Modeling and Simulation - Lecture Notes - 6

Simulation of Linear Systems with Python

Linear systems are a class of systems for which the principle of superposition applies. They are widely used in engineering, physics, and other scientific fields due to their relative simplicity and the powerful analytical tools that exist for their analysis.

By definition, a linear system is one that satisfies the principle of superposition. This means that the output of the system, when subject to multiple inputs, is simply the sum of the outputs that would result from each input applied separately. In mathematical terms, a system is linear if it satisfies the following two properties:

- Additivity: For all inputs $x_1(t)$ and $x_2(t)$, and corresponding outputs $y_1(t)$ and $y_2(t)$, the system's response to the sum of the inputs is the sum of the outputs: $y(x_1(t) + x_2(t)) = y_1(t) + y_2(t)$.
- Homogeneity (or scalability): For any input $x(t)$ and corresponding output $y(t)$, and any scalar a, the system's response to the scaled input is the scaled output: $y(a \cdot x(t)) = a \cdot y(t)$.

From a mathematical perspective, linear systems are typically described using linear differential equations or systems of linear equations, involving operations of differentiation, integration, addition and scalar multiplication.

Linear systems arise in a wide range of applications, including electrical circuits, control systems, signal processing, and mechanical and civil engineering structures, to name just a few. They can often be solved analytically, and for more complex systems, various numerical methods are available for obtaining approximate solutions.

Linear systems theory provides the foundation for the analysis and design of many engineered systems and processes. Despite the simplifying assumptions of linearity, linear system models often provide reasonably accurate descriptions of real-world phenomena and offer useful insights into system behavior.

Examples of Linear and Nonlinear Systems

Linear Systems

Mechanics: Mass-Spring-Damper System

The mass-spring-damper system is a classic example of a linear system in mechanics. This system consists of a mass attached to a spring and a damper (or dashpot). The spring exerts a force proportional to the displacement of the mass ($F = kx$, where k is the spring constant and x is the displacement), and the damper provides a damping force proportional to the velocity of the mass ($F = b\dot{x}$, where b is the damping coefficient and $\dot{x}$ is the velocity). The sum of these forces equals the mass times its acceleration ($F = ma$), leading to the following linear differential equation:

$$m\ddot{x} + b\dot{x} + kx = 0 \tag{1}$$

Electronics: RC Circuit

In electronics, a simple resistor-capacitor (RC) circuit is a linear system. The output voltage across the capacitor, $V_C(t)$, is determined by the charge on the capacitor, $Q(t)$, and the relationship between the current, $I(t)$, the voltage across the resistor, $V_R(t)$, and the input voltage, $V_{in}(t)$, is governed by Kirchhoff's voltage law and Ohm's law:

$$V_{in}(t) = V_R(t) + V_C(t) = I(t)R + \frac{Q(t)}{C} \tag{2}$$

where R is the resistance, and C is the capacitance. This leads to the following linear differential equation:

$$RC\frac{dV_C}{dt} + V_C = V_{in} \tag{3}$$

Nonlinear Systems

Mechanics: Pendulum

A simple pendulum (a mass hanging from a string or rod of fixed length and swinging in a plane) is a nonlinear system. The equation of motion for a simple pendulum of length L with small angle $\theta(t)$ is:

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}\sin(\theta) = 0 \tag{4}$$

This equation is nonlinear because of the $\sin(\theta)$ term. For small angles, we can approximate $\sin(\theta) \approx \theta$ and the system becomes linear.

Electronics: Diode Circuit

In electronics, a simple circuit containing a diode is an example of a nonlinear system. The current-voltage relationship for a diode is governed by the Shockley diode equation, which is nonlinear:

$$I = I_0(e^{V_D/(nV_T)} - 1) \tag{5}$$

where I is the diode current, V_D is the diode voltage, I_0 is the reverse saturation current, n is the ideality factor, and V_T is the thermal voltage.

Complex Linear Systems

Electronics: Operational Amplifiers

Operational Amplifiers (OpAmps) are key building blocks in modern electronic devices. The behavior of an OpAmp circuit is described by linear differential equations. For instance, the inverting amplifier configuration is described by the equation:

$$V_{out}(t) = -R_f \frac{1}{R_i C} \int_{-\infty}^{t} V_{in}(\tau) d\tau \tag{6}$$

where $V_{out}(t)$ and $V_{in}(t)$ are the output and input voltages, R_f is the feedback resistor, R_i is the input resistor, and C is the input capacitance.

Mechanics: Multi-Degree-of-Freedom Systems

In mechanical engineering, systems with multiple degrees of freedom can be described by a set of coupled linear differential equations. For example, a system with two masses, two springs, and two dampers has the following set of differential equations:

$$m_1\ddot{x}_1 + b_1\dot{x}_1 + k_1x_1 - b_2\dot{x}_2 - k_2x_2 = 0 \tag{7}$$

$$m_2\ddot{x}_2 + b_2\dot{x}_2 + k_2x_2 - b_1\dot{x}_1 - k_1x_1 = 0 \tag{8}$$

where x_1 and x_2 are the displacements of the two masses.

