A Tensor Algebraic Approach to Reduced-Order Stable Neo-Hookean Deformations

Gillian P. Reyes

1 Introduction

The goal of this project is to document each step taken so far in creating a reduced-order simulation of the Stable Neo-Hookean approach to deformable solids [1]. The approach is identical to the StVK reduced-order simulation except in calculating internal force - the Finite Element Method is used to discretize partial differential equations of solid continuum mechanics, allowing the motion to be described through the Euler-Lagrange equation,

$$\boldsymbol{M}\ddot{\boldsymbol{u}} + \boldsymbol{D}(\boldsymbol{u},\dot{\boldsymbol{u}}) + \boldsymbol{R}(\boldsymbol{u}) = \boldsymbol{f} \tag{1}$$

where $\boldsymbol{M} \in \mathbb{R}^{2n\times 2n}$ is the mass matrix, $\boldsymbol{D}$ is the damping force, $\boldsymbol{R}$ is internal force, and $\boldsymbol{f}$ is external force [2]. $\boldsymbol{u} \in \mathbb{R}^{2n}$ is the displacement vector. This equation is then reduced by introducing a time-independent matrix, $\boldsymbol{U} \in \mathbb{R}^{2n\times r}$, specifying a basis of some r-dimensional linear subspace. After reduction, the equation becomes:

$$\tilde{\boldsymbol{M}}\ddot{\boldsymbol{q}} + \tilde{\boldsymbol{D}}(\boldsymbol{q},\dot{\boldsymbol{q}}) + \tilde{\boldsymbol{R}}(\boldsymbol{q}) = \tilde{\boldsymbol{f}} \tag{2}$$

where $\boldsymbol{q}, \tilde{\boldsymbol{D}}(q,\dot{q}), \tilde{\boldsymbol{R}}(q), \tilde{\boldsymbol{f}} \in \mathbb{R}^r$, $\tilde{M} \in \mathbb{R}^{r\times r}$, and each can be found through the equations:

$$\boldsymbol{u} = \boldsymbol{U}\boldsymbol{q} \tag{3}$$

$$\tilde{\boldsymbol{M}} = \boldsymbol{U}^T\boldsymbol{M}\boldsymbol{U} \tag{4}$$

$$\tilde{\boldsymbol{D}}(\boldsymbol{q},\dot{\boldsymbol{q}}) = \boldsymbol{U}^T\boldsymbol{D}(\boldsymbol{U}\boldsymbol{q},\boldsymbol{U}\dot{\boldsymbol{q}}) \tag{5}$$

$$\tilde{\boldsymbol{R}}(\boldsymbol{q}) = \boldsymbol{U}^T\boldsymbol{D}(\boldsymbol{U}\boldsymbol{q}) \tag{6}$$

$$\tilde{\boldsymbol{f}} = \boldsymbol{U}^T\boldsymbol{f} \tag{7}$$

$$\tilde{\boldsymbol{K}} = \boldsymbol{U}^T\boldsymbol{K}(\boldsymbol{U}\boldsymbol{q})\boldsymbol{U} \tag{8}$$

Using these equations to reduce the problem speeds up the computation, but everything can be sped up further by treating the calculation of $\boldsymbol{R}$ as cubic polynomial and $\boldsymbol{K}$ as a quadratic polynomial. Then, constant coefficients can be precomputed for these equations, such that

$$\tilde{\boldsymbol{R}}(\boldsymbol{q}) = \boldsymbol{U}^T\boldsymbol{R}(\boldsymbol{q}) = \boldsymbol{P}^i\boldsymbol{q}_i + \boldsymbol{Q}^{ij}\boldsymbol{q}_i\boldsymbol{q}_j + \boldsymbol{S}^{ijk}\boldsymbol{q}_i\boldsymbol{q}_j\boldsymbol{q}_k \tag{9}$$

$$\tilde{\boldsymbol{K}}(\boldsymbol{q}) = \frac{\partial\tilde{\boldsymbol{R}}(\boldsymbol{q})}{\partial\boldsymbol{q}_l} = \boldsymbol{P}^l + (\boldsymbol{Q}^{li} + \boldsymbol{Q}^{il})\boldsymbol{q}_i + (\boldsymbol{S}^{ijl} + \boldsymbol{S}^{ilj} + \boldsymbol{S}^{lij})\boldsymbol{q}_i\boldsymbol{q}_j \tag{10}$$

The reduced Euler-Lagrange equation of motion can then be solved using a Newmark integrator, animating large deformations of deformable models more efficiently than previously possible.

This paper will start from the very beginning of the process, i.e. creating triangle objects to form a triangle mesh for any specified model, and walk through every step until an actual animation is created at the end. OpenGL is used for drawing and displaying images, and C++ was used for the rest of the implementation.

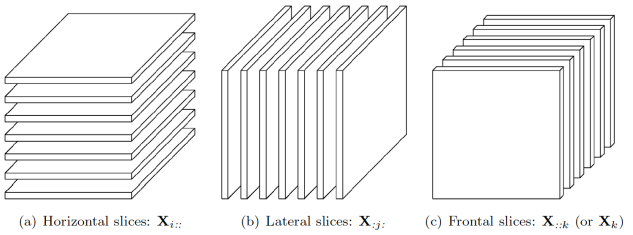

Fig. 1. A visual representation of a third order tensor. It can be imagined in three distinct ways, depending on how you index it. This image is from [3]

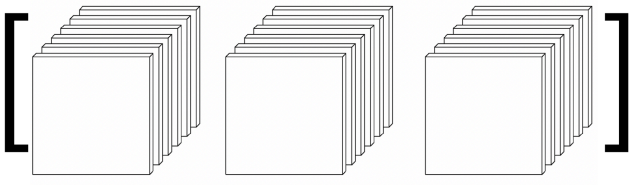

Fig. 2. A visual representation of a fourth order tensor. In code, it is treated as a vector of third order tensors.

2 An Overview of Tensors

Before going into the details of how to implement reduced-order St. Venant-Kirchhoff deformable models, it's important to review tensors, as they show up in many places throughout this paper. The following explanation is influenced by [3], and contains extra findings that emerged through implementation.

A **tensor** is a multidimensional array. For example, a matrix is a second order tensor, and a vector is a first order tensor. An Nth order tensor is an array of N dimensions, and thus indexed using N indices.

Figure 1 shows a visual representation of a third order tensor; it can be imagined as a vector of matrices, where each square slice represents an individual matrix. Figure 2 shows a visual representation of a fourth order tensor, or a vector of third order tensors. Fourth order tensors are the highest dimension tensor needed for this paper, so no other tensor visuals are included.

There are three different kinds of tensor multiplication that we will use throughout this paper.

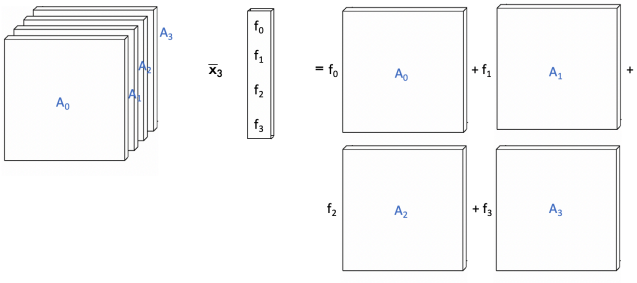

Fig. 3. A visual representation of a 3-mode vector product, where $\boldsymbol{A} \in \mathbb{R}^{n \times n \times 4}$ and $\boldsymbol{f} \in \mathbb{R}^4$

2.1 Scalar Multiplication

This works exactly as it seems like it would. For $a \in \mathbb{R}$, $\boldsymbol{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$,

$$(a\boldsymbol{X})_{i_1,i_2,\ldots,i_n} = a \cdot x_{i_1,i_2,\ldots,i_n} \tag{11}$$

