24BSE2113D- Partial Differential Equation, Transform and Optimization

Computational Methods for Electronics & Robotics Lab

Siju K S

2025-07-16

Table of contents

1 Welcome to the Computational Methods Lab

This interactive lab manual is designed for third-semester B.Tech students in Electronics and Robotics. The goal of this course is to introduce powerful computational techniques for solving common engineering problems using the Python programming language.

1.1 Do you really need to be a successful Robotics Engineer?

https://youtu.be/KsirntbBt6I?si=aZmd3XI02K_bdOB8

1.2 How do you approach learning coding in 2025?

https://youtu.be/6Lcy2N3YcIs?si=cJ-dzjhydrezE2Yi

1.3 Why Python?

While MATLAB is a traditional tool in engineering, Python has become the industry standard in modern robotics, machine learning, and data science. By learning Python, you are acquiring a versatile skill that is highly sought after in today's tech landscape. It's free, open-source, and has a vast ecosystem of libraries that we will leverage throughout this course.

https://youtu.be/uDwyzIUjTFU?si=IN3rPeNjIFu40R-J

1.4 Software Setup

Before you begin, please ensure you have the following installed on your system. We highly recommend installing the **Anaconda Distribution**, which packages Python and all the necessary libraries together.

- **Python 3.8+**

- **NumPy:** For numerical operations.
- **Matplotlib:** For plotting and visualization.
- **SymPy:** For symbolic mathematics (algebra and calculus).
- **SciPy:** For scientific computing, including optimization.

You can run the code in this book interactively by using a **Jupyter Notebook** or an IDE like **Spyder** or **VS Code** with Python extensions.

Part I

Fundamentals

2 Lab Session 1: Python's Scientific Stack

Aim

This session introduces the fundamental libraries that make `Python` a powerhouse for engineering.

2.1 Excercise 1: The Core Trio - `NumPy`, `Matplotlib`, and `SymPy`

Objective: To get comfortable creating arrays, plotting data, and performing symbolic calculations.

2.1.1 Python basics- Recap

> **Starting `Python` coding with `colab`**
>
> You can complete your experiments using `colab`- A cloud jupyter notebook. Please use the following link to run `Python` code in colab. https://colab.google/ (Right click and open in a new tab)

2.1.1.1 Hello World

The classic first program for any language.

```
print("Hello, B.Tech Students!")
```

```
Hello, B.Tech Students!
```

2.1.1.2 Variables and Data Types

Introduction to different data types like integers, floats, and strings.

```
x = 10  # Integer
y = 3.5  # Float
name = "Python"  # String
is_student = True  # Boolean

print(x, y, name, is_student)
```

```
10 3.5 Python True
```

2.1.1.3 Conditional Statements

Using if, elif, and else statements.

```
x = 10  # Integer
if x>5:
    print("x is greater than 5")
elif x==5:
    print("x is 5")
else:
    print("x is less than 5")
```

```
x is greater than 5
```

2.1.1.4 Loops/ Iteratives

Using `for` and `while` loops.

```
# For loop
for i in range(5):
    print("Iteration:", i)
```

```
Iteration: 0
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
```

```
# While loop
n = 0
while n < 3:
    print("While loop iteration:", n)
    n += 1
```

```
While loop iteration: 0
While loop iteration: 1
While loop iteration: 2
```

2.1.1.5 Functions

Defining and calling functions.

```
def add_numbers(a, b):
    return a + b

result = add_numbers(5, 3)
print("Sum:", result)
```

```
Sum: 8
```

2.1.1.6 Basic Numerical Computations using `NumPy`

`NumPy` is useful for numerical operations.

