Notes

Discrete Simulation of Slender Structures

M. Khalid Jawed
Sangmin Lim

Contents

Preface

Kirchhoff introduced a celebrated model for elastic rods in 1859 [1]. A major advancement in the mechanics of elastic rods came in 2008 with the introduction of Discrete Elastic Rods (DER) [2, 3] – a computationally efficient algorithm that can seamlessly capture the geometrically nonlinear dynamics of elastic rods. While the method was originally introduced by the computer graphics community, it has been successfully employed in the context of mechanical engineering problems since 2014 [4].

The original DER papers [2, 3] are fairly short in length, yet packed with advanced concepts from discrete differential geometry. For a detailed exposition of this method, we wrote a book-brief describing the DER method [5]. The book elaborates the theoretical foundation in detail and may not be suitable as a textbook for a 10-week long (1 quarter at UCLA) course. This set of notes is my attempt at a practical introduction of DER. Computer implementation is emphasized throughout the notes.

The student should be able to implement similar methods for plates [6], shells [7], ribbons [8], and other mechanical structures after going through these course notes.

Chapter 1

Newton's method for nonlinear systems

1.1 Solving a Single Equation

Assume that we have to solve a scalar equation

$$f(x) = 0, \tag{1.1}$$

whose solution is x^*. In Newton's method (also known as Newton-Raphson's method), we start with a guess x_0 and keep updating our approximation to the solution using the following rule:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \tag{1.2}$$

where $f'(x_n) = \frac{\partial f}{\partial x}|_{x_n}$ is the derivative of the function $f(x)$ with respect to x evaluated at $x = x_n$. After reaching the desired tolerance such that $|f(x_n)| < \epsilon$ where ϵ is a small number, we set the solution $x^* = x_n$.

Programming Implementation: As shown below, we can formulate a simple algorithm that implements Newton's method where the functions `f` and `df` evaluate $f(x)$ and $f'(x)$.

while $\texttt{err} > \epsilon$

$$\Delta x = \texttt{f}(x)/\texttt{df}(x)$$
$$x = x - \Delta x$$
$$\texttt{err} = \texttt{abs}(\texttt{f}(x))$$

end of while

A MATLAB code that implements the above algorithm for $f(x) = x^2 - 3x + 2$ is shown in Fig. 1.1. Note that there are two solutions to $f(x) = 0$: $x = 1$ and $x = 2$. Depending on the initial guess, Newton's method will reach one of these two solutions. However, it will never give us both of the solutions.

```
% Filename: newton1D.m
% Demo of Newton-Raphson's method in 1D
%
% f(x) = x^2 - 3*x + 2 = 0
% f'(x) = 2*x - 3

function x = newton1D(  )

% Guess solution
x = 5;

% Tolerance
eps = 1e-6;

% Error
err = eps*10; % initialize to a value larger than eps

while err > eps
    deltaX = f(x) / df(x);
    x = x - deltaX;
    err = abs( f(x) );
end

fprintf('Solution is %f\n', x);

end

function s = f(x)
s = x^2 - 3*x + 2;
end

function s = df(x)
s = 2*x - 3;
end
```

Figure 1.1: Code snippet that implements Newton's method for a scalar equation.

1.2 Solving a System of Equations

Assume that we have a system of N nonlinear equations involving N variables:

$$f_1(x_1, x_2, \ldots, x_N) = 0, \tag{1.3}$$
$$f_2(x_1, x_2, \ldots, x_N) = 0, \tag{1.4}$$
$$f_N(x_1, x_2, \ldots, x_N) = 0, \tag{1.5}$$

which can be written succinctly as

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}, \tag{1.6}$$

where

$$\mathbf{f} = \begin{bmatrix} f_1 \\ f_2 \\ \dots \\ f_N \end{bmatrix}, \qquad \text{and} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{bmatrix}. \tag{1.7}$$

The multivariable counterpart to the scalar derivative ($f'(x)$ in § 1.1) is the Jacobian matrix of size $N \times N$:

$$\mathbb{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_N} \\ \dots & \dots & \dots & \dots \\ \frac{\partial f_N}{\partial x_1} & \frac{\partial f_N}{\partial x_2} & \dots & \frac{\partial f_N}{\partial x_N} \end{bmatrix}.$$

In order to solve Eq. 1.5, we will start with a guess to the solution $\mathbf{x}_0$ (which is a vector of size N) and keep updating it until reaching a desired tolerance. The update rule is

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbb{J}(\mathbf{x}) \backslash \mathbf{f}(\mathbf{x}), \tag{1.8}$$

where $\mathbb{J}(\mathbf{x}) \backslash \mathbf{f}(\mathbf{x})$ is the solution to the system of linear equations: $\mathbb{J}(\mathbf{x})\Delta\mathbf{x} = \mathbf{f}(\mathbf{x})$.

Programming Implementation: As shown below, we can formulate a simple algorithm that implements Newton's method where the functions `f` and `J` evaluate $\mathbf{f}(\mathbf{x})$ and $\mathbb{J}(\mathbf{x})$.

while $\texttt{err} > \epsilon$

$$\Delta\mathbf{x} = \texttt{J}(\mathbf{x}) \backslash \texttt{f}(\mathbf{x})$$
$$\mathbf{x} = \mathbf{x} - \Delta\mathbf{x}$$
$$\texttt{err} = \texttt{sum}(\texttt{abs}(\texttt{f}(\mathbf{x})))$$

end of while

A MATLAB code that implements the above algorithm for the following system of equations is shown in Fig. 1.2.

$$f_1 \equiv \quad x_1^3 + x_2 \quad = 0, \tag{1.9}$$
$$f_2 \equiv \quad -x_1 + x_2^3 + 1 \quad = 0, \tag{1.10}$$

The Jacobian is

$$\mathbb{J} = \begin{bmatrix} 3x_1^2 & 1 \\ -1 & 3x_2^2 \end{bmatrix}.$$

```
% Filename: newton2D.m
% Demo of Newton-Raphson's method in 2D
%
% f(x) = [   x1^3+ x2    - 1;
%           -x1   + x2^3 + 1  ]
%
% J(x) = [  3x1^2,   1;
%           -1,      -3x2^2   ]
%

function x = newton2D(  )

% Guess solution
x = [0.5; 0.5];

% Tolerance
eps = 1e-6;

% Error
err = eps*10; % initialize to a value larger than eps

while err > eps
    deltaX = J(x) \ f(x);
    x = x - deltaX;
    err = abs( sum( f(x) ) );
end

fprintf('Solution is x1=%f, x2=%f\n', x(1), x(2));

end

function s = f(x)
s = [x(1)^3 + x(2) - 1; ...
    -x(1) + x(2)^3 + 1];
end

function s = J(x)
s = [   3*x(1)^2,       1;      ...
        -1,             3*x(2)^2];
end
```

Figure 1.2: Code snippet that implements Newton's method for a system of two equations.

Chapter 2

Simulation of Mass-Spring-Damper System

In this Chapter, we will restrict ourselves to systems with a single degree of freedom.[1] The system comprised of a mass m, a spring with stiffness k, and a dashpot with viscous damping coefficient b can be fully described by a single variable, x, denoting the position of the mass. An external force $F = F_0 \sin(\omega t)$, where ω is the driving frequency and F_0 is the driving amplitude, is acting on the mass. We want to simulate the location of the mass, x, as a function of time. In order to develop this simulation, we will first write down the equation of motion in continuous form and then present it in discrete form for implementation in code.

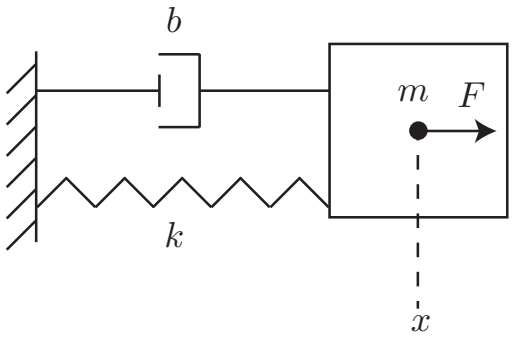

Figure 2.1: A mass-spring-damper system.

2.1 Continuous equations of motion

Let us first tabulate the forces acting on the mass:

1. spring force, $-kx$,
2. damping force, $-b\dot{x}$, and

[1] You may find the wikipedia article on harmonic oscillator useful: https://en.wikipedia.org/wiki/Harmonic_oscillator

3. the external force, $F = F_0 \sin(\omega t)$.

Newton's second law of motion says that the sum of forces is equal to mass times acceleration, and this gives our equation of motion:

$$m\ddot{x} = -kx - b\dot{x} + F_0 \sin(\omega t) \implies m\ddot{x} + kx + b\dot{x} - F_0 \sin(\omega t) = 0, \tag{2.1}$$

We now have to write a software to solve Eq. 2.1.

2.2 Discrete Equations of Motion

In our simulation, we will march forward in time (with time step size Δt) and, at each time step from $t = t_k$ to $t = t_{k+1} = t_k + \Delta t$, we have to solve for the position, $x(t_{k+1})$, and velocity, $\dot{x}(t_{k+1})$. The position and velocity from the previous time step $(x(t_k), \dot{x}(t_k))$ are known.

The discretized form of Eq. 2.1 to move from $t = t_k$ to $t = t_{k+1} = t_k + \Delta t$ is

$$\frac{m}{\Delta t}\left[\frac{x(t_{k+1}) - x(t_k)}{\Delta t} - \dot{x}(t_k)\right] + kx(t_{k+1}) + b\frac{x(t_{k+1}) - x(t_k)}{\Delta t} - F_0 \sin(\omega\, t_{k+1}) = 0, \tag{2.2}$$

which can be solved using Newton's method to compute $x(t_{k+1})$. Note that the velocity $\dot{x}(t_{k+1}) = \frac{x(t_{k+1}) - x(t_k)}{\Delta t}$ can be trivially computed.

We took an *implicit* approach in Eq. 2.2. In an *explicit* method, the discretized equation would be

$$\frac{m}{\Delta t}\left[\frac{x(t_{k+1}) - x(t_k)}{\Delta t} - \dot{x}(t_k)\right] + kx(t_k) + b\dot{x}(t_k) - F_0 \sin(\omega\, t_k) = 0, \tag{2.3}$$

which can be solved for $x(t_{k+1})$ by direct algebraic manipulation without the need for Newton's method. Despite its simplicity, explicit methods typically require very small time step (Δt) for convergence and ultimately require longer computation time.

2.3 Programming Implementation and Example

For our numerical simulation, we must be given the initial conditions, i.e. the position and velocity at time $t = 0$. We will then step forward in time (with time step Δt) and, at each time step, solve Eq. 2.2 adopting the approach described in § 1.1. Here,

$$f(x(t_{k+1})) \equiv \frac{m}{\Delta t}\left[\frac{x(t_{k+1}) - x(t_k)}{\Delta t} - \dot{x}(t_k)\right] + kx(t_{k+1}) + b\frac{x(t_{k+1}) - x(t_k)}{\Delta t} - F_0 \sin(\omega\, t_{k+1}), \tag{2.4}$$

and

$$f'(x(t_{k+1})) = \frac{m}{\Delta t^2} + k + \frac{b}{\Delta t}. \tag{2.5}$$

The algorithm is comprised of the following steps, where the functions `f` and `df` evaluate $f(x)$ and $f'(x)$, respectively.

$t = 0 : \Delta t :$ `maxTime` % container for time
$x =$ `zeros( size (`t`) )` % container for position
$u =$ `zeros( size (`t`) )` % container for velocity
$x(1) = 0$ % initial condition
$u(1) = 0$ % initial condition
for $k = 1 :$ `length`$(t) - 1$ % loop over time steps
    $x_{\text{old}} = x(k)$
    $u_{\text{old}} = u(k)$
    $x_{\text{new}} = x_{\text{old}}$ % guess solution
    while `err` $> \epsilon$
        $\Delta x =$ `f`(x_{new})/`df`$(_{\text{new}})$
        $x_{\text{new}} = x_{\text{new}} - \Delta x$
        `err` $=$ `abs`(`sum`(`f`(x_{new})))
    end of while
    $u_{\text{new}} = \dfrac{x_{\text{new}} - x_{\text{old}}}{\Delta t}$
    $x(k+1) = x_{\text{new}}$ % store the solution
    $u(k+1) = u_{\text{new}}$ % store the solution
end of for

Assume that $m = 1$ kg, $k = 1$ N/m, $b = 10$ Ns/m, $F_0 = 0.1$ N, and $\omega = 1$ Hz. Fig. 2.2 shows the solution x as a function of time t. The code that implements this simulation is shown in Fig. 2.3.

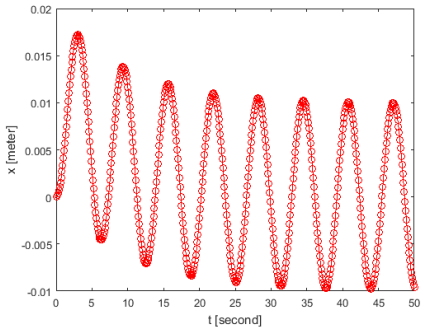

Figure 2.2: Position of the mass as a function of time.

```
% Filename: massSpringDamper.m
% Simulation of a mass-spring-damper system
function massSpringDamper()
global m K b F0 omega dt

m = 1; % mass
K = 1; % spring constant
b = 10; % viscous damping coefficient
F0 = 0.1; % driving amplitude
omega = 1; % driving frequency
maxTime = 50; % total time of simulation
dt = 1e-1; % discrete time step
eps = 1e-6 * F0; % error tolerance

t = 0:dt:maxTime; % time
x = zeros(size(t)); % position
u = zeros(size(t)); % velocity

% Initial condition: position and velocity are zero
x(1) = 0;
u(1) = 0;

for k=1:length(t)-1 % march over time steps
    x_old = x(k);
    u_old = u(k);
    t_new = t(k+1);

    x_new = x_old; % guess solution
    err = eps * 100; % initialize to a large value
    while err > eps
        deltaX = f(x_new, x_old, u_old, t_new) / ...
            df(x_new, x_old, u_old, t_new);
        x_new = x_new - deltaX;
        err = abs( f(x_new, x_old, u_old, t_new) );
    end

    u_new = (x_new - x_old) / dt;

    x(k+1) = x_new; % store solution
    u(k+1) = u_new;
end

% Plot it
plot(t, x, 'ro-'); xlabel('t [second]'); ylabel('x [meter]');
end

function s = f(x_new, x_old, u_old, t_new)
global m K b F0 omega dt
s = m/dt * ( (x_new-x_old)/dt - u_old ) + K*x_new + b*(x_new-x_old)/dt ...
    - F0*sin(omega*t_new);
end

function s = df(x_new, x_old, u_old, t_new)
global m K b dt
s = m/dt^2 + K + b/dt;
end
```

Figure 2.3: Mass-spring-damper simulator.

Chapter 3

Simulation of Multi-Mass Spring Damper System

In Ch. 2, we considered a system with single degree of freedom (DOF). In this Chapter, we will analyze a multi-DOF system comprised of multiple masses, springs, and dampers. In Fig. 3.1, such a system consisting of three sets of mass-spring-damper is shown. The system has three DOFs: $\tilde{x}_1, \tilde{x}_2$, and $\tilde{x}_3$, representing the locations of the masses along x-axis. If the locations in undeformed/natural configuration are x_{10}, x_{20}, and x_{30}, we can introduce three new variables: $x_1 = \tilde{x}_1 - x_{10}$, $x_2 = \tilde{x}_2 - x_{20}$, and $x_3 = \tilde{x}_3 - x_{30}$, that represent the displacements from their natural positions. Using x_1, x_2, and x_3 as DOFs simplify the algebra in the equations of motion, and, therefore, we choose to use these variables as our DOFs.

3.1 Continuous Equations of Motion

Let us tabulate the forces on the first mass m_1:

1. spring forces, $-k_1 x_1$ (left spring) and $k_2 (x_2 - x_1)$ (right spring),
2. damping forces, $-b_1 \dot{x}_1$ (left damper) and $b_2 (\dot{x}_2 - \dot{x}_1)$ (right damper), and
3. the external force, $F_1 = F_0^1 \sin(\omega_1 t)$,

where k_i $(i = 1, 2, 3)$ is the spring stiffness, b_i is the viscous damping coefficient, F_0^i is the driving amplitude, and ω_i is the driving frequency.

Recalling Eq. 2.1 from the single DOF case, the continuous version of the equation of motion for x_1 in this multi-DOF system is

$$m_1 \ddot{x}_1 = -k_1 x_1 + k_2 (x_2 - x_1) - b_1 \dot{x}_1 + b_2 (\dot{x}_2 - \dot{x}_1) + F_0^1 \sin(\omega_1 t), \tag{3.1}$$

After rearranging, the above equation can be rewritten as

$$m_1 \ddot{x}_1 + (k_1 + k_2) x_1 - k_2 x_2 + (b_1 + b_2) \dot{x}_1 - b_2 \dot{x}_2 - F_0^1 \sin(\omega_1 t) = 0, \tag{3.2}$$

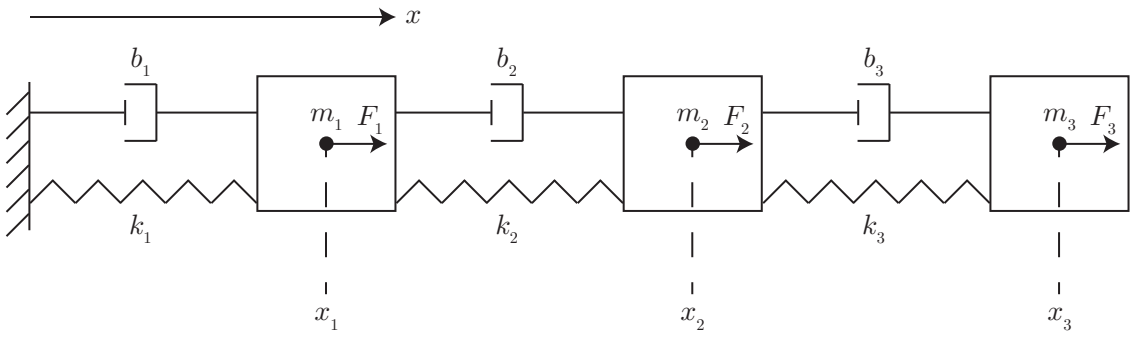

Figure 3.1: Multi-mass spring damper system in 1D.

For x_2, the equation of motion is

$$m_2\ddot{x}_2 + (k_2+k_3)x_2 - k_2x_1 - k_3x_3 + (b_2+b_3)\dot{x}_2 - b_2\dot{x}_1 - b_3\dot{x}_3 - F_0^2\sin(\omega_2 t) = 0, \quad (3.3)$$

and for x_3, the equation of motion is

$$m_3\ddot{x}_3 + k_3x_3 - k_3x_2 + b_3\dot{x}_3 - b_3\dot{x}_2 - F_0^3\sin(\omega_3 t) = 0 \quad (3.4)$$

In case of a general N DOF system comprised of N masses, springs, and dampers, Eq. 3.3 can be generalized for the i-th DOF ($i = 2, \ldots, N-1$):

$$\begin{aligned} m_i\ddot{x}_i + &(k_i+k_{i+1})x_i - k_ix_{i-1} - k_{i+1}x_{i+1} \\ &+ (b_i+b_{i+1})\dot{x}_i - b_i\dot{x}_{i-1} - b_{i+1}\dot{x}_{i+1} - F_0^i\sin(\omega_i t) = 0, \end{aligned} \quad (3.5)$$

and the equations of motion for $i = 1$ and $i = N$ can be handled as special cases:

$$\begin{aligned} \text{For } i=1: \quad & m\ddot{x}_i + (k_i+k_{i+1})x_{i+1} - k_{i+1}x_{i+1} + (b_i+b_{i+1})\dot{x}_i - b_{i+1}\dot{x}_{i+1} - F_0^i\sin(\omega_i t) = 0, \\ \text{For } i=N: \quad & m\ddot{x}_i + k_ix_i - k_ix_{i-1} + b_i\dot{x}_i - b_i\dot{x}_{i-1} - F_0^i\sin(\omega_i t) = 0. \end{aligned}$$

Note that the above two equations can be deduced from Eq. 3.5 by ignoring terms that involve x_0 and x_{N+1}.

3.2 Discrete Equations of Motion

Eq. 3.5 can be discretized as

$$\begin{aligned} f_i \equiv &\frac{m_i}{\Delta t}\left[\frac{x_i(t_{k+1}) - x_i(t_k)}{\Delta t} - \dot{x}_i(t_k)\right] + \\ &(k_i+k_{i+1})\,x_i(t_{k+1}) - k_ix_{i-1} - k_{i+1}x_{i+1} + \\ &(b_i+b_{i+1})\,\frac{x_i(t_{k+1}) - x_i(t_k)}{\Delta t} - b_i\frac{x_{i-1}(t_{k+1}) - x_{i-1}(t_k)}{\Delta t} - b_{i+1}\frac{x_{i+1}(t_{k+1}) - x_{i+1}(t_k)}{\Delta t} \\ &- F_0^i\sin(\omega_i\,t_{k+1}) = 0, \end{aligned} \quad (3.6)$$

where $x_i(t_{k+1})$ are the N unknowns and the positions ($x_i(t_k)$) and velocities ($\dot{x}_i(t_{k+1})$) are known from the previous time step. Upon solving the system of equations provided by Eq. 3.6 for the new positions $x_i(t_{k+1})$, the new velocities can be obtained simply from

$$\dot{x}_i(t_{k+1}) = \frac{x_i(t_{k+1}) - x_i(t_k)}{\Delta t}.$$

3.3 Programming Implementation and Example

Newton's method for a system of equations (§ 1.2) requires the Jacobian associated with the system of equations:

$$\mathbb{J}_{ij} = \frac{\partial f_i}{\partial x_j} = \begin{cases} \frac{m_i}{\Delta t^2} + (k_i + k_{i+1}) + \frac{b_i + b_{i+1}}{\Delta t}, & \text{for } j = i \\ -k_i - \frac{b_i}{\Delta t}, & \text{for } j = i - 1 \\ -k_{i+1} - \frac{b_{i+1}}{\Delta t}, & \text{for } j = i + 1 \\ 0, & \text{otherwise.} \end{cases} \tag{3.7}$$

We can now develop an algorithm that is almost identical to the one provided in Ch. 2, except the number of DOFs. Instead of solving a single scalar equation at each time step, we will be required to solve a system of N equations.

Example: Assume that the physical parameters are provided below:

- $m_1 = 10$ kg, $m_2 = 10^{-2}$ kg, and $m_3 = 10^{-3}$ kg,
- $k_1 = 1$ N/m, $k_2 = 5$ N/m, and $k_3 = 0.1$ N/m,
- $b_1 = 10$ Ns/m, $b_2 = 50$ Ns/m, and $b_3 = 100$ Ns/m,
- $F_0^1 = 0.1$ N, $F_0^2 = 10$ N, and $F_0^3 = 1$ N,
- $\omega_1 = 0.1$ rad/s, $\omega_2 = 10$ rad/s, and $\omega_3 = 2$ rad/s, and
- $x_{10} = 1$ m, $x_{20} = 2$ m, and $x_{30} = 3$ m.

A MATLAB code file that implements the above example (`massSpringDamper_multi.m`) is provided with these notes. Fig. 3.2 shows the positions, x_i, as functions of time.

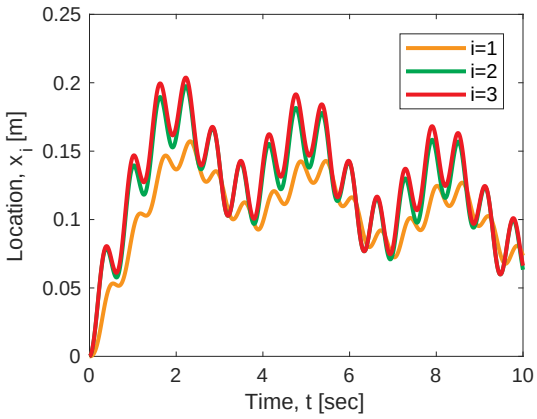

Figure 3.2: Positions of the masses as a function of time.

Chapter 4

Conservative Force and Potential Energy

While simulating a system with a degree of freedom vector $\mathbf{q}$, the forces acting on a degree of freedom (DOF) can often be *conservative*. This is specially the case for elastic structures, e.g. elastic rods. In case of conservative forces, if the *potential* energy associated with the conservative forces is $E_{\text{potential}}$, the force acting on the i-th DOF, q_i, is $F_i = \frac{\partial E_{\text{potential}}}{\partial q_i}$. In this Chapter, we will do a series of problems on simulation of elastic beams.

