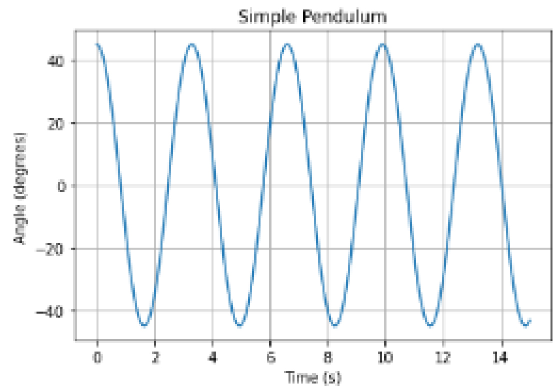
Activity 1: Pendulum

a. Simple Pendulum

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

def simulate_pendulum(length, gravity, theta0, time_step, total_time):
    # Convert the initial angle to radians
    theta0 = np.radians(theta0)

    # Calculate the number of time steps
    num_steps = int(total_time / time_step)

    # Initialize arrays to store the time, angle, and angular velocity
    t = np.zeros(num_steps)
    theta = np.zeros(num_steps)
    omega = np.zeros(num_steps)

    # Set the initial conditions
    t[0] = 0
    theta[0] = theta0
    omega[0] = 0

    # Simulate the pendulum motion using the Euler-Cromer method
    for i in range(1, num_steps):
        t[i] = i * time_step
        omega[i] = omega[i-1] - (gravity/length) * np.sin(theta[i-1]) * time_step
        theta[i] = theta[i-1] + omega[i] * time_step

    return t, theta

# Parameters of the pendulum
length = 2.5  # Length of the pendulum (in meters)
gravity = 9.8  # acceleration due to gravity (in m/s^2)
theta0 = 45.0  # initial angle (in degrees)
time_step = 0.001  # time step for simulation (in seconds)
total_time = 15  # total simulation time (in seconds)