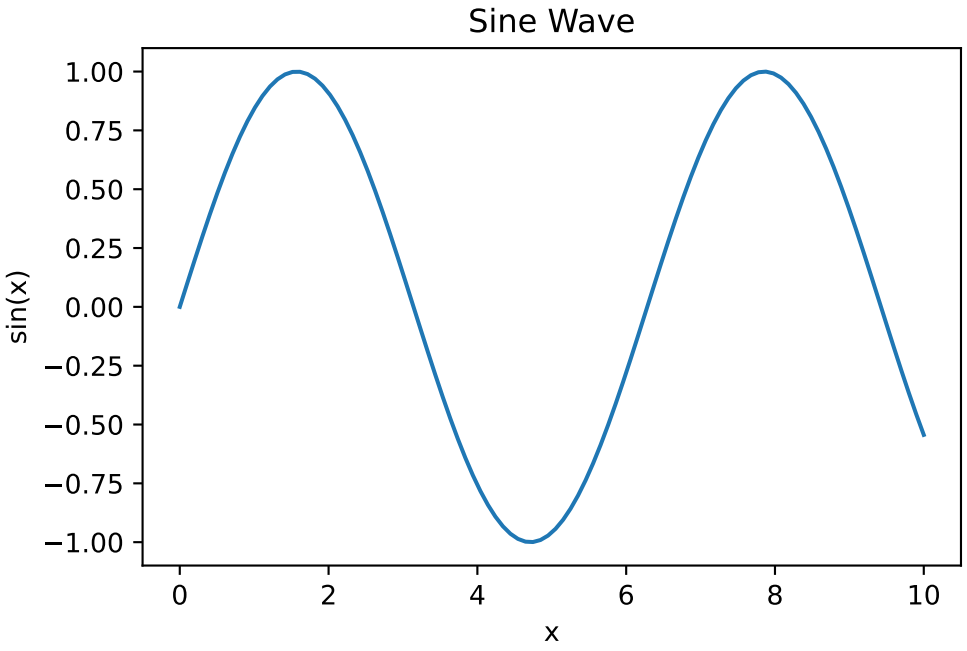
```
#Solve 2x+3y=54x+4y=6
import numpy as np

A = np.array([[2, 3], [4, 4]])
b = np.array([5, 6])

x = np.linalg.solve(A, b)
print("Solution:", x)
```

```
Solution: [-0.5  2. ]
```

2.1.1.7 Plotting Graphs with Matplotlib

Matplotlib is used for simple visualizations. Install using: `pip install matplotlib`

```
#Plotting a sine wave
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y)
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.title("Sine Wave")
plt.show()
```




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

# Parameters
Vs = 5.0  # Source voltage (Volts)
```

```
R = 1000  # Resistance (Ohms)
C = 1e-6  # Capacitance (Farads)
tau = R * C # Time constant