4.1 General Formulation

Let us assume that the system with DOF vector $\mathbf{q}$ of size N has a potential energy $E_{\text{potential}}$. The lumped mass associated with q_i is m_i. Note that the unit of m_i is kg if q_i corresponds to position and kg-m^2 if the DOF represents rotation. For convenience, we can formulate a lumped mass matrix $\mathbf{M}$ of size $N \times N$ whose diagonal components are m_i $(i = 1, \ldots, N)$ and non-diagonal components are zero. The equations of motion of this system can be succinctly written as

$$\mathbf{M}\ddot{\mathbf{q}} + \frac{\partial E_{\text{potential}}}{\partial \mathbf{q}} = \mathbf{0}, \tag{4.1}$$

and, alternatively, the equation of motion for i-th DOF is

$$m_i \ddot{q}_i + \frac{\partial E_{\text{potential}}}{\partial q_i} = 0. \tag{4.2}$$

For generality, assume a damping force $-c_i \dot{q}_i$ is also acting on q_i, where c_i is related to viscous damping. Eq. 4.2 becomes

$$m_i \ddot{\mathbf{q}} + \frac{\partial E_{\text{potential}}}{\partial q_i} + c_i \dot{q}_i = 0. \tag{4.3}$$

The discrete version of the above equation to march from $t = t_k$ to $t = t_{k+1} = t_k + \Delta t$ is

$$f_i \equiv \frac{m_i}{\Delta t}\left[\frac{q_i(t_{k+1}) - q_i(t_k)}{\Delta t} - \dot{q}_i(t_k)\right] + \frac{\partial E_{\text{potential}}}{\partial q_i} + c_i \frac{q_i(t_{k+1}) - q_i(t_k)}{\Delta t} = 0, \tag{4.4}$$

and these N equations can be solved for the N unknowns: $q_i(t_{k+1})$. The new velocities can be computed from $\dot{q}_i(t_{k+1}) = \frac{q_i(t_{k+1}) - q_i(t_k)}{\Delta t}$.

Newton's method requires the Jacobian:

$$\mathbb{J}_{ij} = \frac{\partial f_i}{\partial q_j} = \mathbb{J}_{ij}^{\text{inertia}} + \mathbb{J}_{ij}^{\text{potential}} + \mathbb{J}_{ij}^{\text{viscous}}, \tag{4.5}$$

where

$$\mathbb{J}_{ij}^{\text{inertia}} = \frac{m_i}{\Delta t^2}\delta_{ij}, \tag{4.6}$$

$$\mathbb{J}_{ij}^{\text{potential}} = \frac{\partial^2 E_{\text{potential}}}{\partial q_i \partial q_j}, \tag{4.7}$$

$$\mathbb{J}_{ij}^{\text{viscous}} = \frac{c_i}{\Delta t}\delta_{ij}. \tag{4.8}$$

We can easily add external forces, F_i^{ext}, (e.g. van der Waals) to Eq. 4.4 and these forces can be a functions of $\mathbf{q}$ and $\dot{\mathbf{q}}$. Often times, the Jacobian ($\frac{\partial F^{\text{ext}}}{\partial q_j}$) associated with these terms is not known - in that case, we can ignore it (i.e. $\frac{\partial F^{\text{ext}}}{\partial q_j} \to 0$).

In the following, we will do three problems related to bending of a beam in 2D, before considering deformation of elastic rods. A beam can only undergo bending whereas a rod can bend and twist. We will also include axial stretch (i.e. extensibility) of beams and rods in our formulation; however, it is negligible in practical examples.

4.2 Rigid Spheres and Elastic Beam Falling in Viscous Flow

Referring to Fig. 4.1, let us consider three rigid spheres on an elastic beam that is falling under gravity in a viscous fluid. The viscosity of the fluid is $\mu = 1000$ Pa-s. The radii of the spheres are $R_1 = 0.005$ m, $R_2 = 0.025$ m, and $R_3 = 0.005$ m. The density of the spheres is $\rho_{\text{metal}} = 7000$ kg/m^3 whereas the fluid density is $\rho_{\text{fluid}} = 1000$ kg/m^3. The positions of the spheres are $(x_1, y_1), (x_2, y_2)$, and (x_3, y_3). We can construct a *degrees of freedom* (dof) vector,

$$\mathbf{q} = \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \end{bmatrix} \tag{4.9}$$

In this problem, in addition to gravity and viscous drag, we also have to include the elastic force of the beam. The elastic beam has a length, $l = 0.10$ m, cross-sectional radius, $r_0 = 0.001$ m, and Young's modulus, $E = 1.0 \times 10^9$ Pa. This corresponds to a stretching stiffness of $EA = E\,\pi\,r_0^2$ and bending stiffness of $EI = E\,\pi r_0^4/4$. We can consider the beam to be a combination of springs as shown in Fig. 4.1. The elastic energy of the beam is

$$E^{\text{elastic}} = E_1^s + E_2^s + E^b, \tag{4.10}$$

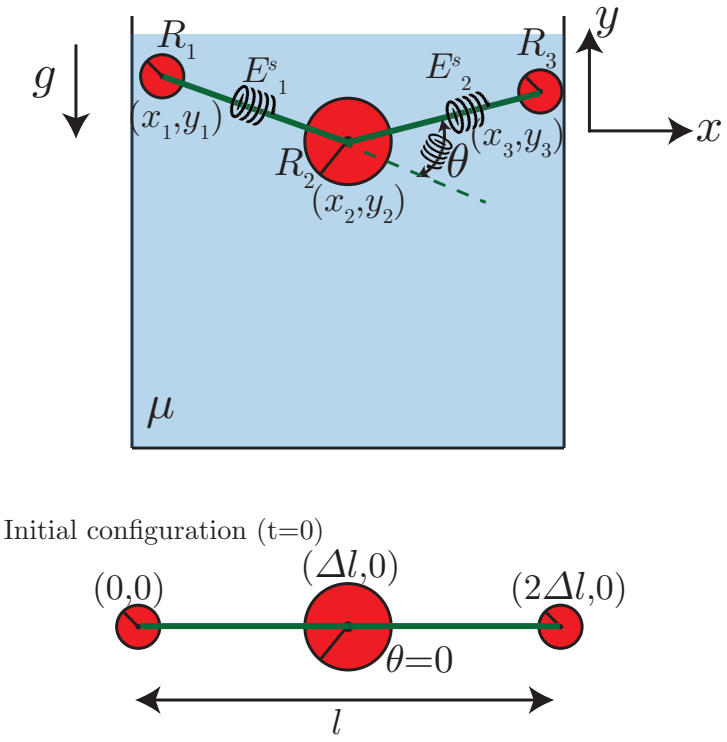

Figure 4.1: Rigid spheres attached to an elastic beam falling in viscous fluid.

where superscript s refers to *stretching*, b corresponds to bending, and

$$E_1^s = \frac{1}{2}EA\left(1 - \frac{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}}{\Delta l}\right)^2 \Delta l, \tag{4.11}$$

$$E_2^s = \frac{1}{2}EA\left(1 - \frac{\sqrt{(x_3-x_2)^2+(y_3-y_2)^2}}{\Delta l}\right)^2 \Delta l, \tag{4.12}$$

$$E^b = \frac{1}{2}\frac{EI}{\Delta l}\left(2\tan\left(\frac{\theta}{2}\right)\right)^2, \tag{4.13}$$

where $\Delta l = \frac{l}{2}$ is the length of each segment between two nodes and θ is the *turning angle*, as shown in Fig. 4.1. Note that θ can be expressed in terms of $x_1, y_1, x_2, y_2, x_3, y_3$. Since elastic force is conservative, the elastic force on a degree of freedom (dof), e.g. x_1, is the negative of the gradient of elastic energy with respect to that dof, e.g. $-\frac{\partial E^{\text{elastic}}}{\partial x_1}$. You do not need to analytically evaluate the elastic force; see Appendix for MATLAB functions that evaluate the gradient of the elastic energies.

Initially, at time $t = 0$, the entire system is at rest (i.e. velocity is zero) and the positions of the nodes are shown in Fig. 4.1. The entire structure starts to fall starting at $t = 0$ under gravity and, at the same time, it experiences a drag force from the fluid. First, write down the six equations of motion and their discretized

version. For convenience, the first two equations of motion are

$$m_1\ddot{x_1} = -(6\pi\mu R_1)\dot{x_1} - \frac{\partial E^{\text{elastic}}}{\partial x_1}, \tag{4.14}$$

$$m_1\ddot{y_1} = -W_1 - (6\pi\mu R_1)\dot{y_1} - \frac{\partial E^{\text{elastic}}}{\partial y_1}, \tag{4.15}$$

where $W_1 = \frac{4}{3}\pi R_1^3(\rho_{\text{metal}} - \rho_{\text{fluid}})g$, $g = 9.8$ m/s^2, $m_1 = \frac{4}{3}\pi R_1^3\rho_{\text{metal}}$. The discretized version is

$$\frac{m_1}{\Delta t}\left[\frac{x_1(t_{k+1}) - x_1(t_k)}{\Delta t} - \dot{x}_1(t_k)\right] = -(6\pi\mu R_1)\frac{x_1(t_{k+1}) - x_1(t_k)}{\Delta t} - \frac{\partial E^{\text{elastic}}}{\partial x_1},$$

$$\frac{m_1}{\Delta t}\left[\frac{y_1(t_{k+1}) - y_1(t_k)}{\Delta t} - \dot{y}_1(t_k)\right] = -W_1 - (6\pi\mu R_1)\frac{y_1(t_{k+1}) - y_1(t_k)}{\Delta t} - \frac{\partial E^{\text{elastic}}}{\partial y_1},$$

This can be re-written as

$$f_1 \equiv \frac{m_1}{\Delta t}\left[\frac{x_1(t_{k+1}) - x_1(t_k)}{\Delta t} - \dot{x}_1(t_k)\right] + (6\pi\mu R_1)\frac{x_1(t_{k+1}) - x_1(t_k)}{\Delta t} + \frac{\partial E^{\text{elastic}}}{\partial x_1} = 0,$$

$$f_2 \equiv \frac{m_1}{\Delta t}\left[\frac{y_1(t_{k+1}) - y_1(t_k)}{\Delta t} - \dot{y}_1(t_k)\right] + W_1 + (6\pi\mu R_1)\frac{y_1(t_{k+1}) - y_1(t_k)}{\Delta t} + \frac{\partial E^{\text{elastic}}}{\partial y_1} = 0.$$

For an *implicit* simulation, the elastic forces have to be evaluated using the positions at $t = t_{k+1}$. An implicit solution using Newton's method will require the Jacobian matrix (size 6×6). Write down the entries of the Jacobian matrix (or elaborate the steps to evaluate the Jacobian). It will involve terms such as $\frac{\partial^2 E^{\text{elastic}}}{\partial x_1 \partial y_1}$ (i.e. the *Hessian* of the energy). You do not need to analytically evaluate such terms; see Appendix for MATLAB functions that evaluate the Hessian of elastic energies.

Assignment 1: Write a solver that simulates the position and the velocity of the sphere as a function of time (between $0 \le t \le 10$ seconds) implicitly and explicitly. Use $\Delta t = 10^{-2}$ s for the implicit simulation and $\Delta t = 10^{-5}$ s for the explicit one. Additional questions to consider:

1. What happens to the turning angle if all the radii (R_1, R_2, R_3) are the same? Does your simulation agree with your intuition?

2. Try changing the time step size (Δt), particularly for your explicit simulation, and use the observation to elaborate the benefits and drawbacks of the explicit and implicit approach.

4.3 Generalized Case of Elastic Beam Falling in Viscous Flow

Generalize the implicit simulator such that the beam is composed of N nodes (N is an odd number). The sphere at the middle node has radius $R_{\text{mid}} = 0.025$ and all the

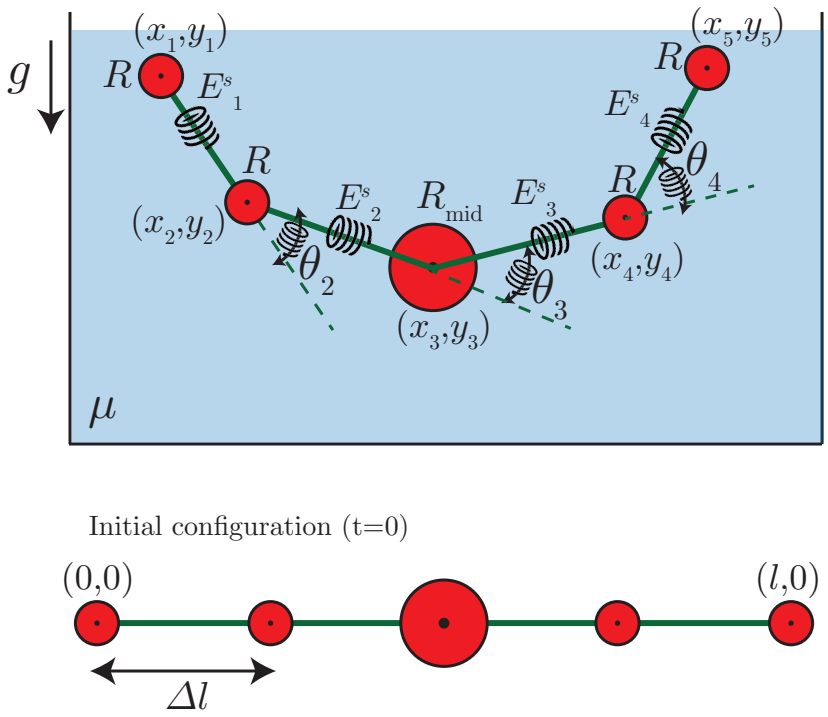

Figure 4.2: $N = 5$ rigid spheres attached to an elastic beam falling in viscous fluid.

other nodes have a sphere with radius $R = \Delta l/10$, where $\Delta l = l/(N-1)$ is the length of each discrete segment. Fig. 4.2 illustrates the system when $N = 5$. You are not required (but encouraged) to write an explicit solver for this problem.

Now we have $N-1$ stretching springs and $N-2$ bending springs. The elastic energy is

$$E^{\text{elastic}} = \sum_{j=1}^{N-1} E^s_j + \sum_{j=2}^{N-1} E^b_j, \tag{4.16}$$

where E^b_j is associated with the turning angle θ_j.

Assignment 2: Write a solver that simulates the system of the sphere as a function of time (between $0 \le t \le 50$ seconds) implicitly with $\Delta t = 10^{-2}$ s and $N = 21$.

1. Include two plots showing the vertical position and velocity of the middle node with time. What is the terminal velocity?

2. Include the final deformed shape of the beam.

3. Discuss the significance of spatial discretization (i.e. the number of nodes, N) and temporal discretization (i.e. time step size, Δt). Any simulation should be sufficiently discretized such that the quantifiable metrics, e.g. terminal velocity, do not vary much if N is increased and Δt is decreased. Include plots of terminal velocity vs. the number of nodes and terminal velocity vs. the time step size.

4.4 Elastic Beam Bending

Recall Euler-Bernoulli beam theory from elementary mechanics class. You look up the solution for a simply-supported aluminum beam subjected to a single point load. The bar has length $l = 1$ m and a constant, circular-tube cross section with outer radius $R = 0.013$ m and inner radius $r = 0.011$ m. A force, $P = 2000$ N, is applied 0.75 m away from the left-hand edge.The modulus of elasticity is $E = 70$ GPa for aluminum and I is the moment of inertia of the cross section, $I = \frac{\pi}{4}(R^4 - r^4)$.

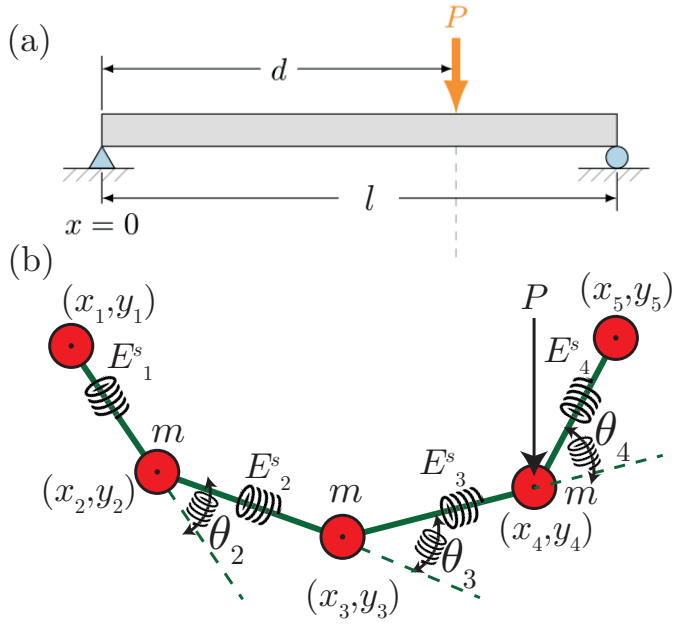

Figure 4.3: (a) Elastic beam and (b) its discrete representation.

We can represent the beam as a mass-spring system with a mass m located at each node where

$$m = \pi(R^2 - r^2)l\rho/(N-1) \tag{4.17}$$

and density of aluminum is $\rho = 2700$ kg/m^3. The formulation of the elastic energy remains the same. Instead of gravity and viscous drag as external forces, the only external force is the force P applied at a node that is located 0.75m away from the first node. If no node is exactly 0.75 m away, take the node that is closest to the desired distance.

Also note that the first node is constrained along both x and y-axes and the last node is constrained along y-axis. For these three constrained degrees of freedom, the governing equations are simply

$$x_1(t_{k+1}) = 0, \tag{4.18}$$
$$y_1(t_{k+1}) = 0, \tag{4.19}$$
$$y_N(t_{k+1}) = 0. \tag{4.20}$$

Assignment 3: Write a solver that simulates the beam as a function of time (between $0 \leq t \leq 1$ seconds) implicitly. Use $\Delta t = 10^{-2}$ s for the implicit simulation and $N = 50$.

1. Plot the maximum vertical displacement, $y_{\max}$, of the beam as a function of time. Depending on your coordinate system, $y_{\max}$ may be negative. Does $y_{\max}$ eventually reach a steady value? Examine the accuracy of your simulation against the theoretical prediction from Euler beam theory:

$$y_{\max} = \frac{Pc\left(L^2 - c^2\right)^{1.5}}{9\sqrt{3}EI\,l} \qquad \text{where} \qquad c = \min(d, l - d) \tag{4.21}$$

2. What is the benefit of your simulation over the predictions from beam theory? To address this, consider a higher load $P = 20000$ such that the beam undergoes large deformation. Compare the simulated result against the prediction from beam theory in Eq. 4.21. Euler beam theory is only valid for small deformation whereas your simulation, if done correctly, should be able to handle large deformation. Optionally, you can make a plot of P vs. $y_{\max}$ using data from both simulation and beam theory and quantify the value of P where the two solutions begin to diverge.

Chapter 5

Discrete Simulation of an Elastic Beam

This note describes a numerical procedure for dynamic simulation of a linear elastic rod in 2D (i.e. an extensible beam). At the heart of the simulation is a geometrically-exact mass-spring representation of the rod. This allows us to represent the elastic energy as packets of discrete bending and stretching energies associated with the nodes on the rod and formulate the equations of motion at each node. This procedure follows the principle adopted by a number of widely used algorithms for simulation of slender structures, e.g. rods [2, 3], ribbons [8], plates [6], and shells [7].

This formulation is the 2D version of the Discrete Elastic Rods [2, 3] (DER) algorithm, a robust and efficient simulation of Kirchhoff elastic rods [9, 1]. Although more commonly used in computer graphics, this simulation has been adapted to study engineering problems [4].

5.1 Kinematics of a Beam

The beam (located on $x-y$ plane) is discretized into N *nodes* $\mathbf{x}_k = [x_k \; y_k]^T$, where $1 \leq k \leq N$. This is our degree of freedom (DOF) vector of size $2N$. These nodes corresponds to $N-1$ *edges*: $\mathbf{e}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$, where $1 \leq k \leq N-1$. In the undeformed state, the length of each edge is $dl = L/(N-1)$, where L is the arc-length of the undeformed rod. We assumed that the length of each edge is the same in the undeformed state. The Young's modulus of the linear elastic material is E, area moment of inertia is I, and cross-sectional area is A. For a rod with circular cross-section, $A = \pi r_0^2$ and $I = \pi r_0^4/4$, where r_0 is the cross-sectional radius.

5.2 Elastic Energy

The elastic energy can be assumed to be a sum of bending and stretching energies. In case of a 3D rod, we will have to also account for the twisting energy. This Chapter is restricted to the 2D case. The complete 3D formulation is provided in Bergou et al. [3], which is a significantly improved version of the algorithm in Ref. [2]. Our

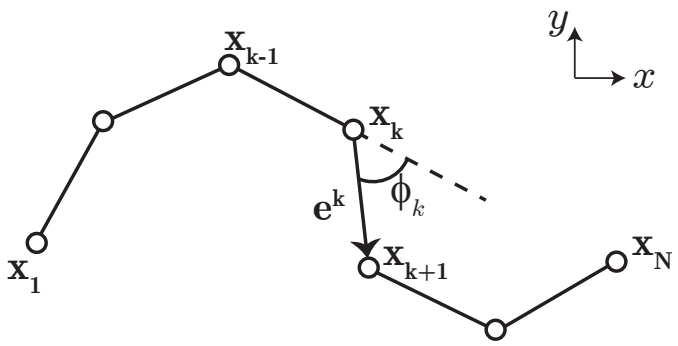

Figure 5.1: Schematic of a discrete rod in 2D.

goal in the next two sections is to express the elastic energies in terms of the nodal coordinates $\mathbf{x}_k$ (our DOF vector).

5.2.1 Discrete Bending energy

The total bending energy of the rod is

$$E_b = \sum_{k=2}^{N-1} E_{b,k}, \tag{5.1}$$

where the bending energy at each internal node $\mathbf{x}_k$ ($k = 2, \ldots, N-1$) is

$$E_{b,k} = \frac{1}{2}\frac{EI}{dl}\left(\kappa_k - \kappa_k^0\right)^2, \tag{5.2}$$

the *discrete* curvature at $\mathbf{x}_k$ is [also see Fig. 5.2]

$$\kappa_k = 2\tan(\phi_k/2), \tag{5.3}$$

the curvature in undeformed state at node $\mathbf{x}_k$ is κ_k^0 (for a naturally straight rod, $\kappa_k^0 = 0$), and the *turning angle* is

$$\phi_k = \tan^{-1}\frac{(\mathbf{e}_{k-1} \times \mathbf{e}_k) \cdot \hat{e}_z}{\mathbf{e}_{k-1} \cdot \mathbf{e}_k}. \tag{5.4}$$

The numerator on the right hand size is the z-component of the cross-product between $\mathbf{e}_{k-1}$ and $\mathbf{e}_k$. Note that *discrete* curvature is dimensionless whereas the unit of natural curvature in standard sense is 1/length. Referring to Fig. 5.2, the standard definition of curvature is equal to κ_k/dl. Also note that, if the beam is naturally straight (i.e. $\kappa_k^0 = 0$ for all values of k), Eq. 5.2 reduces to Eq. 4.13. Refer to the Appendix for MATLAB codes to compute the gradient and Hessian of the bending energies.

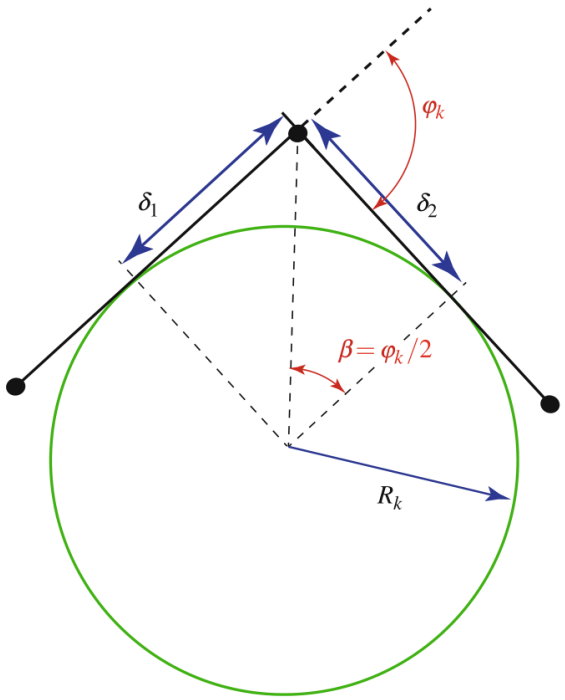

Figure 5.2: The osculating circle of radius $R_k = dl/\kappa_k$ ($\delta_1 = \delta_2 \equiv dl$) is constructed by projecting perpendicular lines from the edges. Elementary geometry is all that is needed to show that $\kappa_k = 2\tan\left(\frac{\phi_k}{2}\right)$. Adapted from [5].

5.2.2 Discrete Stretching energy

The total stretching energy of the rod is

$$E_s = \sum_{k=1}^{N-1} E_s^k, \tag{5.5}$$

where the stretching energy at each edge $\mathbf{e}_k$ is

$$E_s^k = \frac{1}{2} EA\varepsilon^2 dl, \tag{5.6}$$

and the axial stretch at edge $\mathbf{e}_k$ is

$$\varepsilon = 1 - \frac{\sqrt{(x_{k+1} - x_k)^2 + (y_{k+1} - y_k)^2}}{dl}. \tag{5.7}$$

Physically, axial stretch corresponds to the change in length of an edge, normalized by the undeformed length of the edge. Note that the expression for stretching energy in Eq. 5.6 is identical, despite notational differences, to the one introduced in Eq. 4.11. See Appendix for codes to compute the gradient and Hessian of this energy.

