Numerical Simulation and Analysis of Pendulum Dynamics

Luke Keely

Trinity College Dublin,
keelyl@tcd.ie

Abstract. In this lab exercise, we look at numerical simulation for pendulum dynamics over various conditions. We compare three numerical methods: Euler's method, the Trapezoidal Rule, and the Runge-Kutta method. The experiment began with a linear approximation of the pendulum and then moved to more complex scenarios, including non-linear dynamics, damping forces, and external driving forces.

The linear model is suitable for small initial energies, but accuracy decreases with increasing energy. The introduction of nonlinear terms allowed for more realistic motion at higher energies. Damping forces provided realistic simulations, acting closer to real-world systems. The phase space analysis transitioned from periodic to chaotic behavior over varying driving force amplitudes.

The comparative analysis of the numerical methods showed that Euler's method was less accurate while computationally simple; the Trapezoidal Rule was a good balance between computational complexity and accuracy; and the Runge-Kutta method was ideal for high energy simulations at a higher computation cost.

This lab improves our understanding of pendulum dynamics and how to apply numerical methods to practical physics and engineering problems.

Table of Contents

1 Introduction

The pendulum is an excellent example of a system in classical mechanics. In this case, it will serve our dynamic systems for exploring numerical methods for differential equations. A linear differential equation may approximate the simple pendulum; however, in real-world applications, the pendulum has nonlinear behavior.
Furthermore, the pendulum may be affected by forces other than gravity; in this experiment, we look at added damping forces, such as friction or drag, and driving forces, such as a motor or electromagnet. This added complexity provides a look into more advanced dynamics, often seen in engineered systems and real-world applications.

For our pendulum, we use the equation of motion,

$$-\sin(\theta) - k \cdot \omega + A \cdot \cos(\phi \cdot t)$$

where, $-\sin(\theta)$ is the force due to gravity, $-k \cdot \omega$ is a damping force, proportional to $\dot{\theta}$ and, $A \cdot \cos(\phi \cdot t)$ is an external periodic driving force.

The equation above is a nonlinear second-order ODE. Analytical solutions for these types of equations are not available in general. However, numerical methods can be highly useful in understanding the pendulum's motion and the motion of systems like it.

In this experiment, we look at three different numerical methods for solving ODEs, from computationally simple with lower accuracy to more complex algorithms with much greater accuracy needed to compute complex or chaotic systems. These methods are Euler's method, the Trapezoidal Rule, and the fourth-order Runge-Kutta method.

Euler's method is a first-order numerical procedure for solving ordinary differential equations (ODE) with a given initial value. It uses the formula:

$$y_{n+1} = y_n + hf(t_n, y_n)$$

To find the tangent at the point $x4$ and obtain the value of $y(x+h)$, where h is our step size. This method approximates the curve of the solution by a sequence of short-line segments[2].

The **Trapezoidal Rule** is a second-order method from the trapezoidal rule for computing integrals. It used the formula:

$$y_{n+1} = y_n + \frac{1}{2}h(f(t_n, y_n) + f(t_{n+1}, y_{n+1}))$$

This method improves upon Euler's method by better approximating each step[3].

The **fourth-order Runge-Kutta method** (RK4) is a more advanced numerical method for solving ODEs. This method combines four slope estimates for every step. It uses the formula:

$$y_{n+1} = y_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4),$$

Where k_i is each a different slope estimate at different points over the interval[4]. Specifically,
$k1$ is the slope at the beginning of the interval, using $k_1 = f(t_n, y_n)$,
k_2 is the slope at the midpoint of the interval, using $k_2 = f(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2})$,
k_3 is also the slope at the midpoint, but now using $k_3 = f(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2})$, and
k_4 is the slope at the end of the interval, using $k_4 = f(t_n + h, y_n + hk_3)$.

2 Methodology

2.1 Simulation Setup

Linear Pendulum We first set up the most straightforward representation of the pendulum, the linear approximation. When dealing with small oscillations, we may use a linear form of the equation of motion. The equation is simplified using a small-angle approximation where $\sin(\theta) \approx \theta$.
In the code, we create a script called `linear_pendulum.py` that contains a function called `pendulum_motion` used to represent this linear second-order ODE. This linear behavior is an ideal starting point for understanding the pendulum's motion using numerical methods before adding more complex behaviors.

```
# Pendulum's equation of motion
def pendulum_motion(theta, omega, t, k=0.0, phi=0.66667, A=0.0):
  return -theta - k * omega + A * np.cos(phi * t)
```

The `solve_and_plot_pendulum` function uses this equation of motion to calculate the motion over time in little steps. It iteratively updates the angle and angular velocity using the trapezoidal rule. We can then visualize these steps by potting theta versus time and omega versus time. These plots give us an easily understandable idea of the pendulum's motion with differing starting positions and velocities.

Nonlinear Pendulum Moving over the nonlinear pendulum was a straightforward modification; the `pendulum_motion` function was updated to include the nonlinear term $\sin(\theta)$. This nonlinear term increases the system's complexity, making it much more challenging to find analytical solutions, but it can be handled the same in numerical methods. However, the behavior is more realistic and may be better applied to real-world problems.

```
# Pendulum's equation of motion
def pendulum_motion(theta, omega, t, k=0.0, phi=0.66667, A=0.0):
  return -np.sin(theta) - k * omega + A * np.cos(phi * t)
```

The simulation setup and numerical method in the `solve_and_plot_pendulum` function are the same as in the linear case. This allows for an easy comparison between the motion of the models.

