Franklin W. Olin College of Engineering

MTH3199 Applied Math for Engineers – Fall 2025
Assignment 5: Vibration Modes

Assigned: Tuesday, November 4, 2025.
Lab Report Due: Monday, November 17, 2025 (11:59 PM EST).

Overview

Up to this point, we have spent our time developing and testing numerical tools to solve the following three problems:

- Root-finding (Newton's method, bisection method, and secant method)
- Differentiation (using finite-differences to approximate the Jacobian of a function)
- Integration (Runge-Kutta and friends)

In this module, we will apply the tools that we have developed to analyze a nonlinear oscillator (a vibrating box). Roughly, our agenda for this assignment is as follows:

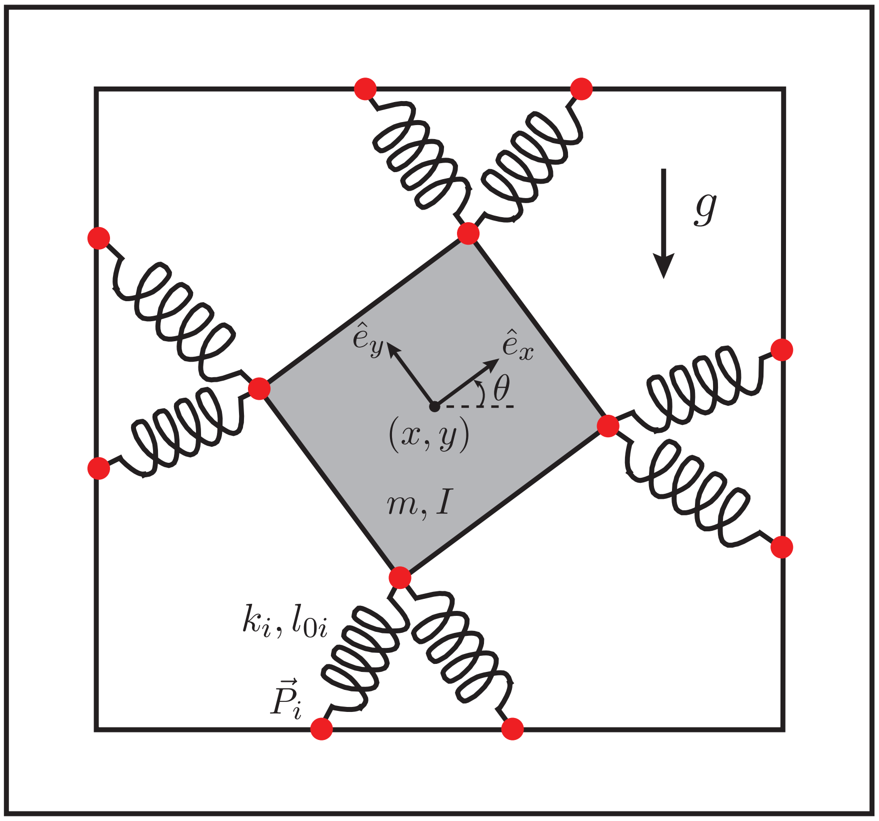
- Apply momentum principles to derive the equations of motion for the system.
- Implement the differential equation in MATLAB
- Numerically integrate the equation of motion using the Runge-Kutta integrator you created in the previous assignment
- Use the multidimensional Newton solver that you created to identify equilibrium states of the system.
- Use the numerical differentiator that you created to compute a linear approximation for the system near an equilibrium state.
- Compute vibration modes and frequencies for the system.
- Compare the behavior of the linear approximation to that of the original nonlinear system.

Overall, this should be a fun example in which we get to capitalize on the tools we've spent this semester to develop, while learning a few additional concepts along the way!