5.3 Time Marching Scheme

For simulation of the rod, we have to compute the time evolution of the x- and y-coordinates of each node. For a total of N nodes, we have $2N$ DOFs:

$$[x_0, y_0, x_1, y_1, \ldots, x_k, y_k, \ldots, x_{N-1}, y_{N-1}]^T.$$

Therefore, we need $2N$ equations of motion that we derive next. Note that if some of the DOFs are *fixed*, we have to ignore the equations corresponding to the fixed DOFs and only solve for the *free* DOFs.

We have to formulate an algorithm to move from time $t = t_k$ to $t = t_{k+1}$ (with $t_{k+1} - t_k = \Delta t$). The DOFs at previous step $\mathbf{x}_k(t_k) = [x_k(t_k)\ y_k(t_k)]^T$ and the velocities $\mathbf{u}_k(t_k) = [u_k(t_k)\ v_k(t_k)]^T$ are known. We have to evaluate $\mathbf{x}_k(t_{k+1})$ as well as $\mathbf{u}_k(t_{k+1})$ by solving the following equations of motion. Each equation is essentially a statement of force balance: inertia - elastic forces = external forces.

$$m_k \frac{x_k(t_{k+1}) - x_k(t_k)}{\Delta t^2} - m_k \frac{u_k(t_k)}{\Delta t} + \frac{\partial}{\partial x_k}(E_b + E_s) = F^{ext}_{k,x}, \tag{5.8}$$

$$m_k \frac{y_k(t_{k+1}) - y_k(t_k)}{\Delta t^2} - m_k \frac{v_k(t_k)}{\Delta t} + \frac{\partial}{\partial y_k}(E_b + E_s) = F^{ext}_{k,y}, \tag{5.9}$$

where the first two terms represent inertia, the third term corresponds to elastic forces, and the remaining term on the right hand side is the external force $\mathbf{F}^{ext}_k = [F^{ext}_{k,x}\ F^{ext}_{k,y}]^T$ at node $\mathbf{x}_k$. The lumped mass m_k at node $\mathbf{x}_k$ is $\rho A dl$ (mass per volume is ρ) for the internal nodes ($0 < k < N-1$) and $\rho A dl/2$ for the end nodes ($k = 0$ and $k = N-1$). The system of equations above can be solved for the nodal coordinates $\mathbf{x}_k$. Velocities at $t = t_{k+1}$ can be readily evaluated: $u_k(t_{k+1}) = (x_k(t_{k+1}) - x_k(t_k))/\Delta t$, $v_k(t_{k+1}) = (y_k(t_{k+1}) - y_k(t_k))/\Delta t$.

The implementation of the numerical simulation can be significantly simplified by noting that Eqs. 5.8-5.9 can be re-written as

$$m_k \frac{x_k(t_{k+1}) - x_k(t_k)}{\Delta t^2} - m_k \frac{u_k(t_k)}{\Delta t} + \frac{\partial}{\partial x_k}\left(E_{b,k-1} + E_{b,k} + E_{b,k+1} + E^{k-1}_s + E^k_s\right) = F^{ext}_{k,x},$$

$$m_k \frac{y_k(t_{k+1}) - y_k(t_k)}{\Delta t^2} - m_k \frac{v_k(t_k)}{\Delta t} + \frac{\partial}{\partial y_k}\left(E_{b,k-1} + E_{b,k} + E_{b,k+1} + E^{k-1}_s + E^k_s\right) = F^{ext}_{k,y},$$

where we used Eq. 5.2 and Eq. 5.6 to note that

$$\frac{\partial}{\partial x_k} E_b = \frac{\partial}{\partial x_k}\left(E_{b,k-1} + E_{b,k} + E_{b,k+1}\right), \qquad \frac{\partial}{\partial y_k} E_b = \frac{\partial}{\partial y_k}\left(E_{b,k-1} + E_{b,k} + E_{b,k+1}\right),$$

$$\frac{\partial}{\partial x_k} E_s = \frac{\partial}{\partial x_k}\left(E^{k-1}_s + E^k_s\right), \qquad \frac{\partial}{\partial y_k} E_s = \frac{\partial}{\partial y_k}\left(E^{k-1}_s + E^k_s\right).$$

The external force depends on the specific problem we are simulating. For example, if the only external force is gravity, the equations of motion are

$$m_k \frac{x_k(t_{k+1}) - x_k(t_k)}{\Delta t^2} - m_k \frac{u_k(t_k)}{\Delta t} + \frac{\partial}{\partial x_k}\left(E_{b,k-1} + E_{b,k} + E_{b,k+1} + E^{k-1}_s + E^k_s\right) = 0,$$

$$m_k \frac{y_k(t_{k+1}) - y_k(t_k)}{\Delta t^2} - m_k \frac{v_k(t_k)}{\Delta t} + \frac{\partial}{\partial y_k}\left(E_{b,k-1} + E_{b,k} + E_{b,k+1} + E^{k-1}_s + E^k_s\right) = -m_k g,$$

where $g = 9.8$ m/s^2 is the acceleration due to gravity (along negative y-axis).

Newton-Raphson method can be used to solve this set of equations. The Jacobian matrix, $J_{ij} = \frac{\partial f_i}{\partial q_j}$, can be expanded to write

$$J_{ij} = \frac{m_i}{\Delta t^2}\delta_{ij} + \frac{\partial^2}{\partial q_i \partial q_j}\left(E_b + E_s\right), \tag{5.10}$$

where we treated the external forces *explicitly*, i.e. $\frac{\partial}{\partial q_j}F_i^{ext}$ is ignored. If gravity is the only external force, $\frac{\partial}{\partial q_j}F_i^{ext} = 0$ and the simulation is fully implicit.

Banded Nature of the Jacobian. Note that $\frac{\partial^2}{\partial q_i \partial q_j}E_b^k$ is non-zero only if $2k-2 \leq i, j \leq 2k+3$, i.e. q_i and q_j correspond to one of the following dof: $[x_{k-1}, y_{k-1}, x_k, y_k, x_{k+1}, y_{k+1}]^T$. Similarly, $\frac{\partial^2}{\partial q_i \partial q_j}E_s^k$ is non-zero only if $2k \leq i, j \leq 2k+3$, i.e. q_i and q_j correspond to one of the following dof: $[x_k, y_k, x_{k+1}, y_{k+1}]^T$. This implies that J is a *banded* matrix and the system of equations can be solved in $O(N)$ time.

Chapter 6

Discrete Twist

This Chapter presents a rapid introduction to the computation of twist in Discrete Elastic Rods (DER). Refer to Ref. [5], Chapter 5 in particular, for details.

Let us briefly review the twist in a rod in a continuous sense. Fig. 6.1(a), a rod with arc-length parameter, s, has two material directors $\mathbf{m}_1(s)$ and $\mathbf{m}_2(s)$ attached with the centerline. These two directors are mutually orthogonal as well as orthogonal to the tangent to the centerline. These two directors can be interpreted as two patches of paint along the rod, as shown in Fig. 6.1(a). Due to loading, the rod has deformed in Fig. 6.1(b) and the rotation angle from $\mathbf{m}_1(s)$ to $\mathbf{m}_1(s+\Delta s)$ is $\Delta\nu$. The continuous twist is

$$\tau(s) = \lim_{\Delta s \to 0} \frac{\Delta \nu}{\Delta s},$$

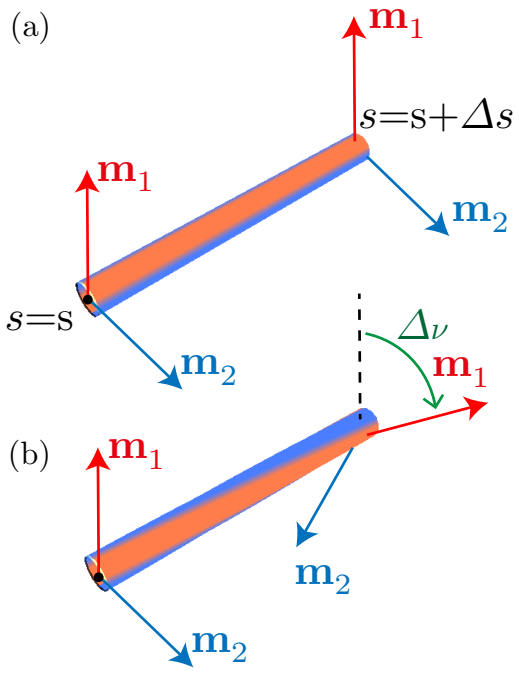

Figure 6.1: (a) Undeformed and (b) deformed shapes of a continuous rod.

which has unit of 1/length.

To implement this concept of twist in a discrete simulation of elastic rods, we need to understand computation of *discrete* twist. We will first go over some preliminary concepts and then provide the mathematical formulation for measurement of discrete twist.

6.1 Preliminaries

Orthonormal Adapted Frame: A discrete rod is composed of N nodes, $\mathbf{x}_k$, with $k = 1, \ldots, N$, and $N-1$ edges $\mathbf{e}^k = \mathbf{x}_{k+1} - \mathbf{x}_k$ with $k = 1, \ldots, N-1$. The *tangent* is the unit vector parallel to each edge, i.e. the k-th tangent is $\mathbf{t}^k = \mathbf{e}^k/|\mathbf{e}^k|$ where $|\mathbf{e}^k|$ is the norm of $\mathbf{e}^k$. An orthonormal frame (or basis) $(\mathbf{m}_1^k, \mathbf{m}_2^k, \mathbf{t}^k)$ is composed of three unit vectors that are orthogonal to one another. This implies $\mathbf{m}_1^k \times \mathbf{m}_2^k = \mathbf{t}^k$, $\mathbf{m}_2^k \times \mathbf{t}^k = \mathbf{m}_1^k$, and $\mathbf{t}^k \times \mathbf{m}_1^k = \mathbf{m}_2^k$. One of the three directors of an *adapted* frame must be the tangent to the discrete rod.

The material frame of a discrete rod, denoted as $(\mathbf{m}_1^k, \mathbf{m}_2^k, \mathbf{t}^k)$, is an orthonormal adapted frame. Twist of a rod can be measured from the material frame; this will be discussed later in this Chapter.

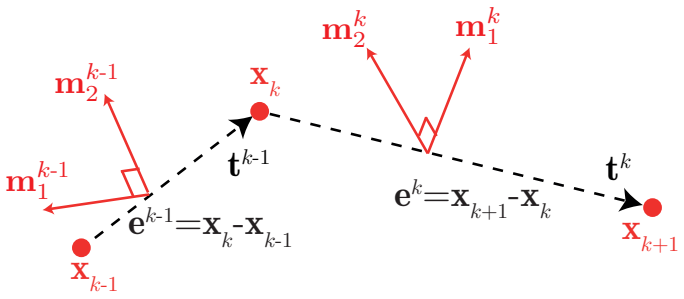

Figure 6.2: Adapted frame $(\mathbf{m}_1^{k-1}, \mathbf{m}_2^{k-1}, \mathbf{t}^{k-1})$ on edge $\mathbf{e}^{k-1}$ and $(\mathbf{m}_1^k, \mathbf{m}_2^k, \mathbf{t}^k)$ on edge $\mathbf{e}^k$.

Signed Angle: In Fig. 6.3, consider two adapted orthonormal frames $(\mathbf{u}^k, \mathbf{v}^k, \mathbf{t}^k)$ and $(\mathbf{m}_1^k, \mathbf{m}_2^k, \mathbf{t}^k)$ that share the tangent $\mathbf{t}^k$ as a common director. The notion of *signed angle* is necessary to measure the relative orientation of two such frames. In Fig. 6.3(a), the signed angle, ν^k, *from* $\mathbf{u}^k$ *to* $\mathbf{m}_1^k$ with *axis* $\mathbf{t}^k$ is positive. The signed angle from $\mathbf{v}^k$ to $\mathbf{m}_2^k$ with axis $\mathbf{t}^k$ is the same quantity, ν^k. In Fig. 6.3(b), this angle is negative.

A code snippet that computes signed angle from one vector to another vector with a given axis is shown in Fig. 6.4. A MATLAB code file for computation of signed angle `signedAngle.m` is provided with these notes.

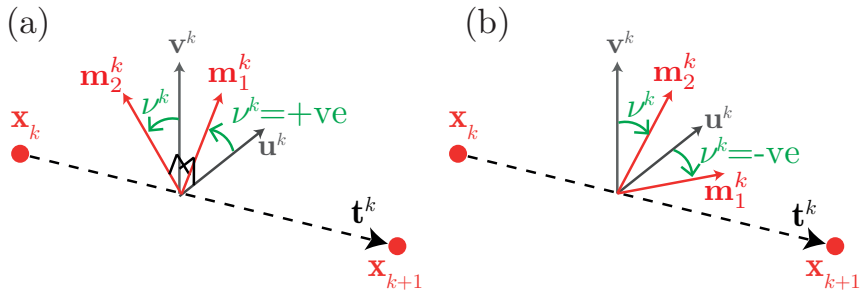

Figure 6.3: Signed angle, ν^k, from $\mathbf{u}^k$ to $\mathbf{m}_1^k$ with axis $\mathbf{t}^k$ is (a) positive and (b) negative. Same is true for the signed angle from $\mathbf{v}^k$ to $\mathbf{m}_2^k$ with axis $\mathbf{t}^k$.

```
function angle = signedAngle( u, v, n )
% "angle" is signed angle from vector "u" to vector "v" with axis "n"
w = cross(u,v);
angle = atan2( norm(w), dot(u,v) );
if (dot(n,w) < 0)
    angle = -angle;
end
end
```

Figure 6.4: Code snippet that implements signed angle.

6.2 Parallel Transport

What is the *most natural* or *twist free* way of moving an adapted frame from one edge ($\mathbf{e}^{k-1}$) to the next ($\mathbf{e}^k$)? The problem is simple if $\mathbf{e}^{k-1}$ and $\mathbf{e}^k$ are parallel. In Fig. 6.5, $(\mathbf{u}^{k-1}, \mathbf{v}^{k-1}, \mathbf{t}^{k-1})$ is an orthonormal adapted frame on edge $\mathbf{e}^{k-1}$. The next edge $\mathbf{e}^k$ in Fig. 6.5(a) is parallel to $\mathbf{e}^{k-1}$ and $\mathbf{t}^k = \mathbf{t}^{k-1}$. Therefore, the adapted frame $(\mathbf{u}^k, \mathbf{v}^k, \mathbf{t}^k)$ on edge $\mathbf{e}^k$ that does not have any twist with $(\mathbf{u}^{k-1}, \mathbf{v}^{k-1}, \mathbf{t}^{k-1})$ can be trivially computed. In fact, the two frames are identical, i.e. $\mathbf{u}^k = \mathbf{u}^{k-1}, \mathbf{v}^k = \mathbf{v}^{k-1}$, and $\mathbf{t}^k = \mathbf{t}^{k-1}$.

This problem becomes non-trivial in Fig. 6.5(b) where the two edges are not parallel and $\mathbf{t}^k \neq \mathbf{t}^{k-1}$. Let us first boil down the problem: given (1) the first tangent,

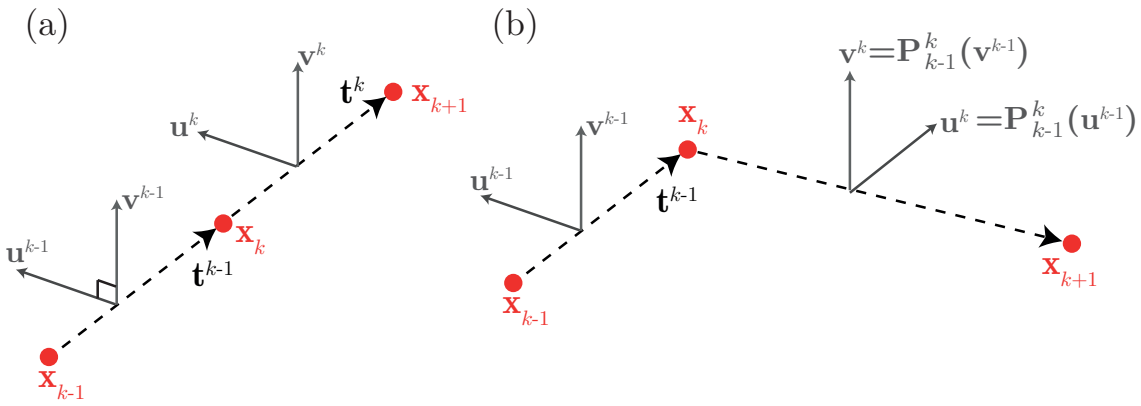

Figure 6.5: Parallel transport.

$\mathbf{t}^{k-1}$, (2) a vector, $\mathbf{u}$, that is orthogonal to the first tangent, and (3) the second tangent, $\mathbf{t}^k$, we need to compute the new vector, $\mathbf{d}$, that is orthogonal to the second tangent and does not have any twist with $\mathbf{u}$. The method of computing $\mathbf{d}$ is called *parallel transport*; let us denote it as $\mathbf{d} = \texttt{parallel\_transport}(\mathbf{u}, \mathbf{t}_1, \mathbf{t}_2)$. For convenience, we introduce the following notation for parallel transport from $(k-1)$-th edge to k-th edge:

$$P_{k-1}^{k}(\mathbf{u}) \equiv \texttt{parallel\_transport}(\mathbf{u}, \mathbf{t}^{k-1}, \mathbf{t}^{k}) \tag{6.1}$$

Parallel transporting a vector involves the following sequential steps:

$$\begin{aligned}
\mathbf{b} &= \mathbf{t}^{k-1} \times \mathbf{t}^{k}, \\
\hat{\mathbf{b}} &= \frac{\mathbf{b}}{|\mathbf{b}|}, \\
\mathbf{n}_1 &= \mathbf{t}^{k-1} \times \hat{\mathbf{b}}, \\
\mathbf{n}_2 &= \mathbf{t}^{k} \times \hat{\mathbf{b}}, \\
\mathbf{d} &= (\mathbf{u} \cdot \mathbf{t}^{k-1})\mathbf{t}^{k} + (\mathbf{u} \cdot \mathbf{n}_1)\mathbf{n}_2 + (\mathbf{u} \cdot \hat{\mathbf{b}})\hat{\mathbf{b}}.
\end{aligned}$$

A code snippet that implements the above computation is provided in Fig. 6.6. A MATLAB code file including this implementation `parallel_transport.m` is provided with these notes.

Integrated Discrete Twist of an Adapted Frame: The integrated discrete twist (or, more simply, *twist*) of an adapted frame [e.g. $(\mathbf{m}_1^k, \mathbf{m}_2^k, \mathbf{t}^k)$ with $k = 1, \ldots, N-1$] at node $\mathbf{x}_k$ (between two edges $\mathbf{e}^{k-1}$ and $\mathbf{e}^k$) is the signed angle from $P_{k-1}^{k}(\mathbf{m}_1^{k-1})$

```
function d = parallel_transport(u, t1, t2)
% This function parallel transports a vector u from tangent t1 to t2
% Input:
% t1 - vector denoting the first tangent
% t2 - vector denoting the second tangent
% u - vector that needs to be parallel transported
% Output:
% d - vector after parallel transport

b = cross(t1, t2);
if (norm(b) == 0 )
    d = u;
else
    b = b / norm(b);
    % The following four lines may seem unnecessary but can sometimes help
    % with numerical stability
    b = b - dot(b,t1) * t1;
    b = b / norm(b);
    b = b - dot(b,t2) * t2;
    b = b / norm(b);

    n1 = cross(t1, b);
    n2 = cross(t2, b);
    d = dot(u,t1) * t2 + dot(u, n1) * n2 + dot(u, b) * b;
end
end
```

Figure 6.6: Code snippet that implements parallel transport.

to $\mathbf{m}_1^k$ about the axis $\mathbf{t}^k$. The word *integrated*, which may often be omitted, is used to distinguish this dimensionless measure of twist from the conventional definition of twist (Eq. 6) with unit of 1/length. The integrated twist can be divided by the *Voronoi length* of the node to obtain the conventional twist, where the Voronoi length $\bar{l}_k = \left(|\bar{\mathbf{e}^{k-1}}| + |\bar{\mathbf{e}^k}|\right)/2$ and $\bar{(\ )}$ represents evaluation in undeformed state.

Assignment 1: Consider a rod with $N = 4$ nodes:

$$\begin{aligned} \mathbf{x}_1 &= [0.00, \quad 0.00, \quad 0.00]\,, \\ \mathbf{x}_2 &= [0.50, \quad 0.00, \quad 0.00]\,, \\ \mathbf{x}_3 &= [0.75, \quad 0.25, \quad 0.00]\,, \qquad \text{and} \\ \mathbf{x}_4 &= [0.75, \quad 0.50, \quad 0.25]\,. \end{aligned}$$

The rod has an adapted material frame and the first material directors $\mathbf{m}_1^k$ (with $k = 1, \ldots, N-1$) are

$$\begin{aligned} \mathbf{m}_1^1 &= [0.00, \quad 0.00, \quad 1.00]\,, \\ \mathbf{m}_1^2 &= [0.00, \quad 0.00, \quad 1.00]\,, \qquad \text{and} \\ \mathbf{m}_1^3 &= \left[0.00, \quad -\frac{1}{\sqrt{2}}, \quad \frac{1}{\sqrt{2}}\right]. \end{aligned}$$

Compute the integrated twist in the rod. The correct answer is provided at the end of this Chapter.

6.3 Space-Parallel Reference Frame

We use a Cartesian coordinate system that serves as the *reference* while computing the nodal coordinates. While computing the twist of the material frame, we also need a reference. One option is to use the *space-parallel* reference frame. Given the nodal coordinates $\mathbf{x}_k$ with $k = 1, \ldots, N$, the steps to establish a space-parallel reference frame is outlined below.

- Start with an arbitrary (but adapted) orthonormal frame at the first edge, $\mathbf{e}^1$. For that purpose, choose any unit vector $\mathbf{u}^1$ that is orthogonal to $\mathbf{t}^1$. The second director is $\mathbf{v}^1 = \mathbf{t}^1 \times \mathbf{u}^1$ and the frame is $(\mathbf{u}^1, \mathbf{v}^1, \mathbf{t}^1)$.
- Sequentially compute the reference frames for the subsequent edges using $\mathbf{u}^k = P_{k-1}^k(\mathbf{u}^{k-1}) \equiv \texttt{parallel\_transport}(\mathbf{u}^{k-1}, \mathbf{t}^{k-1}, \mathbf{t}^k)$ and $\mathbf{v}^k = \mathbf{t}^k \times \mathbf{u}^k$.

When two frames share a common director, a scalar signed angle is needed to represent one frame with respect to the other frame. In our case, all the frames considered will have the tangent to the discrete rod as the third director. In Fig. 6.7 (a), the space-parallel reference frames $[(\mathbf{u}^{k-1}, \mathbf{v}^{k-1}, \mathbf{t}^{k-1})$ and $(\mathbf{u}^k, \mathbf{v}^k, \mathbf{t}^k)]$ and the material frames $[(\mathbf{m}_1^{k-1}, \mathbf{m}_2^{k-1}, \mathbf{t}^{k-1})$ and $(\mathbf{m}_1^k, \mathbf{m}_2^k, \mathbf{t}^k)]$ on two consecutive edges $[\mathbf{e}^{k-1}$

and $\mathbf{e}^k$] are shown. The signed angle from $\mathbf{u}^k$ to $\mathbf{m}_1^k$ is ν^k and the twist (specifically, integrated discrete twist) at node $\mathbf{x}_k$ is

$$\tau_k = \nu^k - \nu^{k-1}. \tag{6.2}$$

The physical interpretation of twist, τ_k, is: if we want to transport the material frame from $(k-1)$-th edge to k-th edge, we need to rotate the frame by an angle τ_k about the tangent.

Bergou et al. used space-parallel reference to evaluate twist in the first publication on DER in 2008 [2]. However, it turned out that the computational efficiency is superior if a *time-parallel* reference frame is used. This leads to a banded Jacobian in the time-stepping scheme and results in $O(N)$ time complexity. This reference frame formulation is discussed in the next section.