2.2 Implementing Numerical Methods

Euler's Method Euler's method is the simplest of the three numerical methods used in the experiment. Its simple implementation makes it an excellent tool for basic systems. However, its accuracy is low, especially when dealing with quickly changing systems. This method is only used once as a stepping stone for the other methods and can be seen in the `method_comparison.py` script.

```
# Euler's Method
theta_vals['euler'][i] = theta_vals['euler'][i-1] + dt * omega_vals['
    euler']
omega_vals['euler'] += dt * f_motion(theta_vals['euler'][i-1],
    omega_vals['euler'], t)
```

Trapezoidal Rule The Trapezoidal rule is more complicated but more accurate than Euler's Method. It provides good approximations for most systems, so it was chosen to simulate the simple linear, simple nonlinear, and damped nonlinear pendulums. The code now calculates the average of the slopes at the beginning and end of each time step, leading to a better step for the pendulum's motion.

```
# Trapezoidal Rule
k1_theta, k1_omega = dt * omega_vals['trap'], dt * f_motion(theta_vals[
    'trap'][i-1], omega_vals['trap'], t)
k2_theta = dt * (omega_vals['trap'] + k1_omega)
k2_omega = dt * f_motion(theta_vals['trap'][i-1] + k1_theta, omega_vals
    ['trap'] + k1_omega, t + dt)
theta_vals['trap'][i] = theta_vals['trap'][i-1] + (k1_theta + k2_theta)
    / 2
omega_vals['trap'] += (k1_omega + k2_omega) / 2
```

Runge-Kutta Method The RK4 method is the most complex and computationally intense but offers the highest accuracy of the chosen methods. This is why it was chosen to simulate the nonlinear damped-driven pendulum. The program calculates four different slopes at each step and takes their weighted average to estimate the next value.

```
# Fourth-Order Runge-Kutta Method
k1 = dt * omega_vals['rk4']
l1 = dt * f_motion(theta_vals['rk4'][i-1], omega_vals['rk4'], t)
k2 = dt * (omega_vals['rk4'] + l1 / 2)
l2 = dt * f_motion(theta_vals['rk4'][i-1] + k1 / 2, omega_vals['rk4'] +
    l1 / 2, t + dt / 2)
k3 = dt * (omega_vals['rk4'] + l2 / 2)
l3 = dt * f_motion(theta_vals['rk4'][i-1] + k2 / 2, omega_vals['rk4'] +
    l2 / 2, t + dt / 2)
k4 = dt * (omega_vals['rk4'] + l3)
l4 = dt * f_motion(theta_vals['rk4'][i-1] + k3, omega_vals['rk4'] + l3,
    t + dt)
theta_vals['rk4'][i] = theta_vals['rk4'][i-1] + (k1 + 2 * k2 + 2 * k3 +
    k4) / 6
omega_vals['rk4'] += (l1 + 2 * l2 + 2 * l3 + l4) / 6
```

2.3 Damped Pendulum Dynamics

Introducing a damping force into the pendulum adds another layer of realism and complexity. The damping force may represent a force like friction or air resistance that is proportional to the velocity and acts in the opposite direction to motion.
The damping force was already integrated into the equation from the simple pendulum scripts. However, the damping constant was left at zero, meaning it had no effect. To enable this damping force, the coefficient was set to 0.5, allowing us to explore the movement of this more complex system.

```
# Pendulum's equation of motion
def pendulum_motion(theta, omega, t, k=0.5, phi=0.66667, A=0.00):
  return -np.sin(theta) - k * omega + A * np.cos(phi * t)
```

2.4 Phase Space Analysis

We use phase space analysis to help us visualize the behavior of dynamic systems. In this experiment, the phase space is used to analyze the motion of the driven nonlinear pendulum. This is the most complex system in the experiment. We can represent the system's trajectory by plotting the angular position θ against the angular velocity ω. This path reveals what conditions allow the system to be stable and periodic and what conditions lead to chaotic motion.

```
plt.scatter(theta_values, omega_values, s=2, c='b')
plt.xlabel('Theta')
plt.ylabel('Omega')
plt.axis('equal')
plt.show()
```

2.5 Comparative Analysis

The comparison between methods is a key part of this experiment. It looks at the performance of each method under varying conditions. Euler's Method, the Trapezoidal rule, and the Runge-Kutta Method are run on both linear and nonlinear systems over different initial conditions. Then, the plots of theta vs. time are superimposed so we can see the path difference and how they dissipate over time.

```
# Function for simulation
def simulate_pendulum(f_motion, initial_theta, plot_dir, ax):
  time_vals = np.zeros(nsteps)
  theta_vals = {'euler': np.zeros(nsteps), 'trap': np.zeros(nsteps), '
      rk4': np.zeros(nsteps)}
  omega_vals = {'euler': 0.0, 'trap': 0.0, 'rk4': 0.0}
```

2.6 Challenges

One of my main issues was visualizing the phase space plots for the driven pendulum. The plots of the trajectories overlapped when theta became too negative. The cause was the pendulum doing many rotations in one direction rather than swimming back and forth. To fix this issue, the code was adjusted to allow the axes to expand as needed. This allowed the plots to capture the pendulum's motion so it could be viewed clearly.

3 Results and Discussion

3.1 Linear and Nonlinear Pendulum Simulations

Small Initial Energy $(\theta, \omega) = (0.2, 0.0)$ Both models show simple harmonic motion when dealing with small angles, such as the one here. The linear model shows the angular displacement, θ, tracing the path of a cosine wave, and the angular velocity ω following a sine wave. The paths of the nonlinear model of the system mirror the linear behavior with such little variation that it is not noticeable by comparing the plots. This behavior highlights the linear approximation's effectiveness in dealing with such initial conditions.

