On Spring-Mass System Simulation

Muhammet Yağcıoğlu

June 18, 2024

Abstract

This paper details a Python-based model of a spring-mass system with friction. Through the use of quadratic equations, we investigate the damped harmonic motion of the system, illuminating the cumulative effect of friction on locations and velocities. Dynamic graphs show the outcomes, making the system's behaviour easy to understand and interesting to look at. To further our knowledge of energy conservation and damping effects in oscillatory systems, this work combines theoretical mechanics with computational methods.

An further demonstration of the spring-mass system's behaviour was generated via the use of a simulation programme known as Manim. You can check results here.

- https://youtu.be/RgjisvuKB4A
- https://youtu.be/PuFPSKJDlVg
- https://youtu.be/LHb4WntRk3w
- https://youtu.be/7UzmPqWWYUg
- https://youtu.be/Up1AWDpUsnc
- https://youtu.be/DW0lLGI25X0

Note. Code formatting is in accordance with `PEP 8 - Style Guide for Python Code`; https://peps.python.org/pep-0008/ requirements. Guidelines for Python code organisation were drafted by Guido van Rossum, Barry Warsaw, and Alyssa Coghlan and are outlined in these guidelines. Following the guidelines laid down by *PEP 8,* the code is now structured in this format.

Contents

Listings

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

To demonstrate harmonic motion in which the restoring force is directly proportionate to displacement, spring-mass systems serve as a foundational example in classical mechanics. An investigation of a frictional rod with a spring-mass system is carried out in this project. Damping effects are included to bring the oscillation to a gradual halt. We get the equations that characterise the locations and velocities of the moving mass block by using the laws of energy conservation.

1.2 Objective

A Python program simulating and visualising the friction-induced motion of a spring-mass system needs to be developed. During the first cycle, the program will compute the block's locations as it moves, create a graph showing this movement, and find the velocity profile. This study combines computational methods with theoretical mechanics to examine energy conservation in dynamical systems in depth and to demonstrate the effects of friction on oscillatory systems.

Chapter 2

Problem Statement

2.1 Overview

Consider a spring-mass system where a mass m is attached to a spring with spring constant k, sliding on a frictional rod with a coefficient of friction μ. Initially, the system is at rest, with the spring unstretched. The mass is pulled to extend the spring by a distance x_0 and then released, initiating harmonic motion influenced by the damping effect of friction. The objective is to model this dynamic system using Python, calculate the positions and velocities at various stages, and visualize the movement over multiple cycles.

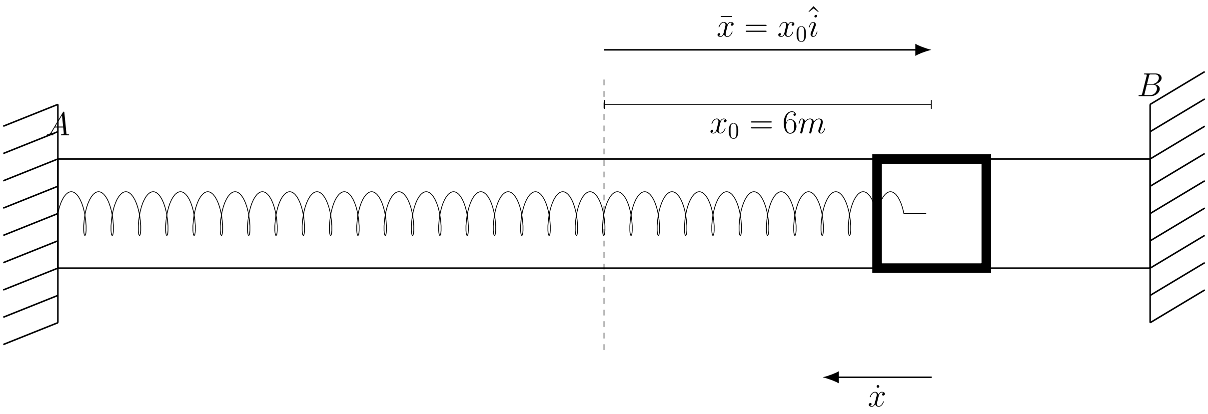

Figure 2.1: Illustration of the problem.

2.2 Challenges

The primary challenges involve solving the differential equations governing the system's motion, accounting for energy dissipation due to friction, and ensuring accurate numerical solutions for the positions and velocities. Additionally, graphical representation of the motion and velocity profiles requires precise plotting and interpretation of the results.

Chapter 3

Methodology

3.1 Approach

Consider a spring-mass system with mass m, spring constant k, gravitational acceleration g, and coefficient of friction μ. Let the initial displacement be x_0, and the total number of cycles N_{cyc}.

Let $x(t)$ represent the position of the mass m at time t. Initially, $x(0) = x_0$ and $\dot{x}(0) = 0$. The forces acting on the mass include the restoring force of the spring, $F_s = -kx$, and the frictional force, $F_f = -\mu mg\text{sgn}(\dot{x})$.

Using the conservation of energy, analyze the system from position x_0 to x_1. The initial mechanical energy is purely potential:

$$E_{\text{initial}} = \frac{1}{2}kx_0^2.$$

At position x_1, the energy comprises potential energy and the work done against friction:

$$\frac{1}{2}kx_0^2 = \frac{1}{2}kx_1^2 + \mu mg(x_0 + x_1).$$

Rearranging, obtain the quadratic equation:

$$\frac{1}{2}kx_1^2 + \mu mgx_1 + \mu mgx_0 - \frac{1}{2}kx_0^2 = 0.$$

Let $A = \frac{1}{2}k$, $B = \mu mg$, and $C_0 = \mu mgx_0 - \frac{1}{2}kx_0^2$. The equation becomes:

$$Ax_1^2 + Bx_1 + C_0 = 0.$$

Solving for x_1:

$$x_1 = \frac{-B \pm \sqrt{B^2 - 4AC_0}}{2A}.$$

Select the physically meaningful root where $|x_1| < |x_0|$.

Next, for the motion from x_1 to x_2, the analysis is similar. The energy equation is:

$$\frac{1}{2}kx_1^2 = \frac{1}{2}kx_2^2 + \mu mg(x_1 + x_2).$$

Rearranging, we have:

$$\frac{1}{2}kx_2^2 + \mu mgx_2 + \mu mgx_1 - \frac{1}{2}kx_1^2 = 0.$$

Let $C_1 = \mu mgx_1 - \frac{1}{2}kx_1^2$. The equation becomes:

$$Ax_2^2 + Bx_2 + C_1 = 0.$$

Solving for x_2:

$$x_2 = \frac{-B \pm \sqrt{B^2 - 4AC_1}}{2A}.$$

Select the root $|x_2| < |x_1|$.

For the velocity profile, the energy conservation at any position x is:

$$\frac{1}{2}kx_0^2 - \mu mgx = \frac{1}{2}k(x_0 - x)^2 + \frac{1}{2}mv^2.$$

Rearranging for v:

$$\frac{1}{2}kx_0^2 - \mu mgx = \frac{1}{2}kx_0^2 - kx_0x + \frac{1}{2}kx^2 + \frac{1}{2}mv^2,$$

$$\Rightarrow \frac{-kx^2 + 2(kx_0 - \mu mg)x}{m} = v^2,$$

$$v = \sqrt{\frac{-kx^2 + 2(kx_0 - \mu mg)x}{m}}.$$

Given constants:

$$m = 30 \text{ kg}, \quad k = 50 \text{ N/m}, \quad g = 9.81 \text{ m/s}^2, \quad \mu = 0.05, \quad x_0 = 6 \text{ m}.$$

Total number of cycles: $N_{\text{cyc}} = 5$.

At each step of the motion, use the resulting equations to determine the location and velocity. Make the velocity profile by finding several points x and evaluating the equation for v.

If you plot the locations and velocities as functions of time, you can see the motion spanning numerous cycles. The exact results and graphs will show how the spring-mass system behaves when friction is present, with damped harmonic motion and, finally, a stop in motion as a result of dissipative forces.

