Three-Dimensional Simulation of Discrete Elastic Rods

Philip Ng

***Abstract*— Beams are used in various applications of engineering, from infrastructure to vehicles. As rigid as they may seem, all beams are elastic; they deform under the loads they are given. The elastic properties of a beam are integral for understanding its various properties, such as its yield point and fracture point. Knowing how a beam will deform in various conditions helps inform engineers whether a structure is safe for use. In this article, we will explore numerical methods based on governing differential equations to generate a three-dimensional simulation of the deformation of a naturally curved elastic rod fixed at one end. A larger version of such a simulation can be utilized to predict the elastic deformation of beams of various geometries under any external condition.**

I. INTRODUCTION AND THEORY

Let us consider a system acted on by conservative forces only. This is a valid assumption because most elastic structures, including beams, are primarily affected by such forces. In the case of the beam in viscous flow, we also have damping and weight forces acting on the system. Thus the general equation of motion for the i-th DOF would be

$$f = m_i \ddot{q}_i + \frac{\partial E_{elastic}}{\partial q_i} - f_i^{ext} = 0 \tag{1}$$

where f_i^{ext} is an external, conservative force. Let us assume the beam as a network of nodes and edges, where the nodes are approximated as masses and the edges are approximated as springs. The angle θ is called the *turn angle* of the system. There are $a = N$ nodes and $b = N - 1$ edges.

To prepare for simulation, we must discretize (1). Below is the discretized equations for the i-th DOF using the implicit and explicit method, respectively. We will be using an implicit simulation because they converge at larger timesteps and thus require less time to compute.

$$\frac{m_i}{\Delta t^2}[\frac{q_i(t_{k+1} - q_i(t_k)}{\Delta t} - \dot{q}_i(t_k)] + \frac{\partial E_{elastic}}{\partial q_i} - f_i^{ext} = 0 \tag{2}$$

We can now apply the discretized equations to vectors and matrices. Since the current problem is in 3-D with twisting, we will have we n = 4N - 1 degrees of freedom. Let us set up the following n DOF vector, $\mathbf{q}$:

$$\mathbf{q} = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ \theta_1 \\ \vdots \\ x_a \\ y_a \end{bmatrix}$$

For the mass component, we will use an n X n diagonal lumped mass matrix $\mathbf{M}$. The components m_{ii} represent the mass of node i

$$\mathbf{M} = \begin{bmatrix} m_{11} & 0 & 0 & \dots & 0 \\ 0 & m_{11} & 0 & \dots & 0 \\ 0 & 0 & m_{22} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & m_{aa} \end{bmatrix}$$

The weight matrix F_g only acts on z-oriented DOFs, so it is zero for all indices except indices divisible by 3.

$$F_g = \begin{bmatrix} 0 \\ 0 \\ W_1 \\ \vdots \\ 0 \\ 0 \\ W_a \end{bmatrix}$$

With these matrices defined, the governing equation (1) becomes

$$\boldsymbol{f} = \boldsymbol{M}\ddot{\boldsymbol{q}} + \frac{\partial E_{potential}}{\partial \boldsymbol{q}} + \boldsymbol{F} = 0 \tag{3}$$

The implicit solving method will require Newton Rhapson Iteration, which further requires calculation of the Jacobian. The Jacobian is the gradient of the governing equation, which yields the discretized equation

$$\boldsymbol{J} = \frac{\boldsymbol{M}}{\Delta t^2} + \frac{\partial^2 E_{potential}}{\partial^2 \boldsymbol{q}} - \frac{\partial f_i^{ext}}{\partial q_j} \tag{4}$$

The gradients and Hessians of the potential energies can be calculated using the functions `gradEs`, `gradEb`, `hessEs`, `hessEb` in the Appendix. Using the vectorized equations (3) and (4), we can generate a simulation of a three-dimensional rod with N nodes with external forces acting on the system.

II. GENERAL IMPLEMENTATION

For our implementation, let $n = 4N - 1$ equal the number of degrees of freedom, and $ne = n - 1$ be the number of edges.

For a three-dimensional beam deflection problem with twist, we will need two different reference frames: a *time-parallel reference frame* and a *material frame*. The material frame provides an orthogonal reference for the location of each node and edge, while the time-parallel reference frame serves as a reference for the twist of the material frame.

