Diana Zhen Zhang

Dr. Marcus Rüter

C&EE 103

27 August, 2023

Numerical Model of an Elastic Pendulum

I. Introduction

Throughout the study of dynamical systems, the elastic pendulum emerges as a captivating subject that combines the intricacies of nonlinear mechanics with elastic materials. In this study, the elastic pendulum, pictured in Figure 1, is further explored through an investigation of its behaviors under varying initial conditions using numerical methods and comprehensive analysis. The nuanced dynamics are a result of the interplay between gravitational force, the elastic restoring force of the spring, and the rotational inertia of the system.

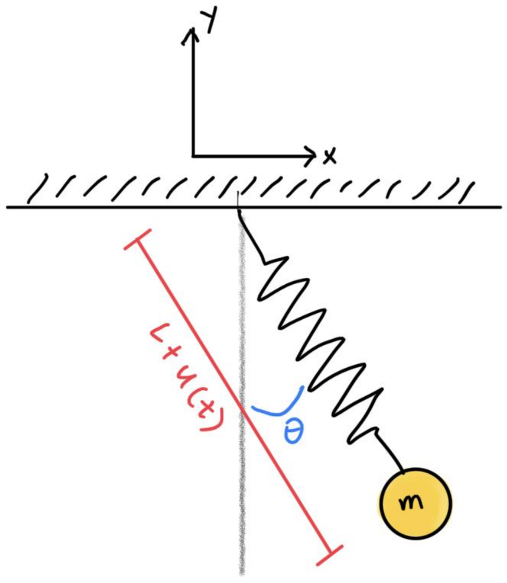

Figure 1: A depiction of the elastic pendulum as a sphere hung from a spring

By following the engineering rationale “as simple as possible, as complex as necessary, a feasible engineering problem may be obtained. This means that any type of friction will be ignored, the spring will be considered massless, the sphere will be considered a rigid body, and the pendulum will only be modeled in two space dimensions rather than three. Additionally, observing the kinetic, gravitational potential, and elastic potential energies that appear in the system is another step that can be taken to realize this model. With the aforementioned assumptions in place, the kinetic energy can be modeled by Equation 1. Where m is the mass of the sphere, $\overline{v}$ is the velocity vector at its center of mass, $\bar{I} = 2/5mr^2$ is the moment of inertia at the center of mass of the sphere with radius r, and θ' is the derivative of its angular position. Observing Figure 1, however, the equation can be further simplified

$$T(t) = \frac{1}{2}m\overline{v}(t)^2 + \frac{1}{2}\bar{I}\theta'(t)^2 \qquad \text{Eq. 1}$$

Observing Figure 1, however, the equation can be further simplified using $\overline{v} = \{u'(t)\ (L + u(t))\theta'(t)\}^T$, where $u(t)$ is the displacement of the spring and L is the length of deformation to ensure that $L + u(t)$ is the total deformed length of the spring at time t.

Regarding the potential energy, it can be expressed as a sum of the gravitational and elastic components as shown in Equation 2. The spring located at UCLA has a gravitational acceleration $g = 9.796\ m/s^2$, a height $h =- (L + u(t))cos\theta$, and a relation between the stretching/compressing force and displacement $F(u(t))$. To find the total energy $E(t)$, the sum of Equations 1 and 2 must be calculated.

$$V(t) = mgh + \int_0^{u(t)} F(u(t))du \qquad \text{Eq. 2}$$

Substituting the first derivatives of angular $\theta(t)$ and cartesian $u(t)$ displacements with the angular velocity $\omega(t)$ and the speed $v(t)$ of the sphere respectively, the system can be turned into the initial value problem modeled in Equation 3. The system is observed from start time t_0 to end time t_f.

$$Y'(t) = f(Y(t)) \qquad \text{Eq. 3}$$

$$Y(t_0) = Y_0$$

In this problem, $Y(t)$, $f(Y(t))$, and Y_0 can be defined as the following

$$\boldsymbol{Y}(t) = \begin{Bmatrix} u(t) \\ \theta(t) \\ v(t) \\ \omega(t) \end{Bmatrix}, \ \boldsymbol{f}(\boldsymbol{Y}(t)) = \begin{Bmatrix} v(t) \\ \omega(t) \\ (L+u(t))\omega(t)^2 + g\cos\theta(t) - \frac{1}{m}\sum_{k=1}^{4} a_k u(t)^k \\ -\frac{2v(t)\omega(t)+g\sin\theta(t)}{(L+u(t))^2+\frac{2}{5}r^2}(L+u(t)) \end{Bmatrix}, \text{ and } \boldsymbol{Y}_0 = \begin{Bmatrix} u_0 \\ \theta_0 \\ v_0 \\ \omega_0 \end{Bmatrix}$$

Using the data of force F and displacement u provided in Table 1, $u(t)$, $v(t)$, $\theta(t)$, $\omega(t)$, and $E(t)$ can be modeled.

Table 1: Data points (u_i, F_i) *measured in meters and Newtons respectively*

u_i	0.05	0.17	0.22	0.28	0.35	0.46	0.51	0.59	0.63	0.69	0.75	0.84	0.98
F_i	0.4	2.2	4.6	5.8	7.3	8.5	9.1	10.7	11.1	12.3	12.8	13.5	14.2
u_i	1.12	1.25	1.33	1.47	1.64	1.82	2.11	2.32	2.64	2.92	3.27	3.65	4.31
F_i	14.9	15.2	15.8	16.4	16.3	16.7	17.1	17.5	18.4	18.9	20.7	21.9	24.1

II. Numerical Methods

II.I Least-Squares Data Fitting

To begin, the data in Table 1 is fitted using a least-squares fitting function, with a degree of at most 4, that expresses the force as a function of the displacement. The least-squares fitting method using a polynomial is a technique used to find the best-fitting polynomial curve that minimizes the sum of squared differences between observed data points and the values predicted by the polynomial model. It’s significant for reducing error when analyzing nonlinear systems because it allows the approximation of complex relationships between variables to be modeled by a simpler polynomial equation.