Vibrating Box Model

We begin with a model of our system. Consider a box that is suspended in midair via some number of springs (which can be seen on the next page). We will describe this system using the following variables and parameters:

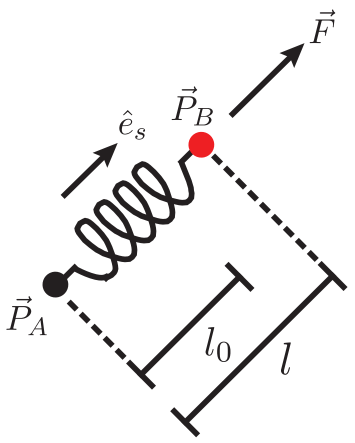
- (x, y, θ) are generalized coordinates that describe the position of the center of mass and the orientation of the box respectively.
- The box has mass m and moment of inertia I (w/respect to its centroid).
- The ith spring has stiffness k_i and natural length l_{0i}.
- The boxy-fixed unit vectors $(\hat{e}_x, \hat{e}_y)$ are aligned with the box's symmetry axes.
- The position vector $\vec{P}_i$ describes one of the mounting point (in the world frame) used by a spring.
- Gravity, g, acts in the vertical direction.



Spring Behavior

We'll begin the derivation of our equations of motion by looking at the behavior of a single spring and then build from there. A spring is parameterized by two quantities: its stiffness, k, and its natural length, l_0:

- The stiffness, k, is the proportionality constant that relates the its length displacement, $|\Delta l| = |l - l_0|$, to the magnitude of the force it exerts, $|\vec{F}|$.
- The natural length, l_0 is the length for which the spring exerts zero force. In other words, it is the length that the spring would be if it was not attached to anything.



$$\vec{F} = -k(l - l_0)\hat{e}_s$$

Let $\vec{P}_A$ and $\vec{P}_B$ be the positions of the two ends of the spring, and $\hat{e}_s$ be the unit vector pointing from A to B:

$$\hat{e}_s = \frac{\vec{P}_B - \vec{P}_A}{l} = \frac{\vec{P}_B - \vec{P}_A}{|\vec{P}_B - \vec{P}_A|} \tag{1}$$

In this configuration, the force exerted by the spring at point B is given by the equation:

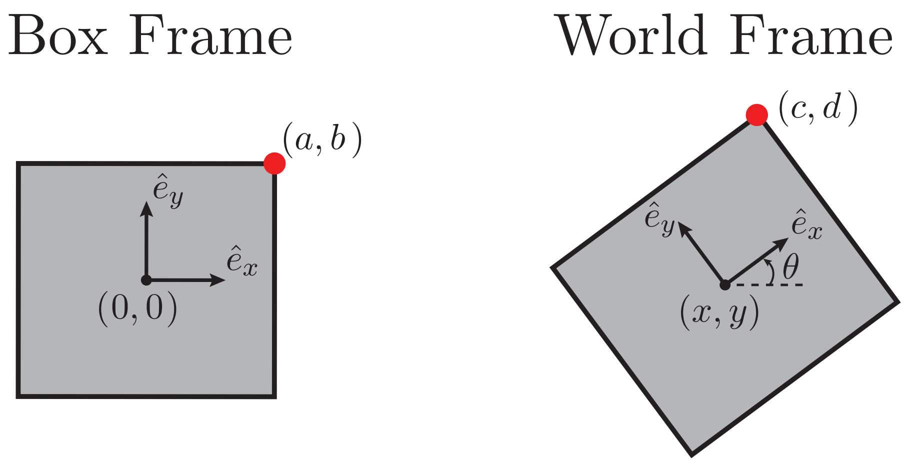
$$\vec{F} = -k(l - l_0)\hat{e}_s \tag{2}$$

where $l = |\vec{P}_B - \vec{P}_A|$ is the current length of the spring. Your first task of this assignment is to implement this behavior as a MATLAB function. A template has been provided below for you to fill out:

```
%computes the force exerted by the spring at one of its ends
%INPUTS:
%PA: the position of the first end of the spring
%PB: the position of the second end of the spring
%k: the stiffness of the spring
%l0: the natural length of the spring
%OUTPUTS:
%F: the force exerted by the spring at end B
function F = compute_spring_force(k,l0,PA,PB)
    %current length of the spring
    l = %your code here
    %unit vector pointing from PA to PB
    e_s = %your code here

    %Force exerted by spring at point B
    F = %your code here
end
```

Box Kinematics

Given the box's current position and orientation, (x, y, θ), we would like to compute the force that is exerted by each of the springs. Assuming that we are given the stiffness and natural length of each spring as part of the problem description, we still need to know the world-frame positions of the spring ends, $(\vec{P}_A, \vec{P}_B)$. One end of the spring will be attached to a static point in the environment, so its world-frame position will be known. However, the other end of the spring will be attached to a fixed point on the box, meaning that its position depends on the coords (x, y, θ).



