NTNU

Linear Elastostatic Problems solved with Non-Local and Local Finite Element Methods using Linear Elements

by

Daniel Osen

A thesis submitted in partial fulfillment for the
Bachelor's degree in Applied Mathematics

in the

Department of Mathematical Sciences
Norwegian University of Science and Technology

October 2016

Declaration of Authorship

I, Daniel Osen, declare that this thesis titled, 'Linear Elastostatic Problems solved with Non-Local and Local Finite Element Methods using Linear Elements' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

NTNU

Abstract

Department of Mathematical Sciences
Norwegian University of Science and Technology

Bachelor's degree in Applied Mathematics

by Daniel Osen

In this thesis, three finite element methods are developed and implemented in MATLAB and applied to 3D linear elastostatic problems.

Acknowledgements

I would like to thank my family, and friends, for supporting me and believing in me. I would also like to thank my project advisor, prof. Anton Evgrafov at the Department of Mathematical Sciences NTNU, for helping me choose a project, aiding me with various problems, and providing me with useful examples and literature, and finally letting me write this project from home.

Contents

List of Figures

For my friends and family...

Chapter 1

The local linear elastostatic model

1.1 Mathematical formulation

Let $\Omega \subset \mathbb{R}^3$ be the interior of an elastic solid at equilibrium. Assume that Ω is bounded, simply connected, with a continuous boundary $\partial\Omega$. Then, in direct tensor form, the local linear elastostatic PDE reads:

$$\boldsymbol{\nabla} \cdot \boldsymbol{\sigma} = -\boldsymbol{f} \quad \text{in } \Omega, \tag{1.1}$$

$$\boldsymbol{\sigma} = \mathsf{C} : \boldsymbol{\varepsilon} \quad \text{in } \Omega, \tag{1.2}$$

$$\boldsymbol{\varepsilon} = \frac{1}{2}[\boldsymbol{\nabla u} + (\boldsymbol{\nabla u})^{\mathbf{T}}] \quad \text{in } \Omega, \tag{1.3}$$

where $\boldsymbol{\sigma}$ is the second-order stress tensor, $\boldsymbol{f}$ is the bodily volume forces, C is the fourth-order stiffness tensor, $\boldsymbol{\varepsilon}$ is the second-order strain tensor, and $\boldsymbol{u}$ is the displacement vector. The Dirichlet and Neumann boundary conditions are:

$$\boldsymbol{u} = \boldsymbol{g_D} \quad \text{on } \partial\Omega_D, \tag{1.4}$$

$$\boldsymbol{\sigma} \cdot \boldsymbol{n} = \boldsymbol{g_{Neu}} \quad \text{on } \partial\Omega_{Neu}, \tag{1.5}$$

where $\boldsymbol{n}$ is the outward normal vector, $\boldsymbol{g_D}$, $\boldsymbol{g_{Neu}}$ some vector functions specifying boundary displacement and traction forces, and $\partial\Omega_{Neu}, \partial\Omega_D$ the Dirichlet and Neumann boundaries with $\partial\Omega_{Neu} \cap \partial\Omega_D = \emptyset$.

Due to the symmetry of stress and strain, and to ensure the existence of a unique strain energy potential[6, p. 11-12], the number of independent entries in the stiffness tensor is reduced to 21 since:

$$C_{ijkl} = C_{ijlk} = C_{jilk} = C_{jikl} = C_{klij}.$$

1.2 Istropic media

If the solid is isotropic, the stress tensor entries may be written as:

$$\mathsf{C}_{ijkl} = \lambda\delta_{ij}\delta_{kl} + \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}),$$

where δ_{ij} is the Kronecker-delta function, and λ, μ are called the Lamé constants, which in principle are both positive real numbers[1]. Otherwise, the solid may be anisotropic, orthotropic or transversely isotropic to name a few.

[1] Lamé parameters, Wikipedia, https://en.wikipedia.org/wiki/Lam%C3%A9_parameters

Chapter 2

Introduction to the finite element method

"The finite element method provides a formalism for generating discrete (finite) algorithms for approximating the solutions of differential equations. It should be thought of as a black box into which one puts the differential equation (boundary value problem) and out of which pops an algorithm for approximating the corresponding solutions."
-Susanne C. Brenner, L. Ridgeway, Scott, [1, p. 1]

The finite element method originated from the need to solve complex elasticity and structural analysis problems[1]. It can be divided into three steps:

1. pose the boundary value problem as a weak formulation,
2. discretize the resulting system of equations,
3. implement algorithm and solve numerically.

The method is based on the calculus of variations, that is, one is interested in minimizing an energy functional, which is done numerically through mesh discretization on a continuous domain by dividing a large problem into smaller parts called finite elements. It has seen much use since its conception, and is often formulated as minimization problems[3, p. 1]. What makes the finite element method so outstanding, is the ease at which numerical methods are developed using Ritz-Galerkin approximation[1, p. 3], and is perhaps why it has been referred to as a black box.

[1]Finite Element Method, Wikipedia: https://en.wikipedia.org/wiki/Finite_element_method#History

2.1 The finite element

The definition of a finite element[1, p. 69] is as follows:

Let $\mathcal{K} \subset \mathbb{R}^n$ be a bounded closed set with nonempty interior and piecewise smooth boundary
(the element domain).
Let $\mathcal{P}$ be a finite-dimensional space of functions on K
(the space of shape functions).
Let $\mathcal{N} = \{N_1, N_2, \ldots, N_k\}$ be a basis for the dual of $\mathcal{P}$
(the set of nodal variables).
Then $(\mathcal{K}, \mathcal{P}, \mathcal{N})$ is called a finite element.

For the purposes of developing finite algorithms to the linear elastostatic model, we will be using the spaces: $\mathcal{K} \subset \Omega$, where $\Omega \subset \mathbb{R}^3$ closed and bounded, the solid's continuum, with the piecewise smooth boundary $\partial\Omega$, $\mathcal{P}$ as the set of real linear polynomials on $\mathcal{K}$, e.g $\mathcal{P} = \mathbb{P}_1(\mathcal{K})$, and $\mathcal{N} = \{N_i \mid N_i(p_j) = \delta_{ij}\ \}$ where $p_j \in \mathcal{P}$ is the jth basis function of $\mathcal{P}$, and δ_{ij} is the Kronecker-delta.

2.2 Triangulation

In constructing the finite elements, which we use to compute the numerical solution, we need to choose some geometry to use in the subdivision of the domain Ω. As we are working in $\mathbb{R}^3$, it is common to divide the domain into either tetrahedrons, prisms, or hexahedrons. We will be using tetrahedral elements. We denote the discretized domain of the solid as Ω_h, writing[4, p. 80]:

$$\Omega_h = \text{clos}\left(\bigcup_{K \in \mathcal{T}_h} K\right),$$

where clos is taken to mean the closure, K are the tetrahedral element domains, and $\mathcal{T}_h$ is the tetrahedral mesh covering Ω, or subdivision, defined as a finite collection of disjoint element domains K with their union equal to the closure of the domain, and h is a measure of the size of the largest element, usually $h = \max_{K \in \mathcal{T}_h} h_k$, where $h_k = \max_{(\boldsymbol{x},\boldsymbol{y}) \in K} ||\boldsymbol{x} - \boldsymbol{y}||$: the greatest euclidean distance between any two vertices in an element.

We note that if the solid's continuum is not a polygon, then we may have that $\Omega_h \neq \Omega$. That is, the computational domain is only an approximation at the boundary. This may

present some difficulties, such as an oscillating error, but we are hoping that the grid induces an approximation in the sense that: $lim_{h\to 0} m(\Omega \setminus \Omega_h) = 0$, where m is some measure such that this is well-defined[4, p. 81]. There are known ways to overcome this, such as when working in $\mathbb{R}^2$, we may employ the use of "triangles" with at most one curved side at the boundary[1, p. 271], or we may use isoparametric finite elements, such as the quadratic tetrahedron in $\mathbb{R}^3$. Overall, it would appear this difficulty is often overlooked by textbooks[5, abstract]. However, we shall state the following:

Suppose we have a weak formulation of the linear elastostatic boundary problem (1.1) such that $V_h, V \subset H$, where H is a Hilbert space, V and V_h are linear subspaces, in fact $\boldsymbol{u} \in V$ and $\boldsymbol{u_h} \in V_h$, where $\boldsymbol{u}$ is the analytical solution and $\boldsymbol{u_h}$ is the numerical solution, and V_h is our closed linear subspace of piecewise linear polynomials, so the weak formulation, and the accompanying Galerkin problem reads:

Find $\boldsymbol{u} \in V,\ \boldsymbol{u}_h \in V_h$ such that:

$$\begin{aligned} a(\boldsymbol{u}, \boldsymbol{v}) &= F(v) \quad \forall \boldsymbol{v} \in V, \\ a(\boldsymbol{u_h}, \boldsymbol{v}) &= F(v) \quad \forall \boldsymbol{v} \in V_h, \end{aligned}$$

where $F \in H'$, and $a(\cdot, \cdot)$ is a symmetric, bounded and coercive bilinear form on H. Then, by lemma 10.1.7[1, p. 272],

$$||\boldsymbol{u} - \boldsymbol{u_h}||_H \leq (1 + \frac{C}{\gamma}) \inf_{v \in V_h} ||\boldsymbol{u} - \boldsymbol{v}||_H + \frac{1}{\gamma} \sup_{\boldsymbol{w} \in V_h \setminus \{0\}} \frac{|a(\boldsymbol{u} - \boldsymbol{u_h}, \boldsymbol{w})|}{||\boldsymbol{w}||_H},$$

where C and γ are the continuity and coercitivity constants respectively. Moreoever if $V_h \subset V$, that is, our linear elements satisfy the boundary conditions exactly, which should be the case for $\Omega_h = \Omega$, where Ω is a sufficiently regular polygon, then the last term on the RHS vanishes. Hence, if we can approximate the domain well enough, we obtain a decent bound on the discretization error. For our numerical examples, we will be using two different domains Ω: the unit sphere, and the unit cube. We also show the mesh for polygonal L-shape.

The generation of Ω_h with tetrahedral elements is done using Delaunay triangulation[2], with MATLAB code written for the unit sphere by Kjetil A. Johannessen, in september of 2012, for the programming project at The Norwegian University of Science and Technology (NTNU) in the applied mathematics course TMA4220. The generation for the unit cube, and the polygonal L-shape was written in MATLAB using delaunay triangulation as well, inspired by Kjetil A. Johannessen's code for the sphere. The generating

[2]Delaunay Triangulation, Wikipedia, https://en.wikipedia.org/wiki/Delaunay_triangulation

code is restricted to uniform mesh refinement, with examples shown in figure 2.1, 2.2, and 2.3.

FIGURE 2.1: 9-Point Cube mesh generated in MATLAB.

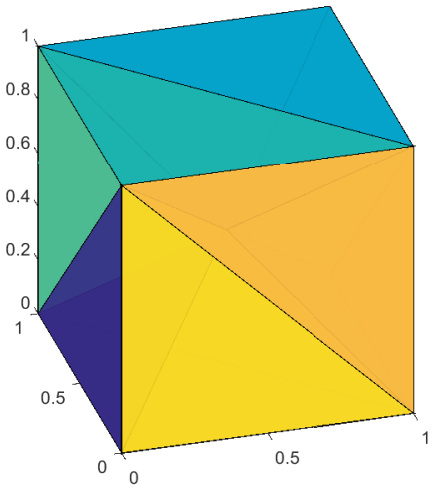


FIGURE 2.2: Uniform L-shape mesh generated in MATLAB.

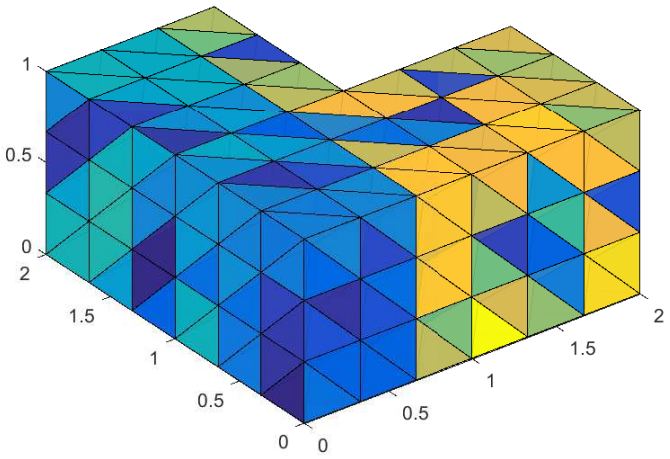
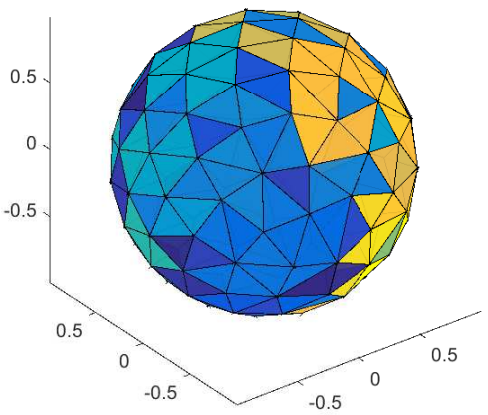


FIGURE 2.3: Sphere mesh generated in MATLAB. Author: Kjetil A. Johannessen.



