Lagrangian Double Pendulum Lab - Computational Simulation and Analysis

Troy Chin
California State University, Fullerton

May 2025

1 Introduction

The focus of this lab is to explore the harmonic motion of a double pendulum in 2D space using the principles of Lagrangian mechanics to solve the governing equations of motion both analytically and numerically. The objective of the lab was to simulate an accurate computational model of the double pendulum and implement numerical methods to solve the equations of motion for the pendulum and visualize its motion over time.

2 Theory

2.1 Lagrangian Mechanics

Lagrangian mechanics provides a powerful formalism for deriving the equations of motion of a mechanical system. The Lagrangian $\mathcal{L}$ is defined as the difference between the kinetic energy T and potential energy V of the system:

$$\mathcal{L} = T - V$$

The equations of motion are derived from the Euler-Lagrange equations:

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{q}_i}\right) - \frac{\partial \mathcal{L}}{\partial q_i} = 0$$

where q_i are the generalized coordinates of the system.

2.2 Double Pendulum Model

Consider a double pendulum with two masses m_1 and m_2, and two rods of lengths l_1 and l_2. The angular displacements from the vertical are θ_1 and θ_2. The position of each mass in Cartesian coordinates is given by:

$$x_1 = l_1 \sin\theta_1 \tag{1}$$
$$y_1 = -l_1 \cos\theta_1 \tag{2}$$
$$x_2 = x_1 + l_2 \sin\theta_2 = l_1 \sin\theta_1 + l_2 \sin\theta_2 \tag{3}$$
$$y_2 = y_1 - l_2 \cos\theta_2 = -l_1 \cos\theta_1 - l_2 \cos\theta_2 \tag{4}$$

2.3 Kinetic and Potential Energies

The kinetic energy T consists of the rotational and translational kinetic energies of both masses:

$$T = \frac{1}{2}m_1(\dot{x}_1^2 + \dot{y}_1^2) + \frac{1}{2}m_2(\dot{x}_2^2 + \dot{y}_2^2)$$

After differentiating the coordinates and simplifying:

$$T = \frac{1}{2}m_1 l_1^2\dot{\theta}_1^2 + \frac{1}{2}m_2\left[l_1^2\dot{\theta}_1^2 + l_2^2\dot{\theta}_2^2 + 2l_1 l_2\dot{\theta}_1\dot{\theta}_2\cos(\theta_1 - \theta_2)\right]$$

The potential energy V due to gravity is:

$$V = -m_1 g l_1\cos\theta_1 - m_2 g(l_1\cos\theta_1 + l_2\cos\theta_2)$$

2.4 Equations of Motion

Substituting into the Euler-Lagrange equations yields the following coupled, nonlinear second-order differential equations:

$$(m_1 + m_2)l_1\ddot{\theta}_1 + m_2 l_2\ddot{\theta}_2\cos(\theta_1 - \theta_2) + m_2 l_2\dot{\theta}_2^2\sin(\theta_1 - \theta_2) + (m_1 + m_2)g\sin\theta_1 = 0$$

$$l_2\ddot{\theta}_2 + l_1\ddot{\theta}_1\cos(\theta_1 - \theta_2) - l_1\dot{\theta}_1^2\sin(\theta_1 - \theta_2) + g\sin\theta_2 = 0$$

These equations are typically solved numerically due to their complexity and sensitivity to initial conditions.

2.5 Numerical Methods

For numerical integration, the 4th-order Runge-Kutta method is employed to solve the coupled system. The method discretizes time and iteratively computes the state evolution using weighted slopes:

$$\theta_i(t + \Delta t) = \theta_i(t) + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$\text{where } k_j = f(t + a_j\Delta t, \theta_i + b_j k_j)$$

The Python script `numerical_methods.py` implements this method to evolve the system over discrete time steps.