# Simulate the pendulum motion
t, theta = simulate_pendulum(length, gravity, theta0, time_step, total_time)

# Convert the angles back to degrees
theta_deg = np.degrees(theta)

# Plot the data points
plt.plot(t, theta_deg)
plt.xlabel('Time (s)')
plt.ylabel('Angle (degrees)')
plt.title('Simple Pendulum')
plt.grid(True)
plt.show()
```

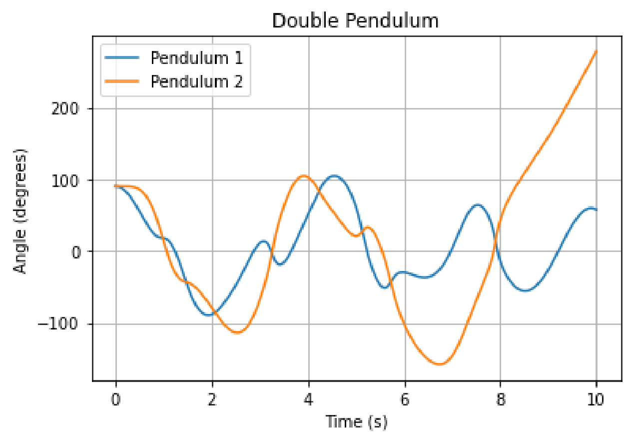


The length of the pendulum, the acceleration due to gravity, the initial angle, the time step, and the overall simulation time are all inputs to the function "simulate_pendulum" that we define in this code. The time and angle arrays are returned after numerically simulating the pendulum's motion using the Euler-Cromer method.

We use the "simulate_pendulum" function to get the time and angle arrays after providing the pendulum's length, gravity, initial angle, time step, and overall time. Finally, we used Matplotlib to plot the data points and convert the angles from radians to degrees.

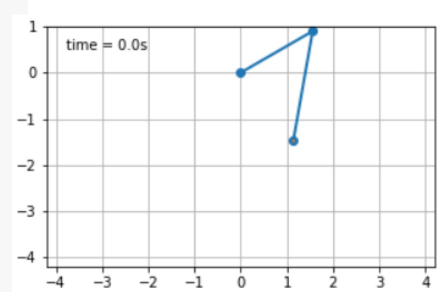
b. Double Pendulum

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

def double_pendulum(t, y, l1, l2, m1, m2, g):
    theta1, omega1, theta2, omega2 = y

    c = np.cos(theta1 - theta2)
    s = np.sin(theta1 - theta2)

    theta1_dot = omega1
    omega1_dot = (m2 * g * np.sin(theta2) * c - m2 * s * (l1 * omega1 ** 2 * c + l2 * omega2 ** 2) -
                  (m1 + m2) * g * np.sin(theta1)) / (l1 * (m1 + m2 * s ** 2))

    theta2_dot = omega2
    omega2_dot = ((m1 + m2) * (l1 * omega1 ** 2 * s - g * np.sin(theta2) + g * np.sin(theta1) * c) +
                  m2 * l2 * omega2 ** 2 * s * c) / (l2 * (m1 + m2 * s ** 2))

    return [theta1_dot, omega1_dot, theta2_dot, omega2_dot]

def simulate_double_pendulum(theta1_0, theta2_0, omega1_0, omega2_0, l1, l2, m1, m2, g, total_time, num_steps):
    t_span = (0, total_time)
    y0 = [theta1_0, omega1_0, theta2_0, omega2_0]

    t = np.linspace(0, total_time, num_steps)
    sol = solve_ivp(double_pendulum, t_span, y0, t_eval=t, args=(l1, l2, m1, m2, g))

    return sol.t, sol.y

# Parameters of the double pendulum
l1 = 1.8  # Length of the first pendulum (in meters)
l2 = 2.4 # Length of the second pendulum (in meters)
m1 = 1.5  # mass of the first pendulum (in kilograms)
m2 = 2.2  # mass of the second pendulum (in kilograms)
g = 9.8  # acceleration due to gravity (in m/s^2)

theta1_0 = np.pi / 2  # initial angle of the first pendulum (in radians)
theta2_0 = np.pi / 2  # initial angle of the second pendulum (in radians)
omega1_0 = 0.0  # initial angular velocity of the first pendulum (in radians/s)
omega2_0 = 0.0  # initial angular velocity of the second pendulum (in radians/s)
total_time = 10.0  # total simulation time (in seconds)
num_steps = 1000  # number of time steps for simulation

# Simulate the double pendulum motion
t, y = simulate_double_pendulum(theta1_0, theta2_0, omega1_0, omega2_0, l1, l2, m1, m2, g, total_time, num_steps)

# Extract the angles and angular velocities
theta1 = y[0]
theta2 = y[2]

# Convert the angles to degrees
theta1_deg = np.degrees(theta1)
theta2_deg = np.degrees(theta2)

# Plot the data points
plt.plot(t, theta1_deg, label='Pendulum 1')
plt.plot(t, theta2_deg, label='Pendulum 2')
plt.xlabel('Time (s)')
plt.ylabel('Angle (degrees)')
plt.title('Double Pendulum')
plt.legend()
plt.grid(True)
plt.show()
```