Chapter 3

Application: Local Isotropic Solids

Suppose an elastic solid is both isotropic and at equilibrium. If the displacement is zero at the boundary, the solid is said to be clamped. This simplifies the problem greatly, and taking advantage of the minor and major symmetries (stress, strain, and arbitraryness of the derivation order on the energy potential), we write the entries of the stress tensor:

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} \end{bmatrix},$$

and because of symmetries and isotropy the unknown stress entries are explicitly given:

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix} = \begin{bmatrix} \lambda+2\mu & \lambda & \lambda & 0 & 0 & 0 \\ \lambda & \lambda+2\mu & \lambda & 0 & 0 & 0 \\ \lambda & \lambda & \lambda+2\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu \end{bmatrix} \begin{bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ 2\epsilon_{23} \\ 2\epsilon_{13} \\ 2\epsilon_{12} \end{bmatrix},$$

where $\epsilon_{kl}(\boldsymbol{u}) = \frac{1}{2}(\frac{\partial u_l}{\partial x_k} + \frac{\partial u_k}{\partial x_l})$ are the entries of the second order strain tensor[1].

[1] Introduction to Elasiticity, Constitutive Relations, Wikiversity, https://en.wikiversity.org/wiki/Introduction_to_Elasticity/Constitutive_relations

3.1 Weak formulation

If we seperate equation (1.1) into its i rows, applying the nabla operator row-wise, we can multiply with the i'th component of a test function $\boldsymbol{v}$, integrate by parts, and then sum the three equations to obtain:

$$\int_\Omega \left(\mu \sum_{i=1}^{3} \nabla u_i \cdot \nabla v_i + (\mu + \lambda)(\nabla \cdot \boldsymbol{u})(\nabla \cdot \boldsymbol{v}) + \mu(\sum_{i \neq j}^{3} \frac{\partial u_i}{\partial x_j} \frac{\partial v_j}{\partial x_i}) \right) \mathrm{d}\Omega = \int_\Omega \sum_{i=1}^{3} f_i v_i \ \mathrm{d}\Omega. \tag{3.1}$$

Then the LHS is a symmetric, bilinear form $a(\boldsymbol{u}, \boldsymbol{v})$. Denoting the RHS as $f(\boldsymbol{v})$, and letting $\boldsymbol{u}, \boldsymbol{v} \in V$, where $V = (H_0^1(\Omega))^3$ is a hilbert space, and f be a bounded linear functional on V, we have a weak formulation, which upon choosing a basis for V, is translated to a Galerkin problem.

We now wish to show that this problem of the clamped isotropic solid is well-posed. To do this, we invoke the Lax-Milligram theorem[2], and thus require only to show boundedness and elipticity of the bilinear form.

We first show boundedness

$$\begin{aligned}
a(\boldsymbol{u}, \boldsymbol{v}) &\leq |a(\boldsymbol{u}, \boldsymbol{v})| \\
&= \left| \mu \sum_i \sum_j < \frac{\partial u_i}{\partial x_j}, \frac{\partial v_i}{\partial x_j} > + (\lambda + \mu) \sum_i \sum_j < \frac{\partial u_i}{\partial x_i}, \frac{\partial v_j}{\partial x_j} > + \mu \sum_{i \neq j} < \frac{\partial u_i}{\partial x_j}, \frac{\partial v_j}{\partial x_i} > \right| \\
&\leq |\mu| \sum_i \sum_j || \frac{\partial u_i}{\partial x_j} || \ || \frac{\partial v_i}{\partial x_j} || + |\lambda + \mu| \sum_i \sum_j || \frac{\partial u_i}{\partial x_i} || \ || \frac{\partial v_j}{\partial x_j} || + |\mu| \sum_{i \neq j} || \frac{\partial u_i}{\partial x_j} || \ || \frac{\partial v_j}{\partial x_i} || \\
&\leq |\mu| \sum_i \sum_j ||\boldsymbol{u}||_V \ ||\boldsymbol{v}||_V + |\lambda + \mu| \sum_i \sum_j ||\boldsymbol{u}||_V \ ||\boldsymbol{v}||_V + |\mu| \sum_{i \neq j} ||\boldsymbol{u}||_V \ ||\boldsymbol{v}||_V \\
&\leq C(|\mu|, |\lambda|) ||\boldsymbol{u}||_V \ ||\boldsymbol{v}||_V,
\end{aligned}$$

where $C(|\mu|, |\lambda|)$ is some positive constant depending on μ and λ. Next we show ellipticity:

$$||u||_V^2 \leq C' \sum_i ||\nabla u_i||^2 + \frac{1}{4} \int_\Omega \sum_{ij} |\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}|^2 \mathrm{d}\Omega \leq C''(\lambda, \mu) a(u, u),$$

[2]Lax-Milligram Wikipedia, https://en.wikipedia.org/wiki/Babu%C5%A1ka%E2%80%93Lax%E2%80%93Milgram_theorem

where we have used Korn's inequality and Poincaré's inequality, in addition to Cauchy-Schwarz Inequality[3], and assume that $\lambda, \mu > 0$, with C', C'' being positive constants.

3.2 Numerical solution

We wish to find a numerical solution $\boldsymbol{u_h}$ on tetrahedral elements using the FEM method with linear shape functions. For each sum and/or term beneath the integral sign on the LHS in equation (3.1) we construct the matrices $\boldsymbol{M_1}$, $\boldsymbol{M_2}$, $\boldsymbol{M_3}$, and on the RHS the vector $\boldsymbol{b}$ so that we obtain the linear system:

$$(\mu \boldsymbol{M_1} + (\mu + \lambda)\boldsymbol{M_2} + \mu \boldsymbol{M_3})\, \boldsymbol{u_h} = \boldsymbol{b},$$

where $\boldsymbol{u_h} = \{u_i\}_{i=1}^{n}$ the solutions at row i of nodes $\boldsymbol{x_j}$. This is an equation of dimension $n = 3m$ where m is the number of nodes, though with the dirichlet condition the solution at the edge faces is known.

As an example of the clamped isotropic problem, consider the numerical solution on the sphere centered at the origin with radius 1, and the homogeneous dirichlet boundary $\boldsymbol{u} = 0$. Let the bulk modulus and the young's modulus of the material be $K = 10^9$ Pa and $E = 10^5$ Pa respectively, which is in the range of rubber[4], giving $\lambda \approx 10^9$ Pa and $\mu \approx 3.33 \cdot 10^4$ Pa. Finally, let the bodily volume forces be $f_1 = 2\lambda + 8\mu$ and $f_2 = f_3 = 0$. The analytical solution is then $u_1 = 1 - r^2$ and $u_2 = u_3 = 0$ on the unit sphere, where r is the euclidean three-dimensional distance from the center. In figure 3.1 part of the numerical solution is displayed. A log-log error plot is shown for $\lambda = 1$ and $\mu = 1$ in figure 5.7, in addition to a plot of the computation time in figure 3.2. The error plot is virtually the same as that for the iterative method solving the local case. Consider also the clamped isotropic cube, using the previous Lamé constants, and where the components of u are the same. For the cube case, that is, $x_1, x_2, x_3 \in [0, 1]$, we have a possible solution $u_1 = u_2 = u_3 = (x_1^2 - x_1)(x_2^2 - x_2)(x_3^2 - x_3)$. The numerical solution is shown in figure 3.3.

.

[3] Korn's inequality, Poincaré's inequality and Cauchy-Schwarz inequality, Wikipedia, https://en.wikipedia.org/wiki/Korn%27s_inequality, https://en.wikipedia.org/wiki/Poincar%C3%A9_inequality, https://en.wikipedia.org/wiki/Cauchy%E2%80%93Schwarz_inequality

[4] Bulk Modulus of Rubber Abstract, Polymer Volume 35, Issue 13, June 1994, Pages 2759-2763, http://www.sciencedirect.com/science/article/pii/0032386194903042

FIGURE 3.1: Paraview: The displacement in x-direction on the unit sphere with number of nodes $N = 1400$.

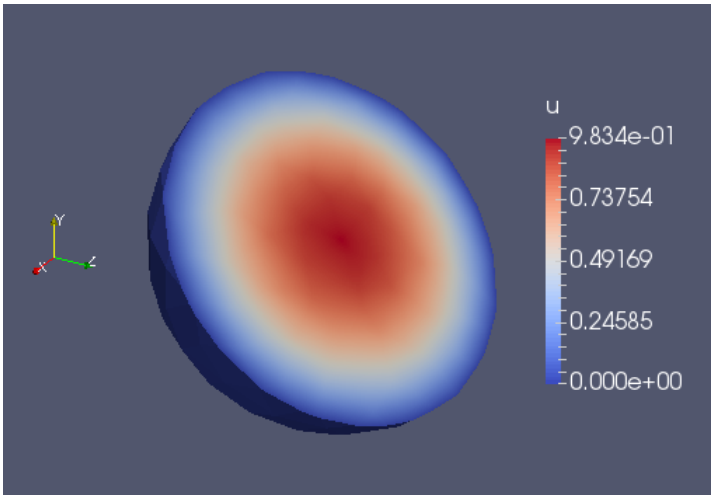


FIGURE 3.2: Increase in computation time as calculated by MATLAB for the isotropic clamped local sphere, y-axis: time in seconds, x-axis: N.

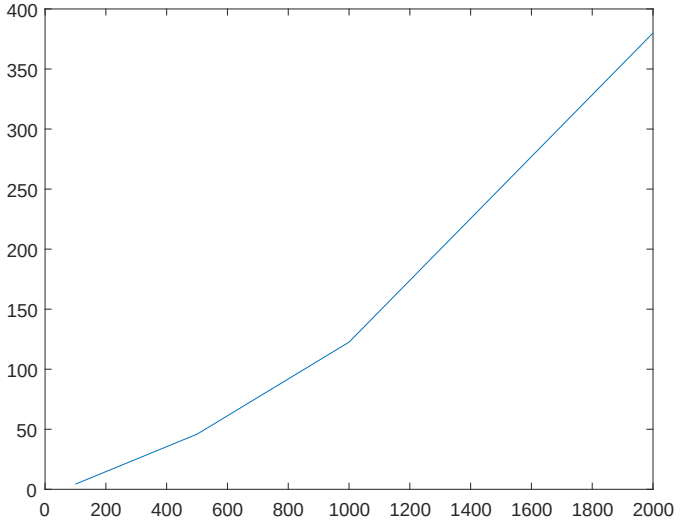
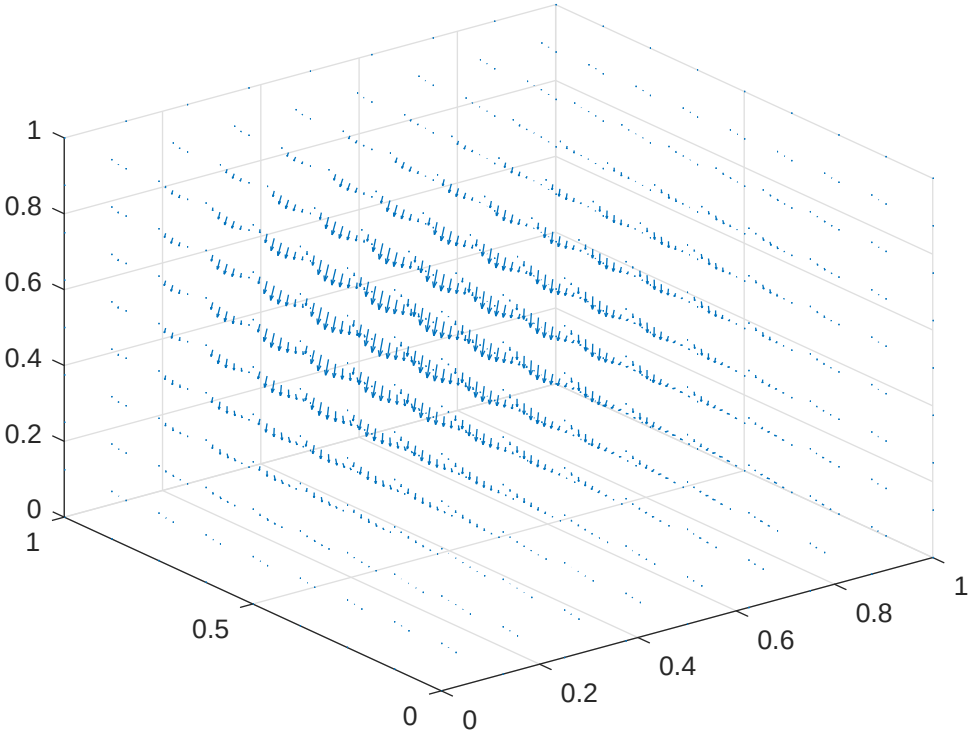


FIGURE 3.3: MATLAB Quiverplot: The numerical solution on the cube displayed as a vector field using arrows for number of nodes $N = 1241$.



Chapter 4

The non-local linear elastosatic model

In the Eringen model (Eringen and Kim, 1974; Eringen et. al., 1977; Eringen, 1978, 1979) the stress field $\boldsymbol{\sigma}$ for an isotropic solid is given as:

$$\boldsymbol{\sigma} = \int_{\Omega} A(\boldsymbol{x}, \boldsymbol{y}) \mathsf{C} : \boldsymbol{\varepsilon}(\boldsymbol{u}(\boldsymbol{y})) \, \mathrm{d}\Omega(\boldsymbol{y}) \quad \forall \boldsymbol{y} \in \Omega, \tag{4.1}$$

where commonly $A(\boldsymbol{x}, \boldsymbol{y}) = a(r)$, the attenuation function dependent on the euclidean distance $r = ||\boldsymbol{y} - \boldsymbol{x}||$, describing the non-locality effects on the strain field from a source $\boldsymbol{y}$ to a field $\boldsymbol{x}$. Moreover, $a(r) \to 0$ as $r \to \infty$, is non-negative and for most purposes $a(r) \approx 0$ for $r \geq R$, where R is known as the finite influence distance[7, p. 3].