To build reference frames we must first find the set of vectors that are parallel to each edge, which we'll call the tangent vectors. This can be done using the helper function `computeTangent`. With that, we can construct the starting reference frame at t = 0 using a space-parallel frame. Below is the pseudocode for this process:

Algorithm 1 Generating Space-Parallel Starting Frame

Initialize vectors **a1, a2** for time-parallel reference director
Compute tangent vectors using `computeTangent`
define **a1** orthogonal to tangent vector and arbitrary vector
normalize **a1**
Define **a2** as orthogonal to **a1**
for all remaining edges **do**
    Compute tangent vectors again
    Apply parallel transport on previous **a1** director
    Normalize new **a1** and define orth. **a2**
end for

Parallel transport can be computed using the `parallelTransport` function, which can be found in the references.

With the space-parallel frame constructed we can obtain the material frame through the function `computeMaterialDirectors` and the starting reference twist vector using `computeRefTwist`.

Next, we must define the Voronoi length l_k, or the theoretical length of each node, using the following formula:

$$l_k = \frac{|e^{k-1}| + |e^k|}{2} \tag{5}$$

We can also find the natural curvature of the beam, κ, using the given `getKappa` function, which can be found in the references. Both the Voronoi length and the natural curvature will be useful in calculating the bending and stretching energies in the helper functions.

Before we begin the time stepping process, we must also define the fixed and free indices. For this case, we will define the first two nodes, or the first seven indices.

A. Helper Functions

The pseudocodes of the functions mentioned in the previous section are detailed here.

1. `getTangent(q)`

To generate a material frame, we use the `getTangent(q)` function, which takes the 4N -1 X 1 DOF vector **q** as an input and outputs the tangent vector for each edge (size ne X 3). Below is the pseudocode:

Algorithm 2 Determining the Tangent Vector of Each Edge

function GETTANGENT(q)
    Define number of edges and nodes
    Define tangent vector (size ne X 3)
    for all edges **do**
        Define position **x0** of current node from **q**
        Define position **x1** of next node from **q**
        Determine vector representing edge, **x1 - x0**
        Insert normalized edge into tangent vector
    end for
end function

2. `computeMaterialDirectors(a1, a2, theta)`

This helper function computes the material directors of each node, taking the first and second time parallel reference directors of size (ne X 3) as inputs, as well as the twist angle vector `theta`. Below is the pseudocode:

Algorithm 3 Determining the Tangent Vector of Each Edge

function COMPUTEMATERIALDIRECTORS(a1, a2, theta)parameters
    Define number of edges and nodes
    Define material directors m_1, m_2 of size (ne X 3)
    for all edges **do**
        Create m1 unit vector using theta vectors
        Create orthogonal m2 unit vector using theta vectors
    end for
end function

3. `computeRefTwist(a1, tangent, refTwist)`

To find the reference twist at each node, this helper function takes one of the time-parallel reference directors `a1`, the tangent vector, `tangent`, and the starting reference twist for each node (taken from the DOF vector **q**) `refTwist`. Using parallel transport, the function outputs an updated reference twist vector. Below is the pseudocode:

Algorithm 4 Determining the Reference Twist at Each Edge

function COMPUTEREFTWIST(a1, tangent, refTwist)
Define number of edges and nodes
for all edges **do**
Store previous and current time-parallel reference director in **u0, u1**
Store previous and current tangent vector in **t0, t1**.
Perform parallel transport on **u0, t0, t1** to get space-parallel vector
Obtain and store signed angle of transported vector in reference twist vector.
end for
end function

4. `computeTimeParallel(a1old, q0, q)`

At each time step, this helper function updates the time-parallel reference directors. It takes the the old time-parallel reference director `a1`, the old DOF vector `q0`, and the DOF vector at the current time step `q`, then uses parallel transport to output an updated time-parallel reference vector. Below is the pseudocode:

Algorithm 5 Update Time-Parallel Reference Directors

1: **function** COMPUTETIMEPARALLEL(a1old, q0, q)
2: Define number of edges and nodes
3: Compute tangent vectors at each edge for old step
4: Compute tangent vectors at each edge for new step
5: **for** all edges **do**
6: Get and store old and new tangent vectors in **t0, t** respectively
7: Create temp variable for **a1old** values
8: Perform parallel transport algorithm on **to, t, a1old**
9: Store results in new **a1** and **a2** vectors
10: **end for**
11: **end function**

5. `getFb(q, m1, m2)`

This function calculates the bending force vector using the gradient and Hessian of the bending energy, given the DOF vector **q** and the material directors **m1** and **m2**.