Complex Nonlinear Systems

Electronics: Transistor Circuits

Transistor circuits are examples of nonlinear systems due to the exponential relationship between the input and output in the transistor's active region. The Ebers-Moll model for a bipolar junction transistor (BJT) is:

$$I_C = I_S \left(e^{\frac{V_{BE}}{V_T}} - 1 \right) - I_S \left(e^{\frac{V_{BC}}{V_T}} - 1 \right) \tag{9}$$

where I_C is the collector current, I_S is the reverse saturation current, V_{BE} and V_{BC} are the base-emitter and base-collector voltages, and V_T is the thermal voltage.

Mechanics: Pendulum with Resistance

A pendulum with air resistance is a more complex nonlinear system. Its equation of motion is:

$$\frac{d^2\theta}{dt^2} + b\frac{d\theta}{dt} + \frac{g}{L}\sin(\theta) = 0 \tag{10}$$

where $\theta(t)$ is the angle of the pendulum, L is the length of the pendulum, g is the acceleration due to gravity, and b is the damping factor representing air resistance. This equation is nonlinear due to the $\sin(\theta)$ term and the presence of the damping term $b\frac{d\theta}{dt}$.

Control Engineering: PID Controller

A Proportional-Integral-Derivative (PID) controller is a widely used control scheme in industry. It is a linear system and can be described by the following differential equation:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt}e(t) \tag{11}$$

where $u(t)$ is the control signal, $e(t)$ is the error signal (difference between the setpoint and the process variable), and K_p, K_i, and K_d are the proportional, integral, and derivative gains, respectively. This equation describes the system behavior in the time domain.

Control Engineering: Nonlinear Control Systems

An example of a nonlinear control system is the control of an inverted pendulum on a cart. This is a classic problem in control theory and is often used to illustrate the concepts of feedback and stabilization. The dynamics of the system are described by a set of nonlinear differential equations, which are derived from the laws of physics:

$$(m_c + m_p)\ddot{x} + m_p l\ddot{\theta}\cos(\theta) - m_p l\dot{\theta}^2\sin(\theta) = F \tag{12}$$

$$m_p l\ddot{\theta} - m_p g\sin(\theta) = -m_p l\ddot{x}\cos(\theta) \tag{13}$$

where x is the position of the cart, θ is the angle of the pendulum, m_c is the mass of the cart, m_p is the mass of the pendulum, l is the length of the pendulum, g is the acceleration due to gravity, and F is the force applied to the cart. The goal is to design a controller that applies a force F to the cart to balance the pendulum upright ($\theta = 0$) and control the position of the cart.

This is challenging due to the nonlinear nature of the system and the fact that the system is underactuated (we can only control the force on the cart, but we want to control both the position of the cart and the angle of the pendulum).

Significance of Linear Systems and Simulation of Non-Linear Systems

Linear systems play a crucial role in the understanding of more complex, nonlinear systems. They serve as a stepping stone, introducing fundamental concepts and methodologies that are applicable in a broader context. The mathematical tractability of linear systems makes them particularly suitable for analysis and understanding. Concepts like stability, time response, frequency response, and the notions of control and optimization, are all elegantly handled within the framework of linear systems theory.

That said, in real-world applications, many systems exhibit nonlinear behaviors. Nonlinear systems can exhibit complex behaviors like limit cycles, bifurcations, and chaos, which are not seen in linear systems. While linear systems theory can sometimes be used to analyze nonlinear systems locally around an equilibrium point (a method known as linearization), the resulting linear models are only valid in a small region around the equilibrium point.

Nonlinear systems can be described by nonlinear differential equations. Simulation of nonlinear systems often involves numerical methods, since most nonlinear differential equations cannot be solved analytically. One common method for simulating nonlinear systems is the method of numerical integration, such as Euler's method or the Runge-Kutta methods.

```
import numpy as np
from scipy.integrate import odeint

# define system in terms of a Numpy array
def Sys(X, t=0):
    # here X[0] = x and X[1] = y
    return np.array([ 3*X[0] - X[0]**2 - 2*X[0]*X[1] ,
    -X[1] + X[0]*X[1] ])

# generate time points
t = np.linspace(0, 10, 200)
# initial values: x = 10, y = 2
Sys0 = np.array([10, 2])

# solve ODE
X, infodict = odeint(Sys, Sys0, t, full_output=True)
infodict['message']
```

This Python code simulates a simple nonlinear system with two variables x and y, described by the differential equations $\dot{x} = 3x - x^2 - 2xy$ and $\dot{y} = -y + xy$, respectively. The `odeint` function from the `scipy.integrate` module is used to numerically integrate the differential equations.

The simulation of nonlinear systems is a vast and complex field. There are many different techniques for simulating different types of nonlinear systems, and there is ongoing research to develop better and more efficient methods.

Advanced Techniques for Nonlinear System Simulation

As the complexity of nonlinear systems increases, so does the complexity of the methods required to analyze and simulate them. Here are some advanced techniques that are commonly used:

Perturbation Methods

Perturbation methods can be used when the nonlinear system includes a small parameter. The small parameter is considered as a perturbation and the system is solved as a series of approximations.