Some suggestions for the attenuation function are:

$$\begin{aligned} a(r) &= k e^{-r^2/l^2}, \\ a(r) &= k < 1 - r^2/r_0^2 >^2, \\ a(r) &= k < 1 - r^2/r_0^2 >, \end{aligned}$$

where l is the material length scale, $< \cdot >$ the Macauley symbol (not to be confused with the inner product $< \cdot, \cdot >$), $r_0 = (m_0 + 1)l$, m_0 is some integer no less than one[7, p. 5]. k is obtained through requiring that:

$$\int_{\mathbb{R}^3} A(\boldsymbol{x}, \boldsymbol{y}) dV(\boldsymbol{y}) = 1.$$

Furthermore, the stress field must be invertible, to ensure a unique strain field, and the strain energy stored in Ω must be positive, implying that the strain energy is zero

only when the strain field identically vanishes in Ω. This can be acheived by requiring that these are L_2-functions, and the acompanying eigenvalue integral equation is nondegenerate[7, p. 6].

4.1 Total potential energy

Consider the total potential energy functional given by [7, p. 12]:

$$\Pi[\boldsymbol{u}] = \frac{1}{2}\int_\Omega\int_\Omega A(\boldsymbol{x},\boldsymbol{y})\left[\boldsymbol{\varepsilon}(\boldsymbol{u}(\boldsymbol{x})) : \mathsf{C} : \boldsymbol{\varepsilon}(\boldsymbol{u}(\boldsymbol{y}))\right] \mathrm{d}\Omega(\boldsymbol{y}) d\Omega(\boldsymbol{x}) \\ - \int_\Omega \boldsymbol{f}\cdot\boldsymbol{u}\ \mathrm{d}\Omega\ - \int_{\partial\Omega_N} \boldsymbol{g}_N\cdot\boldsymbol{u}\ \mathrm{d}s. \tag{4.2}$$

It can be shown that the field $\boldsymbol{u}$ satisfying the boundary conditions of, and minimizing $\Pi[\boldsymbol{u}]$, is a solution to the non-local linear elastostatic PDE. The converse holds as well[7, Theorem 1, p. 12-13]. This can be used to construct a non-local finite element method.

4.2 A non-local finite element method

We begin with dividing Ω into tetrahedral finite elements with subdomain Ω_n, where $n = 1, ..., N$, and then choose a piecewise linear basis for V, as we did with the local method. Furthermore, since the linear shape function $\boldsymbol{\psi}$ is the same in all three dimensions of a single node, we employ a running index $i = 3(j-1)+d$, where j is the node number, and d is the component number. Then we may choose $\boldsymbol{\psi}_i = \psi_j \boldsymbol{e}_d$, where $\boldsymbol{e}_d$ is the dth cartesian unit column vector in $\mathbb{R}^3$.

Hence, we can write the numerical solution $\boldsymbol{u}_h$ restricted to the nth element as:

$$\boldsymbol{u}_h(\boldsymbol{x}) = \boldsymbol{\Psi}_n(\boldsymbol{x})\boldsymbol{d}_n \quad \forall \boldsymbol{x} \in \Omega_n,$$

where $\boldsymbol{\Psi}_n \in \mathbb{R}^{3\times 12}$, that is:

$$\boldsymbol{\Psi}_n(\boldsymbol{x}) = \begin{bmatrix} \psi_{n(1)}(\boldsymbol{x})\boldsymbol{I} & \psi_{n(2)}(\boldsymbol{x})\boldsymbol{I} & \psi_{n(3)}(\boldsymbol{x})\boldsymbol{I} & \psi_{n(4)}(\boldsymbol{x})\boldsymbol{I} \end{bmatrix},$$

and $\boldsymbol{I}$ is the 3x3 identity matrix, and $\boldsymbol{d}_n$ is a column vector of the node displacements for that element. This also gives the representation of the strain on an element n:

$$\boldsymbol{\varepsilon}_n(\boldsymbol{x}) = \boldsymbol{\nabla}^{\boldsymbol{s}}\boldsymbol{\Psi}_n\boldsymbol{d}_n = \frac{1}{2}(\boldsymbol{\nabla}\boldsymbol{\Psi}_n(\boldsymbol{x}) + (\boldsymbol{\nabla}\boldsymbol{\Psi}_n)^T(\boldsymbol{x}))\boldsymbol{d}_n \quad \forall \boldsymbol{x} \in \Omega_n,$$

where the symmetric gradient operator is applied column-wise such that $\boldsymbol{\nabla^s \Psi}_n \in \mathbb{R}^{3x3x12}$. Moreover, the nodal displacements of an element n is related to the ordered nodal displacements of all elements $\boldsymbol{U}$ by:

$$\boldsymbol{d}_n = \boldsymbol{C}_n \boldsymbol{U},$$

where $\boldsymbol{C}_n$ is known as the node connection matrix for element n.

So far, this is no different than what we did in the local case. However, inserting these relations into equation (4.2), we get the quadratic form[7, p. 16-17]:

$$\Pi[\boldsymbol{u}_h] = \frac{1}{2}\boldsymbol{U}^T \boldsymbol{K} \boldsymbol{U} - \boldsymbol{F}^T \boldsymbol{U}, \tag{4.3}$$

which takes its minimum when:

$$\boldsymbol{K}\boldsymbol{U} = \boldsymbol{F}.$$

Hence, solving the linear equation in $\boldsymbol{U}$ gives the numerical solution to the non-local linear elastostatic PDE. The non-local stiffness matrix $\boldsymbol{K}$ and the load vector $\boldsymbol{F}$ is given by:

$$\boldsymbol{K} = \sum_{n=1}^{N}\sum_{m=1}^{N} \boldsymbol{C}_n^T \left(\int_{\Omega_n} \int_{\Omega_m} \left[A(\boldsymbol{x}, \boldsymbol{y}) (\boldsymbol{\nabla^s \Psi}_n)^T(\boldsymbol{x}) : \mathsf{C} : \boldsymbol{\nabla^s \Psi}_m(\boldsymbol{y}) \right] \mathrm{d}\Omega(\boldsymbol{y}) \mathrm{d}\Omega(\boldsymbol{x}) \right) \boldsymbol{C}_m,$$

$$\boldsymbol{F} = \sum_{n=1}^{N} \boldsymbol{C}_n^T \left(\int_{\partial\Omega_{Neu} \cap \partial\Omega_n} \boldsymbol{\Psi}_n^T(\boldsymbol{x}) \cdot \boldsymbol{g_{Neu}} \mathrm{d}S \; + \; \int_{\Omega_n} \boldsymbol{\Psi}_n^T(\boldsymbol{x}) \cdot \boldsymbol{f} \mathrm{d}\Omega \right).$$

The load vector is the same as in the local elastostatic FEM method (3.1) though in this case we have included the Neumann boundary term. If we chose the attenuation function A to be Dirac's delta $\delta(\boldsymbol{y} - \boldsymbol{x})$ then the problem formally reduces to the local one[7, p. 4].

4.2.1 Generating the stiffness matrix

For the nth tetrahedral element defined by the set of cartesian coordinates $\{(x_{n(i)}, y_{n(i)}, z_{n(i)})\}_{i=1}^4$, we first solve for the coefficients of the linear shape functions restricted to that element,

by interpolation, and then collect these coefficients in a matrix.

$$\begin{bmatrix} x_{n(1)} & y_{n(1)} & z_{n(1)} & 1 \\ x_{n(2)} & y_{n(2)} & z_{n(2)} & 1 \\ x_{n(3)} & y_{n(3)} & z_{n(3)} & 1 \\ x_{n(4)} & y_{n(4)} & z_{n(4)} & 1 \end{bmatrix} \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \\ d_1 & d_2 & d_3 & d_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where a_i, b_i, c_i, d_i are the coefficients of the ith linear shape function in the canonical basis of monomials. Hence the non-zero component of the ith linear shape function belonging to the nth element, is the scalar function $\psi_{n(i)}(\boldsymbol{x}) = (a_{n(i)}, b_{n(i)}, c_{n(i)}, d_{n(i)}) \cdot (\boldsymbol{x}^T, 1)^T$, $\forall \boldsymbol{x} \in \Omega_n$. Moreover, we can express the symmetric gradients with:

$$\begin{aligned} (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_n)_{kl} &= \frac{1}{2} \left(\frac{\partial}{\partial x_l} \boldsymbol{e}_k^T \cdot \boldsymbol{\Psi}_n + \frac{\partial}{\partial x_k} \boldsymbol{e}_l^T \cdot \boldsymbol{\Psi}_n \right) \\ &= \frac{1}{2} \left[(\tfrac{\partial \psi_{n(1)}}{\partial x_l} \boldsymbol{e}_k^T + \tfrac{\partial \psi_{n(1)}}{\partial x_k} \boldsymbol{e}_l^T) \boldsymbol{I}, \ldots, (\tfrac{\partial \psi_{n(4)}}{\partial x_l} \boldsymbol{e}_k^T + \tfrac{\partial \psi_{n(4)}}{\partial x_k} \boldsymbol{e}_l^T) \boldsymbol{I} \right], \end{aligned}$$

where the partial derivatives are really the coefficients, and the resulting entry a 1x12 vector.

In the next step, we need to take the double index contraction with the isotropic stiffness tensor, and using einstein summation notation where we sum over repeated indices we write,

$$\mathsf{C} : \boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m = \mathsf{C}_{ijkl} (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{lk} = (\lambda \delta_{ij} \delta_{kl} + \mu(\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk})) \, (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{lk},$$

and after some consideration obtain:

$$\begin{aligned} (\mathsf{C} : \boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{11} &= (\lambda + 2\mu)(\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{11} + \lambda (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{22} + \lambda (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{33}, \\ (\mathsf{C} : \boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{22} &= \lambda (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{11} + (\lambda + 2\mu)(\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{22} + \lambda (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{33}, \\ (\mathsf{C} : \boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{33} &= \lambda (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{11} + \lambda (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{22} + (\lambda + 2\mu)(\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{33}, \\ (\mathsf{C} : \boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{12} &= \mu (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{12} + \mu (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{21}, \\ (\mathsf{C} : \boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{13} &= \mu (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{13} + \mu (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{31}, \\ (\mathsf{C} : \boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{23} &= \mu (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{23} + \mu (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_m)_{32} \end{aligned}$$

which as noted earlier are the only entries we need due to the symmetries, and is completely analogous to the local problem. We further note that the entries of the shape gradients are invariant with respect to the indices k, l, since the operator is symmetric, which we use to simplify notation in the next step.

We take the double index contraction with the other shape gradients to obtain the

12x12 system:

$$
\begin{aligned}
(\boldsymbol{\nabla}^{\boldsymbol{s}}\boldsymbol{\Psi}_n)^T : (\mathsf{C} : \boldsymbol{\nabla}^{\boldsymbol{s}}\boldsymbol{\Psi}_m) = (\lambda + 2\mu) \sum_{i=1}^{i=3} (\boldsymbol{\nabla}^{\boldsymbol{s}}\boldsymbol{\Psi}_n)_{ii}^T \cdot (\boldsymbol{\nabla}^{\boldsymbol{s}}\boldsymbol{\Psi}_m)_{ii} \\
+ \lambda \sum_{i=1}^{3} ((\boldsymbol{\nabla}^{\boldsymbol{s}}\boldsymbol{\Psi}_n)_{ii}^T \cdot \sum_{j \neq i}^{3} (\boldsymbol{\nabla}^{\boldsymbol{s}}\boldsymbol{\Psi}_m)_{jj}) \\
+ 2\mu \sum_{i=1}^{3} \sum_{j \neq i}^{3} (\boldsymbol{\nabla}^{\boldsymbol{s}}\boldsymbol{\Psi}_n)_{ij}^T \cdot (\boldsymbol{\nabla}^{\boldsymbol{s}}\boldsymbol{\Psi}_m)_{ij} \,.
\end{aligned}
$$

A further simplifying notation can be found in[7, p. 17], defining:

$$
\boldsymbol{k}_{nm} = \int_{\Omega_n} \int_{\Omega_m} A(\boldsymbol{x}, \boldsymbol{y}) \, (\boldsymbol{\nabla}^{\boldsymbol{s}}\boldsymbol{\Psi}_n)^T (\boldsymbol{x}) : \mathsf{C} : \boldsymbol{\nabla}^{\boldsymbol{s}}\boldsymbol{\Psi}_m(\boldsymbol{y}) \mathrm{d}\Omega(\boldsymbol{y}) \mathrm{d}\Omega(\boldsymbol{x}),
$$

so that we may write:

$$
\boldsymbol{K} = \sum_{n=1}^{N} \sum_{m=1}^{N} \boldsymbol{C}_n^T \boldsymbol{k}_{nm} \boldsymbol{C}_m.
$$

Hence, the non-local information is stored in $\boldsymbol{k}_{nm}$, and then for each n added to the stiffness matrix via the node connection matrices, which is determined by the ordering of the nodes in the mesh.