# Create a time vector from 0 to 5*tau
t = np.linspace(0, 5 * tau, 100)

# Calculate voltage using the formula
Vc = Vs * (1 - np.exp(-t / tau))

# Plotting the result
plt.figure(figsize=(8, 5))
plt.plot(t, Vc, label=f'RC = {tau}s')
plt.title('Capacitor Charging Voltage')
plt.xlabel('Time (s)')
plt.ylabel('Voltage (V)')
plt.grid(True)
plt.legend()
plt.show()
```

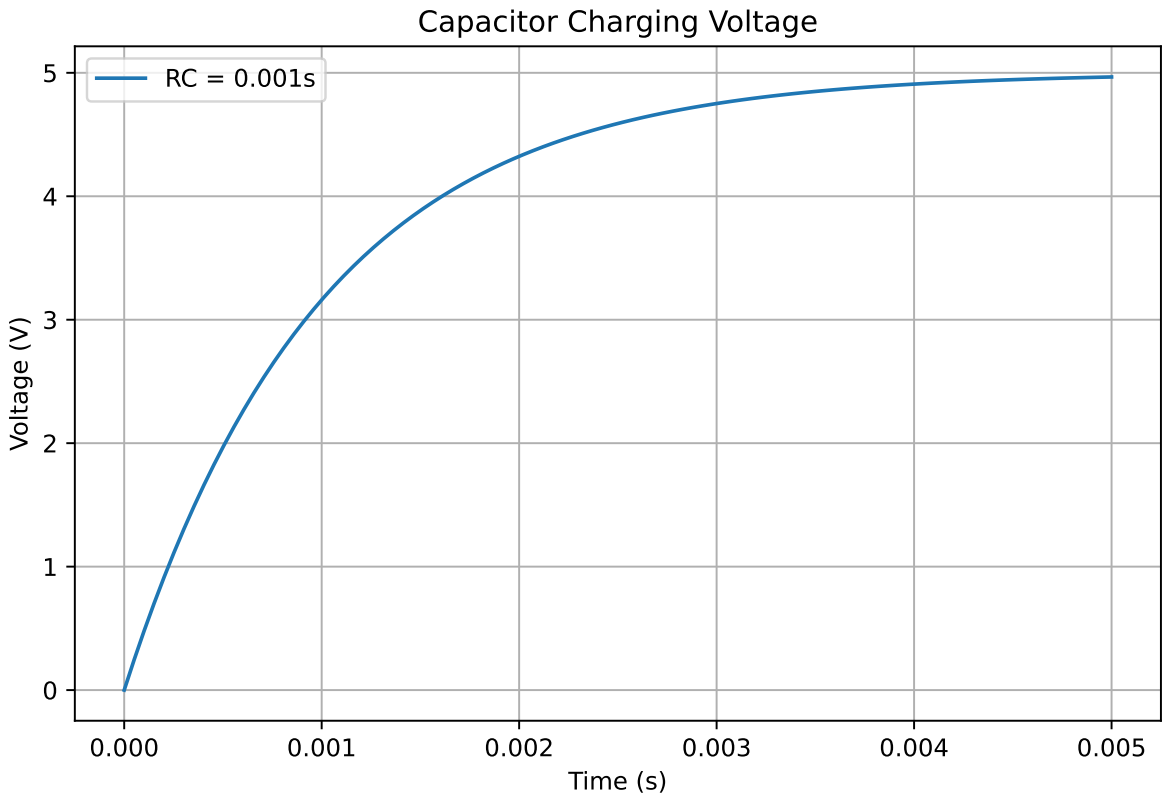


Figure 2.1: Voltage across a charging capacitor in an RC circuit.

2.1.1.8 Solution of PDE Using SciPy

`SciPy` provides numerical solvers for differential equations and optimizations.

> **Installing `Scipy`**
>
> The `Scipy` library can be installed using the `pip install scipy` command in terminal or using `!pip install scipy` in colab.

```
#Solving a Simple PDE  u/ x +  u/ t=0
from sympy import symbols, Function, Eq, Derivative, pdsolve

# Define variables
x, t = symbols('x t')
u = Function('u')(x, t)
```

```
# Define a simple PDE:  u/ x +  u/ t = 0
pde = Eq(Derivative(u, x) + Derivative(u, t), 0)

# Solve the PDE using pdsolve
solution = pdsolve(pde)
# Print the solution
print(solution)
```

```
Eq(u(x, t), F(-t + x))
```

```
# Example: Solving an ODE as an approximation for a PDE
from scipy.integrate import solve_ivp
import numpy as np
from scipy.integrate import solve_ivp
import numpy as np
import matplotlib.pyplot as plt

def pde_rhs(t, u):
    return -0.5 * u  # Example equation

sol = solve_ivp(pde_rhs, [0, 10], [1], t_eval=np.linspace(0, 10, 100))
plt.plot(sol.t, sol.y[0])
plt.xlabel('Time')
plt.ylabel('Solution')
plt.show()
```

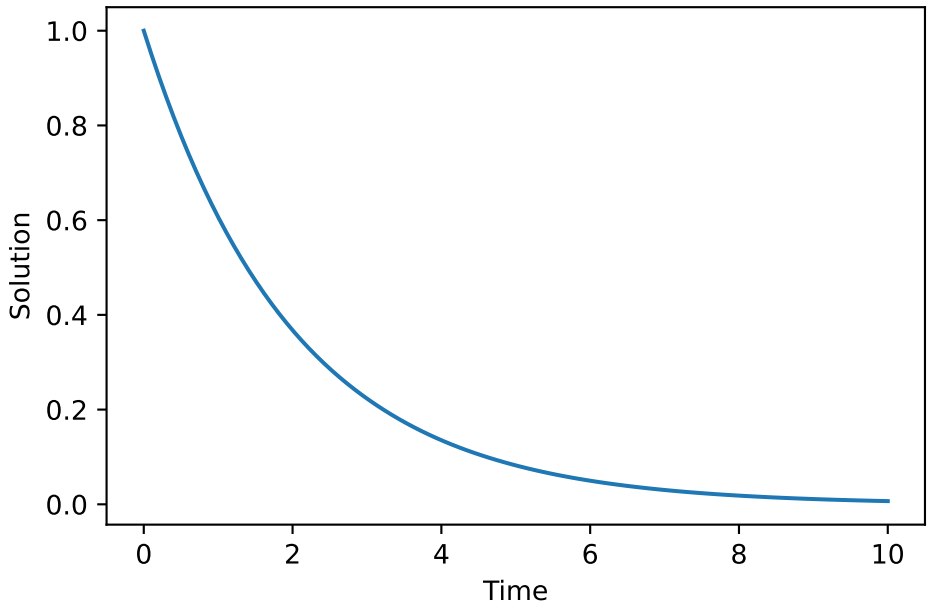


Figure 2.2: Solution of PDE.

2.1.1.9 Basic Optimization Using `SymPy`

`SymPy` is a symbolic mathematics library that can be used to derive analytical solutions to PDEs.