Describing Function Method

The describing function method is a technique used for analyzing certain types of nonlinear control systems. The method is particularly useful when the system includes nonlinearities such as saturation or dead-zone nonlinearity.

Phase Plane Analysis

Phase plane analysis involves examining the trajectories of a system in a 2D plane, known as the phase plane.

Lyapunov's Direct Method

Lyapunov's direct method is a powerful technique used to assess the stability of a nonlinear system. Although this method does not provide a solution to the system, it offers valuable insights about its behavior.

Techniques for Simulating Linear Systems

Simulating a linear system typically involves constructing a mathematical model of the system, implementing this model in code, and then performing the simulation to observe the system's behavior over time. There are a variety of mathematical techniques and tools available for simulating linear systems.

Modeling Linear Systems

Linear systems can often be modeled using linear differential equations or systems of linear equations. The process of modeling a system involves identifying the system's inputs, outputs, and internal states, and then writing down equations that describe how these are related.

The set of all outputs resulting from any given input forms the state space of the system, a concept that is key to understanding linear systems. For a simple linear system with one input and one output (SISO), the state space may be a two-dimensional plane. For more complex systems with multiple inputs and outputs (MIMO), the state space may have many more dimensions.

Linear differential equations are a cornerstone of system dynamics and control, allowing us to describe physical systems, like electrical circuits, mechanical systems, thermal systems, or more abstract ones like economies and populations. The solutions of these equations describe the time-evolution of the system.

Implementing the Model in Python

Once a mathematical model has been constructed, it can be implemented in Python using the numpy and scipy libraries. These libraries provide high-level functions that abstract away much of the complexity of the numerical computations. For example, the ‘odeint‘ function from scipy can be used to solve systems of ordinary differential equations, while the ‘linalg‘ module from numpy can be used to solve systems of linear equations.

A typical workflow might look like this:

1. Define the system of equations that models your system.
2. Define the time span over which you want to simulate the system.
3. Set initial conditions.
4. Call ‘odeint‘ or other suitable solver to numerically integrate the system of equations.
5. Analyze and visualize the results using, for example, ‘matplotlib‘ or other data visualization tools.

It's important to mention that numerical solutions only approximate the true solutions. The accuracy of the approximation depends on the solver used and the time step size, among other factors. Nonetheless, for many practical purposes, the numerical solutions obtained are often sufficiently accurate.

The following is a Python example of a simple harmonic oscillator modeled as a second-order linear system. This is a simple physical system, consisting of a mass, a spring, and a damper:

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

# system parameters
m = 1.0  # mass
k = 0.5  # spring constant
b = 0.2  # damping constant

# system equations
def model(y, t):
    x, v = y  # position, velocity
    dxdt = v
    dvdt = -(b/m)*v - (k/m)*x
    return [dxdt, dvdt]

# initial condition
y0 = [1, 0]  # initial position and velocity

# time points
t = np.linspace(0, 20, 1000)

# solve ODE
y = odeint(model, y0, t)

# plot results
plt.figure()
plt.plot(t, y[:, 0], label='x(t)')
plt.plot(t, y[:, 1], label='v(t)')
plt.legend()
plt.show()
```

Numerical Simulation with Python's ODEINT

The ‘odeint‘ function is part of the ‘scipy.integrate‘ module in Python and is used to numerically solve systems of ordinary differential equations (ODEs). This function can solve both linear and nonlinear ODEs.

Under the hood, ‘odeint‘ uses advanced numerical methods, specifically a method known as "LSODA" (Livermore Solver for Ordinary Differential equations with Automatic method switching for stiff and nonstiff problems). This method automatically chooses between two methods: Adams method for nonstiff systems and BDF (Backward Differentiation Formula) for stiff systems.

In terms of usage, ‘odeint‘ takes three main arguments:

- A function that defines the ODEs.
- An initial condition vector.
- A sequence of time points at which to solve for the dependent variables.

Here's a quick example of how it works. Let's consider a simple first-order ODE, also known as an exponential decay model:

```
from scipy.integrate import odeint
import numpy as np

# function that returns dy/dt
def model(y, t):
    k = 0.3
    dydt = -k * y
    return dydt

# initial condition
y0 = 5

# time points
t = np.linspace(0, 20)

# solve ODE
y = odeint(model, y0, t)
```

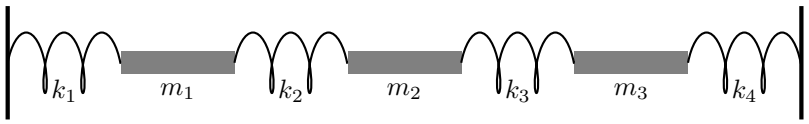
In this example, ‘model‘ is a function that returns the derivative of y at any time t given the current state y (it's the function that represents the ODE). ‘y0‘ is the initial condition, and ‘t‘ is an array that contains the time points at which the solution will be evaluated. The ‘odeint‘ function returns an array ‘y‘ which is the solution of the differential equation at each point in ‘t‘.

For systems of equations, ‘y‘ and ‘y0‘ would be vectors, and the ‘model‘ function would return a vector of derivatives.

Project Sample: Simulating a Mass-Spring System

In this project, you will model and simulate the behavior of a mass-spring system consisting of three masses and four springs. The system is depicted in the following diagram:

Here we will assume that the right-most spring is attached to a wall and its other end is connected to mass m_3. Also, let's denote the displacements of the three masses from their equilibrium positions by x_1, x_2, and x_3. The equilibrium positions are when all the springs are at their natural lengths and the system is at rest. We'll define these displacements as positive to the right.



The equations of motion for the masses are given by Newton's second law:

$$
\begin{aligned}
m_1\ddot{x}_1 &= -k_1x_1 + k_2(x_2 - x_1) \\
m_2\ddot{x}_2 &= -k_2(x_2 - x_1) + k_3(x_3 - x_2) \\
m_3\ddot{x}_3 &= -k_3(x_3 - x_2) - k_4x_3
\end{aligned}
$$

In these equations, $\ddot{x}_i$ is the second derivative of x_i with respect to time, which represents the acceleration of mass m_i. The term k_ix_i represents the force exerted by spring k_i on mass m_i due to its displacement from the equilibrium position, and the term $k_i(x_{i+1} - x_i)$ represents the force exerted by spring k_i on mass m_i due to the difference in displacements of mass m_i and mass m_{i+1}.

Your tasks in this project are as follows:

1. Write a Python program that uses the 'odeint' function from 'scipy.integrate' to simulate this system. Assume the masses are all 1 kg, the spring constants are all 100 N/m. Choose suitable initial conditions.
2. Use your program to simulate the system for different sets of parameters and initial conditions.
3. Analyze and discuss the results. What happens when you change the spring constants or the masses? What happens when you change the initial conditions?
4. (Optional) Extend your program to handle more masses and springs. How does the behavior of the system change as you add more masses and springs?

```
# Import necessary libraries
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# Define system of ODEs
def system(w, t, p):
    """
    Defines the differential equations for the coupled spring-mass system.

    Arguments:
        w :  vector of the state variables:
                  w = [x1, y1, x2, y2, x3, y3]
        t :  time
        p :  vector of the parameters:
                  p = [m1, m2, m3, k1, k2, k3, k4]
    """
    x1, y1, x2, y2, x3, y3 = w
    m1, m2, m3, k1, k2, k3, k4 = p

    # Create sysODE = (x1', y1', x2', y2', x3', y3')
    sysODE = [y1,
              (-k1 * x1 + k2 * (x2 - x1)) / m1,
              y2,
              (-k2 * (x2 - x1) + k3 * (x3 - x2)) / m2,
              y3,
              (-k3 * (x3 - x2) - k4 * x3) / m3]
    return sysODE

# Parameter values
# Masses:
m1 , m2 , m3 = 1.0, 2.0, 3.0
# Spring constants
k1 = k2 = k3 = k4 = 20.0

# Initial conditions
# x1 and x2 are the initial displacements; y1 and y2 are the initial velocities
x1 , x2 , x3 = 0.1, 0.0, 0.0
y1 = y2 = y3 = 0.0

# ODE solver parameters
abserr = 1.0e-8
relerr = 1.0e-6
stoptime = 10.0
numpoints = 2000
```

```
# Create the time samples for the output of the ODE solver.
t = np.linspace(0, stoptime, numpoints)

