Contents

Simulation of elasticity, biomechanics and virtual surgery

Joseph M. Teran

IAS/Park City Mathematics Series
Volume XX, XXXX

Simulation of elasticity, biomechanics and virtual surgery

Joseph M. Teran

Introduction

This short lecture series...

UCLA Mathematics Department, Box 951555, Los Angeles, CA 90095-1555
E-mail address: jteran@math.ucla.edu

LECTURE 1
Context

LECTURE 2

Numerical Solutions of the Equations of Elasticity

This lecture provides details and exercises pertaining to the implementation of numerical solvers for the equations of elasticity. We begin with a review of the Finite Element Method (FEM) as it applies to Poisson's equation. We will then apply this framework to numerically solve the equations of elasicity, first in dimension $d = 1$ with quasistatic evolution, then in dimension $d = 2$ with inertial effects included.

1. Numerical Solution of Poisson's Equation via the Finite Element Method

Let us consider numerical solutions of (variable coefficient) Poisson's equation:

$$-\nabla \cdot (\beta \nabla u) = f \quad \in \Omega \tag{2.1a}$$

$$u = p \quad \in \partial\Omega_d \tag{2.1b}$$

$$\beta \nabla u \cdot \hat{n} = q \quad \in \partial\Omega_n \tag{2.1c}$$

Here, $\Omega \subset \mathbb{R}^d$ is open; $\partial\Omega_d$ and $\partial\Omega_n$ partition the boundary of Ω and define where *Dirichlet* (d) and *Neumann* (n) boundary conditions are applied, respectively; $\hat{n}$ is the outward-pointing unit normal on $\partial\Omega_n$; $\beta, f \colon \Omega \to \mathbb{R}$, $p \colon \partial\Omega_d \to \mathbb{R}$, and $q \colon \partial\Omega_q \to \mathbb{R}$ are given, and β is bounded below by some positive number; and the goal is to solve for the unknown function $u \colon \Omega \to \mathbb{R}$. For now, we'll ignore the details of the function spaces that u, β, f, p, and q are assumed to belong in.

For the numerical solution of (2.1a), we'll use a discretization based on the Finite Element Method (FEM). Briefly, an FEM discretization focuses on the weak formulation of the partial differential equation. The derivation involves multiplying the differential equation by a test function (chosen from some suitable function space), integrating by parts, and applying boundary conditions:

$$- \nabla \cdot (\beta \nabla u) = f \tag{2.2a}$$

$$\Rightarrow \int_\Omega -\nabla \cdot (\beta \nabla u)\, v = \int_\Omega f v \tag{2.2b}$$

$$\Rightarrow \int_\Omega \beta \nabla u \cdot \nabla v = \int_\Omega f v + \int_{\partial\Omega} (\beta \nabla u \cdot \hat{n})\, v \tag{2.2c}$$

$$\Rightarrow \int_\Omega \beta \nabla u \cdot \nabla v = \int_\Omega f v + \int_{\partial\Omega_n} q v \tag{2.2d}$$

where the derivation from (2.2c) to (2.2d) is possible by stipulating that $v \equiv 0$ on $\partial\Omega_d$. Clearly, then, if u satisfies (2.1a), then u also satisfies (2.2d) for all appropriate v. Under suitable conditions, it turns out to be the case that the converse holds as well. Since (2.2d) involves only first order derivatives in u, its analysis is often preferable to (2.1a).

We then suppose that u can be approximated by a function in a finite-dimensional function space (the *(discretized) solution space*) (e.g., continuous piecewise linear functions over some tessellation of Ω), i.e., $u \approx \tilde{u} = \sum_j u_j e_j$ for basis elements $\{e_j\}$. Likewise, v is selected from some other (often related) finite-dimensional function space (the *(discretized) test space*), thus giving a system of equations for the coefficients u_j. For example, if v is taken from the same space as $\tilde{u}$, and we let v be, in turn, each of the basis functions, then we obtain

$$\sum_j \left(\int_\Omega \beta \nabla e_j \cdot \nabla e_i \right) u_j = \int_\Omega f e_i + \int_{\partial\Omega_n} q e_i \tag{2.3}$$

for all i, which may be expressed as a linear system of equations:

$$A\vec{u} = \vec{b}, \tag{2.4}$$

where

$$A_{ij} = \int_\Omega \beta \nabla e_i \cdot \nabla e_j \tag{2.5a}$$

$$\vec{u}_i = u_i \tag{2.5b}$$

$$\vec{b}_i = \int_\Omega f e_i + \int_{\partial\Omega_n} q e_i \tag{2.5c}$$

Strictly speaking, the above is only correct if v varies *precisely* throughout the same function space which $\tilde{u}$ belongs to. This is generally not the case. For example, in the presence of Dirichlet boundary conditions, $\tilde{u}$ will generally be nonzero along Ω_d, however v will be chosen from a function space which vanishes along Ω_d (hence these functions spaces will only align under homogeneous Dirichlet boundary conditions). How this manifests itself in the linear system will be investigated in the exercises.

Note that the sparsity of A is directly related to the overlap of the supports of the e_j's. Typically we choose the e_j's such that the support of one basis element overlaps the supports of only a small constant number of other basis elements, in which case A will have a number of nonzero entries proportional to the number of basis elements.

Let's now focus on dimension $d = 1$ and use *linear finite elements.* Let Ω be an interval (a, b), and let $\{x_j\}$ be grid points in (a, b), with $a = x_0 < x_1 < \cdots < x_n = b$. Often times, the x_j's will be equally spaced, which makes the analysis simpler, but it's not necessary. We'll take $\tilde{u}$ (the discretized solution) and v (the discretized test function) to be continuous and *piecewise linear* on each segment (x_{j-1}, x_j), with $v(a)$ and/or $v(b)$ vanishing if a Dirichlet condition is specified at a and/or b, respectively. This space of continuous piecewise linear functions is spanned by the *nodal basis functions* (also called "hat" functions, because of the shape of their graphs) e_j, (uniquely) defined such that

$$e_j(x_k) = \delta_{jk} = \begin{cases} 1, & j = k \\ 0, & j \neq k \end{cases} \tag{2.6}$$

1.1. Exercises

Unless otherwise noted, assume dimension $d = 1$ linear finite elements with the basis $\{e_j\}$ given by (2.6), although these results extend to higher order and higher dimensional finite elements.