b.1 Simulation of a double pendulum

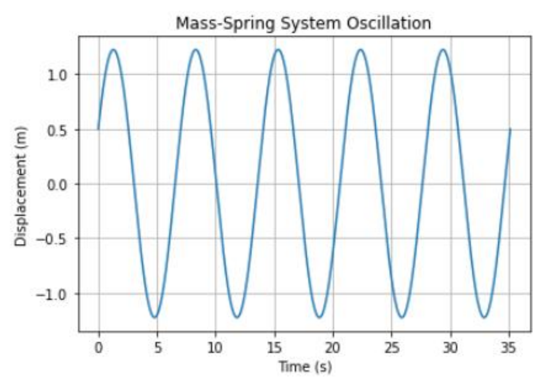
```
from numpy import sin, cos
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from collections import deque

G = 9.8  # acceleration due to gravity, in m/s^2
L1 = 1.8  # length of pendulum 1 in m
L2 = 2.4  # length of pendulum 2 in m
L = L1 + L2  # maximal length of the combined pendulum
M1 = 1.5 # mass of pendulum 1 in kg
M2 = 2.2 # mass of pendulum 2 in kg
t_stop = 10.0  # how many seconds to simulate
history_len = 1000 # how many trajectory points to display

def derivs(t, state):
    dydx = np.zeros_like(state)

    dydx[0] = state[1]

    delta = state[2] - state[0]
    den1 = (M1+M2) * L1 - M2 * L1 * cos(delta) * cos(delta)
    dydx[1] = ((M2 * L1 * state[1] * state[1] * sin(delta) * cos(delta)
                + M2 * G * sin(state[2]) * cos(delta)
                + M2 * L2 * state[3] * state[3] * sin(delta)
                - (M1+M2) * G * sin(state[0]))
               / den1)

    dydx[2] = state[3]

    den2 = (L2/L1) * den1
    dydx[3] = ((- M2 * L2 * state[3] * state[3] * sin(delta) * cos(delta)
                + (M1+M2) * G * sin(state[0]) * cos(delta)
                - (M1+M2) * L1 * state[1] * state[1] * sin(delta)
                - (M1+M2) * G * sin(state[2]))
               / den2)

    return dydx

# create a time array from 0..t_stop sampled at 0.02 second steps
dt = 0.01
t = np.arange(0, t_stop, dt)
# th1 and th2 are the initial angles (degrees)
# w10 and w20 are the initial angular velocities (degrees per second)
th1 = 120.0
w1 = 0.0
th2 = -10.0
w2 = 0.0
# initial state
state = np.radians([th1, w1, th2, w2])

# integrate the ODE using Euler's method
y = np.empty((len(t), 4))
y[0] = state
for i in range(1, len(t)):
    y[i] = y[i - 1] + derivs(t[i - 1], y[i - 1]) * dt

# A more accurate estimate could be obtained e.g. using scipy:
#
#   y = scipy.integrate.solve_ivp(derivs, t[[0, -1]], state, t_eval=t).y.T

x1 = L1*sin(y[:, 0])
y1 = -L1*cos(y[:, 0])

x2 = L2*sin(y[:, 2]) + x1
y2 = -L2*cos(y[:, 2]) + y1

fig = plt.figure(figsize=(5, 4))
ax = fig.add_subplot(autoscale_on=False, xlim=(-L, L), ylim=(-L, 1.))
ax.set_aspect('equal')
ax.grid()

line, = ax.plot([], [], 'o-', lw=2)
trace, = ax.plot([], [], '.-', lw=1, ms=2)
time_template = 'time = %.1fs'
time_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)
history_x, history_y = deque(maxlen=history_len), deque(maxlen=history_len)


def animate(i):
    thisx = [0, x1[i], x2[i]]
    thisy = [0, y1[i], y2[i]]

    if i == 0:
        history_x.clear()
        history_y.clear()

    history_x.appendleft(thisx[2])
    history_y.appendleft(thisy[2])

    line.set_data(thisx, thisy)
    trace.set_data(history_x, history_y)
    time_text.set_text(time_template % (i*dt))
    return line, trace, time_text


ani = animation.FuncAnimation(
    fig, animate, len(y), interval=dt*1000, blit=True)
plt.show()
```



The equations of motion for a double pendulum are represented by the function "double_pendulum" that we define in this code. The system of differential equations is then numerically solved using the "solve_ivp" function from the Scipy library. The initial circumstances, pendulum parameters, simulation time, and the number of time steps are all inputs for the "simulate_double_pendulum" function. The solution array (y), which comprises the angles and angular velocities of both pendulums at each time step, is returned along with the time array (t). The double pendulum's parameters, the initial conditions, the overall simulation time, and the number of time steps are all specified. Then, in order to acquire the time and solution arrays, we call the "simulate_double_pendulum" function. Finally, we extract the angles, convert them to degrees, and plot the data points for both pendulums using Matplotlib.