# Pack up the parameters and initial conditions:
p = [m1, m2, m3, k1, k2, k3, k4]
w0 = [x1, y1, x2, y2, x3, y3]

# Call the ODE solver.
wsol = odeint(system, w0, t, args=(p,), atol=abserr, rtol=relerr)

# Plot the solution
plt.figure()
plt.plot(t, wsol[:, 0], label="x1")
plt.plot(t, wsol[:, 2], label="x2")
plt.plot(t, wsol[:, 4], label="x3")
plt.legend(loc='best')
plt.title('Mass Displacements for the\nCoupled Spring-Mass System')
plt.xlabel('Time (s)')
plt.ylabel('Displacement (m)')
plt.grid()
plt.show()
```

These are the parameters for the ODE solver, odeint, which is used in the Python code to solve the system of ordinary differential equations:

abserr = 1.0e-8: This is the absolute error tolerance for the ODE solver. It's used to determine when the solver should stop refining its solution because further refinements wouldn't significantly change the result. Lower values make the solver more accurate, but also make it slower.

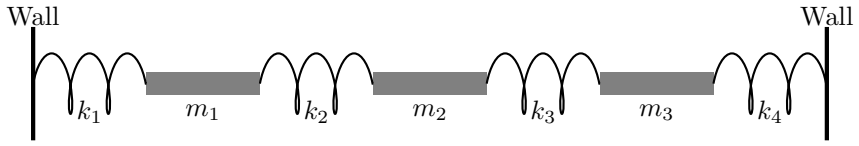
relerr = 1.0e-6: This is the relative error tolerance for the ODE solver. It works similarly to the absolute error tolerance, but it scales with the size of the solution. This means that the solver will tolerate larger errors when the solution is large and smaller errors when the solution is small. Like the absolute error tolerance, lower values make the solver more accurate but slower.

stoptime = 10.0: This is the time at which the simulation will stop. In this case, the simulation will run for 10 seconds of simulated time.

numpoints = 250: This is the number of points at which the solution will be sampled. The ODE solver doesn't produce a continuous function as its output, but instead produces a solution at a finite set of points in time. These points are evenly spaced between 0 and stoptime. The value of numpoints determines how many such points there will be. More points mean a more detailed solution, but also take more time to compute.

Project: Simulating a Mass-Spring System using Laplace Transforms

In this project, you will model and simulate the behavior of a mass-spring system consisting of three masses and four springs using Laplace transforms. The system is depicted in the following diagram:



The equations of motion for the masses are given by Newton's second law:

$$
\begin{aligned}
m_1\ddot{x}_1 &= -k_1x_1 + k_2(x_2 - x_1) \\
m_2\ddot{x}_2 &= -k_2(x_2 - x_1) + k_3(x_3 - x_2) \\
m_3\ddot{x}_3 &= -k_3(x_3 - x_2) - k_4x_3
\end{aligned}
$$

Your tasks in this project are as follows:

1. Use Laplace transforms by hand to solve the equations of motion for the system, assuming initial rest conditions.
2. Simulate the motion of the masses over time and visualize the results with appropriate plots.
3. Analyze the effect of different parameters (masses, spring constants, initial conditions) on the behavior of the system.
4. Extend the problem to include damping and/or external forcing. Implement the extended model in Python and analyze the behavior of the system.

A damping term and an external force can be included in the system of equations as follows:

$$\begin{aligned} m_1\ddot{x}_1 + d_1\dot{x}_1 + k_1x_1 - k_2(x_2 - x_1) &= F_1(t) \\ m_2\ddot{x}_2 + d_2\dot{x}_2 + k_2(x_2 - x_1) - k_3(x_3 - x_2) &= F_2(t) \\ m_3\ddot{x}_3 + d_3\dot{x}_3 + k_3(x_3 - x_2) - k_4x_3 &= F_3(t) \end{aligned}$$

Where d_1, d_2, and d_3 are damping coefficients and $F_1(t)$, $F_2(t)$, and $F_3(t)$ are external forces as functions of time.

```
# Import necessary libraries
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# Define system of ODEs
def system(w, t, p):
    """
    Defines the differential equations for the damped and forced coupled spring-mass system.

    Arguments:
        w :  vector of the state variables:
                  w = [x1, y1, x2, y2, x3, y3]
        t :  time
        p :  vector of the parameters:
                  p = [m1, m2, m3, k1, k2, k3, k4, d1, d2, d3, F1, F2, F3]
    """
    x1, y1, x2, y2, x3, y3 = w
    m1, m2, m3, k1, k2, k3, k4, d1, d2, d3, F1, F2, F3 = p

    # Create sysODE = (x1', y1', x2', y2', x3', y3')
    sysODE = [y1,
              (F1(t) - d1*y1 - k1*x1 + k2*(x2 - x1)) / m1,
              y2,
              (F2(t) - d2*y2 - k2*(x2 - x1) + k3*(x3 - x2)) / m2,
              y3,
              (F3(t) - d3*y3 - k3*(x3 - x2) - k4*x3) / m3]
    return sysODE

# Parameter values
# Masses:
m1 , m2 , m3 = 1.0, 1.0, 1.0
# Spring constants
k1 = k2 = k3 = k4 = 10.0
# Damping coefficients
d1 = 0.1
```

```
d2 = 0.2
d3 = 0.3
# External forces
F1 = lambda t: 1
F2 = lambda t: 0
F3 = lambda t: 0

# Initial conditions
# x1 and x2 are the initial displacements; y1 and y2 are the initial velocities
x1 , x2 , x3 = 0.0, 0.0, 0.0
y1 = y2 = y3 = 0.0

# ODE solver parameters
abserr = 1.0e-8
relerr = 1.0e-6
stoptime = 5.0
numpoints = 1000

# Create the time samples for the output of the ODE solver.
t = np.linspace(0, stoptime, numpoints)

# Pack up the parameters and initial conditions:
p = [m1, m2, m3, k1, k2, k3, k4, d1, d2, d3, F1, F2, F3]
w0 = [x1, y1, x2, y2, x3, y3]

# Call the ODE solver.
wsol = odeint(system, w0, t, args=(p,), atol=abserr, rtol=relerr)

# Plot the solution
plt.figure()
plt.plot(t, wsol[:, 0], label="x1")
plt.plot(t, wsol[:, 2], label="x2")
plt.plot(t, wsol[:, 4], label="x3")
plt.legend(loc='best')
plt.title('Mass Displacements for the\nDamped and Forced Coupled Spring-Mass System')
plt.xlabel('Time (s)')
plt.ylabel('Displacement (m)')
plt.grid()
plt.show()
```

Analysis and Visualization of System Behavior

After implementing the model and performing the simulation, the next step is to analyze and visualize the system's behavior.

Analyzing System Behavior

There are several important characteristics of a system's behavior that we may be interested in, such as its stability, its response to different inputs, and its frequency response. These can often be determined by studying the system's mathematical model and the results of the simulation.

Sensitivity Analysis

Sensitivity analysis helps us understand how the variation in the output of a model can be attributed to different sources of variation in the model's inputs. This type of analysis can be particularly useful in complex systems where many parameters are uncertain.

For example, in the mass-spring-damper system, you could examine how the system's response changes with varying levels of damping or different spring constants. This analysis can be done by running the model with different parameter values and observing the changes in the output.