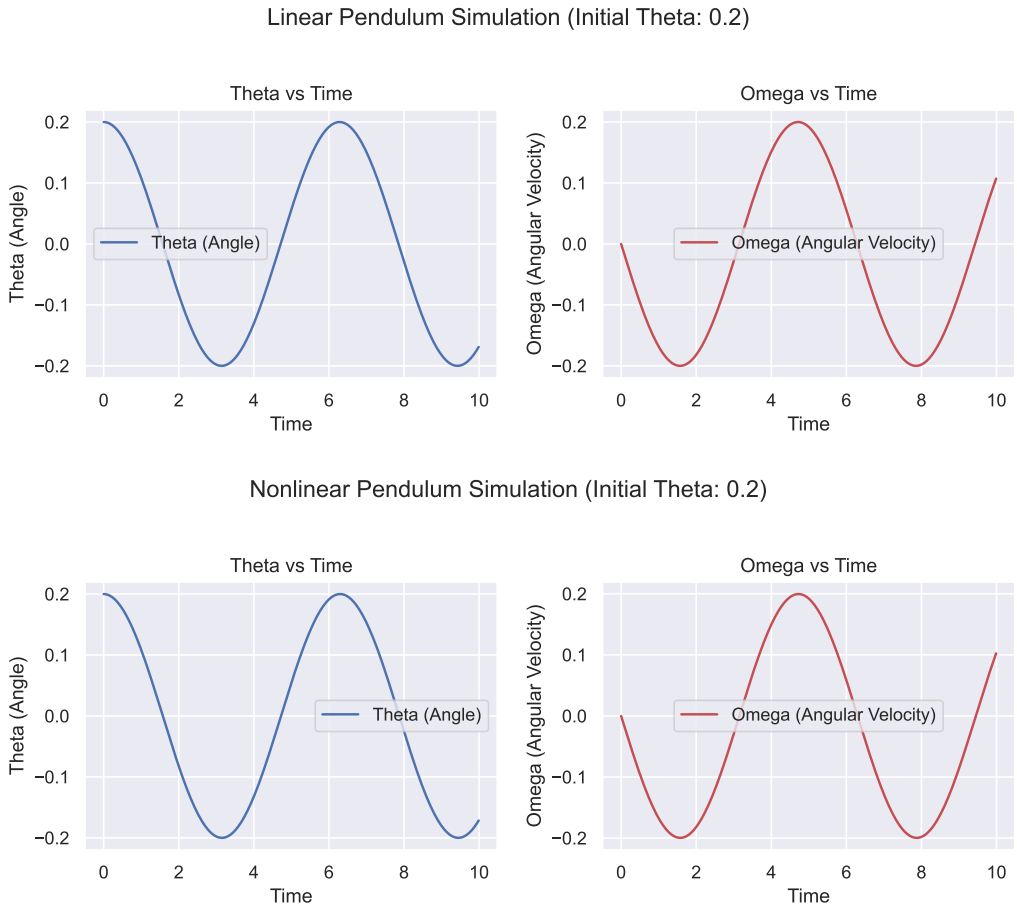

Fig. 3.1: Linear vs Nonlinear $(\theta, \omega) = (0.2, 0.0)$

This approximation works because the sine function behaves at small angles, where the angle itself may approximate the sine of a small angle. The behavior of the sine function carries over to our mechanical system, where we let $\sin\theta \approx \theta$ for small angles, turning our complex nonlinear equations into a good linear approximation, as seen in the plots.

Moderate Initial Energy $(\theta, \omega) = (1.0, 0.0)$ and $(\theta, \omega) = (0.0, 1.0)$ When we increase the angle to something a bit larger or increase the initial velocity, the linear pendulums again display simple harmonic motion with larger amplitudes than the pendulum with a smaller angle. Still, the behavior is very much the same as the previous run.

Where for (1.0, 0.0) the angular displacement, θ, traces a cosine wave and the angular velocity ω a sine wave; and for (0.0, 1.0) the angular displacement, θ, traces a sine wave and the angular velocity ω a cosine wave.

However, there are some visual differences when we compare the linear approximations to the nonlinear equations of motion. The periods of the nonlinear pendulums are longer than the linear pendulums, and the angular velocities change more quickly.

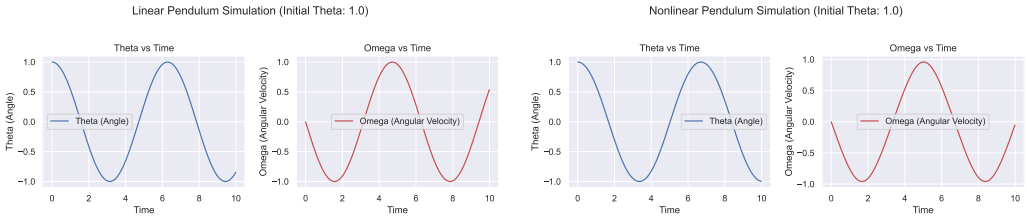

Fig. 3.2: Linear vs Nonlinear $(\theta, \omega) = (1.0, 0.0)$

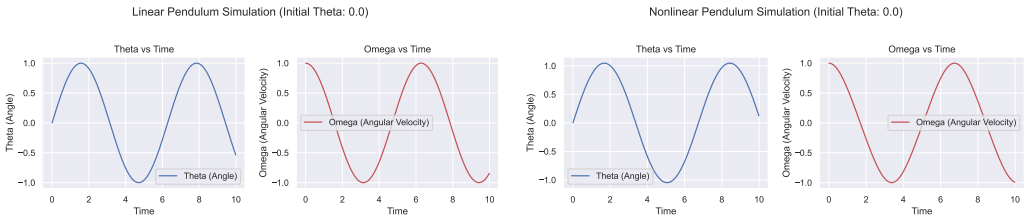

Fig. 3.3: Linear vs Nonlinear $(\theta, \omega) = (0.0, 1.0)$

This deviation represents the limits of the linear approximation, especially at higher initial energies where the linear approximations become increasingly inaccurate.

Large Initial Energy $(\theta, \omega) = (3.14, 0.0)$ As we approach the maximum angle, the behavior of the two models becomes very different. With its simple equation, the linear model struggles to represent the expected motion of the pendulum. The linear plots look similar to what was seen in the previous.
However, the nonlinear model shows very different and realistic behavior. We can see a much slower take-off at the end of the swing, followed by a quicker swing towards the bottom. We expect this from the pendulum's natural dynamics at more extreme conditions.