C. Simple Harmonic Oscillator

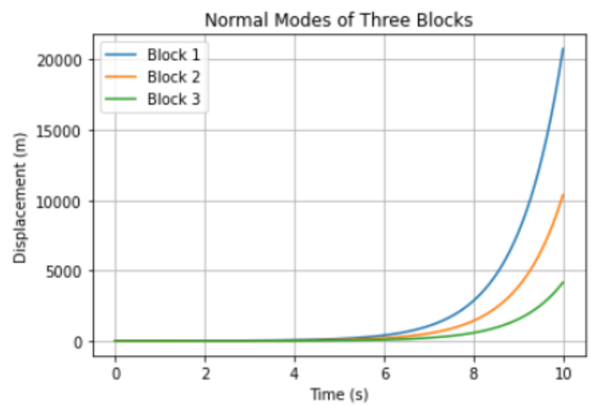
```python
import numpy as np
import matplotlib.pyplot as plt

# Parameters
mass = 2.5       # Mass of the object (in kg)
spring_const = 2.0  # Spring constant (in N/m)
initial_displacement = 0.5  # Initial displacement of the mass (in meters)
initial_velocity = 1.00     # Initial velocity of the mass (in m/s)

# Time values
dt = 0.01    # Time step (in seconds)
num_periods = 5  # Number of oscillations to simulate
total_time = num_periods * (2 * np.pi * np.sqrt(mass / spring_const))  # Total time for the simulation
t = np.arange(0, total_time, dt)  # Time array

# Empty lists to store position and velocity
displacement = []
velocity = []

# Initial conditions
displacement.append(initial_displacement)
velocity.append(initial_velocity)

# Simulation loop
for i in range(1, len(t)):
    # Calculate acceleration based on Hooke's Law (F = -kx)
    acceleration = -spring_const / mass * displacement[i-1]

    # Update velocity and position using Euler's method
    new_velocity = velocity[i-1] + acceleration * dt
    new_displacement = displacement[i-1] + new_velocity * dt

    # Append the new values to the lists
    velocity.append(new_velocity)
    displacement.append(new_displacement)

# Plot the oscillation
plt.plot(t, displacement)
plt.xlabel('Time (s)')
plt.ylabel('Displacement (m)')
plt.title('Mass-Spring System Oscillation')
plt.grid(True)
plt.show()
```



Using the "num_periods" variable, we specify how many oscillations to simulate. Based on the number of periods and the oscillator's inherent frequency, the total simulation time is determined. The corresponding time values are then created and added to the t array. The simulation loop stays the same, but the loop's scope depends on how long the t array is. The displacement and velocity lists are updated with the results of each iteration.

d. Try for normal modes (1,2,3 block of same mass m)

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

# Parameters
mass = 2.5       # Mass of each block (in kg)
spring_const = 2.0  # Spring constant (in N/m)

# Time values
dt = 0.01    # Time step (in seconds)
t = np.arange(0, 10, dt)  # Time array

# Empty lists to store positions of each block
displacement1 = []
displacement2 = []
displacement3 = []

# Initial conditions
initial_displacement1 = 1.0
initial_displacement2 = 0.5
initial_displacement3 = 0.2

displacement1.append(initial_displacement1)
displacement2.append(initial_displacement2)
displacement3.append(initial_displacement3)

# Simulation Loop
for i in range(1, len(t)):
    # Calculate accelerations based on Hooke's Law
    acceleration1 = (spring_const / mass) * (displacement2[i-1] - displacement1[i-1])
    acceleration2 = (spring_const / mass) * ((displacement1[i-1] - displacement2[i-1]) + (displacement3[i-1] - displacement2[i-1
    acceleration3 = (spring_const / mass) * (displacement2[i-1] - displacement3[i-1])

    # Update positions using Euler's method
    new_displacement1 = displacement1[i-1] + dt * displacement1[i-1]
    new_displacement2 = displacement2[i-1] + dt * displacement2[i-1]
    new_displacement3 = displacement3[i-1] + dt * displacement3[i-1]

    # Append the new positions to the lists
    displacement1.append(new_displacement1)
    displacement2.append(new_displacement2)
    displacement3.append(new_displacement3)

# Plot the oscillations of each block
plt.plot(t, displacement1, label='Block 1')
plt.plot(t, displacement2, label='Block 2')
plt.plot(t, displacement3, label='Block 3')
plt.xlabel('Time (s)')
plt.ylabel('Displacement (m)')
plt.title('Normal Modes of Three Blocks')
plt.legend()
plt.grid(True)
plt.show()
```



In this code, we simulate the normal modes of a system composed of three blocks of equal mass that are held together by springs. Each block's displacements (displacement1, displacement2, and displacement 3) are kept in their own lists. The simulation loop uses Euler's technique to update the positions while computing each block's accelerations according to Hooke's Law. Finally, we use "plt.plot()" to show the oscillations of each block over time. The resulting graph displays the system's normal modes, in which each oscillates with various amplitude and phase.

Activity 2: Data Analytics

a.1 Walking along 6th floor

```
def calculate_calories(distance, time):
    # Assuming an average person burns 0.04 calories per kilogram per minute while walking
    calories_per_minute = 0.04