(1) Here is one way to incorporate the Dirichlet boundary conditions into (2.5). For concreteness, suppose $\Omega_d = \{a\}$ (with $\Omega = (a,b)$). Then we still have $\tilde{u} = \sum_{j=0}^{n} u_j e_j$, but our space of test functions $\{v\}$ is slightly smaller, $\operatorname{span}\{e_1, \ldots, e_n\}$, to ensure that $v(a) = 0$. Thus we obtain n equations of the form (2.3), plus one more equation from the Dirichlet boundary condition ($u_0 = u(a)$). This (ostensibly) gives enough equations to solve for the $n+1$ coefficients $\{u_j\}$.

Another conceptual way to incorporate the Dirichlet boundary conditions is to consider the function $w = u - u(a)e_0$, which satisfies a similar Poisson problem as u but now with homogeneous Dirichlet boundary conditions. Thus, the space of test functions $\{v\}$ and the solution space of w coincide, and one may use eqs (2.5) directly.

In practice, one can actually just ignore the Dirichlet boundary conditions when initially computing the entries of A and $\vec{b}$ (i.e., just use (2.5a) and (2.5c)), then "correct" A and $\vec{b}$ to account for the Dirichlet boundary conditions afterwards. Describe a *simple* algorithm to effect this "correction" procedure. Try to keep A symmetric.

(2) Show that the linear system (2.4) has a unique solution if one specifies Dirichlet boundary conditions at either $x = a$ or $x = b$ or both (i.e., $u(a) = u_a$ and/or $u(b) = u_b$). For simplicity, you may assumed $\beta \equiv 1$. What happens if both boundary conditions are Neumann? Specifically address the conditions which $\vec{b}$ must satisfy, how this translates into a condition on f, and "how uniquely" $\tilde{u}$ is determined.

(3) Compute the integral in (2.5a) when $\beta \equiv 1$. If you wish, for simplicitly, you may assume the x_j's are equally spaced, i.e., $x_j = a + jh$, where $h = (b-a)/n$. What is the structure of the matrix A (e.g., describe the sparsity)?

(4) Give an expression to reasonably approximate the integrals in (2.5a) and (2.5c) for aribtrary β, f, and q (assume β and f are sufficiently "well-behaved"). You may have to consider boundary and interior grid vertices separately.

(5) Write some code (e.g., a Matlab script) to solve the following boundary value problem using linear finite elements:

$$\begin{aligned} -\left((1+x)\,u'(x)\right)' &= (1+x)^{-2} \qquad x \in (0,1) \\ u(0) &= 1 \\ (1+(1))\,u'(1) &= -1/2 \end{aligned}$$

(6) The FEM terminology for the matrix A in (2.4) is the *(global) stiffness matrix* (the meaning of the term "global" will become obvious shortly). With dimension $d = 1$ linear finite elements, it is relatively straightforward to compute the entries of the stiffness matrix, even given that one has to account for differences between boundary and interior grid vertices. However, the situation becomes more complicated with higher order or higher dimensional finite elements, and it becomes tedious to consider all the special boundary cases. As such, it is common practice to assemble the global stiffness matrix element by element (i.e., cell by cell), with each element contributing partially to several entries of the global stiffness matrix. The collection of these contributions from a single element may

be put into a small matrix, the *local stiffness matrix* for that element. The computation of the local stiffness matrix is identical to that of the global stiffness matrix upon replacing the integrations over Ω with integrations localized over the element. For example, back to dimension $d = 1$ linear finite elements, the local stiffness matrix over $I_j = (x_{j-1}, x_j)$ would consist of 2^2 (nonzero) entries (since only 2 basis functions are supported on I_j). Each of these entries would be added to accumulated global stiffness matrix entries:

$$\begin{aligned} A_{j-1,j-1} \mathrel{+}= & \int_{I_j} \beta \nabla e_{j-1} \cdot \nabla e_{j-1} \\ A_{j-1,j}, A_{j,j-1} \mathrel{+}= & \int_{I_j} \beta \nabla e_{j-1} \cdot \nabla e_j \\ A_{jj} \mathrel{+}= & \int_{I_j} \beta \nabla e_j \cdot \nabla e_j \end{aligned}$$

This suggests the following procedure to build the (global) stiffness matrix A. First, initialize the sparsity pattern of A (with zeros). Then iterate through each element and add the contributions of the element's local stiffness matrix to the corresponding entries in A.

Verify this procedure computes the same stiffness matrix as before, and modify your program from Exercise 5 to build the stiffness matrix in this fashion.

2. Neo-Hookean Elasticity with Quasistatics Evolution in Dimension 1

Let us now consider the numerical solution of the equations of elasticity with the Neo-Hookean constitutive model. To begin, for simplicity, we will consider using quasistatic evolution in dimension $d = 1$. The subsequent section will extend this into dimension $d = 2$ and add inertial effects.

2.1. Elasticity

Recall that the equations of elasticity are given by the boundary value problem

$$\rho_0(\mathbf{X}) \frac{\partial^2 \mathbf{u}}{\partial t^2} = \nabla^{\mathbf{X}} \cdot \mathbf{P} + \mathbf{f}^{\text{ext}} \quad \in \Omega(t) \tag{2.7a}$$

$$\mathbf{u}(t, \cdot) = \mathbf{g}(t, \cdot) \quad \in \partial\Omega_d(t) \tag{2.7b}$$

$$(\mathbf{P} \cdot \hat{\mathbf{n}})(t, \cdot) = \mathbf{h}(t, \cdot) \quad \in \partial\Omega_n(t) \tag{2.7c}$$

where $\rho_0 \colon \Omega_0 \to \mathbb{R}$ is the mass density (as a function of $\mathbf{X}$, the undeformed coordinates); $\mathbf{u} \colon \Omega_0 \to \Omega(t), \mathbf{X} \mapsto \boldsymbol{\phi}(\mathbf{X}) - \mathbf{X}$ is the unknown displacement; $\mathbf{P}$ is the first Piola-Kirchoff stress (which takes a specific form for Neo-Hookean, to be given later); $\mathbf{f}^{\text{ext}}$ is the given (external) force; and $\mathbf{g}$ and $\mathbf{h}$ specify the Dirichlet and Neumann boundary conditions, respectively. For simplicity, we'll assume a uniform mass density, i.e., $\rho_0 \equiv 1$.