Let (a, b) and (c, d) be the box-frame and world-frame coordinates of one of the mounting points of the spring. The coordinates are related to one another via a rigid body transformation (in other words, a translation and rotation):

$$\begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} \cos(\theta), & -\sin(\theta) \\ \sin(\theta), & \cos(\theta) \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix} \tag{3}$$

To see why this is the case, note that the body-fixed unit vectors $\hat{e}_x$ and $\hat{e}_y$ are given by the world-frame coordinates:

$$\hat{e}_x = \begin{bmatrix}\cos(\theta)\\ \sin(\theta)\end{bmatrix}, \quad \hat{e}_y = \begin{bmatrix}-\sin(\theta)\\ \cos(\theta)\end{bmatrix} \tag{4}$$

To get from the centroid of the box to the mounting point, we must travel length a in the $\hat{e}_x$ direction, and length b in the $\hat{e}_y$ direction. From this, we see that:

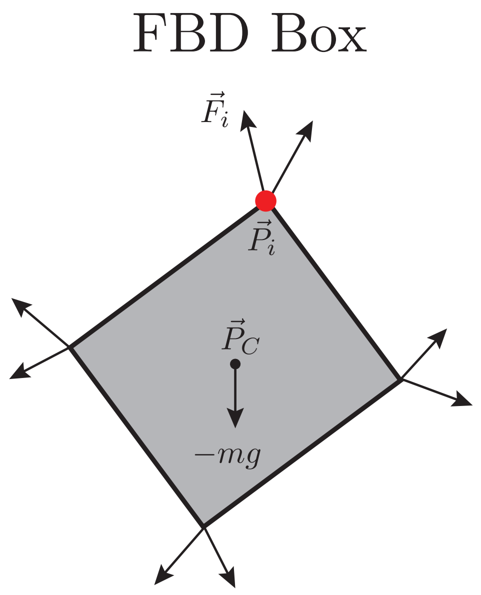
$$\begin{bmatrix}c\\ d\end{bmatrix} - \begin{bmatrix}x\\ y\end{bmatrix} = a\hat{e}_x + b\hat{e}_y = a\begin{bmatrix}\cos(\theta)\\ \sin(\theta)\end{bmatrix} + b\begin{bmatrix}-\sin(\theta)\\ \cos(\theta)\end{bmatrix} = \begin{bmatrix}\cos(\theta), & -\sin(\theta)\\ \sin(\theta), & \cos(\theta)\end{bmatrix}\begin{bmatrix}a\\ b\end{bmatrix} \tag{5}$$

which can be rearranged to get equation 4. Implement this relationship as a MATLAB function by filling out the template provided below:

```
%Computes the rigid body transformation that maps a set
%of points in the box-frame to their corresponding
%world-frame coordinates
%INPUTS:
%x: the x position of the centroid of the box
%y: the y position of the centroid of the box
%theta: the orientation of the box
%Plist_box: a 2 x n matrix of points in the box frame
%OUTPUTS:
%Plist_world: a 2 x n matrix of points describing
%the world-frame coordinates of the points in Plist_box
function Plist_world = compute_rbt(x,y,theta,Plist_box)
    % your code here
end
```

Momentum Principles

Now that we have enough information to compute the forces exerted by each of the springs, we can apply momentum principles to figure out the linear and angular acceleration of the box. To do this, lets first take a look at the free body diagram of the system:



Here, we see that the ith spring exerts force $\vec{F}_i$ at mounting point $\vec{P}_i$ (represented in the world-frame). Gravity exerts a force of $-mg$ in the vertical direction, and acts at the center of mass, $\vec{P}_C = (x, y)^T$ of the box. The net force and torque (w/respect to the centroid) are thus given by:

$$\sum \vec{F}_{external} = \begin{bmatrix}0\\ -mg\end{bmatrix} + \sum_{i=1}^{n} \vec{F}_i \tag{6}$$

$$\sum \vec{\tau}_{C,external} = (\vec{P}_C - \vec{P}_C) \times \begin{bmatrix} 0 \\ -mg \end{bmatrix} + \sum_{i=1}^{n} (\vec{P}_i - \vec{P}_C) \times \vec{F}_i \tag{7}$$

$$\sum \vec{\tau}_{C,external} = \sum_{i=1}^{n} (\vec{P}_i - \vec{P}_C) \times \vec{F}_i \tag{8}$$

Applying the linear momentum principle (Newton's second law), we get:

$$\frac{d}{dt}\left(m\vec{V}_C\right) = \sum \vec{F}_{external}, \quad \rightarrow \quad m\begin{bmatrix} \ddot{x} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} 0 \\ -mg \end{bmatrix} + \sum_{i=1}^{n} \vec{F}_i \tag{9}$$

Since we are tracking the center of mass of the box, the angular momentum principle reduces to $\tau = I\ddot{\theta}$:

$$I\ddot{\theta} = \sum \vec{\tau}_{C,external}, \quad \rightarrow \quad I\ddot{\theta} = \sum_{i=1}^{n} (\vec{P}_i - \vec{P}_C) \times \vec{F}_i \tag{10}$$

With this, we can now compute linear and angular acceleration of the box for any position and orientation. Time to implement it in MATLAB!

```
%Computes the linear and angular acceleration of the box
%given its current position and orientation
%INPUTS:
%x: current x position of the box
%y: current y position of the box
%theta: current orientation of the box
%box_params: a struct containing the parameters that describe the system
%Fields:
%box_params.m: mass of the box
%box_params.I: moment of inertia w/respect to centroid
%box_params.g: acceleration due to gravity
%box_params.k_list: list of spring stiffnesses
%box_params.l0_list: list of spring natural lengths
%box_params.P_world: 2 x n list of static mounting
%   points for the spring (in the world frame)
%box_params.P_box: 2 x n list of mounting points
%    for the spring (in the box frame)
%
%OUTPUTS
%ax: x acceleration of the box
%ay: y acceleration of the box
%atheta: angular acceleration of the box
function [ax,ay,atheta] = compute_accel(x,y,theta,box_params)
    %your code here
end
```

Numerical Simulation of the Nonlinear System

At this point, you have written a function that computes the linear and angular acceleration of the box for any given position and orientation. In order to use the numerical integrators that you have written, we must first convert this to a vector-valued 1st-order ODE. To do this, we will use the vector V to represent the state of the system:

$$V = \begin{bmatrix} x \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} \tag{11}$$

The time derivative of our state vector is given by:

$$\dot{V} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \ddot{x} \\ \ddot{y} \\ \ddot{\theta} \end{bmatrix} \tag{12}$$

Our numerical integrators need to take in a rate function, $\dot{V} = f(t, V)$ as input. Implement this rate function in MATLAB:

```
%Rate function encoding the  nonlinear equations
%of motion for the vibrating box.
%INPUTS:
%t: the current time
%V = [x;y;theta;dxdt;dydt;dthetadt]: state vector
%box_params: a struct containing the parameters that describe the system
%Fields:
%box_params.m: mass of the box
%box_params.I: moment of inertia w/respect to centroid
%box_params.g: acceleration due to gravity
%box_params.k_list: list of spring stiffnesses
%box_params.l0_list: list of spring natural lengths
%box_params.P_world: 2 x n list of static mounting
%   points for the spring (in the world frame)
%box_params.P_box: 2 x n list of mounting points
%    for the spring (in the box frame)
%
%OUTPUTS
%dVdt = [dxdt;dydt;dthetadt;d2xdt2;d2ydt2;d2thetadt2]:
%   the time derivative of the state vector
function dVdt = box_rate_func(t,V,box_params)
    %your code here
end
```