```
from scipy.optimize import minimize

def objective(x):
    return x**2 + 2*x + 1

result = minimize(objective, 0)  # Start search at x=0
print("Optimal x:", result.x)
```

```
Optimal x: [-1.00000001]
```

2.2 Practice tasks

2.2.1 T-1: Representing an Analog Signal

Concept: In electronics, continuous analog signals (like AC voltage) are sampled at discrete time intervals to be processed by a digital system (like a microcontroller or computer). A `NumPy` array is the perfect way to store these sampled values.

Python Skills: * `np.linspace()`: To create an array of evenly spaced time points. * `np.sin()`: An element-wise function that applies the sine function to every value in an array.

Task: Generate and plot a 50 Hz sine wave voltage signal with a peak voltage of 5V, sampled for 3 cycles.

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

# --- Parameters ---
frequency = 50  # Hz
peak_voltage = 5.0  # Volts
cycles = 3
sampling_rate = 1000 # Samples per second

# --- Time Array Generation ---
# Duration of 3 cycles is 3 * (1/frequency)
duration = cycles / frequency
# Create 1000 points per second * duration
num_samples = int(sampling_rate * duration)
t = np.linspace(0, duration, num_samples)

# --- Signal Generation ---
# The formula for a sine wave is V(t) = V_peak * sin(2 * pi * f * t)
voltage = peak_voltage * np.sin(2 * np.pi * frequency * t)

# --- Visualization ---
plt.figure(figsize=(10, 4))
plt.plot(t, voltage)
plt.title('Digital Representation of a 50 Hz Sine Wave')
plt.xlabel('Time (s)')
plt.ylabel('Voltage (V)')
plt.grid(True)
plt.show()
```

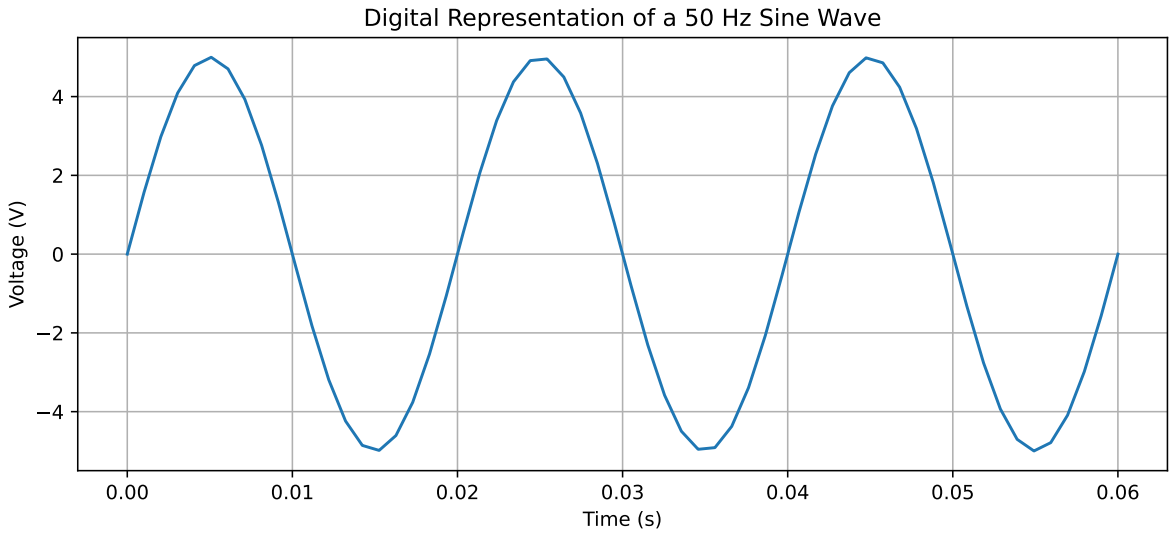


Figure 2.3: A 5V, 50 Hz sine wave sampled over time.