2.2. Neo-Hookean Constitutive Model

The Neo-Hookean constitutive model relates the stress $\mathbf{P}$ to the deformation gradient $\mathbf{F} := \frac{\partial \boldsymbol{\phi}}{\partial \mathbf{X}}$ via

$$\Psi(\mathbf{F}) := \frac{\mu}{2}\left(F_{ij}F_{ji} - 2\right) - \mu \log J + \frac{\lambda}{2}\log^2 J, \tag{2.8a}$$

$$\mathbf{P}(\mathbf{F}) := \frac{\partial \Psi}{\partial \mathbf{F}} = \mu \mathbf{F} + \left(\lambda \log J - \mu\right)\mathbf{F}^{-T}. \tag{2.8b}$$

(Recall that $J := \det \mathbf{F}$.) This manifests itself in dimension $d = 1$ in terms of the displacement u as

$$P(u) = \mu\left(\frac{du}{dX} + 1\right) + \left(\lambda \log\left(\frac{du}{dX} + 1\right) - \mu\right)\frac{1}{\frac{du}{dX} + 1}.$$

2.3. Inversion-Robust Neo-Hookean

Neo-Hookean as formulated above will not be robust to element inversions, due to the $\log J$ terms. To remedy this, we replace the logarithm in (2.8a) with a cubic Taylor approximation around 1:

$$\begin{aligned} r(x) &= (x-1) - \frac{1}{2}(x-1)^2 + \frac{1}{3}(x-1)^3 \\ &= -\frac{11}{6} + 3x - \frac{3}{2}x^2 + \frac{1}{3}x^3, \end{aligned} \tag{2.9a}$$

$$\begin{aligned} r'(x) &= 1 - (x-1) + (x-1)^2 \\ &= 3 - 3x + x^2. \end{aligned} \tag{2.9b}$$

$$\begin{aligned} r''(x) &= -1 + 2(x-1) \\ &= -3 + 2x. \end{aligned} \tag{2.9c}$$

This gives

$$\begin{aligned} \Psi(\mathbf{F}) &= \frac{\mu}{2}\left(F_{ij}F_{ji} - 2\right) - \mu r(J) + \frac{\lambda}{2} r(J)^2, \\ \mathbf{P}(\mathbf{F}) &= \frac{\partial \Psi}{\partial \mathbf{F}} = \mu \mathbf{F} + \left(\lambda r(J) - \mu\right) r'(J)\frac{\partial J}{\partial \mathbf{F}}. \end{aligned}$$

Again, specializing this for $d = 1$ dimension, we obtain

$$P(u) = \mu\left(\frac{du}{dX} + 1\right) + \left(\lambda r\left(\frac{du}{dX} + 1\right) - \mu\right) r'\left(\frac{du}{dX} + 1\right). \tag{2.10}$$

2.4. Quasistatic Evolution

We begin by studying (2.7) at equilibrium, which is the basis for quasistatic evolution. This reduces the equations to

$$-\nabla^{\mathbf{X}} \cdot \mathbf{P} = \mathbf{f}^{\text{ext}}. \tag{2.11}$$

The equivalent weak formulation is

$$\int_{\Omega_0} w_{i,j} P_{ij} d\mathbf{X} = \int_{\partial \Omega_n} w_i h_i dS(\mathbf{X}) + \int_{\Omega_0} w_i f_i^{\text{ext}} d\mathbf{X}$$

for all test functions $\mathbf{w}$. Recall that summation over repeated indices is implied, and comma'ed indices indicate differentiation.

As for Poisson, we let the coordinates of $\mathbf{w}$ vary over the nodal basis functions N_i. In dimension $d = 1$, this reduces to the system of equations

$$0 = q_i(\vec{u}) := \int_a^b \frac{\partial N_i}{\partial X} P(F(u(X)))dX - b_i; \tag{2.12a}$$

$$b_i := \int_a^b N_i f^{\text{ext}} dX + [\text{Neumann boundary terms}] \tag{2.12b}$$

for each grid vertex i, where $u := u_i N_i$. The Neumann boundary terms consist of neither, one, or both of $N_i(b)h(b)$ and/or $-N_i(a)h(a)$, depending on whether b and/or a, respectively, belong to $\partial\Omega_n$.

For Neo-Hookean, P depends *non-linearly* on the displacement u, hence one must use a non-linear solver, such as Newton iteration, to solve (2.12) for $\vec{u} := (u_i)$. The Newton step looks like

$$\frac{\partial q_i}{\partial \vec{u}}(\vec{u})\,\Delta\vec{u} + q_i(\vec{u}) = 0;$$
$$\vec{u} \leftarrow \vec{u} + \Delta\vec{u}$$

where

$$\frac{\partial q_i}{\partial u_j}(\vec{u}) = \int_a^b \frac{\partial N_i}{\partial X}\frac{\partial P}{\partial F}(F(u))\frac{\partial N_j}{\partial X} dX$$

Thus, the computation of $\Delta\vec{u}$ in each Newton iteration amounts to solving a variable coefficient Poisson problem. The coefficient is $\partial P/\partial F$, which we can express via (2.10) as

$$\frac{\partial P}{\partial F} = \mu + \lambda r'\left(\frac{du}{dX}+1\right)^2 + \left(\lambda r\left(\frac{du}{dX}+1\right) - \mu\right) r''\left(\frac{du}{dX}+1\right).$$

This gives all the necessary pieces to implement a quasistatic evolution of the equations of elasticity (2.7).

2.5. Implementation

Let us assume a regular grid on (a, b), such that we have n grid vertices $x_i := a + (i-1)\Delta x$, $\Delta x := (b-a)/(n-1)$, and i ranges from 1 to n, inclusive. We break the implementation of such a solve into several steps, outlined below and expounded upon in the subsequence subsubsections.

- Compute $\vec{b} := (b_1, \ldots, b_n)$ (from (2.12b)).
- Implement a procedure to compute $\vec{q} := (q_1, \ldots, q_n)$ (from (2.12a)).
- Implement a procedure to compute $\partial q/\partial\vec{u}$.
- Solve for the Newton increment $\Delta\vec{u}$.

We provide psuedocode for all steps in a syntax similar to Matlab or Octave.

2.5.1. *Compute* $\vec{b}$. We begin by considering the computation of $\vec{b}$. This is a natural place to start because, first, it remains constant throughout the Newton iterations within a single time step; and second, its computation is identical to that for the right-hand side in Poisson's equation. One natural algorithm to compute $\vec{b}$ might be a vertex-based approach, where each b_i is computed explicitly and in isolation, and we consider the boundary grid vertices specially. However, as outlined in the last exercise from the previous section, an element-based approach scales better with dimension and order, hence we will focus on this approach for this and subsequent computations.