```
# Define a range of damping coefficients
d1_values = np.linspace(0.1, 1.0, 5)

plt.figure()

# Run the simulation for each damping coefficient
for d1 in d1_values:
    # Update the parameter list
    p = [m1, m2, m3, k1, k2, k3, k4, d1, d2, d3, F1, F2, F3]
    wsol = odeint(system, w0, t, args=(p,), atol=abserr, rtol=relerr)

    # Plot the response of the first mass
    plt.plot(t, wsol[:, 0], label=f'd1 = {d1}')

plt.legend(loc='best')
plt.title('Effect of Varying Damping Coefficient on System Response')
plt.xlabel('Time (s)')
plt.ylabel('Displacement (m)')
plt.grid()
plt.show()
```

System Identification

Sometimes, you may not know the exact mathematical model that describes your system, but you have observational data of the system's behavior. In such a case, you can use system identification techniques to develop a model based on the data.

For example, if you have data of a mass-spring-damper system's displacement over time, you could use system identification techniques to estimate the values of the mass, damping coefficient, and spring constant. One such method is the least squares method, which can be implemented in Python using libraries like numpy or scipy.

For system identification, we usually need some observational data. However, in this case, let's assume that the displacements obtained from the simulation are our observations. We will then try to estimate the damping coefficients that could have produced these observations.

```
from scipy.optimize import minimize

# Define a cost function
def cost_function(damping_coefficients):
    d1, d2, d3 = damping_coefficients

    # Simulate the system with the given damping coefficients
    p = [m1, m2, m3, k1, k2, k3, k4, d1, d2, d3, F1, F2, F3]
    wsol = odeint(system, w0, t, args=(p,), atol=abserr, rtol=relerr)

    # Calculate the difference between the simulated and observed displacements
    diff = wsol - observed_displacements

    # Return the sum of squared differences
    return np.sum(diff**2)

# Assume that the displacements obtained from the initial simulation are the observations
observed_displacements = wsol

# Use a numerical optimization algorithm to find the damping coefficients that minimize the
initial_guess = [d1, d2, d3]
result = minimize(cost_function, initial_guess)
estimated_damping_coefficients = result.x

print(f'Estimated damping coefficients: {estimated_damping_coefficients}')
```

Multi-Objective Optimization

In many practical scenarios, you might want to optimize the system's behavior according to multiple objectives. For instance, you might want a mechanical system to reach a certain position as quickly as possible, but also with minimal overshoot and settling time. This can be achieved using multi-objective optimization techniques such as the Pareto front method.

For multi-objective optimization, let's assume that we want to minimize both the maximum displacement of the first mass and the total energy consumed by the system. The energy can be calculated as the sum of the kinetic and potential energies of the masses and springs.

```
def calculate_energy(wsol, p):
    m1, m2, m3, k1, k2, k3, k4, d1, d2, d3, F1, F2, F3 = p
    x1, y1, x2, y2, x3, y3 = np.split(wsol, 6, axis=1)

    # Calculate the kinetic and potential energies
    kinetic_energy = 0.5 * (m1*y1**2 + m2*y2**2 + m3*y3**2)
    potential_energy = 0.5 * (k1*x1**2 + k2*(x2-x1)**2 + k3*(x3-x2)**2 + k4*x3**2)

    # The total energy is the sum of kinetic and potential energies
    total_energy = np.sum(kinetic_energy + potential_energy)
    return total_energy
# Define a multi-objective cost function
def multi_objective_cost_function(damping_coefficients):
    d1, d2, d3 = damping_coefficients

    # Simulate the system with the given damping coefficients
    p = [m1, m2, m3, k1, k2, k3, k4, d1, d2, d3, F1, F2, F3]
    wsol = odeint(system, w0, t, args=(p,), atol=abserr, rtol=relerr)

    # The first objective is to minimize the maximum displacement of the first mass
    objective1 = np.max(wsol[:, 0])
    # The second objective is to minimize the total energy consumed by the system
    objective2 = calculate_energy(wsol, p)