In this project, this method is implemented in a way that allows for any polynomial with a degree of up to 4 to be implemented. Consider the following initializations:

```
nDP = length(F_i); % Number of force displacement data points
polyOrd = 4; % Polynomial order for least squares
p = zeros(polyOrd,length(F_i)); % Vectors of basis functions
M = zeros(polyOrd,polyOrd); % M-matrix
b = zeros(polyOrd,1); % Right-hand side b-vector
uPlot = linspace(0,5,1000); % u-nodes for plotting
FPlot = zeros(1,length(uPlot)); % F-values for plotting
```

Using this information, the data-fitting begins with a loop that calculates the basis functions for each polynomial order up to `polyOrd`. For each order, it calculates the corresponding basis function by raising the `u_i` values to the power of the current order (`currOrd`). This loop is programmed as shown below.

```
for currOrd = 1:polyOrd
p(currOrd,:) = u_i .^ currOrd;
end
```

Next, another ‘for’ loop, shown below, is used to accumulate values for constructing the M-matrix and the b-vector in the least squares system. For each data point, it computes the product of the basis function vector (`p(:,fDisp)`) with its own transpose then adds it to the `M` matrix. It also updates the `b` vector by adding the product of the basis function vector and the corresponding force value (`F_i(fDisp)`).

```
for fDisp = 1:nDP
M = M + p(:,fDisp) * p(:,fDisp)';
b = b + p(:,fDisp) * F_i(fDisp);
end
```

The system of equations is then solved for the coefficients `a` that minimize the residual error in fitting the data. The result is a column vector containing the coefficients of the polynomial. These coefficients are then stored in the `data` array starting from index `5` (as `g`, `L`, `m`, and `r` occupy indices `1` to `4`) for later use.

```
a = M \ b;
data(5:5 + polyOrd - 1) = a;

for currOrd = 1:polyOrd
fPlot = a(currOrd) * uPlot.^(currOrd);
FPlot = FPlot + fPlot;
end
```

The final 'for' loop, shown in the snippet above, in this method is responsible for generating the fitted force-displ;acement curve (`FPlot`) by evaluating the polynomial using the calculated coefficients `a` for each polynomial order. It accumulates the contributions from each order to the overall curve. The end result of these operations is the fitted force-displacement curve `FPlot`, which is ready to be plotted against the measured data points. Ultimately, this process involves approximating the best-fitting polynomial function to describe the relationship between force and displacement based on the given data points.

II.II Energy Calculation

As explained in the introduction, this initial value problem is driven by the ability to observe the changes, or lack thereof, in energy at varying times. Thus, it is important for the total energy of the system to be calculated. In this case, it is simply an implementation of the content in Equations 1 and 2. Thus, after initializing arrays `KE`, `PE`, and `E` to represent kinetic energy, potential energy, and total energy respectively, the following 'for' loop is programmed to operate over the number of nodes.

```
for n = 1:nN
% Calculate kinetic energy
v_vector = [y(3, n, 1), (L + y(1, n, 1)) * y(4, n, 1)];
KE(n) = 0.5 * m * dot(v_vector, v_vector);
% Calculate gravitational potential energy
h = L + y(1, n, 1);
PE_gravity = m * g * h;
% Calculate elastic potential energy
PE_elastic = 0.5 * Ve * y(1, n, 1)^2;
% Total energy at time node t_n
E(n) = KE(n) + PE_gravity + PE_elastic;
end
```

This loop begins by calculating the kinetic energy at each time node `n`. It computes the velocity vector `v_vector` for the pendulum using its linear velocity `y(3, n, 1)`) and tangential velocity (`L + y(1, n, 1)) * y(4, n, 1)`) before computing the kinetic energy using the formula `0.5 * m * v^2`. The next section calculates the gravitational potential energy due to the pendulum's height h above the reference point by finding the product of the mass, the gravitational acceleration, and the height. The elastic portion of the potential energy is found based on displacement (`y(1, n, 1)`) of the spring from its equilibrium position using the formula `0.5 * Ve * u^2` where Ve is the elastic potential energy coefficient. Lastly, the loop concludes by computing the total energy. In other words, the sum of kinetic and potential energy found in the previous sections is calculated and stored in the `E` array.

II.III Heun's Method

For the analysis of a single elastic pendulum the method used to approximate the solution is Heun's Method. Heun's method, also known as the Improved Euler method, is a numerical integration technique commonly employed for solving ordinary differential equations (ODEs). Specifically designed for first-order ODEs, Heun's method enhances the accuracy of the Euler method by introducing a prediction and a correction step in each iteration to guide a more accurate approximation of the solution trajectory. The method is particularly useful for approximating solutions to initial value problems where the derivative of the function is given with respect to the independent variable. Mathematically, Heun's method is defined as follows. Given an initial condition $y(t_0) = y_0$ and an ordinary differential equation $\frac{dy}{dt} = f(t, y)$, where t represents the independent variable and y is the dependent variable, the method proceeds first with a prediction step to calculate the intermediate value y_{pred} at the next time step $t_1 = t_0 + h * f(t_0, y_0)$. Then a correction step uses the predicted value y_{pred} to compute an average slope at both t_0 and t_1 as follows: $k_1 = f(t_0, y_0)$ and $k_2 = f(t_1, y_{pred})$. Finally, the improved estimate y_1 at $t_1$is computed using the weighted $y_1 = y_0 + \frac{h}{2}(k_1 + k_2)$. In this project, these steps are consolidated into one function `computeHeunSol` shown below. The function f that is referenced holds the derivatives of the state variables.

```
function y = computeHeunSol(y, data, h, nSI)
for n = 1:nSI

k1 = f(y(:, n), data); % k1 using f and current state y
k2 = f(y(:, n) + h * k1, data); % k2 using f and predicted state
y(:, n + 1) = y(:, n) + 0.5 * h * (k1 + k2); % Update y

end
end
```

Ultimately, Heun's method offers a compromise between computational efficiency and accuracy, making it a valuable tool for approximating solutions to differential equations, particularly when higher-order methods may be computationally expensive or unnecessary. Its two-step process, involving prediction and correction, contributes to mitigating the error accumulation observed in the standard Euler method.