The implementation thus boils down to computing the integrals

$$\int_{x_i}^{x_{i+1}} N_j f^{\text{ext}} dX$$

where $j \in \{i, i+1\}$ (all other values of j integrate to zero). The result of this integral is then added to a running accumulation of b_j. We'll assume we've been given the values of f^{ext} at the midpoints of each interval (x_i, x_{i+1}), and approximate the integral via the midpoint rule, leading to the following implementation:

```
function b = eval_b(x1, xn, n, fext, h1, hn)
b = zeros([n 1]);
dX = (xn - x1) / (n - 1);
for i = 1:n-1
    b(i  ) += fext(i) * dX / 2; % fext(i) is fext evaluated at
    b(i+1) += fext(i) * dX / 2; %     x_{i+1/2}
end
b(1) += h1; % add Neumann boundary terms
b(n) += hn; % add Neumann boundary terms
```

2.5.2. *Computing $\vec{q}$.* We turn now toward computing $\vec{q}$ via (2.12a) given the current Newton approximation u. Again, we'll use an element-based approach, meaning we need to compute the integrals

$$\int_{x_i}^{x_{i+1}} \frac{\partial N_j}{\partial X} P\left(\frac{\partial u}{\partial X}\right) dX$$

where, again, $j \in \{i, i+1\}$. Note the components of the integrand are constant; indeed, on (x_i, x_{i+1}),

$$\begin{aligned} \frac{\partial N_i}{\partial X} &= -\frac{1}{\Delta X} \\ \frac{\partial N_{i+1}}{\partial X} &= \frac{1}{\Delta X} \\ P\left(\frac{\partial u}{\partial X}\right) &= P\left(\frac{u_{i+1} - u_i}{x_{i+1} - x_i}\right). \end{aligned}$$

It follows that

$$\int_{x_i}^{x_{i+1}} \frac{\partial N_j}{\partial X} P\left(\frac{\partial u}{\partial X}\right) dX = \pm P\left(\frac{u_{i+1} - u_i}{x_{i+1} - x_i}\right).$$

This leads to the following procedure to compute $\vec{q}$:

```
function q = eval_q(x1, xn, n, mu, lambda, b, u)
q = -b;
dX = (xn - x1) / (n - 1);
for i = 1:n-1
    du_dx = (u(i+1) - u(i)) / dX;
    r     = du_dx - du_dx^2 / 2 + du_dx^3 / 3;
    dr    = 1 - du_dx + du_dx^2;
    P     = mu * (du_dx + 1) + (lambda * r - mu) * dr;
    q(i  ) -= P; % (-) since dN_{i  }/dX = -1/dX
    q(i+1) += P; % (+) since dN_{i+1}/dX = +1/dX
end
```

2.5.3. *Computing $\partial q/\partial \vec{u}$.* The nex step is to consider the computation of $\partial q/\partial \vec{u}$, which will be a sparse $n \times n$ matrix. The integrals we need to compute this time are

$$\int_{x_i}^{x_{i+1}} \frac{\partial N_j}{\partial X} \frac{\partial N_k}{\partial X} \frac{\partial P}{\partial F} \left(\frac{\partial u}{\partial X} \right) dX$$

where $j, k \in \{i, i+1\}$. Again, the integrand is constant, so the integration is trivial. The expression for $\partial P/\partial F$ comes from differentiating (2.10):

$$\frac{\partial P}{\partial F} = \mu + \lambda r'(F)^2 + (\lambda r(F) - \mu) r''(F).$$

The procedure looks as follows:

```
function dq_du = eval_dq_du(x1, xn, n, mu, lambda, u)
% create an empty tri-diagonal sparse matrix
dq_du = sparse([1:n 2:n 1:n-1], [1:n 1:n-1 2:n], 0);
dX = (xn - x1) / (n - 1);
for i = 1:n-1
    du_dx = (u(i+1) - u(i)) / dX;
    r     = du_dx - du_dx^2 / 2 + du_dx^3 / 3;
    dr    = 1 - du_dx + du_dx^2;
    d2r   = -1 + 2 * du_dx;
    dP_dF = mu + lambda * dr^2 + (lambda * r - mu) * d2r;
    dq_du(i  ,i  ) += dP_dF / dX;
    dq_du(i  ,i+1) -= dP_dF / dX;
    dq_du(i+1,i  ) -= dP_dF / dX;
    dq_du(i+1,i+1) += dP_dF / dX;
end
```

2.5.4. *Computing the Newton Increment $\Delta\vec{u}$.* With the above procedures in hand, computing the Newton increment $\Delta\vec{u}$ becomes very easy. There is one thing to note regarding Dirichlet boundary conditions. When computing $\Delta\vec{u}$, we will assume that the current Newton approximation $\vec{u}$ agrees with its Dirichlet boundary conditions at the current time-step. Hence, we will want zeros in $\Delta\vec{u}$ in those entries corresponding to Dirichlet boundary conditions, which can be effected by slightly altering the matrix and right-hand side:

```
dq_du = eval_dq_du(x1, xn, n, mu, lambda, u);
q     = eval_q    (x1, xn, n, mu, lambda, b, u);
if dirichlet_bc_at_1
    dq_du(1,1) = 1;
    dq_du(1,2) = 0;
    dq_du(2,1) = 0;
    q(1) = 0;
end
if dirichlet_bc_at_n
    dq_du(n,n) = 1;
    dq_du(n,n-1) = 0;
    dq_du(n-1,n) = 0;
    q(n) = 0;
end
delta_u = dq_du \ -q;
```

2.5.5. *The Full Newton Method.* We now have all the pieces to solve the nonlinear equation (2.12):

```
function u = solve( ...
    x1, xn, n, ...
    mu, lambda, ...
    dirichlet_bc_at_1, g1, dirichlet_bc_at_n, gn, ...
    h1, hn, ... % Neumann boundary conditions
    fext, ...
    tol, u)
b = construct_b(x1, xn, n, fext, h1, hn);
if dirichlet_bc_at_1
    u(1) = g1;
end
if dirichlet_bc_at_n
    u(n) = gn;
end
do
    dq_du = eval_dq_du(x1, xn, n, mu, lambda, u);
    q     = eval_q    (x1, xn, n, mu, lambda, b, u);
    if dirichlet_bc_at_1
        dq_du(1,1) = 1;
        dq_du(1,2) = 0;
        dq_du(2,1) = 0;
        q(1) = 0;
    end
    if dirichlet_bc_at_n
        dq_du(n,n) = 1;
        dq_du(n,n-1) = 0;
        dq_du(n-1,n) = 0;
        q(n) = 0;
    end
    delta_u = dq_du \ -q;
    u += delta_u;
while max(abs(delta_u)) < tol
```

2.5.6. *Example Problem.* We can test the code with the following example problem.