3 Methodology

Both Python and MATLAB were used for this lab to compare results and efficiency. The key libraries included were:

- `numpy` - For numerical operations
- `matplotlib` - For plotting and constructing graphical models
- `matplotlib.animation` - To animate the double pendulum's motion
- `csv` - To gather data on the double pendulum's position and velocity in both the angular and radial directions
- `scipy.optimize` - To perform proper curve fitting and error propagation analysis on the data gathered for this lab.

A curve fitting routine was implemented to compare the motion between the angular velocities of both the first and second masses using a theoretical model of the pendulum. The data was given as follows:

- Angle 1: A=0.2327, freq=0.9874, phase=1.0357, c=0.0580
- Angle 2: A=0.4952, freq=3.1571, phase=-1.7485, c=-0.1698

4 Implementation

4.1 Python

```
from pendulum import DoublePendulum
from numerical_methods import NumericalMethods
from visualization import Visualization
from data_logger import DataLogger
import numpy as np

def main():
    """Main entry point of the system."""
    # Define initial conditions
    mass1, mass2 = 1.0, 1.0
    length1, length2 = 1.0, 1.0
    angle1, angle2 = np.pi / 4, np.pi / 2 # Initial angles in radians
    velocity1, velocity2 = 0.0, 0.0  # Initial angular velocities in radians per second
    g = 9.81

    # Initialize the double pendulum and numerical methods
    pendulum = DoublePendulum(mass1, mass2, length1, length2, angle1, angle2, velocity1, velocity2, g)
    methods = NumericalMethods(dt=0.01)

    # Set up data logger
    logger = DataLogger()

    # Run simulation
    time_steps = 1000
    dt = 0.01
    for t in range(time_steps):
        # Call compute_state correctly
        current_state = pendulum.compute_state()  # Get the current state of the pendulum
        logger.log_state(current_state)  # Log the current state

        # Update the pendulum state
        pendulum.step(dt)  # Update the state using your existing step method

        # Use the current state as the initial conditions for the ODE solver
        next_state = methods.solve_ode(pendulum.equations_of_motion, current_state)

        # Log the next state if needed or print it
        print(next_state)

    # Prepare data for visualization
    angles1 = [state[0] for state in logger.data]
    angles2 = [state[1] for state in logger.data]
    velocities1 = [state[2] for state in logger.data]
    velocities2 = [state[3] for state in logger.data]
    # Create visualization instance and animate
    visualization = Visualization(logger, pendulum, dt)
    visualization.animate(frames=len(logger.data))  # Adjust frames as needed

    # Plot angles and velocities
    visualization.plot_angles_and_velocities(t_max=time_steps * dt, dt=dt,
                                            angles1=angles1,
                                            angles2=angles2,
                                            velocities1=velocities1,
                                            velocities2=velocities2)

    # Plot phase space of pendulum
    visualization.plot_phase_space()

    # Save logged data to CSV
    logger.save_to_csv('double_pendulum_data.csv')

    # Perform and visualize curve fitting
    visualization.curve_fit_angles(filename='double_pendulum_data.csv', dt=dt)

if __name__ == "__main__":
    main()
```