2.2 n-Mode Vector Product

When a tensor is multiplied by a vector, its order reduces by 1. An n-mode vector product of a tensor $\boldsymbol{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ with a vector $\boldsymbol{v} \in \mathbb{R}^{I_n}$ is denoted by $\boldsymbol{X} \bar{\times}_n \boldsymbol{v}$. Element-wise,

$$(\boldsymbol{X} \bar{\times}_n \boldsymbol{v})_{i_1,\ldots,i_{n-1},i_{n+1},\ldots,i_N} = \sum_{i_n=1}^{I_n} x_{i_1,i_2,\ldots,i_N} \cdot v_{i_n} \tag{12}$$

So the resulting tensor $(\boldsymbol{X} \bar{\times}_n \boldsymbol{v}) \in \mathbb{R}^{I_1 \times \ldots \times I_{n-1} \times I_{n+1} \times \ldots \times I_N}$. A visual representation of a 3-mode vector product can be seen in Figure 3. For n-mode vector products, precedence matters, since it changes which dimension of the tensor is collapsed. Therefore, if $m < n$,

$$\boldsymbol{X} \bar{\times}_m \boldsymbol{a} \bar{\times}_n \boldsymbol{b} = (\boldsymbol{X} \bar{\times}_m \boldsymbol{a}) \bar{\times}_{n-1} \boldsymbol{b} = (\boldsymbol{X} \bar{\times}_n \boldsymbol{b}) \bar{\times}_m \boldsymbol{a} \tag{13}$$

This is an important rule that will be used later.

2.3 n-Mode Matrix Product

When a tensor is multiplied by a matrix, one of its dimensions changes. An n-mode matrix product of a tensor

$\boldsymbol{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ with a matrix $\boldsymbol{U} \in \mathbb{R}^{J \times I_n}$ is denoted by $\boldsymbol{X} \times_n \boldsymbol{U}$. Element-wise, this multiplication results in:

$$(\boldsymbol{X} \times_n \boldsymbol{U})_{i_1,\ldots,i_{n-1},j,i_{n+1},\ldots,i_N} = \sum_{i_n=1}^{I_n} x_{i_1,i_2,\ldots,i_N} \cdot u_{j,i_n} \tag{14}$$

and the resulting tensor $(\boldsymbol{X} \times_n \boldsymbol{U}) \in \mathbb{R}^{I_1 \times \ldots \times I_{n-1} \times J \times I_{n+1} \times \ldots \times I_N}$.

If $n \neq m$, then

$$\boldsymbol{X} \times_n \boldsymbol{A} \times_m \boldsymbol{B} = \boldsymbol{X} \times_m \boldsymbol{B} \times_n \boldsymbol{A} \tag{15}$$

If the modes are the same, then

$$\boldsymbol{X} \times_n \boldsymbol{A} \times_n \boldsymbol{B} = \boldsymbol{X} \times_n (\boldsymbol{B}\boldsymbol{A}) \tag{16}$$

These two rules will also be used later. Another important rule which will be used later is, for a tensor $\boldsymbol{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$, a matrix $\boldsymbol{A} \in \mathbb{R}^{I_n, J}$, and a vector $\boldsymbol{b} \in \mathbb{R}^J$,

$$\boldsymbol{X} \bar{\times}_n (\boldsymbol{A}\boldsymbol{b}) = \boldsymbol{X} \times_n \boldsymbol{A}^{\boldsymbol{T}} \bar{\times}_n \boldsymbol{b} \tag{17}$$

3 Creating the Triangles

When modeling solids using a discrete method, the model is represented by a mesh of polygons. In this paper, triangles were used, so all results of this paper are based on triangle calculations. The implementation of our triangle object can be found in **TRIANGLE.cpp** and **TRIANGLE.h**.

In the code for this paper, a triangle object was created for cleaner implementation. Important variables to keep track of are:

1. a vector of rest vertices. This should stay constant throughout the simulation. In our code, this variable is **_restPose**.
2. a vector of actual positions, which update as the whole mesh moves. In our code, this variable is **_vertices**.
3. a material model of some sort, to keep track of λ and μ values for force calculations. In our code, this variable is **_material**.
4. rest area; this is used in calculating the internal force and stiffness matrix. In our code, this isn't stored as a variable, but can be calculated with the function **restArea()**.
5. precomputed coefficients for the cubic polynomial interpretation of internal force and quadratic polynomial for the tangent stiffness matrix. This will be discussed in a later section. In our code, the variables for the cubic polynomial terms are **_cubicCoef** for the cubic term, **_cubic2** for the quadratic term, and **_linearCoef** for the linear term. For the quadratic polynomial, the terms are **_quadraticCoef** for the quadratic term, **_quad2** for the linear term, and **_constCoef** for the constant term.

In addition to storing variables, there are a few useful functions that can go into this object as well:

1. getter/setter functions for all private variables.
2. a function that computes $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{u}}$: **pFpu()**.
3. a function that computes the deformation gradient, $\boldsymbol{F}$: **computeF()**.
4. a function to compute the internal force for the triangle given the deformation gradient: **computeForceVector()**.
5. a function to compute the force Jacobian for the triangle given the deformation gradient: **computeForceJacobian()**.

The last three functions are not necessary for the cubic polynomial approach, but are useful for debugging purposes. All terms used in this section will be explained in the following subsections, which will go into more details of implementation.

3.1 Computing the Deformation Gradient

When a solid is deformed, the **Deformation Gradient**, $\boldsymbol{F} \in \mathbb{R}^{2\times 2}$, is the linear mapping from a rest vertex to its deformed position (without translation). This mapping can be used in the calculation of strain energy, so it's an important value to calculate if the cubic polynomial approach is not being used.

The deformation gradient can be found using the equation:

$$\boldsymbol{F} = \boldsymbol{D_s} \boldsymbol{D_m}^{-1} = \begin{bmatrix} (x_2 - x_0) & (x_4 - x_0) \\ (x_3 - x_1) & (x_5 - x_1) \end{bmatrix} \boldsymbol{D_m}^{-1} = \begin{bmatrix} f_0 & f_2 \\ f_1 & f_3 \end{bmatrix} \tag{18}$$

where $\boldsymbol{D_s}$ is the spatial matrix made of the deformed vertices, $\boldsymbol{D_m}$ is the material matrix made of the rest vertices, and the three vertices of the triangle are $(x_0, x_1), (x_2, x_3), (x_4, x_5)$. Since the rest vertices stay constant throughout the simulation, $\boldsymbol{D_m}^{-1}$ is constant. As stated above, this calculation can be found in **TRIANGLE.cpp** within the function **computeF()**.

3.2 Implementing Stable Neo-Hookean

This section is not necessary for the cubic polynomial approach, but it's useful as a debugging tool. In the code for this paper, the implementation of Stable Neo-Hookean is tied to the material of the triangle, and is called by the functions that calculate internal force and the force Jacobian for an individual triangle. The implementation of the following model can be found in the file **NEOHOOKEAN.cpp**.

A Stable Neo-Hookean deformable model is defined by the Stable Neo-Hookean strain energy:

$$\psi = \frac{\mu}{2}(\text{tr}(\boldsymbol{F}^T\boldsymbol{F}) - 3) + \frac{\lambda}{2}(\det(\boldsymbol{F}) - 1 - \frac{\mu}{2})^2 \tag{19}$$

where λ and μ are Lamé coefficients.