In the local problem, only a single main loop is needed, since each tetrahedral element contributes only once to the stiffness matrix. In the non-local case, we need two such loops, since each element contributes N times, when the toal is N elements. Hence, give all operations within a loop run in constant time, we would obtain a lower bound on the computational time of N^2, which is computationally heavy to solve, in addition to a dense $3N \times 3N$ system of equations instead of a sparse one in the local case. However, in [7, p. 7] it is noted that the contributions $n \neq m$ are vanishing, as the related finite elements become too far from one another with respect to the finite influence distance R. We can consider this property by making the estimate $a(r > R) = 0$, thus ignoring any such contributions entirely. It is also suggested to use the so-called geodetic distance instead of the Euclidean distance in the attenuation function, that is, the smallest path not crossing the boundary, since otherwise would be unphysical, depending on the mesh. However, this may have significant computational costs for complex meshes, and was not used.

Chapter 5

Application: Non-Local Isotropic Solids

As an example, we take the attenuation function to be

$$A(\boldsymbol{x}, \boldsymbol{y}) = ke^{-r^2/l^2},$$

with the euclidean distance r. Let Ω therefore be any convex, suitable domain, and since it is convex, the geodetic distance would be the same as the euclidean distance. Then, we need to determine the constant k from the equation:

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} ke^{-(x^*-x')^2/l^2-(y^*-y')^2/l^2-(z^*-z')^2/l^2}\mathrm{d}x^*\mathrm{d}y^*\mathrm{d}z^* = 1,$$

and with a change of variables $x = x^* - x'$, $y = y^* - y'$, $z = z^* - z'$ it is apparent that the result is independent of the field x', y', z' since the resulting equation gives:

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} ke^{-x^2/l^2-y^2/l^2-z^2/l^2}\mathrm{d}x\mathrm{d}y\mathrm{d}z = 1,$$

which is equivalent to:

$$k\left(\int_{-\infty}^{\infty} e^{-x^2/l^2}\mathrm{d}x\right)^3 = 1,$$

where for $l > 0$ the integral value is widely known[1],

$$\int_{-\infty}^{\infty} e^{-x^2/l^2}\mathrm{d}x = l\sqrt{\pi},$$

[1] Error Function, Wikipedia, https://en.wikipedia.org/wiki/Error_function

thus we obtain:

$$k = \left(\frac{1}{l\sqrt{\pi}}\right)^3.$$

For the integrals involved in constructing the load vector $\boldsymbol{F}$, and the stiffness matrix $\boldsymbol{K}$ we will be using 5-point Gaussian quadrature rules for the volume integrals, and 4-point Gaussian quadrature rules for the surface integrals. These are exact, up to rounding error, if the integrand is a polyonomial of up to a certain degree[8, p. 271-272], but with the presence of the attenuation function, the integrands are non-polynomial, though the quadratures may improve as $l \to 0$.

According to [7, p. 4], the attenuation function's tendency to vanish at a certain distance is made sensical only through the material length scale l. In particular, l can be taken to be smaller than the smallest dimension of the solid in consideration, and it is the ratio r/l which is of interest. In MATLAB, $e^{-10^n} \approx 0$ for $n \geq 3$, hence any ratio $r/l \geq 32$ gives 0 for our choice of attenuation function. On the other hand, if $r(x, y) < 10^{-16}\ \forall x \in \Omega_m,\ \forall y \in \Omega_n$, less than machine precision, then the two elements are indistinguishable from one another. If say we wanted the diffusion of nonlocality effects to be completely arrested by the nearest next element, and the smallest distance between any two distinct elements is $r \geq h$, a generous choice of about $l \leq 1/(32h)$ will ensure this. Then, approximately, we can easily scale this to consider the nearest two elements, and so on. This could be very useful in estimating the sizes of lists which should be pre-allocated when storing sparse matrices as separate lists, since the stiffness matrix becomes less dense with tighter cut-offs. In this thesis, storage of the stiffness matrix as sparse was only used for the iterative method (described later). We can do a quick measure of the distance between the centers of elements for determining whether or not the attenutation function is considered 0.

A problem we may consider is the cube clamped at one end, with a simple force pushing down on the top. Let the boundary faces of the cube be divided such that the boundary conditions, and the choice of bodily volume forces are:

$$\begin{aligned}
\boldsymbol{\nabla} \cdot \boldsymbol{\sigma} &= \boldsymbol{f}/x_3; && \text{in } \Omega, \\
\boldsymbol{u} &= 0 && \text{on } \partial\Omega_{Clamped}, \\
\boldsymbol{\sigma} \cdot n &= 0 && \text{on } \partial\Omega_{No-Load}, \\
\boldsymbol{\sigma} \cdot n &= \boldsymbol{f} && \text{on } \partial\Omega_{Load},
\end{aligned}$$

where $\boldsymbol{f} = \boldsymbol{f}(x_1, x_2, x_3) = -x_3(x_1^2 - x_1)(x_2^2 - x_2)e_3$, and $\partial\Omega_{Load}$ is defined by the plane $x_3 = 1$ restricted to the cube, $\partial\Omega_{Clamped}$ is defined by the plane $x_2 = 0$ restricted to the

cube, and $\partial\Omega_{No-Load} = \partial\Omega \setminus (\partial\Omega_{Clamped} \cup \partial\Omega_{Load})$. This choice of $\boldsymbol{f}$ is continuous at the edges of the boundaries, and is a downward acting force on the top face.

In this case we choose $l = 0.1$ and consider $r > 2l$ as the cut-off point for the diffusion of the nonlocality effects, with number of nodes $M = 189$. We arbitrarily let $\lambda = 2, \mu = 2$. The resulting stiffness matrix K is shown in figure 5.1. The same problem was also solved with no cut-off point, letting all elements contribute to the stiffness matrix, shown in figure 5.2.

FIGURE 5.1: Stiffness matrix of cube being pushed on the top face, computed in MATLAB, with cut-off.

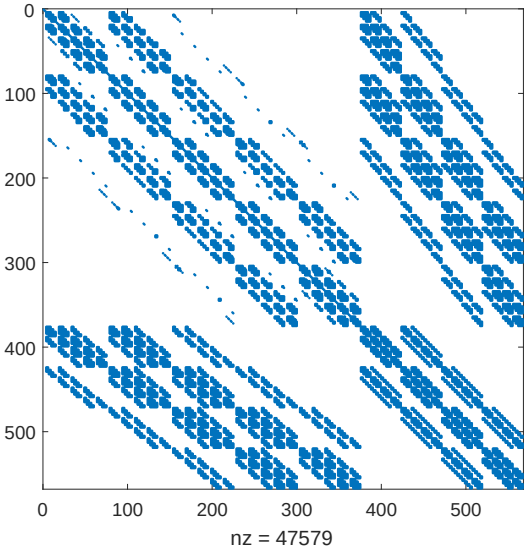
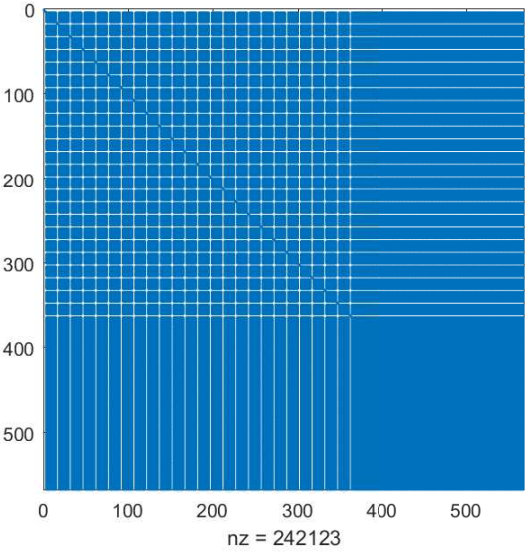
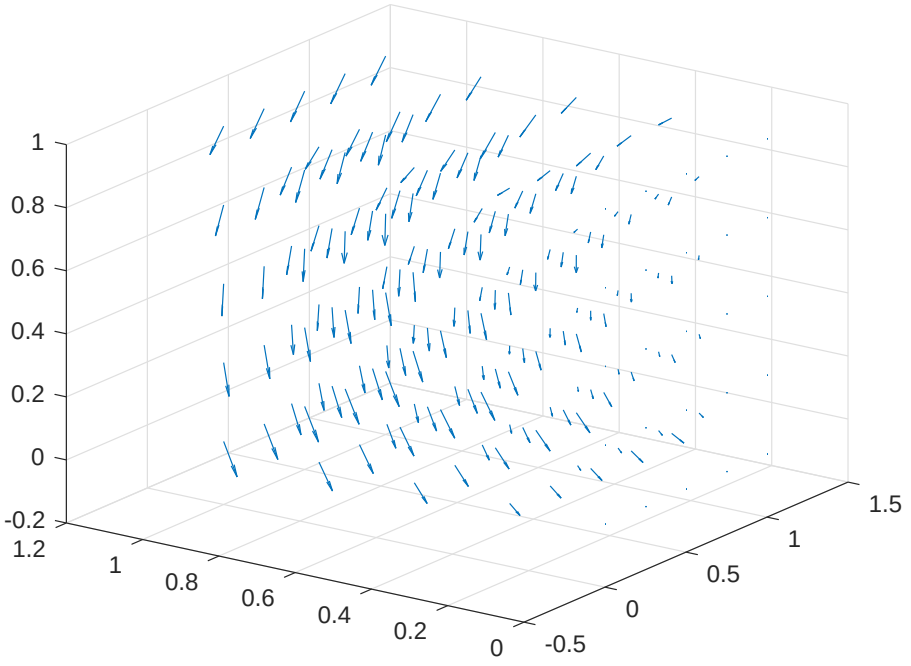


FIGURE 5.2: Stiffness matrix of cube being pushed on the top face, computed in MATLAB, without cut-off.



As is apparent from comparing the two matrices, the system becomes much more dense as the number of contributing elements in the cross-stiffness matrix $\boldsymbol{k}_{nm}$ for $n \neq m$ increases. We can calculate the Frobenius norm of the difference $||K_{cut} - K_{nocut}||_F \approx 1.75$. Assuming that K_{nocut} is the "correct" answer, then the relative error in the Frobenius norm is about 0.0796, indicating a substantial loss of information, though judging by the quiverplots of both (only the no cut-off case is shown) the behaviour of the numerical solution is similar. The two different stiffness matrices nicely illustrates the idea behind vanishing cross-stifness matrices $\boldsymbol{k}_{nm}$ where $n \neq m$. In figure 5.3 the numerical solution is shown for the no cut-off case.

FIGURE 5.3: MATLAB Quiverplot, cube pushed down on top face, no cut-off, 189 nodes.



We can also make a comparison with the clamped cube in the local method. We let $\lambda = 10^9$ Pa and $\mu = 3.33 \cdot 10^4$ Pa as before, using the same values for l as above, and not enforcing any cut-off. We choose the number of nodes $M = 189$. The potential energy of the non-local numerical solution U is $\Pi^h[U] = \frac{1}{2}U^T KU - F^T U < \Pi^h[U_{loc}] \approx 1 \cdot 10^7$, which is lower than the local solution, as expected. The stiffness matrix is shown in figure 5.4 for the non-local clamped cube, and for the local clamped cube. The local case exhibits striking similiaties to earlier case where we enforced a cut-off.

FIGURE 5.4: MATLAB Stiffness Matrix, clamped cube, 189 nodes, non-local.