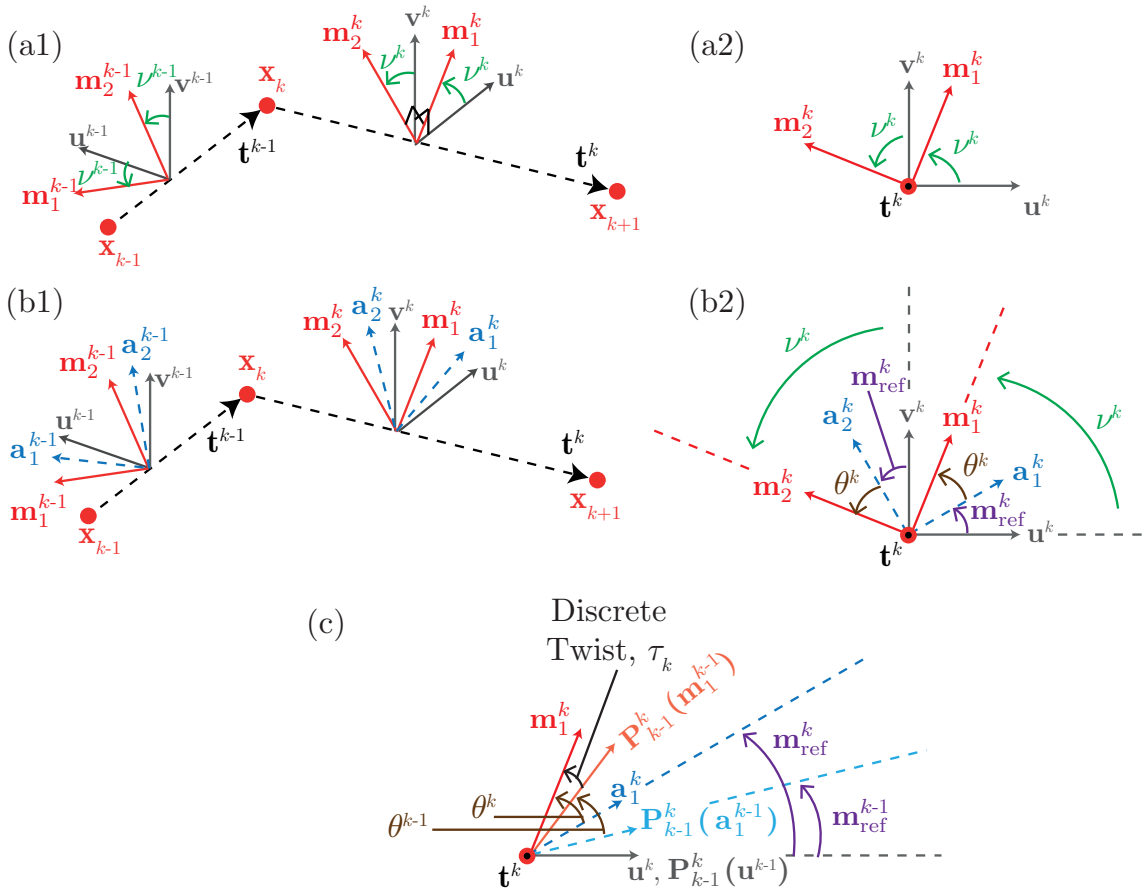

Figure 6.7: (a) Space-parallel reference frame $[(\mathbf{u}^{k-1}, \mathbf{v}^{k-1}, \mathbf{t}^{k-1})$ and $(\mathbf{u}^k, \mathbf{v}^k, \mathbf{t}^k)]$ and material frame $[(\mathbf{m}_1^{k-1}, \mathbf{m}_2^{k-1}, \mathbf{t}^{k-1})$ and $(\mathbf{m}_1^k, \mathbf{m}_2^k, \mathbf{t}^k)]$. (b) Time-parallel reference frame $[(\mathbf{a}_1^{k-1}, \mathbf{a}_2^{k-1}, \mathbf{t}^{k-1})$ and $(\mathbf{a}_1^k, \mathbf{a}_2^k, \mathbf{t}^k)]$ in addition to the space-parallel reference frame and material frame. (c) Discrete twist and relevant rotation angles.

Assignment 2: Consider the same rod described in **Assignment 1**.

- Construct a space-parallel reference frame for the rod. Start with $\mathbf{u}_1^1 = [0,\ 0,\ 1]$ at the first edge and use parallel transport to compute the three directors of the frame at the subsequent edges.
- Evaluate the angle ν^k $(k = 1, \ldots, N-1)$ at each edge.
- Compute twist using Eq. 6.2.

The correct answer is provided at the end of this Chapter.

6.4 Time-Parallel Reference Frame

In the dynamic simulation of an elastic rod, the twist varies with time, i.e. the material frame also changes with time. This far, we did not invoke time-dependence of the frames. Unlike space-parallel reference frame that is parallel transported forward in space, time-parallel reference frame is transported forward in time and we denote this frame on the edge $\mathbf{e}^k$ at time $t = t_j$ as $\big(\mathbf{a}_1^k(t_j), \mathbf{a}_2^k(t_j), \mathbf{t}^k(t_j)\big)$. This frame can be computed using following steps:

- At time $t = 0$, assign a space-parallel reference frame $[(\mathbf{u}^k, \mathbf{v}^k, \mathbf{t}^k)$ with $k = 1, \ldots, N-1]$ to each edge. Initialize the time-parallel reference frame as the space-parallel frame, i.e. $\big(\mathbf{a}_1^k(0) = \mathbf{u}^k, \mathbf{a}_2^k(0) = \mathbf{v}^k, \mathbf{t}^k(0)\big)$.
- Sequentially compute the time-parallel reference frames for the subsequent time steps using $\mathbf{a}_1^k(t_{j+1}) = \texttt{parallel\_transport}(\mathbf{a}_1^k(t_j), \mathbf{t}^k(t_j), \mathbf{t}^k(t_{j+1}))$, where $t_{j+1} = t_j + \Delta t$, and $\mathbf{a}_2^k(t_{j+1}) = \mathbf{t}^k(t_{j+1}) \times \mathbf{a}_1^k(t_{j+1})$.

6.5 Calculation of Twist using Time-Parallel Reference Frame

Referring to Fig. 6.7(b), the signed angle from the first (or second) space-parallel reference director, $\mathbf{u}^k$ (or $\mathbf{v}^k$), to the first (or second) time-parallel reference director, $\mathbf{a}_1^k$ (or $\mathbf{a}_2^k$) is the *reference twist angle*, m_{ref}^k. The signed angle from the first (or second) time-parallel reference director, $\mathbf{a}_1^k$ (or $\mathbf{a}_2^k$), to the first (or second) material director, $\mathbf{m}_1^k$ (or $\mathbf{m}_2^k$) is the *twist angle*, θ^k. Note that the $(N-1)$ twist angles of a discrete rod with N nodes will be considered as the degrees of freedom in the DER algorithm (Ch. **??**). From Fig. 6.7(b2), we notice

$$\nu^k = \theta^k + m_{\text{ref}}^k, \tag{6.3}$$

The integrated twist, τ_k, between two consecutive edges, $\mathbf{e}^{k-1}$ and $\mathbf{e}^k$, at node, $\mathbf{x}_k$, is

$$\tau_k = \nu^k - \nu^{k-1} = \big(\theta^k + m_{\text{ref}}^k\big) - \big(\theta^{k-1} + m_{\text{ref}}^{k-1}\big) = \Delta\theta_k + \Delta m_{k,\text{ref}}, \tag{6.4}$$

where $\Delta\theta_k = \theta^k - \theta^{k-1}$ and the *integrated reference twist* is $\Delta m_{k,\text{ref}} = m_{\text{ref}}^k - m_{\text{ref}}^{k-1}$. This is shown schematically in Fig. 6.7(c). The method to compute the integrated reference twist $\Delta m_{k,\text{ref}}$ given the DOFs ($\mathbf{x}_k$ and θ^k) is outlined below.

- Evaluate the time-parallel reference frame $\left(\mathbf{a}_1^k(t_{j+1}), \mathbf{a}_2^k(t_{j+1}), \mathbf{t}^k(t_{j+1})\right)$ with $k = 1, \ldots, N-1$ at the current time step $t = t_{j+1}$, given the frame from the previous step $\left(\mathbf{a}_1^k(t_j), \mathbf{a}_2^k(t_j), \mathbf{t}^k(t_j)\right)$.
- Parallel transport the first time-parallel reference director from $(k-1)$-th edge to k-th edge with $k = 2, \ldots, N$, i.e. compute $P_{k-1}^k\left(\mathbf{a}_1^{k-1}(t_{j+1})\right)$.
- The reference twist, $\Delta m_{k,\text{ref}}$, at node $\mathbf{x}_k$ is the signed angle from $P_{k-1}^k\left(\mathbf{a}_1^{k-1}(t_{j+1})\right)$ to $\mathbf{a}_1^k(t_{j+1})$ about the axis $\mathbf{t}^k(t_{j+1})$, i.e.
$\Delta m_{k,\text{ref}} = \texttt{signedAngle}\left(P_{k-1}^k\left(\mathbf{a}_1^{k-1}(t_{j+1})\right), \mathbf{a}_1^k(t_{j+1}), \mathbf{t}^k(t_{j+1})\right)$

A code snippet illustrating the computation of reference twist is shown in Fig. 6.8. A MATLAB code file `computeReferenceTwist.m` containing this calculation is also provided with these notes.

```
1   function refTwist = computeReferenceTwist(u1, u2, t1, t2)
2   % refTwist is the reference twist to move u1 with tangent t1 to u2 with
3   % tangent t2.
4 - ut = parallel_transport(u1, t1, t2);
5 - refTwist = signedAngle(ut, u2, t2);
6 - end
```

Figure 6.8: Code snippet that computes the integrated reference twist.

Assignment 3: The DOF vector of a rod with N nodes and $(N-1)$ edges has a size of $(4N-1)$ and is defined as

$$\mathbf{q} = \left[\mathbf{x}_1, \theta^1, \mathbf{x}_2, \theta^2, \ldots, \mathbf{x}_{N-1}, \theta^{N-1}, \mathbf{x}_N\right]^T.$$

The configuration of a rod with $N = 5$ nodes at three different times are provided below. The unit of the nodal coordinates is meter and that of the twist angles is radian.
At $t = 0$ s,

$$\begin{aligned}\mathbf{q} = [\ & 2.000e-02 \quad 0.000e+00 \quad 0.000e+00 \quad 0.000e+00 \quad 6.306e-03 \\ & 1.898e-02 \quad 0.000e+00 \quad 0.000e+00 \quad -1.602e-02 \quad 1.197e-02 \\ & 0.000e+00 \quad 0.000e+00 \quad -1.641e-02 \quad -1.143e-02 \quad 0.000e+00 \\ & 0.000e+00 \quad 5.673e-03 \quad -1.918e-02 \quad 0.000e+00]^T\end{aligned}$$

At $t = 0.05$ s,

$$\begin{aligned}\mathbf{q} = [\ & 2.000e-02 \quad 0.000e+00 \quad 0.000e+00 \quad 0.000e+00 \quad 6.306e-03 \\ & 1.898e-02 \quad 0.000e+00 \quad -2.006e-01 \quad -1.463e-02 \quad 1.281e-02 \\ & -8.462e-03 \quad 2.191e-01 \quad -1.441e-02 \quad -8.443e-03 \quad -1.827e-02 \\ & 4.726e-01 \quad 7.321e-03 \quad -1.712e-02 \quad -1.796e-02]^T\end{aligned}$$

At $t = 0.10$ s,

$$\begin{aligned}\mathbf{q} = [\quad & 2.000e-02 \quad 0.000e+00 \quad 0.000e+00 \quad 0.000e+00 \quad 6.306e-03 \\ & 1.898e-02 \quad 0.000e+00 \quad -3.908e-01 \quad -1.119e-02 \quad 1.474e-02 \\ & -1.497e-02 \quad 3.572e-01 \quad -9.813e-03 \quad 3.396e-04 \quad -3.337e-02 \\ & 9.616e-01 \quad 1.117e-02 \quad -9.421e-03 \quad -3.688e-02]^T\end{aligned}$$

At $t = 0$ s, the first director of the space-parallel reference frame on the first edge is $\mathbf{u}^1 = [-8.110e-01,\ -5.851e-01,\ -0.000e+00]$.

- Construct the space-parallel reference frame for $t = 0$ s. This initial frame is identical to the time-parallel reference frame at $t = 0$ s. Compute the material frame.

  Given the twist angle θ^k and the time-parallel reference directors $(\mathbf{m}_1^k, \mathbf{m}_2^k, \mathbf{t}^k)$, the material frame directors can be computed from the following relations:

$$\begin{aligned}\mathbf{m}_1^k &= \cos\theta^k \mathbf{a}_1^k + \sin\theta^k \mathbf{a}_2^k, \\ \mathbf{m}_2^k &= -\sin\theta^k \mathbf{a}_1^k + \cos\theta^k \mathbf{a}_2^k.\end{aligned}$$

- Construct the time-parallel reference frames at $t = 0.05$ and $t = 0.10$ s.
- Construct the material frames at $t = 0.05$ and $t = 0.10$ s.
- Compute the reference twist, $\Delta m_{k,\mathrm{ref}} (k = 2, \ldots, N-1)$, at $t = 0, 0.05$, and 0.10 s.
- Compute the integrated discrete twist, τ_k $(k = 2, \ldots, N-1)$, of the rod.

Answer key:

- **Assignment 1:** The integrated twist is $\tau_2 = 0$ at $\mathbf{x}_2$ and $\tau_3 = 0.34$ at $\mathbf{x}_3$.
- **Assignment 2:** The first directors of the space-parallel reference frame is

$$\begin{aligned} \mathbf{u}^1 &= [0.00, \quad 0.00, \quad 1.00], \\ \mathbf{u}^2 &= [0.00, \quad 0.00, \quad 1.00], \qquad \text{and} \\ \mathbf{u}^3 &= \left[-\frac{1}{3}, \quad -\frac{2}{3}, \quad \frac{2}{3}\right]. \end{aligned}$$

  The signed angles from this reference frame to the material frame are $\nu^1 = 0, \nu^2 = 0$, and $\nu^3 = 0.34$. The integrated discrete twists are $\tau_2 = 0$ and $\tau_3 = 0.34$. This matches the solution from **Assignment 1**.

- **Assignment 3: Time-parallel reference frame:** At $t = 0$ s,

$$\begin{aligned} \mathbf{a}_1^1 &= [-8.110e-01 \quad -5.851e-01 \quad -0.000e+00] \\ \mathbf{a}_1^2 &= [2.995e-01 \quad -9.541e-01 \quad 0.000e+00] \\ \mathbf{a}_1^3 &= [9.999e-01 \quad -1.659e-02 \quad 0.000e+00] \\ \mathbf{a}_1^4 &= [3.310e-01 \quad 9.436e-01 \quad 0.000e+00] \end{aligned}$$

  At $t = 0.05$ s,

$$\begin{aligned} \mathbf{a}_1^1 &= [-8.110e-01 \quad -5.851e-01 \quad -0.000e+00] \\ \mathbf{a}_1^2 &= [2.839e-01 \quad -9.589e-01 \quad -3.064e-03] \\ \mathbf{a}_1^3 &= [1.000e+00 \quad 7.584e-03 \quad 5.308e-03] \\ \mathbf{a}_1^4 &= [3.706e-01 \quad 9.288e-01 \quad 2.810e-04] \end{aligned}$$

  At $t = 0.10$ s,

$$\begin{aligned} \mathbf{a}_1^1 &= [-8.110e-01 \quad -5.851e-01 \quad -0.000e+00] \\ \mathbf{a}_1^2 &= [2.529e-01 \quad -9.672e-01 \quad -2.191e-02] \\ \mathbf{a}_1^3 &= [9.982e-01 \quad 4.802e-02 \quad 3.730e-02] \\ \mathbf{a}_1^4 &= [4.213e-01 \quad 9.069e-01 \quad -3.504e-03] \end{aligned}$$

  Reference twist: At $t = 0$ s,

$$\Delta m_{2,\text{ref}} = 0.000e+00 \qquad \Delta m_{3,\text{ref}} = 0.000e+00 \qquad \Delta m_{4,\text{ref}} = 0.000e+00$$

  At $t = 0.05$ s,

$$\Delta m_{2,\text{ref}} = -2.667e-01 \qquad \Delta m_{3,\text{ref}} = -5.648e-01 \qquad \Delta m_{4,\text{ref}} = -2.895e-01$$

At $t = 0.10$ s,

$$\Delta m_{2,\text{ref}} = -5.050e-01 \qquad \Delta m_{3,\text{ref}} = -1.028e+00 \qquad \Delta m_{4,\text{ref}} = -6.649e-01$$

Integrated discrete twist: The integrated discrete twist can be readily computed from Eq. 6.4, given $\Delta m_{k,\text{ref}}$ and θ^k.

At $t = 0$ s,

$$\tau_2 = 0.000e+00 \qquad \tau_3 = 0.000e+00 \qquad \tau_4 = 0.000e+00$$

At $t = 0.05$ s,

$$\tau_2 = -4.674e-01 \qquad \tau_3 = -1.451e-01 \qquad \tau_4 = -3.597e-02$$

At $t = 0.10$ s,

$$\tau_2 = -8.957e-01 \qquad \tau_3 = -2.803e-01 \qquad \tau_4 = -6.051e-02$$

Chapter 7

Discrete Elastic Rods Algorithm

This Chapter presents an applied introduction to the Discrete Elastic Rods (DER) algorithm for 3D simulation of elastic rods. In previous Chapters (e.g. Ch. 5), we considered 2D simulation of beams that can bend and stretch. In this Chapter, we will consider bending, stretching, and twisting for a complete physically-based simulation of rods. After reading this Chapter, the reader should be able to implement this algorithm in a computer program using the utility functions provided in the Appendix.

7.1 Equations of Motion

The DOF vector of a rod with N nodes and $(N-1)$ edges has a size of $(4N-1)$ and is defined as

$$\mathbf{q} = \left[\mathbf{x}_1, \theta^1, \mathbf{x}_2, \theta^2, \ldots, \mathbf{x}_{N-1}, \theta^{N-1}, \mathbf{x}_N\right]^T$$

where $\mathbf{x}_k$ $(k = 1, \ldots, N)$ are the nodal coordinates and θ^k is the twist angle at edge $\mathbf{e}^k = \mathbf{x}_{k+1} - \mathbf{x}_k$ (see Ch. 6). Similar to the 2D equations of motion in Eq. 4.4 2D, the equations in the 3D scenario to march from $t = t_j$ to $t = t_{j+1} = t_j + \Delta t$ is

$$f_i \equiv \frac{m_i}{\Delta t}\left[\frac{q_i(t_{j+1}) - q_i(t_j)}{\Delta t} - \dot{q}_i(t_j)\right] + \frac{\partial E_{\text{elastic}}}{\partial q_i} - f_i^{\text{ext}} = 0, \tag{7.1}$$

where $i = 1, \ldots, 4N-1$, the *old* DOF $q_i(t_j)$ and velocity $\dot{q}_i(t_j)$ are known, E_{elastic} is the elastic energy evaluated at $q_i(t_{j+1})$, f_i^{ext} is the external force (or moment for twist angles), e.g. gravity, and m_i is the lumped mass at each DOF. Note that, $m_i = \frac{1}{2}\Delta m\, r_0^2$ (moment of inertia) for twist angles where Δm is the mass of the edge and r_0 is the cross-sectional area (assuming a solid circular cross-section). The Jacobian for Eq. 7.1 is

$$\mathbb{J}_{ij} = \frac{\partial f_i}{\partial q_j} = \mathbb{J}_{ij}^{\text{inertia}} + \mathbb{J}_{ij}^{\text{elastic}} + \mathbb{J}_{ij}^{\text{ext}}, \tag{7.2}$$

where

$$\mathbb{J}_{ij}^{\text{inertia}} = \frac{m_i}{\Delta t^2}\delta_{ij}, \tag{7.3}$$

$$\mathbb{J}_{ij}^{\text{elastic}} = \frac{\partial^2 E_{\text{elastic}}}{\partial q_i \partial q_j}, \tag{7.4}$$

$$\mathbb{J}_{ij}^{\text{ext}} = -\frac{\partial f_i^{\text{ext}}}{\partial q_j}. \tag{7.5}$$

We can solve the $(4N-1)$ equations of motion in Eq. 7.1 to obtain the *new* DOF $\mathbf{q}(t_{j+1})$. The new velocity is simply

$$\dot{\mathbf{q}}(t_{j+1}) = \frac{\mathbf{q}(t_{j+1}) - \mathbf{q}(t_j)}{\Delta t}. \tag{7.6}$$

The main challenge in the solution procedure is evaluating the gradient of the elastic energy ($\frac{\partial E_{\text{elastic}}}{\partial q_i}$) as well as its Hessian ($\frac{\partial^2 E_{\text{elastic}}}{\partial q_i \partial q_j}$). This will be discussed in § 7.3. A second challenge is a rigorous computation of the twist angles (details in Ch. 6) that necessitates a few additional steps in the time marching scheme, compared with the 2D case. The main steps of the algorithm are outlined below in § 7.2

7.2 Pseudocode of DER

Algorithm 1 Discrete Elastic Rods

Require: $\mathbf{q}(t_j), \dot{\mathbf{q}}(t_j)$ ▷ DOFs and velocities at $t = t_j$
Require: $\left(\mathbf{a}_1^k(t_j), \mathbf{a}_2^k(t_j), \mathbf{t}^k(t_j)\right)$ ▷ Reference frame at $t = t_j$
Require: `free_index` ▷ Index of the free DOFs
Ensure: $\mathbf{q}(t_{j+1}), \dot{\mathbf{q}}(t_{j+1})$ ▷ DOFs and velocities at $t = t_{j+1}$
Ensure: $\left(\mathbf{a}_1^k(t_{j+1}), \mathbf{a}_2^k(t_{j+1}), \mathbf{t}^k(t_{j+1})\right)$ ▷ Reference frame at $t = t_{j+1}$

function DISCRETE_ELASTIC_RODS($\mathbf{q}, \dot{\mathbf{q}}(t_j), \left(\mathbf{a}_1^k(t_j), \mathbf{a}_2^k(t_j), \mathbf{t}^k(t_j)\right)$)
  Guess: $\mathbf{q}^{(1)}(t_{j+1}) \leftarrow \mathbf{q}(t_j)$
  $n \leftarrow 1$
  while `error` $>$ `tolerance` **do**
    Compute reference frame $\left(\mathbf{a}_1^k(t_{j+1}), \mathbf{a}_2^k(t_{j+1}), \mathbf{t}^k(t_{j+1})\right)^{(n)}$ using $\mathbf{q}^{(n)}(t_{j+1})$
    Compute reference twist $\Delta m_{k,\text{ref}}^{(n)}$ $(k = 2, \ldots, N-1)$
    Compute material frame $\left(\mathbf{m}_1^k(t_{j+1}), \mathbf{m}_2^k(t_{j+1}), \mathbf{t}^k(t_{j+1})\right)^{(n)}$
    Compute $\mathbf{f}$ and $\mathbb{J}$ ▷ Eqs. 7.1 and 7.2; see Algorithm 2
    $\mathbf{f}_{\text{free}} \leftarrow \mathbf{f}$ `(free_index)`
    $\mathbb{J}_{\text{free}} \leftarrow \mathbb{J}$ `(free_index, free_index)`
    $\Delta \mathbf{q}_{\text{free}} \leftarrow \mathbb{J}_{\text{free}} \backslash \mathbf{f}_{\text{free}}$
    $\mathbf{q}^{(n+1)}$ `(free_index)` $\leftarrow \mathbf{q}^{(n)}$ `(free_index)` $-\Delta \mathbf{q}_{\text{free}}$ ▷ Update free DOFs
    `error` $\leftarrow$ `sum ( abs (` $\mathbf{f}_{\text{free}}$ `) )`
    $n \leftarrow n + 1$
  end while

  $\mathbf{q}(t_{j+1}) \leftarrow \mathbf{q}^{(n)}(t_{j+1})$
  $\dot{\mathbf{q}}(t_{j+1}) \leftarrow \frac{\mathbf{q}(t_{j+1}) - \mathbf{q}(t_j)}{\Delta t}$
  $\left(\mathbf{a}_1^k(t_{j+1}), \mathbf{a}_2^k(t_{j+1}), \mathbf{t}^k(t_{j+1})\right) \leftarrow \left(\mathbf{a}_1^k(t_{j+1}), \mathbf{a}_2^k(t_{j+1}), \mathbf{t}^k(t_{j+1})\right)^{(n)}$
  return $\mathbf{q}(t_{j+1}), \dot{\mathbf{q}}(t_{j+1}), \left(\mathbf{a}_1^k(t_{j+1}), \mathbf{a}_2^k(t_{j+1}), \mathbf{t}^k(t_{j+1})\right)$
end function

7.3 Gradient and Hessian of Elastic Energies

The total elastic energy of an elastic rod is

$$E_{\text{elastic}} = \underbrace{\sum_{k=1}^{N-1} E_k^s}_{\text{stretching energy}} + \underbrace{\sum_{k=2}^{N-1} E_k^b}_{\text{bending energy}} + \underbrace{\sum_{k=2}^{N-1} E_k^t}_{\text{twisting energy}}, \tag{7.7}$$

where E_k^s is the stretching energy associated with the edge $\mathbf{e}^k = \mathbf{x}_{k+1} - \mathbf{x}_k$, E_k^b is the bending energy associated with the node $\mathbf{x}_k$ (due to the turning angle between $\mathbf{e}^{k-1}$ and $\mathbf{e}^k$), E_k^t is the twisting energy associated with the node $\mathbf{x}_k$ (due to the rotation of the material frame as it moves from $\mathbf{e}^{k-1}$ to $\mathbf{e}^k$). The analytical expressions of these energies in terms of the DOFs are discussed next.

Stretching Energy:

$$E_k^s = \frac{1}{2} EA \left(\frac{|\mathbf{x}_{k+1} - \mathbf{x}_k|}{|\bar{\mathbf{e}^k}|} - 1 \right)^2 |\bar{\mathbf{e}^k}|, \tag{7.8}$$

where E is the Young's modulus, A is the cross-sectional area (i.e. EA is the stretching stiffness), and $|\bar{\mathbf{e}^k}|$ is the length of the edge $\mathbf{e}^k$ in undeformed state. See Appendix for MATLAB function that evaluates the gradient and Hessian of the stretching energy. The derivations and analytical expressions of these quantities are provided in Ref. [5].