In computing the internal force of a single triangle, the Piola-Kirchhoff stress tensor (the derivative of the strain energy by the deformation gradient) is used. The Piola-Kirchhoff stress tensor is:

$$\frac{\partial \psi}{\partial \boldsymbol{F}} = \mu\boldsymbol{F} + (\lambda\det(\boldsymbol{F}) - \lambda - \mu)\begin{bmatrix} f_3 & -f_1 \\ -f_2 & f_0 \end{bmatrix} \tag{20}$$

In computing the force Jacobian of a single triangle, the energy Hessian is used:

$$\begin{aligned}\frac{\partial^2 \psi}{\partial \boldsymbol{F}^2} = \mu I + \frac{\partial}{\partial F_i}(\lambda\det(\boldsymbol{F})\begin{bmatrix} f_3 & -f_1 \\ -f_2 & f_0 \end{bmatrix}) \\ -(\lambda+\mu)\frac{\partial}{\partial F_i}\begin{bmatrix} f_3 & -f_1 \\ -f_2 & f_0 \end{bmatrix}\end{aligned} \tag{21}$$

3.3 Computing dF/dx

Before we can calculate internal force and the force jacobian, we need to find the derivative of $\boldsymbol{F}$ in terms of $\boldsymbol{x}$. Remember that $\boldsymbol{F} \in \mathbb{R}^{2\times 2}$ and $\boldsymbol{x} \in \mathbb{R}^6$. This means that

$$\begin{aligned}\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} &= \begin{bmatrix} \frac{\partial \boldsymbol{F}}{\partial x_0} & \frac{\partial \boldsymbol{F}}{\partial x_1} & \frac{\partial \boldsymbol{F}}{\partial x_2} & \frac{\partial \boldsymbol{F}}{\partial x_3} & \frac{\partial \boldsymbol{F}}{\partial x_4} & \frac{\partial \boldsymbol{F}}{\partial x_5} \end{bmatrix} \\ &= \begin{bmatrix} \begin{bmatrix} -1 & -1 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ -1 & -1 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \end{bmatrix}\end{aligned} \tag{22}$$

This was derived using equation 18 for $\boldsymbol{F}$. Notice that this is a vector of matrices, i.e. a third order tensor. While we can use this as a third order tensor, it's easier to vectorize it and treat it as a matrix.

3.3.1 Flattening Matrices and Tensors

Flattening a tensor reduces it into a matrix, and flattening a matrix reduces it into a vector (also called vectorizing). To vectorize a matrix, we append the next column onto the bottom of the previous. So, as an example, if

$$\boldsymbol{A} = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

then

$$\text{vec}(\boldsymbol{A}) = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

To flatten a tensor, we do this process on each level - so if we have a fourth order tensor, or a matrix of matrices, we first flatten the outer matrix into a vector of matrices, then vectorize the matrices inside. This step, which is the same for both third and fourth order tensors, can be done as shown below:

$$\boldsymbol{A} = \begin{bmatrix} \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} & \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} \end{bmatrix}$$

then

$$\text{vec}(\boldsymbol{A}) = \begin{bmatrix} \text{vec}\left(\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}\right) & \text{vec}\left(\begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix}\right) \end{bmatrix} = \begin{bmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{bmatrix}$$

3.4 Computing Internal Force and the Force Jacobian (The Non-Polynomial Way)

This section is unnecessary for the cubic polynomial approach, but as with all the other sections, it's useful for debugging purposes. The internal force inside of the triangle can be computed using the equation

$$\boldsymbol{f} = \frac{\partial \psi}{\partial \boldsymbol{u}} = \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right)^T \cdot \text{vec}\left(\frac{\partial \psi}{\partial \boldsymbol{F}}\right) \tag{23}$$

where the terms are defined by equations 22 and 20, respectively. This is implemented in the **computeForceVector()** function in **TRIANGLE.cpp**. The force Jacobian for each triangle can be computed using the equation

$$\frac{\partial^2 \psi}{\partial \boldsymbol{u}^2} = \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right)^T \cdot \frac{\partial^2 \psi}{\partial \boldsymbol{F}^2} \cdot \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right) \tag{24}$$

where the terms are defined by equations 22 and 21. This is implemented in the **computeForceJacobian()** function in **TRIANGLE.cpp** Notice how $\boldsymbol{x}$, or current vertex placement, and $\boldsymbol{u}$, vertex *displacement*, are used interchangeably here. In general, this is possible because the internal force at rest position $\bar{\boldsymbol{x}}$ is 0, and $\boldsymbol{x} = \bar{\boldsymbol{x}} + \boldsymbol{u}$. However, it is important to note that these will **not** be interchangeable in the cubic polynomial approach.

4 Creating the Triangle Mesh

Now that a triangle object has been established, a triangle mesh can be built. In the code for this paper, the triangle mesh is a separate object from the individual triangles, and it stores all of the triangles in a vector. Everything can be found within the file **TRIANGLE_MESH.cpp**. When constructing the triangle mesh, it may be useful to store a map of each vertex to its global position, so that future calculations are easier. In this implementation, the following were stored as private variables:

1. a vector of triangles, i.e. the triangle mesh: **_triangles**.
2. a vertex to global index mapping: **_vertexToIndex**.
3. a vector of vertex placements: **_vertices**.
4. a vector of vertex rest positions: **_restVertices**.
5. a vector of vertex displacements: **_u**.
6. a vector of vertex indices representing which vertices are **constrained**: **_constrainedVertices**.
7. a vector of vertex indices representing which vertices are **unconstrained**: **_unconstrainedVertices**.
8. a vector for internal force: **_f**.
9. a vector for external force: **_fExternal**.
10. a mass matrix: **_mass**.
11. a vector for velocity and a vector for acceleration: unreduced velocity is **_velocity**, reduced velocity is **_rv**; unreduced acceleration is **_acceleration**, reduced acceleration is **_ra**.
12. the basis reduction matrix: **_U**.
13. a vector for the reduced basis vector: **_q**.

5 Implementing the Euler-Lagrange Equation of Motion

After setting up the triangle mesh and applying some kind of external force, the system is animated by solving the Euler-Lagrange equation of motion. By solving equation 1, the displacements for the next time step can be found and added to the vertex placements. In order to do this, each term of the equation must be calculated.

5.1 Implementing Internal Force

While this step is not necessary for the cubic polynomial approach, the methods are similar. In the previous section, the individual internal force for each triangle was calculated using equation 23. To calculate the global internal force, $\boldsymbol{R}(\boldsymbol{u}) \in \mathbb{R}^{2n}$, all of the individual internal forces are added up. This can be done by looping through every triangle, then looping through every vertex of the triangle. For each vertex, find where it belongs in the global context, and, if it is not constrained, add the force to the global vector. Here's an example through pseudocode below:

```
1:  function GLOBALINTERNALFORCE(null)
2:      global_vector.setZero()
3:      for triangle in triangles do
4:          force ← triangle.force()
5:          for vertex in triangle vertices do
6:              if vertex is unconstrained then
7:                  i ← global_index(vertex)
8:                  add to global vector
9:              end if
10:         end for
11:     end for
12: end function
```

The actual implementation for this can be found in **TRIANGLE_MESH.cpp** in the function **computeUnprecomputedMaterialForces()**.

5.2 Implementing the Tangent Stiffness Matrix

The process here is similar to the process of calculating the global internal force. In the previous section, the individual force Jacobian for each triangle was calculated using equation 24. To calculate the global stiffness matrix, $\boldsymbol{K}(\boldsymbol{u}) \in \mathbb{R}^{2n \times 2n}$, all of the individual force Jacobians are added up. This can be done by looping through every triangle, then looping through every vertex of the triangle twice. For each vertex, find where it belongs in the global context, and, if it is not constrained, add the force to the global stiffness matrix. Here's an example through pseudocode below:

```
1:  function GLOBALSTIFFNESSMATRIX(null)
2:      global_stiffness.setZero()
3:      for triangle in triangles do
4:          jacobian ← triangle.jacobian()
5:          for vertex in triangle vertices do
6:              if vertex is unconstrained then
7:                  i ← global_index(vertex)
8:                  for vertex_2 in triangle vertices do
9:                      if vertex_2 is unconstrained then
10:                         j ← global_index(vertex_2)
11:                         add to global stiffness matrix
```

```
            end if
        end for
    end if
  end for
end function
```

The actual implementation for this can be found in **TRIANGLE_MESH.cpp** in the function **computeUnprecomputedStiffnessMatrix(MATRIX& K)**.