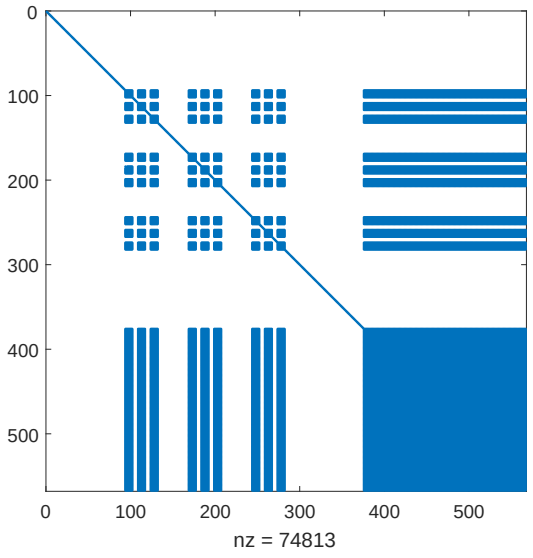
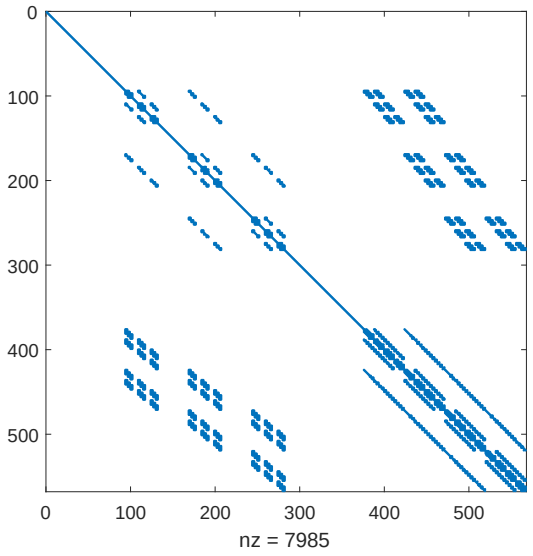
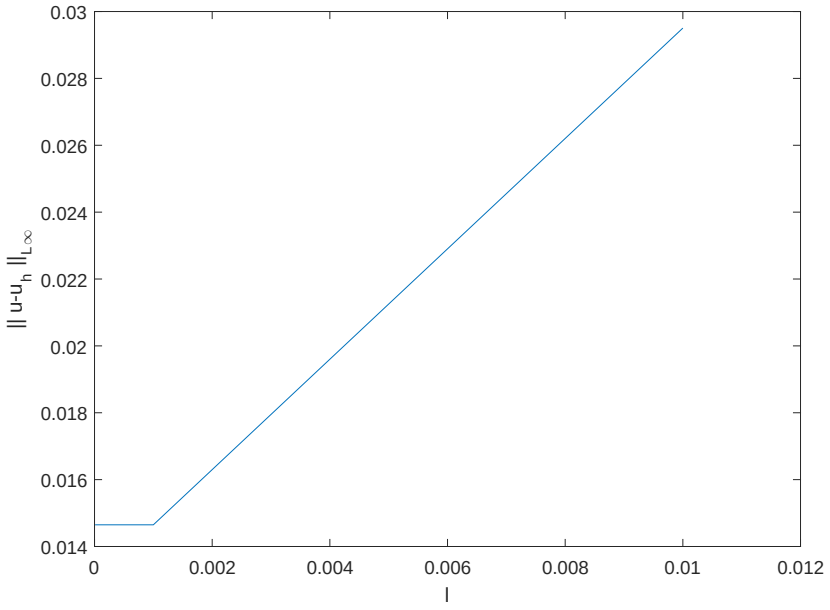


FIGURE 5.5: MATLAB Stiffness Matrix, clamped cube, 189 nodes, local.



Moreover, we can examine the effect of decreasing l on the non-local solution to the clamped cube, to see if it approaches the known local solution. As we can see in figure

FIGURE 5.6: MATLAB Approaching Local Solution, fully clamped cube, 189 nodes, numerical max norm.



5.6, this appears to be the case, although as l becomes very small, we see no difference in $l = 0.0001$ and $l = 0.00001$.

5.1 Iterative Numerical Solution

Once again we introduce another stress field $\boldsymbol{\sigma}$, but this time dependent on a nonlocality correction strain $\boldsymbol{\varepsilon}^c$. Let

$$\boldsymbol{\sigma} = \mathsf{C} : (\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}^c) \quad \text{in } \Omega,$$

where C is the isotropic stress tensor as before. Furthermore, we introduce the free auxiliary strain $\boldsymbol{e}$

$$\begin{aligned} \boldsymbol{e}(\boldsymbol{x}) &= \boldsymbol{\varepsilon}(\boldsymbol{x}) \quad \text{in } \Omega, \\ \boldsymbol{\varepsilon}^c(\boldsymbol{x}) &= \boldsymbol{e}(\boldsymbol{x}) - \int_\Omega A(\boldsymbol{x}, \boldsymbol{y}) \boldsymbol{e}(\boldsymbol{y}) \, \mathrm{d}\Omega(\boldsymbol{y}) \quad \forall \boldsymbol{x} \in \Omega, \end{aligned}$$

This method, as described in [7, p. 19], is based on adding a corrective term at each iteration to approximate nonlocality effects. This term is calculated using information of the previous numerical solution. Let ε_k^c be the kth iteration of the nonlocality correction strain, and let $\boldsymbol{\varepsilon}_0^c = 0$ such that $\boldsymbol{U}_0$ is the equivalent local numerical solution. If we have the local numerical solution, we may then calculate

$$\boldsymbol{\varepsilon}_0^c(\boldsymbol{x}) = \boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_n \boldsymbol{C}_n \boldsymbol{U}_0(\boldsymbol{x}) - \int_{\Omega_n} A(\boldsymbol{x}, \boldsymbol{y}) \boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi}_n \boldsymbol{C}_n \boldsymbol{U}_0(\boldsymbol{y}) \, \mathrm{d}\Omega(\boldsymbol{y}) \quad \forall \boldsymbol{x} \in \Omega_n,$$

and use this to obtain $\boldsymbol{U}_1$ by solving the new system:

$$\begin{aligned} \boldsymbol{K}\boldsymbol{U}_1 &= \boldsymbol{F} + \boldsymbol{F}_0^c, \\ \boldsymbol{F}_0^c &= \sum_{n=1}^{N} \boldsymbol{C}_n^T \int_{\Omega_n} (\boldsymbol{\nabla}^{\boldsymbol{s}} \boldsymbol{\Psi})_n^T : \mathsf{C} : \boldsymbol{\varepsilon}_0^c \, \mathrm{d}\Omega, \end{aligned}$$

and so on.

Let us first examine the numerical convergence of the local numerical solution $\boldsymbol{U}_0$ using this method. We use the clamped unit sphere and let $\lambda = 1$ and $\mu = 1$ arbitrarily. As figure 5.7 suggests the error is decreasing, which suggets convergence of the method, but very slowly. The implemention of this method uses sparse index lists for the sparse stiffness matrix and vectorized quadratures, and so we show the computation time increase with N (as calculated by MATLAB) as well in figure 5.8. Overall, this is much faster than the local code combining three matrices in figure 3.2, which does not take advantage of such coding techniques.

FIGURE 5.7: Log-log error plot of local clamped isotropic sphere, y-axis: $\log ||\boldsymbol{U}_0 - \boldsymbol{U}||_\infty$, x-axis: $\log(h)$, where $h = 1/N^3$. Linear fit gradient: 0.14.

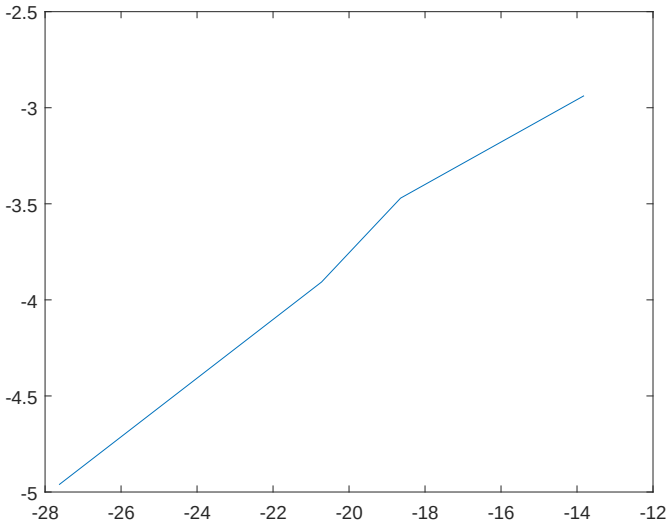
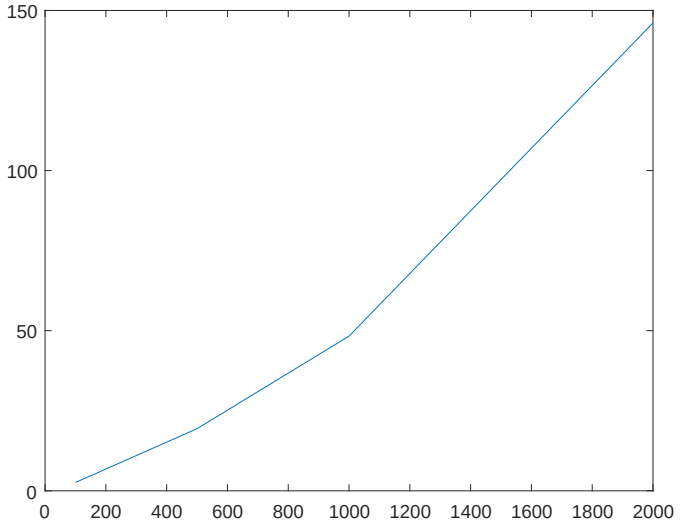


FIGURE 5.8: Increase in computation time of clamped isotropic of local sphere, y-axis: time in seconds, x-axis: N.



Chapter 6

Conclusion

Overall, the numerical results have been found unsatisfactory. There appears to be bugs or incorrectly implemented FEM solvers. It takes a very long time to compute a solution on even smaller sized problems, especially in the non-local method. However, the optimizations that were done confered considerable improvement in computation time. Unfortunately, it was never discovered what was wrong with the MATLAB implementation, and because of this and lack of time, the iterative implementation was abandoned (as it ultimately uses the local solution at the first time-step, i.e. the $n = m$ elements in the cross-stiffness matrix).

Evidence for the implementation being incorrect can be found in the obtained numerical convergence rates. In an attempt to find what the "correct" numerical rate would be, the local-case was also implemented in Python using the FENICS framework, coupled with Numpy and PyPlot. Here we made comparisons between the clamped cube and clamped sphere. Implementing using this framework is much easier (perhaps due to the nature of Python), but there is little transparency, and so perhaps difficult to make suitable modifications for the non-local methods. It was also the case that we were running out of memory in Python, so the error was only computed for small N.

The 'errornorm' function in FENICS will first interpolate the numerical and exact solution to a higher-order common space (the default is 3rd order polynomials) and then subtract the fields before computing the integral. Letting $dx = dy = dz = 1/N$ where N is the number of cell divisions in directions x, y, z (in the cube case) we can calculate the numerical convergence rate from the error model $E_i = Ch_i^r$, where E_i, h_i is the error and stepsize at experimental run i, r is the estimated (constant) convergence rate and C is some constant. Results from the clamped isotropic local cube gave convergence rates

approaching zero for both the numerical L_2 norm and L_∞ norm error in the Fenics implementation, while seemingly approaching 0 and 0.14 in the MATLAB implementation. Results were no better with the spherical case.

Chapter 7

Implementation

The non-local and local methods were implemented in MATLAB, not using any predefined FEM libraries available. As stated earlier, the code for the generation of the sphere mesh is the work of Kjetil A. Johannensen, and is freely available for download to anyone, specifically made for the students writing the project in the finite element methods course TMA4220 in 2012. This encompasses the files *getSphere.m*, and *shell.m*. At the heart of this, lies the built-in functions *delaunay*, $TriRep$ and $freeBoundary$, which does the actual triangulization and edge detection once the points have been given. The method used in the generation of the unit cube and L-shape are found in *getCube.m* and *getLshape.m*, which also take advantage of the built-in triangulation functions.

The methods used for numerical integration, the quadrature rules, are found in quadrature3D.m and quadrature2Dx.m, and rely on the notion of barycentric co-ordinates. The scripts with the actual FEM methods are found in *classical.m* and *nonlocal.m*. These contain a lot of commented-out code, depending on what case is being studied. Finally, this thesis itself was written in LaTeX, using TeXstudio, on an open-source template created by Steve R. Gunn, and modified by Sunil Patel[1]. A script called $writeVTK$ was used to build the Paraview image shown for the sphere, and is the work of Daniel Petersheim[2].

One of the difficulties in implementing the methods in MATLAB stem from the fact that anonymous function calls are very slow, hence it is recommended to avoid such calls as much as possible. In the case of quadratures for the product of shape functions and the attenuation function, we have vectorized the code, passing only the coefficients of the shape functions instead of creating a new function for each product. In addition,

[1] www.sunilpatel.co.uk

[2] https://www.mathworks.com/matlabcentral/fileexchange/25784-writevtk

if a matrix has rather few non-zero elements, it should be stored as a sparse matrix (speeding up computation and reducing memory), but indexing into sparse matrices is very slow, as it has to shift all the elements each time you zero a value or add a non-zero value, and so we have to store index lists instead of directly operating on sparse matrices, and build the entire matrix as sparse afterwards. It is also recommended to pre-allocate any matrices, vectors, etc. as much as possible, i.e. if the size is known or can be estimated beforehand.