- $\Omega_0 = (0,1)$ (so $a = 0$ and $b = 1$)
- Dirichlet boundary condition at $x = 0$: $u(0,t) = g(0,t) = \sin t$
- Neumann boundary condition at $x = 1$: $P(1,t) = 0$
- $f^{\text{ext}} \equiv 0$
- $E = 1000$ and $\nu = 0.3$; so $\mu = E/(2(1+\nu)) = 384.$ and $\lambda = E\nu/((1+\nu)(1-2\nu)) = 577.$

Since a quasistatics evolution has no inertial terms, we expect the solution displacement to simply be $u(X,t) = \sin t = g(0,t)$.

This is a pretty simple test, so feel free to experiment with other combinations of parameters.

3. Neo-Hookean Elasticity with Backward Euler Evolution in Dimension 2

Let us again recall the elasticity equations:

$$\rho_0 \frac{\partial^2 \mathbf{u}}{\partial t^2} = \nabla^{\mathbf{X}} \cdot \mathbf{P} + \mathbf{f}^{\text{ext}} \quad \in \Omega(t) \tag{2.14a}$$

$$\mathbf{u}(t,\cdot) = \mathbf{g}(t,\cdot) \quad \in \partial\Omega_d(t) \tag{2.14b}$$

$$\left(\mathbf{P}\cdot\hat{\mathbf{n}}\right)(t,\cdot) = \mathbf{h}(t,\cdot) \quad \in \partial\Omega_n(t) \tag{2.14c}$$

With non-negligible inertial effects, a more sophisticated temporal evolution than quasistatic evolution must be used. One such scheme is Backward Euler, which is desirable due to its unconditional stability and dissipation of oscillatory errors. We'll also see that it is not significantly more complex than quasistatics. However, the transition from dimension $d = 1$ to dimension $d = 2$ does offer a fair amount of complexity, which we shall investigate.

3.1. Backward Euler

To formulate the Backward Euler time-stepping, we introduce an auxiliary variable, $\mathbf{v}$ ("velocity"), to transform (2.14) into a first-order system (in time):

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{v};$$
$$\rho_0 \frac{\partial \mathbf{v}}{\partial t} = \nabla^{\mathbf{X}} \cdot \mathbf{P} + \mathbf{f}^{\text{ext}}.$$

The Backward Euler time discretization thus gives

$$\frac{1}{\Delta t}\left(\mathbf{u}\left(t+\Delta t,\cdot\right) - \mathbf{u}(t,\cdot)\right) = \mathbf{v}\left(t+\Delta t,\cdot\right); \tag{2.15a}$$

$$\rho_0 \frac{1}{\Delta t}\left(\mathbf{v}\left(t+\Delta t,\cdot\right) - \mathbf{v}(t,\cdot)\right) = \left(\nabla^{\mathbf{X}} \cdot \mathbf{P}\right)_{t+\Delta t} + \mathbf{f}^{\text{ext}}\left(t+\Delta t,\cdot\right). \tag{2.15b}$$

Eliminating $\mathbf{v}\left(t+\Delta t,\cdot\right)$ from (2.15) gives a (nonlinear) equation which must be solved for $\mathbf{u}\left(t+\Delta t,\cdot\right)$ at each time-step. For present notational purposes, let us refer to the unknown $\mathbf{u}\left(t+\Delta t,\cdot\right)$ as simply $\mathbf{u}$, and to the known $\mathbf{u}(t,\cdot)$ and $\mathbf{v}(t,\cdot)$ as $\mathbf{u}_0$ and $\mathbf{v}_0$, respectively. Thus, (2.15) is equivalent to

$$\rho_0 \mathbf{u} - \Delta t^2 \nabla^{\mathbf{X}} \cdot \mathbf{P} = \rho_0\left(\mathbf{u}_0 + \Delta t \mathbf{v}_0\right) + \Delta t^2 \mathbf{f}^{\text{ext}}, \tag{2.16}$$

where $\nabla^{\mathbf{X}} \cdot \mathbf{P}$ is evaluated at $t + \Delta t$ (hence depends on the uknown $\mathbf{u}$) and $\mathbf{f}^{\text{ext}}$ is also evaluated at $t + \Delta t$.

3.2. Weak Formulation and Finite Element Discretization

Let us now derive the weak formulation of equation (2.16) to obtain a finite element discretization in space. We dot product by a test function $\mathbf{w}$, integrate over Ω_0, apply integration by parts, and simplify the integrals over $\partial\Omega_0$ by stipulating that $\mathbf{w} \equiv 0$ on $\partial\Omega_d := \partial\Omega_d(t + \Delta t)$ and substituting the Neumann condition over $\partial\Omega_n := \partial\Omega_n(t + \Delta t)$. It will be notationally convenient, at this point, to once again use index notation and implied summation. We will use subscripts to denote coordinates $(1, \ldots, d)$ and (later) superscripts to denote grid vertices.

(2.17a)

$$\rho_0 u_i w_i - \Delta t^2 P_{ij,j} w_i = \left(\rho_0 \left((u_0)_i + \Delta t (v_0)_i\right) + \Delta t^2 f_i^{\text{ext}}\right) w_i$$

(2.17b)

$$\int\limits_{\Omega_0} \rho_0 u_i w_i - \Delta t^2 P_{ij,j} w_i = \int\limits_{\Omega_0} \left(\rho_0 \left((u_0)_i + \Delta t (v_0)_i\right) + \Delta t^2 f_i^{\text{ext}}\right) w_i$$

(2.17c)

$$\int\limits_{\Omega_0} \rho_0 u_i w_i + \Delta t^2 P_{ij} w_{i,j} = \int\limits_{\Omega_0} \left(\rho_0 \left((u_0)_i + \Delta t (v_0)_i\right) + \Delta t^2 f_i^{\text{ext}}\right) w_i + \Delta t^2 \int\limits_{\partial\Omega_n} h_i w_i.$$

For the finite element discretization, we'll suppose that $\Omega_0 \subset \mathbb{R}^d$ is tesselated by simplices $S^1, \ldots, S^N$ with grid vertices $\mathbf{X}^1, \ldots, \mathbf{X}^n \in \mathbb{R}^d$ and with S_i^α denoting the i^{th} grid vertex of simplex S^α (we shall let "$a \in S^\alpha$" denote the relation that grid vertex a is one of the grid vertices of simplex S^α). Our finite element space will consist of continuous $\mathbb{R}^d$-valued functions which are affine over each simplex S^α. This space is spanned by the nodal basis ("hat") functions $\{N^a\}$; N^a takes the value 1 at $\mathbf{X}^a$ and the value 0 at all other grid vertices. (Strictly speaking, of course, it's really the *projection* of this function space on each coordinate that is spanned by the nodal basis functions.)