II.IV Forward Euler Method

For the second part of this project, where three different elastic pendulums are observed, the Forward Euler method is implemented to approximate the solution. Often referred to simply as Euler's method, it is another fundamental numerical technique employed for approximating solutions to ordinary differential equations . Belonging to the family of explicit one-step methods and is particularly suited for first-order ODEs, it provides a straightforward way to compute an approximate trajectory of a dynamic system. Given an initial condition $y(t_0) = y_0$ and an ordinary differential equation $\frac{dy}{dt} = f(t, y)$, where t represents the independent variable and y is the dependent variable, the method begins with increment calculation, in which the change in the dependent variable over a time step `h` is calculated following $\Delta y = h * f(t_0, y_0)$. Then the computed increment Δy is added to the current value of y_0 to obtain the next approximation at y_1 at time $t = t_0 + h$ following the form $y_1 = y_0 + \Delta y$. This process is iterated over the entire allotted time. In the program, it is implemented as a function `computeFEulerSol` as shown below.

```
function y = computeFEulerSol(y,data,h,nSI)
for n = 1:nSI % Loop over all subintervals
y(:,n+1) = y(:,n) + h * f(y(:,n),data); % Compute y_n+1 using
Forward Euler
end

end
```

Although Euler's method is simple to implement, it comes with limitations. Notably, its global accuracy may deteriorate as the step size `h` increases due to error accumulation, and it is more suitable for problems where the solution does not exhibit rapid variations or strong nonlinearities. Despite its limitations, Euler's method serves as a foundational concept for understanding more advanced numerical integration techniques and plays a crucial role in introductory courses on numerical methods and computational mathematics.

II.V Piecewise Quadratic Interpolation

Piecewise quadratic interpolation can play a valuable role in solving nonlinear systems by providing a way to approximate the behavior of nonlinear functions within specific intervals. When working with nonlinear systems of equations or functions, challenges often present themselves when trying to find exact analytical solutions. In this observation of an elastic pendulum, the mathematical implementation of this method begins by initializing the `tPlot`, `yPlot`, and `EPlot` arrays where `tPlot` is an array of t-nodes for plotting, ranging from `t0` to `tf`, with the same number of elements as the original time array `t`.

```
tPlot = linspace(t0, tf, length(t)); % t-nodes for plotting
yPlot = zeros(4, length(tPlot)); % y-values (at t-nodes) for
plotting
EPlot = zeros(1, length(tPlot)); % E-values (at t-nodes) for
plotting
```

Then, three anonymous functions `L_n`, `L_np1`, and `L_np2` are defined. These are Lagrangean basis functions that will be used to interpolate values between t-nodes. They depend on the subinterval index n and the `tPlotSI` values within that subinterval. These functions follow the form in Equation 3, for a given set of data points (x_i, y_i) for $i = 1, 2, 3, \ldots, n$ and ensure that the resulting interpolant is quadratic and smoothly transitions between adjacent subintervals.

$$L_i(x) = \frac{(x-x_1)(x-x_2)\ldots(x-x_{i-1})(x-x_{i+1})\ldots(x-x_n)}{(x_i-x_1)(x-x_2)\ldots(x_i-x_{i-1})(x_i-x_{i+1})\ldots(x_i-x_n)} \qquad \text{Eq. 3}$$

```
L_n = @(tPlotSI, n) (tPlotSI - t(n+1)) .* (tPlotSI - t(n+2)) /
((t(n) - t(n+1)) * (t(n) - t(n+2)));
L_np1 = @(tPlotSI, n) (tPlotSI - t(n)) .* (tPlotSI - t(n+2)) /
((t(n+1) - t(n)) * (t(n+1) - t(n+2)));
L_np2 = @(tPlotSI, n) (tPlotSI - t(n)) .* (tPlotSI - t(n+1)) /
((t(n+2) - t(n)) * (t(n+2) - t(n+1)));
```

After the basis function definitions, a loop, shown below, is set up to iterate over every subinterval, denoted by the variable `n`. Within each subinterval, the indices of the three t-nodes forming the subinterval are stored in the indices array, and the corresponding t-nodes for plotting (`tPlotSI`) are extracted from tPlot.

```
for n = 1:2:nSI % Loop over every other subinterval
indices = [n n+1 n+2]; % Indices of t-nodes in 2 subintervals
tPlotSI = tPlot(indices); % t-nodes in 2 subintervals
```

Lastly, the loop concludes by evaluating the Lagrangean interpolant for both the `y` values and the `E` values at the `tPlotSI` nodes. The formulas for the interpolants are constructed using the basis functions `L_n`, `L_np1`, and `L_np2`. The `yPlot` and `EPlot` arrays are updated with the interpolated values for the current subinterval.

```
yPlot(:, indices) = y(:, n) * L_n(tPlotSI, n) + y(:, n+1) * ...
L_np1(tPlotSI, n) + y(:, n+2) * L_np2(tPlotSI, n);
EPlot(indices) = E(n) * L_n(tPlotSI, n) + E(n+1) *
L_np1(tPlotSI, n) ...
+ E(n+2) * L_np2(tPlotSI, n);
```

It's important to note that while piecewise quadratic interpolation can be helpful for visualization, local approximation, and derivative estimation, it might not accurately capture all nuances of highly nonlinear systems. For more accurate solutions, sophisticated numerical methods specifically designed for nonlinear systems, like Newton's method or the secant method, are usually employed. Nonetheless, it can still provide valuable insights and aid in the initial stages of analysis and problem-solving.

III. Results and Discussion

III.I) Part I: One Elastic Pendulum With Heun's Method

In Figures 2-4, the oscillatory behaviors of the elastic pendulum are depicted through the solutions to the initial value problem. These include the displacement $u(t)$, linear velocity $v(t)$, angular position $\theta(t)$, and angular velocity $\omega(t)$. Unlike the model of a pendulum, a non-elastic string, the periods and amplitudes of oscillations are not always consistent despite the elimination of external forces, the mass of the spring, and any plastic deformation of the sphere.