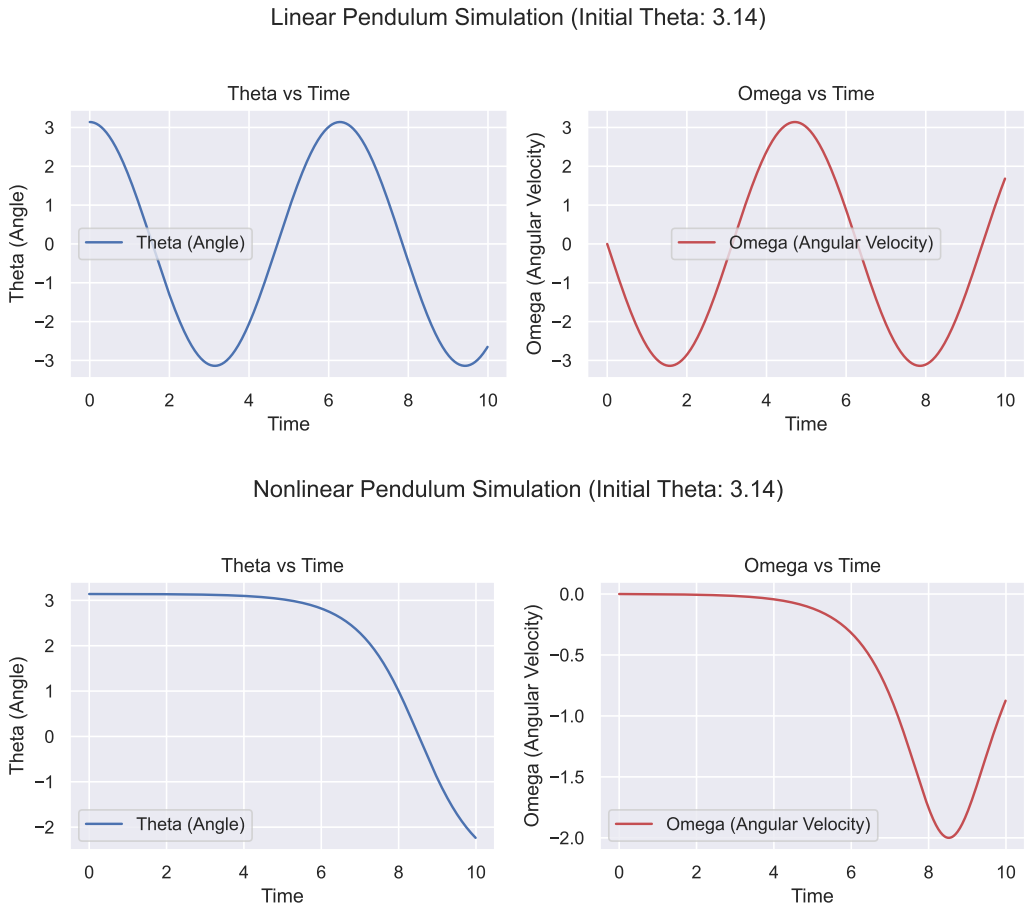

Fig. 3.4: Linear vs Nonlinear $(\theta, \omega) = (3.14, 0.0)$

This setup clearly shows the breakdown of the linear approximation when we stray from our small angle approximation.

3.2 Damped Pendulum Simulations

The system is made more complex by adding a damping force into the equation of motion for our nonlinear pendulum.

The damping force makes the system more realistic by simulating forces such as air resistance or friction.
The force is proportional to the angular velocity of the pendulum, and with a given coefficient of 0.5, we get a clear picture of its effects on the system's motion and decay.

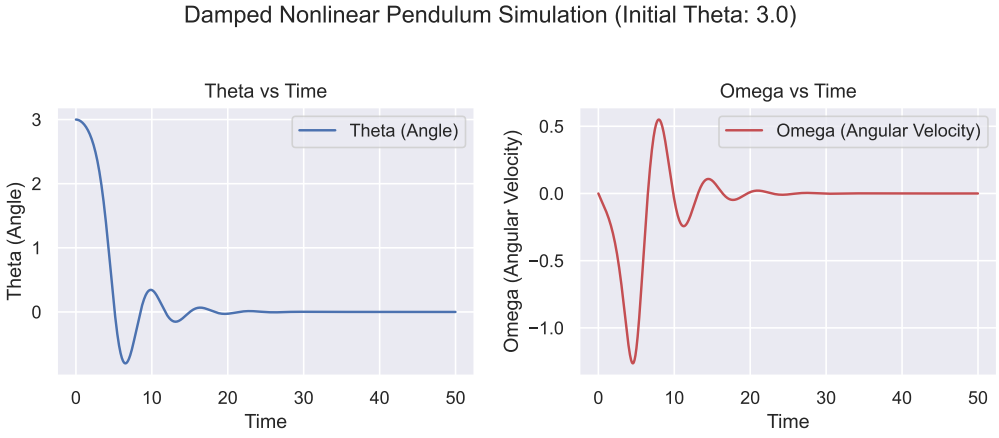

Fig. 3.5: Damped Nonlinear Pendulum $(\theta, \omega) = (3.0, 0.0)$

We see a big difference in this model from what we've seen in the undamped models.
In the damped pendulum, the amplitude of each swing decreases by about half each period. This behavior continues until the pendulum comes to rest at its equilibrium.

This method wasn't more complicated to implement than if we used an undamped pendulum compared to an analytical method, highlighting the utility of numerical methods for solving increasingly complex systems.

3.3 Phase Space Analysis

The phase space plots for A = 0.9 and A = 1.07 have periodic behavior, where the motion is consistent and predictable and is bounded by closed trajectories. The plots for A = 1.5, A = 1.47, and A = 1.35 show the system approaching chaos where the path is irregular and never repeats. These behaviors show the pendulum's transition from order to chaos as the amplitude of the driving force crosses the threshold, illustrating the sensitivity of initial conditions in dynamical systems.

Periodic Motion

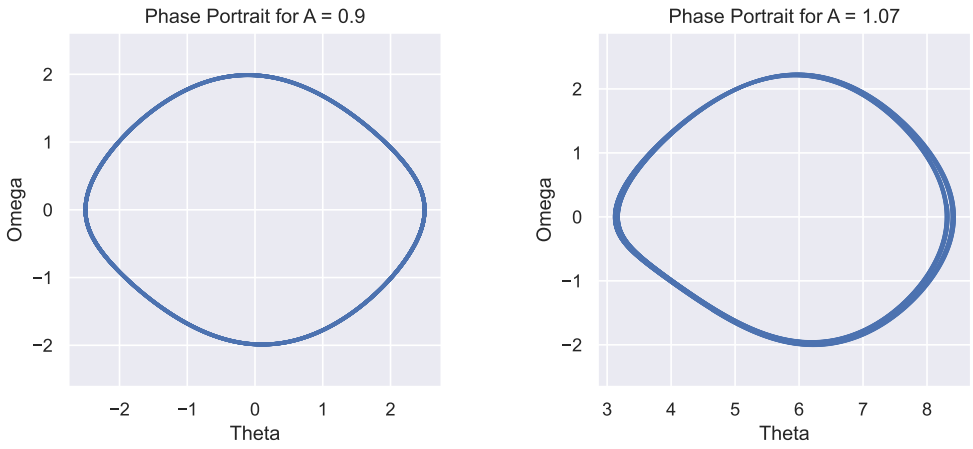

Fig. 3.6: Phase portraits with periodic motion

A = 0.9 This phase portrait shows a clear closed loop. This represents a stable periodic system. The plotted trajectory suggests that the motion of the systems is regular and predictable.

A = 1.07 Increasing the amplitude of the driving force to 1.07 keeps the system periodic, but it distorts the shape of the trajectory. This suggests that the system is moving towards more complex dynamics.