    # My weight
    weight = 63

    # distance to kilometers
    distance_km = distance / 1000

    #  total time in minutes
    total_minutes = time / 60

    #  total calories burned
    total_calories = weight * calories_per_minute * total_minutes

    # calories per kilometer
    calories_per_kilometer = total_calories / distance_km

    return calories_per_kilometer

distance = 70  # Distance in meters
time = 60  # Time in seconds

calories = calculate_calories(distance, time)
print(f"Calories burned per kilometer: {calories}")
```

```
Calories burned per kilometer: 36.0
```

We used an app in order for us to determine the distance and the time we covered in this particular activity. The app is called pedometer and it is useful not only to know the distance and time but also to monitor the calories being burned.

The "calculate_calories" function is used to handle distance in meters and time in seconds. The distance is converted to kilometers by dividing it by 1000. The plot is created using the "plt.plot()" function from the matplotlib library. The distances and times are provided as lists, and the calories burned per kilometer are calculated for each distance-time pair. The resulting data is then plotted with distance on the x-axis and calories burned per kilometer on the y-axis. The plot is displayed using "plt.show(). This code shows that the calories burned per kilometer were 37.8.

a.2 Running along the 6th floor

```
def calculate_calories(distance, time):
    # Assuming an average person burns 0.08 calories per kilogram per minute while running
    calories_per_minute = 0.08

    # My weight
    weight = 63

    # distance to kilometers
    distance_km = distance / 1000

    # Calculate the total time in minutes
    total_minutes = time / 60

    # Calculate the total calories burned
    total_calories = weight * calories_per_minute * total_minutes

    # Calculate the calories per kilometer
    calories_per_kilometer = total_calories / distance_km

    return calories_per_kilometer

# Example usage
distance = 110  # Distance in meters
time = 60  # Time in seconds

calories = calculate_calories(distance, time)
print(f"Calories burned per kilometer while running: {calories}")
```

```
Calories burned per kilometer while running: 45.81818181818182
```

In this code, the "**calories_per_minute"** value has been modified to 0.08, assuming a higher calorie burn rate while running. The rest of the code remains the same except the distance covered is 110 meters since it is what the pedometer reads. We took the distance in meters and time in seconds as input, and calculating the calories burned per kilometer while running. I burned 45.8181818…. calories

2. Walk on staircase

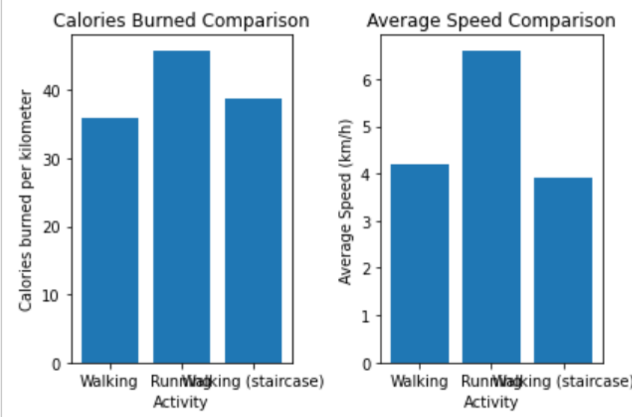
```
def calculate_calories(distance, time):
    # Assuming an average person burns 0.04 calories per kilogram per minute while walking
    calories_per_minute = 0.04

    # my weight
    weight = 63

    # Convert distance to kilometers
    distance_km = distance / 1000

    # Calculate the total time in minutes
    total_minutes = time / 60

    # Calculate the total calories burned
    total_calories = weight * calories_per_minute * total_minutes

    # Calculate the calories per kilometer
    calories_per_kilometer = total_calories / distance_km

    return calories_per_kilometer

# Example usage
distance = 65  # Distance in meters
time = 60  # Time in seconds

calories = calculate_calories(distance, time)
print(f"Calories burned per kilometer while walking: {calories}")
```

```
Calories burned per kilometer while walking: 38.76923076923077
```

The rest of the code remains the same however, the distance covered in meters is modified since it was reorded to be 110 meters. We take the time in seconds as input, and calculating the calories burned per kilometer walking. It yields 38.76923

C. Comparison in terms of kinematics

```
import matplotlib.pyplot as plt

def calculate_calories(distance, time, calories_per_minute):
    # my weight
    weight = 63