Appendix

7.0.1 MATLAB Code

Below are the implementations in MATLAB of the quadrature rules used for integration in the local and non-local[3].

```
function [I] = quadrature3D(p1,p2,p3,p4,Nq,g)
alpha = 1/4 + 3*sqrt(5)/20;
beta = 1/4 - sqrt(5)/20;
I = 0;
volume = (1/6)*abs(det([p1-p4,p2-p4,p3-p4]));
if (Nq == 1)
z_q = [p1 p2 p3 p4]*[1/4 1/4 1/4 1/4]';
rho = 1;
I = g(z_q)*rho;
I = I*volume;
elseif (Nq == 4)
z_q = [p1 p2 p3 p4]*[alpha beta beta beta;
beta alpha beta beta;
beta beta alpha beta;
beta beta beta alpha]';
rho = 1/4;
for (i=1:Nq)
I = I + rho*g(z_q(:,i));
end
I = I*volume;
elseif (Nq == 5)
z_q = [p1 p2 p3 p4]*[1/4 1/4 1/4 1/4;
```

[3] TMA4220 Project Part 1, Wiki math NTNU, https://wiki.math.ntnu.no/_media/tma4220/2014h/problem_set.pdf

```
1/2 1/6 1/6 1/6;
1/6 1/2 1/6 1/6;
1/6 1/6 1/2 1/6;
1/6 1/6 1/6 1/2]';
rho = [-4/5 9/20 9/20 9/20 9/20];
for (i=1:Nq)
I = I + rho(i)*g(z_q(:,i));
end
I = I*volume;
end
return
```

```
 function [I] = quadrature2Dx(p1,p2,p3,Nq,g)
I = 0;
n = cross(p2-p1,p3-p1);
area = (1/2)*dot(n,n)^(1/2);
if (Nq == 1)
z_q = [p1 p2 p3]*[1/3 1/3 1/3]';
rho = 1;
I = g(z_q)*rho;
I = I*area;
elseif (Nq == 3)
z_q = [p1 p2 p3]*[1/2 1/2 0;
1/2 0 1/2;
0 1/2 1/2]';
rho = 1/3;
for (i=1:Nq)
I = I + g(z_q(:,i))*rho;
end
I = I*area;
elseif (Nq == 4)
z_q = [p1 p2 p3]*[1/3 1/3 1/3;
3/5 1/5 1/5;
1/5 3/5 1/5;
1/5 1/5 3/5]';
rho = [-9/16 25/48 25/48 25/48];
for(i=1:Nq)
I= I + g(z_q(:,i))*rho(i);
end
I = I*area;
end
return
```

7.0.2 Mesh generation

There are three mesh generation scripts written in MATLAB, but the one for the sphere will not be shown here, as it is someone else's work. The code for the sphere is available at NTNU's TMA4220 project page for 2014[4]

```
function [p tri edge] = getCube(N),
m = round(log(N)/log(8));
vec = zeros(m+1,1);
for i=1:(m+1)
vec(i) = (i-1)*1/m;
end
[ x,y,z] = meshgrid(vec,vec,vec);
X = [x(:),y(:),z(:)];
vec = zeros(m,1);
for i=1:m
vec(i) = 1/(2*m) + (i-1)*1/m;
end
[ x,y,z] = meshgrid(vec,vec,vec);
X_c = [x(:),y(:),z(:)];
p = [X;X_c];
tri = delaunay(p);
topology = TriRep(tri,p);
edge = freeBoundary(topology);
end
```

[4]TMA4220 Project 2014, Wiki math NTNU, http://www.math.ntnu.no/emner/TMA4220/2014h/project/Grids.zip

```
 function [p tri edge] = getLshape(N),
p0 = getCube(N);
[m,n] = size(p0);
Ix = zeros(m,n);
Iy = Ix;
Ix(:,1) = Ix(:,1)+1;
Iy(:,2) = Iy(:,2)+1;
p = [p0+Ix;p0;p0+Iy];
p = unique(p,'rows');
tri = delaunay(p);
triremove = [];
[mtri,ntri] = size(tri);
for i=1:mtri
u = p(tri(i,:),:);
ucenter = (u(1,1:2)+u(2,1:2)+u(3,1:2)+u(4,1:2))*1/4;
if ucenter(1)>1
if ucenter(2)>1
triremove = [triremove;i];
end
end
end
tri = removerows(tri,'ind',triremove);
topology = TriRep(tri,p);
edge = freeBoundary(topology);
end
```

7.0.3 FEM methods

The three scripts, for the local and non-local case, and the imcomplete iterative non-local case, cotain a lot commented-out code, as mentioned earlier. As it is shown, the code solves one specific case. In addition, some code has been moved into sub scripts, shown at the end. We also include the Python code for the FENICS framework.

Local Method:

```
N = 8^4;
[p,tri,edge] = getCube(N);
[N,notused] = size(p);
lambda=10^9;
mu=3.33*10^4;
m1 = zeros(12,12);
m2 = zeros(12,12);
mtemp = zeros(12,12);
m_global = zeros(3*N,3*N);
m_global = sparse(m_global);
b_global = zeros(3*N,1);
b_global = sparse(b_global);
v1 = zeros(1,12);
%cube case
f_1 = @(x) (-1)*((lambda+2*mu)*2*(x(2)^2-x(2))*(x(3)^2-x(3))
+mu*2*(x(1)^2-x(1))*(x(3)^2-x(3))
+mu*2*(x(1)^2-x(1))*(x(3)^2-x(3))
+(lambda+mu)*(2*x(1)-1)*(2*x(2)-1)*(x(3)^2-x(3))
+(lambda+mu)*(2*x(1)-1)*(2*x(3)-1)*(x(2)^2-x(2)));
f_2 = @(x) (-1)*((lambda+2*mu)*2*(x(1)^2-x(1))*(x(3)^2-x(3))
+mu*2*(x(2)^2-x(2))*(x(3)^2-x(3))
+mu*2*(x(2)^2-x(2))*(x(3)^2-x(3))
+(lambda+mu)*(2*x(2)-1)*(2*x(1)-1)*(x(3)^2-x(3))
+(lambda+mu)*(2*x(2)-1)*(2*x(3)-1)*(x(1)^2-x(1)));
f_3 = @(x) (-1)*((lambda+2*mu)*2*(x(2)^2-x(2))*(x(1)^2-x(3))
+mu*2*(x(3)^2-x(3))*(x(1)^2-x(3))
+mu*2*(x(3)^2-x(3))*(x(1)^2-x(1))
+(lambda+mu)*(2*x(3)-1)*(2*x(2)-1)*(x(1)^2-x(1))
+(lambda+mu)*(2*x(3)-1)*(2*x(1)-1)*(x(2)^2-x(2)));
[Nk,Np] = size(tri); %Nk number of elements, Np number of points in tetrahedron
Nq = 5; %number of quadrature points
ip = eye(4); %4x4 identity matrix
cp = zeros(4); %4x4 coefficient matrix for linear basis functions
b = zeros(3*4,1); %empty load vector for each tetrahedron using runnning index
u_true = zeros(3*N,1); %empty vector for analytical solution
%sphere case
r_x = @(x) 1-x(1)^2-x(2)^2-x(3)^2; %analytical solution 1st component, the others are 0
%cube case
u_cube = @(x) (x(1)^2-x(1))*(x(2)^2-x(2))*(x(3)^2-x(3));
%%% Construct Stiffness Matrix and Loading Vector %%%
for k=1:Nk
%grab tetrahedron points corresponding to element with nodes
%tri(k,:)
pk = [p(tri(k,:),:),ones(4,1)];
```

```
%solve for linear basis function coefficients
cp = pk\ip;
%grab physical volume of tetrahedron
vk = (1/6)*abs(det([pk(1,1:3)'-pk(4,1:3)',pk(2,1:3)'-pk(4,1:3)',pk(3,1:3)'-pk(4,1:3)']));
%Construct first matrix
for i=1:4
for j=1:4
v1(3*(j-1)+1) = vk*cp(1:3,i)'*cp(1:3,j);

%construct third matrix in same loop
m3((3*(i-1)+1):(3*(i-1)+3),(3*(j-1)+1):(3*(j-1)+3)) = vk*[0,cp(1,j)*cp(2,i),cp(1,j)*cp(3,i);
cp(2,j)*cp(1,i),0,cp(2,j)*cp(3,i);
cp(3,j)*cp(1,i),cp(3,j)*cp(2,i),0];
end
%construct a toeplitz matrix of the integrals, but only use the first three rows
m1((3*(i-1)+1):(3*(i-1)+3),1:12) = [v1;[v1(12),v1(1:11)];[v1(11:12),v1(1:10)]];
end
%Construct second matrix
m2 = vk*[cp(1:3,1);cp(1:3,2);cp(1:3,3);cp(1:3,4)]*[cp(1:3,1)',cp(1:3,2)',cp(1:3,3)',cp(1:3,4)'];

%Construct load
for i=1:4
%done component wise for each node
fshape_1 = @(x) f_1(x)*[x(1),x(2),x(3),1]*cp(:,i);
fshape_2 = @(x) f_2(x)*[x(1),x(2),x(3),1]*cp(:,i);
fshape_3 = @(x) f_3(x)*[x(1),x(2),x(3),1]*cp(:,i);
%compute the integral for each component using quadrature rule
%using running index, this is done in physical space
b(3*(i-1)+1) = quadrature3D(pk(1,1:3)',pk(2,1:3)',pk(3,1:3)',pk(4,1:3)',Nq,fshape_1);
b(3*(i-1)+2) = quadrature3D(pk(1,1:3)',pk(2,1:3)',pk(3,1:3)',pk(4,1:3)',Nq,fshape_2);
b(3*(i-1)+3) = quadrature3D(pk(1,1:3)',pk(2,1:3)',pk(3,1:3)',pk(4,1:3)',Nq,fshape_3);
end
%add to global matrix, following node order 1,...,N (with 3 entries
%for each node sine u is a 3d vector field
m_local = lambda*m1 + (lambda+mu)*m2 + mu*m3;
pos= 3*(tri(k,:)-1)+1;
for i=1:4
for j=1:4
m_global(pos(i):pos(i)+2,pos(j):pos(j)+2) = m_global(pos(i):pos(i)+2,pos(j):pos(j)+2)
+ m_local(3*(i-1)+1:3*(i-1)+3,3*(j-1)+1:3*(j-1)+3);
end
b_global(pos(i):(pos(i)+2)) = b_global(pos(i):pos(i)+2)+ b((3*(i-1)+1):(3*(i-1)+3));
%cube case
u_true(pos(i)) = u_cube(pk(i,1:3));
u_true(pos(i)+1) = u_true(pos(i));
u_true(pos(i)+2) = u_true(pos(i));
end
end
%%% PARSE DIRICHlET BOUNDARY
% Get edge boundary and set dirichlet solution
[Ne, Npe] = size(edge);
% here I force the solution to be u=0 on the edges.

for k=1:Ne
edgepos = 3*(edge(k,:)-1)+1; %grab running index
```

```
for i=1:3
for j=1:3
%Idea: set row to 0 and diagonal to 1 for m
%and to 0 for that index of the load vector, to force that u*1=0
m_global(edgepos(i)+(j-1),:) = 0;
%set column to zero, since that u is supposed to be 0
%maintaining symmetry
m_global(:,edgepos(i)+(j-1)) = 0;
%set diagonal to 1
m_global(edgepos(i)+(j-1),edgepos(i)+(j-1)) = 1;
end
b_global(edgepos(i):edgepos(i)+2) = 0;
end
end
% solve using backslash
u = m_global\b_global;
% write VTK
%string = strcat('classical',num2str(i));
%writeVTK(string,tri,p,full(u(1:3:end)));
%quiver3(p(:,1),p(:,2),p(:,3),u(1:3:end),u(2:3:end),u(3:3:end))
```