Chaotic Motion

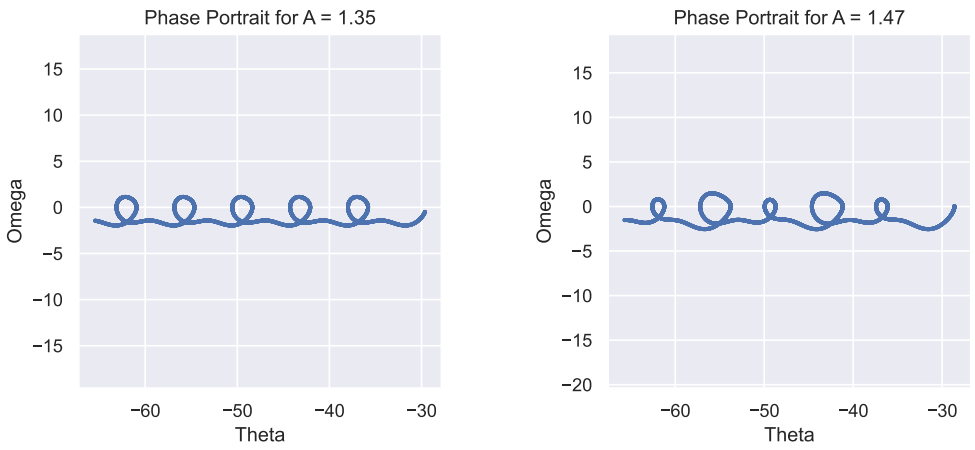

Fig. 3.7: Phase portraits approaching chaotic motion

A = 1.35 At this amplitude, the pendulum begins to cross over into chaotic motion. The phase space portrait shows loops moving to the left. This extended leftward motion signifies that the pendulum is swinging in loops anti-clockwise. However, the motion is still predictable.

A = 1.47 Further, increasing the amplitude of the driving force disrupts the order of the system. The loops moving to the left become less uniform. This is a sign that the system is moving further into chaos.

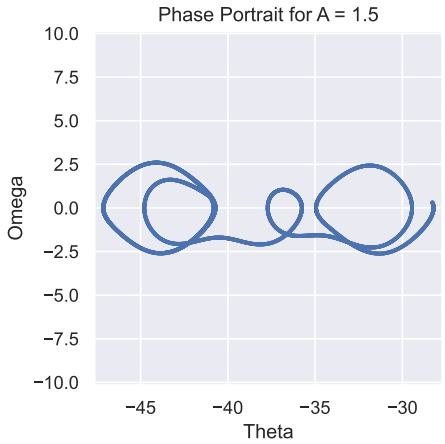

Fig. 3.8: Phase portrait with chaotic motion

A = 1.5 At our highest amplitude, the pendulum reaches the most chaotic state tested. The trajectory is irregular and shows no indication of repetition. At this point, long-term prediction becomes impossible.

3.4 Comparative Analysis of Numerical Methods

The three numerical methods, the Euler, Trapezoidal (Trap), and Runge-Kutta (RK4) methods, were tested and compared across a range of initial angular displacements with both linear and nonlinear models. This lets us see their performance and accuracy when simulating dynamic systems.

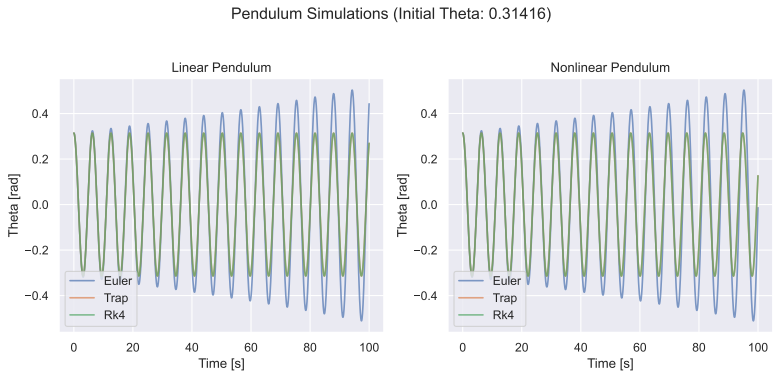

Fig. 3.9: Simple pendulum with low energy

Initial Theta: 0.31416 At the smallest initial displacement, all three methods perform very closely with small differences in the period. As the simulation carries on over several oscillations, we can see that the Euler method begins to fail as the maximum displacement increases each swing. The trap and RK4 methods match paths, keeping the same maximum displacement.

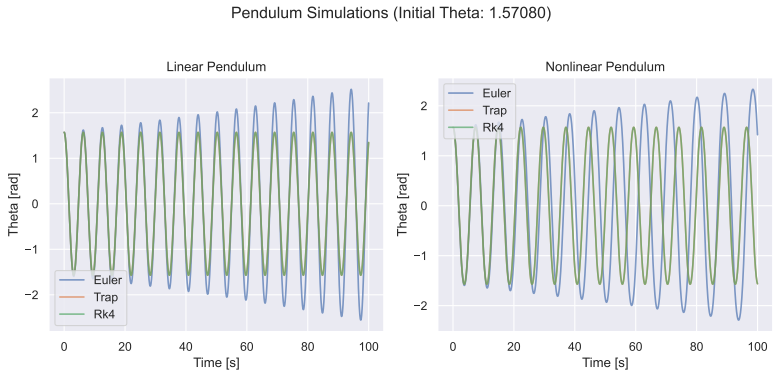

Fig. 3.10: Simple pendulum with moderate energy

Initial Theta: 1.57080 As the initial angle increases, we can see the Euler method begin to fail more, as the period begins to increase, as well as the maximum displacement, which is noticeable in the nonlinear model. The trap and RK4 methods again match paths, keeping the same maximum displacement.

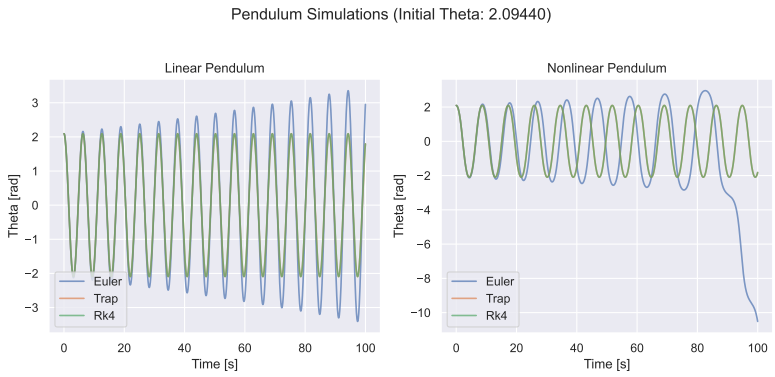

Fig. 3.11: Simple pendulum with high energy

Initial Theta: 2.09440 As the initial angle increases, the Euler method fails after about 90 seconds on the nonlinear model. The drop in the line signifies the pendulum is no longer swinging back and forth but looping in the anti-clockwise direction. The trap and RK4 methods seem to handle the large energy well.