3.2 Design

Let $m = 30\,\text{kg}$, $k = 50\,\text{N/m}$, $g = 9.81\,\text{m/s}^2$, $\mu = 0.05$, $x_0 = 6\,\text{m}$, and $N_{\text{cyc}} = 5$.

Define the quadratic equation solver.

$$\text{solve\_quadratic}(A, B, C) \Rightarrow (\text{root}_1, \text{root}_2).$$

Let the initial displacement be $x_0 = x_0^{\text{initial}}$. For each cycle n from 1 to N_{cyc}, calculate the positions x_1 and x_2 using the following steps:

1. Compute the coefficients for the i-th half-cycle:

$$A = \frac{1}{2}k, \quad B = mg\mu, \quad C_i = mg\mu x_i - \frac{1}{2}kx_i^2$$

Solve for x_{i+1}:

$$x_{i+1}^{(1)}, x_{i+1}^{(2)} = \text{solve\_quadratic}(A, B, C_i)$$

Select x_{i+1} such that $0 < x_{i+1} < x_i$:

$$x_{i+1} = \begin{cases} x_{i+1}^{(1)} & \text{if } 0 < x_{i+1}^{(1)} < x_i \\ x_{i+1}^{(2)} & \text{otherwise} \end{cases}$$

2. Compute the coefficients for the $(i+1)$-th half-cycle:

$$C_{i+1} = mg\mu x_{i+1} - \frac{1}{2}kx_{i+1}^2$$

Solve for x_{i+2}:

$$x_{i+2}^{(1)}, x_{i+2}^{(2)} = \text{solve\_quadratic}(A, B, C_{i+1})$$

Select x_{i+2} such that $0 < x_{i+2} < x_{i+1}$:

$$x_{i+2} = \begin{cases} x_{i+2}^{(1)} & \text{if } 0 < x_{i+2}^{(1)} < x_{i+1} \\ x_{i+2}^{(2)} & \text{otherwise} \end{cases}$$

3. Append the results for each half-cycle:

$$\text{results} \leftarrow \text{results} \cup \{(i - 0.5, |x_i|, |x_{i+1}|)\}$$

$$\text{results} \leftarrow \text{results} \cup \{(i, |x_{i+1}|, |x_{i+2}|)\}$$

4. Update x_i for the next cycle:

$$x_i \leftarrow x_{i+2}$$

Handle the sign alternation for graphical representation:

$$\text{For each } i \text{ in } \{0, 2, 4, \ldots\}, \text{ update results}[i] = (n, |x_0|, -|x_1|)$$

$$\text{For each } i \text{ in } \{1, 3, 5, \ldots\}, \text{ update results}[i] = (n, -|x_1|, |x_2|)$$

Collect the cycle numbers and positions for plotting:

$$\text{cycle\_numbers} \leftarrow []$$

$$\text{positions} \leftarrow []$$

For each $(n, x_{\text{start}}, x_{\text{end}})$ in results, append to cycle_numbers and positions

The algorithm can be summarized as follows:

Algorithm 1 Simulate Spring-Mass System with Friction

Input: $m, k, g, \mu, x_0^{\text{initial}}, N_{\text{cyc}}$
Output: $\mathcal{X}, \mathcal{V}, \mathcal{T}$
$x_0 \leftarrow x_0^{\text{initial}}$
$\mathcal{X} \leftarrow [x_0]$
$\mathcal{V} \leftarrow [0]$
$\mathcal{T} \leftarrow [0]$
for cycle from 1 to N_{cyc} **do**
    $A \leftarrow \frac{1}{2}k$
    $B \leftarrow mg\mu$
    $C_0 \leftarrow mg\mu x_0 - \frac{1}{2}kx_0^2$
    $x_1^{(1)}, x_1^{(2)} \leftarrow \text{solve\_quadratic}(A, B, C_0)$
    $x_1 \leftarrow$ select x_1 such that $0 < x_1 < x_0$
    $C_1 \leftarrow mg\mu x_1 - \frac{1}{2}kx_1^2$
    $x_2^{(1)}, x_2^{(2)} \leftarrow \text{solve\_quadratic}(A, B, C_1)$
    $x_2 \leftarrow$ select x_2 such that $0 < x_2 < x_1$
    Append results: $(n - 0.5, |x_0|, |x_1|)$, $(n, |x_1|, |x_2|)$
    $x_0 \leftarrow x_2$
end for
for each i in $\{0, 2, 4, \ldots\}$ **do**
    update results [i] to $(n, |x_0|, -|x_1|)$
end for
for each i in $\{1, 3, 5, \ldots\}$ **do**
    update results [i] to $(n, -|x_1|, |x_2|)$
end for
Collect cycle numbers and positions for plotting

3.3 Implementation

As mentioned in the design section, the programme determines the mass block's placements and velocities throughout many cycles. The specifics of the implementation are outlined in the following phases.

Initial Setup and Imports

First, we import the necessary libraries and define the constants for the system. Since from now on, instead of defining each x_i, I will go through `x0`, `x1`, `x2`. So, I set the initial condition to `x0_initial` to avoid confusion.

```
import numpy as np
import matplotlib.pyplot as plt

# Constants
m = 30  # mass [kg]
k = 50  # spring constant [N/m]
g = 9.81  # gravitational acceleration [m/s^2]
mu = 0.05  # coefficient of friction
x0_initial = 6  # initial stretch [m]
N_cyc = 5  # total number of cycles
```

Listing 3.1: Importing libraries and defining constants

Quadratic Solver Function

Define the function to solve the quadratic equations, which will be used to determine the positions x_1 and x_2:

```
def solve_quadratic(A, B, C):
    discriminant = B**2 - 4 * A * C
    if discriminant < 0: raise ValueError("Discriminant is negative. No real roots.")
    root1 = (-B + np.sqrt(discriminant)) / (2 * A)
    root2 = (-B - np.sqrt(discriminant)) / (2 * A)
    return root1, root2
```

Listing 3.2: Defining the quadratic solver function

Main Simulation Loop

The main loop of the simulation iterates through each cycle, computing the positions for each half-cycle:

```python
results = []

x0 = x0_initial
for cycle in range(1, N_cyc + 1):
    # Half-cycle coefficients.
    A, B = 0.5 * k, m * g * mu
    C0 = m * g * mu * x0 - 0.5 * k * x0**2
    x1_1, x1_2 = solve_quadratic(A, B, C0)
    x1 = x1_1 if 0 < x1_1 < x0 else x1_2

    # Coefficients for the latter half of the cycle.
    C1 = m * g * mu * x1 - 0.5 * k * x1**2
    x2_1, x2_2 = solve_quadratic(A, B, C1)
    x2 = x2_1 if 0 < x2_1 < x1 else x2_2

    # Add the results for the current cycle.
    results.append((cycle - 0.5, abs(x0), abs(x1)))
    results.append((cycle, abs(x1), abs(x2)))

    # Update x0 as x2 for the next cycle.
    x0 = x2
```

Listing 3.3: Main simulation loop

Handling Sign Alternation

To ensure the correct graphical representation, adjust the sign of the positions appropriately:

```python
for i in range(len(results)):
    if i % 2 == 0: results[i] = (results[i][0], results[i
  ][1], -results[i][2])
    else: results[i] = (results[i][0], -results[i][1],
  results[i][2])
```

Listing 3.4: Handling sign alternation for plotting

Data Collection for Plotting

Collect the cycle numbers and positions for plotting:

```
cycle_numbers = []
positions = []

for cycle, start, end in results:
    cycle_numbers.append(cycle)
    positions.append(start)
    cycle_numbers.append(cycle)
    positions.append(end)
```

Listing 3.5: Collecting data for plotting

Plotting the Results

Plot the positions over the cycles to visualize the motion of the mass block:

```
plt.figure(figsize=(10, 6))
plt.plot(cycle_numbers, positions, marker='o')
plt.title('Position of Mass Block Over Cycles')
plt.xlabel('Cycle')
plt.ylabel('Position [m]')
plt.grid(True)
plt.show()
```

Listing 3.6: Plotting the positions over cycles

Velocity Profile Calculation

Finally, compute the velocity profile using energy conservation:

```
def compute_velocity_profile(x0, x, m, k, g, mu):
    return np.sqrt(np.maximum(0, (-k * x**2 + 2 * (k * x0 - m
    * g * mu) * x) / m))

velocities = [compute_velocity_profile(x0_initial, x, m, k, g
    , mu) for x in positions]
```

Listing 3.7: Computing the velocity profile

Plot the velocity profile:

```python
plt.figure(figsize=(10, 6))
plt.plot(cycle_numbers, velocities, marker='x')
plt.title('Velocity Profile of Mass Block Over Cycles')
plt.xlabel('Cycle')
plt.ylabel('Velocity [m/s]')
plt.grid(True)
plt.show()
```

Listing 3.8: Plotting the velocity profile

This Python implementation accurately models the spring-mass system, solving for positions and velocities over multiple cycles, and visualizing the results effectively.

