Project Report – Python Programming Project 2025
Title: Nonlinear Driven–Damped Pendulum Simulation and Visualization using Python Scientific Libraries
Course: Introduction to Computer Systems and Programming (Python)
Group Members:

- Surasit Khongcharoen 6838311521
- Khanatat Tansuriwongse 6838026721
- Korn Boonyasirikul 6838007821
- Jeerapat Pornaroontam 6838045621

1. Objective of the Project

The main objective of this project is to simulate and visualize the motion of a **nonlinear driven–damped pendulum** using Python and scientific libraries that were not taught directly in class.

In this project, we aim to:

1. Model the motion of a pendulum with **damping** (friction/air resistance) and **external periodic driving force**.
2. Use **SciPy** to solve the nonlinear ordinary differential equation (ODE) of the pendulum numerically.
3. Use **Matplotlib** to create:
    - Time–series plots of angular position and angular velocity.
    - A phase diagram (θ vs. $\dot{\theta}$).
    - A real-time animation of the pendulum's motion.
4. Calculate the **kinetic energy (KE)**, **potential energy (PE)**, and **total mechanical energy (E)** of the pendulum over time.
5. Practice self-learning by reading external documentation and tutorials for SciPy, Matplotlib, and advanced plotting tools (gridspec, animation, patches).

This project demonstrates how programming concepts from class (functions, lists/arrays, if-else, loops) can be combined with external libraries to simulate a real physical system and present the results in a clear, visual way.

2. Description of Input and Output

2.1 Input

In this program, the “inputs” are mainly **internal parameters** defined at the top of the script. The user can change these values directly in the code to explore different behaviors of the pendulum.

Key input parameters:

- `g = 9.81`
  Acceleration due to gravity (m/s²).
- `ell = 1`
  Length of the pendulum (m).
- `theta0 = np.deg2rad(170)`
  Initial angular displacement (in degrees, converted to radians).
  In this case, the pendulum starts almost upside down at 170°.
- `theta_dot0 = 0`
  Initial angular velocity (rad/s).
- `mu = 0.2`
  Damping coefficient. Larger `mu` means more air resistance/friction.
- `F_0 = 2`
  Amplitude of the external driving force.
- `omega = 3.1305`
  Angular frequency of the driving force.
- `time_range = 40`
  Total simulation time (seconds).
- `fps = 60`
  Frames per second for the animation and time sampling.

These parameters define the physical system and the resolution of the simulation. The user can modify them to see how the behavior of the pendulum changes (e.g., more damping, different driving frequency, different starting angle).

2.2 Output

The program produces several types of outputs:

1. **Numerical solution arrays**
   - `theta`: angular position $\theta(t)$ in radians.
   - `theta_dot`: angular velocity $\dot{\theta}(t)$ in rad/s.
   - `t`: time points at which the solution is computed.

    - `theta_deg` and `theta_dot_deg`: the same values converted to degrees for easier interpretation in plots.
2. **Energy values**
    - KE: kinetic energy as a function of time.
    - PE: potential energy.
    - E: total mechanical energy (KE + PE).

The program also prints:
Max total energy: ...
Min total energy: ...

    - This shows how the total energy varies due to damping and external forcing.
3. **Visual outputs (plots and animation)**
    - **Time series plot (subplot ax1)**

        θ(t) in degrees.

        $\dot{\theta}$(t) in degrees per second.

        Both are plotted versus time t.

    - **Phase diagram (subplot ax2)**

        θ (deg) on the x-axis.

        $\dot{\theta}$ (deg/s) on the y-axis.

        Shows the trajectory of the pendulum in phase space with a moving point.

    - **Pendulum animation (subplot ax3)**

        A real-time animation displaying the pendulum as a rod and a bob.

        The bob moves according to θ(t), and the rod length is `ell`.

All visual outputs appear in a single figure window arranged using `GridSpec`, giving a combined view of motion, phase diagram, and the actual pendulum.

3. Details of the Libraries Used

In this project, we used several Python libraries beyond the basic ones from class. These include:

3.1 NumPy (`import numpy as np`)

Purpose:

- To handle arrays and perform efficient numerical operations.

Usage in the code:

- Convert angles from degrees to radians and back:

```
theta0 = np.deg2rad(170)
theta_deg = np.rad2deg(theta)
```

- Create time arrays for evaluation:

```
t_eval = np.linspace(0, time_range, fps*time_range)
```

- Perform element-wise operations such as `np.sin(theta), np.cos(theta)`, and array-based energy calculations.
- Compute maximum and minimum energy values:

```
np.max(E), np.min(E)
```

NumPy makes it easy and efficient to work with large sets of numerical data.