The least-squares approximation of the solutions govern the general accuracy of this analysis. When comparing Figure 2 and Figure 3, which were fitted with second degree and fourth degree polynomials respectively, it is evident that the lower degree provides a less accurate estimation which can be attributed to the large number of data points deviating from the fitting polynomial. Generally speaking, a higher-order polynomial can provide a more flexible curve that can closely fit complex or nonlinear patterns in the data. This can be beneficial when the relationship between the independent and dependent variables is intricate. It can also reduce the residuals (the differences between the observed data and the fitted curve) as it captures more details in the data distribution, resulting in a better overall fit.

However, several important considerations come into play. First, higher-order polynomials can lead to overfitting, where the model captures noise rather than the true underlying trend in the data, undermining its ability to generalize to new data. Additionally, numerical instability can arise due to deteriorating condition of the equations, potentially causing further inaccuracies in coefficient estimation. The sensitivity of higher-order polynomials to small data variations can make the model less robust, and the erratic behavior of these polynomials outside the data range can hamper reliable extrapolation. This introduces a trade-off between bias and variance, and finding the right balance is crucial for achieving accurate and generalizable results. It's prudent to assess different polynomial orders, validate the model's performance on unseen data, and consider regularization techniques to mitigate overfitting and enhance the model's reliability.

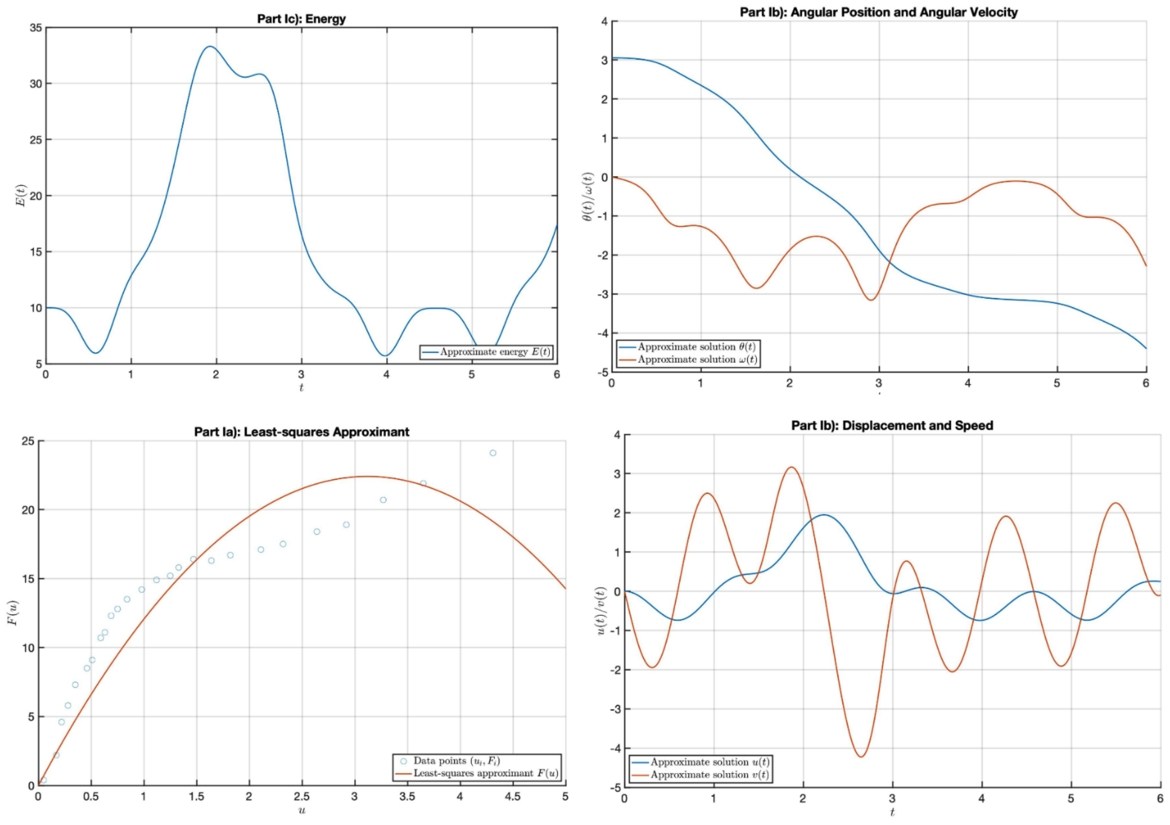

Figure 2: Effects of approximation by least squares function of degree 2

Next, a comparison between Figures 3 and 4 unveils the importance of a high number of nodes, as the solutions in Figure 3 (2000 nodes) depict smooth and continuous graphs. The disparity becomes especially evident at the local maxima, where the graphs using 100 nodes contain linear segments. The number of nodes, or time steps, significantly affects the accuracy of Heun's method. In the predictor step, an estimate of the solution at the next time step is computed using Forward Euler. In the corrector step, the average of the slopes at the current and predicted time steps is used to refine the estimate. Therefore, with more nodes, the time interval between each node decreases, allowing finer variations to be captured and generating a more accurate approximation of the true solution, especially for systems with rapidly changing behavior. However, using more nodes also requires more computational effort. There's a trade-off between accuracy and computational cost: as the number of nodes increases, the solution becomes more accurate, but the computation becomes more time-consuming.

