De-Mystifying Reduced-Order St. Venant-Kirchhoff Deformable Models

Gillian P. Reyes

The goal of this paper is to document each step taken in recreating the results of Real-Time Subspace Integration for St. Venant-Kirchhoff Deformable Models [1], so that others may use this as a guide to replicate this work.

1 Introduction

The goal of this paper is to document each step taken in recreating Barbič and James's research from their 2005 paper [1]. In this research, they utilize past work on real-time deformable objects and model reduction in solid mechanics to speed up computations and allow for large deformations. Finite Element Method is used to discretize partial differential equations of solid continuum mechanics, allowing motion to be described through the Euler-Lagrange equation,

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

where $M \in \mathbb{R}^{2n\times 2n}$ is the mass matrix, D is the damping force, R is internal force, and f is external force. $u \in \mathbb{R}^{2n}$ is the displacement vector. This equation is then further reduced by introducing a time-independent matrix, $U \in \mathbb{R}^{2n\times r}$, specifying a basis of some r-dimensional linear subspace. After this reduction, the equation and each of its terms can be solved using:

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

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

$$u = Uq \tag{3}$$

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

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

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

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

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

Using these equations to reduce the problem speeds up the computation of motion to a degree, but everything can be sped up further if you treat the calculation of R as cubic polynomial and K as a quadratic polynomial. Then, you can precompute constant coefficients to these equations, such that

$$\tilde{R}(q) = U^T R(q) = P^i q_i + Q^{ij} q_i q_j + S^{ijk} q_i q_j q_k \tag{9}$$

$$\tilde{K}(q) = \frac{\partial \tilde{R}(q)}{\partial q_l} = P^l + (Q^{li} + Q^{il}) q_i + (S^{ijl} + S^{ilj} + S^{lij}) q_i q_j \tag{10}$$

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.

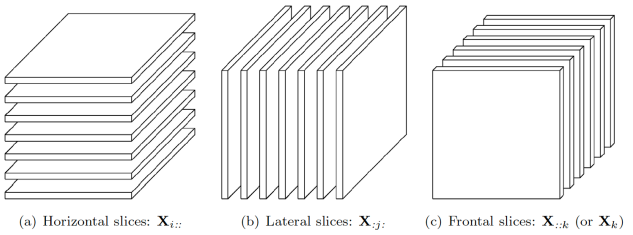

Fig. 1. A visual representation of a third order tensor. It can be imagined in three distinct ways, depending on how you index it

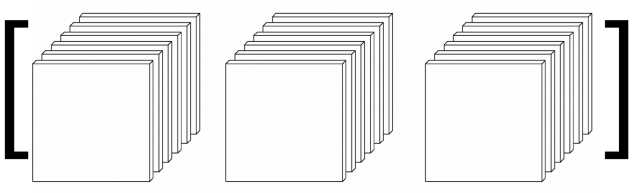

Fig. 2. A visual representation of a fourth order tensor. In code, it is treated as a vector of third order tensors.

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.

2 An Overview of Tensors

Before going into the details of how to implement reduced-order St. Venant-Kirchoff deformable models, it's important to review tensors, as they show up in many places throughout this paper. The following explanation is influenced by Kolda and Bader's paper on Tensor Decomposition and Applications [2], 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

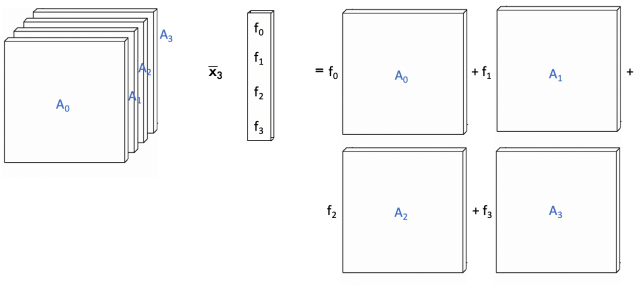

Fig. 3. A visual representation of a 3-mode vector product, where $A \in \mathbb{R}^{n \times n \times 4}$ and $f \in \mathbb{R}^4$

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.

2.1 Scalar Multiplication

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

$$(aX)_{i_1,i_2,\ldots,i_n} = a * x_{i_1,i_2,\ldots,i_n} \tag{11}$$

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 $X \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ with a vector $v \in \mathbb{R}^{I_n}$ is denoted by $X \bar{\times}_n v$. Elementwise,

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

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

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

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

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

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

If $n \neq m$, then

$$X \times_n A \times_m B = X \times_m B \times_n A \quad (15)$$

If the modes are the same, then

$$X \times_n A \times_n B = X \times_n (BA) \quad (16)$$

These two rules will also be used later.

3 Creating the Triangles

When modelling 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.

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.
2. a vector of actual positions, which update as the whole mesh moves.
3. a material model of some sort, to keep track of lambda and mu values for force calculations.
4. rest area; this is used in calculating the internal force and stiffness matrix.
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 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 $\frac{\partial F}{\partial u}$.
3. a function that computes the deformation gradient, F.
4. a function to compute the internal force for the triangle given the deformation gradient.
5. a function to compute the force jacobian for the triangle given the deformation gradient.

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**, $F \in \mathbb{R}^{2\times 2}$, 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:

$$F = D_s D_m^{-1} = \begin{bmatrix} (x_2 - x_0) & (x_4 - x_0) \\ (x_3 - x_1) & (x_5 - x_1) \end{bmatrix} D_m^{-1} = \begin{bmatrix} f_0 & f_2 \\ f_1 & f_3 \end{bmatrix} \quad (17)$$

Where D_s is the spatial matrix made of the deformed vertices, 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, D_m^{-1} is constant.

3.2 Implementing StVK

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 StVK 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.