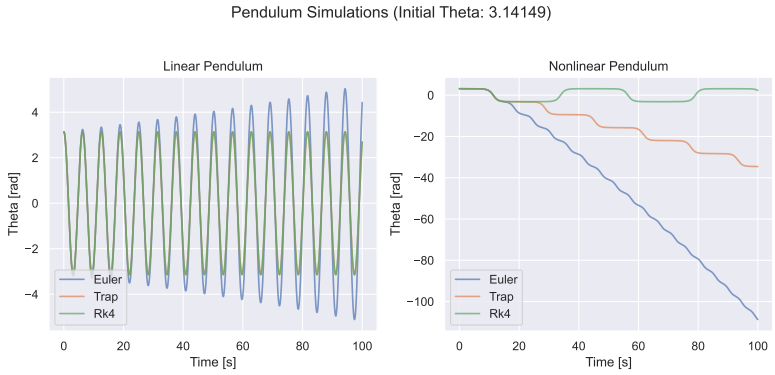

Fig. 3.12: Simple pendulum with very high energy

Initial Theta: 3.14149 Pushing the energy to the extreme, we see where the RK4 method is worth the extra computing power. At this point, the linear approximation has become entirely inaccurate, as established earlier. When looking at the nonlinear model, the Euler method diverges quickly, looping in an anti-clockwise direction. The trap method follows this behavior, although straying less slowly. The RK4 remains the most stable, handling extreme conditions well.

Discussion This comparative analysis clearly shows how the Runge-Kutta (RK4) method outperforms the Euler and Trapezoidal methods, especially in high-energy systems or over long periods of time. The Euler method requires much smaller time steps to be accurate, making it less reliable.
RK4 is stable and has high precision, even in high-energy systems. This comes at the cost of high computing power. The Euler method may be suitable and preferred for lower energy systems or simulations over short timescales due to requiring much less computing power. The Trapezoidal rule is an excellent middle ground, offering better accuracy than Euler's method without the computational intensity of RK4.

4 Conclusion

In the lab exercise, we looked at the dynamics of a pendulum using numerical simulations. We focused on the different aspects of motion under varying conditions. Beginning with the linear approximation of the pendulum then moving to the more complex systems containing nonlinear dynamics, damping forces, and driving forces. Each additional complexity layer gives us a better look at pendulum behavior and applications of numerical methods in real-world systems.

This lab looked at three numerical methods: Euler's method, the Trapezoidal Rule, and the Runge-Kutta method. Each method was applied to the different aspects of the lab, where appropriate. We examine the methods and compare their effectiveness and limitations over different cases. While computationally simple, Euler's method was less accurate, particularly when dealing with systems with high energy or looking at movement over long periods. The Trapezoidal Rule was a good balance between computational complexity and accuracy, which made it a reliable choice for most cases. When pushing initial conditions to the extreme, the Runge-Kutta method was ideal, with its high accuracy at a higher computation cost.

The linear and nonlinear models of the pendulum showed the utility and limitations of linear approximations in physics. The linear model worked well for small initial energies, but as the energy increased, the nonlinear model became needed to simulate the real motion of the pendulum. Introducing a damping force made the pendulum closer to the real-world case, as the motion eventually came to rest at an equilibrium. This damping force in a real system would be due to forces like friction and drag.

The phase space analysis showed the pendulum's transition from periodic to chaotic behavior over varying external forces. This provided a basic understanding of chaotic systems and how phase space analysis can be used to understand a system's motion better.

In conclusion, this lab exercise provided a broad view of pendulum dynamics using numerical simulations. It underlines the importance of using appropriate numerical methods depending on how fast the system is changing and how accurate you need the results to be. This basic pendulum system is a great starting point for analyzing dynamic systems, as when the complexity increases, we may still use the same techniques for examining the motion of the complex systems.

Future experimentation could include more complex systems or look at the use of these techniques in engineering applications.

References

1. Department of Physics, Trinity College Dublin, *SF Computational Lab Handbook*.
2. Freecodecamp.org, *Euler's Method Explained with Examples*.
3. Wikipedia, *Trapezoidal rule (differential equations)*.
4. Wikipedia, *Runge–Kutta methods*.
5. Documentation for NumPy, Matplotlib, and Seaborn.
6. Python Software Foundation, *Python Documentation*.

A Code

A.1 linear_pendulum.py

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os

sns.set_theme()

# Check and create a directory for plots
plot_dir = 'linear_plots'
if not os.path.exists(plot_dir):
    os.makedirs(plot_dir)

# Pendulum's equation of motion
def pendulum_motion(theta, omega, t, k=0.0, phi=0.66667, A=0.0):
    return -theta - k * omega + A * np.cos(phi * t)

# Function to solve and plot the pendulum motion using the trapezoidal rule
def solve_and_plot_pendulum(nsteps, initial_conditions, motion_func, plot_dir):
    theta, omega, t, dt = initial_conditions
    thetas, omegas, times = np.zeros(nsteps), np.zeros(nsteps), np.zeros(nsteps)

    for step in range(nsteps):
        thetas[step], omegas[step] = theta, omega
        times[step] = t

        # Trapezoidal rule
        k1_theta, k1_omega = dt * omega, dt * motion_func(theta, omega, t)
        k2_theta = dt * (omega + k1_omega)
        k2_omega = dt * motion_func(theta + k1_theta, omega + k1_omega, t + dt)

        theta += (k1_theta + k2_theta) / 2
        omega += (k1_omega + k2_omega) / 2
        t += dt

    # Plotting
    plt.figure(figsize=(9, 4))
    plt.subplot(1, 2, 1)
    plt.plot(times, thetas, label='Theta (Angle)')
    plt.xlabel('Time')
    plt.ylabel('Theta (Angle)')
    plt.title('Theta vs Time')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(times, omegas, label='Omega (Angular Velocity)', color='r')
    plt.xlabel('Time')
    plt.ylabel('Omega (Angular Velocity)')
```