2.2.2 T-2: Modeling a Noisy Sensor Reading

Concept: Real-world sensor data is never perfect. It's often corrupted by random noise. We can simulate this by adding a random component to our ideal signal.

Python Skills: - Array Addition: Simply using + to add two arrays of the same shape.

- `np.random.normal()`: To generate Gaussian noise, which is a common model for electronic noise.

Task: Take the 5V sine wave from the previous example and add Gaussian noise with a standard deviation of 0.5V to simulate a noisy sensor reading.

```
# We can reuse the 't' and 'voltage' arrays from the previous example
noise_amplitude = 0.5 # Standard deviation of the noise in Volts

# Generate noise with the same shape as our voltage array
noise = np.random.normal(0, noise_amplitude, voltage.shape)

# Create the noisy signal by adding the noise to the ideal signal
noisy_voltage = voltage + noise

# --- Visualization ---
plt.figure(figsize=(10, 4))
plt.plot(t, voltage, label='Ideal Signal', linestyle='--')
```

```
plt.plot(t, noisy_voltage, label='Noisy Sensor Reading', alpha=0.75)
plt.title('Ideal vs. Noisy Signal')
plt.xlabel('Time (s)')
plt.ylabel('Voltage (V)')
plt.legend()
plt.grid(True)
plt.show()
```

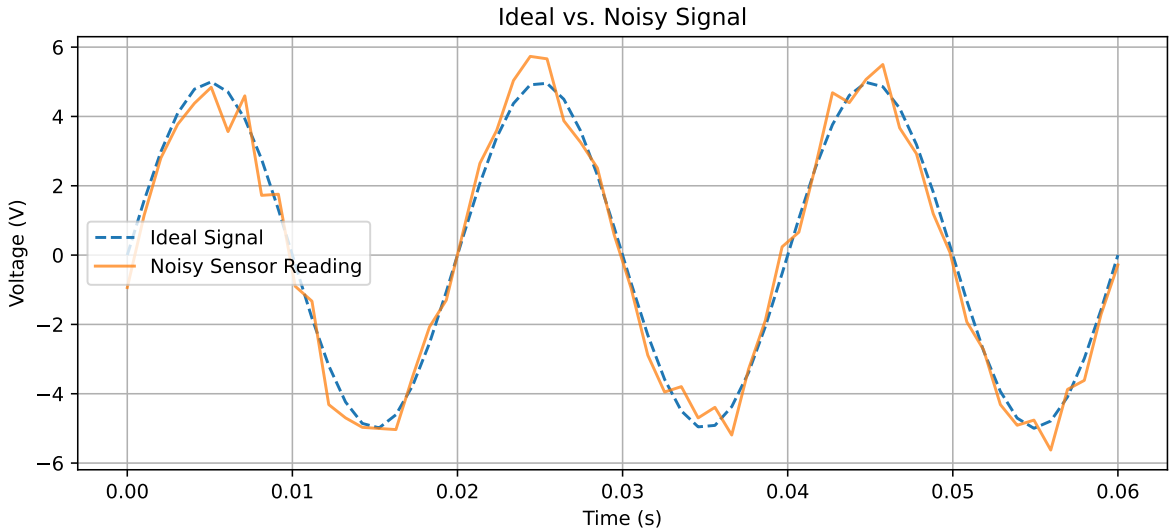


Figure 2.4: Ideal sine wave vs. a simulated noisy sensor reading.

2.2.3 T-3: Forward Kinematics of a 2-Link Robot Arm

Concept: Forward kinematics in robotics is the process of calculating the position of the robot's end-effector (e.g., its gripper) based on its joint angles. For a simple 2D arm, this involves basic trigonometry.