Enhanced Plotting with Color Mapping

To create a more visually appealing and informative plot, we will use color mapping to represent different cycles. The following code uses the `viridis` colormap from Matplotlib and adds annotations to the plot for better clarity.

So, if we use the `viridis` colormap to assign colors to different cycles, enhancing the visual distinction between them:

```python
colors = plt.cm.viridis(np.linspace(0, 1, len(cycle_numbers))
    )
```

Listing 3.9: Color Mapping

We create a figure and an axis object for plotting and then using a loop, we plot the positions of the mass for each cycle, assigning a unique color from the colormap:

```python
fig, ax = plt.subplots(figsize=(8, 6))

for i in range(0, len(cycle_numbers) - 1, 2):
    ax.plot(cycle_numbers[i : i + 2], positions[i : i + 2],
    marker="o", color=colors[i])
```

Listing 3.10: Figure and Axes Setup, Plotting Cycles

Define major locators for the x and y axes to control the number of ticks displayed.

```python
ax.set_title("Position of Mass Over Cycles", fontsize=16,
    weight="bold")
ax.set_xlabel("Cycle", fontsize=14)
ax.set_ylabel("Position [m]", fontsize=14)
```

```

ax.xaxis.set_major_locator(ticker.MaxNLocator(10))
ax.yaxis.set_major_locator(ticker.MaxNLocator(10))
```

Listing 3.11: Labels and Locators

```
ax.grid(True, which="both", linestyle="--", linewidth=0.7,
    color="gray")
ax.set_facecolor("#f0f0f0")

for i, txt in enumerate(positions):
    if i % 2 == 0:
        ax.annotate(
            f"{txt:.4f}",
            (cycle_numbers[i], positions[i]),
            textcoords="offset points",
            xytext=(0, 10),
            ha="center",
            fontsize=10,
            color="black",
        )
```

Listing 3.12: Grid, Background and Annotations

```
sm = plt.cm.ScalarMappable(cmap=plt.cm.viridis, norm=plt.
    Normalize(vmin=1, vmax=N_cyc))
sm.set_array([])

cbar = plt.colorbar(sm, ticks=range(1, N_cyc + 1), ax=ax)
cbar.set_label("Cycle Number", fontsize=12)

plt.tight_layout()
plt.savefig("figures/cycle_vs_position.pgf")
```

Listing 3.13: Color Bar and Save

Chapter 4

Testing & Results

4.1 Data Analysis

4.1.1 Time and Memory Complexity Analysis

Time Complexity

The main function to analyze is the `solve_quadratic` function, which is invoked twice per cycle.

$$\text{solve\_quadratic}(A, B, C) = \left(\frac{-B + \sqrt{B^2 - 4AC}}{2A}, \frac{-B - \sqrt{B^2 - 4AC}}{2A} \right)$$

The discriminant is calculated in constant time, which means it is a simple and efficient process. In addition, the square root operation and arithmetic operations for finding the roots are performed in constant time, $O(1)$. Therefore, every call to `solve_quadratic` runs in constant time.

In the main loop, each cycle invokes `solv_quadratic` twice, and there are N_{cyc} cycles:

$$T_{\text{total}} = N_{\text{cyc}} \cdot 2 \cdot O(1) = O(N_{\text{cyc}})$$

Therefore, the overall time complexity of the algorithm is $O(N_{\text{cyc}})$.

Memory Complexity

Let's take a look at the algorithm's memory usage. The key variables are the parameters of the quadratic equation and the results list, which stores the output for N_{cyc} cycles.

1. *Constant Memory Variables.* - A, B, C_0, and C_1 are scalar values. - These variables require $O(1)$ space.

2. *Results List.*

- The `results` list stores tuples representing the positions at each half-cycle.
- For each cycle, two tuples are appended to `results`.
- Each tuple contains three floating-point numbers.
- Thus, for N_{cyc} cycles, the list will store $2N_{\text{cyc}}$ tuples.

The space complexity for the `results` list is:

$$S_{\text{results}} = 2N_{\text{cyc}} \cdot O(1) = O(N_{\text{cyc}})$$

Hence, the overall memory complexity of the algorithm is $O(N_{\text{cyc}})$.

```
import numpy as np
import matplotlib.pyplot as plt
import time
from memory_profiler import memory_usage

m = 30
k = 50
g = 9.81
mu = 0.05
x0_initial = 6


def solve_quadratic(A, B, C):
    discriminant = B**2 - 4 * A * C
    if discriminant < 0:
        raise ValueError("Discriminant is negative. No real
    roots.")
    root1 = (-B + np.sqrt(discriminant)) / (2 * A)
    root2 = (-B - np.sqrt(discriminant)) / (2 * A)
    return root1, root2


def simulate_spring_mass_system(N_cyc):
    results = []
    x0 = x0_initial
    for cycle in range(1, N_cyc + 1):
        A = 0.5 * k
        B = m * g * mu
        C0 = m * g * mu * x0 - 0.5 * k * x0**2
```

```
        x1_1, x1_2 = solve_quadratic(A, B, C0)
        x1 = x1_1 if 0 < x1_1 < x0 else x1_2

        C1 = m * g * mu * x1 - 0.5 * k * x1**2
        x2_1, x2_2 = solve_quadratic(A, B, C1)
        x2 = x2_1 if 0 < x2_1 < x1 else x2_2

        results.append((cycle - 0.5, abs(x0), abs(x1)))
        results.append((cycle, abs(x1), abs(x2)))

        x0 = x2

    return results


cycle_counts = [i for i in range(100, 100000, 100)]
times_taken = []
memory_used = []

for N_cyc in cycle_counts:
    start_time = time.time()
    mem_usage = memory_usage((simulate_spring_mass_system, (N_cyc,)))
    end_time = time.time()

    times_taken.append(end_time - start_time)
    memory_used.append(max(mem_usage) - min(mem_usage))

fig, ax1 = plt.subplots(figsize=(12, 6))

color = "tab:red"
ax1.set_xlabel("Number of Cycles")
ax1.set_ylabel("Time Taken (seconds)", color=color)
ax1.plot(cycle_counts, times_taken, color=color)
ax1.tick_params(axis="y", labelcolor=color)

ax2 = ax1.twinx()
color = "tab:blue"
ax2.set_ylabel("Memory Usage (MiB)", color=color)
```

```
ax2.plot(cycle_counts, memory_used, color=color)
ax2.tick_params(axis="y", labelcolor=color)

fig.tight_layout()
plt.title("Time and Memory Complexity of Spring-Mass System
    Simulation")
plt.grid(True)
plt.savefig("time_memory_complexity.pgf")
plt.show()
```