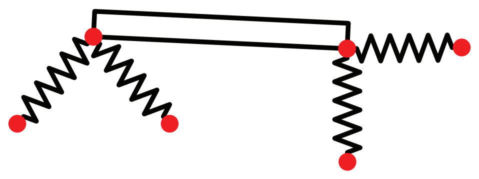
You now have all the necessary pieces to simulate the system. Use your numerical integrator to simulate the vibrating box for a set of system parameter values and initial conditions of your choice. The template below can be used for initializing the simulation

```
function simulate_box()
    %define system parameters
    box_params = struct();
    box_params.m = %your code here
    box_params.I = %your code here
    box_params.g = %your code here
    box_params.k_list = %your code here
    box_params.l0_list = %your code here
    box_params.P_world = %your code here
    box_params.P_box = %your code here

    %load the system parameters into the rate function
    %via an anonymous function
    my_rate_func = @(t_in,V_in) box_rate_func(t_in,V_in,box_params);

    x0 = %your code here
    y0 = %your code here
    theta0 = %your code here
    vx0 = %your code here
    vy0 = %your code here
    vtheta0 = %your code here
    V0 = [x0;y0;theta0;vx0;vy0;vtheta0];
    tspan = %your code here

    %run the integration
    % [tlist,Vlist] = your_integrator(my_rate_func,tspan,V0,...)

end
```

Create an animation of the system to validate your simulation. To help you get started, some example code that will generate plots of the springs has been provided below:

```
function spring_plotting_example()

    num_zigs = 5;
    w = .1;

    hold on;
    spring_plot_struct = initialize_spring_plot(num_zigs,w);

    axis equal; axis square;
    axis([-3,3,-3,3]);
    for theta=linspace(0,6*pi,1000)
        P1 = [.5;.5];
        P2 = 2*[cos(theta);sin(theta)];
        update_spring_plot(spring_plot_struct,P1,P2)
        drawnow;
    end


end

%updates spring plotting object so that spring is plotted
%with ends located at points P1 and P2
function update_spring_plot(spring_plot_struct,P1,P2)
    dP = P2-P1;
    R = [dP(1),-dP(2)/norm(dP);dP(2),dP(1)/norm(dP)];
    plot_pts = R*spring_plot_struct.zig_zag;

    set(spring_plot_struct.line_plot,...
        'xdata',plot_pts(1,:)+P1(1),...
        'ydata',plot_pts(2,:)+P1(2));

    set(spring_plot_struct.point_plot,...
    'xdata',[P1(1),P2(1)],...
    'ydata',[P1(2),P2(2)]);
end

%create a struct containing plotting info for a single spring
%INPUTS:
%num_zigs: number of zig zags in spring drawing
%w: width of the spring drawing
function spring_plot_struct = initialize_spring_plot(num_zigs,w)
    spring_plot_struct = struct();

    zig_ending = [.25,.75,1; ...
                 -1,1,0];

    zig_zag = zeros(2,3+3*num_zigs);
    zig_zag(:,1) = [-.5;0];
    zig_zag(:,end) = [num_zigs+.5;0];

    for n = 0:(num_zigs-1)
        zig_zag(:,(3+3*n):2+3*(n+1)) = zig_ending + [n,n,n;0,0,0];
    end
    zig_zag(1,:)=(zig_zag(1,:)-zig_zag(1,1))/(zig_zag(1,end)-zig_zag(1,1));
    zig_zag(2,:)=zig_zag(2,:)*w;

    spring_plot_struct.zig_zag = zig_zag;
    spring_plot_struct.line_plot = plot(0,0,'k','linewidth',2);
    spring_plot_struct.point_plot = plot(0,0,'ro','markerfacecolor','r','markersize',7);
end
```