5.3 Implementing the Mass Matrix

The mass matrix does not change over time, so this can be initialized outside of the motion function. How mass is implemented is up to the creator. The mass matrix for this implementation can be found in the function **setMassMatrix(bool reduction)**.

5.4 Implementing the Damping Matrix

The damping matrix is calculated using the equation

$$\boldsymbol{D}(\boldsymbol{u}) = (\alpha \boldsymbol{M} - \beta \boldsymbol{K}(\boldsymbol{u})) \tag{25}$$

After computing the global stiffness matrix, this should be straightforward. The alpha and beta values are constant values that are up to the creator, depending on how strong the desired force is. In this implementation, $\alpha = 0.01$ and $\beta = 0.02$, and the calculation of the damping matrix can be found within the function **stepMotion(float dt, const VEC2& outerForce, bool reduced)**.

5.5 Implementing the Newmark Integrator

After calculating all of the individual terms in the Euler-Lagrange equation of motion, the equation can be solved for a given time-step using an implicit Newmark integrator. Implicit Newmark integrators are second-order accurate, and they only require one step, instead of having to solve the equation in a few iterations. While the pseudocode below is written for unreduced displacements, this iteration stays exactly the same when $\boldsymbol{u}$ is replaced with $\boldsymbol{q}$.

function NEWMARKINTEGRATOR(*null*)
  $\boldsymbol{u}_{i+1} \leftarrow \boldsymbol{u}_i$
  Evaluate internal forces $\boldsymbol{R}(\boldsymbol{u}_i)$
  Evaluate the stiffness matrix $\boldsymbol{K}(\boldsymbol{u}_i)$
  Calculate the damping matrix, $\boldsymbol{D}(\boldsymbol{u}_i) = \alpha \boldsymbol{M} - \beta \boldsymbol{K}$
  Calculate the system matrix, $\boldsymbol{A} = \alpha_1 \boldsymbol{M} + \alpha_4 \boldsymbol{D}(\boldsymbol{u}_i) - \boldsymbol{K}$
  residual $\leftarrow (\alpha_3 \boldsymbol{M} - \alpha_6 \boldsymbol{D})\ddot{\boldsymbol{u}}_i + (\alpha_2 \boldsymbol{M} - \alpha_5 \boldsymbol{D})\dot{\boldsymbol{u}}_i + \boldsymbol{R}(\boldsymbol{u}_i) + \boldsymbol{f}_{i+1}$
  $\Delta \boldsymbol{u} \leftarrow \boldsymbol{A}^{-1} \cdot$ residual
  $\boldsymbol{u}_{i+1} \leftarrow \boldsymbol{u}_{i+1} + \Delta \boldsymbol{u}$
  $\dot{\boldsymbol{u}}_{i+1} \leftarrow \alpha_4(\boldsymbol{u}_{i+1} - \boldsymbol{u}_i) + \alpha_5 \dot{\boldsymbol{u}}_i + \alpha_6 \ddot{\boldsymbol{u}}_i$
  $\ddot{\boldsymbol{u}}_{i+1} \leftarrow \alpha_1(\boldsymbol{u}_{i+1} - \boldsymbol{u}_i) - \alpha_2 \dot{\boldsymbol{u}}_i - \alpha_3 \ddot{\boldsymbol{u}}_i$
end function

The alpha constants in this integrator are determined by the equations

$$\alpha_1 = \frac{1}{\tilde{\beta}(\Delta t)^2}, \alpha_2 = \frac{1}{\tilde{\beta}\Delta t}, \alpha_3 = \frac{1-2\tilde{\beta}}{2\tilde{\beta}}, \alpha_4 = \frac{\tilde{\gamma}}{\tilde{\beta}\Delta t},$$
$$\alpha_5 = 1 - \frac{\tilde{\gamma}}{\tilde{\beta}}, \alpha_6 = \left(1 - \frac{\tilde{\gamma}}{2\tilde{\beta}}\right)\Delta t$$

Where $0 \leq \tilde{\beta} \leq 0.5$ and $0 \leq \tilde{\gamma} \leq 1$. We chose $\tilde{\beta} = 0.25$ and $\tilde{\gamma} = 0.5$ for this implementation, but Barbič and James chose $\tilde{\beta} = 0$ and $\tilde{\gamma} = 0.5$.

6 Generating Precomputed Coefficients - REWRITE FOR NEOHOOKEAN

At this point in the paper, a fully function simulator of Stable Neo-Hookean deformable models can be created; however, they are still unreduced and lack any precomputation. If pre-computation does not interest you, feel free to skip to the section titled **Generating a Deformation Basis** - after creating a deformation basis, reducing everything is as simple as following equations 3 - 8.

To fully understand everything that happens in this step, we will derived the precomputed constant coefficients before reducing the equation. In order to have the ability to precompute constant coefficients, all variables must be extractable. In the case of unreduced Stable Neo-Hookean models, this means that we should be able to separate vertex positions, $\boldsymbol{x}$, out of the internal force and stiffness matrix calculation. Again, for simplicity, we began by separating $\boldsymbol{x}$ from the internal force equation 23 and force Jacobian equation 24 for an individual triangle. The global constants can then be created in two different ways. Method 1 follows a similar implementation as the creation of the non-polynomial stiffness matrix, and runs faster, but has significant memory issues and is no longer in the code. Method 2 creates the constants by adding all of these smaller triangle constants into their proper position in the global context. This runs much slower, but is memory efficient.

6.1 Coefficients for the Tangent Stiffness Matrix

Looking at equation 21, there is one quadratic term, and one constant matrix, $\mu \boldsymbol{I} - (\lambda + \mu)\frac{\partial}{\partial \boldsymbol{F_i}} \begin{bmatrix} f_3 & -f_1 \\ -f_2 & f_0 \end{bmatrix}$. From this, we can derive the constant coefficient term from equation 24:

$$\boldsymbol{Q} = \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right)^T \left(\mu \boldsymbol{I} - (\lambda+\mu)\frac{\partial}{\partial \boldsymbol{F}_i}\begin{bmatrix} f_3 & -f_1 \\ -f_2 & f_0 \end{bmatrix}\right) \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right) \tag{26}$$

The quadratic term is harder. Expanded, the quadratic term, $\lambda \det(\boldsymbol{F}) \begin{bmatrix} f_3 & -f_1 \\ -f_2 & f_0 \end{bmatrix}$ is:

$$\lambda \det(\boldsymbol{F}) \begin{bmatrix} f_3 & -f_1 \\ -f_2 & f_0 \end{bmatrix} =$$
$$\lambda \begin{bmatrix} f_3^2 & -f_2 f_3 & -f_1 f_3 & 2f_0 f_3 - f_1 f_2 \\ -f_2 f_3 & f_2^2 & 2f_1 f_2 - f_0 f_3 & -f_0 f_2 \\ -f_1 f_3 & 2f_1 f_2 - f_0 f_3 & f_1^2 & -f_0 f_1 \\ 2f_0 f_3 - f_1 f_2 & -f_0 f_2 & -f_0 f_1 & f_0^2 \end{bmatrix}$$

In order to separate out all of the $\boldsymbol{x}$'s, we can first separate all of the $\boldsymbol{f}$ terms. This can be done using tensors. By working backwards from the matrices above, a fourth order tensor, $\boldsymbol{C_f}$, can be derived, such that

$$\boldsymbol{C_f} = \left[\begin{bmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&1 \end{bmatrix} \begin{bmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&-1 \\ 0&0&0&0 \end{bmatrix} \begin{bmatrix} 0&0&0&0 \\ 0&0&0&-1 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix} \begin{bmatrix} 0&0&0&2 \\ 0&0&-1&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix}\right.$$
$$\begin{bmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&-1&0 \end{bmatrix} \begin{bmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&0&1&0 \\ 0&0&0&0 \end{bmatrix} \begin{bmatrix} 0&0&0&0 \\ 0&0&2&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix} \begin{bmatrix} 0&0&-1&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix}$$
$$\begin{bmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&-1&0&0 \end{bmatrix} \begin{bmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&2&0&0 \\ -1&0&0&0 \end{bmatrix} \begin{bmatrix} 0&0&0&0 \\ 0&1&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix} \begin{bmatrix} 0&-1&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix}$$
$$\begin{bmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&-1&0&0 \\ 2&0&0&0 \end{bmatrix} \begin{bmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ -1&0&0&0 \end{bmatrix} \begin{bmatrix} 0&0&0&0 \\ -1&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix} \left.\begin{bmatrix} 1&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix}\right] \tag{27}$$

