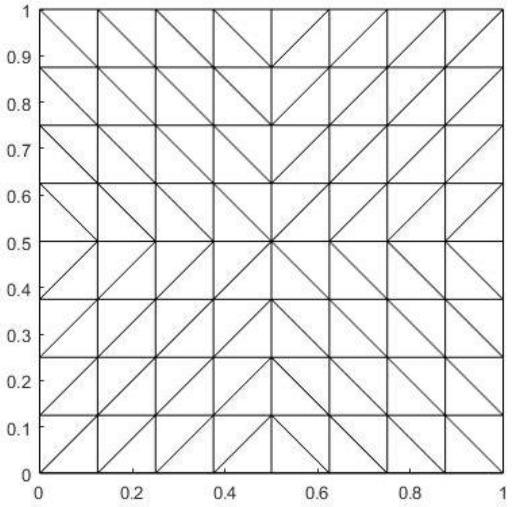
animate(t,v,u,10,140)

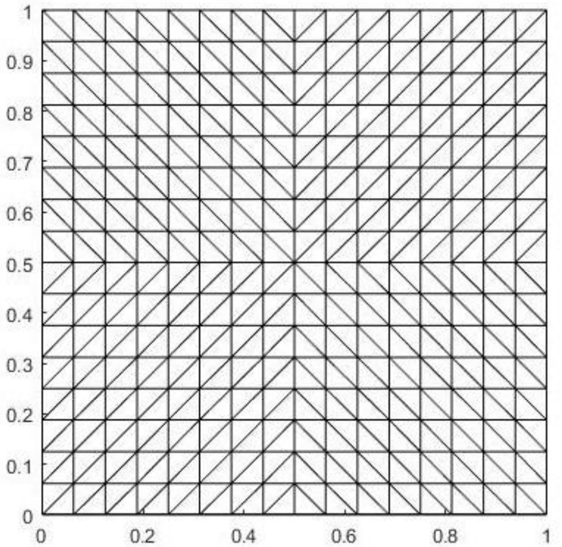


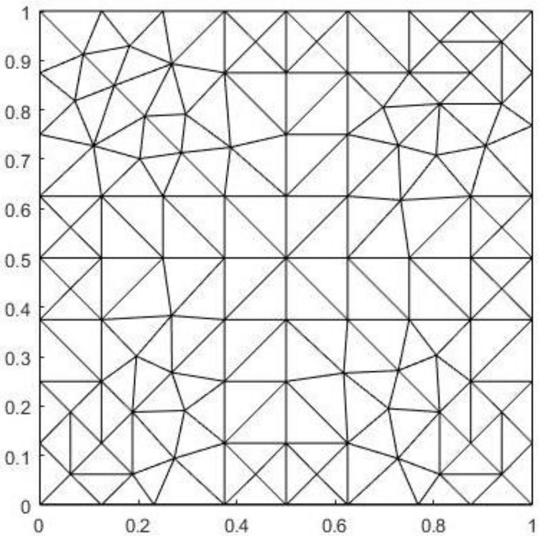
Let’s Make One Together!
Pick a Mesh

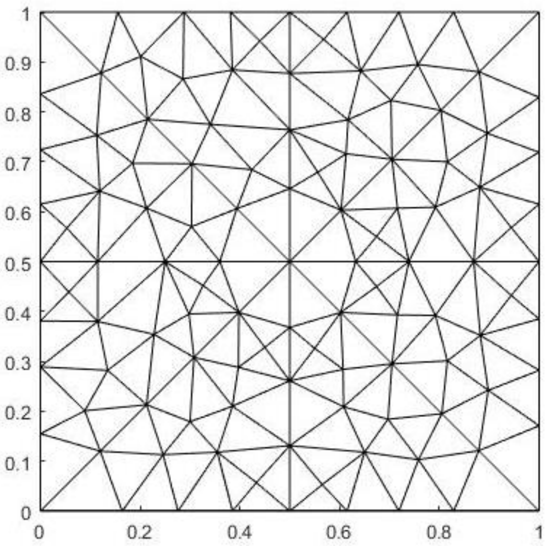


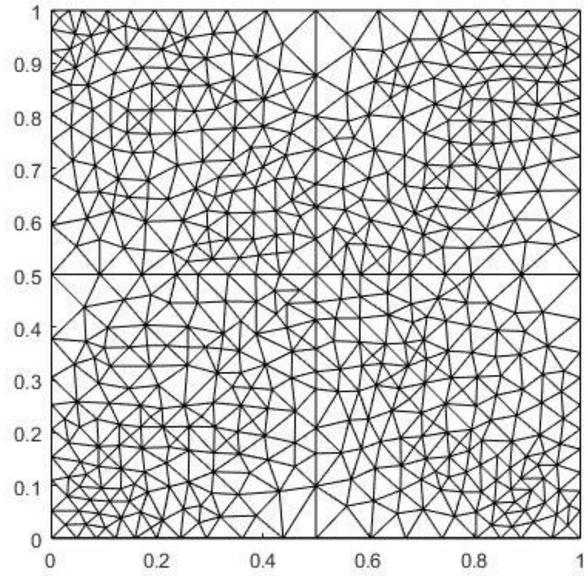