Non-local Method

```
%Non-Local Finite Element Method Script
%For the linear elastic element
%Author: Daniel Osen 2016

%Choose Computational Domain
%Sphere Case, N = number of nodes
%N = 189;
%[p,tri,edge] = getSphere(N);

%Cube case, N = number of nodes (afterwards)
N = 8^4;
[p,tri,edge] = getCube(N);
[N,~] = size(p);

%Grab number of elements Nk, NP=4
[Nk,Np] = size(tri);
[N_edge,Np_edge] = size(edge);
%Pre-Allocate matrices
ip = eye(4);
id = eye(3);
M_global = zeros(3*N,3*N);
B_global = zeros(3*N,1);
U_classical = zeros(3*N,1);
list_dirichlet = [];
list_loadface = [];
list_noloadface = [];

%Attenuation Function (Euclidean Distance)
%material length scale l>0,
atn_l = 0.00001;%
%Finite influence distance parameter
atn_alpha = 1;
atn_r = atn_l*(1+atn_alpha);
%Normalization constant (Sphere)
atn_k = 1/(atn_l^3*(pi)^(3/2));
%function call
atn = @(x,y) atn_k*exp(-(norm(y-x))^2/atn_l^2);

%Lame constants
lambda=2;
mu=2;

%solutions and load functions
%clamped sphere
%u_1 = @(x) 1-norm(x);
%u_2 = @(x) 0;
%u_3 = @(x) 0;
%f_1 = @(x) 2*lambda+8*mu;
%f_2 = @(x) 0;
%f_3 = @(x) 0;
%cube clamped at one end
%f_surf_1 = @(x) 0;
%f_surf_2 = @(x) 0;
%f_surf_3 = @(x) -(x(1)^2-x(1))^2*(x(2)^2-x(2))^2;
```

```
%f_1 = @(x) 0;
%f_2 = @(x) 0;
%f_3 = @(x) -x(3)*(x(1)^2-x(1))^2*(x(2)^2-x(2))^2;
%cube clamped at all ends
f_1 = @(x) (-1)*((lambda+2*mu)*2*(x(2)^2-x(2))*(x(3)^2-x(3))
+mu*2*(x(1)^2-x(1))*(x(3)^2-x(3))
+mu*2*(x(1)^2-x(1))*(x(3)^2-x(3))
+(lambda+mu)*(2*x(1)-1)*(2*x(2)-1)*(x(3)^2-x(3))
+(lambda+mu)*(2*x(1)-1)*(2*x(3)-1)*(x(2)^2-x(2)));
f_2 = @(x) (-1)*((lambda+2*mu)*2*(x(1)^2-x(1))*(x(3)^2-x(3))
+mu*2*(x(2)^2-x(2))*(x(3)^2-x(3))
+mu*2*(x(2)^2-x(2))*(x(3)^2-x(3))
+(lambda+mu)*(2*x(2)-1)*(2*x(1)-1)*(x(3)^2-x(3))
+(lambda+mu)*(2*x(2)-1)*(2*x(3)-1)*(x(1)^2-x(1)));
f_3 = @(x) (-1)*((lambda+2*mu)*2*(x(2)^2-x(2))*(x(1)^2-x(3))
+mu*2*(x(3)^2-x(3))*(x(1)^2-x(3))
+mu*2*(x(3)^2-x(3))*(x(1)^2-x(1))
+(lambda+mu)*(2*x(3)-1)*(2*x(2)-1)*(x(1)^2-x(1))
+(lambda+mu)*(2*x(3)-1)*(2*x(1)-1)*(x(2)^2-x(2)));
f_surf_1 = @(x) 0;
f_surf_2 = @(x) 0;
f_surf_3 = @(x) 0;
u_true = zeros(3*N,1);
u_cube = @(x) (x(1)^2-x(1))*(x(2)^2-x(2))*(x(3)^2-x(3));



%Quadrature points and weights, fixed at 5 for the attenuation function.
Nq = 5;
rho = [-4/5 9/20 9/20 9/20 9/20];

%Begin iteration over elements
for t=1:Nk

%Grab tetrahedron points
pt = [p(tri(t,:),:),ones(4,1)];

%solve for linear basis function coefficients
cpt = pt\ip;

%grab physical volume of tetrahedron
vt = (1/6)*abs(det([pt(1,1:3)'-pt(4,1:3)',pt(2,1:3)'-pt(4,1:3)',pt(3,1:3)'-pt(4,1:3)']));

%Build tensor of derivatives
tdiv = build_derivative_nonlocal(cpt);

%Construct node connection matrix
pos_t= 3*(tri(t,:)-1)+1;
C_t = build_nodeconnection_nonlocal(pos_t,N);

%Get quadrature points
z_qt = [pt(1,1:3)',pt(2,1:3)',pt(3,1:3)',pt(4,1:3)']*[1/4 1/4 1/4 1/4;1/2 1/6 1/6 1/6;1/6 1/2 1/

%Local Solution
u_true(pos_t) = u_cube(pt(:,1:3));
```

```
u_true(pos_t+1) = u_true(pos_t);
u_true(pos_t+2) = u_true(pos_t);

%Inner element Loop
for k=1:Nk

%Grab tetrahedron points
pk = [p(tri(k,:),:),ones(4,1)];

%If the distance between the elements is large enough,
%it is computationally efficient to ignore any contributions,
%at presumably a small loss of error with the correct choice of the
%finite influence distance R. This distance is only considered
%large or small relative to the material length scale l. Hence if
%r/l >> 1, then atn is considered 0, so we set R = l(1+alpha), where
%alpha is some positive number, possibly small.
dist_center = sum(pk(:,1:3))/4-sum(pt(:,1:3))/4;
if norm(dist_center)<inf

%solve for linear basis function coefficients
cpk = pk\ip;

%grab physical volume of tetrahedron
vk = (1/6)*abs(det([pk(1,1:3)'-pk(4,1:3)',pk(2,1:3)'-pk(4,1:3)',pk(3,1:3)'-pk(4,1:3)']));

%Build tensor of derivatives
kdiv = build_derivative_nonlocal(cpk);

%Build contributing stiffness matrix
M_tk = build_stiffness_nonlocal(lambda,mu,tdiv,kdiv);

%Integrate stiffness matrix
%Apart from the attenuation function, everything beneath the
%integral sign, as described in the thesis, is constant.
%This allows us to take the stiffness matrix out of the integral.
%We integrate the attenuation function using the quadrature rules
%employed in the classical case. Here we limit the quadrature rule
%to 5 points.
int_atn = 0;
%get quadrature points
z_qk = [pk(1,1:3)',pk(2,1:3)',pk(3,1:3)',pk(4,1:3)']
*[1/4 1/4 1/4 1/4;1/2 1/6 1/6 1/6;1/6 1/2 1/6 1/6;1/6 1/6 1/2 1/6;1/6 1/6 1/6 1/2]';
for i=1:5
for j=1:5
int_atn = int_atn + rho(i)*rho(j)*atn(z_qt(:,i),z_qk(:,j));
end
end
%scale with volume and make to local stiffness matrix
int_atn = int_atn*vt*vk;
M_tk = int_atn*M_tk;

%Build node connection matrix
pos_k= 3*(tri(k,:)-1)+1;
C_k = build_nodeconnection_nonlocal(pos_k,N);
```

```
%Add to global matrix
M_global = M_global + C_t'*M_tk*C_k;

else
end

end

%Build load vector.
B_t = build_load_volume(f_1,f_2,f_3,cpt,pt,vt);
%Add to global load vector
B_global = B_global + C_t'*B_t;

%Cube Boundary
%Check if element has edges at boundary
[ind1,~,~] = find(edge==tri(t,1));
[ind2,~,~] = find(edge==tri(t,2));
[ind3,~,~] = find(edge==tri(t,3));
[ind4,~,~] = find(edge==tri(t,4));
%We now have 4 lists of row indices, possibly empty,
%each row index is where an edge with a node from the tetrahedron
%has been found
%It remains to find if there is a shared row index between
%at least 3 of those lists.
indlist = [ind1;ind2;ind3;ind4];
if(isempty(indlist)==0)
uniqueind = unique(indlist);
countmatrix = [uniqueind,histc(indlist,uniqueind)];
[indlength,notused] = size(countmatrix);
edgelist_t = [];
for i=1:indlength
if countmatrix(i,2) >= 3
edgelist_t = [edgelist_t;countmatrix(i,1)];
end
end
edgelist_length = length(edgelist_t);
%edgelist_t now contains all the row indices of edge
%which has at exactly three nodes shared with the nodes in the
%tetrahedron
for j=1:edgelist_length
%Cube/sphere case clamped at all ends (only dirichlet boundary)
list_dirichlet = [list_dirichlet;edgelist_t(j)];
%Cube case clamped at one end
%{
p_edge_t = p(edge(edgelist_t(j),:),:);
%If y is 0, we are the clamped boundary and add this to be parsed
%later

if abs(p_edge_t(1,2))<eps && abs(p_edge_t(2,2))<eps && abs(p_edge_t(3,2))<eps
list_dirichlet = [list_dirichlet;edgelist_t(j)];
%if z=1 we are being pushed by a force and need to integrate
elseif abs(1-p_edge_t(1,3))<eps && abs(1-p_edge_t(2,3))<eps && abs(1-p_edge_t(3,3))<eps
B_surf_t = zeros(12,1);
list_loadface = [list_loadface;edgelist_t(j)];
for i=1:4
```

```
%NB: Anonymous functions in matlab are notoriously slow but...
%multiply the vector load function with the vector shape function
%done component wise for each node
fshape_1 = @(x) f_surf_1(x)*[x(1),x(2),x(3),1]*cpt(:,i);
fshape_2 = @(x) f_surf_2(x)*[x(1),x(2),x(3),1]*cpt(:,i);
fshape_3 = @(x) f_surf_3(x)*[x(1),x(2),x(3),1]*cpt(:,i);
%compute the integral for each component using quadrature rule
%using running index, this is done in physical space
B_surf_t(3*(i-1)+1) = quadrature2Dx(p_edge_t(1,1:3)',p_edge_t(2,1:3)'
,p_edge_t(3,1:3)',4,fshape_1);
B_surf_t(3*(i-1)+2) = quadrature2Dx(p_edge_t(1,1:3)',p_edge_t(2,1:3)'
,p_edge_t(3,1:3)',4,fshape_2);
B_surf_t(3*(i-1)+3) = quadrature2Dx(p_edge_t(1,1:3)',p_edge_t(2,1:3)'
,p_edge_t(3,1:3)',4,fshape_3);
end
B_global = B_global + C_t'*B_surf_t;
else
%we are neither at the clamped boundary nor at the boundary
%being pushed, in this case the forces are zero, hence the
%integral is zero, and we don't have to do anything.
list_noloadface = [list_noloadface;edgelist_t(j)];
end
%}

end
else
end

end

%Parse Dirichlet Boundary (Cube)
Ne = length(list_dirichlet);
for s=1:Ne
edgepos = 3*(edge(list_dirichlet(s),:)-1)+1;
for i=1:3
for j=1:3
%Idea: set row to 0 and diagonal to 1 for m
%and to 0 for that index of the load vector, to force that u*1=0
M_global(edgepos(i)+(j-1),:) = 0;
%set column to zero, since that u is supposed to be 0
%maintaining symmetry
M_global(:,edgepos(i)+(j-1)) = 0;
%set diagonal to 1
M_global(edgepos(i)+(j-1),edgepos(i)+(j-1)) = 1;
end
B_global(edgepos(i):edgepos(i)+2) = 0;
end
end
%{
%Parse Dirchlet Boundary (Clamped Sphere)
[Ne, Npe] = size(edge);
for k=1:Ne
edgepos = 3*(edge(k,:)-1)+1; %grab running index
for i=1:3
for j=1:3
```

```
%Idea: set row to 0 and diagonal to 1 for m
%and to 0 for that index of the load vector, to force that u*1=0
M_global(edgepos(i)+(j-1),:) = 0;
%set column to zero, since that u is supposed to be 0
%maintaining symmetry
M_global(:,edgepos(i)+(j-1)) = 0;
%set diagonal to 1
M_global(edgepos(i)+(j-1),edgepos(i)+(j-1)) = 1;
end
B_global(edgepos(i):edgepos(i)+2) = 0;
end
end
%}
%Solve using backslash
U_global = M_global\B_global;
quiver3(p(:,1),p(:,2),p(:,3),U_global(1:3:end),U_global(2:3:end),U_global(3:3:end))
%spy(M_global)
```

Non-local Iterative Method

```
%Iterative non-local method
%Author: Daniel Osen
N = 200;
[p,tri,edge] = getSphere(N);

%Grab number of elements Nk, NP=4
[Nk,Np] = size(tri);
[N_edge,Np_edge] = size(edge);
%Pre-Allocate matrices
ip = eye(4);
id = eye(3);
%M_global = sparse(zeros(3*N,3*N));
B_global = zeros(3*N,1);
u_true = zeros(3*N,1);
list_dirichlet = [];
list_loadface = [];
list_noloadface = [];
I_list = zeros(144*N,1);
J_list = zeros(144*N,1);
X_list = zeros(144*N,1);
lambda=1;
mu=1;

%solutions and load functions
%clamped sphere
u_1 = @(x) 1-norm(x);
u_2 = @(x) 0;
u_3 = @(x) 0;
f_1 = @(x) 2*lambda+8*mu;
f_2 = @(x) 0;
f_3 = @(x) 0;

%Quadrature points and weights, fixed at 5 for the attenuation function.
Nq = 5;
%rho = [-4/5 9/20 9/20 9/20 9/20];
```

```
for t=1:Nk
%Grab tetrahedron points
pt = [p(tri(t,:),:),ones(4,1)];

%solve for linear basis function coefficients
cpt = pt\ip;

%grab physical volume of tetrahedron
vt = (1/6)*abs(det([pt(1,1:3)'-pt(4,1:3)',pt(2,1:3)'-pt(4,1:3)',pt(3,1:3)'-pt(4,1:3)']));

%Build tensor of derivatives
tdiv = build_derivative_nonlocal(cpt);

%Construct node connection matrix
pos_t= 3*(tri(t,:)-1)+1;
%C_t is now sparse, constructed with triplets
C_t = build_nodeconnection_nonlocal(pos_t,N);

%Build contributing stiffness matrix
M_tk = build_stiffness_nonlocal(lambda,mu,tdiv,tdiv);

%integrate
M_tk = M_tk*vt;

%Add to global stiffness matrix
%To speed up code, we will store values in triplets, and sparse them
%together afterwards
%M_coords = C_t'*M_tk*C_t;
%Sparse lists, direct computation, no node connection matrix necessary
I_coords = zeros(144,1);
J_coords = I_coords;
X_vals = J_coords;
for i=1:12
for j=1:12
I_coords(12*(i-1)+j) = pos_t(floor((i-1)/3+1))+mod(i-1,3); %t'th-element
J_coords(12*(i-1)+j) = pos_t(floor((j-1)/3+1))+mod(j-1,3); %k'th element
X_vals(12*(i-1)+j) = M_tk(i,j); %ok
end
end
%[I_coords,J_coords,X_vals] = find(M_coords);
I_list(144*(t-1)+1:144*(t-1)+144) = I_coords;
J_list(144*(t-1)+1:144*(t-1)+144) = J_coords;
X_list(144*(t-1)+1:144*(t-1)+144) = X_vals;
%M_global = sparse(M_global + C_t'*M_tk*C_t);
%Build load vector volume (hard-coded Nq=5)
B_t = build_load_volume(f_1,f_2,f_3,cpt,pt,vt);

%Build load vector surface (hard-coded Nq=4)

%Add to global load vector
B_global = B_global + C_t'*B_t;

%tru solution
for i=1:4
```

```
u_true(pos_t(i)) = 1-norm(pt(i,1:3))^2;
end
%Cube Boundary
%{
%Check if element has edges at boundary
[ind1,~,~] = find(edge==tri(t,1));
[ind2,~,~] = find(edge==tri(t,2));
[ind3,~,~] = find(edge==tri(t,3));
[ind4,~,~] = find(edge==tri(t,4));
%We now have 4 lists of row indices, possibly empty,
%each row index is where an edge with a node from the tetrahedron
%has been found
%It remains to find if there is a shared row index between
%at least 3 of those lists.
indlist = [ind1;ind2;ind3;ind4];
if(isempty(indlist)==0)
uniqueind = unique(indlist);
countmatrix = [uniqueind,histc(indlist,uniqueind)];
[indlength,~] = size(countmatrix);
edgelist_t = [];
for i=1:indlength
if countmatrix(i,2) >= 3
edgelist_t = [edgelist_t;countmatrix(i,1)];
end
end
edgelist_length = length(edgelist_t);
for j=1:edgelist_length
list_dirichlet = [list_dirichlet;edgelist_t(j)];
end
end
%}
disp(t);
end
%Parse Dirichlet Boundary (Cube)
%Ne = length(list_dirichlet);
M_global = sparse(I_list,J_list,X_list,3*N,3*N);
Ne = length(edge);
for s=1:Ne
edgepos = 3*(edge(s,:)-1)+1;
for i=1:3
for j=1:3
%Idea: set row to 0 and diagonal to 1 for m
%and to 0 for that index of the load vector, to force that u*1=0
M_global(edgepos(i)+(j-1),:) = 0;
%set column to zero, since that u is supposed to be 0
%maintaining symmetry
M_global(:,edgepos(i)+(j-1)) = 0;
%set diagonal to 1
M_global(edgepos(i)+(j-1),edgepos(i)+(j-1)) = 1;
end
B_global(edgepos(i):edgepos(i)+2) = 0;
end
end
U = M_global\B_global;
h = 1/N^3;
```

```
U_error = norm(U-u_true,inf)
%x_n = [x_n;log(h)];
%y_n = [y_n;log(U_error)];

% Begin iteration
itmax = 3;
%{
for it=1:itmax
for t=1:Nk
%Grab tetrahedron points
pt = [p(tri(t,:),:),ones(4,1)];

%solve for linear basis function coefficients
cpt = pt\ip;

%grab physical volume of tetrahedron
vt = (1/6)*abs(det([pt(1,1:3)'-pt(4,1:3)',pt(2,1:3)'-pt(4,1:3)',pt(3,1:3)'-pt(4,1:3)']));

%Build tensor of derivatives
tdiv = build_derivative_nonlocal(cpt);
%Construct node connection matrix
pos_t= 3*(tri(t,:)-1)+1;
%C_t is now sparse, constructed with triplets
C_t = build_nodeconnection_nonlocal(pos_t,N);

%Build local strain
strain_t = zeros(3,3);
for i=1:3
for j=1:3
strain_t(i,j) = tdiv(1,:,i,j)*C_t*U;
end
end
end
end
%}
```