```
48      plt.title('Omega vs Time')
49      plt.legend()
50
51      plt.suptitle(f"Linear Pendulum Simulation (Initial Theta: {
            initial_conditions[0]})")
52      plt.tight_layout(rect=[0, 0.03, 1, 0.95])
53      plt.savefig(f"{plot_dir}/pendulum_{initial_conditions[0]}.pdf")
54      plt.savefig(f"{plot_dir}/pendulum_{initial_conditions[0]}.png")
55      plt.show()
56
57  # Initial conditions
58  initial_conditions_list = [
59      #theta, omega, t, dt
60      (0.2, 0.0, 0.0, 0.01),
61      (1.0, 0.0, 0.0, 0.01),
62      (3.14, 0.0, 0.0, 0.01),
63      (0.0, 1.0, 0.0, 0.01)
64  ]
65
66  for initial_conditions in initial_conditions_list:
67      solve_and_plot_pendulum(1000, initial_conditions, pendulum_motion,
            plot_dir)
```

A.2 nonlinear_pendulum.py

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os

sns.set_theme()

# Check and create a directory for plots
plot_dir = 'nonlinear_plots'
if not os.path.exists(plot_dir):
    os.makedirs(plot_dir)

# Pendulum's equation of motion
def pendulum_motion(theta, omega, t, k=0.0, phi=0.66667, A=0.0):
    return -np.sin(theta) - k * omega + A * np.cos(phi * t)

# Function to solve and plot the pendulum motion using the trapezoidal rule
def solve_and_plot_pendulum(nsteps, initial_conditions, motion_func, plot_dir):
    theta, omega, t, dt = initial_conditions
    thetas, omegas, times = np.zeros(nsteps), np.zeros(nsteps), np.zeros(nsteps)

    for step in range(nsteps):
        thetas[step], omegas[step] = theta, omega
        times[step] = t

        # Trapezoidal rule
        k1_theta, k1_omega = dt * omega, dt * motion_func(theta, omega, t)
        k2_theta = dt * (omega + k1_omega)
        k2_omega = dt * motion_func(theta + k1_theta, omega + k1_omega, t + dt)

        theta += (k1_theta + k2_theta) / 2
        omega += (k1_omega + k2_omega) / 2
        t += dt

    # Plotting
    plt.figure(figsize=(9, 4))
    plt.subplot(1, 2, 1)
    plt.plot(times, thetas, label='Theta (Angle)')
    plt.xlabel('Time')
    plt.ylabel('Theta (Angle)')
    plt.title('Theta vs Time')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(times, omegas, label='Omega (Angular Velocity)', color='r')
    plt.xlabel('Time')
    plt.ylabel('Omega (Angular Velocity)')
    plt.title('Omega vs Time')
    plt.legend()
```

```
50
51      plt.suptitle(f"Nonlinear Pendulum Simulation (Initial Theta: {
            initial_conditions[0]})")
52      plt.tight_layout(rect=[0, 0.03, 1, 0.95])
53      plt.savefig(f"{plot_dir}/pendulum_{initial_conditions[0]}.pdf")
54      plt.savefig(f"{plot_dir}/pendulum_{initial_conditions[0]}.png")
55      plt.show()
56
57  # Initial conditions
58  initial_conditions_list = [
59      #theta, omega, t, dt
60      (0.2, 0.0, 0.0, 0.01),
61      (1.0, 0.0, 0.0, 0.01),
62      (3.14, 0.0, 0.0, 0.01),
63      (0.0, 1.0, 0.0, 0.01)
64  ]
65
66  for initial_conditions in initial_conditions_list:
67      solve_and_plot_pendulum(1000, initial_conditions, pendulum_motion,
            plot_dir)
```

A.3 nonlinear_damped_pendulum.py

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os

sns.set_theme()

# Check and create a directory for plots
plot_dir = 'damped_nonlinear_plots'
if not os.path.exists(plot_dir):
    os.makedirs(plot_dir)

# Pendulum's equation of motion
def pendulum_motion(theta, omega, t, k=0.5, phi=0.66667, A=0.00):
    return -np.sin(theta) - k * omega + A * np.cos(phi * t)

# Function to solve and plot the pendulum motion using the trapezoidal rule
def solve_and_plot_pendulum(nsteps, initial_conditions, motion_func, plot_dir):
    theta, omega, t, dt = initial_conditions
    thetas, omegas, times = np.zeros(nsteps), np.zeros(nsteps), np.zeros(nsteps)

    for step in range(nsteps):
        thetas[step], omegas[step] = theta, omega
        times[step] = t

        # Trapezoidal rule
        k1_theta, k1_omega = dt * omega, dt * motion_func(theta, omega, t)
        k2_theta = dt * (omega + k1_omega)
        k2_omega = dt * motion_func(theta + k1_theta, omega + k1_omega, t + dt)

        theta += (k1_theta + k2_theta) / 2
        omega += (k1_omega + k2_omega) / 2
        t += dt

    # Plotting
    plt.figure(figsize=(9, 4))
    plt.subplot(1, 2, 1)
    plt.plot(times, thetas, label='Theta (Angle)')
    plt.xlabel('Time')
    plt.ylabel('Theta (Angle)')
    plt.title('Theta vs Time')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(times, omegas, label='Omega (Angular Velocity)', color='r')
    plt.xlabel('Time')
    plt.ylabel('Omega (Angular Velocity)')
    plt.title('Omega vs Time')
    plt.legend()
```