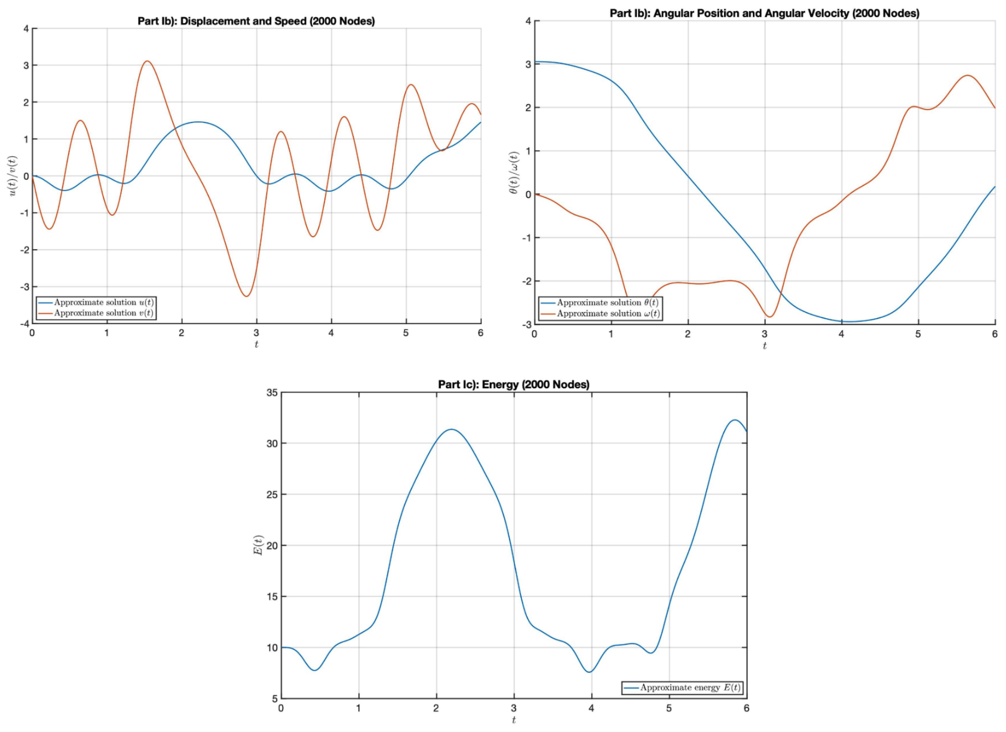

Figure 3: Approximations of solutions using 2000 nodes with Heun's method

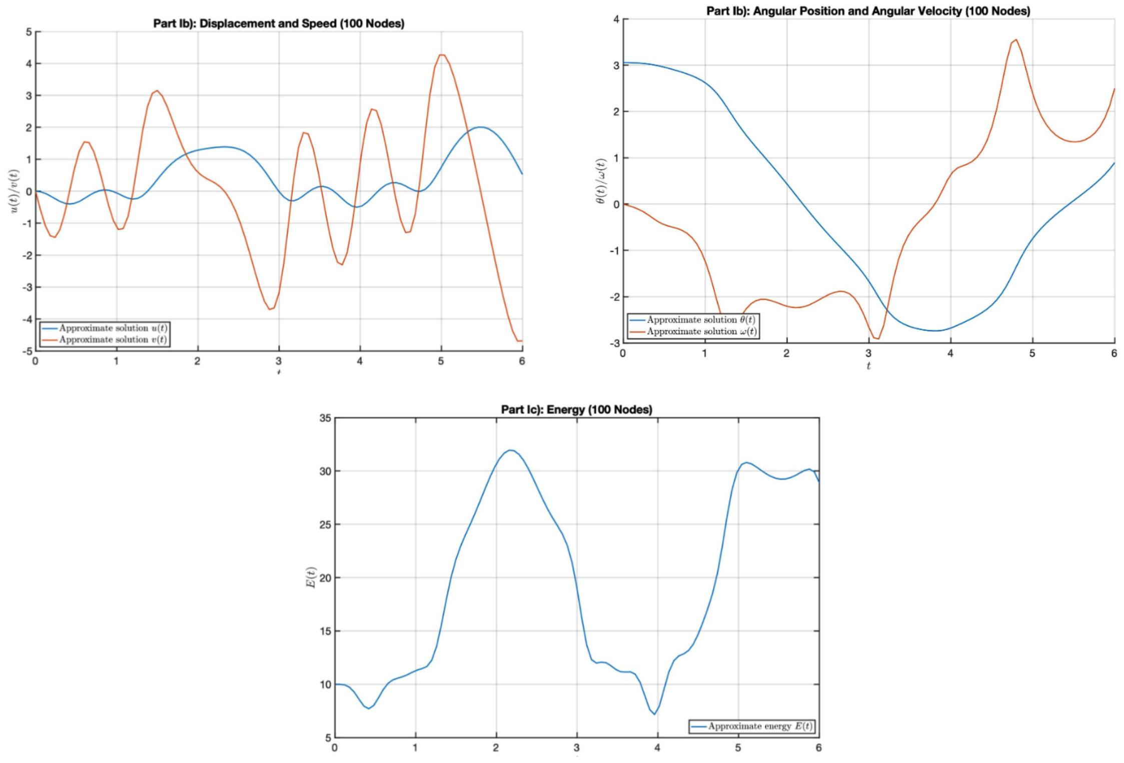

Figure 4: Approximations of solution using 100 nodes with Heun's method

Following a larger number of subintervals, the generation of the pendulum animation varies greatly. As the program iterates over 2000 nodes, it is able to produce a slow but smooth oscillation.

Another important note regarding the use of Heun's method is the effect of errors accumulating from iterations. Taking the graph of $E(t)$ for example, the expectation is that this system has constant total energy. No energy is lost to deformation of the sphere because it is a rigid body nor is any lost to friction because any resistances between surfaces and the environment are neglected. However, it is unrealistic to expect the energy calculated in this project to be equal to a constant. The peaks and valleys found in the graph of total energy can be attributed to the predictor-corrector steps in Heun's method. During the predictor step, the method is a first-order method, meaning that its global truncation error (the difference between the true solution and the numerical approximation) tends to accumulate linearly with each time step. This results in a relatively fast increase in error as the number of time steps increases. This is the same occurrence in the Forward Euler method, as it implements this same time-stepping scheme. However, Heun’s method implements a corrector step that calculates the derivative at the predicted point, and then uses this derivative to make a more informed estimate of the solution's behavior. The iteration over these two steps is responsible for the peak-and-valley behavior of the energy graph.