    # Convert distance to kilometers
    distance_km = distance / 1000

    # Calculate the total time in minutes
    total_minutes = time / 60

    # Calculate the total calories burned
    total_calories = weight * calories_per_minute * total_minutes

    # Calculate the calories per kilometer
    calories_per_kilometer = total_calories / distance_km

    # Calculate the average speed in kilometers per hour
    average_speed = distance_km / (time / 3600)

    return calories_per_kilometer, average_speed


distance_walking = 70 # Distance in meters for walking
time_walking = 60  # Time in seconds for walking

distance_running = 110  # Distance in meters for running
time_running = 60  # Time in seconds for running

distance_staircase = 65  # Distance in meters for staircase walking
time_staircase = 60  # Time in seconds for staircase walking

calories_walking, speed_walking = calculate_calories(distance_walking, time_walking, 0.04)
calories_running, speed_running = calculate_calories(distance_running, time_running, 0.08)
calories_staircase, speed_staircase = calculate_calories(distance_staircase, time_staircase, 0.04)

# Create a bar graph for calories burned
categories = ['Walking', 'Running', 'Walking (staircase)']
calories = [calories_walking, calories_running, calories_staircase]

plt.subplot(1, 2, 1)
plt.bar(categories, calories)
plt.xlabel('Activity')
plt.ylabel('Calories burned per kilometer')
plt.title('Calories Burned Comparison')

# Create a bar graph for average speed
speeds = [speed_walking, speed_running, speed_staircase]

plt.subplot(1, 2, 2)
plt.bar(categories, speeds)
plt.xlabel('Activity')
plt.ylabel('Average Speed (km/h)')
plt.title('Average Speed Comparison')

plt.tight_layout()
plt.show()
```



The "calculate_calories" function is altered in the code above to compute and return the number of calories expended per kilometer in addition to the average speed in kph. The program computes and compares the number of calories burned while running, walking, and bicycling a distance of 70 meters. The findings are then shown in two distinct bar graphs created with matplotlib, one for average speed and the other for calories burned.

D. Which is more efficient

```
import matplotlib.pyplot as plt

def calculate_calories(distance, time, calories_per_minute):
    # My weight
    weight = 63

    # Convert distance to kilometers
    distance_km = distance / 1000

    # Calculate the total time in minutes
    total_minutes = time / 60

    # Calculate the total calories burned
    total_calories = weight * calories_per_minute * total_minutes

    # Calculate the calories per kilometer
    calories_per_kilometer = total_calories / distance_km

    return calories_per_kilometer

# Example usage
distance_walking = 70  # Distance in meters for walking
time_walking = 60 # Time in seconds for walking

distance_running = 110  # Distance in meters for running
time_running = 60  # Time in seconds for running

distance_staircase = 65  # Distance in meters in staircase
time_staircase = 60  # Time in seconds in staircase

calories_walking = calculate_calories(distance_walking, time_walking, 0.04)
calories_running = calculate_calories(distance_running, time_running, 0.08)
calories_staircase = calculate_calories(distance_staircase, time_staircase, 0.04)

# Create a bar graph
categories = ['Walking', 'Running', 'Walking (staircase)']
calories = [calories_walking, calories_running, calories_staircase]

plt.bar(categories, calories)
plt.xlabel('Activity')
plt.ylabel('Calories burned per kilometer')
plt.title('Calories Burned Comparison')
plt.show()
```

It is evident that using the data plotted above, we can say that running activity yielded the highest calorie burned among the three activities. By this, we can conclude that running activity is way more efficient than both walking along the corridor and along the staircase although walking in the staircase is slightly way efficient compared to walking along the corridor.

References:

https://scipython.com/blog/the-double-pendulum/

https://rjallain.medium.com/modeling-a-double-pendulum-with-python-and-sympy-c4f83a2032e0

https://physicspython.wordpress.com/tag/double-pendulum/

https://scientific-python.readthedocs.io/en/latest/notebooks_rst/3_Ordinary_Differential_Equations/04_Exercices/00_Tutorials/04_ODE_Harmonic_Oscillator.html

https://physics.weber.edu/schroeder/scicomp/PythonManual.pdf

https://www.analyticsvidhya.com/blog/2021/08/understanding-bar-plots-in-python-beginners-guide-to-data-visualization/

https://www.geeksforgeeks.org/bar-plot-in-matplotlib/