During programming implementation, it is important to note that E_k^s is only affected by two nodes – $\mathbf{x}_k$ and $\mathbf{x}_{k+1}$. The location of these nodes in the DOF vector is

$$\texttt{ind} = [4k-3, 4k-2, 4k-1, 4k+1, 4k+2, 4k+3], \tag{7.9}$$

and the non-zero part of the gradient $\frac{\partial E_k^s}{\partial \mathbf{q}}$ is a vector of size 6. Similarly, the non-zero part of the Hessian $\frac{\partial^2 E_k^s}{\partial \mathbf{q} \partial \mathbf{q}}$ is a matrix of size 6×6. The function `gradEs_hessEs()` provided in the Appendix will output this "small" non-zero vector `dF` of size 6 and matrix `dJ` of size 6×6. This "small" vector can be placed into the full $4N-1$ sized gradient vector `F` and $(4N-1) \times (4N-1)$ sized Hessian matrix with two lines of code in MATLAB: `F(ind) = F(ind) + dF` and `J(ind, ind) = J(ind, ind) + dJ`.

Bending Energy:

$$E_k^b = \frac{1}{2} EI \left(|\boldsymbol{\kappa}_k - \boldsymbol{\kappa}_k^0| \right)^2 \frac{1}{\bar{l}_k}, \tag{7.10}$$

where EI is the bending stiffness (Young's modulus times area moment of inertia), $\boldsymbol{\kappa}_k$ is the curvature (the discrete integrated version) vector at node $\mathbf{x}_k$, $\boldsymbol{\kappa}_k^0$ is the *natural* (undeformed) curvature at the same node, $\bar{l}_k = \left(|\bar{\mathbf{e}^{k-1}}| + |\bar{\mathbf{e}^k}| \right)/2$ is the Voronoi length and $\bar{(\,)}$ represents evaluation in undeformed state. For a rod with circular cross-section, $I = \pi r_0^4/4$. If $\mathbf{t}^k = \frac{\mathbf{e}^k}{|\mathbf{e}^k|}$ is the tangent on k-th edge, the curvature vector at

node $\mathbf{x}_k$ can be evaluated from the following relations:

$$
\begin{aligned}
(\kappa b)_k &= \frac{2\mathbf{t}^{k-1} \times \mathbf{t}^k}{1 + \mathbf{t}^{k-1} \cdot \mathbf{t}^k}, \\
\kappa_1 &= \frac{1}{2}(\kappa b)_k \cdot \left(\mathbf{m}_2^{k-1} + \mathbf{m}_2^k\right), \\
\kappa_2 &= -\frac{1}{2}(\kappa b)_k \cdot \left(\mathbf{m}_1^{k-1} + \mathbf{m}_1^k\right), \\
\boldsymbol{\kappa}_k &= \left[\kappa_1,\ \kappa_2\right],
\end{aligned}
$$

where $(\kappa b)_k$ is the curvature binormal. This measure of curvature is an *integrated* quantity with no unit; conventionally, curvature has unit of 1/length. See Appendix for MATLAB function that evaluates the integrated curvature. A MATLAB function that computes the gradient and Hessian of the bending energy is also provided in the Appendix.

Similar to the case of stretching energy, it should be noted that E_k^b only depends on three nodes and two twist angles – $\mathbf{x}_{k-1}, \theta^{k-1}, \mathbf{x}_k, \theta^k$, and $\mathbf{x}_{k+1}$ – that correspond to the following 11 positions in the DOF vector:

$$
\begin{aligned}
\texttt{ind} = \big[&4k-7, 4k-6, 4k-5, 4k-4, 4k-3, && (7.11)\\
&4k-2, 4k-1, 4k, 4k+1, 4k+2, 4k+3\big]. && (7.12)
\end{aligned}
$$

The non-zero part of the gradient $\frac{\partial E_k^b}{\partial \mathbf{q}}$ is a vector of size 6; let us denote it as `dF`. The non-zero part of the Hessian $\frac{\partial^2 E_k^s}{\partial \mathbf{q} \partial \mathbf{q}}$ is a matrix of size 6×6; let us denote is `dJ`. The function `gradEb_hessEb()` in the Appendix outputs this "small" `dF` vector and `dJ` matrix. They can be placed into the full gradient vector and Hessian matrix with the following two lines in MATLAB: `F(ind) = F(ind) + dF` and `J(ind, ind) = J(ind, ind) + dJ`.

Twisting Energy:

$$
E_k^t = \frac{1}{2} GJ \tau_k^2 \frac{1}{\bar{l}_k}, \tag{7.13}
$$

where GJ is the twisting stiffness (shear modulus times polar moment of inertia) and τ_k is the integrated twist at node $\mathbf{x}_k$; see § 6.5 for the methods to evaluate twist. For a rod with circular cross-section, $J = \pi r_0^4/2$. See Appendix for MATLAB function that evaluates the gradient and Hessian of the twisting energy. Twisting energy E_k^t at the k-th node depends on 11 DOFs, representing $\mathbf{x}_{k-1}, \theta^{k-1}, \mathbf{x}_k, \theta^k$, and $\mathbf{x}_{k+1}$. The outputs from `gradEt_hessEt()` function in the Appendix can be placed into the full gradient vector and Hessian matrix using a method identical to the bending energy.

Algorithm 2 shows the steps of computing the full gradient and Hessian of elastic energies of a rod.

Algorithm 2 Gradient and Hessian of Elastic Energy in a Rod

Require: $\mathbf{q} = \left[\mathbf{x}_0, \theta^0, \mathbf{x}_1, \theta^1, \ldots, \theta^{N-1}, \mathbf{x}_N\right]^T$ ▷ Degrees of Freedom
Require: EA, EI, GJ ▷ Elastic stiffness in Eqs. 7.8, 7.10, 7.13
Require: $\bar{l}_k$ $(1 \leq k \leq N)$ ▷ Undeformed Voronoi length
Require: $\bar{\mathbf{e}}^k (1 \leq k < N)$ ▷ Undeformed edge length
Require: $\bar{\boldsymbol{\kappa}}_k^0 (1 < k < N)$ ▷ Natural curvature
Require: Reference frame, $\left(\mathbf{a}_1^k(t_{j+1}), \mathbf{a}_2^k(t_{j+1}), \mathbf{t}^k(t_{j+1})\right)$, where $1 \leq k < N$
Require: Reference twist, $\Delta m_{k,\text{ref}}$ $(1 \leq k < N)$
Require: Material frame, $\left(\mathbf{m}_1^k(t_{j+1}), \mathbf{m}_2^k(t_{j+1}), \mathbf{t}^k(t_{j+1})\right)$, where $1 \leq k < N$
Ensure: `F` ▷ $4N-1$ sized elastic gradient vector, $\frac{\partial E_{\text{elastic}}}{\partial \mathbf{q}}$
Ensure: `J` ▷ $4N-1 \times 4N-1$ sized elastic Hessian matrix, $\mathbb{J}^{\text{elastic}}$

function GRAD_HESS_ELASTIC_ROD($\mathbf{q}$)
    `F` ← `zeros(4N-1,1)`
    `J` ← `zeros(4N-1,4N-1)`
    for $k \leftarrow 1$ to $N-1$ **do**
        `ind` ← locations of $\mathbf{x}_k$ and $\mathbf{x}_{k+1}$ ▷ Eq. 7.9
        `[dF, dJ]` ← `gradEs_hessEs`$(\mathbf{x}_k, \mathbf{x}_{k+1})$ ▷ Code in Appendix
        `F (ind)` = `F (ind)` + `dF`
        `J (ind)` = `F (ind, ind)` + `dJ`
    end for

    for $k \leftarrow 2$ to $N-1$ **do**
        `ind` ← locations of $\mathbf{x}_{k-1}, \theta^{k-1}, \mathbf{x}_k, \theta^k$, and $\mathbf{x}_{k+1}$ ▷ Eq. 7.12
        `[dF, dJ]` ← `gradEb_hessEb`$(\mathbf{x}_{k-1}, \theta^{k-1}, \mathbf{x}_k, \theta^k, \mathbf{x}_{k+1})$ ▷ Code in Appendix
        `F (ind)` = `F (ind)` + `dF`
        `J (ind)` = `J (ind, ind)` + `dJ`
        `[dF, dJ]` ← `gradEt_hessEt`$(\mathbf{x}_{k-1}, \theta^{k-1}, \mathbf{x}_k, \theta^k, \mathbf{x}_{k+1})$ ▷ Code in Appendix
        `F (ind)` = `F (ind)` + `dF`
        `J (ind)` = `J (ind, ind)` + `dJ`
    end for
    return `F` and `J`
end function

Assignment: An elastic rod with a total length $l = 20$ cm is naturally curved with radius $R_n = 2$ cm. The location of its N nodes at $t = 0$ are

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

where $\Delta\theta = \frac{l}{R_n}\frac{1}{N-1}$. The twist angles θ^k $(k = 1, \ldots, N-1)$ at $t = 0$ are 0. The first two nodes and the first twist angle remain fixed throughout the simulation (i.e. one end is clamped). The physical parameters are: density $\rho = 1000$ kg/m^3, cross-sectional radius $r_0 = 1$ mm, Young's modulus $E = 10$ MPa, shear modulus $G = \frac{E}{3}$ (corresponding to an incompressible material), and gravitational acceleration $g = [0, 0, -9.81]^T$. Choose an appropriate time step size Δt and number of nodes N.

- Write a computer program that simulates the deformation of this rod under gravity from $t = 0$ to $t = 5$ s.
- Plot the z-coordinate of the last node ($\mathbf{x}_N$) with time. The solution can be found at the end of this Chapter.

Answer key:

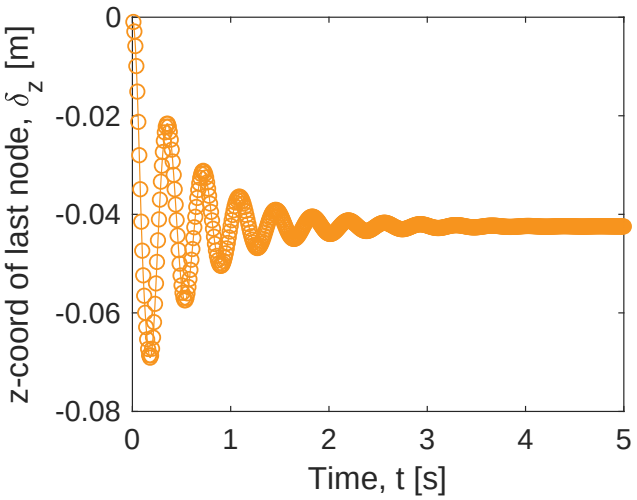

Figure 7.1: z-coordinate of the last node as a function of time. The simulation used $\Delta t = 0.01$ s and $N = 50$. The result may vary depending on Δt and N. However, the steady state value of $\delta_z \approx -0.04$ m should be recovered in a simulation with adequately large N and sufficiently small Δt.

Chapter 8

Discrete Elastic Plates and Shells

This Chapter presents an applied introduction to Discrete Elastic Plates (DEP) [10, 11, 12, 6] and Discrete Elastic Shells (DES) [7, 13] algorithms for 3D simulation of plates and shells, respectively. In this Chapter, we will the elastic energies – bending, stretching, and shearing – of a plate and shell. After reading this Chapter, the reader should be able to implement these algorithms using the utility functions provided in the Appendix. It is important to keep in mind that, while beam and rod simulations are usually physically accurate, plate and shell simulation's accuracy can vary based on the size of mesh, among other factors. Nonetheless, this class of simulations are widely in the computer graphics community where visual realism is more important than physical accuracy.

8.1 Kinematics, Meshing, and Degrees of Freedom

Referring to Fig. 8.1(a), the plate or shell is first discretized into triangular mesh. Ideally, each triangular element should be an equilateral triangle. A variety of software tools are available for meshing, e.g. `generateMesh` function of MATLAB. Fig. 8.2 shows a code snippet that uses MATLAB to mesh a rectangular plate.

Assume that the mesh contains N nodes and, therefore, the plate or shell structure is comprised of $3N$ degrees of freedom (DOFs). The DOF vector is

$$\mathbf{q} = [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N]^T, \tag{8.1}$$

where $\mathbf{x}_i$ $(1 \leq i \leq N)$ is the Cartesian coordinates of the i-th node and superscript ^{T} indicates transposition operation. This DOF vector varies as a function of time, i.e. $\mathbf{q} = \hat{\mathbf{q}}(t)$. The simulation software that will develop should be able to march forward in time and, at each time, compute this DOF vector.

A key component of the kinematics of a discrete plate or shell is the concept of "bend angle" [13] shown in Fig. 8.1(b). Let us first define a few quantities required for convenience. An "edge" is a vector that connects two consecutive nodes. Each triangular element is comprised of three edges. Any edge that is shared by two triangular elements is a "hinge" about which the nodes ($\mathbf{x}_2$ and $\mathbf{x}_3$ in Fig. 8.1(b)) can bend. Fig. 8.1(b) shows the edges by dashed lines and hinges by solid lines. All

the edges, except the ones on the boundary of the structure, are hinges. Each hinge is associated with four nodes: $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2$, and $\mathbf{x}_3$. The hinge vector goes from $\mathbf{x}_0$ and $\mathbf{x}_1$. Two triangular elements (indicated by *Triangle 1* and *Triangle 2* in Fig. 8.1(b)) are used to define the bend angle θ. The normal vector, $\mathbf{n}_1$, on Triangle 1 is the unit vector along $(\mathbf{x}_1 - \mathbf{x}_0) \times (\mathbf{x}_2 - \mathbf{x}_0)$. The other normal vector $\mathbf{n}_2$, is the unit vector along $(\mathbf{x}_1 - \mathbf{x}_0) \times (\mathbf{x}_3 - \mathbf{x}_0)$. The bend angle is defined as the *signed* angle from $\mathbf{n}_1$ to $\mathbf{n}_2$ along the hinge vector $(\mathbf{x}_1 - \mathbf{x}_0)$. *Signed* angle means that θ is positive if curling fingers on right hand from $\mathbf{n}_1$ to $\mathbf{n}_2$ is along the hinge vector. Otherwise, the bend angle θ is negative.

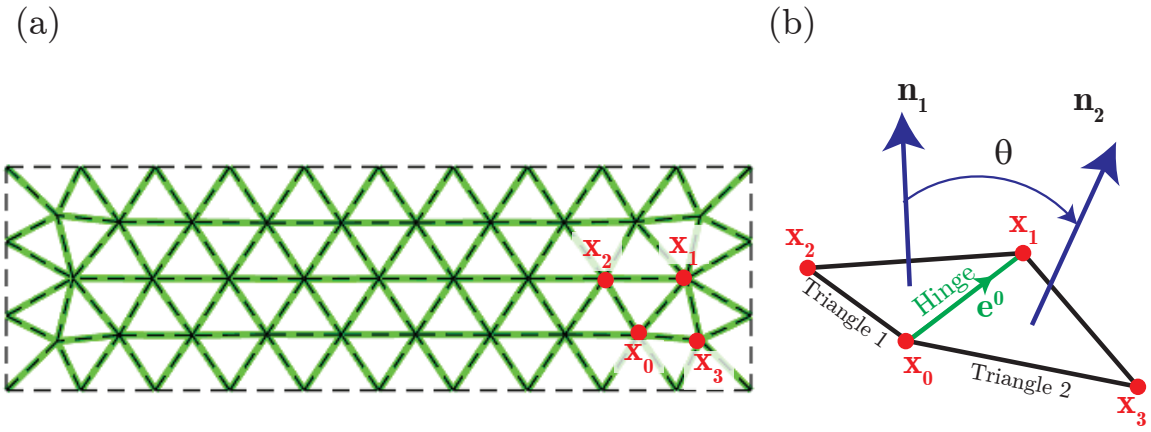

Figure 8.1: (a) Triangular mesh on a plate. (b) Bending angle, θ, associated with a "hinge" and, in turn, related to four nodes, $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2$, and $\mathbf{x}_3$.

8.2 Elastic Energies of a Plate

The total elastic energy in two dimensional plate is the sum of the elastic stretching and bending energies [10], i.e.

$$E_{\text{elastic}} = \sum_{k=1}^{N_{\text{edge}}} E_{\text{s}}^k + \sum_{k=1}^{N_{\text{hinge}}} E_{\text{b}}^k, \qquad \text{where} \tag{8.2a}$$

$$E_{\text{s}}^k = \frac{1}{2} k_s \left(\frac{\|\mathbf{e}^k\|}{l_k} - 1 \right)^2 \tag{8.2b}$$

$$E_{\text{b}}^k = \frac{1}{2} k_b \theta_k^2, \tag{8.2c}$$

```
% Filename: generateMesh.m
% Demo of triangular mesh generation on a rectangular region
% Khalid Jawed (khalidjm@seas.ucla.edu)

l = 0.1; % length of the rectangle
w = 0.03; % width of the rectangle
maxMeshSize = w/5; % maximum size of the mesh
minMeshSize = maxMeshSize/2; % minimum size of the mesh

gd = [3; 4; 0; l; l; 0; 0; 0; w; w];
g = decsg(gd);
model = createpde;
geometryFromEdges(model,g); % create geometry (rectangle in our case)

FEMesh = generateMesh(model,'Hmax', maxMeshSize, 'Hmin', ...
    minMeshSize, 'GeometricOrder', 'linear'); % generate the mesh

% Extract the elements and nodes
Elements = FEMesh.Elements;
Nodes = FEMesh.Nodes;

% Plot
figure(2);
hold on
[~, numElements] = size(Elements);
for c=1:numElements
    node1_number = Elements(1,c);
    node2_number = Elements(2,c);
    node3_number = Elements(3,c);
    node1_position = Nodes(:, node1_number);
    node2_position = Nodes(:, node2_number);
    node3_position = Nodes(:, node3_number);

    x_arr = [node1_position(1), node2_position(1), node3_position(1), ...
        node1_position(1)];
    y_arr = [node1_position(2), node2_position(2), node3_position(2), ...
        node1_position(2)];
    plot( x_arr, y_arr, 'ro-');
end
hold off
axis equal
box on
```

Figure 8.2: Code snippet that uses MATLAB to generate mesh on a rectangular plate.

where the elastic stretching and bending stiffness are

$$k_s = \frac{\sqrt{3}}{2} Y h l_k^2, \qquad \text{and} \tag{8.3a}$$

$$k_b = \frac{2}{\sqrt{3}} \frac{Y h^3}{12}, \tag{8.3b}$$

respectively, Y is the Young's modulus, h is the plate thickness, $\|\mathbf{e}^k\|$ is the length of the k-th edge (l_k is the undeformed edge length), N_{edge} is the total number of edges in the mesh, θ_k is the bend angle at the k-th hinge, and N_{hinge} is the total number of hinges. Often, modified versions of the bending energy in Eq. 8.2 are used, e.g. instead of θ_k^2, an alternative is to use $\|\mathbf{n}_1 - \mathbf{n}_2\|^2$ [11, 12, 10]. This discrete

representation of the energy functional (using $\|\mathbf{n}_1 - \mathbf{n}_2\|^2$) has been shown to converge to the continuum limit of Föppl-von Kármán equations used to describe the nonlinear mechanics of thin plates [11, 12, 10].

Similar to simulation of beams and rods, we have to compute the gradient and Hessian of the elastic energies. Computing the gradient $\left(\frac{\partial E_s^{\text{plate}}}{\partial \mathbf{q}}\right)$ and Hessian $\left(\frac{\partial^2 E_s^{\text{plate}}}{\partial \mathbf{q}\partial \mathbf{q}}\right)$ of the stretching energy, $E_{\text{s}}^{\text{plate}}$ are straightforward. Bending energy's gradient $\left(\frac{\partial E_b^{\text{plate}}}{\partial \mathbf{q}}\right)$ and Hessian $\left(\frac{\partial^2 E_b^{\text{plate}}}{\partial \mathbf{q}\partial \mathbf{q}}\right)$ require non-trivial algebraic manipulation; a good resource is Ref. [13]. MATLAB codes to compute the gradient and Jacobian are provided in the Appendix.

8.3 Elastic Energies of a Shell

The undeformed shape of a plate is 2D and all the bend angles are zero. In case of a shell, it is a 3D structures and the bend angles in their undeformed (stress-free) state are not necessarily zero. In fact, plate-type structures are a subset of shell structures. We only require minor modification to Eq. 8.2 to get the discrete elastic energy of a shell,

$$E_{\text{elastic}} = \sum_{k=1}^{N_{\text{edge}}} E_{\text{s}}^k + \sum_{k=1}^{N_{\text{hinge}}} E_{\text{b}}^k, \qquad \text{where} \tag{8.4a}$$

$$E_{\text{s}}^k = \frac{1}{2} k_s \left(\frac{\|\mathbf{e}^k\|}{l_k} - 1 \right)^2 \tag{8.4b}$$

$$E_{\text{b}}^k = \frac{1}{2} k_b \left(\theta_k - \bar{\theta}_k \right)^2, \tag{8.4c}$$

where we inserted $\bar{\theta}_k$ term – denoting the bend angle in the undeformed configuration – in the expression for E_b^{shell}. Typically, initial configuration ($\mathbf{q}(0)$) is the same as the undeformed configuration. In a simulation code, this quantity (as well as the undeformed edge length $\|\bar{\mathbf{e}}^k\|$) will be computed and stored prior to moving into the time marching scheme (i.e. solving the equations of motion).

8.4 Equations of Motion

We can follow a time-stepping procedure identical to Ch. 7. The equations of motion for a plate or shell to march from $t = t_j$ to $t = t_{j+1} = t_j + \Delta t$ is

$$f_i \equiv \frac{m_i}{\Delta t} \left[\frac{q_i(t_{j+1}) - q_i(t_j)}{\Delta t} - \dot{q}_i(t_j) \right] + \frac{\partial E_{\text{elastic}}}{\partial q_i} - f_i^{\text{ext}} = 0, \tag{8.5}$$

where $i = 1, \ldots, 3N$, the *old* DOF $q_i(t_j)$ and velocity $\dot{q}_i(t_j)$ are known, E_{elastic} is the elastic energy evaluated at $q_i(t_{j+1})$, f_i^{ext} is the external force (e.g. gravity), and m_i is

the lumped mass at each DOF. The Jacobian for Eq. 7.1 is

$$\mathbb{J}_{ij} = \frac{\partial f_i}{\partial q_j} = \mathbb{J}_{ij}^{\text{inertia}} + \mathbb{J}_{ij}^{\text{elastic}} + \mathbb{J}_{ij}^{\text{ext}}, \tag{8.6}$$

where

$$\mathbb{J}_{ij}^{\text{inertia}} = \frac{m_i}{\Delta t^2}\delta_{ij}, \tag{8.7}$$

$$\mathbb{J}_{ij}^{\text{elastic}} = \frac{\partial^2 E_{\text{elastic}}}{\partial q_i \partial q_j}, \tag{8.8}$$

$$\mathbb{J}_{ij}^{\text{ext}} = -\frac{\partial f_i^{\text{ext}}}{\partial q_j}. \tag{8.9}$$

We can solve the $3N$ equations of motion in Eq. 8.5 to obtain the *new* DOF $\mathbf{q}(t_{j+1})$. The new velocity is simply

$$\dot{\mathbf{q}}(t_{j+1}) = \frac{\mathbf{q}(t_{j+1}) - \mathbf{q}(t_j)}{\Delta t}. \tag{8.10}$$

The Jacobian matrix of plate and shell simulation in Eq. 8.6 is sparse but non-banded, and, in contrast with beam and rod models, we cannot achieve $O(N)$ time complexity. The difference in computation between rod vs. plate simulation becomes more and more significant as the number of nodes increases. Naturally, whenever possible, a rod model is favorable over a plate model.

An excellent resource for solving sparse linear systems is the **PARDISO** solver project [14, 15, 16] (https://www.pardiso-project.org/). It comes with a free software for academic users that is compatible with C/C++, MATLAB, and other languages and is computationally efficient.

8.5 Programming Implementation

This section discusses the programming implementation of the gradient and Hessian of the elastic energies. The other components of a discrete elastic plate or shell simulation are very similar to elastic beam simulation.

In a computer program, the elastic gradient is a vector of size $3N$. The gradient can be written as

$$\frac{\partial E_{\text{elastic}}}{\partial q_i} = \sum_{k=1}^{N_{\text{edge}}} \frac{\partial}{\partial q_i} E_{\text{s}}^k + \sum_{k=1}^{N_{\text{hinge}}} \frac{\partial}{\partial q_i} E_{\text{b}}^k, \tag{8.11}$$

where each of the terms under the summation symbol, $\left(\frac{\partial}{\partial q_i} E_{\text{s}}^k\right)$ and $\left(\frac{\partial}{\partial q_i} E_{\text{b}}^k\right)$, has non-zero component of size 6 and 12 for stretching and bending, respectively. Recall that each discrete stretching energy is associated with one edge, i.e. two nodes, corresponding to $2 \times 3 = 6$ DOFs. Each hinge in Fig. 8.1(b) is related to four nodes and $3 \times 4 = 12$ DOFs. The compute codes in the Appendix will output these non-zero

gradients of size 6 or 12. These "small" gradients will need to be mapped to the larger $3N$ sized gradient.