And

$$\lambda \det(\boldsymbol{F}) \begin{bmatrix} f_3 & -f_1 \\ -f_2 & f_0 \end{bmatrix} = \lambda \boldsymbol{C_f} \bar{\times}_4 \text{vec}(\boldsymbol{F}) \bar{\times}_3 \text{vec}(\boldsymbol{F}) \tag{28}$$

This still only separates $\boldsymbol{F}$, not $\boldsymbol{x}$. However, since $\boldsymbol{F} = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\boldsymbol{x}$, we can rearrange this equation using our tensor product rules (equations 13-17) so that

$$\begin{aligned} \boldsymbol{C_f} \bar{\times}_4 \text{vec}(\boldsymbol{F}) \bar{\times}_3 \text{vec}(\boldsymbol{F}) &= \boldsymbol{C_f} \bar{\times}_4 \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\boldsymbol{x}\right) \bar{\times}_3 \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\boldsymbol{x}\right) \\ &= \boldsymbol{C_f} \bar{\times}_4 \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right) \text{vec}(\boldsymbol{x}) \bar{\times}_3 \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right) \text{vec}(\boldsymbol{x}) \\ &= \boldsymbol{C_f} \times_4 \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right)^T \bar{\times}_4 \text{vec}(\boldsymbol{x}) \times_3 \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right)^T \bar{\times}_3 \text{vec}(\boldsymbol{x}) \\ &= \left[\boldsymbol{C_f} \times_4 \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right)^T \times_3 \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right)^T\right] \bar{\times}_4 \text{vec}(\boldsymbol{x}) \bar{\times}_3 \text{vec}(\boldsymbol{x}) \end{aligned} \tag{29}$$

Since $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}$ is constant (refer back to equation 22), this can be pushed into our constant, so that our equation 21 quadratic constant can be written as

$$\boldsymbol{C}_{dpdf} = \boldsymbol{C_f} \times_4 \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right)^T \times_3 \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right)^T \tag{30}$$

This constant is still for equation 21, not the force Jacobian. The only step left in calculating the triangle force Jacobian's quadratic coefficient is to multiply it on both sides by $\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right)$:

$$\begin{aligned} \boldsymbol{C} &= \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right)^T \boldsymbol{C}_{dpdf} \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right) \\ &= \boldsymbol{C}_{dpdf} \times_2 \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right)^T \times_1 \text{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right)^T \end{aligned} \tag{31}$$

With both the quadratic coefficient and the constant matrix, the force jacobian can now be written as a quadratic polynomial:

$$\frac{\partial^2 \psi}{\partial \boldsymbol{x}^2} = \boldsymbol{C} \bar{\times}_4 \text{vec}(\boldsymbol{x}) \bar{\times}_3 \text{vec}(\boldsymbol{x}) + \boldsymbol{Q} \tag{32}$$

The implementation of this can be found in the file **TRIANGLE.cpp** within the constructor.

6.1.1 Creating the Global Tangent Stiffness Matrix Coefficients Method 1

Now that the constant terms for the force Jacobian for each triangle have been computed, we can create global con-

stant terms for the tangent stiffness matrix in a similar manner to how the global tangent stiffness matrix was computed earlier.

```
1: function GLOBALSTIFFNESSCONSTANTS(null)
2:     Q_global.setZero()
3:     C_global.setZero()
4:     for triangle in triangles do
5:         Q ← triangle.getQ()
6:         C ← triangle.getC()
7:         for vertex in triangle vertices do
8:             if vertex is unconstrained then
9:                 i ← global_index(vertex)
10:                for vertex_2 in triangle vertices do
11:                    if vertex_2 is unconstrained then
12:                        j ← global_index(vertex_2)
13:                        Q_global(i,j) ← Q_global(i,j) + Q(vertex, vertex_2)
14:                        for vertex_3 in triangle vertices do
15:                            k ← global_index(vertex_3)
16:                            for vertex_4 in triangle vertices do
17:                                l ← global_index(vertex_4)
18:                                C_global(i,j,k,l) ← C_global(i,j,k,l) + C(vertex, vertex_2, vertex_3, vertex_4)
19:                            end for
20:                        end for
21:                    end if
22:                end for
23:            end if
24:        end for
25:    end for
26: end function
```

Note that the global coefficients match the order of the triangle coefficients; that is, the constant term, $\boldsymbol{Q}_{global}$, is a matrix, while the global coefficient for the quadratic term, $\boldsymbol{C}_{global}$, is a fourth-order tensor. The difference is in their dimensions; while $\boldsymbol{Q} \in \mathbb{R}^{6\times 6}$, $\boldsymbol{Q}_{global} \in \mathbb{R}^{2n\times 2n}$, and while $\boldsymbol{C} \in \mathbb{R}^{6\times 6\times 6\times 6}$, $\boldsymbol{C}_{global} \in \mathbb{R}^{2n\times 2n\times 2n\times 2n}$.

Also note that this pseudocode is abbreviated; while it only shows adding $\boldsymbol{C}$ and $\boldsymbol{Q}$ to one position in the global coefficients, you actually have to add it to every possible combination of the current vertices' x and y coordinates. For example, if $vertex$ and $vertex_2$ are unconstrained, then

$$\begin{gathered}\boldsymbol{Q}_{global}(i,j) += \boldsymbol{Q}(vertex(x), vertex_2(x)),\\ \boldsymbol{Q}_{global}(i+1,j) += \boldsymbol{Q}(vertex(y), vertex_2(x)),\\ \boldsymbol{Q}_{global}(i,j+1) += \boldsymbol{Q}(vertex(x), vertex_2(y)), \text{ and}\\ \boldsymbol{Q}_{global}(i+1,j+1) += \boldsymbol{Q}(vertex(y), vertex_2(y)).\end{gathered}$$

After all coefficients are precomputed, the global stiffness matrix calculation is as easy as computing:

$$\boldsymbol{K}(\boldsymbol{x}) = \boldsymbol{C}_{global}\bar{\times}_4\boldsymbol{x}\bar{\times}_3\boldsymbol{x} + \boldsymbol{Q}_{global} \tag{33}$$

This implementation is not included in the current code, but can be found by going back into previous versions of the repository. This method is no longer used because when n (the number of vertices), is sufficiently large, the user will run into memory issues, making the code impossible to run. To overcome this issue, a new method was created. Method 2 will be covered in the reduction portion of this writeup, since it incorporates reduction directly into the global build.

6.2 Coefficients for Internal Force

In equation 20, there is one cubic term and one linear term. Since the force Jacobian is the derivative of internal force, the linear coefficient for internal force is actually also $\boldsymbol{Q}$ (eq 26). The connection between the cubic term and the quadratic term, however, is not as clean, because the cubic term can be reduced into a fourth order tensor as well. We are able to reduce the coefficient for the cubic term into a fourth order tensor because equation 20 is vectorized, reducing its order by one.