    # The cost function is a weighted sum of the two objectives
    return 0.5 * objective1 + 0.5 * objective2

# Use a numerical optimization algorithm to find the damping coefficients that minimize the
result = minimize(multi_objective_cost_function, initial_guess)
optimized_damping_coefficients = result.x
print(f'Optimized damping coefficients: {optimized_damping_coefficients}')
```

0.1 Drone Control System

For example, if you have a drone control system, you might want to optimize the drone's path to minimize both the travel time and the energy consumption. This can be done by defining a cost function that takes into account both objectives, and then using an optimization algorithm to find the optimal control inputs.

In this example, we implemented a simple drone control system. The aim was to find the optimal control inputs (in terms of thrust and angle) that minimize the total cost, which was defined as a combination of the drone's travel time and energy consumption. The drone's motion was modeled as a simple physical system affected by gravity and its control inputs.

Here's what each part of the code does:

Import necessary libraries: We imported libraries needed for computations (numpy), differential equation solving (*scipy.integrate.odeint*), optimization (*scipy.optimize.minimize*), and plotting (*matplotlib.pyplot*).

Define the drone model: We represented the drone's motion using a system of ordinary differential equations (ODEs). The ODEs are defined by the drone_model function, which computes the drone's acceleration based on its current state and the control inputs (thrust and angle).

Define the cost function: The $cost_function$ represents the objective that we want to minimize during optimization. This function first simulates the drone's motion given a set of control inputs, and then computes the total cost as a weighted sum of the drone's travel time and energy consumption.

Optimize control inputs: Using Scipy's minimize function, we sought the set of control inputs that minimize the cost function. An initial guess for the thrust and angle was provided, and the optimization process iteratively adjusted these values to find a solution that yields the lowest possible cost.

Simulate drone's motion with optimized inputs: After obtaining the optimal control inputs, we simulated the drone's motion again using these inputs.

Visualize the results: Finally, we plotted the drone's path as a function of time to visualize the results of the optimization.

```
# Drone control optimization:

# Import necessary libraries
import numpy as np
from scipy.integrate import odeint
from scipy.optimize import minimize
```

```
import matplotlib.pyplot as plt

# Define the model of the drone
def drone_model(state, t, control_inputs):
    # Extract the state variables and control inputs
    x, y, z, vx, vy, vz = state
    thrust, angle = control_inputs

    # Compute the acceleration based on the thrust and angle
    ax = thrust * np.cos(angle)
    ay = thrust * np.sin(angle)
    az = -9.81  # gravity

    # The system of ODEs consists of the velocities and accelerations
    return [vx, vy, vz, ax, ay, az]

# Define the cost function for the optimization
def cost_function(control_inputs):
    # Define the initial state of the drone
    initial_state = [0, 0, 0, 0, 0, 0]

    # Define the time points for the simulation
    t = np.linspace(0, 5, 100)

    # Simulate the drone's motion
    result = odeint(drone_model, initial_state, t, args=(control_inputs,))

    # Compute the travel time (which is simply the total time) and energy consumption
    travel_time = t[-1]
    energy_consumption = np.trapz(np.abs(control_inputs), t)

    # The cost function is a weighted sum of the two objectives
    return 0.5 * travel_time + 0.5 * energy_consumption

# Use a numerical optimization algorithm to find the inputs that minimize the cost function
initial_inputs = [1, np.pi/4]  # initial guess for the thrust and angle
result = minimize(cost_function, initial_inputs, bounds=[(0, 2), (0, 2*np.pi)])

# Extract the optimized control inputs
optimized_inputs = result.x

# Initial conditions
x0, y0, z0 = 0, 0, 0  # initial position
v_x0, v_y0, v_z0 = 0, 0, 1.0  # initial velocity

initial_state = [x0, y0, z0, v_x0, v_y0, v_z0]  # [position, velocity]
```

```
# Simulate the drone's motion with the optimized control inputs
t = np.linspace(0, 10, 100)
optimized_result = odeint(drone_model, initial_state, t, args=(optimized_inputs,))

# Plot the drone's path
plt.figure()
plt.plot(t, optimized_result[:, 0], label='x')
plt.plot(t, optimized_result[:, 1], label='y')
plt.plot(t, optimized_result[:, 2], label='z')
plt.legend(loc='best')
plt.title('Drone\'s path with optimized control inputs')
plt.xlabel('Time (s)')
plt.ylabel('Position (m)')
plt.grid()
plt.show()