Algorithm 6 Obtain Bending Force

1: **function** GETFB(q, m1, m2)
2: Define number of nodes
3: Initialize vectors to store **f** and **J**
4: **for** all nodes except first and last **do**
5: **n0** = transpose of previous node position
6: **n1** = transpose of current node position
7: **n2** = transpose of next node position
8: Store material directors of previous edge, **m1e, m2e**
9: Store material directors of current edge, **m1ef m2f**
10: Get and store old and new tangent vectors in
11: Compute **dF, dJ** using `gradEbhessEB` function
12: **F = F - dF**
13: **J = J - dJ**
14: **end for**
15: **end function**

Note $F = F - dF$ and $J = J - dJ$ is only applied to the 11 indices corresponding to the previous, current, and subsequent node.

6. `getFs(q)`

This function calculates the bending force vector using the gradient and Hessian of the stretching energy, given the DOF vector **q**.

Algorithm 7 Obtain Bending Force

1: **function** GETFS(q)
2: Define number of nodes
3: Initialize vectors to store **f** and **J**
4: **for** all nodes except first and last **do**
5: **n1** = transpose of current node position
6: **n2** = transpose of next node position
7: Compute **dF, dJ** using `gradEshessEs` function
8: **F = F - dF**
9: **J = J - dJ**
10: **end for**
11: **end function**

Note $F = F - dF$ and $J = J - dJ$ is only applied to the 6 indices corresponding to the x, y, and z positions of the current and subsequent node.

7. `getFt(q, refTwist)`

This function calculates the twisting force vector using the gradient and Hessian of the twisting energy, given the DOF vector **q** and the reference twist vector `refTwist`.

Algorithm 8 Obtain Bending Force

function GETFT(q, refTwist)
    getFtq
    Define number of nodes
    Initialize vectors to store **f** and **J**
    for all nodes except first and last **do**
        n0 = transpose of previous node position
        n1 = transpose of current node position
        n2 = transpose of next node position
        Store twisting angles of previous and current edge, **te, tf**
        Compute **dF, dJ** using `gradEthessEt` functions
        F = F - dF
        J = J - dJ
    end for
end function

Note $F = F - dF$ and $J = J - dJ$ is only applied to the 11 indices corresponding to the previous, current, and subsequent node.

8. `getKappa(q, m1, m2)`

This function calculates the natural curvature vector κ using the current DOF vector `q` and the material directors `m1, m2`. The mathematical calculation of κ is computed by the `computeKappa` function found in the Appendix.

Algorithm 9 Obtain natural curvature

function GETFB(q, m1, m2)
    Define number of nodes
    Initialize vectors to store **f** and **J**
    for all nodes except first and last **do**
        n0 = transpose of previous node position
        n1 = transpose of current node position
        n2 = transpose of next node position
        Store material directors of previous edge, **m1e, m2e**
        Store material directors of current edge, **m1ef m2f**
        Compute κ using `computekappa` function
        store κ values in curvature array
    end for
end function

B. Main Function

With the helper functions defined, we utilize Newton-Rhapson iteration to simulate the deformation of the rod and update the DOF vector. For each time step, the material and time-parallel reference frames are updated, and the bending, twisting, and stretching forces and Jacobians are determined. From there, a simple Newton's update yields the new DOFs, which is used to further update the reference frames.

Algorithm 10 Discrete Elastic Rods SImulation

Guess **q**
i = 1
while error $>$ tolerance **do**
    Compute time-parallel reference **a1,a2,t** frame using **q**, **q0**, and old **a1**.
    Compute material frame **m1,m2,t** using **q**, **q0**, and old **a1**.
    Compute Fb, Jb using `gradEbhessEb`
    Compute Fs, Js using `gradEshessEs`
    Compute Ft Jt using `gradEthessEt`
    Sum elastic forces and Jacobians
    Define elastic force $\mathbf{f_{free}}$ in (3)
    Get full Jacobian $\mathbf{J_{free}}$ in (4) by adding elastic Jacobian
    Hey $\mathbf{q_{free}} = \mathbf{J_{free}} \backslash \mathbf{f_{free}}$
    error = $\Sigma|f_{free}|$
    i = i + 1
end while
Update DOFs **q** and velocity **u**
Update reference frame **a1, a2**
Update material directors **m1, m2**
Plot the rod position using `plotrod`

Because an implicit and time-parallel is implemented, the program is computationally efficient. The results of the simulation are presented in the following section.