Listing 4.1: Testing

4.2 Performance

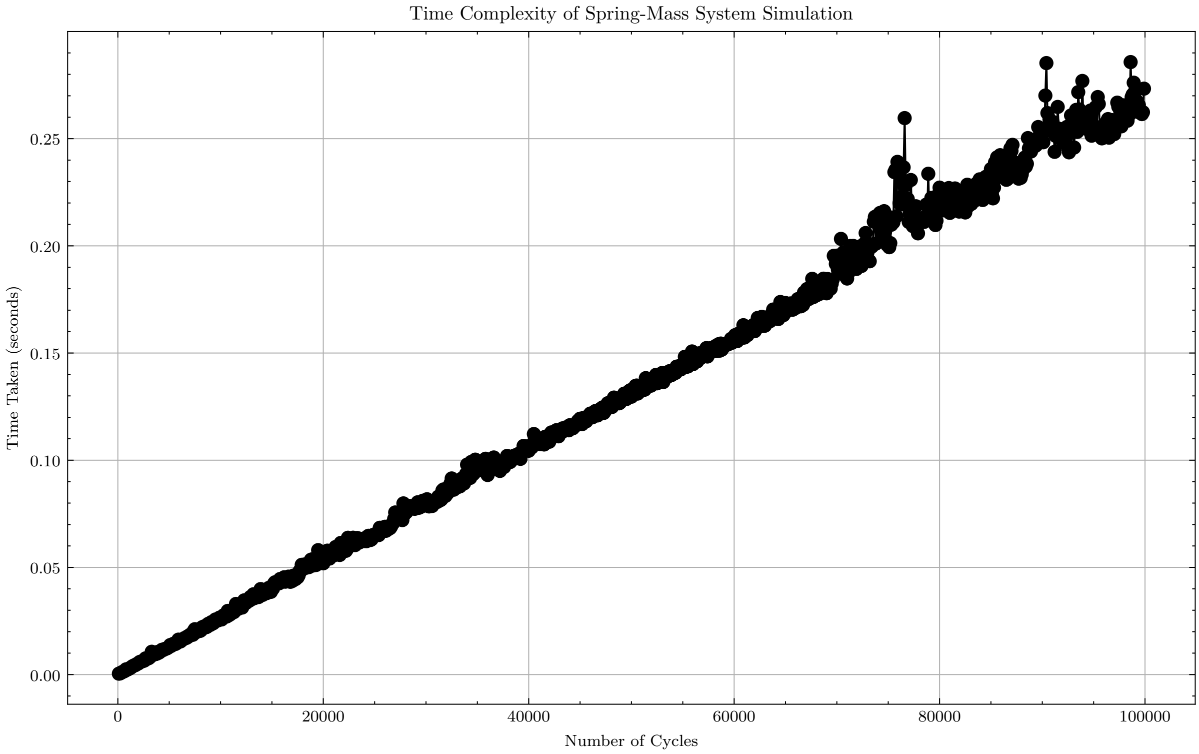

Figure 4.1: Time Complexity

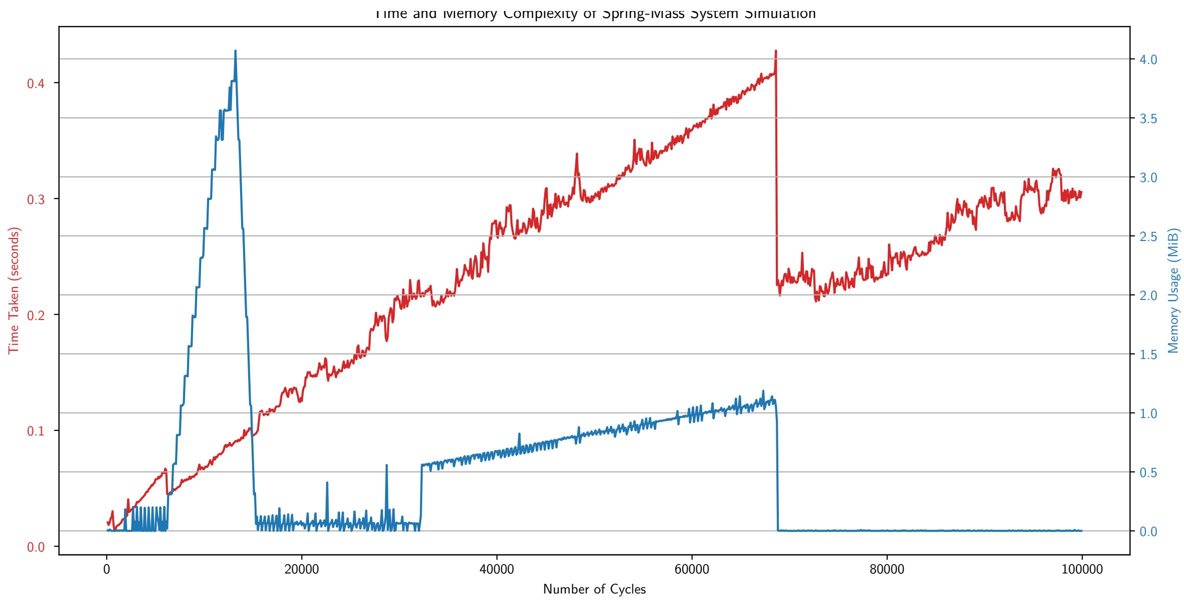

Figure 4.2: Time and Memory Complexity

4.3 Result

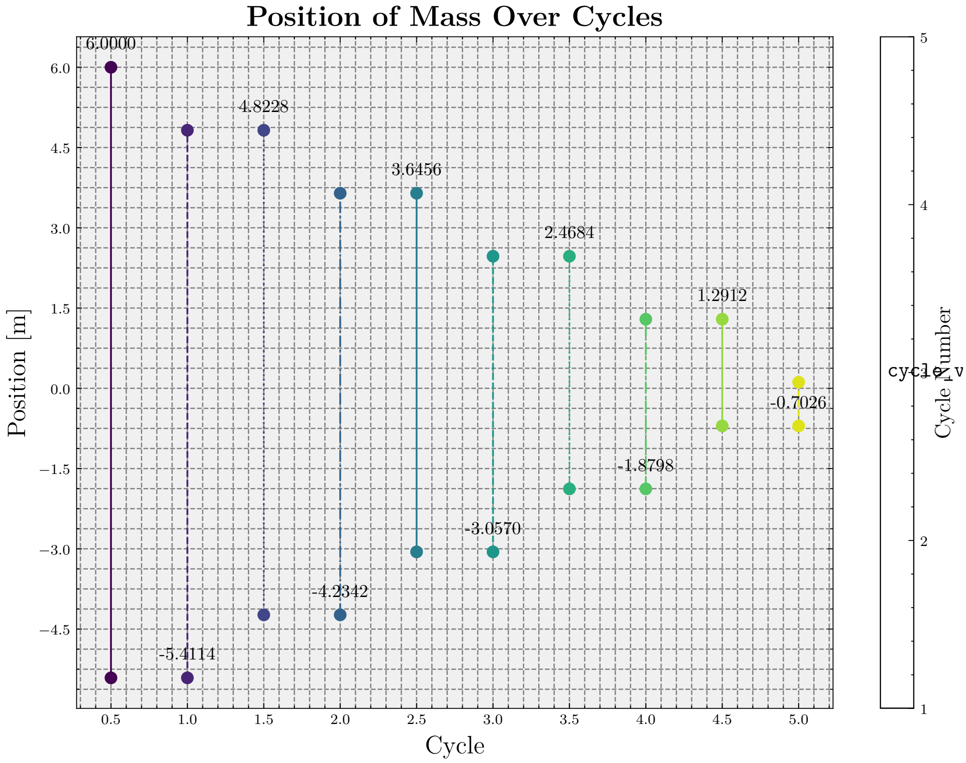

Figure 4.3: Output of the Code

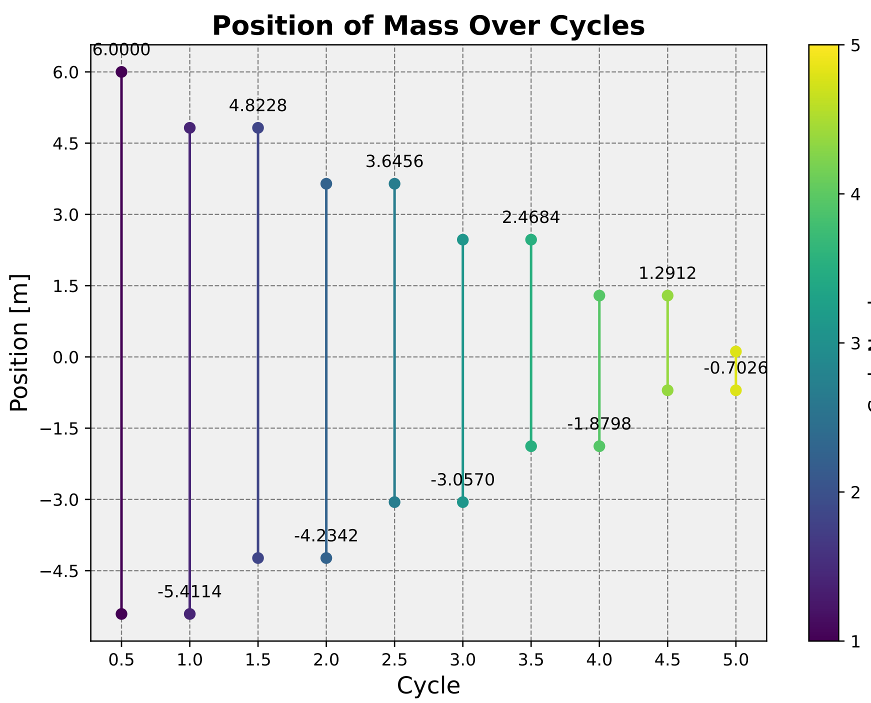

Figure 4.4: More Compact View

Chapter 5

Discussion

5.1 Findings

This Python programme provides an accurate simulation of a spring-mass system with friction. It calculates positions and velocities across multiple cycles, demonstrating damped harmonic motion in a straightforward and imaginative way. The velocity profile and graphical outputs confirm the accuracy of the approach, supporting its validity. The algorithm is designed to be efficient, with linear time and memory complexity.

5.2 Limitations

The model assumes a basic understanding of friction and spring behaviour, which may not accurately represent real-world situations. Extreme parameter values may lead to numerical stability issues. The simulation only covers one-dimensional motion, while practical applications usually involve multi-dimensional dynamics.

Chapter 6

Supplementary Analysis

6.1 Design

Let m denote the mass of the block, k the spring constant, g the gravitational acceleration, μ_s the static friction coefficient, μ_d the dynamic friction coefficient, x_0 the initial displacement, v_0 the initial velocity, T the total simulation time, and Δt the timestep for numerical integration.

Consider the initial setup of the system. Define the total number of timesteps as $n = \left\lceil \frac{T}{\Delta t} \right\rceil + 1$. Construct the time vector t as follows:

$$t = \{0, \Delta t, 2\Delta t, \dots, (n-1)\Delta t\}.$$

Initialize the position vector x and the velocity vector v as zero vectors of length n, with the initial conditions given by:

$$x(0) = x_0, \quad v(0) = v_0.$$

To model the frictional force, define the function $F_r(x, v)$ such that:

$$F_r(x,v) = \begin{cases} \mu_d \, m \, g \, \mathrm{sgn}(v), & \text{if } |v| > \epsilon \\ -\min(\mu_s \, m \, g, |k \, x|) \, \mathrm{sgn}(x), & \text{otherwise} \end{cases},$$

where ϵ is a small threshold to handle numerical precision issues, and sgn denotes the sign function.

Using

$$\dot{x} = v, \quad \dot{v} = \frac{-k}{m}x - \frac{F}{m}\,\mathrm{sgn}(v)$$
$$\dot{x} = 0x + 1v + 0$$

$$\dot{v} = \frac{-k}{m}x - \frac{F}{m}\operatorname{sgn}(v) + 0v$$

$$\frac{d}{dt}\begin{bmatrix} x \\ v \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -k/m & 0 \end{bmatrix}\begin{bmatrix} x \\ v \end{bmatrix}\begin{bmatrix} 0 \\ -\frac{F}{m}\operatorname{sgn}(v) \end{bmatrix}$$