If you need something to test your system with, an example set of system parameters has been provided below (along with a visualization of the system):



```
LW = 10; LH = 1; LG = 3;
m = 1; Ic = (1/12)*(LH^2+LW^2);

g = 1; k = 20; k_list = [.5*k,.5*k,2*k,5*k];

l0 = 1.5*LG;

Pbl_box = [-LW;-LH]/2;
Pbr_box = [LW;-LH]/2;
Ptl_box = [-LW;LH]/2;
Ptr_box = [LW;LH]/2;

boundary_pts = [Pbl_box,Pbr_box,Ptr_box,Ptl_box,Pbl_box];

Pbl1_world = Pbl_box + [-LG;-LG];
Pbl2_world = Pbl_box + [LG;-LG];

Pbr1_world = Pbr_box + [0;-l0];
Pbr2_world = Pbr_box + [l0;0];

P_world = [Pbl1_world,Pbl2_world,Pbr1_world,Pbr2_world];
P_box = [Pbl_box,Pbl_box,Pbr_box,Pbr_box];

%define system parameters
box_params = struct();
box_params.m = m;
box_params.I = Ic;
box_params.g = g;
box_params.k_list = k_list;
box_params.l0_list = l0*ones(size(P_world,2));
box_params.P_world = P_world;
box_params.P_box = P_box;
box_params.boundary_pts = boundary_pts;
```

Linearization and Modal Analysis

Finding Equilibrium States

We would like to analyze the behavior of our system near an equilibrium configuration. To do this, we must first locate an equilibrium configuration. Fortunately, you have already written the perfect numerical tool for the job: your multi-Newton solver!

At an equilibrium configuration, V_{eq}, the system remains at rest ($\dot{V} = 0$). If we plug this into the rate function that encodes the motion equations for our system, we get:

$$0 = \dot{V} = f(t, V_{eq}) \tag{13}$$

Note that the equations of motion for the unforced mass-spring system are **time-invariant** (they do not depend on time), so we can remove the time variable from the rate function equation:

$$0 = f(V_{eq}) \tag{14}$$

In other words, the equilibrium state, V_{eq}, is the root of the rate function. As such, your task for this section is to use your multi-Newton solver to find an equilibrium state for this system:

$$V_{eq} = \begin{bmatrix} x_{eq} \\ y_{eq} \\ \theta_{eq} \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{15}$$

Note that, depending on your choice of parameters, the system may have more than one equilibrium configuration. Verify that the state vector you found is actually the equilibrium by running a simulation using $V(t=0) = V_{eq}$ as your initial condition. You should find that the system will remain at rest if it starts at the equilibrium point.

Linearization

Let us now consider some small perturbation, $\Delta V(t)$ from the equilibrium state, V_{eq}. In other words:

$$\Delta V(t) = V(t) - V_{eq}, \quad \rightarrow \quad V(t) = \Delta V(t) + V_{eq} \tag{16}$$

Since the equilibrium configuration, V_{eq}, is a constant, we find that if we take the derivative of both sides of the equation, we get:

$$\Delta\dot{V} = \dot{V}, \qquad \Delta\ddot{V} = \ddot{V} \tag{17}$$

Plugging all of this into our rate function relationship (and removing t as in input to our rate function, since it is time-invariant), we get:

$$\Delta\dot{V} = f(V_{eq} + \Delta V) \tag{18}$$

We can approximate our rate function $f(V_{eq} + \Delta V)$ using its tangent approximation (i.e. its first-order Taylor expansion):

$$\Delta\dot{V} = f(V_{eq} + \Delta V) \approx f(V_{eq}) + J(V_{eq})\Delta V \tag{19}$$

where $J(V_{eq})$ is the **Jacobian** of our rate function, evaluated at the equilibrium state, V_{eq}. Since, V_{eq} is an equilibrium point, $f(V_{eq}) = 0$ (in fact, that is how we just solved for V_{eq} in the previous section!). Thus:

$$\Delta\dot{V} = f(V_{eq} + \Delta V) \approx 0 + J(V_{eq})\Delta V = J(V_{eq})\Delta V \tag{20}$$

$$\Delta\dot{V} \approx \underbrace{J(V_{eq})}_{A}\Delta V = A\Delta V, \qquad A = J(V_{eq}) \tag{21}$$

This is a 1st-order **linear** differential equation (our linearized system) that approximates the nonlinear system near the equilibrium state. To linearize the system, we need to compute the Jacobian of the rate function at the equilibrium configuration, $J(V_{eq})$. Fortunately, you already have written the perfect numerical tool for the job: the approximate Jacobian function (which you used as a building block for writing your multi-Newton solver)!