4.2 MATLAB

```
% File name: pendulum.m
% Author: Troy Chin
% Date: 15 November 2024
% Version: 1.0
% Purpose: Reimplementation of the Double Pendulum Lab in MATLAB with animation

% Define necessary parameters
m1 = 1.0; % mass1
m2 = 1.0; % mass2
l1 = 1.0; % length1
l2 = 1.0; % length2
g = 9.81; % Gravitational constant

% Define initial conditions
theta_1 = pi/4; % angle1
theta_2 = pi/6; % angle2
theta_dot1 = 0.0; % velocity1
theta_dot2 = 0.0; % velocity2

% Time span and step size
t_span = [0 5]; % Time from 0 to 20 seconds
dt = 0.01; % Time step
time = t_span(1):dt:t_span(2); % Time array

% Initial conditions for the system
y0 = [theta_1, theta_dot1, theta_2, theta_dot2]; % Initial state vector

% Define the system of differential equations (Euler-Lagrange equations)
% Equations for first and second pendulum
```

```
eqns = @(t, y) [
    y(2);
    (-g*(2*m1+m2)*sin(y(1)) - m2*g*sin(y(1)-2*y(3)) - 2*sin(y(1)-y(3))*m2*(y(4)^2*l2 + y(2)^2*l1*cos(y(
    y(4);
    (2*sin(y(1)-y(3))*(y(2)^2*l1*(m1+m2) + g*(m1+m2)*cos(y(1)) + y(4)^2*l2*m2*cos(y(1)-y(3))))/(l2*(2*m
];

% Solve the system using ode45
[time, sol] = ode45(eqns, time, y0);

% Extract positions from the solution
theta1 = sol(:,1);
theta2 = sol(:,3);

% Calculate x and y positions of the masses
x1 = l1 * sin(theta1); % x position of mass 1
y1 = -l1 * cos(theta1); % y position of mass 1
x2 = x1 + l2 * sin(theta2); % x position of mass 2
y2 = y1 - l2 * cos(theta2); % y position of mass 2

% Set up the figure for animation
figure;
axis equal; % Ensure equal scaling on both axes
xlim([-2, 2]); % Adjust limits based on your pendulum's motion
ylim([-2, 2]);
hold on;
grid on;

% Initial plot of the pendulum components
rod1 = line([0, x1(1)], [0, y1(1)], 'LineWidth', 2, 'Color', 'r'); % First rod
rod2 = line([x1(1), x2(1)], [y1(1), y2(1)], 'LineWidth', 2, 'Color', 'b'); % Second rod
mass1 = plot(x1(1), y1(1), 'ro', 'MarkerFaceColor', 'r'); % Mass 1
mass2 = plot(x2(1), y2(1), 'bo', 'MarkerFaceColor', 'b'); % Mass 2

% Title and labels
title('Double Pendulum Motion');
xlabel('X position (m)');
ylabel('Y position (m)');

% Animation loop
for i = 1:length(time)
    % Update the positions of the rods and masses
    rod1.XData = [0, x1(i)];
    rod1.YData = [0, y1(i)];
    rod2.XData = [x1(i), x2(i)];
    rod2.YData = [y1(i), y2(i)];
    mass1.XData = x1(i);
    mass1.YData = y1(i);
    mass2.XData = x2(i);
    mass2.YData = y2(i);

    % Pause to slow down the animation
    pause(0.01);
```

```
end

% Plot the results (angles and angular velocities)
figure;
subplot(2, 1, 1);
plot(time, theta1, 'r', 'LineWidth', 1.5);
hold on;
plot(time, theta2, 'b', 'LineWidth', 1.5);
title('Pendulum Angles vs Time');
xlabel('Time (s)');
ylabel('Angle (rad)');
legend('Theta 1', 'Theta 2');
grid on;

subplot(2, 1, 2);
plot(time, sol(:, 2), 'r', 'LineWidth', 1.5); % Angular velocity of pendulum 1
hold on;
plot(time, sol(:, 4), 'b', 'LineWidth', 1.5); % Angular velocity of pendulum 2
title('Pendulum Angular Velocities vs Time');
xlabel('Time (s)');
ylabel('Angular Velocity (rad/s)');
legend('Theta dot 1', 'Theta dot 2');
grid on;

% Optional: Save the final figure as a PNG image
% saveas(gcf, 'double_pendulum_simulation.png');
```