Python Skills:

- Using arrays to represent vectors (link lengths).
- Using scalar variables for parameters (joint angles).
- Basic arithmetic and trigonometric functions (`np.cos`, `np.sin`, `np.deg2rad`).

Task: Calculate and plot the position of a 2-link planar robot arm with link lengths L1=1.0m and L2=0.7m for given joint angles theta1=30° and theta2=45°.

```
# --- Parameters ---
L1 = 1.0  # Length of link 1
L2 = 0.7  # Length of link 2
theta1_deg = 30
theta2_deg = 45

# Convert angles to radians for numpy's trig functions
theta1 = np.deg2rad(theta1_deg)
theta2 = np.deg2rad(theta2_deg)

# --- Kinematics Calculations ---
# Position of the first joint (end of L1)
x1 = L1 * np.cos(theta1)
y1 = L1 * np.sin(theta1)

# Position of the end-effector (end of L2) relative to the first joint
# The angle of the second link is theta1 + theta2
x2 = x1 + L2 * np.cos(theta1 + theta2)
y2 = y1 + L2 * np.sin(theta1 + theta2)

# --- Visualization ---
plt.figure(figsize=(6, 6))
# Plot the arm links
plt.plot([0, x1], [0, y1], 'r-o', linewidth=3, markersize=10, label='Link 1')
plt.plot([x1, x2], [y1, y2], 'b-o', linewidth=3, markersize=10, label='Link 2')

# Plot the base and end-effector positions for clarity
plt.plot(0, 0, 'ko', markersize=15, label='Base')
plt.plot(x2, y2, 'gX', markersize=15, label='End-Effector')

plt.title('2-Link Robot Arm Kinematics')
plt.xlabel('X Position (m)')
plt.ylabel('Y Position (m)')
plt.grid(True)
plt.axis('equal') # Important for correct aspect ratio
plt.legend()
plt.show()

print(f"End-effector is at position: ({x2:.2f}, {y2:.2f})")
```