Let's now implement this linearization in MATLAB. If we plug back $\dot{V} = \Delta\dot{V}$ and $\Delta V = (V - V_{eq})$ into our linearized system, we get the following equation:

$$\dot{V} = J(V_{eq})\Big(V - V_{eq}\Big) = A\Big(V - V_{eq}\Big) \tag{22}$$

We can then compare the behavior of the linearized system with the original nonlinear system. Implement the linear rate function in MATLAB:

```
my_linear_rate = @(t_in,V_in) J_approx*(V_in-V_eq);
```

then, run both systems using initial conditions near the equilibrium configuration and compare their results:

```
dx0 = %your code here
dy0 = %your code here
dtheta0 = %your code here
vx0 = %your code here
vy0 = %your code here
vtheta0 = %your code here

%small number used to scale initial perturbation
epsilon = %your code here

V0 = Veq + epsilon*[dx0;dy0;dtheta0;vx0;vy0;vtheta0];
tspan = %your code here

%run the integration of nonlinear system
% [tlist_nonlinear,Vlist_nonlinear] =...
% your_integrator(my_rate_func,tspan,V0,...);

%run the integration of linear system
% [tlist_linear,Vlist_linear] =...
% your_integrator(my_linear_rate,tspan,V0,...);
```

Generate plots comparing the linear and nonlinear system for varying perturbation magnitudes (your vary the ϵ parameter in the template above to test different magnitudes).

Modal Analysis

Because this is an undamped 2nd-order system, you will find that the rate function Jacobian, A, has the form:

$$\begin{bmatrix}\Delta\dot{x}\\ \Delta\dot{y}\\ \Delta\dot{\theta}\\ \Delta\ddot{x}\\ \Delta\ddot{y}\\ \Delta\ddot{\theta}\end{bmatrix} \approx A \begin{bmatrix}\Delta x\\ \Delta y\\ \Delta\theta\\ \Delta\dot{x}\\ \Delta\dot{y}\\ \Delta\dot{\theta}\end{bmatrix}, \qquad A = \begin{bmatrix}\{0\} & I\\ -Q & \{0\}\end{bmatrix} \tag{23}$$

where $\{0\}$ is a 3×3 matrix of zeros, and I is the 3×3 identity matrix (why is this?). We can rewrite this linear approximation as the following 2nd-order differential equation:

$$\begin{bmatrix}\Delta\ddot{x}\\ \Delta\ddot{y}\\ \Delta\ddot{\theta}\end{bmatrix} = -Q\begin{bmatrix}\Delta x\\ \Delta y\\ \Delta\theta\end{bmatrix}, \quad \rightarrow \quad \begin{bmatrix}\Delta\ddot{x}\\ \Delta\ddot{y}\\ \Delta\ddot{\theta}\end{bmatrix} + Q\begin{bmatrix}\Delta x\\ \Delta y\\ \Delta\theta\end{bmatrix} = 0 \tag{24}$$

We can define $U = [\Delta x, \Delta y, \Delta\theta]^T$ as our 3×1 truncated state vector. Plugging this into our ODE, we get:

$$\ddot{U} + QU = 0 \tag{25}$$

Let's look for solutions of the form $U(t) = U_0 \cos(\omega_n t)$. Plugging this in, we get:

$$-\omega_n^2 \cos(\omega_n t) U_0 + Q U_0 \cos(\omega_n t) = 0 \tag{26}$$

Dividing by $\cos(\omega_n t)$ and factoring out U_0 gives us:

$$\left(Q - \omega_n^2 I\right) U_0 = 0 \tag{27}$$

where I is the 3×3 identity matrix. From this, we see that U_0 is an eigenvector of Q with eigenvalue $-\omega_n^2$. Note that you would get the same exact result if you were looking for solutions of the form $U(t) = U_0 \sin(\omega_n t)$. The vector U_0 is called a **mode-shape** of the system, and ω_n is a **resonant frequency** (or natural frequency). Note that since Q is a 3×3 matrix, you will get **three** mode-shape/resonant frequency pairs.