The vectorized cubic term, before separating $\boldsymbol{x}$, is

$$\text{vec}\left(\lambda\det(\boldsymbol{F})\begin{bmatrix} f_3 & -f_1\\ -f_2 & f_0\end{bmatrix}\right)$$

which expands to

$$\lambda\begin{bmatrix} f_0f_3^2 - f_1f_2f_3\\ f_1f_2^2 - f_0f_2f_3\\ f_1^2f_2 - f_0f_1f_3\\ f_0^2f_3 - f_0f_1f_2\end{bmatrix}$$

In order to separate out all of the $\boldsymbol{x}$'s, we can first separate all of the $\boldsymbol{f}$ terms, like we did for the force jacobian. By working backwards from the matrix above, two fourth order tensor, $\boldsymbol{A}_f$ can be derived such that

$$\boldsymbol{A}_\mu = \begin{bmatrix} \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&1 \end{bmatrix} & \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&0&-1\\0&0&0&0 \end{bmatrix} & \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix} & \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix} \\ \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&-1&0 \end{bmatrix} & \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&1&0\\0&0&0&0 \end{bmatrix} & \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix} & \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix} \\ \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix} & \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix} & \begin{bmatrix} 0&0&0&0\\0&1&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix} & \begin{bmatrix} 0&-1&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix} \\ \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix} & \begin{bmatrix} 0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix} & \begin{bmatrix} 0&0&0&0\\-1&0&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix} & \begin{bmatrix} 1&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0 \end{bmatrix} \end{bmatrix} \tag{34}$$

where

$$\boldsymbol{A_f} \bar{\times}_4 \operatorname{vec}\boldsymbol{F} \bar{\times}_3 \operatorname{vec}\boldsymbol{F} \bar{\times}_2 \operatorname{vec}\boldsymbol{F} = \operatorname{vec}\left(\lambda \det(\boldsymbol{F}) \begin{bmatrix} f_3 & -f_1 \\ -f_2 & f_0 \end{bmatrix}\right) \tag{35}$$

Then, by replacing $\boldsymbol{F} = \left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\boldsymbol{x}\right)$ and pushing $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}$ to the left side of the equation, we end up with a fourth order tensor, $\boldsymbol{A}_{pk}$, such that

$$\boldsymbol{A}_{pk} = \boldsymbol{A_f} \times_4 \operatorname{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right) \times_3 \operatorname{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right) \times_2 \operatorname{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right) \tag{36}$$

$\boldsymbol{A}_{pk}$ is the coefficient for the cubic term in the Piola-Kirchhoff equation (20). To make this a coefficient for internal force, it must be multiplied once more by vec $\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right)$ (eq. 23):

$$\boldsymbol{A} = \boldsymbol{A}_{pk} \times_1 \operatorname{vec}\left(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}\right) \tag{37}$$

With both the cubic coefficient and the linear matrix, the internal force can now be written as a cubic polynomial:

$$\frac{\partial \psi}{\partial \boldsymbol{x}} = \boldsymbol{A} \bar{\times}_4 \operatorname{vec}(\boldsymbol{x}) \bar{\times}_3 \operatorname{vec}(\boldsymbol{x}) \bar{\times}_2 \operatorname{vec}(\boldsymbol{x}) + \boldsymbol{Q} \operatorname{vec}(\boldsymbol{x}) \tag{38}$$

The implementation of this can be found int **TRIANGLE.cpp** within the constructor function.

6.2.1 Creating the Global Internal Force Coefficients Method 1

Now that the coefficients for the internal force for each triangle has been computed, we can create global constant coefficients for internal force in a similar manner to how the global internal force was computed earlier.

```
1:  function GLOBALFORCECONSTANTS(null)
2:      P_global.setZero()
3:      A_global.setZero()
4:      for triangle in triangles do
5:          P ← triangle.getQ()
6:          A ← triangle.getA()
7:          for vertex in triangle vertices do
8:              if vertex is unconstrained then
9:                  i ← global_index(vertex)
10:                 for vertex_2 in triangle vertices do
11:                     j ← global_index(vertex_2)
12:                     P_global(i,j) ← P_global(i,j) + P(vertex, vertex_2)
13:                     for vertex_3 in triangle vertices do
14:                         k ← global_index(vertex_3)
15:                         for vertex_4 in triangle vertices do
16:                             l ← global_index(vertex_4)
17:                             A_global(i,j,k,l) ← A_global(i,j,k,l) + A(vertex, vertex_2, vertex_3, vertex_4)
18:                         end for
19:                     end for
20:                 end for
21:             end if
22:         end for
23:     end for
24: end function
```

Like the pseudocode for computing the global stiffness matrix coefficients, the assignment lines are abbreviated in the same way. Refer back to **Creating the Global Tangent Stiffness Matrix Coefficients Method 1** for a refresher.

Also note how a separate global matrix, $\boldsymbol{P}_{global}$, was created here, despite the fact the linear internal force coefficient is identical to the constant force Jacobian coefficient for a single triangle. This is because the loops they run through are slightly different - in the pseudocode above, $\boldsymbol{Q}$ is added to $\boldsymbol{P}_{global}$ so long as the first vertex is unconstrained. In the pseudocode for the stiffness matrix, $\boldsymbol{Q}$ is added to $\boldsymbol{Q}_{global}$ only if the first **and** the second vertex are unconstrained. $\boldsymbol{P}_{global} \neq \boldsymbol{Q}_{global}$!

After computing coefficients, evaluating the internal force at any point in time is as simple as calculating:

$$\boldsymbol{R}(\boldsymbol{x}) = \boldsymbol{A}_{global} \bar{\times}_4 \boldsymbol{x} \bar{\times}_3 \boldsymbol{x} \bar{\times}_2 \boldsymbol{x} + \boldsymbol{P}_{global} \boldsymbol{x} \tag{39}$$

This implementation is not included in the current code for the same reason as the stiffness matrix method 1, but can be found by going back into previous versions of the repository. Method 2 will be covered in the reduction portion of this writeup, since it incorporates reduction directly into the global build.

7 Adjusting Polynomials for Displacement

As mentioned in section 3.4, $\boldsymbol{x}$ and $\boldsymbol{u}$ are no longer interchangeable through the cubic polynomial implementation of computing force.

7.1 Adjusting Internal Force

While it's true that

$$\boldsymbol{R}(\bar{\boldsymbol{x}}) = 0$$

the individual terms within the internal force equation are non-zero at rest position $\bar{\boldsymbol{x}}$. That is,

$$\boldsymbol{A} \bar{\times}_4 \bar{\boldsymbol{x}} \bar{\times}_3 \bar{\boldsymbol{x}} \bar{\times}_2 \bar{\boldsymbol{x}} \neq 0$$
$$\boldsymbol{A} \bar{\times}_4 \bar{\boldsymbol{x}} \bar{\times}_3 \bar{\boldsymbol{x}} \bar{\times}_2 \bar{\boldsymbol{x}} = -\boldsymbol{Q} \bar{\boldsymbol{x}}$$

Since the terms are non-zero at $\bar{\boldsymbol{x}}$, $\boldsymbol{u}$ cannot just replace $\boldsymbol{x}$ in this equation. We can, however, derive an alternate equation to deal with displacement. The first step is to insert $\bar{\boldsymbol{x}} + \boldsymbol{u}$ for every $\boldsymbol{x}$.

$$\begin{aligned} &\boldsymbol{A} \bar{\times}_4 (\bar{\boldsymbol{x}} + \boldsymbol{u}) \bar{\times}_3 (\bar{\boldsymbol{x}} + \boldsymbol{u}) \bar{\times}_2 (\bar{\boldsymbol{x}} + \boldsymbol{u}) \\ &\quad = (\boldsymbol{A} \bar{\times}_4 \bar{\boldsymbol{x}}) \bar{\times}_3 (\bar{\boldsymbol{x}} + \boldsymbol{u}) \bar{\times}_2 (\bar{\boldsymbol{x}} + \boldsymbol{u}) \\ &\qquad + (\boldsymbol{A} \bar{\times}_4 \boldsymbol{u}) \bar{\times}_3 (\bar{\boldsymbol{x}} + \boldsymbol{u}) \bar{\times}_2 (\bar{\boldsymbol{x}} + \boldsymbol{u}) \end{aligned} \tag{40}$$