3.2 SciPy – `solve_ivp` (`from scipy.integrate import solve_ivp`)

Purpose:

- To numerically solve the system of ODEs that describes the driven–damped pendulum.

The equation of motion for the pendulum is:

$$\ddot{\theta}(t) = -\mu\dot{\theta}(t) - \frac{g}{\ell}\sin(\theta(t)) + F_0\sin(\omega t)$$

We rewrite this second-order ODE as a system of first-order ODEs:

- Let $(y_0 = \theta)$, $(y_1 = \dot{\theta})$
- Then:
    - $(\dot{y}_0 = y_1)$
    - $(\dot{y}_1 = -\mu y_1 - \frac{g}{\ell}\sin(y_0) + F_0\sin(\omega t))$

In the code, this is implemented as:

```
def pendulum_ODE(t, y):
    return (y[1],
            -mu*y[1] + (-g*np.sin(y[0])/ell) + F_0*np.sin(omega*t))
```

Then we call:

```
sol = solve_ivp(
    pendulum_ODE,
    [0, time_range],
    (theta0, theta_dot0),
    t_eval=np.linspace(0, time_range, fps*time_range),
    rtol=1e-10, atol=1e-12
)
```

- `solve_ivp` integrates the system over the interval `[0, time_range]` with a very small relative and absolute tolerance to get accurate results.
- `sol.y` contains the arrays for θ and θ̇.
- `sol.t` contains the time points.

3.3 Matplotlib (`import matplotlib.pyplot as plt`)

Purpose:

- To create all 2D plots and show the final figure with subplots and animation.

Usage:

- Create a figure:

```
fig = plt.figure(figsize=(10, 10))
```

- Add subplots for time series, phase diagram, and pendulum animation.

Set labels, titles, axis limits, and grids:

```
ax1.set_title('Simple Pendulum: Angular position, velocity vs time')
ax1.set_xlabel('Time (seconds)')
ax1.set_ylabel(r'$\theta$ (deg) $\dot \theta$ (deg/s)')
ax1.grid()
```

3.4 Matplotlib Gridspec (`import matplotlib.gridspec as gridspec`)

Purpose:

- To organize multiple subplots with different sizes in one figure.

Usage:

```
gs = gridspec.GridSpec(2, 2, width_ratios=[1, 2], height_ratios=[1, 1])
ax1 = fig.add_subplot(gs[0, 0])   # time series
ax2 = fig.add_subplot(gs[1, 0])   # phase diagram
ax3 = fig.add_subplot(gs[:, 1])   # pendulum animation
```

This allows the pendulum animation to occupy a larger area on the right, while the time series and phase diagram share the left side.

3.5 Matplotlib Animation (`import matplotlib.animation as animation`)

Purpose:

- To create a real-time animation of the pendulum and the evolving plots using `FuncAnimation`.

Usage:

Define an initialization function:

```
def init_anim():
    theta_curve_1.set_data([], [])
    theta_dot_curve_1.set_data([], [])
    phase_curve_2.set_data([], [])
    phase_dot_2.set_data([], [])
    line_3.set_data([0, x_positions[0]], [0, y_positions[0]])
    circle_3.set_center((x_positions[0], y_positions[0]))
    return theta_curve_1, theta_dot_curve_1, phase_curve_2, phase_dot_2,
line_3, circle_3
```

Define an update function called at each frame:

```
def animate_all(i):
    # update time series
    theta_curve_1.set_data(t[:i+1], theta_deg[:i+1])
    theta_dot_curve_1.set_data(t[:i+1], theta_dot_deg[:i+1])

    # update phase diagram
    phase_curve_2.set_data(theta_deg[a:i+1], theta_dot_deg[a:i+1])
    phase_dot_2.set_data([theta_deg[i]], [theta_dot_deg[i]])

    # update pendulum position
    x = x_positions[i]
    y = y_positions[i]
    line_3.set_data([0, x], [0, y])
    circle_3.set_center((x, y))

    return theta_curve_1, theta_dot_curve_1, phase_curve_2, phase_dot_2,
line_3, circle_3
```

Create the animation:

```
ani_all = animation.FuncAnimation(
    fig,
    animate_all,
    frames=len(t),
    interval=1000/fps,
    blit=True,
    init_func=init_anim,
    repeat=False
)
```

The animation displays both the motion of the pendulum and the development of the graphs over time.