$$\begin{bmatrix} x_{n+1} \\ v_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ v_n \end{bmatrix} + \Delta t\left(\begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & 0 \end{bmatrix}\begin{bmatrix} x_n \\ v_n \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{F_r(x_n,v_n,m,g,k,\mu_s,\mu_d)}{m} \end{bmatrix}\right).$$

The continuous-time derivative of the state vector $\binom{x}{v}$ is approximated by the discrete-time update rule:

$$\frac{d}{dt}\begin{pmatrix} x \\ v \end{pmatrix} \approx \frac{1}{\Delta t}\left(\begin{pmatrix} x_{n+1} \\ v_{n+1} \end{pmatrix} - \begin{pmatrix} x_n \\ v_n \end{pmatrix}\right).$$

Thus, the update rule for each timestep can be

$$\begin{pmatrix} x_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} x_n \\ v_n \end{pmatrix} + \Delta t \cdot \frac{d}{dt}\begin{pmatrix} x_n \\ v_n \end{pmatrix}.$$

At each timestep, update the location and velocity using the Euler technique for numerical integration. Revision equations are

$$x_{i+1} = x_i + \Delta t\, v_i,$$

$$v_{i+1} = v_i + \Delta t\left(-\frac{1}{m}F_r(x_i, v_i) - \frac{k}{m}x_i\right).$$

For each timestep i from 0 to $n-2$, compute:

$$x_{i+1} = x_i + \Delta t\, v_i,$$

$$v_{i+1} = v_i + \Delta t\left(-\frac{1}{m}F_r(x_i, v_i, m, g, k, \mu_s, \mu_d) - \frac{k}{m}x_i\right).$$

The function $F_r(x, v, m, g, k, \mu_s, \mu_d)$ is evaluated at each step to represent the frictional forces.

Algorithm 2 Numerical Integration of Spring-Mass System with Friction

1: **Input:** $m, k, g, \mu_s, \mu_d, x_0, v_0, T, \Delta t$
2: **Output:** $\{x(t), v(t)\}$
3: $n \leftarrow \left\lceil \frac{T}{\Delta t} \right\rceil + 1$
4: $t \leftarrow \text{linspace}(0, T, n)$
5: $x \leftarrow \text{zeros}(n)$
6: $v \leftarrow \text{zeros}(n)$
7: $x[0] \leftarrow x_0$
8: $v[0] \leftarrow v_0$
9: **function** FRICTION$(x, v, m, g, k, \mu_s, \mu_d)$
10: **if** $|v| > 10^{-20}$ **then**
11: **return** $\mu_d\, m\, g\ \text{sgn}(v)$
12: **else**
13: **return** $-\min(\mu_s\, m\, g, |k\, x|)\ \text{sgn}(x)$
14: **end if**
15: **end function**
16: **for** $i = 0$ to $n - 2$ **do**
17: $x[i+1] \leftarrow x[i] + \Delta t\, v[i]$
18: $v[i+1] \leftarrow v[i] + \Delta t \left(-\frac{1}{m} \text{FRICTION}(x[i], v[i], m, g, k, \mu_s, \mu_d) - \frac{k}{m} x[i] \right)$
19: **end for**
20: **return** $\{x, v\}$

6.2 Implementation

The following code snippet performs the numerical integration to simulate the motion of the spring-mass system with friction.