Here is an example. Assume that the bending at the k-th hinge involves node number m, n, o, and p, where m, n, o, and p are integers. The location of these nodes in the DOF vector is

$$\texttt{ind} = \big[3m-2, 3m-1, 3m, 3n-2, 3n-1, 3n, \\ 3o-2, 3o-1, 3o, 3p-2, 3p-1, 3p\big]. \quad (8.12)$$

The output of MATLAB code that computes the gradient of a discrete bending energy is a vector of size 12; let us denote it as `dF`. The full elastic gradient is stored in a $3N$ sized vector denoted as `F`. The output `dF` can be placed into `F` by a single line of code: `F (ind) = F (ind) + dF`.

Algorithm 3 shows a pseudocode to compute the gradient and Hessian of the elastic energies.

Algorithm 3 Gradient and Hessian of Elastic Energy Calculation

Require: $\mathbf{q}$ ▷ Degrees of Freedom
Require: k_s, k_b ▷ Elastic stiffness in Eq. 8.3
Require: $\bar{\theta}^k, l^k$ ▷ Undeformed bend angle and edge length
Ensure: `F` ▷ $3N$ sized elastic gradient vector, $\frac{\partial E_{\text{elastic}}}{\partial q_i}$
Ensure: `J` ▷ $3N \times 3N$ sized elastic Hessian matrix, $\mathbb{J}^{\text{elastic}}$

function GRAD_HESS_ELASTIC($\mathbf{q}$)
    `F` ← `zeros(3N,1)`
    `J` ← `zeros(3N,3N)`
    for $k \leftarrow 1$ to N_{edge} **do**
        $\mathbf{x}_0 \leftarrow$ nodal coordinates of first node of the k-th edge
        $\mathbf{x}_1 \leftarrow$ nodal coordinates of second node of the k-th edge
        `ind` ← locations of the two nodes in the DOF vector ▷ Eq. 8.12
        `[dF, dJ]` ← `gradEs_hessEs_Shell`($\mathbf{x}_0, \mathbf{x}_1$) ▷ Code in Appendix
        `F (ind)` = `F (ind)` + `dF`
        `J (ind)` = `J (ind, ind)` + `dJ`
    end for

    for $k \leftarrow 1$ to N_{hinge} **do**
        $\mathbf{x}_0 \leftarrow$ nodal coordinates of first node of the k-th hinge
        $\mathbf{x}_1 \leftarrow$ nodal coordinates of second node of the k-th hinge
        $\mathbf{x}_2 \leftarrow$ nodal coordinates of the third node on Triangle 1 ▷ See Fig. 8.1
        $\mathbf{x}_3 \leftarrow$ nodal coordinates of the third node on Triangle 2
        `ind` ← locations of the four nodes in the DOF vector ▷ Eq. 8.12
        `[dF, dJ]` ← `gradEb_hessEb_Shell`($\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$) ▷ Code in Appendix
        `F (ind)` = `F (ind)` + `dF`
        `J (ind)` = `F (ind, ind)` + `dJ`
    end for
    return `F` and `J`
end function

Assignment: Consider a clamped beam of rectangular cross-section. The length of the beam is $l = 0.1$m, width is $w = 0.01$m, and thickness is $h = 0.002$m. The Young's modulus is $Y = 10^7$ Pa. Density of the beam is 1000 kg/m^3.

- Write a computer program that uses a beam model to simulate the deformation of this thin beam under gravity.
- Write a computer program that uses a plate model to simulate the same problem.
- Compare your results from (1) beam model, (2) plate model, and (3) Euler-Bernoulli beam theory. Explore the dependence of your results with the mesh size in your plate simulation.

Chapter 9

Neural ODE for Dynamic Structure Model

This Chapter introduces applied knowledge of neural ordinary differential equation (Neural ODE). Neural ODE, as the name suggests, is an ordinary differential equation that is formulated by a neural network framework developed in 2018 by Chen et al. [17]. In particular, we will look into examples and applications of Neural ODE for mechanical/structural models. To facilitate the understanding, basic neural network formulation and the interplay of important hyper-parameters will be briefly introduced. PyTorch [18] will be used as the main application programming interface (API) for this Chapter, which is the dominating platform for the deep learning development community. After reading this Chapter, the reader should be able to understand and implement multi-layer perceptron (MLP) and the neural ODE application for a mechanical model.

9.1 Neural Network

The neural network was first suggested in 1943 by McCulloch and Pitts from biological findings from nervous activity [19]. With technological developments in computation, data, and popularization of back propagation (BP) methods [20], it has gained popularity again in the recent days for myriads of applications. The neural network can be understood as a function estimator for application purposes. With its innate data-hungry characteristics, the neural network finds the functional relationship between the set of input and output. After training of the network, regular trained MLP can interpolate within the scope of training and can be used as a function. network, regular trained MLP can interpolate within the scope of training, and can be used as a function.

9.1.1 Forward Pass

Figure 9.1 shows the basic structure of a neural network with two hidden layers and an output layer. x_1, x_2, x_3 are the inputs, o_1, o_2 are outputs. Inputs and outputs are

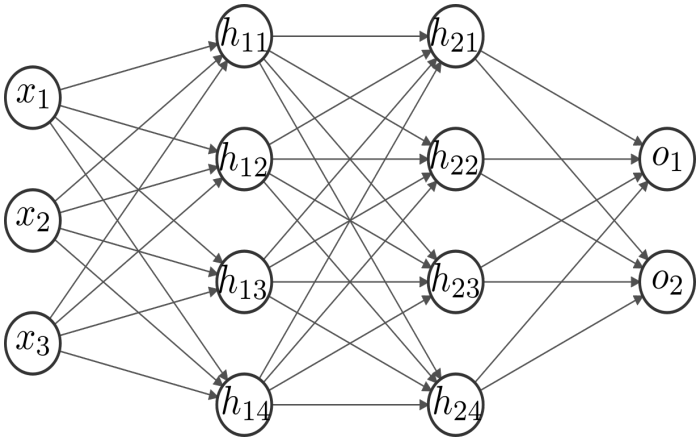

Figure 9.1: Example of a neural network architecture

related through hidden layers. Each hidden layer values can be expressed as a linear sum of weights and biases with an activation function. For example, the value for the first neuron of the first hidden layer can be represented as

$$h_{11} = f(W_{11}x_1 + b_{11}) + f(W_{21}x_2 + b_{21}) + f(W_{31}x_3 + b_{31}) \tag{9.1}$$

, where W and b with subscripts represent the weight components and bias components for the weight matrix of the first hidden layer, and f representing nonlinear activation function. There exist different activations functions such as sigmoid, tanh, ReLU ,leaky ReLU, Maxout. Typically for the hidden layers choice of ReLU and leaky ReLU is a good approach [21]. In a matrix form, the first hidden layer can be represented to be

$$\mathbf{h_1} = \mathbf{f}(\mathbf{W_1}\mathbf{x} + \mathbf{b_1}) \tag{9.2}$$

, where $\mathbf{h_1} \in \mathbb{R}^{m\times 1}$, $\mathbf{W_1} \in \mathbb{R}^{m\times n}$, $\mathbf{x} \in \mathbb{R}^{n\times 1}$,$\mathbf{b_1} \in \mathbb{R}^{m\times 1}$ with $m =$ (number of neurons in first layer), and $n =$ (number of input neurons).
Then the relationship between each hidden layers, input and output for the neural network in Figure 9.1 can be represented as

$$\mathbf{h_1} = \mathbf{f}(\mathbf{W_1}\mathbf{x} + \mathbf{b_1}) \tag{9.3}$$

$$\mathbf{h_2} = \mathbf{f}(\mathbf{W_2}\mathbf{h_1} + \mathbf{b_2}) \tag{9.4}$$

$$\mathbf{o} = \mathbf{g}(h_2) \tag{9.5}$$

, where $\mathbf{h_2} \in \mathbb{R}^{k\times 1}$, $\mathbf{W_1} \in \mathbb{R}^{k\times m}$, $\mathbf{b_2} \in \mathbb{R}^{k\times 1}$ with $k =$ (number of neurons in second layer). We can use the same activation function for the output layers too but the output layer typically uses a different activation function from the hidden layers and is dependent upon the type of prediction required by the model.

Activation functions are crucial components of a neural network as a way to deal with nonlinear patterns of data, making the neural network capable to solve complex problems. With increased understanding of the application of the activation function, multiple activation functions have been suggested, such as sigmoid, tanh, softmax,

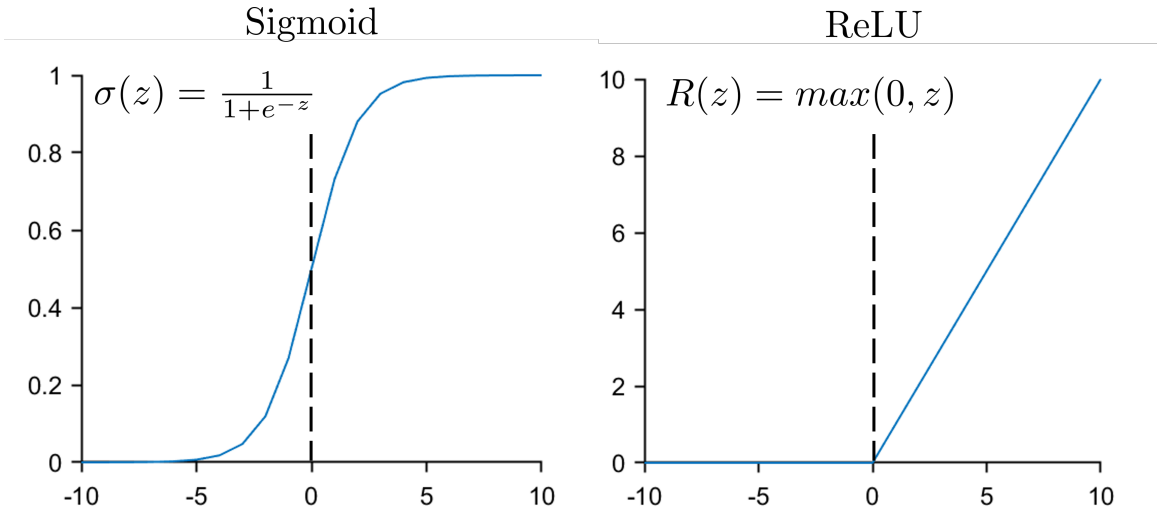

Figure 9.2: Nonlinear activation function : Sigmoid, ReLU

ReLU ,leaky ReLU, ELU and maxout. Basic nonlinear activation functions such as sigmoid, tanh and softmax can exhibit vanishing gradient problem which can lead to training getting stuck due to too small value of gradient with respect to the loss. This can often happen as the number of layers increases. ReLU and its family is the most frequently used activation function. The family of ReLU was developed due to the fact that a ReLU could cause dead neurons in the neural network, and you also need to carefully monitor the learning rate. A family of ReLU, such as leaky ReLU, makes the neurons not die by preventing the ReLU value from being 0 and having "leaks" for the negative input to the ReLU.

The choice of the output activation functions is highly dependent on the problem (whether it is a classification or a regression problem, etc.). Sigmoid can be used for a simple binary classification problem. Softmax is often used for multi-class classification by mapping multinominal probability distribution across the class. ReLU is often used for the regression problem.

9.1.2 Back Propagation

Using the relationship shown in Section 9.1.1, our goal for training is to update the weights and biases of the hidden layers through gradient descent to reduce the loss to be as small as possible. The loss should be some measure of the difference between the ground truth and the output calculated through the neural network and chosen based on the object of the networks. This process of finding out the gradient from loss to backwards through the neural network is called back propagation. Using the gradients found through backward stepping, we can use gradient descent to find the best network parameters (weights and biases). In optimizing the gradient descent, there are several methods that are used, but adaptive moment estimation (ADAM) and root mean squared propagation (RMSprop) algorithms are the safe algorithms that have been proven efficacy for use on a wide range of deep neural networks. One

step of this update process for all the number of samples is called an **epoch**. The details of the history and development of different optimization algorithms could be found in the Ian Goodfellow's Deep Learning, Chapter 8 [22]. In this subsection, we will conceptually understand the back propagation process and work out a simple example of the manual calculation of backpropagation.

Gradient Descent

To understand the back propagation process, one needs to understand gradient descent first, which is an update rule for the neural network parameters (weights and biases). In Figure 9.3, a function $J(\theta_1)$, parametrized by θ_1 is being optimized through gradient descent with a learning rate ($\alpha > 0$). **Learning rate** can be interpreted as step size when performing gradient descent. The update rule of gradient descent is shown as $\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$, by taking the gradient of the function $J(\theta_1)$ with respect to the parameter θ_1 at the given instance of parameter value, we can reach local minimum through this gradient descent algorithm. As the name tells, we descend to the local minimum of a function by updating the parameter of the function. One interesting point to look at is the learning rate, α. By setting the learning rate $\alpha > 0$, we can ensure the parameter reaches the local minimum and descends its way down regardless of the sign of the gradient. Case 1 and 2, denoted in Figure 9.3 show each positive and negative gradient value, and, as expected, when the gradient is positive, the parameter update is done through subtraction of the (gradient)*(learning rate), where when it is negative, the negative sign of the gradient makes it into an addition. It is also important to note that the learning rate should be a value very small (< 1), due to the zig-zagging of the parameter update. When the learning rate is big convergence might not be achieved.

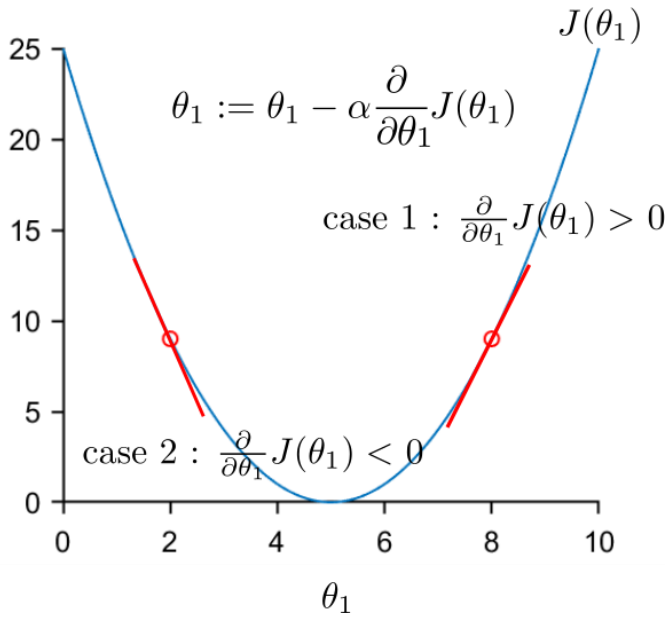

Figure 9.3: Gradient descent update rule

The goal of updating the network parameters can be approached a similar way. However, when it comes to the deep neural network or DNNs, the approach to finding the gradient within all the training data for a deep network increases the computational cost exponentially. Compared to the example provided in the Figure 9.3, the size of parameters that need to be updated in a single epoch (θ_i in the Figure) of the actual neural network should be, 2*(input dimension)*(the number of neurons of a neural network) and the updates of this theta requires going through (the number of samples) with much more complex target function. The computational cost both increases with data size and the number of parameters of DNN that we need to train (weights and biases).

Therefore, the methods based on stochastic gradient descent (SGD) are often used. The main difference of SGD from the gradient descent is that the loss is approximated over the data through randomized mini-batches or sampling methods. The benefit of this method is that as the data size grows, the accuracy of this loss estimate becomes more accurate. By introducing stochastic gradient descent, we can overcome the challenges of computational cost introduced by handling the disadvantage into advantage directly. While we now understand that SGD is used to train the NN, the optimization for this approach also varies. There exist techniques based on 1) momentum (incentivizing change in parameter by the large gradient at an instance), or 2) adaptive learning rates by scaling the learning rate inversely proportional to the square root of the sum of all the historical squared values of the gradient (such as AdaGrad)[22]. However, the most known **optimizer** that are discovered to be reliable to the users are ADAM and RMSprop.

Calculating Gradients Step-by-Step (Scalar Example)

Now that we understand the gradient descent, we will learn how to find the gradient for each weights and biases of the neural network. It is important to note that the gradient we calculate here is neither numerically nor symbolically calculated. The essence of this simplistic method comes from utilizing the chain rule in a reverse manner [23]. We will first look into scalar examples, learn about the gate view of the gradients, and go through matrix/tensor derivative examples. The major work shown in this course text is inspired and adapted from Stanford CS231n course notes available open source [21].

First, we should steer ourselves to understand how the chain rule works backwards. Assuming that we were given a function f, where $f(w, b, x) = wx+b$, and a derivative of some loss $L(f)$ with respect to function f, $\frac{\partial L}{\partial f}$. Then, we can calculate the derivative of the loss L, with respect to w, x, and a using the chain rule relationship between $\frac{\partial L}{\partial f}$ and the local gradients. In other words, $\frac{\partial L}{\partial w} = \frac{\partial f}{\partial w}\frac{\partial L}{\partial f}$, $\frac{\partial L}{\partial b} = \frac{\partial f}{\partial b}\frac{\partial L}{\partial f}$, $\frac{\partial L}{\partial x} = \frac{\partial f}{\partial x}\frac{\partial L}{\partial f}$.

$$\mathbf{f}(\mathbf{w},\mathbf{x}) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

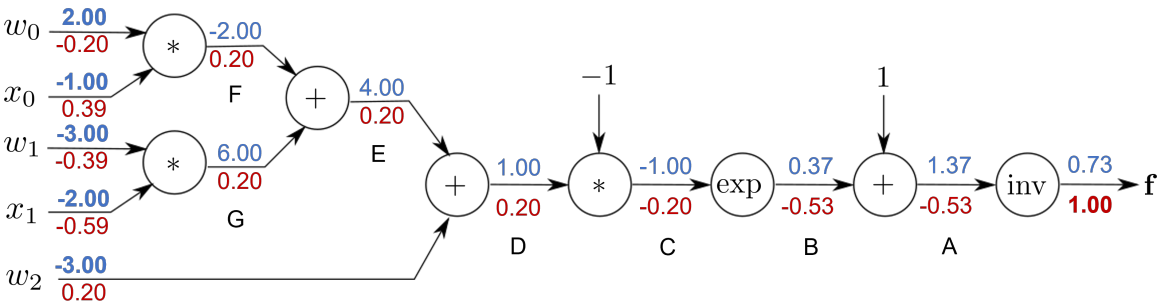

Figure 9.4: Scalar example : 2 dimensional neuron with sigmoid activation function

Table 9.1: Forward and derivative trace for scalar example. (*right*, and $left$ direction refers to the direction on Figure 9.4)

Forward primal trace ($left \rightarrow right$)	Derivative trace ($right \rightarrow left$)
$w_0 = 2, x_0 = -1, w_1 = -3,$	
$x_1 = -2, w_2 = -3$ (given)	$\frac{\partial L}{\partial f} = 1$ (given)
$w_0 x_0 = F = -2$	$\frac{\partial L}{\partial A} = \frac{\partial f}{\partial A}\frac{\partial L}{\partial f}$
$w_1 x_1 = G = 6$	$\frac{\partial L}{\partial B} = \frac{\partial A}{\partial B}\frac{\partial L}{\partial A}$
$F + G = E = 4$	$\frac{\partial L}{\partial C} = \frac{\partial B}{\partial C}\frac{\partial L}{\partial B}$
$E + w_2 = D = 1$	$\frac{\partial L}{\partial D} = \frac{\partial C}{\partial D}\frac{\partial L}{\partial C}$
$(-1) * D = C = -1$	$\frac{\partial L}{\partial E} = \frac{\partial D}{\partial E}\frac{\partial L}{\partial D}$
$e^C = B = 0.37$	$\frac{\partial L}{\partial F} = \frac{\partial E}{\partial F}\frac{\partial L}{\partial E}$
$B + 1 = A = 1.37$	$\frac{\partial L}{\partial G} = \frac{\partial E}{\partial G}\frac{\partial L}{\partial E}$
$\frac{1}{A} = f = 0.73$	

We can see it more clearly with the scalar example shown in Figure 9.4 and Table 9.1.2. Here, the values denoted over the arrows are the forward pass, and the values denoted below the arrows are the values from the backward pass. After the forward pass, we can get intermediate values and their relations, as shown on the right side of the table. Using these, we can now follow the derivative trace. Let's first find the intermediate gradient $\frac{\partial L}{\partial A}$. We assumed $\frac{\partial L}{\partial f} = 1$ is given, we can also find $\frac{\partial f}{\partial A}$ by taking the derivative of the relationship between f and A given by the last row of the primal trace, $\frac{1}{A} = f$, the derivative of f, with respect to A can be then defined to be $-\frac{1}{A^2}$. We can then substitute the value of A from the primal trace and can get the value of $\frac{\partial f}{\partial A} = -\frac{1}{A^2} = -\frac{1}{(1.37)^2} = -0.5328$. This value is denoted as red below the arrow on Figure 9.4. Since we have every primal trace and intermediate relations defined from the primal trace, we then can use the same logic to calculate the gradients for every intermediate value like the example shown above.

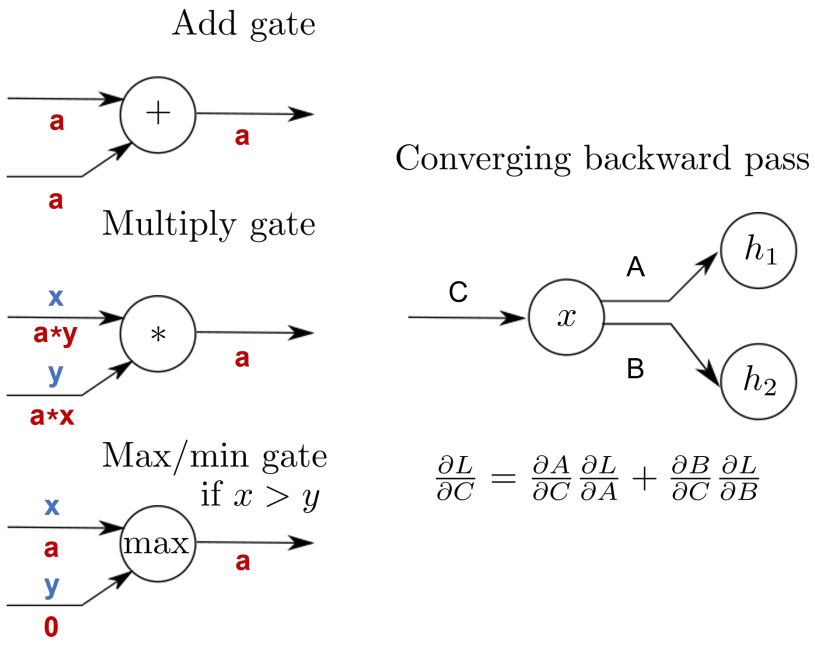

Figure 9.5: Common gate rules for backward pass of gradient

In order to simplify the manual back propagation process, Some rules have been found and could be easily applied when manually calculating the gradients. These rules are shown in Figure 9.5

Calculating Gradients Step-by-Step (Neural Network Example)

The only difference from the Scalar example for the neural network example is that each intermediate values are in vector form. This implies that we need to understand how to calculate

1. Derivative of a scalar with respect to a vector
2. Derivative of a vector with respect to a vector
3. **Derivative of a scalar with respect to a matrix**
4. **Chain rule for vectors**

Until Chapter 7 of this course note, we have dealt with gradient and hessian of the elastic energies. Therefore, we have already learned how to take a derivative of a scalar with respect to a vector from finding out the gradient of elastic energy and a vector with respect to a vector from calculating the hessian of the elastic energy. We will introduce the derivative of a scalar with respect to a matrix here and learn how the chain rule for vectors is organized through the neural network example. The derivative of a scalar y, with respect to a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, can be expressed as,

$$\nabla_{\mathbf{A}} y = \begin{bmatrix} \frac{y}{\partial a_{11}} & \frac{y}{\partial a_{12}} & \cdots & \frac{y}{\partial a_{1n}} \\ \frac{y}{\partial a_{21}} & \frac{y}{\partial a_{22}} & \cdots & \frac{y}{\partial a_{2n}} \\ \cdots & \cdots & \ddots & \vdots \\ \frac{y}{\partial a_{m1}} & \frac{y}{\partial a_{m}2} & \cdots & \frac{y}{\partial a_{mn}} \end{bmatrix} \tag{9.6}$$

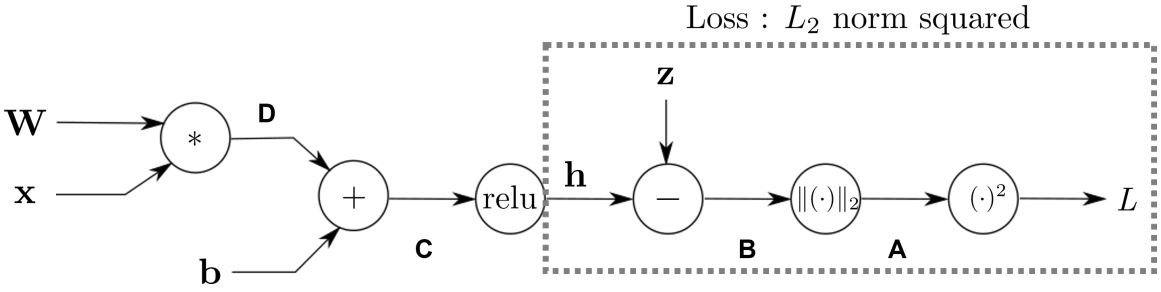

Figure 9.6: Back propagation : neural network example

Similar to the scalar example, Figure 9.6 and Table 9.2 shows the forward and backward trace of the neural network propagation. For this example initial setup is given as,