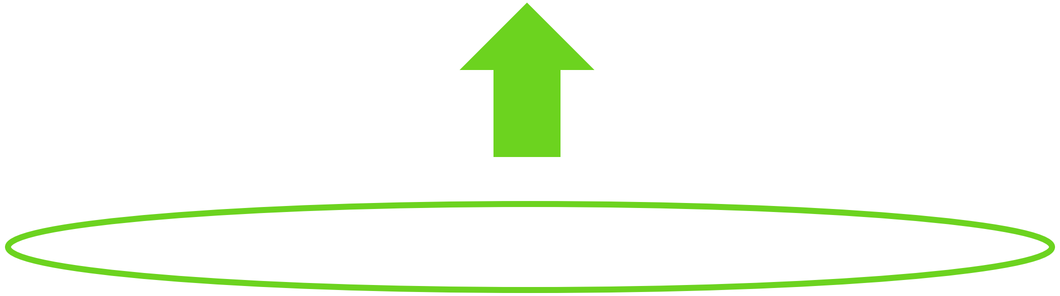


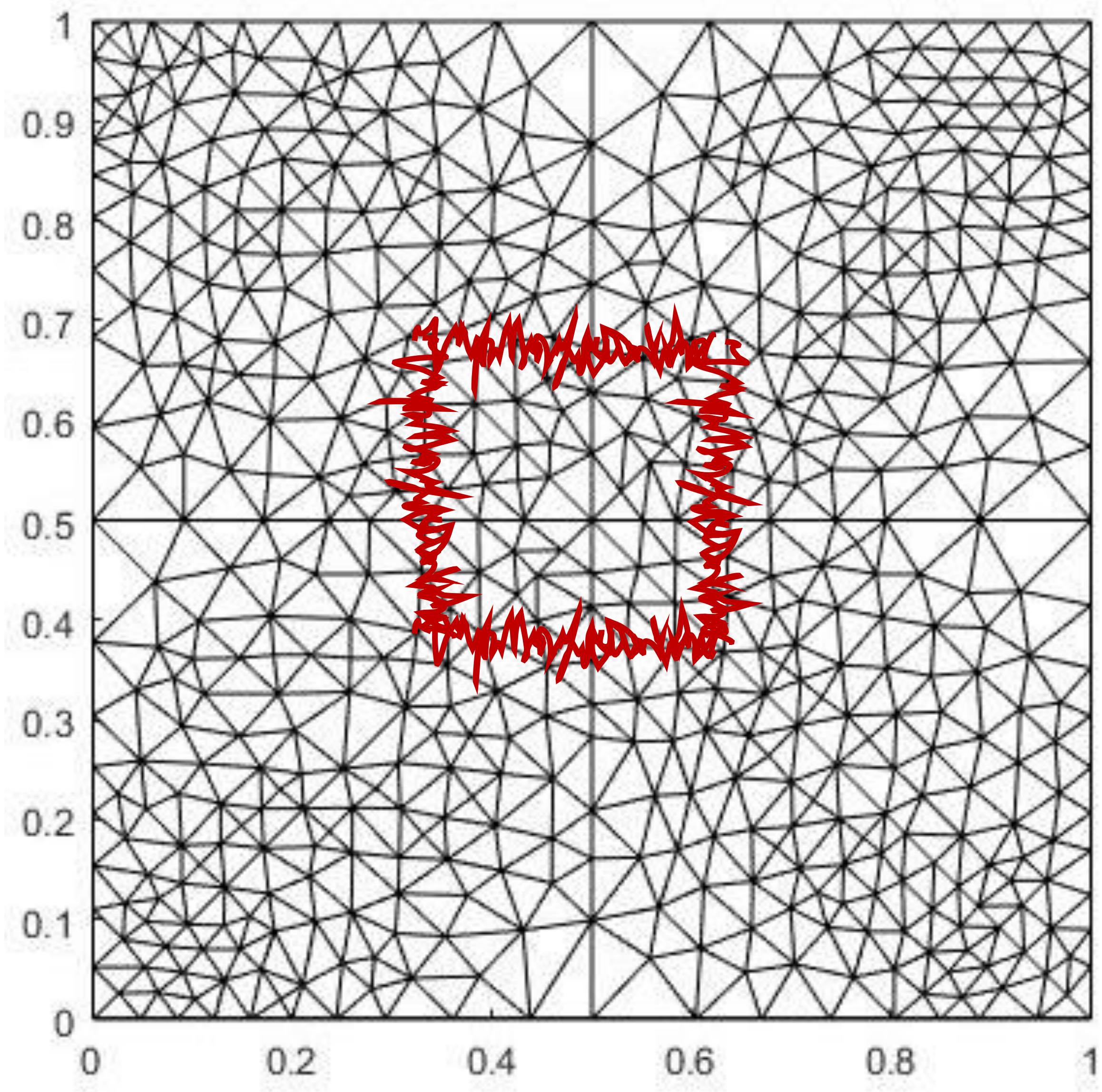






Pick a Force and Boundary






```
u=getuWithBoundary(A,F,v,.4,'and',.6,'and',.4,'and',.6);
```

Thank you for coming!

Simulating Elasticity in Two Dimensions

Shea Yonker

Host: Chris Deotte