We thus take $\mathbf{w} = (w_i) = (\delta_{ij} N^a)$ in (2.17c), where j ranges over $1, \ldots, d$ and a ranges over $1, \ldots, n$, to obtain dn equations for $\mathbf{u}$. Likewise, we discretize each coordinate of $\mathbf{u}$ as $u_i = u_i^b N^b$, giving a (nonlinear) system of equations for the coefficients u_i^b:

$$0 = q_i^a(\mathbf{u}) := \left(\int_{\Omega_0} \rho_0 N^a N^b\right) u_i^b + \Delta t^2 \left(\int_{\Omega_0} P_{ij} N^a_{,j}\right) - b_i^a; \tag{2.18a}$$

$$b_i^a := \int_{\Omega_0} \left(\rho_0 \left((u_0)_i + \Delta t (v_0)_i\right) + \Delta t^2 f_i^{\text{ext}}\right) N^a + \Delta t^2 \int_{\partial\Omega_n} h_i N^a. \tag{2.18b}$$

It is worth noting that this is very similar to the nonlinear system of equations which arise in a discretization of quasistatic evolution. The differences are only in the right-hand side $\vec{b}$ and the additional identity-like block to (2.18a).

Like in dimension $d = 1$, we shall solve (2.18) via Newton iteration:

$$\frac{\partial \vec{q}}{\partial \vec{u}}(\vec{u})\, \Delta\vec{u} + \vec{q}(\vec{u}) = 0;$$
$$\vec{u} \leftarrow \vec{u} + \Delta\vec{u}.$$

3.3. Implementation Details in Dimension 2

We will now consider specifically dimension $d = 2$ and go through the implementation details of the various computational steps necessary to advance one time step, from time t to time $t + \Delta t$.

3.3.1. *Computing $\vec{b}$.* Like in dimension $d = 1$, we will evaluate $\vec{b}$ via an element-based loop, requiring the evaluation of the integrals

$$\int_{S^\alpha} \left(\rho_0 \left((u_0)_i + \Delta t (v_0)_i\right) + \Delta t^2 f_i^{\text{ext}}\right) N^a.$$

(We'll address the Neumann terms later.) Let us suppose we are given the value of $\mathbf{f}^{\text{ext}}$ at each grid vertex $\mathbf{X}^b$ and the value of ρ_0 for each element S^α. Then we can expand each of $(u_0)_i$, $(v_0)_i$, and f_i^{ext} as $(u_0)_i^b N^b$, $(v_0)_i^b N^b$, and $f_i^{\text{ext},b} N^b$, respectively, giving

$$\begin{aligned}
&\int_{S^\alpha} \left(\rho_0 \left((u_0)_i + \Delta t (v_0)_i\right) + \Delta t^2 f_i^{\text{ext}}\right) N^a \\
&\quad = \int_{S^\alpha} \left(\rho_0^\alpha \left((u_0)_i^b N^b + \Delta t (v_0)_i^b N^b\right) + \Delta t^2 f_i^{\text{ext},b} N^b\right) N^a \\
&\quad = \int_{S^\alpha} \left(\rho_0^\alpha \left((u_0)_i^b + \Delta t (v_0)_i^b\right) + \Delta t^2 f_i^{\text{ext},b}\right) N^a N^b \\
&\quad = \left(\rho_0^\alpha \left((u_0)_i^b + \Delta t (v_0)_i^b\right) + \Delta t^2 f_i^{\text{ext},b}\right) \int_{S^\alpha} N^a N^b.
\end{aligned}$$

Note that this will be nonzero only for $a, b \in S^\alpha$. It thus suffices to evaluate $\int_{S^\alpha} N^a N^b$ for $a, b \in S^\alpha$. These may be computed by a change of coordinates to the standard $\{(0,0),(1,0),(0,1)\}$ triangle, ultimately giving

$$\int_{S^\alpha} N^a N^a = \frac{1}{6}\operatorname{area}(S^\alpha), \tag{2.20a}$$

$$\int_{S^\alpha} N^a N^b = \frac{1}{12}\operatorname{area}(S^\alpha) \quad (a \neq b) \tag{2.20b}$$

and we can compute area (S^α) via

$$\operatorname{area}(S^\alpha) = \frac{1}{2}\begin{vmatrix} \mathbf{X}^{S_1^\alpha} & \mathbf{X}^{S_2^\alpha} & \mathbf{X}^{S_3^\alpha} \\ 1 & 1 & 1 \end{vmatrix}.$$

In the implementation below, we assume area (S^α) and $\int_{\partial\Omega_n} \mathbf{h} N^a$ have been precomputed for each α and a, respectively.

```
function b = eval_b(tris, X, tri_areas, rho, fext, h, u0, v0, dt)
% tris(k,:)    = the 3 grid vertices within triangle k
% X(a,:)       = the coordinates of grid vertex a
% tri_areas(k) = the area of triangle k
% rho(k)       = the mass density within triangle k
% fext(a,:)    = the external force on grid vertex a
% h(a,:)       = the Neumann boundary condition on grid vertex a
% u0(a,:)      = the previous time-step's displacement at
%                     grid vertex a
% v0(a,:)      = the previous time step's velocity at grid vertex a
% dt           = the time step increment
b = dt^2 * h;
for k = 1:size(tris,1)
    tri = tris(k,:);
    b(tri,:) += tri_areas(k) ...
             * [2 1 1;1 2 1;1 1 2]/12 ...
             * (rho(k) * (u0(tri,:) + dt * v0(tri,:)) ...
              + dt^2 * fext(tri,:));
end
```

3.3.2. *Computing $\vec{q}$.* Computation of $\vec{q}$ requires computing the integrals

$$\begin{aligned}&\left(\int_{S^\alpha} \rho_0 N^a N^b\right) u_i^b + \Delta t^2 \left(\int_{S^\alpha} P_{ij} N^a_{,j}\right) - b_i^a \\ &= \rho_0^\alpha \left(\int_{S^\alpha} N^a N^b\right) u_i^b + \Delta t^2 \left(\int_{S^\alpha} P_{ij} N^a_{,j}\right) - b_i^a,\end{aligned}$$

where we have used the fact that ρ_0 is given element-wise. We already know the value of $\int N^a N^b$ from the previous subsubsection, so we need only address $\int P_{ij} N^a_{,j}$.