III.II) Part II: Three Elastic Pendulums With Forward Euler Method

The motion of three elastic pendulums that begin at different initial angles (175º, 176º, and 177º) but start moving at the same time can exhibit interesting and complex behavior. These pendulums are influenced by both gravitational and elastic forces, making their dynamics more intricate compared to simple pendulums. As each pendulum starts moving from its respective initial angle, it oscillates back and forth under the combined effects of gravity and the elastic force. The behavior of the pendulums will be characterized by their initial conditions, including the initial angles, initial velocities, and other relevant parameters. In this scenario, the variation occurs in the initial angle at which the oscillation begins.

The angle at which an elastic pendulum is released, known as the initial angle or amplitude, has been observed in this simulation to play a crucial role in shaping the nature of its subsequent oscillations. The implications of the initial angle on the oscillations of an elastic pendulum are multi-faceted and can be outlined as follows. Firstly, the amplitude of the pendulum's oscillations is intimately tied to the initial angle. A larger initial angle results in a more significant initial displacement of the pendulum from its equilibrium position, thus the pendulum embarks on oscillations that take it to greater heights when released with a larger initial angle. While the period of oscillation, or the time taken for the pendulum to complete one full cycle, remains fairly constant for small variations in the initial angle, the frequency of oscillation experiences subtle changes. The frequency decreases slightly as the initial angle increases, implying that pendulums with larger initial angles oscillate more slowly.

The energy conservation within the system is also influenced by the initial angle. Larger initial angles lead to higher initial potential energy and lower initial kinetic energy. As the pendulum oscillates, energy oscillates between these two forms, while the total energy remains constant, assuming minimal effects from friction and damping. Importantly, nonlinear effects start becoming more evident as the initial angle increases. The pendulum's behavior deviates from simple harmonic motion, as the restoring force arising from the elastic potential energy becomes stronger with larger amplitudes. This can result in intricate behaviors such as period doubling, chaotic motion, or resonance.

Lastly, the stability of the pendulum's motion can be affected by the initial angle. Pendulums released close to the vertical position (around 180 degrees) can encounter unstable equilibria, leading to unpredictable motion or tipping over.

IV. Conclusion

Throughout this project, the specifications of an elastic pendulum were successfully modeled through the implementations of numerical methods including least-squares data fitting, Heun’s method, Forward Euler method, and piecewise quadratic interpolation. The benefits of each method were addressed, along with its limitations. With Heun’s method it can be addressed as simply an improvement of the Forward Euler method, as it implements wiser predictions by considering the derivative of the function when generating a step size. However, because this introduces another level of complexity, it easily becomes a computational burden. The methods used to refine data, least-squares fitting and piecewise quadratic interpolation, have been proven to benefit from a higher order polynomial. However, the consequence of infinitely increasing the degree is actually a less accurate fit due to the involvement of noise and irrelevant data.

Ultimately, this model and solution of the elastic pendulum proves the efficiency of using various numerical methods to solve complex, nonlinear engineering problems when simplified to a logical complexity all while achieving the balance between computational intricacy and rational adaptability.

V. Appendix

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%
% Final Project: The Elastic Pendulum
%
% Author: Marcus Rüter
% Date: 08/27/2023
% Edited by: Diana Zhen Zhang (805777341)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%

%% Clear Cache
clc;
clearvars;
close all;

%% Initialization
whichProblem = 1;
% 1 - Part I: One pendulum
% 2 - Part II: Three pendulums

% Methods: % 1 - Heun's method
% 2 - Forward Euler Method

set(0,'DefaultFigureRenderer','painters'); % Create vector
graphics
if whichProblem == 1 % Part I
whichMethod = 1;
nPend = 1;
t0 = 0;
tf = 6;
nSI = 2000; % number of sub intervals
% Define conditions per method
elseif whichProblem == 2

% Part II
whichMethod = 2;
nPend = 3;
t0 = 0;
tf = 6;
nSI = 20000; % Define conditions per method
else
error('Unknown problem!');
end

g = 9.796; % Gravity of Earth at UCLA [m/s^2]
```

```
L = 1.7; % Undeformed length of pendulum [m]
m = 0.6; % Mass of the sphere [kg]
r = 0.3; % Radius of the sphere [m]
data(1) = g; % Store g in data array
data(2) = L; % Store L in data array
data(3) = m; % Store m in data array
data(4) = r; % Store r in data array
Ve = 0; % Elastic potential energy
u0 = 0; % Initial value for u
theta0 = 175 * pi/180; % Initial value for theta (Part I)
v0 = 0; % Initial value for v
omega0 = 0; % Initial value for omega
nN = nSI + 1; % Number of nodes
h = (tf-t0)/nSI; % Subinterval length h
t = 0:h:tf; % Array of t-nodes
y = zeros(2, nN); % Approximate solution vector at nodes t_n
y(1,1,1) = u0; % Initial condition for u (pendulum 1)
y(2,1,1) = theta0; % Initial condition for theta (pendulum 1)
y(3,1,1) = v0; % Initial condition for v (pendulum 1)
y(4,1,1) = omega0; % Initial condition for omega (pendulum 1)
y(1,1,2) = u0; % Initial condition for u (pendulum 2)
y(2,1,2) = 176 * pi/180; % Initial condition for theta (pendulum
2)
y(3,1,2) = v0; % Initial condition for v (pendulum 2)
y(4,1,2) = omega0; % Initial condition for omega (pendulum 2)
y(1,1,3) = u0; % Initial condition for u (pendulum 3)
y(2,1,3) = 177 * pi/180; % Initial condition for theta (pendulum
3)
y(3,1,3) = v0; % Initial condition for v (pendulum 3)
y(4,1,3) = omega0; % Initial condition for omega (pendulum 3)
xSphere = zeros(1,nPend); % x-position of the sphere(s)
ySphere = zeros(1,nPend); % y-position of the sphere(s)
plotSpring = zeros(1,nPend); % Array for plotting each spring
plotSphere = zeros(1,nPend); % Array for plotting each sphere
dummyX = zeros(2,1); % Dummy x- and y-posit. of spring for plot.
dummyC = 0; % Dummy center of sphere for plotting