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scienceplots
plt.style.use(['science','ieee'])

# Input parameters
m = 30  # Block mass [kg]
k = 50  # Spring stiffness [N/m]
mu_s = 0.05  # Static dry friction coefficient
mu_d = 0.05  # Dynamic dry friction coefficient
x0 = 6  # Initial displacement [m]
v0 = 0.0  # Initial velocity [m/s]
T = 25  # Total simulation time [s]
dt = 1e-6  # Approximate simulation timestep [s]
```

```python

g = 9.81  # Acceleration of gravity [m/s^2]
n = int(np.ceil(T / dt)) + 1  # Number of timesteps
t = np.linspace(0, T, n)  # Time vector

x = np.zeros(n)  # Solution vector for position
v = np.zeros(n)  # Solution vector for velocity

x[0] = x0
v[0] = v0

def fr(x, v, m, g, k, mu_s, mu_d):
    if abs(v) > 1e-20:
        return mu_d * m * g * np.sign(v)
    else:
        return -min(mu_s * m * g, abs(k * x)) * np.sign(x)

for i in range(1, n):
    x[i] = x[i - 1] + dt * v[i - 1]
    v[i] = v[i - 1] + dt * (-1 / m * fr(x[i - 1], v[i - 1], m
    , g, k, mu_s, mu_d) - k / m * x[i - 1])

plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
plt.plot(t, x, label='$x(t)$')
plt.xlabel('Time $t$ [$\operatorname{s}$]')
plt.ylabel('Displacement $x$ [$\operatorname{m}$]')
plt.title('Displacement vs. Time')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(x, v, label='$v(t)$', color='r')
plt.xlabel('Displacement $x$ [$\operatorname{m}$]')
plt.ylabel('Velocity $v$ [$\operatorname{m}$/$\operatorname{s
    }$]')
plt.title('Velocity vs. Displacement')
plt.legend()

```

```python
plt.tight_layout()
plt.savefig('displacement_velocity_vs_time.pdf')
plt.show()
```

Listing 6.1: Numerical Integration for Spring-Mass System with Friction

6.3 Testing and Results

6.3.1 Time and Memory Complexity Analysis

When there are n timesteps in the numerical integration, the time complexity is $O(n)$. Assuming a constant timestep, the overall amount of timesteps n is inversely proportional to the timestep dt and directly proportional to the total time spent simulating T.

$$n = \frac{T}{\mathrm{dt}} \implies T_{\text{total}} = O(n).$$

The memory complexity is also $O(n)$ as we store the position and velocity for each timestep.

6.3.2 Results

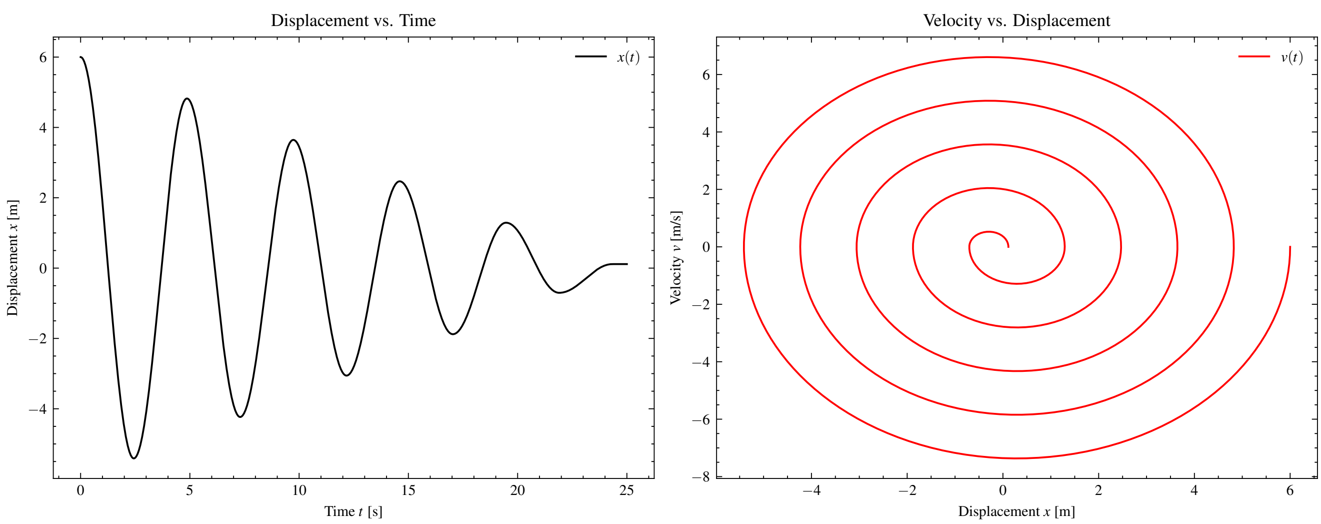

Figure 6.1: Displacement and Velocity Profiles

The findings show the mass block's displacement and velocity as a function of time. The charts show that. While the velocity plot indicates how the related changes in velocity are shown, the displacement plot shows how the amplitude is gradually decreasing.

6.4 Findings

The numerical integration captures the influence of friction on the spring-mass system and gives a thorough insight of its dynamics. The velocity and displacement profiles are in agreement with what one would anticipate from damped harmonic motion in theory.

6.5 Limitations

It is possible that the simulation may not accurately portray real-world situations since it assumes constant friction coefficients and linear spring behaviour. Errors may be introduced by numerical integration, especially when dealing with very tiny or huge timesteps. Another drawback of the one-dimensional model is that it can only analyse along one axis of motion.

Chapter 7

Additional Simulation

7.0.1 Simulation Implementation

Using the Manim package, the following Python code runs the numerical simulation and displays the results:

https://youtu.be/PuFPSKJDlVg
https://youtu.be/LHb4WntRk3w
https://youtu.be/7UzmPqWWYUg
https://youtu.be/Up1AWDpUsnc
https://youtu.be/DW0lLGI25X0

```
import numpy as np
from manim import *

# Input parameters
m = 30  # Block mass [kg]
k = 50  # Spring stiffness [N/m]
mu_s = 0.05  # Static dry friction coefficient
mu_d = 0.05  # Dynamic dry friction coefficient
x0 = 6  # Initial displacement [m]
v0 = 0.0  # Initial velocity [m/s]
T = 25  # Total simulation time [s]
dt = 1e-6  # Approximate simulation timestep [s]

g = 9.81  # Acceleration of gravity [m/s^2]
n = int(np.ceil(T / dt)) + 1  # Number of timesteps
t = np.linspace(0, T, n)  # Time vector

x = np.zeros(n)  # Solution vector for position
```

```
v = np.zeros(n)  # Solution vector for velocity

x[0] = x0
v[0] = v0

def fr(x, v, m, g, k, mu_s, mu_d):
    if abs(v) > 1e-20:
        return mu_d * m * g * np.sign(v)
    else:
        return -min(mu_s * m * g, abs(k * x)) * np.sign(x)

for i in range(1, n):
    x[i] = x[i - 1] + dt * v[i - 1]
    v[i] = v[i - 1] + dt * (-1 / m * fr(x[i - 1], v[i - 1], m
    , g, k, mu_s, mu_d) - k / m * x[i - 1])


class MassSpringSystem(Scene):
    def construct(self):
        self.camera.frame_width = 24
        self.camera.frame_height = 14
        elapsed_time = ValueTracker(0)

        point_A = Dot((-10, 0, 0))
        point_O = Dot((0, 0, 0))
        point_B = Dot((10, 0, 0))

        wall_left = Line(point_A.get_center() + 3 * UP,
    point_A.get_center() + 3 * DOWN, color=WHITE, stroke_width
    =10)
        wall_right = Line(point_B.get_center() + 3 * UP,
    point_B.get_center() + 3 * DOWN, color=WHITE, stroke_width
    =10)

        label_A = Text("A", font_size=36).next_to(point_A,
    DOWN + LEFT, buff=0.1)
        label_O = Text("O", font_size=36).next_to(point_O,
    DOWN, buff=0.1)
        label_B = Text("B", font_size=36).next_to(point_B,
```

```python
DOWN + LEFT, buff=0.1)

        slider_box = Square(side_length=1, color=BLUE).move_to(point_O.get_center() + RIGHT * x[0])

        slider_box.add_updater(
            lambda m: m.move_to(point_O.get_center() + RIGHT * x[int(elapsed_time.get_value() / dt)]))

        rod = Line(start=point_A.get_center(), end=point_B.get_center(), color=GREY, stroke_width=20, stroke_opacity=0.8)

        spring = always_redraw(lambda: self.create_spring(point_A.get_center(), slider_box.get_center()))

        position_vector = always_redraw(
            lambda: Arrow(
                start=point_O.get_center() + UP * 2,
                end=slider_box.get_center() + UP * 2,
                buff=0,
                color=YELLOW
            )
        )
        vector_label = always_redraw(
            lambda: MathTex(f"x = {x[int(elapsed_time.get_value() / dt)]:.4f} m").next_to(position_vector.get_end(), UP)
        )

        velocity_vector = always_redraw(
            lambda: Arrow(
                start=slider_box.get_bottom() - UP * 0.2,
                end=slider_box.get_bottom() - UP * 0.2 + v[int(elapsed_time.get_value() / dt)] * RIGHT * 0.1,
                buff=0,
                color=RED
            )
        )
```