Migrated Code: builders build derivative nonlocal

```
%Build tensor of derivatives for nonlocal method
function [tdiv] = build_derivative_nonlocal(cpt),
tdiv = zeros(1,12,3,3);
id = eye(3);
for i=1:3
for j=1:3
for m=1:4
tdiv(1,(3*(m-1)+1):(3*(m-1)+3),i,j) = 0.5*(cpt(j,m)*id(:,i)'+cpt(i,m)*id(:,j)')*id;
end
end
end
end
```

build nodeconnection nonlocal

```
%Build Node Connection Matrix for NonLocal Method
```

```
%Optimized with triplets storage for sparsity
%Author: Daniel Osen
function [C] = build_nodeconnection_nonlocal(pos_t,N),
%C = zeros(12,3*N);
I = 1:12;
J = zeros(1,12);
X = ones(1,12);
for i=1:12
%The indexing gives pos(1)+0, pos(1)+1, pos(1)+2,
%pos(2)+0,...,pos(4)+2 etc, referring to the running index.
%C(i,pos_t(floor((i-1)/3+1))+mod(i-1,3)) = 1;
J(i) = pos_t(floor((i-1)/3+1))+mod(i-1,3);
end
C = sparse(I,J,X,12,3*N,12);
end
```

build load volume

```
%Build load vector from volume integrals (5 quadrature points hard-coded)
function [B_t] = build_load_volume(f_1,f_2,f_3,cpt,pt,vt),
B_t = zeros(12,1);
z_q = [pt(1,1:3)' pt(2,1:3)' pt(3,1:3)' pt(4,1:3)']*[1/4 1/4 1/4 1/4;
1/2 1/6 1/6 1/6;
1/6 1/2 1/6 1/6;
1/6 1/6 1/2 1/6;
1/6 1/6 1/6 1/2]';
rho = [-4/5 9/20 9/20 9/20 9/20];
for j=1:4
I_1 = 0;
I_2 = 0;
I_3 = 0;
for i=1:5
I_1 = I_1 + rho(i)*f_1(z_q(:,i))*cpt(:,j)'*[z_q(:,i);1];
I_2 = I_2 + rho(i)*f_2(z_q(:,i))*cpt(:,j)'*[z_q(:,i);1];
I_3 = I_3 + rho(i)*f_3(z_q(:,i))*cpt(:,j)'*[z_q(:,i);1];
end
B_t(3*(j-1)+1) = I_1*vt;
B_t(3*(j-1)+2) = I_2*vt;
B_t(3*(j-1)+3) = I_3*vt;
end
end
```

build stiffness nonlocal

```
%Builds Constant Stiffness Matrix for NonLocal Method
function [M_tk] = build_stiffness_nonlocal(lambda,mu,tdiv,kdiv),
M_tk = zeros(12,12);
for i=1:3
M_tk = M_tk + (lambda+2*mu)*tdiv(1,:,i,i)'*kdiv(1,:,i,i);
for j=1:3
if i~=j
M_tk = M_tk + lambda*tdiv(1,:,i,i)'*kdiv(1,:,j,j);
M_tk = M_tk + 2*mu*tdiv(1,:,i,j)'*kdiv(1,:,i,j);
end
```

```
end
end
end
```

Python FEM solver

```
#Python FEM FENICS Solver 2016
#Daniel Osen
from dolfin import *
import numpy as np
import matplotlib.pyplot as plt


#To Run: python linear_elasticity.py

def my_solver(N):

# variables
mu = 1.0
lambda_  = 1.0

# Create mesh and define function space
mesh = BoxMesh(0.0,0.0,0.0,1.0,1.0,1.0,N, N, N)
# Create mesh from sphere and define function space
#mesh = Mesh(Sphere(Point(0.0,0.0,0.0),1.0),N)
#plot(mesh,interactive=True)
#mesh = SphereMesh()
V = VectorFunctionSpace(mesh, 'P', 1) #linear elements




# Define boundary conditions
tol = 1E-3

#box case
def clamped_boundary(x, on_boundary):

cube_left = near(x[0],0)
cube_right = near(x[0],1.0)
cube_front = near(x[1],0)
cube_back = near(x[1],1.0)
cube_bottom = near(x[2],0)
cube_top = near(x[2],1.0)

cube_any = cube_left or cube_right or cube_front or cube_back or cube_bottom or cube_top

return on_boundary and cube_any
#sphere case
'''
def clamped_boundary_sphere(x,on_boundary):
if sqrt(x[0]*x[0]+x[1]*x[1]+x[2]*x[2]) > 1 - tol:
return on_boundary and True
else:
return on_boundary and False
```

```
,,,

#define source term box case

string_f1 = "(-1)*((lambda_+2*mu)*2*(x[1]*x[1]-x[1])*(x[2]*x[2]-x[2])
+mu*2*(x[0]*x[0]-x[0])*(x[2]*x[2]-x[2])
+mu*2*(x[0]*x[0]-x[0])*(x[2]*x[2]-x[2])
+(lambda_+mu)*(2*x[0]-1)*(2*x[1]-1)*(x[2]*x[2]-x[2])
+(lambda_+mu)*(2*x[0]-1)*(2*x[2]-1)*(x[1]*x[1]-x[1]))"
string_f2 = "(-1)*((lambda_+2*mu)*2*(x[0]*x[0]-x[0])*(x[2]*x[2]-x[2])
+mu*2*(x[1]*x[1]-x[1])*(x[2]*x[2]-x[2])
+mu*2*(x[1]*x[1]-x[1])*(x[2]*x[2]-x[2])
+(lambda_+mu)*(2*x[1]-1)*(2*x[0]-1)*(x[2]*x[2]-x[2])
+(lambda_+mu)*(2*x[1]-1)*(2*x[2]-1)*(x[0]*x[0]-x[0]))"
string_f3 = "(-1)*((lambda_+2*mu)*2*(x[1]*x[1]-x[1])*(x[0]*x[0]-x[0])
+mu*2*(x[2]*x[2]-x[2])*(x[0]*x[0]-x[0])
+mu*2*(x[2]*x[2]-x[2])*(x[0]*x[0]-x[0])
+(lambda_+mu)*(2*x[2]-1)*(2*x[1]-1)*(x[0]*x[0]-x[0])
+(lambda_+mu)*(2*x[2]-1)*(2*x[0]-1)*(x[1]*x[1]-x[1]))"

f = Expression((string_f1,string_f2,string_f3),lambda_ = lambda_, mu=mu)
#sphere case

#f = Constant((2*lambda_+8*mu,0.0,0.0))




#initialize boundary
bc = DirichletBC(V, Constant((0, 0, 0)), clamped_boundary)

# Define strain and stress
def epsilon(u):
return 0.5*(nabla_grad(u) + nabla_grad(u).T)
#return sym(nabla_grad(u))
def sigma(u):
return lambda_*nabla_div(u)*Identity(d) + 2*mu*epsilon(u)
# Define variational problem
u = TrialFunction(V)
d = u.geometric_dimension() # no of space dim
v = TestFunction(V)
T = Constant((0, 0, 0))
a = inner(sigma(u), epsilon(v))*dx

string_exp = "(x[0]*x[0]-x[0])*(x[1]*x[1]-x[1])*(x[2]*x[2]-x[2])"
u_exact = Expression((string_exp,string_exp,string_exp),domain=mesh)


L = dot(f, v)*dx + dot(T, v)*ds
#the ds part is zero for full dirichlet boundary,
#however, since it may be inexact we specify a zero traction

# Compute solution
u = Function(V)
solve(a == L, u, bc)
```

```
# Compute Error
#box case
#string_exp = "(x[0]*x[0]-x[0])*(x[1]*x[1]-x[1])*(x[2]*x[2]-x[2])"
#u_exact = Expression((string_exp,string_exp,string_exp))

#sphere case
#string_exp = "1-pow( x[0]*x[0] + x[1]*x[1] + x[2]*x[2] ,0.5)"
#u_exact = Expression((string_exp,"0.0","0.0"))

#L2-Norm (u_exact interpolated as third degree over mesh?)
error_L2 = errornorm(u_exact,u,'L2')

#L-inf norm comparison over mesh points
vertex_values_u_exact = u_exact.compute_vertex_values(mesh)
vertex_values_u = u.compute_vertex_values(mesh)
error_max = np.max(np.abs(vertex_values_u-vertex_values_u_exact))

# Plot solution
#plot(u,title='displacement',mode='displacement',interactive=True)
#plot(u,title='displacement',interactive=True)


print("L_Inf = {}, L_2 = {}, N = {}".format(error_max,error_L2,N))

return error_max,error_L2,N

if __name__ == "__main__":

#Do computation:
maxit = 4
nvals = np.zeros(maxit)
l_inf = np.zeros(maxit)
l_2 = np.zeros(maxit)
for i in range(0,maxit):
l_inf[i],l_2[i],nvals[i] = my_solver(2**(i+1))

r = np.zeros(maxit)
g = np.zeros(maxit)
for i in range(1,maxit):
r[i-1] = np.log(l_2[i]/l_2[i-1])/np.log(nvals[i-1]/nvals[i])
g[i-1] = np.log(l_inf[i]/l_inf[i-1])/np.log(nvals[i-1]/nvals[i])
print("convergence rate L2 = {}".format(r))
print("convergence rate Linf = {}".format(g))
'''
```

Bibliography

[1] Susanne C. Brenner and Scott L. Ridgeway. The mathematical theory of finite element methods, 3rd edition. *Texts in applied mathematics, Springer*, 2008.

[2] George Strang, Gilbert: Fix. An analysis of the finite element method. *Prentice Hall*, 1973.

[3] Philippe G. Ciarlet. The finite element method for elliptic problems. *Studies in mathematics and its applications, Pierre and Marie Curie University, North-Holland Publishing Company*, 1978.

[4] Alfio Quarteroni. Numerical models for differential problems, second edition. *Modelling, Simulation & Applications, Springer*, 2014.

[5] Theodor Krauthammer. Accuracy of the finite element method near a curved boundary. *Computers and Structures, Department of Civil Engineering, University of Illinois at Urbana-Champaign, IL 61803, U.S.A.*, 1979.

[6] Axel Kaselow. The stress sensitivity approach: Theory and application. *Zur Erlangung des des akademischen Grades Doktor der Naturwissenschaften am Institut fur Geologische Wissenschaften der Freien Universitat Berlin, in der Fachrichtung Geophysik genehmigte Dissertation*, 2004.

[7] Castrenze Polizzotto. Nonlocal elasticity and related variational principles. *International journal of Solids and Structures 38, Dipartimento di Ingegneria Strutturale & Geotecnica, Università di Palermo, Vale delle Scienze, 90128 Palermo, Italy*, 2001.

[8] J.E. Akin. Finite element analysis with error estimators, an introduction to the fem and adaptive error analysis for engineering students. *George R. Brown, School of Engineering*, 2005.

[9] Rashid Abu Al-Rub George Z. Voyiadjis. Determination of the material intrinsic length scale of gradient plasticity theory. *Department of Civil and Environmental Engineering, Louisiana State University*, 2004.