III. DISCUSSION

In our study, a naturally curved rod of natural radius 2cm and total length 20cm is secured at one end and subjected to gravitational force. The location of its N nodes are

$$\mathbf{x_k} = [R_n cos((k-1)\Delta\theta), R_n sin((k-1)\Delta\theta), 0]$$

where $\Delta\theta = \frac{1}{R_n(N-1)}$, the curvature between each node. As mentioned earlier, the first two nodes are clamped on one end and free at the other end. The physicsal parameters are constant and uniform: density $\rho = 1000kg/m^3$, cross sectional radius $r_0 = 1mm$, Young's Modulus $E = 10MPa$, and shear modulus $G = E/3$. We used a time step of $\Delta t = 0.01s$. Running a 5 second simulation yielded the following result:

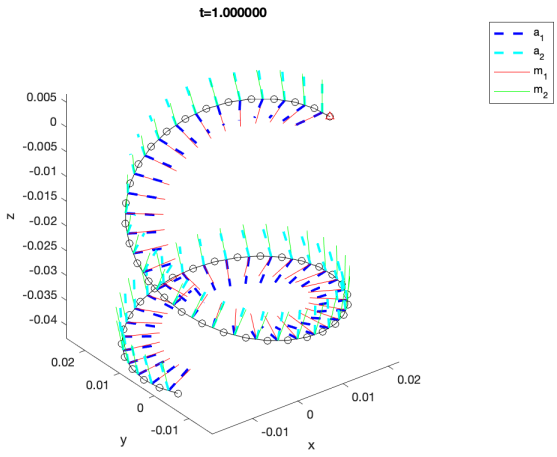

Fig. 1. Beam simplified as network of springs and spheres[1]

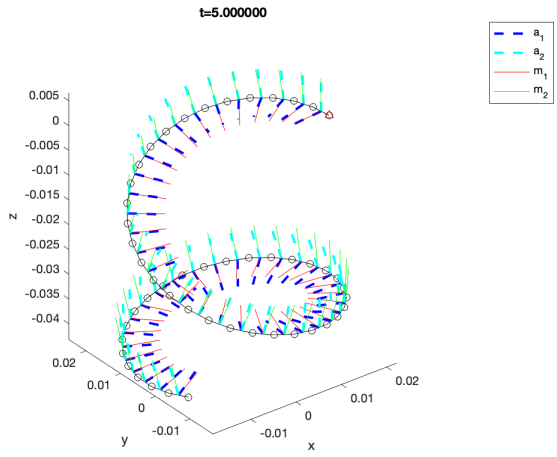

Fig. 2. Naturally curved rod position at t = 1.0s.)[1]

After five seconds it seems little has changed, but upon closer inspection, one can observe that the rod hangs slightly lower due to its weight. If the Young's Modulus of the rod is lowered, one can observe more significant deformation. The minute oscillations in the z-position of the last node are plotted below:

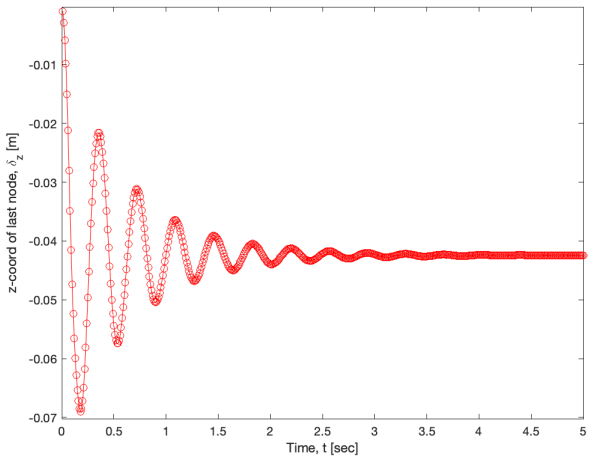

Fig. 3. Z-position of the last node over time.[1]

Note that the z-position of the rod begins oscillating when released, but levels out to a steady state position of roughly -0.042m lower than its starting position.

REFERENCES

[1] K. Jawed, Discretized Structures Notes.Structures Computer Interaction Lab. 2023.