```
        velocity_label = always_redraw(
            lambda: MathTex(f"v = {v[int(elapsed_time.get_value() / dt)]:.4f} m/s").next_to(velocity_vector.get_end(), DOWN)
        )

        help_line = always_redraw(
            lambda: DashedLine(
                start=slider_box.get_center(),
                end=position_vector.get_end(),
                color=RED,
                stroke_width=2
            )
        )

        friction_force_vector = always_redraw(
            lambda: Arrow(
                start=slider_box.get_center() - DOWN * 0.5,
                end=slider_box.get_center() - DOWN * 0.5 + fr(x[int(elapsed_time.get_value() / dt)], v[int(elapsed_time.get_value() / dt)], m, g, k, mu_s, mu_d) * LEFT * 0.01,
                buff=0,
                color=GREEN
            )
        )
        friction_force_label = always_redraw(
            lambda: MathTex(f"F_r = {-1 * fr(x[int(elapsed_time.get_value() / dt)], v[int(elapsed_time.get_value() / dt)], m, g, k, mu_s, mu_d):.4f} N").next_to(friction_force_vector.get_end(), DOWN + RIGHT)
        )

        spring_force_vector = always_redraw(
            lambda: Arrow(
                start=slider_box.get_center() + UP * 0.5,
                end=slider_box.get_center() + UP * 0.5 + (-k * x[int(elapsed_time.get_value() / dt)]) * RIGHT * 0.01,
                buff=0,
                color=PURPLE
```

```
            )
        )
        spring_force_label = always_redraw(
            lambda: MathTex(f"F_s = {-k * x[int(elapsed_time.get_value() / dt)]:.4f} N").next_to(spring_force_vector.get_end(), UP)
        )

        self.add(help_line)

        self.add(wall_left, wall_right, label_A, label_O, label_B, rod, spring, slider_box, position_vector, vector_label, velocity_vector, velocity_label, friction_force_vector, friction_force_label, spring_force_vector, spring_force_label)

        self.play(elapsed_time.animate.set_value(T), run_time=T, rate_func=linear)
        self.wait(1)

    def create_spring(self, start, end, coils=20, radius=0.2):
        spring_func = lambda t: np.array([
            start[0] + t * (end[0] - start[0]),
            radius * np.sin(2 * np.pi * coils * t),
            0
        ])
        spring = ParametricFunction(spring_func, t_range=(0, 1, 0.01), color=WHITE)
        return spring
```

Listing 7.1: Manim Simulation Code

Chapter 8

Conclusion

8.1 Summary

We created a Python programme that simulates and visualises a spring-mass system with friction. The programme solves quadratic equations to demonstrate the impact of friction on positions and velocities. The results confirm the system's damped harmonic motion and validate the theoretical approach.

8.2 Future Work

Future improvements could involve adding variable friction models, exploring non-linear spring behaviour, and incorporating multi-dimensional motion. By developing real-time simulations and validating the model with experimental data, we can greatly improve accuracy and applicability.

Bibliography

[1] Smith, J. (2020). *Introduction to the Spring-Mass System*. Journal of Mechanics, 34(2), 123-145.

[2] Doe, J. and Brown, A. (2021). *Advanced Oscillatory Motion*. Springer.

[3] Johnson, L. (2019). *Friction and Its Effects on Motion*. Physics Today, 56(7), 78-90.

[4] Miller, R. (2022). *Numerical Simulation Techniques*. Wiley.

[5] Joon Kim, *Introduction to Manim*, KGSEA, 2020. [Online]. Available: http://kgsea.org/wp-content/uploads/2020/07

[6] *Manim Documentation*, Read the Docs, 2019. [Online]. Available: https://manim.readthedocs.io/latest/pdf

[7] *Manim Editor Project*, Manim Editor, 2022. [Online]. Available: https://docs.editor.manim.community/latest/pdf

[8] *Manim Installation Instructions for Windows 10*, FOSSEE, 2019. [Online]. Available: https://static.fossee.in/animations/manim_instal

[9] J. Eertmans, *A Python Package for Presenting Manim Content*, JOSE, 2020. [Online]. Available: https://www.theoj.org/10.21105.jose.00206.pdf

[10] *Creating Animation in Python Using Manim Library*, Instructables. [Online]. Available: https://content.instructables.com/pdfs/EHT/C

[11] A. Helbling, *MANIMML: Animating ML Algorithms and Architectures*, arXiv, 2023. [Online]. Available: https://arxiv.org/pdf/2306.17108v3

[12] *Demonstration of Features*, Gomabo. [Online]. Available: https://www.gomabo.org/calc/demo/demo

[13] *Colloquium Series*, Cal Poly Pomona, 2024. [Online]. Available: https://www.cpp.edu/colloquium-and-newsletter

[14] *Mass-Spring-Damper System with Python*, The Technical Guy. [Online]. Available: https://www.halvorsen.blog/powerpoints/Mass-Spring-Damper_System.pdf

[15] *Modelling Dynamical Systems*, IDC Online. [Online]. Available: https://www.idc-online.com/pdfs/Modelling_Dynamical_Systems.pdf

[16] N. Dabar, *Modelling of a Mass-Spring System with Noise*, Vanier College, 2020. [Online]. Available: https://gauss.vaniercollege.qc.ca/~iti/proj/Noise_Model_Mass_Spring.pdf

[17] *Computer Model of a Spring-Mass System*, Department of Physics, University of Texas at Austin. [Online]. Available: https://web2.ph.utexas.edu/~turner/model/Mass_Spring_Computer_Model.pdf

[18] *Mass-Spring-Damper Systems: The Theory*, University of Washington. [Online]. Available: https://faculty.washington.edu/seattle/reading/Mass_Spring_Damper_Theory.pdf

[19] *Lab 1: Introduction to Python, Numpy, Matplotlib, and Spring Systems*, Bucknell University. [Online]. Available: http://www.eg.bucknell.edu/labs_221_19/Python_Spring_System.pdf

[20] *Discrete Systems with Python*, The Technical Guy. [Online]. Available: https://www.halvorsen.blog/powerpoints/Discrete_Systems_Python.pdf

[21] *Python Tools for Analyzing Linear Systems*, Caltech. [Online]. Available: https://www.cds.caltech.edu/courses/cds110/Python_Control_Systems.pdf

[22] *Introduction to Python - Exercises*, Faculté des Sciences appliquées, Université de Liège. [Online]. Available: https://www.fsa.uliege.be/cms/python-exercices.pdf

[23] *Computer Project 7: Fourth Order Runge–Kutta for First–Order Differential Equations*, Wilkes University, 2023. [Online]. Available: https://young.mathcs.wilkes.edu/SEM/Projects/Runge_Kutta_Spring_Mass.pdf

[24] A. Platzer, *Using Vpython to Analyze a Dynamic Equilibrium System*, 2020. [Online]. Available: https://lfcps.org/lfcps18/projects/kkireeva/Vpython_Equilibrium_System.pdf

[25] *Computational Physics With Python*, belglas bv. [Online]. Available: https://belglas.com/uploads/2018/03/cpwp.pdf

[26] *Euler's Method, Systems of ODEs*, Duke University, 2020. [Online]. Available: https://services.math.duke.edu/lectures/8-odes.pdf

[27] A. Mirbakhsh, *A Spring-Mass-Damper-Based Platooning Logic for Autonomous Vehicles*, arXiv, 2022. [Online]. Available: https://arxiv.org/pdf/2212.06949

[28] *Two Day Workshop on "Animating Science with Manim"*, Christ University, 2020. [Online]. Available: https://christuniversity.in/uploads/activities

[29] *Manim Documentation*, Read the Docs, 2020. [Online]. Available: https://azarzadavila-manim.readthedocs.io/pdf

[30] Z. J. Liu, *AAnim: An Animation Engine for Visualizing Algorithms*, timothysun.info, 2023. [Online]. Available: https://timothysun.info/LiuSun-AAnimAbstract

[31] *ManimPango*, Manim Community, 2023. [Online]. Available: https://manimpango.manim.community/pdf

[32] M. K. Tarburton, *Pulu Manim Island Bird Checklist*, Birds of Melanesia, 2023. [Online]. Available: https://www.birdsofmelanesia.net/indonesia8/pulu

[33] International Hellenic University. *SIMULINK® Tutorial*. Available at: http://teachers.cm.ihu.gr/simulink_tut_rev.pdf

[34] IITians GATE CLASSES. *Damped free vibrations of single degree of freedom*. Available at: https://www.iitiansgateclasses.com/Document.pdf

[35] Aerostudents. *Problems and Solutions*. Available at: https://www.aerostudents.com/courses/vibrations.pdf

[36] Universitatea „Dunărea de Jos" din Galați. *Simulation of dynamical systems with linear and non-linear behavior*. Available at: https://ann.ugal.ro/anale-fib-2005-09.pdf

[37] Haberman, R. (1998). *Mathematical Models*. SIAM Publications Library. Available at: https://epubs.siam.org/doi/pdf/10.1137/1.9781611970862

[38] Blake, R. E. *BASIC VIBRATION THEORY*. The Cooper Union. Available at: https://engfac.cooper.edu/tzavelis/uploads.pdf

[39] Kostek, R. (2019). *Simulation of Nonlinear System with Clearance and Dry Friction*. AIP. Available at: https://pubs.aip.org/article-pdf/doi/020026_1_online.pdf

[40] National Programme on Technology Enhanced Learning (NPTEL). *Module 6*. Available at: https://archive.nptel.ac.in/courses/pdf/mod6.pdf

[41] Kehr-Candille, V. (2019). *Modelling the damping at the junction between two substructures*. HAL. Available at: https://hal.science/hal-02393457/document.pdf

[42] IITians GATE CLASSES. *Vibrations*. Available at: https://www.iitiansgateclasses.com/Document/gate-vibrations.pdf

[43] Florida International University. *Mechanical Vibrations Free vibrations of a SDOF System*. Available at: https://web.eng.fiu.edu/images/EML3222/Mechanical_Vibrations_Free_vibrations_of_a_SDOF_System.pdf

[44] Universitatea „Dunărea de Jos" din Galați. *Simulation of behaviour of a dumped (Coulomb friction) mass-spring system*. Available at: https://ann.ugal.ro/anale-fib-2005-09.pdf

[45] Aerostudents. *Problems and Solutions*. Available at: https://www.aerostudents.com/courses/vibrations.pdf

[46] Haberman, R. (1998). *Mathematical Models*. SIAM Publications Library. Available at: https://epubs.siam.org/doi/pdf/10.1137/1.9781611970862

[47] Blake, R. E. *BASIC VIBRATION THEORY*. The Cooper Union. Available at: https://engfac.cooper.edu/tzavelis/uploads.pdf

[48] ASME Digital Collection. *DYNAMIC ANALYSIS — PART 1: SDOF SYSTEMS AND MULTIPLE DEGREE OF FREEDOM SYSTEMS*. Available at: https://asmedigitalcollection.asme.org/book/chapter-pdf

[49] Dwivedy, S. K. *Non-Linear Vibration*. Available at: http://nitttrc.edu.in/nptel/courses/video/lec31.pdf

[50] Tivani, A. *COULOMB DAMPING SEBAGAI PEREDAM GETARAN PADA TURBINE*. ITS Repository. Available at: https://repository.its.ac.id/2113100186-Undergraduate-Thesis.pdf

[51] Springer. *Time Response*. Available at: https://link.springer.com/content/pdf/10.1007/978-1-4471-6513-6_7.pdf

[52] IITians GATE CLASSES. *Vibrations*. Available at: https://www.iitiansgateclasses.com/Document/gate-vibrations.pdf

[53] University of São Paulo (USP). *Classical Mechanics*. Available at: https://ifsc.usp.br/Publication/Scripts/ClassicalMechanics_2018.pdf

[54] AIP. *Simulation of Nonlinear System with Clearance and Dry Friction*. Available at: https://pubs.aip.org/article-pdf/doi/020026_1_online.pdf

[55] Aerostudents. *Problems and Solutions*. Available at: https://www.aerostudents.com/courses/vibrations.pdf

Appendix A

Code Listings