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}, \mathbf{W} = \begin{bmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \\ W_{31} & W_{32} & W_{33} \\ W_{41} & W_{42} & W_{43} \end{bmatrix} \tag{9.7}$$

Table 9.2: Forward and derivative trace for neural network example. (*right*, and *left* direction refers to the direction on Figure 9.6)

Forward primal trace ($left \rightarrow right$)	Derivative trace ($right \rightarrow left$)
$\mathbf{D} = \begin{bmatrix} W_{11}x_1 + W_{12}x_2 + W_{13}x_3 \\ W_{21}x_1 + W_{22}x_2 + W_{23}x_3 \\ W_{31}x_1 + W_{32}x_2 + W_{33}x_3 \\ W_{41}x_1 + W_{42}x_2 + W_{43}x_3 \end{bmatrix}$	$\frac{\partial L}{\partial A} = 2A$
$\mathbf{C} = \mathbf{D}+\mathbf{b}$	$\frac{\partial L}{\partial \mathbf{B}} = \frac{\partial A}{\partial \mathbf{B}} \frac{\partial L}{\partial A}$ - scalar w.r.t vector
$= \begin{bmatrix} W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1 \\ W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2 \\ W_{31}x_1 + W_{32}x_2 + W_{33}x_3 + b_3 \\ W_{41}x_1 + W_{42}x_2 + W_{43}x_3 + b_4 \end{bmatrix}$	$\frac{\partial L}{\partial \mathbf{h}} = \frac{\partial \mathbf{B}}{\partial \mathbf{h}} \frac{\partial L}{\partial \mathbf{B}}$ - vector w.r.t vector
$\mathbf{h} = max(\mathbf{0},\mathbf{C})$	$\frac{\partial L}{\partial \mathbf{C}} = \frac{\partial L}{\partial \mathbf{h}}$ - if C > 0, else 0
$= max(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} , \begin{bmatrix} W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1 \\ W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2 \\ W_{31}x_1 + W_{32}x_2 + W_{33}x_3 + b_3 \\ W_{41}x_1 + W_{42}x_2 + W_{43}x_3 + b_4 \end{bmatrix}$	$\frac{\partial L}{\partial \mathbf{D}} = \frac{\partial \mathbf{C}}{\partial \mathbf{D}} \frac{\partial L}{\partial \mathbf{C}}$ - vector w.r.t vector
$\mathbf{B} = \mathbf{h} - \mathbf{z}$	$\frac{\partial L}{\partial \mathbf{b}} = \frac{\partial \mathbf{C}}{\partial \mathbf{b}} \frac{\partial L}{\partial \mathbf{C}}$ - vector w.r.t vector
$A = \|\mathbf{B}\|_2$	$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial \mathbf{D}}{\partial \mathbf{x}} \frac{\partial L}{\partial \mathbf{D}} = \mathbf{W}^T \frac{\partial L}{\partial \mathbf{D}}$- vector w.r.t vector
$L = A^2$	$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{D}} \mathbf{x}^T$ - vector w.r.t tensor

$$\mathbf{h} = max(\mathbf{0}, \mathbf{W}\mathbf{x} + \mathbf{b}) = max(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} , \begin{bmatrix} W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1 \\ W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2 \\ W_{31}x_1 + W_{32}x_2 + W_{33}x_3 + b_3 \\ W_{41}x_1 + W_{42}x_2 + W_{43}x_3 + b_4 \end{bmatrix}) \tag{9.8}$$

, and also $\mathbf{z} \in \mathbb{R}^{4\times 1}$ being the ground truth vector for the given input $\mathbf{x}$. The difference is noted in the derivative trace, where there are some operations that require methods of taking derivatives of different sized vectors and scalars with respect to the vectors. Note that the last two rows of the derivative trace shows how the derivative with respect to tensor is calculated for $\mathbf{x}$ and $\mathbf{W}$. Also, it is important to note that the derivative of vectors is intentionally written right to the left based on our convention of denominator layout notation.

Back Propagation Pseudocode

We learned how gradient descent works and how to calculate the gradients of the networks with respect to the loss. We now can make updates on our neural network. In total, the pseudocode for back propagation for a neural network and DNN can be expressed simply as the pseudocode below.

Algorithm 4 Back Propagation Pseudocode

Require: Neural network architecture forward pass
Require: Initialization of the weights
Require: Mini-batching of the dataset
Require: Other necessary regularization
Require: Choice of optimizer
Require: Preset of learning rate (lr)
Ensure: Single update of network parameters **W**s and **b**s

1: **function** $\text{BP}_{\text{UPDATE}}$(batchsize, lr)
2: Calculate the gradients of the network parameters (**W**,**b**) w.r.t. **L**
3: Run optimizer update (batchsize, lr)
4: **return** Updated network parameters
5: **end function**

9.1.3 Regularization in Neural Network

So far, we have looked briefly into building a simple neural network. To learn quickly about the core idea of the neural network (or MLP), we have not introduced some of the essential components of the neural networks, which cannot be ignored in improving model performances. Especially, regularization methods, which comprise of initialization of weights and biases, batch normalization, data augmentation, drop-out layers, model ensemble, and more, are some of the other topics that you can look into to understand the neural network better and improve your model. The book Deep Learning, Chapter 7 [22], provides in depth analysis of regularization techniques.

9.2 Neural Ordinary Differential Equation

This course focuses on ML-assisted Physics simulation (or MAL Physics) approach for the application of the Neural ODE. Neural ODE provides higher generality and simpler network architectures and capabilities to extrapolate in time for a dynamical system which is more suitable for MAL Physics approach than the regular MLP or DNN-based solution. Simply put, Neural ODE is an approach to make the ordinary differential equation in the form of neural network.

9.2.1 Neural ODE Architecture

Ordinary differential equation (ODE) is an equation which are composed of unknown function and its derivatives with a single independent variable. One may define a general ODE into,

$$\frac{\partial \mathbf{h}(\mathbf{t})}{\partial \mathbf{t}} = f(\mathbf{h}(\mathbf{t}), \mathbf{h}'(\mathbf{t}), ..., \mathbf{t}) \tag{9.9}$$

, where t is the independent variable and h is the dependent variable. The goal of solving an ODE is to find out the unknown equation $\mathbf{h}(\mathbf{t})$ from the relationship shown in Equation 9.9, or a solution of an initial value problem (a particular value) h at some time t_{final}. The goal of neural ODE is to train the unknown function f in to a neural network using the output from the ODE solver of function ($\frac{\partial \mathbf{h}(\mathbf{t})}{\partial \mathbf{t}}$), and known input of function $(\mathbf{h}(\mathbf{t}), \mathbf{h}'(\mathbf{t}), ..., \mathbf{t})$. This dynamics f is evaluated by a black-box differential equation solver ,whenever necessary, to determine the solution with the desired accuracy. The training process of neural ODE is depicted in Figure 9.7.

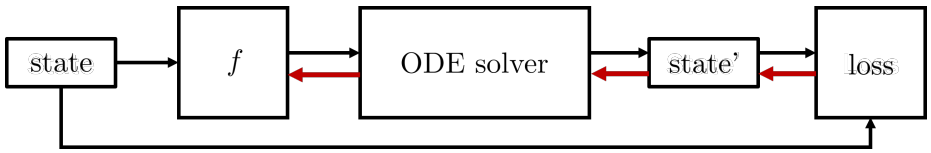

Figure 9.7: Neural ODE architecture schematics during training. Black arrows refer to forward pass. Red arrows refer to back propagation. f is the ODE in the form of NN, which is our objective of training. state' refers to the output from the ODE solver.

The input for the training of the neural ODE is the time series of known states $(\mathbf{h}(\mathbf{t}), \mathbf{h}'(\mathbf{t}), ..., \mathbf{t})$. The state is fed into the neural network f, then the output of the neural ODE is fed into the ODE solver. The ODE solvers then output the state (denoted as state' in the figure), which is the solution of the Ordinary differential equation f. The loss is measured based on the difference between the NN output state' and state, which were the input to the system. The main contribution of the authors of the neural ODE is that they have developed a method that associated the back propagation through the existing ODE solver, including the introduction of the memory-efficient back propagation through the ODE solver through adjoint sensitivity method [17]. When the neural ODE f is trained, we can use the trained

ODE in conjunction with an ODE solver and treat it as any other ODEs that are often used in the field of engineering.

In this course, we approach learning the Neural ODE in terms of application to a dynamic system. However, it is important to note that another strength of neural ODE can make a residual relationship for a Resnet-like neural network architecture into a dynamics problem where the number of a residual layer need not be defined [17].

9.2.2 Application of Neural ODE to Dynamic System

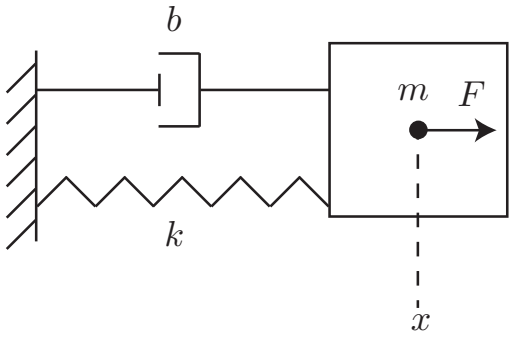

Figure 9.8: A single DOF mass-spring-damper system.

The application of Neural ODE to dynamic systems can achieve multiple objectives, such as system ID, augmenting the known physics model with Neural ODE filling in the missing physics, building reduced order models, and more. In this section, we will explain a simple example by formulating a single DOF mass spring damper system into a neural ODE.

For a physical system, the independent variable is time, and the dependent variable is position, velocity, or acceleration. When modeling a physical system, we usually model the system using Newton's Second law, $\mathbf{F} = \mathbf{ma}$. Often times however, when doing experiment of a dynamic system, there exists missing physics, perturbation, measurement error, and other errors that arises. Neural ODE can help identify the system, augment the missing physics, and can help build a reduced model.

Similar to simple example from Chapter 2, a 1-DOF dynamic system shown in Figure 9.8 with $\mathbf{F} = 0$ will be used. However, the approach is backward, using neural ODE from only the time series of state data as both input and output. Here we will denote the equations of motion, or the dynamic model of this physical system to be,

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{c}{m} \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} \tag{9.10}$$

, where k is the spring constant, c is the damping constant, and m is mass. For the example that we will introduce in this section, the ground truth values are $k = 10$, $m = 1$, $c = 1$. However, for this example, we assume that we do not have any knowledge of the system. The only information we have is the time series of state values (position, x and velocity, $\dot{x}$) generated from the ground truth. Given n as size

of data, m as batch size, and l as the batch time, we can randomly pick multiple batches of time series data for every iteration, as shown in Figure 9.9. The code snippet for data generation is shown in Figure 9.10. Our goal is to use these batches of data to train the neural network to verify and compare the result of the trained neural ODE with the given model.

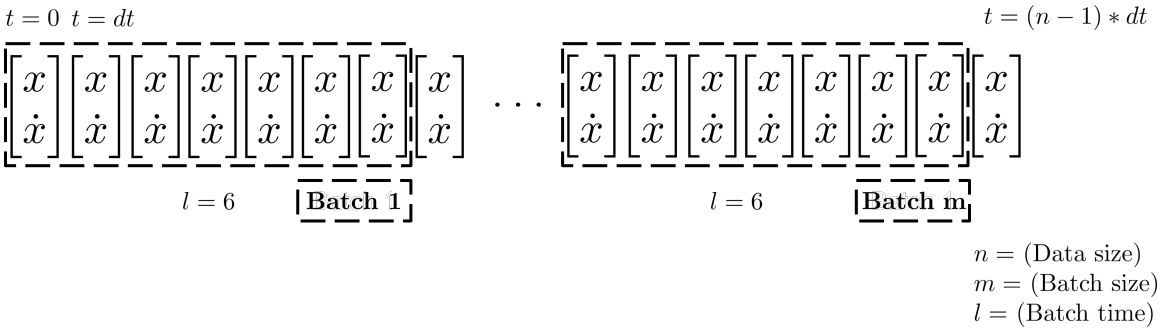

Figure 9.9: Description of data size, batch size, batch time for 1DOF example

```python
# Set parameters
viz = True
test_freq = 100
save_data = False
save_freq = 10
method = 'rk4'
batch_time = 20 # time series data interval (20 time step worth of data) for a single batch
data_size = 100 # total number of data
data_size_test = 1000 # total number of data for testing
batch_size = 5 # number of batches to pick from. size should be
niters = 1000 # number of iterations.

# Generate ground truth and batch function.
true_y0 = torch.tensor([[2., 0.]]).to(device) # initial condition for training
t = torch.linspace(0., 10., data_size).to(device)

true_y0_test = torch.tensor([[0., 2.]]).to(device) # initial condition for test
t_test = torch.linspace(0., 10., data_size).to(device)

class Lambda(nn.Module):

    def forward(self, t, y):
        m, c, k = 1, 1, 10
        coeffMatrix = torch.tensor([[0,-k/m],[1,-c/m]]).to(device)
        # first column displacement, second column velocity
        return torch.mm(y,coeffMatrix)


with torch.no_grad(): # disable gradient calculation for true values output
    true_y = odeint(Lambda(), true_y0, t, method=method)
    true_y_test = odeint(Lambda(), true_y0_test, t_test, method=method)
print(true_y.shape)

def get_batch():
    s = torch.from_numpy(np.random.choice(np.arange(data_size - batch_time, dtype=np.int64), batch_size, replace=False))
    batch_y0 = true_y[s]  # (M, D)
    batch_t = t[:batch_time]  # (T)
    batch_y = torch.stack([true_y[s + i] for i in range(batch_time)], dim=0)  # (T, M, D)
    return batch_y0.to(device), batch_t.to(device), batch_y.to(device)
```

Figure 9.10: Code snippet that populates the training and test data

As depicted in Figure. 9.7, we need to first define the ODE f as a neural network. For our example, we used a single-layer neural network with 20 neurons to train the ODE. From the existing literature, neural ODE does not require a deep neural network to be trained with accuracy. Most of the architecture applied for dynamic

systems have 1 or 2 hidden layers depending on their tasks. The detail of the neural ODE architecture snippet is shown in Figure. 9.11.

```
# Building NN ODE

class ODEFunc(nn.Module):

    def __init__(self):
        super(ODEFunc, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(1, 20),
            nn.Softplus(),
            nn.Linear(20, 1),
        )

        self.register_buffer('coeffMatrix1',torch.tensor([[0,0],[1,-1/1]]).float())
        self.register_buffer('coeffMatrix2',torch.tensor([[0,-1/1],[0,0]]).float())

        for m in self.net.modules():
            if isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, mean=0, std=0.1)
                nn.init.constant_(m.bias, val=0)

    def forward(self, t, y):
        # print("doing forward pass")
        yp = y[...,0:1]
        yp.requires_grad_(True) # defines that the gradient is needed to be calculated for the yp
        # print(y.shape)
        # print(y[:,:,0:1].shape)
        out = self.net(yp) + self.net(-yp)
        # print(out.shape)
        deriv = torch.autograd.grad([out.sum()],[yp],retain_graph=True,create_graph=True)
        grad = deriv[0]
        if grad is not None:
            return torch.matmul(y,self.coeffMatrix1) + torch.matmul(torch.cat((grad,torch.zeros_like(grad)),-1),self.coeffMatrix2)
```

Figure 9.11: Code snippet that uses PyTorch to build neural ODE function

Once the neural ODE is built, then the neural ODE needs to be fed in with the input to generate the time differential of the state. The resulting output from the neural ODE is then fed into the ODE solver (*torchdiffeq* library) that solves for the solution of the ODE and also enables seamless integration with PyTorch backpropagation. The loss is calculated as the mean absolute error (MAE). After the forward pass, gradients are calculated, and the optimizer steps to update the weight of the neural ODE. It is important to note that in every iteration, a new batch of data set is used for a forward and backward pass in this example. The detailed code snippet of this process is shown in Figure. 9.12. The results shown in Figure. **??** shows how well the dynamics are captured after 100 iteration and 1000 iterations. The green lines are the ground truth, and the blue dotted lines are the results from neural ODE. Even though the initial 100 iterations shown in Figure. 9.13 are very inaccurate, after 1000 iterations, we can see from Figure. 9.14 that the neural ODE can capture the dynamics accurately for both training and test data. The full Google Colab file is provided in the lecture.

```
# Main function
if __name__ == '__main__':

    ii = 0
    func = ODEFunc().to(device) # transform the tensor fitted to device either CPU and GPU

    # optimizer = optim.RMSprop(func.parameters(), lr=1e-3)
    optimizer = optim.Adam(func.parameters(), lr=1e-3)
    end = time.time()

    for itr in range(1, niters + 1):
        # print(str(itr) + ": before the iteration")
        optimizer.zero_grad() # initialize
        batch_y0, batch_t, batch_y = get_batch()
        pred_y = odeint(func, batch_y0, batch_t).to(device)
        loss = torch.mean(torch.abs(pred_y - batch_y))
        loss.backward()
        optimizer.step()
        # print(str(itr) + "after backward")

        if itr % test_freq == 0:

            # with torch.no_grad():
            pred_y = odeint(func, true_y0, t)
            loss = torch.mean(torch.abs(pred_y - true_y))
            print('Iter {:04d} | Training total Loss {:.6f}'.format(itr, loss.item()))
            visualize(true_y, pred_y, t, func, ii)

            pred_y_test = odeint(func, true_y0_test, t_test)
            loss = torch.mean(torch.abs(pred_y_test - true_y_test))
            print('Iter {:04d} | Test total Loss {:.6f}'.format(itr, loss.item()))
            visualize(true_y_test, pred_y_test, t_test, func, ii)
            ii += 1

        if itr % save_freq == 0:
            pred_y = odeint(func, true_y0, t)
            pred_y_test = odeint(func, true_y0_test, t_test)
            savedata(true_y, pred_y, t, func, itr, 'train')
            savedata(true_y_test, pred_y_test, t_test, func, itr, 'test')

        # print(str(itr) + "after test")
        end = time.time()
```

Figure 9.12: Code snippet that uses PyTorch to build training architecture

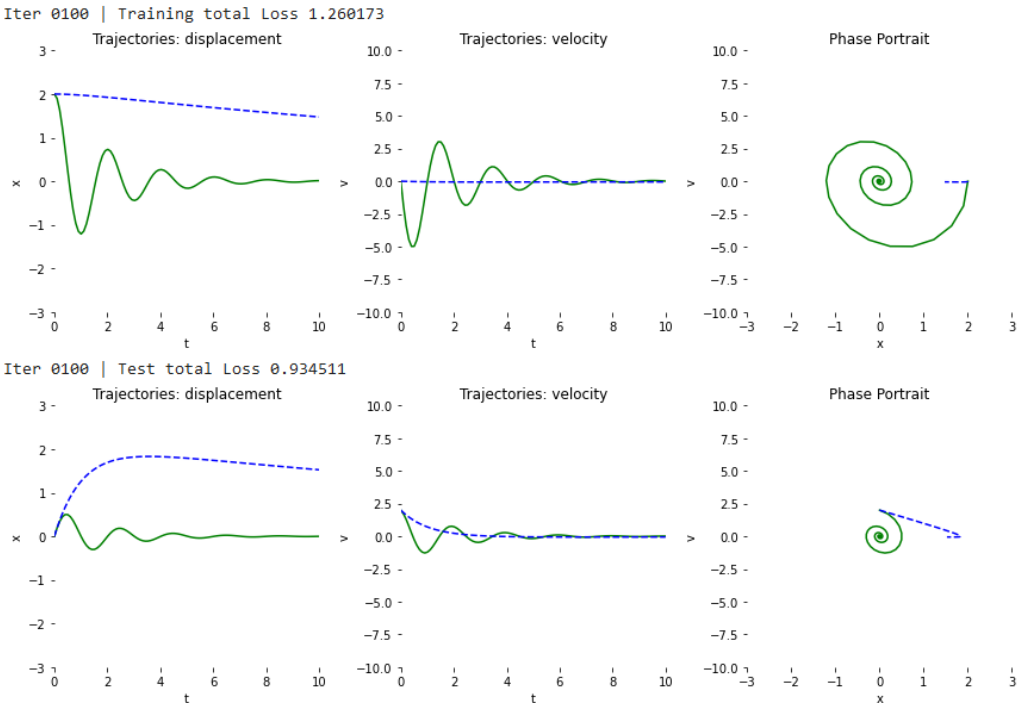

Figure 9.13: Time response of training (upper row) and test (lower row) of neural ODE at 100 iterations

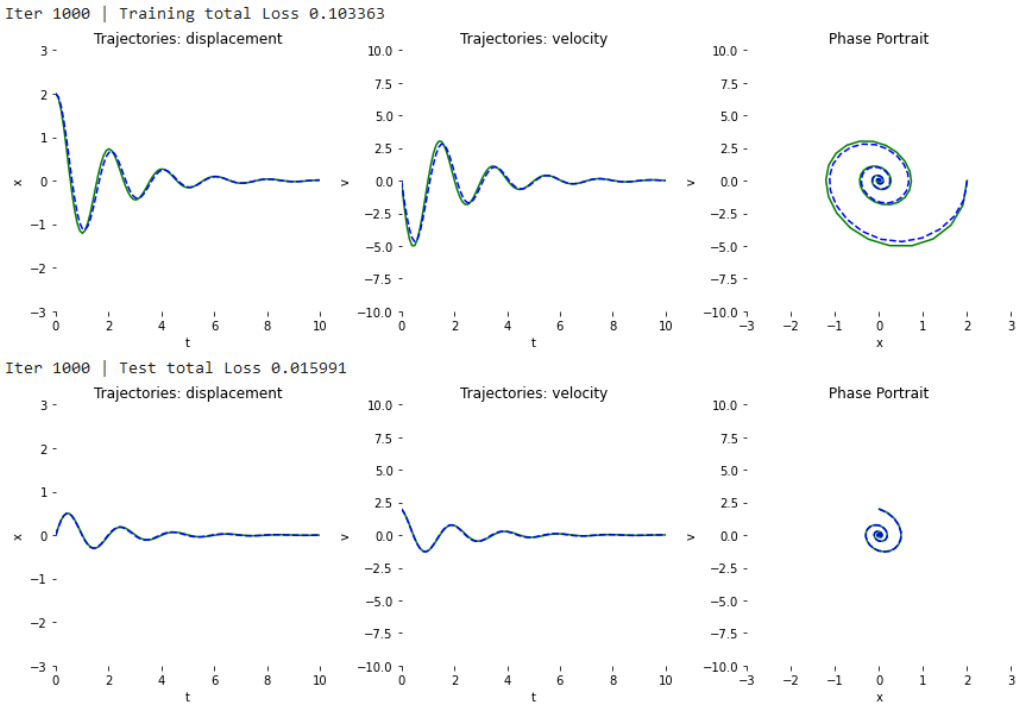

Figure 9.14: Time response of training (upper row) and test (lower row) of neural ODE at 1000 iterations

9.2.3 Application of Neural ODE to DER

Neural ODE is a technique that benefits from the structure of its architecture. We cannot limit the application of the neural ODE with DER or any other dynamic system in a single way. As previously mentioned in Section. 9.2.2, the known application of neural ODE to dynamic systems vary from system ID, augmenting the known physics model, filling in the missing physics, building reduced order models, and others.

However, in this section, we will introduce one example of the application of neural ODE can be for modeling of unknown external forces. Fig. 9.15 shows an example of a 1 DOF mass spring damper system where spring constant and elastic forces are unknown. We then can formulate the unknown spring forces into neural ODE from the experiment. DER can be explained as a multi-DOF mass spring damper system. We can use the experiment and simulation in conjunction with each other to develop and compensate for the unknown dynamics of the external forces due to fluid interaction, vibration, and others.

Another application of DER coupling with neural ODE is the development of the reduced-order model. A recent study from Li et al., utilizes neural ODE as a tool to reduce the energy function of a 3D slinky into a 2D formulation. The generated force/energy data was from the 3D slinky, and the reduced elastic energy term was considered as unknown dynamics in 2D. The results successfully display a regeneration of the 3D motion through 2D DER [24].