3.6 Matplotlib Patches (`import matplotlib.patches as mpatches`)

Purpose:

- To draw the pendulum bob as a **filled circle** on the animation subplot.

Usage:

```
circle_3 = mpatches.Circle((x0, y0), 0.05, fc='r', zorder=3, animated=True)
ax3.add_patch(circle_3)
```

Here, `Circle` is a patch object, with center `(x0, y0)`, radius `0.05`, and red face color.

---

4. Explanation of the Code, Results, and Physics

4.1 Structure of the Code

The code can be divided into the following parts:

1. **Import libraries**
   All necessary libraries are imported (NumPy, SciPy, Matplotlib, etc.).
2. **Define physical and simulation parameters**
   Constants such as `g`, `ell`, `theta0`, `theta_dot0`, `mu`, `F_0`, `omega`, `time_range`, and `fps` are defined.
3. **Define the ODE system**
   The function `pendulum_ODE(t, y)` returns the right-hand side of the system of ODEs based on the nonlinear driven–damped pendulum equation.

4. **Solve the ODE numerically**
   `solve_ivp` is used to compute $\theta(t)$ and $\dot{\theta}(t)$ across the time range with a fixed number of evaluation points for smooth plots and animation.
5. **Convert units and compute energies**
   - Convert θ and $\dot{\theta}$ to degrees for plotting.

Compute KE, PE, and total E using the function `pendulum_energy()`:

```
v = ell * theta_dot
KE = 0.5 * m * v**2
PE = m * g * ell * (1 - np.cos(theta))
E = KE + PE
```

6. **Set up the figure and subplots**
   Using `GridSpec`, the program creates:
   - Top-left: time series plot.
   - Bottom-left: phase diagram.
   - Right: pendulum animation.
7. **Initialize and update animation**
   The program precomputes the bob positions (`x_positions`, `y_positions`) to make the animation faster and updates all curves and the pendulum bob in each frame.

4.2 Interpretation of the Results

- The **time series plot** shows how the angle and angular velocity change over time. Because the system is driven and damped, the motion can be quite complex and may not be a simple periodic oscillation.
- The **phase diagram** (θ vs. $\dot{\theta}$) shows the trajectory in phase space. For a simple, undamped, undriven pendulum, this would be a closed curve. For a driven–damped pendulum, the pattern can become more complicated and may show spirals or strange attractors depending on parameters.
- The **pendulum animation** helps us visualize the actual physical motion, making it easier to connect the graphs with the real movement.
- The **energy plot (internal)**, via `Max total energy` and `Min total energy`, shows that total mechanical energy is not constant:
  - Damping ($\mu > 0$) removes energy from the system.
  - The driving force adds energy into the system.

Depending on the balance between damping and driving, the system can reach a steady-state oscillation, show resonance-like behavior, or even chaotic-like motion for some parameter ranges.

5. References

Below are some references and resources that can be cited in the report (the group should adjust based on actual sources used):

1. SciPy documentation – `integrate.solve_ivp`:
    - Official SciPy documentation for solving initial value problems of ODEs.
2. Matplotlib documentation – `pyplot`, `gridspec`, `animation`, and `patches`:
    - Official Matplotlib documentation explaining the plotting and animation functions.
3. Standard physics textbooks on classical mechanics and oscillations that discuss the nonlinear pendulum and driven–damped systems.

(Replace with specific titles/web links if required by the instructor.)

6. AI Tools and Tutorial Declaration

We, the group members, declare the following:

- We wrote the core simulation code and understood each part of the program.
- We used https://www.youtube.com/watch?v=WNJJuSWfuTY and https://www.youtube.com/watch?v=p_di4Zn4wz4&t=956s as tutorials.
- We used **external documentation and tutorials** mainly for:
    - Understanding the usage of `scipy.integrate.solve_ivp`.
    - Learning how to use `matplotlib.animation.FuncAnimation`.
    - Learning how to use `gridspec` and `patches` for layout and drawing shapes.
- We also used **AI tools** to assist in:
    - Improving the structure and clarity of the project report.
    - Getting suggestions on code optimization (for example, precomputing positions and using blitting for animation).
    - Checking explanations of the physics and the numerical methods in simple language.

All final decisions about the code, parameter choices, and explanations were made by the group. We understand the working of the program and can explain it during any presentation or viva if required.