5 Results

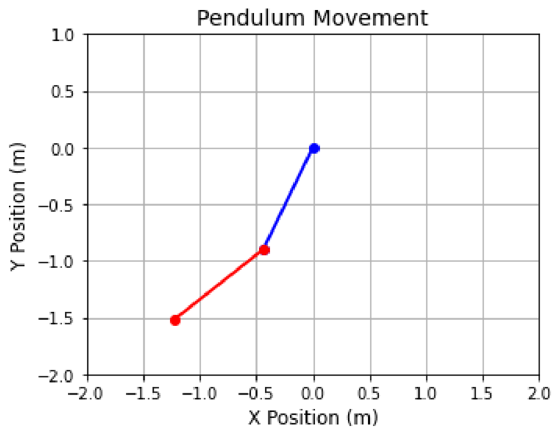

Figure 1: Graphical model of Double Pendulum in Python

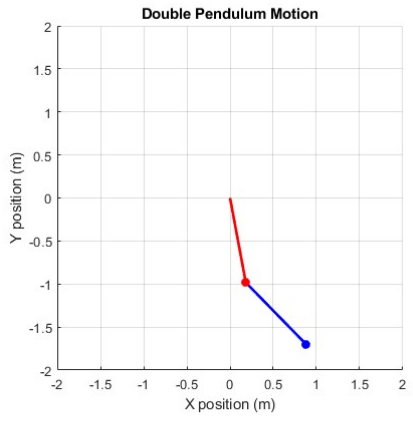

Figure 2: Graphical Model of Double Pendulum in MATLAB

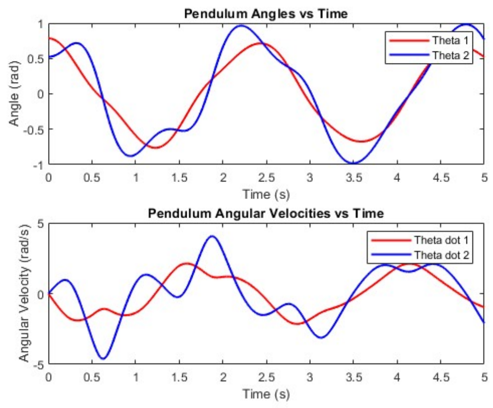

Figure 3: Graphical model of Double Pendulum in Python

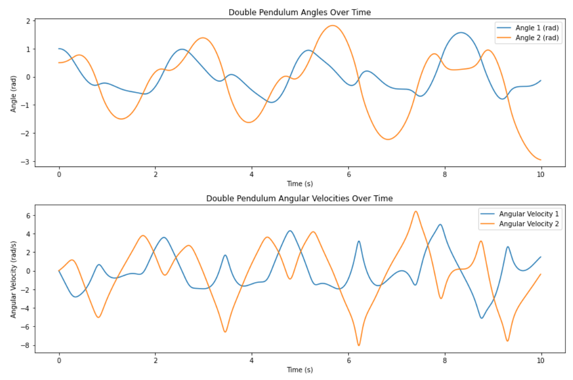

Figure 4: Graphical Model of Double Pendulum in MATLAB

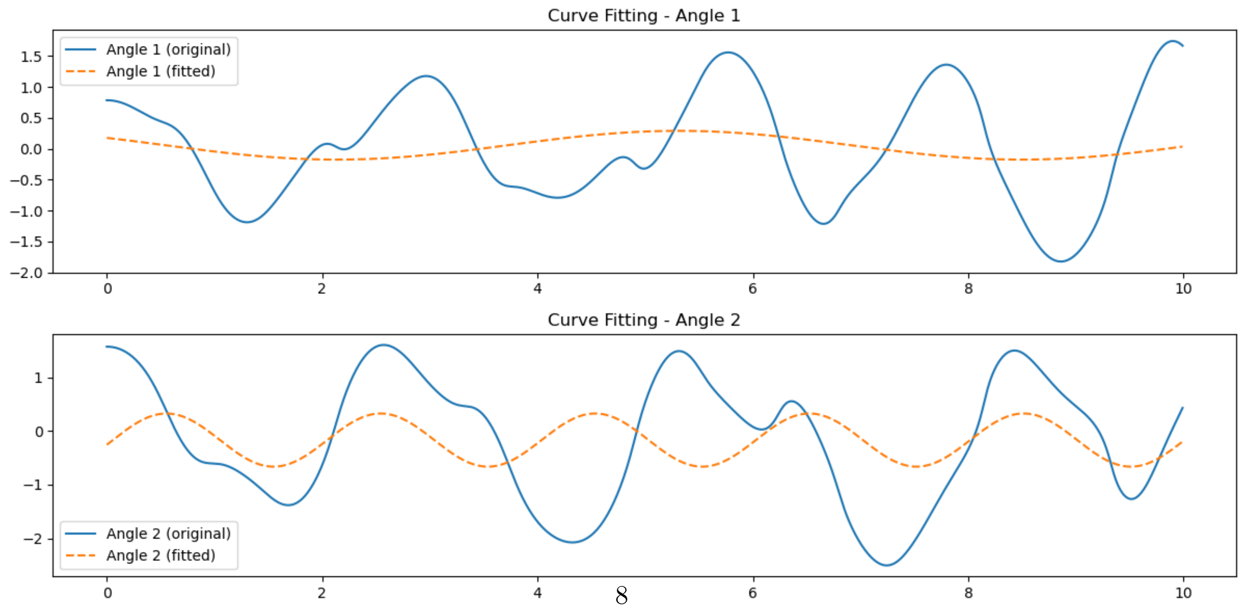

Figure 5: Curve Fitting Model

6 Discussion

The double pendulum system is a hallmark example of deterministic chaos—exhibiting extreme sensitivity to initial conditions and nonlinear coupling between components. During simulation, small variations in starting angles or numerical precision produced dramatically different trajectories over time. This aligns with theoretical expectations and highlights the challenges of long-term prediction in chaotic systems. The results from both Python and MATLAB implementations demonstrated good consistency, validating the correctness of the numerical methods. The Python visualization, in particular, allowed for intuitive observation of the pendulum's complex motion and phase-space trajectories, which are difficult to derive analytically. The curve fitting applied to the angular displacement data provided insight into the system's short-term quasi-periodic behavior. However, the fitting parameters diverged over time as chaotic divergence set in—further confirming the system's nonlinearity and sensitivity. A potential source of error lies in numerical integration drift, particularly when step sizes are not sufficiently small. While the 4th-order Runge-Kutta method balances accuracy and efficiency, future iterations of this lab could explore symplectic integrators that better conserve energy in Hamiltonian systems over long time spans.