% Measured u-values of spring
u_i = [0.05 0.17 0.22 0.28 0.35 0.46 0.51 0.59 0.63 0.69 0.75
0.84 0.98 ...
1.12 1.25 1.33 1.47 1.64 1.82 2.11 2.32 2.64 2.92 3.27 3.65
4.31];

% Measured F-values of spring
F_i = [0.4 2.2 4.6 5.8 7.3 8.5 9.1 10.7 11.1 12.3 12.8 13.5 14.2
14.9 15.2 ...
15.8 16.4 16.3 16.7 17.1 17.5 18.4 18.9 20.7 21.9 24.1];
```

```
nDP = length(F_i); % Number of force displacement data points
polyOrd = 4; % Polynomial order for least squares
p = zeros(polyOrd,length(F_i)); % Vectors of basis functions
M = zeros(polyOrd,polyOrd); % M-matrix
b = zeros(polyOrd,1); % Right-hand side b-vector
uPlot = linspace(0,5,1000); % u-nodes for plotting
FPlot = zeros(1,length(uPlot)); % F-values (at u-nodes) for
plotting

%% Part Ia) Least-squares Data Fitting
for currOrd = 1:polyOrd
p(currOrd,:) = u_i .^ currOrd;
end
for fDisp = 1:nDP
M = M + p(:,fDisp) * p(:,fDisp)';
b = b + p(:,fDisp) * F_i(fDisp);
end
a = M \ b;
data(5:5 + polyOrd - 1) = a;
for currOrd = 1:polyOrd
fPlot = a(currOrd) * uPlot.^(currOrd);
FPlot = FPlot + fPlot;
end

%% Parts Ib) and IIb) Compute Numerical Results for Non-linear
IVP
switch whichMethod % Compute result based on numerical method
case 1 % .. method
for i = 1:nPend % Loop over all pendulums
y(:,:,i) = computeHeunSol(y(:,:,i),data,h,nSI);
end
case 2 % .. method
for i = 1:nPend % Loop over all pendulums
y(:,:,i) = computeFEulerSol(y(:,:,i),data,h,nSI);
end
otherwise
error('Unknown numerical method!');
end

%% Part Ic) Compute Energy
if whichProblem == 1 % Only required for Part I
KE = zeros(1, nN); % Initialize array for kinetic energy
PE = zeros(1, nN); % Initialize array for potential energy
E = zeros(1, nN);
for n = 1:nN

% Calculate kinetic energy
```

```
v_vector = [y(3, n, 1), (L + y(1, n, 1)) * y(4, n, 1)];
KE(n) = 0.5 * m * dot(v_vector, v_vector);
% Calculate gravitational potential energy
h = L + y(1, n, 1);
PE_gravity = m * g * h;
% Calculate elastic potential energy
PE_elastic = 0.5 * Ve * y(1, n, 1)^2;
% Total energy at time node t_n
E(n) = KE(n) + PE_gravity + PE_elastic;
end
end




%% Parts Id) and IIc) Create Pendulum Animation
figure(1); % Open figure 1
hold on; % Put hold to on
for i = 1:nPend % Loop over all pendulums
plotSpring(i) = plot(dummyX,dummyX,'LineWidth',5); % Plot init.
spring
plotSphere(i) = plot(dummyC,dummyC,'.','MarkerSize',250); % Plot
sph.
end
plot(0,0,'.','MarkerSize',50); % Plot pin
title('The Elastic Pendulum'); % Set title
xlabel('x','Interpreter','LaTeX'); % Set x-label
ylabel('y','Interpreter','LaTeX'); % Set y-label
xlim([-1.15*(L+max(y(1,:))) 1.15*(L+max(y(1,:)))]); % Set
x-limits
ylim([-1.15*(L+max(y(1,:))) 1.15*(L+max(y(1,:)))]); % Set
y-limits
grid on; % Turn grid on
axis square; % Use equal lengths in the x- and y-direc.
set(gcf,'Position',[50 50 900 900]); % Set position and size of
figure
set(gca,'LineWidth',3,'FontSize',18); % Set axis widths and font
size
for n = 1:nSI % Loop over all subintervals
for i = 1:nPend % Loop over all pendulums
% Calculate x- and y-positions of sphere (or spring) at t_n+1
xSphere(i) = (L+y(1,n+1,i))*sin(y(2,n+1,i));
ySphere(i) = -(L+y(1,n+1,i))*cos(y(2,n+1,i));
% Copy positions of sphere into plotSpring and plotSphere
set(plotSpring(i),'xdata',[0 xSphere(i)],'ydata',[0
ySphere(i)]);
set(plotSphere(i),'xdata',xSphere(i),'ydata',ySphere(i));
```

```
end
if n == 1
pause % Stop initial frame of animation
end
drawnow; % Make sure to update pendulum plot
end

%% Part Ie) Construct Interpolant
if whichProblem == 1 % Create interpolant only for Part I
[tPlot,yPlot,EPlot] = constructPQuadInterpl(t,y,E,nSI,t0,tf);
end

%% Part If) Plot Results for Part I
if whichProblem == 1 % Create plots only for Part I
figure(2); % Open figure 2 for results from Part Ia)
hold on; % Put hold to on
plot(u_i,F_i,'o','Markersize',10) % Plot (u_i,F_i) data points
plot(uPlot,FPlot,'LineWidth',2); % Plot least-sq. F(u)
interpolant
title('Part Ia): Least-squares Approximant'); % Set title
xlabel('$u$','Interpreter','LaTeX'); % Set x-label
ylabel('$F(u)$','Interpreter','LaTeX'); % Set y-label
xlim([0 5]); % Set x-limits
ylim([0 25]); % Set y-limits
grid on; % Turn grid on
legend('Data points $(u_i,F_i)$','Least-squares approximant
$F(u)$',...
'Location','SE','Interpreter','LaTeX'); % Create legend
set(gcf,'Position',[30 350 1200 750]); % Set position and size
of fig.
set(gca,'LineWidth',2,'FontSize',20); % Set axis widths and font
size
figure(3); % Open figure 3 for results from Part Ic)
plot(t,E,'LineWidth',2); % Plot interpolant E(t)
title('Part Ic): Energy'); % Set title
xlabel('$t$','Interpreter','LaTeX'); % Set x-label
ylabel('$E(t)$','Interpreter','LaTeX'); % Set y-label
xlim([t0 tf]); % Set x-limits
grid on; % Turn grid on
legend('Approximate energy
$E(t)$','Location','SE','Interpreter',...
'LaTeX'); % Create legend
set(gcf,'Position',[30 350 1200 750]); % Set position and size
of fig.
set(gca,'LineWidth',2,'FontSize',20); % Set axis widths and font
size
figure(4); % Open figure 4 for results from Part Ib)
```