To this end, recall that inversion-robust Neo-Hookean defines $\mathbf{P}$ in terms of $\mathbf{F}$ by

$$\begin{aligned}\mathbf{P} &= \mu\mathbf{F} + (\lambda r(J) - \mu)\, r'(J) \frac{\partial J}{\partial \mathbf{F}} \\ &= \mu\mathbf{F} + (\lambda r(J) - \mu)\, r'(J) J \mathbf{F}^{-T}\end{aligned} \tag{2.21a}$$

$$r(x+1) = x - \frac{1}{2}x^2 + \frac{1}{3}x^3 \tag{2.21b}$$

$$r'(x+1) = 1 - x + x^2. \tag{2.21c}$$

We thus must compute $\mathbf{F}$ from $\mathbf{u}$. First, notice that $N^a(\mathbf{X})$ may be interpreted as the barycentric coordinate of $\mathbf{X}$ with respect to $\mathbf{X}^a$ within S^α (assuming that $a \in S^\alpha$). In other words, $N^a(\mathbf{X})$ is equal to ξ^a, where

$$\begin{pmatrix} \mathbf{X}^{S_1^\alpha} & \mathbf{X}^{S_2^\alpha} & \mathbf{X}^{S_3^\alpha} \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \xi^{S_1^\alpha} \\ \xi^{S_2^\alpha} \\ \xi^{S_3^\alpha} \end{pmatrix} = \begin{pmatrix} \mathbf{X} \\ 1 \end{pmatrix}.$$

It follows that

$$\mathbf{N}^{S^\alpha}(\mathbf{X}) = \begin{pmatrix} N^{S_1^\alpha}(\mathbf{X}) \\ N^{S_2^\alpha}(\mathbf{X}) \\ N^{S_3^\alpha}(\mathbf{X}) \end{pmatrix} = \begin{pmatrix} \mathbf{X}^{S_1^\alpha} & \mathbf{X}^{S_2^\alpha} & \mathbf{X}^{S_3^\alpha} \\ 1 & 1 & 1 \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{X} \\ 1 \end{pmatrix}$$

and hence

$$\frac{\partial \mathbf{N}^{S^\alpha}}{\partial \mathbf{X}} = \begin{pmatrix} \frac{\partial N^{S_1^\alpha}}{\partial \mathbf{X}} \\ \frac{\partial N^{S_2^\alpha}}{\partial \mathbf{X}} \\ \frac{\partial N^{S_3^\alpha}}{\partial \mathbf{X}} \end{pmatrix} = \begin{pmatrix} \mathbf{X}^{S_1^\alpha} & \mathbf{X}^{S_2^\alpha} & \mathbf{X}^{S_3^\alpha} \\ 1 & 1 & 1 \end{pmatrix}^{-1} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}.$$

With the ability to compute $\partial \mathbf{N}^{S^\alpha}/\partial \mathbf{X}$ in hand, and using the fact that $\mathbf{u} = \mathbf{u}^a N^a$, we can now compute $\mathbf{F}$:

$$\mathbf{F} = \frac{\partial \mathbf{u}}{\partial \mathbf{X}} + \mathbf{I} = \mathbf{u}^{S^\alpha} \frac{\partial \mathbf{N}^{S^\alpha}}{\partial \mathbf{X}} + \mathbf{I} \tag{2.22}$$

Using (2.22) in (2.21) will thus give $\mathbf{P}$. Finally, we can collect all $P_{ij} N^a_{,j}$ terms into a single (constant) 2×3 matrix as $\mathbf{P}\left(\partial \mathbf{N}^{S^\alpha}/\partial \mathbf{X}\right)^T$ (the rows are indexed by i, the columns by a). The implementation for computing $\vec{q}$ is then as follows.

```
function q = eval_q(tris, X, tri_areas, rho, mu, lambda, u, b, dt)
% tris(k,:)     = the 3 grid vertices within triangle k
% X(a,:)        = the coordinates of grid vertex a
% tri_areas(k) = the area of triangle k
% rho(k)        = the mass density within triangle k
% mu            = mu
% lambda        = lambda
% u(a,:)       = the vector of u_i's at grid vertex a
% b(a,:)       = the vector of b_i's at grid vertex a
% dt           = the time step increment
q = -b;
for k = 1:size(tris,1)
    tri = tris(k,:);
    dN_dX = [X(tri,:)';1 1 1] \ [1 0;0 1;0 0];
    F     = u(tri,:)' * dN_dX + [1 0;0 1];
    J     = det(F);
    dJ_dF = [F(2,2) -F(2,1);-F(1,2) F(1,1)];
    rJ    = (J-1) - (J-1)^2 / 2 + (J-1)^3 / 3;
    drJ   = 1 - (J-1) + (J-1)^2;
    P     = mu * F + (lambda * rJ - mu) * drJ * dJ_dF;
    q(tri,:) += tri_areas(k) ...
              * (rho(k) * [2 1 1;1 2 1;1 1 2]/12 * u(tri,:) ...
               + dt^2 * (dN_dX * P'));
end
```

3.3.3. *Computing $\partial\vec{q}/\partial\vec{u}$.* It is probably not too surprising that the most complicated computation is evaluating $\partial\vec{q}/\partial\vec{u}$. To begin, we use (2.18a) and differentiate with respect to u_j^b:

$$\frac{\partial q_i^a}{\partial u_j^b} = \delta_{ij}\int_{\Omega_0}\rho_0 N^a N^b + \Delta t^2 \int_{\Omega_0}\frac{\partial P_{ik}}{\partial F_{\ell m}}\frac{\partial F_{\ell m}}{\partial u_j^b}N_{,k}^a.$$

Given that $\mathbf{F} = \mathbf{u}^{S^\alpha}\left(\partial\mathbf{N}^{S^\alpha}/\partial\mathbf{X}\right) + \mathbf{I}$ (from (2.22)), one can show that $\partial F_{\ell m}/\partial u_j^b = \delta_{j\ell}N_{,m}^b$. Substituting this into the equation above yields

$$\frac{\partial q_i^a}{\partial u_j^b} = \delta_{ij}\int_{\Omega_0}\rho_0 N^a N^b + \Delta t^2\int_{\Omega_0}\frac{\partial P_{ik}}{\partial F_{jm}}N_{,k}^a N_{,m}^b. \tag{2.23}$$

We can already compute the first integral via (2.20), so the challenge remains to evaluate $\partial P_{ik}/\partial F_{jm}$. Using (2.21),

$$\begin{aligned}\frac{\partial P_{ik}}{\partial F_{jm}} &= \delta_{ij}\delta_{km}\mu \\ &\quad + \left(\lambda r'(J)^2 + (\lambda r(J) - \mu)\, r''(J)\right)\frac{\partial J}{\partial F_{ik}}\frac{\partial J}{\partial F_{jm}} \\ &\quad + (\lambda r(J) - \mu)\, r'(J)\frac{\partial^2 J}{\partial F_{ik}\partial F_{jm}}.\end{aligned}$$