The curve fitting models applied to $\theta_1(t)$ and $\theta_2(t)$ significantly diverged from the original time series data, particularly as time increased. This discrepancy stems from the fact that the fitted models (likely composed of sinusoidal or low-order polynomial components) cannot adequately approximate the behavior of a chaotic system like the double pendulum. Chaotic dynamics are highly sensitive to initial conditions and characterized by non-periodic, non-repeating motion. As such, any attempt to fit these trajectories using standard periodic or smooth functional forms results in only a crude approximation—typically accurate only in the short-term transient phase. Over time, even slight deviations in phase or amplitude quickly accumulate, resulting in large prediction errors. This mismatch highlights a key insight: while curve fitting may be useful for analyzing quasi-periodic signals, it breaks down when applied to strongly nonlinear and chaotic systems. For a more faithful representation of the double pendulum's dynamics, one must rely on numerical integration and phase space analysis rather than closed-form approximations.

7 Conclusion

This lab successfully employed Lagrangian mechanics and numerical methods to simulate and analyze the motion of a double pendulum. Using Python and MATLAB, we derived the system's nonlinear differential equations, implemented numerical solvers, and visualized the motion and phase space behavior.

Key takeaways include:

- The double pendulum system exemplifies chaotic dynamics, where long-term prediction becomes infeasible due to sensitivity to initial conditions.
- Numerical solutions using the 4th-order Runge-Kutta method provided stable short-term simulations and qualitatively matched expected trajectories.
- Curve fitting and phase-space analysis offered insights into transient periodicity and long-term divergence.

Future work may include the implementation of energy-preserving integrators, comparison against experimental physical pendulum systems, or extending the model into 3D or with damping and external forcing terms. Overall, this lab provided a compelling demonstration of applying computational physics techniques to analyze a classic chaotic system.