Use MATLAB's eig function to compute the mode-shapes and resonant frequencies of your system for your choice of parameters (and near the equilibrium of your choice). For each of the three mode shapes that you find, simulate the nonlinear system using an initial displacement from equilibrium that will result in the motion being entirely described by that mode shape:

```
U_mode = %your code here
omega_n = %your code here

%small number
epsilon = %your code here

V0 = Veq + epsilon*[Umode;0;0;0];
tspan = %your code here

%run the integration of nonlinear system
% [tlist_nonlinear,Vlist_nonlinear] =...
% your_integrator(my_rate_func,tspan,V0,...);
```

Compare the simulated nonlinear behavior to the predicted behavior from the modal decomposition:

$$\begin{bmatrix} \Delta x \\ \Delta y \\ \Delta \theta \end{bmatrix} = U_0 \cos(\omega t) \tag{28}$$

or in MATLAB:

```
x_modal = Veq(1)+epsilon*Umode(1)*cos(omega_n*tlist);
y_modal = Veq(2)+epsilon*Umode(2)*cos(omega_n*tlist);
theta_modal = Veq(3)+epsilon*Umode(3)*cos(omega_n*tlist);
```

Generate plots comparing the predicted vibration mode to the simulated nonlinear behavior. Please generate at least one animation that shows the nonlinear system vibrating at a **single** frequency (all the motion is described by a single mode shape). You may need to adjust the magnitude of the initial displacement so that the motion is visible while keep the state close enough to the equilibrium so that the linear approximation holds.

Deliverables and Submission Guidelines

Your primary deliverable will be in the form of a lab report that documents what you have accomplished and learned. How you write the lab report is up to you, but there are a few items that you should make sure to include, and a few submission guidelines.

- Remember that each team will be submitting a single assignment. You will need to add yourself to your team's 'assignment05' group on Canvas. Make sure to include the names of you and your teammates at the top of your lab report
- The lab report should be saved as a pdf.

- At the start of the lab report, please approximate how much time each teammate spent working on the assignment (you won't be judged on time spent, but I need it to gauge the assignment difficulty).
- Include a description in your own words of the process that you used to model and simulate the vibrating box. Make sure to include a discussion on linearization and modal analysis.
- If you had to start this mini-project over from scratch, what would you do the same? What would you do differently?
- Write a short reflection describing three things that you learned while working on this project.
- **Video minimum deliverable:** Please submit a video showing the nonlinear system vibrating at a single frequency (all the motion is described by a single mode shape). **Upload the video to youtube, then submit the link on Canvas**.
- **Video optional extra content:** This is not required, but I would love to see animations for all three mode shapes/frequencies, as well as an animation for a large initial displacement (where the linear approximation doesn't hold). If you do want to include these as deliverables, please submit all of this as a single video (and not 3-4 separate videos).
- Include a set of plots comparing the behavior of the nonlinear system with the linear approximation. This could either be plots of the absolute displacements (x, y, θ) vs. t on **three separate axes**, or (if you only want to use one set of axes) plots of the displacements from equilibrium, $(\Delta x = x - x_{eq}, \Delta y = y - y_{eq}, \Delta\theta = \theta - \theta_{eq})$ vs. t on the same set of axes. For the second option, you may need to adjust the scaling of $\Delta\theta$ so that it is visible in comparison to Δx and Δy (i.e. plot $a\Delta\theta$ vs. t for some scaling constant a instead of $\Delta\theta$ vs. t). If you do scale $\Delta\theta$ make sure to include the value of the scaling factor, a, either in the plot or the text of the lab report. **Note** I want to see a set of plots for a small enough initial displacement that the linear approximation is accurate. Furthermore, **I also want you to make** a set of plots for a large enough initial displacement that the linear approximation is not a good approximation.
- **For each of the mode shapes/resonant frequencies** generate a set of plots showing the nonlinear system vibrating at the resonant frequency (all the motion is described by the corresponding mode shape) overlaid with the linear solution of $U_0 \cos(\omega_n t)$ predicted by that mode shape. As before, this could either be plots of the absolute displacements (x, y, θ) vs. t on three separate axes, or plots of the relative displacement, $(\Delta x, \Delta y, a\Delta\theta)$ vs. t on the same set of axes.