Looking at the first term,

$$\begin{aligned} (\boldsymbol{A} \bar{\times}_4 \bar{\boldsymbol{x}}) \bar{\times}_3 (\bar{\boldsymbol{x}} + \boldsymbol{u}) \bar{\times}_2 (\bar{\boldsymbol{x}} + \boldsymbol{u}) &= (\boldsymbol{A} \bar{\times}_4 \bar{\boldsymbol{x}} \bar{\times}_3 \bar{\boldsymbol{x}}) \bar{\times}_2 (\bar{\boldsymbol{x}} + \boldsymbol{u}) \\ &\quad + (\boldsymbol{A} \bar{\times}_4 \bar{\boldsymbol{x}} \bar{\times}_3 \boldsymbol{u}) \bar{\times}_2 (\bar{\boldsymbol{x}} + \boldsymbol{u}) \\ &= (\boldsymbol{A} \bar{\times}_4 \bar{\boldsymbol{x}} \bar{\times}_3 \bar{\boldsymbol{x}} \bar{\times}_2 \bar{\boldsymbol{x}}) + (\boldsymbol{A} \bar{\times}_4 \bar{\boldsymbol{x}} \bar{\times}_3 \bar{\boldsymbol{x}} \bar{\times}_2 \boldsymbol{u}) \\ &\quad + (\boldsymbol{A} \bar{\times}_4 \bar{\boldsymbol{x}} \bar{\times}_3 \boldsymbol{u} \bar{\times}_2 \bar{\boldsymbol{x}}) + (\boldsymbol{A} \bar{\times}_4 \bar{\boldsymbol{x}} \bar{\times}_3 \boldsymbol{u} \bar{\times}_2 \boldsymbol{u}) \end{aligned} \tag{41}$$

Notice that one of the final terms here is $(\boldsymbol{A} \bar{\times}_4 \bar{\boldsymbol{x}} \bar{\times}_3 \bar{\boldsymbol{x}} \bar{\times}_2 \bar{\boldsymbol{x}})$. Since $\boldsymbol{R}(\bar{\boldsymbol{x}}) = 0$, $(\boldsymbol{A} \bar{\times}_4 \bar{\boldsymbol{x}} \bar{\times}_3 \bar{\boldsymbol{x}} \bar{\times}_2 \bar{\boldsymbol{x}}) = -\boldsymbol{Q} \bar{\boldsymbol{x}}$, so this term can be ignored.

The second term of equation 40 can be deconstructed in a similar manner:

$$\begin{aligned} &(\boldsymbol{A} \bar{\times}_4 \boldsymbol{u}) \bar{\times}_3 (\bar{\boldsymbol{x}} + \boldsymbol{u}) \bar{\times}_2 (\bar{\boldsymbol{x}} + \boldsymbol{u}) \\ &\quad = (\boldsymbol{A} \bar{\times}_4 \boldsymbol{u} \bar{\times}_3 \bar{\boldsymbol{x}} \bar{\times}_2 \bar{\boldsymbol{x}}) + (\boldsymbol{A} \bar{\times}_4 \boldsymbol{u} \bar{\times}_3 \bar{\boldsymbol{x}} \bar{\times}_2 \boldsymbol{u}) \\ &\qquad + (\boldsymbol{A} \bar{\times}_4 \boldsymbol{u} \bar{\times}_3 \boldsymbol{u} \bar{\times}_2 \bar{\boldsymbol{x}}) + (\boldsymbol{A} \bar{\times}_4 \boldsymbol{u} \bar{\times}_3 \boldsymbol{u} \bar{\times}_2 \boldsymbol{u}) \end{aligned} \tag{42}$$

Notice that the last term is structured like the initial equation for $\boldsymbol{x}$, yet we have many extra terms between equations 41 and 42. These extra terms form a quadratic term and modify the linear constant. The linear constant for internal force now becomes:

$$\boldsymbol{P} = \boldsymbol{Q} + (\boldsymbol{A} \bar{\times}_3 \bar{\boldsymbol{x}} \bar{\times}_2 \bar{\boldsymbol{x}}) + (\boldsymbol{A} \bar{\times}_4 \bar{\boldsymbol{x}} \bar{\times}_2 \bar{\boldsymbol{x}}) + (\boldsymbol{A} \bar{\times}_4 \bar{\boldsymbol{x}} \bar{\times}_3 \bar{\boldsymbol{x}}) \tag{43}$$

and the new quadratic term is defined as:

$$\boldsymbol{B} = (\boldsymbol{A} \bar{\times}_2 \bar{\boldsymbol{x}}) + (\boldsymbol{A} \bar{\times}_3 \bar{\boldsymbol{x}}) + (\boldsymbol{A} \bar{\times}_4 \bar{\boldsymbol{x}}) \tag{44}$$

where $\boldsymbol{B} \in \mathbb{R}^{6 \times 6 \times 6}$, and is thus a third-order tensor.

With these new terms, the redefined internal force equation for displacement is:

$$\frac{\partial \psi}{\partial \boldsymbol{u}} = \boldsymbol{A} \bar{\times}_4 \boldsymbol{u} \bar{\times}_3 \boldsymbol{u} \bar{\times}_2 \boldsymbol{u} + \boldsymbol{B} \bar{\times}_3 \boldsymbol{u} \bar{\times}_2 \boldsymbol{u} + \boldsymbol{P} \boldsymbol{u} \tag{45}$$

The global coefficients can be built the same way as before, with an extra tensor to store values of $\boldsymbol{B}$. With these modified global coefficients, the calculation of global internal force is now:

$$\boldsymbol{R}(\boldsymbol{u}) = \boldsymbol{A}_{global} \bar{\times}_4 \boldsymbol{u} \bar{\times}_3 \boldsymbol{u} \bar{\times}_2 \boldsymbol{u} + \boldsymbol{B}_{global} \bar{\times}_3 \boldsymbol{u} \bar{\times}_2 \boldsymbol{u} + \boldsymbol{P}_{global} \boldsymbol{u} \tag{46}$$

7.2 Adjusting the Stiffness Matrix

The force Jacobian is adjusted using the same method as the internal force. When $\bar{\boldsymbol{x}} + \boldsymbol{u}$ is substituted for every $\boldsymbol{x}$, the resulting force Jacobian equation is

$$\frac{\partial^2 \psi}{\partial \boldsymbol{u}^2} = \boldsymbol{C} \bar{\times}_4 \bar{\boldsymbol{x}} \bar{\times}_3 \bar{\boldsymbol{x}} + \boldsymbol{C} \bar{\times}_4 \bar{\boldsymbol{x}} \bar{\times}_3 \boldsymbol{u} + \boldsymbol{C} \bar{\times}_3 \bar{\boldsymbol{x}} \bar{\times}_3 \boldsymbol{u} + \boldsymbol{C} \bar{\times}_4 \boldsymbol{u} \bar{\times}_3 \boldsymbol{u} \tag{47}$$

The extra terms here form a new linear term and modify the constant matrix. The constant matrix now becomes:

$$\boldsymbol{Q} = \boldsymbol{Q} + \boldsymbol{C} \bar{\times}_4 \bar{\boldsymbol{x}} \bar{\times}_3 \bar{\boldsymbol{x}} \tag{48}$$

and the linear term is defined as:

$$\boldsymbol{G} = \boldsymbol{C} \bar{\times}_4 \bar{\boldsymbol{x}} + \boldsymbol{C} \bar{\times}_3 \bar{\boldsymbol{x}} \tag{49}$$

where $\boldsymbol{G}$ is a third-order tensor.