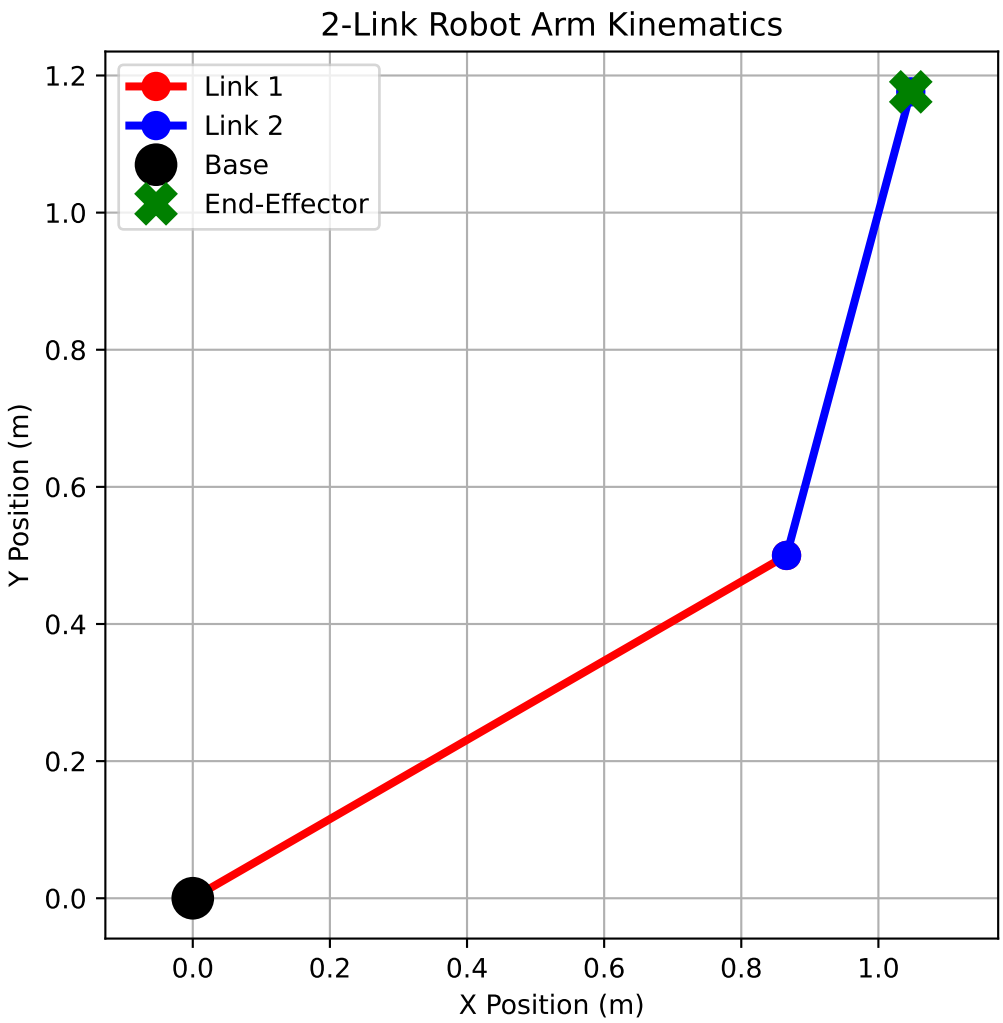


Figure 2.5: Position of a 2-link robot arm using forward kinematics.

```
End-effector is at position: (1.05, 1.18)
```

2.2.4 T-4: Rotating a Sensor's Coordinate Frame

Concept: A robot often has sensors (like a camera or a Lidar) mounted at an angle. To understand the sensor data in the robot's own coordinate frame, we need to rotate the data points. This is a fundamental operation in robotics and computer vision, done using a rotation matrix.

Python Skills:

- Creating a 2D NumPy array (a matrix).
- Matrix multiplication using the `@`operator.
- Transposing an array (`.T`) for correct multiplication dimensions.

Task: A sensor detects an object at coordinates (2, 0) in its own frame. The sensor is rotated 45 degrees counter-clockwise relative to the robot's base. Find the object's coordinates in the robot's frame.

```
# Angle of the sensor relative to the robot
angle_deg = 45
angle_rad = np.deg2rad(angle_deg)

# Point detected in the sensor's frame [x, y]
p_sensor = np.array([[2], [0]]) # As a column vector

# 2D Rotation Matrix
# R = [[cos(theta), -sin(theta)],
#      [sin(theta),  cos(theta)]]
R = np.array([[np.cos(angle_rad), -np.sin(angle_rad)],
              [np.sin(angle_rad),  np.cos(angle_rad)]])

# The transformation: p_robot = R @ p_sensor
p_robot = R @ p_sensor

# --- Visualization ---
plt.figure(figsize=(6, 6))
# Plot sensor's axes
plt.quiver(0, 0, np.cos(angle_rad), np.sin(angle_rad), color='r', scale=3, label="Sensor x'-a
plt.quiver(0, 0, -np.sin(angle_rad), np.cos(angle_rad), color='g', scale=3, label="Sensor y'-

# Plot the point in the robot's frame
plt.plot(p_robot[0], p_robot[1], 'bo', markersize=10, label='Point in Robot Frame')
# For context, let's show where the point was in the sensor's frame (if it weren't rotated)
# This is just for visualization
plt.plot(p_sensor[0], p_sensor[1], 'ko', markersize=10, alpha=0.5, label='Original point (rel


plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid(True)
plt.axis('equal')
plt.xlim(-1, 3)
```

```
plt.ylim(-1, 3)
plt.title("Coordinate Frame Rotation")
plt.xlabel("Robot X-axis")
plt.ylabel("Robot Y-axis")
plt.legend()
plt.show()

print("Rotation Matrix:\n", np.round(R, 2))
print(f"\nPoint in Sensor Frame: {p_sensor.flatten()}")
print(f"Point in Robot Frame:  {np.round(p_robot.flatten(), 2)}")
```