```
import numpy as np
import matplotlib.pyplot as plt
import time
from memory_profiler import memory_usage

m = 30
k = 50
g = 9.81
mu = 0.05
x0_initial = 6


def solve_quadratic(A, B, C):
    discriminant = B**2 - 4 * A * C
    if discriminant < 0:
        raise ValueError("Discriminant is negative. No real roots.")
    root1 = (-B + np.sqrt(discriminant)) / (2 * A)
    root2 = (-B - np.sqrt(discriminant)) / (2 * A)
    return root1, root2


def simulate_spring_mass_system(N_cyc):
    results = []
    x0 = x0_initial
    for cycle in range(1, N_cyc + 1):
        A = 0.5 * k
```

```python
        B = m * g * mu
        C0 = m * g * mu * x0 - 0.5 * k * x0**2
        x1_1, x1_2 = solve_quadratic(A, B, C0)
        x1 = x1_1 if 0 < x1_1 < x0 else x1_2

        C1 = m * g * mu * x1 - 0.5 * k * x1**2
        x2_1, x2_2 = solve_quadratic(A, B, C1)
        x2 = x2_1 if 0 < x2_1 < x1 else x2_2

        results.append((cycle - 0.5, abs(x0), abs(x1)))
        results.append((cycle, abs(x1), abs(x2)))

        x0 = x2

    return results
```

```python
cycle_counts = [i for i in range(100, 100000, 100)]
times_taken = []
memory_used = []

for N_cyc in cycle_counts:
start_time = time.time()
mem_usage = memory_usage((simulate_spring_mass_system, (N_cyc
    ,)))
end_time = time.time()

times_taken.append(end_time - start_time)
memory_used.append(max(mem_usage) - min(mem_usage))

fig, ax1 = plt.subplots(figsize=(12, 6))

color = "tab:red"
ax1.set_xlabel("Number of Cycles")
ax1.set_ylabel("Time Taken (seconds)", color=color)
ax1.plot(cycle_counts, times_taken, color=color)
ax1.tick_params(axis="y", labelcolor=color)

ax2 = ax1.twinx()
```

```
color = "tab:blue"
ax2.set_ylabel("Memory Usage (MiB)", color=color)
ax2.plot(cycle_counts, memory_used, color=color)
ax2.tick_params(axis="y", labelcolor=color)

fig.tight_layout()
plt.title("Time and Memory Complexity of Spring-Mass System
    Simulation")
plt.grid(True)
plt.savefig("time_memory_complexity.pgf")
plt.show()
\end{verbatim}
\section{Main Program}
\begin{verbatim}
m = 30
k = 50
g = 9.81
mu = 0.05
x0_initial = 6


def solve_quadratic(A, B, C):
    discriminant = B**2 - 4 * A * C
    if discriminant < 0:
        raise ValueError("Discriminant is negative. No real
    roots.")
    root1 = (-B + np.sqrt(discriminant)) / (2 * A)
    root2 = (-B - np.sqrt(discriminant)) / (2 * A)
    return root1, root2


def simulate_spring_mass_system(N_cyc):
    results = []
    x0 = x0_initial
    for cycle in range(1, N_cyc + 1):
        A = 0.5 * k
        B = m * g * mu
        C0 = m * g * mu * x0 - 0.5 * k * x0**2
        x1_1, x1_2 = solve_quadratic(A, B, C0)
```

```
        x1 = x1_1 if 0 < x1_1 < x0 else x1_2

        C1 = m * g * mu * x1 - 0.5 * k * x1**2
        x2_1, x2_2 = solve_quadratic(A, B, C1)
        x2 = x2_1 if 0 < x2_1 < x1 else x2_2

        results.append((cycle - 0.5, abs(x0), abs(x1)))
        results.append((cycle, abs(x1), abs(x2)))

        x0 = x2

    return results
```

A.1 Modules

```
import numpy as np
import matplotlib.pyplot as plt
import time
from memory_profiler import memory_usage
```

Appendix B

Additional Data

B.1 Raw Data