```
50
51      plt.suptitle(f"Damped Nonlinear Pendulum Simulation (Initial Theta:
            {initial_conditions[0]})")
52      plt.tight_layout(rect=[0, 0.03, 1, 0.95])
53      plt.savefig(f"{plot_dir}/pendulum_{initial_conditions[0]}.pdf")
54      plt.savefig(f"{plot_dir}/pendulum_{initial_conditions[0]}.png")
55      plt.show()
56
57  # Initial conditions
58  initial_conditions_list = [
59      #theta, omega, t, dt
60      (3.0, 0.0, 0.0, 0.01),
61  ]
62
63  for initial_conditions in initial_conditions_list:
64      solve_and_plot_pendulum(5000, initial_conditions, pendulum_motion,
            plot_dir)
```

A.4 phase_portraits.py

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os

sns.set_theme()

# Check and create a directory for plots
plot_dir = 'phase_portraits'
if not os.path.exists(plot_dir):
    os.makedirs(plot_dir)

# Parameters
A_values = [0.90, 1.07, 1.35, 1.47, 1.5]
k = 0.5
phi = 0.66667
transient = 5000  # Number of iterations to skip as transient
nsteps = 10000  # Total number of steps
dt = 0.01  # Time step

# Pendulum's equation of motion
def f_damped_driven(theta, omega, t, A):
    return -np.sin(theta) - k * omega + A * np.cos(phi * t)

for A in A_values:
    theta = 0.2
    omega = 0.0
    t = 0.0
    iteration_number = 0

    theta_values = []
    omega_values = []
    last_theta = theta

    for step in range(nsteps):
        iteration_number += 1

        # Trapezoid rule
        k1a = dt * omega
        k1b = dt * f_damped_driven(theta, omega, t, A)
        k2a = dt * (omega + k1b)
        k2b = dt * f_damped_driven(theta + k1a, omega + k1b, t + dt, A)

        theta += (k1a + k2a) / 2
        omega += (k1b + k2b) / 2
        t += dt

        # Adjust theta to stop jumps in the phase portrait
        while np.abs(theta - last_theta) > np.pi:
            theta -= 2 * np.pi * np.sign(theta - last_theta)

        last_theta = theta  # Update last_theta for the next iteration

        if iteration_number > transient:
            theta_values.append(theta)
```

```
            omega_values.append(omega)

    # Plot the phase portrait
    plt.figure(figsize=(4, 4))
    plt.scatter(theta_values, omega_values, s=2, c='b')
    plt.xlabel('Theta')
    plt.ylabel('Omega')
    plt.title(f'Phase Portrait for A = {A}')
    plt.axis('equal')
    plt.tight_layout()
    plt.savefig(f"{plot_dir}/PhasePortrait_A_{A}.pdf")
    plt.savefig(f"{plot_dir}/PhasePortrait_A_{A}.png")
    plt.show()
```

A.5 method_comparison.py

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os

sns.set_theme()

# Parameters for pendulum
k = 0.0  # Damping
phi = 0.66667  # Driving force frequency
A = 0.0  # Driving force amplitude

# Pendulum's equations of motion
def f_linear(theta, omega, t):
    return -theta - k * omega + A * np.cos(phi * t)

def f_nonlinear(theta, omega, t):
    return -np.sin(theta) - k * omega + A * np.cos(phi * t)

# Initialize variables
initial_theta_values = [np.pi / 10, np.pi / 2, np.pi / 1.5, np.pi -
    0.0001]
dt = 0.01
nsteps = 10000

plot_dir = 'method_comparison_plots'
if not os.path.exists(plot_dir):
    os.makedirs(plot_dir)

# Function for simulation
def simulate_pendulum(f_motion, initial_theta, plot_dir, ax):
    time_vals = np.zeros(nsteps)
    theta_vals = {'euler': np.zeros(nsteps), 'trap': np.zeros(nsteps),
        'rk4': np.zeros(nsteps)}
    omega_vals = {'euler': 0.0, 'trap': 0.0, 'rk4': 0.0}

    theta_vals['euler'][0] = theta_vals['trap'][0] = theta_vals['rk4'
        ][0] = initial_theta
    time_vals[0] = 0.0

    # Simulation loop
    for i in range(1, nsteps):
        t = i * dt
        time_vals[i] = t

        # Euler's Method
        theta_vals['euler'][i] = theta_vals['euler'][i-1] + dt *
            omega_vals['euler']
        omega_vals['euler'] += dt * f_motion(theta_vals['euler'][i-1],
            omega_vals['euler'], t)

        # Trapezoidal Rule
        k1_theta, k1_omega = dt * omega_vals['trap'], dt * f_motion(
            theta_vals['trap'][i-1], omega_vals['trap'], t)
        k2_theta = dt * (omega_vals['trap'] + k1_omega)
```

```
        k2_omega = dt * f_motion(theta_vals['trap'][i-1] + k1_theta,
            omega_vals['trap'] + k1_omega, t + dt)
        theta_vals['trap'][i] = theta_vals['trap'][i-1] + (k1_theta +
            k2_theta) / 2
        omega_vals['trap'] += (k1_omega + k2_omega) / 2

        # Fourth-Order Runge-Kutta Method
        k1 = dt * omega_vals['rk4']
        l1 = dt * f_motion(theta_vals['rk4'][i-1], omega_vals['rk4'], t
            )
        k2 = dt * (omega_vals['rk4'] + l1 / 2)
        l2 = dt * f_motion(theta_vals['rk4'][i-1] + k1 / 2, omega_vals[
            'rk4'] + l1 / 2, t + dt / 2)
        k3 = dt * (omega_vals['rk4'] + l2 / 2)
        l3 = dt * f_motion(theta_vals['rk4'][i-1] + k2 / 2, omega_vals[
            'rk4'] + l2 / 2, t + dt / 2)
        k4 = dt * (omega_vals['rk4'] + l3)
        l4 = dt * f_motion(theta_vals['rk4'][i-1] + k3, omega_vals['rk4
            '] + l3, t + dt)
        theta_vals['rk4'][i] = theta_vals['rk4'][i-1] + (k1 + 2 * k2 +
            2 * k3 + k4) / 6
        omega_vals['rk4'] += (l1 + 2 * l2 + 2 * l3 + l4) / 6

    # Plotting
    for method in theta_vals:
        ax.plot(time_vals, theta_vals[method], label=method.capitalize
            (), alpha=0.7)
    ax.set_xlabel('Time [s]')
    ax.set_ylabel('Theta [rad]')
    ax.legend()
    ax.set_title(f'{plot_dir.capitalize()} Pendulum (Initial Theta: {
        initial_theta})')


for initial_theta in initial_theta_values:
    fig, axs = plt.subplots(1, 2, figsize=(10, 5))

    # Simulate and plot linear pendulum
    simulate_pendulum(f_linear, initial_theta, 'Linear', axs[0])
    axs[0].set_title('Linear Pendulum')

    # Simulate and plot nonlinear pendulum
    simulate_pendulum(f_nonlinear, initial_theta, 'Nonlinear', axs[1])
    axs[1].set_title('Nonlinear Pendulum')

    fig.suptitle(f'Pendulum Simulations (Initial Theta: {initial_theta
        :.5f})')

    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.savefig(f"{plot_dir}/{initial_theta}_comparison_plot.pdf")
    plt.savefig(f"{plot_dir}/{initial_theta}_comparison_plot.png")
    plt.show()
```