A St. Venant-Kirchhoff deformable model is defined by the StVK strain energy, where

$$\psi = \mu ||F^T F - I||^2 + \frac{\lambda}{2} tr(F^T F - I)^2 \quad (18)$$

, and λ and μ are Lamé coefficients.

In computing the internal force of a single triangle, the Piola-Kirchhoff stress tensor, i.e. the derivative of the strain

energy by the deformation gradient, is used. The Piola-Kirchhoff stress tensor is

$$\frac{\partial \psi}{\partial F} = 4\mu F F^T F - 4\mu F + 2\lambda F tr(F^T F) - 4\lambda F \tag{19}$$

In computing the force jacobian of a single triangle, the energy hessian is used. The energy hessian is:

$$\begin{aligned}\frac{\partial^2 \psi}{\partial F^2} = 4\mu[\frac{\partial F}{\partial F_i} F^T F + F \frac{\partial F}{\partial F_i}^T F + F F^T \frac{\partial F}{\partial F_i}] \\ +2\lambda \frac{\partial F}{\partial F_i} tr(F^T F) - 4[\mu - \lambda] I\end{aligned} \tag{20}$$

3.3 Computing dF/dx

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

$$\begin{aligned}\frac{\partial F}{\partial x} &= \begin{bmatrix} \frac{\partial F}{\partial x_0} & \frac{\partial F}{\partial x_1} & \frac{\partial F}{\partial x_2} & \frac{\partial F}{\partial x_3} & \frac{\partial F}{\partial x_4} & \frac{\partial F}{\partial x_5} \end{bmatrix} \\ &= \begin{bmatrix} \begin{bmatrix} -1 & -1 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ -1 & -1 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \end{bmatrix}\end{aligned} \tag{21}$$

This was derived using equation 17 for 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

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

then

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

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:

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

then

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

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

$$f = \frac{\partial \psi}{\partial u} = vec(\frac{\partial F}{\partial x}) * vec(\frac{\partial \psi}{\partial F}) \tag{22}$$

where the terms are defined by equations 21 and 19, respectively. The force jacobian for each triangle can be computed using the equation

$$\frac{\partial^2 \psi}{\partial u^2} = vec(\frac{\partial F}{\partial x}) * \frac{\partial^2 \psi}{\partial F^2} * vec(\frac{\partial F}{\partial x})^T \tag{23}$$

where the terms are defined by equations 21 and 20. Notice how x, or current vertex placement, and u, vertex *displacement*, are used interchangably here. In general, this is possible because the interal force at rest position $\bar{x}$ is 0, and $x = \bar{x} + u$. However, it is important to note, because these will **not** be interchangable 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. 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.
2. a vertex to global index mapping.
3. a vector of vertex placements.
4. a vector of vertex rest positions.
5. a vector of vertex displacements.
6. a vector of vertex indices representing which vertices are **constrained**.
7. a vector of vertex indices representing which vertices are **unconstrained**.
8. a vector for inteneral force.
9. a vector for external force.
10. a mass matrix.
11. a vector for velocity and a vector for acceleration.
12. the basis reduction matrix.
13. a vector for the reduced basis vector.

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 isn't 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 22. To calculate the global internal force, $R(u) \in \mathbb{R}^{2n}$, 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
```

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 23. To calculate the global stiffness matrix, $K(u) \in \mathbb{R}^{2n \times 2n}$, 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
12:                     end if
13:                 end for
14:             end if
15:         end for
16:     end for
17: end function
```

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.

5.4 Implementing the Damping Force

The damping force is calculated using the equation

$$D(u,\dot{u}) = (\alpha M + \beta K(u))\dot{u} \tag{24}$$

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.

5.5 Implementing the Newmark Integrator

An explanation of how to take a step in time.

6 Generating Precomputed Coefficients

An explanation of how to create precomputed coefficients for the cubic polynomial approach for the unreduced problem.

6.1 Coefficients for the Tangent Stiffness Matrix

An explanation of how to derive the coefficients for the tangent stiffness matrix equation.

6.1.1 Constant Coefficient for the Tangent Stiffness Matrix

An explanation of how to derive the constant coefficient for the tangent stiffness matrix.

6.1.2 Quadratic Coefficient for the Tangent Stiffness Matrix

An explanation of how to derive the quadratic coefficient for the tangent stiffness matrix.

6.1.3 Creating the Global Tangent Stiffness Matrix Coefficients

An explanation of how to derive the global tangent stiffness matrix.

6.2 Coefficients for Internal Force

An explanation of how to derive the coefficients for the internal force equation.

6.2.1 Linear Coefficient for Internal Force

An explanation of how to derive the linear coefficient for internal force.

6.2.2 Cubic Coefficient for for Internal Force

An explanation of how to derive the cubic coefficient for internal force.

6.2.3 Creating the Global Internal Force Coefficients

An explanation of how to derive the global internal force.

7 Generating a Deformation Basis

An explanation of how to generate a deformation basis.

8 Reducing the Euler-Lagrange Equation of Motion

An explanation of how to push U through so that we can reduce the order of our calculations.

8.1 Reducing the Internal Force

An explanation of how to reduce the calculation of the internal forces, changing the coefficients of the polynomial.

8.2 Reducing the Global Tangent Stiffness Matrix

An explanation of how to reduce the calculation of the stiffness, changing the coefficients of the polynomial.

8.3 Reducing All Other Forces

An explanation of how to reduce all other forces.

9 Running the program

A brief explanation of how to run the program

10 Conclusions

End it by explaining how I hope this helps anyone who wants to try implementing this on their own. Explain how this can probably be optimized further by flattening all tensors.

Acknowledgements

Thanks to Barbič for writing the paper and my advisor Theodore Kim.

References

[1] 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.

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