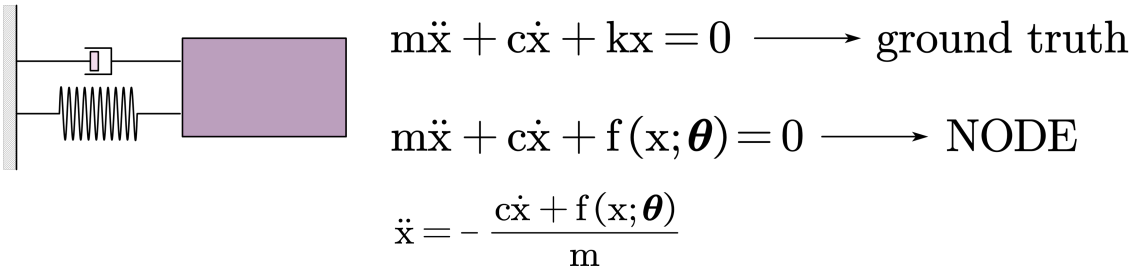

Figure 9.15: Example of filling in the unknown spring force using Neural ODE

Appendices

A MATLAB Code for Force and Jacobian in 2D (Beam)

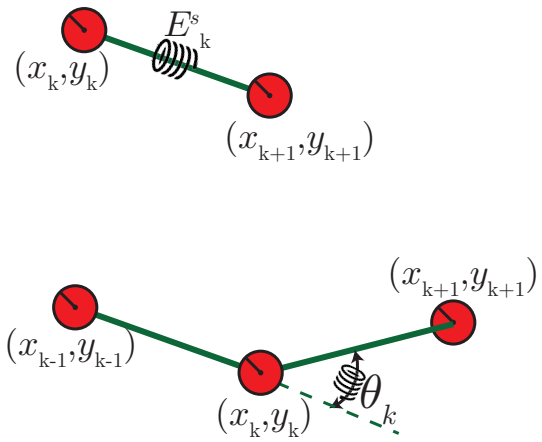

Figure 16: Stretching and bending springs.

Gradient and Hessian of Stretching Energy:

1. `F = gradEs(xk, yk, xkp1, ykp1, l_k, EA)` takes six inputs: $x_k, y_k, x_{k+1}, y_{k+1}, \Delta l, EA$; and returns a vector with 4 elements that contains the following gradient:

$$\begin{bmatrix} \frac{\partial E^s_k}{\partial x_k} \\ \frac{\partial E^s_k}{\partial y_k} \\ \frac{\partial E^s_k}{\partial x_{k+1}} \\ \frac{\partial E^s_k}{\partial y_{k+1}} \end{bmatrix} \tag{11}$$

   Derivative of E^s_k with respect to any other degree of freedom is zero.

2. `J = hessEs(xk, yk, xkp1, ykp1, l_k, EA)` takes six inputs: $x_k, y_k, x_{k+1}, y_{k+1}, \Delta l, EA$; and returns a matrix with 4×4 elements that contains the following Hessian:

$$\begin{bmatrix} \frac{\partial^2 E_k^s}{\partial x_k \partial x_k} & \frac{\partial^2 E_k^s}{\partial y_k \partial x_k} & \frac{\partial^2 E_k^s}{\partial x_{k+1} \partial x_k} & \frac{\partial^2 E_k^s}{\partial y_{k+1} \partial x_k} \\ \frac{\partial^2 E_k^s}{\partial x_k \partial y_k} & \frac{\partial^2 E_k^s}{\partial y_k \partial y_k} & \frac{\partial^2 E_k^s}{\partial x_{k+1} \partial y_k} & \frac{\partial^2 E_k^s}{\partial y_{k+1} \partial y_k} \\ \frac{\partial^2 E_k^s}{\partial x_k \partial x_{k+1}} & \frac{\partial^2 E_k^s}{\partial y_k \partial x_{k+1}} & \frac{\partial^2 E_k^s}{\partial x_{k+1} \partial x_{k+1}} & \frac{\partial^2 E_k^s}{\partial y_{k+1} \partial x_{k+1}} \\ \frac{\partial^2 E_k^s}{\partial x_k \partial y_{k+1}} & \frac{\partial^2 E_k^s}{\partial y_k \partial y_{k+1}} & \frac{\partial^2 E_k^s}{\partial x_{k+1} \partial y_{k+1}} & \frac{\partial^2 E_k^s}{\partial y_{k+1} \partial y_{k+1}} \end{bmatrix} \tag{12}$$

Gradient and Hessian of Bending Energy:

1. `F = gradEb(xkm1, ykm1, xk, yk, xkp1, ykp1, curvature0, l_k, EI)` takes eight inputs: $x_{k-1}, y_{k-1}, x_k, y_k, x_{k+1}, y_{k+1}$, κ_k^0, $\Delta l, EI$; and returns a vector with 6 elements that contains the following gradient:

$$\begin{bmatrix} \frac{\partial E_k^b}{\partial x_{k-1}} \\ \frac{\partial E_k^b}{\partial y_{k-1}} \\ \frac{\partial E_k^b}{\partial x_k} \\ \frac{\partial E_k^b}{\partial y_k} \\ \frac{\partial E_k^b}{\partial x_{k+1}} \\ \frac{\partial E_k^b}{\partial y_{k+1}} \end{bmatrix} \tag{13}$$

   Derivative of E_k^b with respect to any other degree of freedom is zero. In case of a naturally straight rod (e.g. the examples in Chapter 4), the natural curvature κ_k^0 is zero.

2. `J = hessEb(xkm1, ykm1, xk, yk, xkp1, ykp1, curvature0, l_k, EI)` takes eight inputs: $x_{k-1}, y_{k-1}, x_k, y_k, x_{k+1}, y_{k+1}, \Delta l, \kappa_k^0, EI$; and returns a matrix with 6×6 elements that contains the following Hessian:

$$\begin{bmatrix} \frac{\partial^2 E_k^b}{\partial x_{k-1} \partial x_{k-1}} & \frac{\partial^2 E_k^b}{\partial y_{k-1} \partial x_{k-1}} & \frac{\partial^2 E_k^b}{\partial x_k \partial x_{k-1}} & \frac{\partial^2 E_k^b}{\partial y_k \partial x_{k-1}} & \frac{\partial^2 E_k^b}{\partial x_{k+1} \partial x_{k-1}} & \frac{\partial^2 E_k^b}{\partial y_{k+1} \partial x_{k-1}} \\ \frac{\partial^2 E_k^b}{\partial x_{k-1} \partial y_{k-1}} & \frac{\partial^2 E_k^b}{\partial y_{k-1} \partial y_{k-1}} & \frac{\partial^2 E_k^b}{\partial x_k \partial y_{k-1}} & \frac{\partial^2 E_k^b}{\partial y_k \partial y_{k-1}} & \frac{\partial^2 E_k^b}{\partial x_{k+1} \partial y_{k-1}} & \frac{\partial^2 E_k^b}{\partial y_{k+1} \partial y_{k-1}} \\ \frac{\partial^2 E_k^b}{\partial x_{k-1} \partial x_k} & \frac{\partial^2 E_k^b}{\partial y_{k-1} \partial x_k} & \frac{\partial^2 E_k^b}{\partial x_k \partial x_k} & \frac{\partial^2 E_k^b}{\partial y_k \partial x_k} & \frac{\partial^2 E_k^b}{\partial x_{k+1} \partial x_k} & \frac{\partial^2 E_k^b}{\partial y_{k+1} \partial x_k} \\ \frac{\partial^2 E_k^b}{\partial x_{k-1} \partial y_k} & \frac{\partial^2 E_k^b}{\partial y_{k-1} \partial y_k} & \frac{\partial^2 E_k^b}{\partial x_k \partial y_k} & \frac{\partial^2 E_k^b}{\partial y_k \partial y_k} & \frac{\partial^2 E_k^b}{\partial x_{k+1} \partial y_k} & \frac{\partial^2 E_k^b}{\partial y_{k+1} \partial y_k} \\ \frac{\partial^2 E_k^b}{\partial x_{k-1} \partial x_{k+1}} & \frac{\partial^2 E_k^b}{\partial y_{k-1} \partial x_{k+1}} & \frac{\partial^2 E_k^b}{\partial x_k \partial x_{k+1}} & \frac{\partial^2 E_k^b}{\partial y_k \partial x_{k+1}} & \frac{\partial^2 E_k^b}{\partial x_{k+1} \partial x_{k+1}} & \frac{\partial^2 E_k^b}{\partial y_{k+1} \partial x_{k+1}} \\ \frac{\partial^2 E_k^b}{\partial x_{k-1} \partial y_{k+1}} & \frac{\partial^2 E_k^b}{\partial y_{k-1} \partial y_{k+1}} & \frac{\partial^2 E_k^b}{\partial x_k \partial y_{k+1}} & \frac{\partial^2 E_k^b}{\partial y_k \partial y_{k+1}} & \frac{\partial^2 E_k^b}{\partial x_{k+1} \partial y_{k+1}} & \frac{\partial^2 E_k^b}{\partial y_{k+1} \partial y_{k+1}} \end{bmatrix} \tag{14}$$

3. A utility function is provided to compute the discrete curvature in 2D setting: `kappa = computekappa2D(xkm1, ykm1, xk, yk, xkp1, ykp1 )` takes six inputs: $x_{k-1}, y_{k-1}, x_k, y_k, x_{k+1}, y_{k+1}$. The output `kappa` is a scalar representing the discrete integrated curvature κ_k at node $[x_k,\ y_k]$.

B MATLAB Code for Force and Jacobian in 3D (Rod)

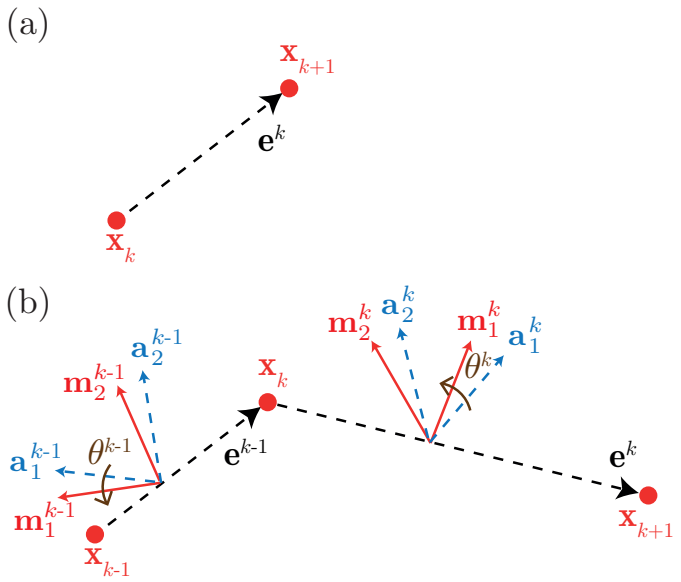

Figure 17: (a) Stretching and (b) bending and twisting energies.

Gradient and Hessian of Stretching Energy: The energy associated with the stretching of the edge $\mathbf{e}^k = \mathbf{x}_{k+1} - \mathbf{x}_k$ depends on six DOFs: $\mathbf{q}_{\text{local}} = [\mathbf{x}_k,\ \mathbf{x}_{k+1}]^T$.

1. `[dF, dJ] = gradEs_hessEs(node0, node1, l_k, EA)` takes four inputs: `node0` is a 1×3 vector representing $\mathbf{x}_k$, `node1` is a 1×3 vector representing $\mathbf{x}_{k+1}$, `l_k` is the undeformed length of the edge $|\bar{\mathbf{e}^k}|$, and `EA` is the stretching stiffness (Young's modulus times cross-sectional area).

2. `dF` is a 6×1 vector representing the gradient of the energy: $\frac{\partial E_k^s}{\partial (q_{\text{local}})_i}$, with $i = 1, \ldots, 6$ and
$$E_k^s = \frac{1}{2} EA \left(\frac{|\mathbf{x}_{k+1} - \mathbf{x}_k|}{|\bar{\mathbf{e}^k}|} - 1 \right)^2 |\bar{\mathbf{e}^k}| \tag{15}$$
is the stretching energy associated with that edge.

3. `dJ` is a 6×6 symmetric matrix representing the Hessian of the energy: $\frac{\partial^2 E_k^s}{\partial (q_{\text{local}})_i \partial (q_{\text{local}})_j}$, with $i = 1, \ldots, 6$ and $j = 1, \ldots, 6$.

Gradient and Hessian of Bending Energy: The energy associated with the bending at node $\mathbf{x}_k$ (due to the turning of the edge $\mathbf{e}^{k-1} = \mathbf{x}_k - \mathbf{x}_{k-1}$ to $\mathbf{e}^k = \mathbf{x}_{k+1} - \mathbf{x}_k$) depends on eleven DOFs: $\mathbf{q}_{\text{local}} = \left[\mathbf{x}_{k-1},\ \theta^{k-1},\ \mathbf{x}_k,\ \theta^k,\ \mathbf{x}_{k+1}\right]^T$.

1. `[dF, dJ] = gradEb_hessEb(node0, node1, node2, m1e, m2e, m1f, m2f, kappaBar, l_k, EI)` takes ten inputs:
   `node0` is a 1×3 vector representing $\mathbf{x}_{k-1}$,
   `node1` is a 1×3 vector representing $\mathbf{x}_k$,
   `node2` is a 1×3 vector representing $\mathbf{x}_{k+1}$,
   `m1e` is a 1×3 vector representing $\mathbf{m}_1^{k-1}$,
   `m2e` is a 1×3 vector representing $\mathbf{m}_2^{k-1}$,
   `m1f` is a 1×3 vector representing $\mathbf{m}_1^{k}$,
   `m2f` is a 1×3 vector representing $\mathbf{m}_1^{k}$,
   `kappaBar` is a 1×2 vector representing the natural curvature $\boldsymbol{\kappa}_k^0$ (the discrete integrated version) at node $\mathbf{x}_k$,
   `l_k` is the Voronoi length $\bar{l}_k = \left(|\bar{\mathbf{e}^{k-1}}| + |\bar{\mathbf{e}^{k}}|\right)/2$ and $\bar{(\ )}$ represents evaluation in undeformed state, and
   `EI` is the bending stiffness (Young's modulus times area moment of inertia).

2. `dF` is a 11×1 vector representing the gradient of the energy: $\frac{\partial E_k^b}{\partial (q_{\text{local}})_i}$, with $i = 1, \ldots, 11$ and
$$E_k^b = \frac{1}{2} EI \left(|\boldsymbol{\kappa}_k - \boldsymbol{\kappa}_k^0|\right)^2 \frac{1}{\bar{l}_k} \tag{16}$$
   is the bending energy associated with the node $\mathbf{x}_k$.

3. `dJ` is a 11×11 symmetric matrix representing the Hessian of the energy: $\frac{\partial^2 E_k^b}{\partial (q_{\text{local}})_i \partial (q_{\text{local}})_j}$, with $i = 1, \ldots, 11$ and $j = 1, \ldots, 11$.

4. A utility function is provided to compute the curvature at the nodes: `kappa = computekappa(node0, node1, node2, m1e, m2e, m1f, m2f )` takes ten inputs:
   `node0` is a 1×3 vector representing $\mathbf{x}_{k-1}$,
   `node1` is a 1×3 vector representing $\mathbf{x}_k$,
   `node2` is a 1×3 vector representing $\mathbf{x}_{k+1}$,
   `m1e` is a 1×3 vector representing $\mathbf{m}_1^{k-1}$,
   `m2e` is a 1×3 vector representing $\mathbf{m}_2^{k-1}$,
   `m1f` is a 1×3 vector representing $\mathbf{m}_1^{k}$, and
   `m2f` is a 1×3 vector representing $\mathbf{m}_1^{k}$.
   The output `kappa` is a 1×2 vector representing the discrete integrated curvature $\boldsymbol{\kappa}_k$ at node $\mathbf{x}_k$.

Gradient and Hessian of Twisting Energy: The energy associated with the twisting at node $\mathbf{x}_k$ (due to the turning of the material frames from edge $\mathbf{e}^{k-1}$ to $\mathbf{e}^k$) depends on eleven DOFs: $\mathbf{q}_{\text{local}} = \left[\mathbf{x}_{k-1},\ \theta^{k-1},\ \mathbf{x}_k,\ \theta^k,\ \mathbf{x}_{k+1}\right]^T$.

1. `[dF, dJ] = gradEt_hessEt(node0, node1, node2, theta_e, theta_f, refTwist, l_k, GJ)` takes eight inputs:
   `node0` is a 1×3 vector representing $\mathbf{x}_{k-1}$,
   `node1` is a 1×3 vector representing $\mathbf{x}_k$,

`node2` is a 1×3 vector representing $\mathbf{x}_{k+1}$,
`theta_e` is a scalar representing the twist angle θ^{k-1},
`theta_f` is a scalar representing the twist angle θ^{k},
`refTwist` is a scalar representing the discrete reference twist, $\Delta m_{k,\text{ref}}$ at $\mathbf{x}_k$,
`l_k` is the Voronoi length $\bar{l}_k = \left(|\bar{\mathbf{e}^{k-1}}| + |\bar{\mathbf{e}^{k}}|\right)/2$ and $\bar{(\ )}$ represents evaluation in undeformed state, and
`GJ` is the twisting stiffness (shear modulus times polar moment of inertia).

2. `dF` is a 11×1 vector representing the gradient of the energy: $\frac{\partial E_k^t}{\partial (q_{\text{local}})_i}$, with $i = 1, \ldots, 11$ and

$$E_k^t = \frac{1}{2} GJ \tau_k^2 \frac{1}{\bar{l}_k} \tag{17}$$

is the twisting energy associated with the node $\mathbf{x}_k$.

3. `dJ` is a 11×11 symmetric matrix representing the Hessian of the energy: $\frac{\partial^2 E_k^t}{\partial (q_{\text{local}})_i \partial (q_{\text{local}})_j}$, with $i = 1, \ldots, 11$ and $j = 1, \ldots, 11$.

C MATLAB Code for Force and Jacobian of Shells

Gradient and Hessian of Stretching Energy: The energy associated with the stretching of the edge $\mathbf{e}^k = \mathbf{x}_1 - \mathbf{x}_0$ depends on six DOFs: $\mathbf{q}_{\text{local}} = [\mathbf{x}_0,\ \mathbf{x}_1]^T$.

1. `[dF, dJ] = gradEs_hessEs_Shell(node0, node1, l_k, ks)` takes four inputs: `node0` is a 1×3 vector representing $\mathbf{x}_0$, `node1` is a 1×3 vector representing $\mathbf{x}_1$, `l_k` is the undeformed length of the edge $\mathbf{e}^k$, and `ks` is the stretching stiffness in Eq. 8.3.

2. `dF` is a 6×1 vector representing the gradient of the energy: $\frac{\partial E_s^k}{\partial (q_{\text{local}})_i}$, with $i = 1, \ldots, 6$. Eq. 8.4 provides the expression for the stretching energy.

3. `dJ` is a 6×6 symmetric matrix representing the Hessian of the energy: $\frac{\partial^2 E_s^k}{\partial (q_{\text{local}})_i \partial (q_{\text{local}})_j}$, with $i = 1, \ldots, 6$ and $j = 1, \ldots, 6$.

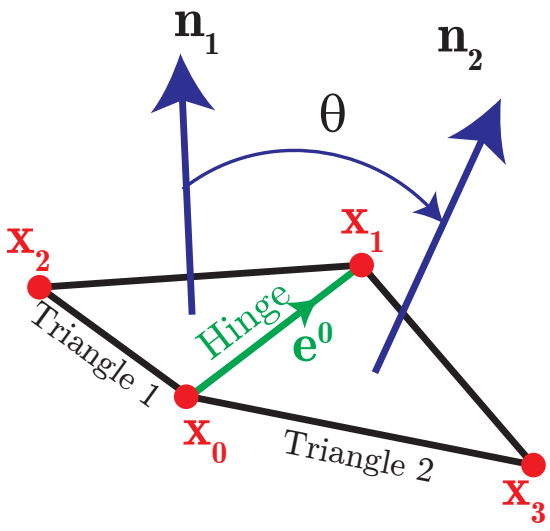

Figure 18: Bending angle.

Gradient and Hessian of Bending Energy: The energy associated with the bending at a hinge $\mathbf{x}_1 - \mathbf{x}_0$ depends on twelve DOFs: $\mathbf{q}_{\text{local}} = [\mathbf{x}_0,\ \mathbf{x}_1,\ \mathbf{x}_2,\ \mathbf{x}_3]^T$.

1. `[dF, dJ] = gradEb_hessEb_Shell(node0, node1, node2, node3, thetaBar, kb)` takes six inputs:
   `node0` is a 1×3 vector representing $\mathbf{x}_0$,
   `node1` is a 1×3 vector representing $\mathbf{x}_1$,
   `node2` is a 1×3 vector representing $\mathbf{x}_2$,
   `node3` is a 1×3 vector representing $\mathbf{x}_3$,
   `thetaBar` is a scalar representing the undeformed bend angle $\bar{\theta}^k$, and
   `kb` is the bending stiffness in Eq. 8.3.

2. `dF` is a 12×1 vector representing the gradient of the energy: $\frac{\partial E_b^k}{\partial (q_{\text{local}})_i}$, with $i = 1, \ldots, 12$. Eq. 8.4 provides an expression for the bending energy at each hinge.

3. `dJ` is a 12×12 symmetric matrix representing the Hessian of the energy: $\frac{\partial^2 E_b^k}{\partial (q_{\text{local}})_i \partial (q_{\text{local}})_j}$, with $i = 1, \ldots, 12$ and $j = 1, \ldots, 12$.

4. A utility function is provided to compute the bend angle at the hinge: `theta = getTheta(node0, node1, node2, node3)` takes four inputs:
   `node0` is a 1×3 vector representing $\mathbf{x}_0$,
   `node1` is a 1×3 vector representing $\mathbf{x}_1$,
   `node2` is a 1×3 vector representing $\mathbf{x}_2$, and
   `node3` is a 1×3 vector representing $\mathbf{x}_3$.
   The output `theta` (scalar variable) is the signed bend angle at the hinge.

5. `[dF, dJ] = gradEb_hessEb_Shell(node0, node1, node2, node3, thetaBar, kb)` requires two additional functions `gradTheta` and `hessTheta`, which are also provided with these notes.

Bibliography

[1] G. Kirchhoff, "Ueber das gleichgewicht und die bewegung eines unendlich dünnen elastischen stabes.," *J. Reine Angew Math*, vol. 56, pp. 285–313, 1859.

[2] M. Bergou, M. Wardetzky, S. Robinson, B. Audoly, and E. Grinspun, "Discrete elastic rods," *ACM Trans Graph*, vol. 27, no. 3, p. 63, 2008.

[3] M. Bergou, B. Audoly, E. Vouga, M. Wardetzky, and E. Grinspun, "Discrete viscous threads," *ACM Trans Graph*, vol. 29, no. 4, p. 116, 2010.

[4] M. K. Jawed, F. Da, J. Joo, E. Grinspun, and P. M. Reis, "Coiling of elastic rods on rigid substrates," *Proc. Natl. Acad. Sci.*, vol. 111, no. 41, pp. 14663–14668, 2014.

[5] M. K. Jawed, A. Novelia, and O. O'Reilly, *A primer on the kinematics of Discrete Elastic Rods*. Springer-Verlag, 2018 (in press).

[6] D. Baraff and A. Witkin, "Large steps in cloth simulation," in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 43–54, ACM, 1998.

[7] E. Grinspun, A. N. Hirani, M. Desbrun, and P. Schröder, "Discrete shells," in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pp. 62–67, Eurographics Association, 2003.

[8] Z. Shen, J. Huang, W. Chen, and H. Bao, "Geometrically exact simulation of inextensible ribbon," in *Computer Graphics Forum*, vol. 34, pp. 145–154, Wiley Online Library, 2015.

[9] B. Audoly and Y. Pomeau, *Elasticity and geometry*. Oxford Univ. Press, 2010.

[10] T. Savin, N. A. Kurpios, A. E. Shyer, P. Florescu, H. Liang, L. Mahadevan, and C. J. Tabin, "On the growth and form of the gut," *Nature*, vol. 476, no. 7358, pp. 57–62, 2011.

[11] H. S. Seung and D. R. Nelson, "Defects in flexible membranes with crystalline order," *Physical Review A*, vol. 38, no. 2, p. 1005, 1988.

[12] H. Liang and L. Mahadevan, "The shape of a long leaf," *Proceedings of the National Academy of Sciences*, vol. 106, no. 52, pp. 22049–22054, 2009.

[13] R. Tamstorf and E. Grinspun, "Discrete bending forces and their jacobians," *Graphical models*, vol. 75, no. 6, pp. 362–370, 2013.

[14] D. Kourounis, A. Fuchs, and O. Schenk, "Toward the next generation of multiperiod optimal power flow solvers," *IEEE Transactions on Power Systems*, vol. 33, no. 4, pp. 4005–4014, 2018.

[15] F. Verbosio, A. De Coninck, D. Kourounis, and O. Schenk, "Enhancing the scalability of selected inversion factorization algorithms in genomic prediction," *Journal of Computational Science*, vol. 22, pp. 99–108, 2017.

[16] A. De Coninck, B. De Baets, D. Kourounis, F. Verbosio, O. Schenk, S. Maenhout, and J. Fostier, "Needles: toward large-scale genomic prediction with marker-by-environment interaction," *Genetics*, vol. 203, no. 1, pp. 543–555, 2016.

[17] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, "Neural ordinary differential equations," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, (Red Hook, NY, USA), p. 6572–6583, Curran Associates Inc., 2018.

[18] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[19] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[21] F. L. et. al, "Lecture notes in deep learning for computer vision," Spring 2022.

[22] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[23] A. G. Baydin, B. A. Pearlmutter, and A. A. Radul, "Automatic differentiation in machine learning: a survey," *CoRR*, vol. abs/1502.05767, 2015.

[24] Q. Li, T. Wang, V. Roychowdhury, and M. K. Jawed, "Rapidly encoding generalizable dynamics in a euclidean symmetric neural network: a slinky case study," *arXiv preprint arXiv:2203.11546*, 2022.