With these modifications, the new force Jacobian equation is:

$$\frac{\partial^2 \psi}{\partial \boldsymbol{u}^2} = \boldsymbol{C} \bar{\times}_4 \boldsymbol{u} \bar{\times}_3 \boldsymbol{u} + \boldsymbol{G} \bar{\times}_3 \boldsymbol{u} + \boldsymbol{Q}_{\boldsymbol{u}} \tag{50}$$

The global coefficients can be built the same way as before, with an extra tensor to store values of $\boldsymbol{G}$. Thus, the calculation of the global stiffness matrix is now:

$$\boldsymbol{K}(\boldsymbol{u}) = \boldsymbol{C}_{global} \bar{\times}_4 \boldsymbol{u} \bar{\times}_3 \boldsymbol{u} + \boldsymbol{G}_{global} \bar{\times}_3 \boldsymbol{u} + \boldsymbol{Q}_{global} \tag{51}$$

8 Generating a Deformation Basis

There are many ways to generate a deformation basis. It's a hard, open problem in solid mechanics, as there is no unanimously used algorithm for it. The approach used for this paper was the mass-PCA approach. This was achieved by running the unreduced program to record some chosen number of distinct deformations. Each vertex displacement for a single deformation was normalized and stored in one column of a matrix. In addition to displacements, the internal force at each time step was normalized and included as a column in the matrix, and the normalized external force for each different simulation. The code for this process can be found in **simulation.cpp** within the function **readCommandLine()**. The resulting matrix $\boldsymbol{X}$ is exported as a file, and read back into the program in **TRIANGLE_MESH.cpp** when creating the triangle mesh.

After computing this matrix, SVD was run on $\boldsymbol{X}$, and the first r columns of the $\boldsymbol{U}_{SVD}$ matrix became the deformation basis, $\boldsymbol{U} \in \mathbb{R}^{2n \times r}$. This basis works well for stationary objects, but it does not incorporate translation. To create a basis that allows for translation, create two vectors $\boldsymbol{t_x}, \boldsymbol{t_y} \in \mathbb{R}^{2n}$, such that $\boldsymbol{t_x}[i] = 1$ when i is an x coordinate, and 0 otherwise; and $\boldsymbol{t_y}[i] = 1$ when i is a y coordinate, and 0 otherwise.

Normalize both $\boldsymbol{t_x}$ and $\boldsymbol{t_y}$, then append them so they are the first two columns of $\boldsymbol{U}$. Then, adjust every column that is not the translation columns, as shown below:

```
1: function ADDTRANSLATION(null)
2:     for c ∈ U do
3:         if c is not t_x or t_y then
4:             c ← c − t_x^T c t_x
5:             c ← c − t_y^T c t_y
6:         end if
7:     end for
8: end function
```

The implementation for a stationary deformation basis can be found in **TRIANGLE_MESH.cpp** within the function **basisNoTranslation()**. No implementation including translation has been written so far.

9 Reducing the Euler-Lagrange Equation of Motion

Now that a deformation basis has been generated, it can be used to reduce the order of the Euler-Lagrange equation of motion calculation. The next three subsections explain how these reductions can be done.

9.1 Reducing the Internal Force

There are two different ways to reduce the global internal force.

9.1.1 Method 1

This is only relevent for the method 1 implementation of the global internal force. In order to reduce the internal forces the cubic polynomial way, the displacement adjusted formula must be used. Assuming that equation 46 is used, reducing internal force is as simple as reducing every precomputed coefficient:

$$\tilde{\boldsymbol{A}}_{global} = \boldsymbol{A}_{global} \times_4 \boldsymbol{U}^T \times_3 \boldsymbol{U}^T \times_2 \boldsymbol{U}^T \times_1 \boldsymbol{U}^T \tag{52}$$

$$\tilde{\boldsymbol{B}}_{global} = \boldsymbol{B}_{global} \times_3 \boldsymbol{U}^T \times_2 \boldsymbol{U}^T \times_1 \boldsymbol{U}^T \tag{53}$$

$$\tilde{\boldsymbol{P}}_{global} = \boldsymbol{U}^T \boldsymbol{P}_{global} \boldsymbol{U} \tag{54}$$

With these reduced coefficients, the reduced internal force is:

$$\tilde{\boldsymbol{R}}(\boldsymbol{q}) = \tilde{\boldsymbol{A}}_{global} \bar{\times}_4 \boldsymbol{q} \bar{\times}_3 \boldsymbol{q} \bar{\times}_2 \boldsymbol{q} + \tilde{\boldsymbol{B}}_{global} \bar{\times}_3 \boldsymbol{q} \bar{\times}_2 \boldsymbol{q} + \tilde{\boldsymbol{P}}_{global} \boldsymbol{q} \tag{55}$$

9.1.2 Method 2

As stated earlier, the current implementation of this paper does not follow method 1 of building the global internal force. The method that is currently used combines the reduction of the global internal force into its initial creation in order to save on memory.

9.2 Reducing the Global Tangent Stiffness Matrix

There are two different ways to reduce the global tangent stiffness matrix.

9.2.1 Method 1

This is only relevent for the method 1 implementation of the global tangent stiffness matrix. In order to reduce the tangent stiffness matrix the polynomial way, the displacement adjusted formula must be used. Assuming that equation 51 is used, reducing the stiffness matrix is as simple as reducing every precomputed coefficient:

$$\tilde{\boldsymbol{C}}_{global} = \boldsymbol{C}_{global} \times_4 \boldsymbol{U}^T \times_3 \boldsymbol{U}^T \times_2 \boldsymbol{U}^T \times_1 \boldsymbol{U}^T \tag{56}$$

$$\tilde{\boldsymbol{G}}_{global} = \boldsymbol{G}_{global} \times_3 \boldsymbol{U}^T \times_2 \boldsymbol{U}^T \times_1 \boldsymbol{U}^T \tag{57}$$

$$\tilde{\boldsymbol{Q}}_{global} = \boldsymbol{U}^T \boldsymbol{Q}_{global} \boldsymbol{U} \tag{58}$$

With these reduced coefficients, the reduced stiffness matrix is:

$$\tilde{\boldsymbol{K}}(\boldsymbol{q}) = \tilde{\boldsymbol{C}}_{global} \bar{\times}_4 \boldsymbol{q} \bar{\times}_3 \boldsymbol{q} + \tilde{\boldsymbol{G}}_{global} \bar{\times}_3 \boldsymbol{q} + \tilde{\boldsymbol{Q}}_{global} \tag{59}$$

9.2.2 Method 2

As stated earlier, the current implementation of this paper does not follow method 1 of building the global tangent stiffness matrix. The method that is currently used combines the reduction of the global tangent stiffness matrix into its initial creation in order to save on memory.

9.3 Reducing All Other Forces

For reducing external force and mass, follow equations 4 and 7. The reduced damping matrix can be calculated after mass and stiffness are reduced, so that

$$\tilde{\boldsymbol{D}} = \alpha \tilde{\boldsymbol{M}} - \beta \tilde{\boldsymbol{K}}(\boldsymbol{q}) \tag{60}$$

After everything is reduced, run the Newmark integrator exactly as before. Refer back to the section titled **Implementing the Newmark Integrator** for a reminder of how to implement it.

10 Current Bugs

Help

11 Potential Future Steps

This is an ongoing list of potential future steps for this project. As of now, possible extensions are:

1. There is the possibility that some of the tensors this paper uses can be flattened in some way. Most of the tensors are sparse, and storing the entire structure full of zeros is a waste of space and computation time.

References

[1] Smith, B., de Goes, F., and Kim, T., 2018. "Stable neohookean flesh simulation". *ACM Trans. on Graphics,* **37**(2), Mar, p. Article 12. 15 pages.

[2] Barbic, J., and James, D., 2005. "Real-time subspace integration for st. venant-kirchhoff deformable models". *ACM Trans. on Graphics,* **23**(3), Aug, pp. 982–990.

[3] Kolda, T., and Bader, B., 2009. "Tensor decompositions and applications". *SIAM Review,* **51**(3), Aug, pp. 455–500.