```
hold on; % Put hold to on
plot(tPlot,yPlot(1,:),'LineWidth',2); % Plot interpolated u(t)
plot(tPlot,yPlot(3,:),'LineWidth',2); % Plot interpolated v(t)
title('Part Ib): Displacement and Speed'); % Set title
xlabel('$t$','Interpreter','LaTeX'); % Set x-label
ylabel('$u(t) / v(t)$','Interpreter','LaTeX'); % Set y-label
xlim([t0 tf]); % Set x-limits
grid on; % Turn grid on
legend('Approximate solution $u(t)$','Approximate solution
$v(t)$',...
'Location','SW','Interpreter','LaTeX'); % Create legend
set(gcf,'Position',[30 350 1200 750]); % Set position and size
of fig.
set(gca,'LineWidth',2,'FontSize',20); % Set axis widths and font
size
figure(5); % Open figure 5 for results from Part Ib)
hold on; % Put hold to on
plot(tPlot,yPlot(2,:),'LineWidth',2); % Plot interpolated
theta(t)
plot(tPlot,yPlot(4,:),'LineWidth',2); % Plot interpolated
omega(t)
title('Part Ib): Angular Position and Angular Velocity'); % Set
title
xlabel('$t$','Interpreter','LaTeX'); % Set x-l.
ylabel('$\theta(t) / \omega(t)$','Interpreter','LaTeX'); % Set
y-l.
xlim([t0 tf]); % Set x-limits
grid on; % Turn grid on
legend('Approximate solution $\theta(t)$', ...
'Approximate solution $\omega(t)$','Location','SW',...
'Interpreter','LaTeX'); % Create legend
set(gcf,'Position',[30 350 1200 750]); % Set position and size
of fig.
set(gca,'LineWidth',2,'FontSize',20); % Set axis widths and font
size
end

%% Function computeHeunSol
function y = computeHeunSol(y, data, h, nSI)
for n = 1:nSI
k1 = f(y(:, n), data); % Calculate k1 using f and current state
y
k2 = f(y(:, n) + h * k1, data); % Calculate k2 using f and
predicted state k1
y(:, n + 1) = y(:, n) + 0.5 * h * (k1 + k2); % Update y for next
time step
end
```

```
end

%% Function computeFEulerSol
function y = computeFEulerSol(y,data,h,nSI)
for n = 1:nSI % Loop over all subintervals
y(:,n+1) = y(:,n) + h * f(y(:,n),data); % Compute y_n+1 using
Forward Euler
end
end

%% Function f
function fReturn = f(Y, data)
% Define the ODEs based on the given system
% Y is a vector containing [u, theta, v, omega]
u = Y(1);
theta = Y(2);
v = Y(3);
omega = Y(4);
% Extract constants from the data vector
g = data(1);
L = data(2);
m = data(3);
r = data(4);
a = data(5:end);
% Calculate the sum term for the polynomial force F(u)
sumTerm = 0;
for ord = 1:length(a)
sumTerm = sumTerm + a(ord) * u^ord;
end
% Compute the derivatives of the state variables
du_dt = v;
dtheta_dt = omega;
dv_dt = (L+u) * omega^2 + g * cos(theta) - sumTerm / m;
domega_dt = -((2*v*omega + g*sin(theta)) / ((L+u)^2 +
(2/5)*r^2)) * (L+u);
% Return the derivatives as a column vector
fReturn = [du_dt; dtheta_dt; dv_dt; domega_dt];
end

%% Function constructPQuadInterpl
function [tPlot, yPlot, EPlot] = constructPQuadInterpl(t, y, E,
nSI, t0, tf)
% Initialization
tPlot = linspace(t0, tf, length(t)); % t-nodes for plotting
yPlot = zeros(4, length(tPlot)); % y-values (at t-nodes) for
plotting
```

```
EPlot = zeros(1, length(tPlot)); % E-values (at t-nodes) for
plotting
% Define Lagrangean basis functions as anonymous functions
L_n = @(tPlotSI, n) (tPlotSI - t(n+1)) .* (tPlotSI - t(n+2)) /
((t(n) - t(n+1)) * (t(n) - t(n+2)));
L_np1 = @(tPlotSI, n) (tPlotSI - t(n)) .* (tPlotSI - t(n+2)) /
((t(n+1) - t(n)) * (t(n+1) - t(n+2)));
L_np2 = @(tPlotSI, n) (tPlotSI - t(n)) .* (tPlotSI - t(n+1)) /
((t(n+2) - t(n)) * (t(n+2) - t(n+1)));
for n = 1:2:nSI % Loop over every other subinterval
indices = [n n+1 n+2]; % Indices of t-nodes in 2 subintervals
tPlotSI = tPlot(indices); % t-nodes in 2 subintervals
% Evaluate Lagrangean interpolant for y and E at tPlotSI
yPlot(:, indices) = y(:, n) * L_n(tPlotSI, n) + y(:, n+1) *
L_np1(tPlotSI, n) + y(:, n+2) * L_np2(tPlotSI, n);
EPlot(indices) = E(n) * L_n(tPlotSI, n) + E(n+1) *
L_np1(tPlotSI, n) + E(n+2) * L_np2(tPlotSI, n);
end
end
```