# Print the optimized control inputs
print(f"Optimized thrust: {optimized_inputs[0]}, Optimized angle: {optimized_inputs[1]}")
```

Workshop 1: Modeling the Motion of Celestial Bodies

Objective

The goal of this workshop is to understand the process of constructing a mathematical model for simulating the motion of celestial bodies. For simplicity, we will consider a two-body problem: the Sun and the Earth. We aim to calculate and visualize the orbit of Earth around the Sun.

Part 1: Mathematical Modelling and Manual Calculation

Background

Introduce Newton's law of universal gravitation, which is given by

$$F = \frac{G \cdot m1 \cdot m2}{r^2}$$

where F is the force between the masses, G is the gravitational constant, $m1$ and $m2$ are the two masses, and r is the distance between the centers of the two masses. This law governs the motion of celestial bodies in our universe.

Task 1: Constructing the Mathematical Model

The first task is to use Newton's law of universal gravitation and Newton's second law of motion to construct a mathematical model representing the motion of the Earth around the Sun. You need to create two differential equations: one for the x-direction and one for the y-direction. This task requires some knowledge of physics and calculus.

- **Newton's Law of Motion:** This law states that the force on an object is equal to its mass times its acceleration. Therefore, we can write the acceleration in the x and y directions as:

$$a_x = -G \cdot \frac{m_s}{r^2} \cdot \cos(\theta) \tag{14}$$

$$a_y = -G \cdot \frac{m_s}{r^2} \cdot \sin(\theta) \tag{15}$$

  where:

  - a_x and a_y are the accelerations in the x and y directions,
  - G is the gravitational constant,
  - m_s is the mass of the Sun,
  - r is the distance between the Earth and the Sun, and
  - θ is the angle between the line connecting the Earth and the Sun and the x-axis.

- **Second Order Differential Equations:** Using the acceleration expressions above, we can write the system of second order differential equations that governs the motion of the Earth as:

$$\frac{d^2x}{dt^2} = a_x = -G \cdot \frac{m_s}{(x^2+y^2)^{\frac{3}{2}}} \cdot x \tag{16}$$

$$\frac{d^2y}{dt^2} = a_y = -G \cdot \frac{m_s}{(x^2+y^2)^{\frac{3}{2}}} \cdot y \tag{17}$$

  Note: Here, $\cos(\theta)$ and $\sin(\theta)$ can be replaced with $\frac{x}{r}$ and $\frac{y}{r}$ respectively, since $r = \sqrt{x^2+y^2}$ and θ is the angle made by the position vector r with the x-axis. That's how we get x and y in place of $\cos(\theta)$ and $\sin(\theta)$ in the final equations.

Task 2: Defining the Initial Conditions

Next, define the initial conditions for the Earth's motion. This should include the initial position of the Earth relative to the Sun and the initial velocity of the Earth. You can use the average distance from the Earth to the Sun as the initial position and the average orbital speed of the Earth as the initial velocity.

- **Initial Conditions:** To solve the system of differential equations, we need to define the initial conditions for the Earth's motion. This includes the initial position and velocity of the Earth relative to the Sun. Let's use the average distance from the Earth to the Sun, $1\,AU = 1.496 \times 10^{11}\,m$, as the initial x-coordinate and set the initial y-coordinate to zero. The initial velocity in the x-direction is zero, and the initial velocity in the y-direction is the average orbital speed of the Earth, $v = 29.78 \times 10^3\,m/s$. Thus, our initial conditions are:
  - $x(0) = 1.496 \times 10^{11}\,m$
  - $y(0) = 0$
  - $v_x(0) = 0$
  - $v_y(0) = 29.78 \times 10^3\,m/s$

Task 3: Transformation to First Order Differential Equations

The differential equations obtained in Task 1 will be second order. However, it's easier to work with first order differential equations when we want to numerically solve them. So, your task is to transform the second order differential equations into a system of first order differential equations.

- **Transformation to First Order Differential Equations:** The differential equations obtained in Task 1 are second-order because they involve the second derivatives of the Earth's position (i.e., its acceleration). However, numerical methods for solving differential equations typically work

with first-order differential equations. Therefore, we need to transform our second-order equations into a system of first-order equations. This can be done by defining two new variables, $u = v_x$ and $v = v_y$, which represent the velocities in the x and y directions, respectively. Then we obtain the following first-order differential equations:

- $\frac{dx}{dt} = u$
- $\frac{dy}{dt} = v$
- $\frac{du}{dt} = -\frac{GM_s x}{(x^2+y^2)^{3/2}}$
- $\frac{dv}{dt} = -\frac{GM_s y}{(x^2+y^2)^{3/2}}$

where x and y are the positions in the x and y directions, and u and v are the velocities in the x and y directions, respectively.

Task 4: Manual Solution of the Differential Equations

While it's not possible to find an exact solution to these differential equations manually, you can find approximate solutions for short periods of time. Try to solve the first order differential equations manually for a small time step.

Part 2: Simulation and Visualization with Python

Task 1: Implementing the Differential Equations in Python

The first task in this part is to implement the system of first order differential equations in Python. You can define a function that takes the current state of the system and the current time as inputs and returns the derivatives of the state variables. Use the 'odeint' function from scipy's integrate module to numerically solve these differential equations.

Task 2: Running the Simulation

Now that you have implemented the differential equations, the next step is to run the simulation. You need to call the 'odeint' function with the correct arguments, including the initial conditions and the time array. For the time array, you can use numpy's 'linspace' function to create an array of evenly spaced time points.

Task 3: Visualizing the Results

After running the simulation, you should visualize the results. You can use matplotlib's 'plot' function to create a 2D plot of the Earth's trajectory. On the x-axis, plot the x-coordinates of the Earth, and on the y-axis, plot the y-coordinates.

Task 4: Performing a Sensitivity Analysis

The last task is to perform a sensitivity analysis. You should change the initial conditions or the parameters in the differential equations and see how these changes affect the results. You can create multiple plots to compare the different results.

Task 5: Discussing the Results

Discuss the results from the simulation and the visualization. What does the shape of the Earth's trajectory tell you about the nature of its motion? How do changes in the initial conditions or parameters affect the trajectory? What are the limitations of this model?

Note

You are expected to have a basic understanding of classical mechanics, differential equations, and Python programming before attending this workshop.