This gives all the information necessary to evaluate $\partial\vec{q}/\partial\vec{u}$, though the implementation is still relatively complex from all the implicitly summed indices (as a sanity check, one sees the symmetry when interchanging $a, i \leftrightarrow b, j$). We note that $\partial\vec{q}/\partial\vec{u}$ is stored as a $2n \times 2n$ matrix. Row r within the matrix refers to grid vertex $\lceil r/2\rceil$ and coordinate $2 - r \bmod 2$. Thus, in practice, when inverting $\partial\vec{q}/\partial\vec{u}$, one needs to "flatten" the right-hand side and then "unflatten" the solution (which will turn out to be $\Delta\vec{u}$ in the Newton iteration).

```
function dq_du = eval_dq_du( ...
    tris, X, tri_areas, rho, mu, lambda, u, dt)
% tris(k,:)    = the 3 grid vertices within triangle k
% X(a,:)       = the coordinates of grid vertex a
% tri_areas(k) = the area of triangle k
% rho(k)       = the mass density within triangle k
% mu           = mu
% lambda       = lambda
% u(a,:)       = the vector of u_i's at grid vertex a
% dt           = the time step increment
n = size(X,1);
dq_du_vals = zeros([4 * 9 * size(tris,1) 1]);
dq_du_rows = dq_du_vals;
dq_du_cols = dq_du_vals;
index = 1;
for k = 1:size(tris,1)
    tri = tris(k,:);
    dN_dX = [X(tri,:)';1 1 1] \ [1 0;0 1;0 0];
    F     = u(tri,:)' * dN_dX + [1 0;0 1];
    J     = det(F);
    dJ_dF = [F(2,2) -F(2,1);-F(1,2) F(1,1)];
    rJ    = (J-1) - (J-1)^2 / 2 + (J-1)^3 / 3;
    drJ   = 1 - (J-1) + (J-1)^2;
    d2rJ  = -1 + 2 * (J-1);
    c1    = lambda * drJ^2 + (lambda * rJ - mu) * d2rJ;
    c2    = (lambda * rJ - mu) * drJ;
    for i = 1:2
        for j = 1:2
            dPik_dFjm = c1 * dJ_dF(i,:)' * dJ_dF(:,j)';
            if(i == j)
                dPik_dFjm(i,i) += mu;
            else
                dPik_dFjm(i,j) += c2;
                dPik_dFjm(j,i) -= c2;
            end
            local_dq_du = dt^2 * dN_dX * dPik_dFjm * dN_dX';
            if(i == j)
                local_dq_du += rho(k) * [2 1 1;1 2 1;1 1 2]/12;
            end
            local_dq_du *= tri_areas(k);
            for a = 1:3
                for b = 1:3
                    r = 2 * tri(a) + i - 2;
                    c = 2 * tri(b) + j - 2;
                    dq_du_vals(index) = local_dq_du(a,b);
                    dq_du_rows(index) = r;
                    dq_du_cols(index) = c;
                    ++index;
                end
            end
        end
    end
```

```
end
dq_du = sparse(dq_du_rows, dq_du_cols, dq_du_vals, 2*n, 2*n);
```

3.3.4. *Computing the Newton Increment* $\Delta\vec{u}$. As alluded to in the previous subsubsection, the stored matrix representing $\partial\vec{q}/\partial\vec{u}$ is $2n\times 2n$, requiring some "flattening" and "unflattening" (alternatively, one can use flattened vectors throughout, though the index manipulation would be somewhat more complicated). This is effected via the *reshape* function used below.

```
n = size(X,1);
dq_du = eval_dq_du(tris, X, tri_areas, rho, mu, lambda, u, dt);
q     = eval_q    (tris, X, tri_areas, rho, mu, lambda, u, b, dt);
for i = 1:length(dirichlet_vertices)
    a = dirichlet_vertices(i);
    dq_du(2*a-1,:) = 0;
    dq_du(2*a  ,:) = 0;
    dq_du(:,2*a-1) = 0;
    dq_du(:,2*a  ) = 0;
    dq_du(2*a-1,2*a-1) = 1;
    dq_du(2*a  ,2*a  ) = 1;
    q(a,:) = [0 0];
end
delta_u = reshape(dq_du \ reshape(-q', [2*n 1]), [2 n])';
```

3.3.5. *The Full Newton Method.* The entire solve procedure for a single time step looks as follows.

```
function u = solve( ...
    tris, X, tri_areas, ...
    rho, fext, mu, lambda, ...
    h, ...
    dirichlet_vertices, dirichlet_values, ...
    u0, v0, ...
    dt, ...
    tol, u)
b = eval_b(tris, X, tri_areas, rho, fext, h, u0, v0, dt);
for i = 1:length(dirichlet_vertices)
    u(dirichlet_vertices(i),:) = dirichlet_values(i);
end
n = size(X,1);
delta_u = inf([n 2]);
while max(max(abs(delta_u))) > tol
    dq_du = eval_dq_du( ...
                tris, X, tri_areas, rho, mu, lambda, u, dt);
    q     = eval_q( ...
                tris, X, tri_areas, rho, mu, lambda, u, b, dt);
    for i = 1:length(dirichlet_vertices)
        a = dirichlet_vertices(i);
        dq_du(2*a-1,:) = 0;
        dq_du(2*a  ,:) = 0;
        dq_du(:,2*a-1) = 0;
        dq_du(:,2*   ) = 0;
        dq_du(2*a-1,2*a-1) = 1;
        dq_du(2*a  ,2*a  ) = 1;
        q(a,:) = [0 0];
    end
    delta_u = reshape(dq_du \ reshape(-q', [2*n 1]), [2 n])';
    u += delta_u;
end
```

3.3.6. *Example Problem.* We can test the code with the following example problem.

- $\Omega_0 = (-1, 1)^2$
- Dirichlet boundary conditions at $\mathbf{X}_1 = \pm 1$: $u(\mathbf{X}, t) = \sin|\mathbf{X}_1|t$
- Zero Neumann boundary conditions at $\mathbf{X}_2 = \pm 1$
- $\rho_0 \equiv 1$
- $\mathbf{f}^{\text{ext}} = \mathbf{0}$
- $E = 1000$ and $\nu = 0.3$; so $\mu = E/(2(1+\nu)) = 384.$ and $\lambda = E\nu/((1+\nu)(1-2\nu)) = 577.$

LECTURE 3

Techniques to Deal with Element Inversion

Bibliography