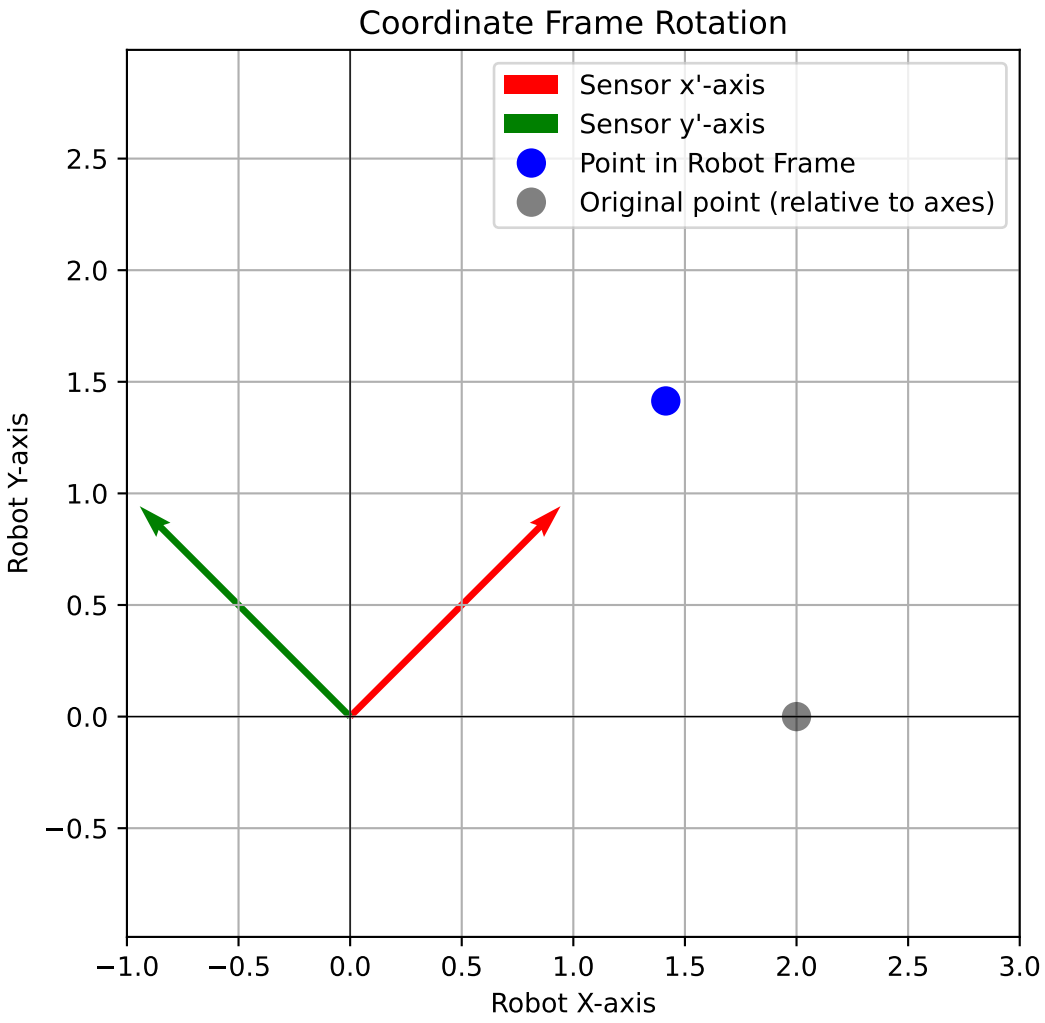


Figure 2.6: Rotating a point from a sensor's frame to the robot's frame.

```
Rotation Matrix:
 [[ 0.71 -0.71]
 [ 0.71  0.71]]

Point in Sensor Frame: [2 0]
Point in Robot Frame:  [1.41 1.41]
```

2.2.5 T-5: Applying a Simple Digital Filter

Concept: To clean up the noisy signal from Example 3, we can apply a digital filter. The simplest is a moving average filter, which replaces each data point with the average of itself and its neighbors. This smooths out sharp fluctuations (noise).

Python Skills:

- Looping over an array.
- Array slicing: array[start:end].
- `np.mean()`: To calculate the average of a set of numbers.

Task: Apply a 5-point moving average filter to the noisy_voltage signal created in Example 3 and plot the result to see the smoothing effect.

```python
# Let's regenerate the noisy signal for a self-contained example
# (In a real notebook, you would reuse the variable from before)
frequency = 50
peak_voltage = 5.0
duration = 3 / frequency
t = np.linspace(0, duration, int(1000 * duration))
voltage = peak_voltage * np.sin(2 * np.pi * frequency * t)
noise = np.random.normal(0, 0.5, voltage.shape)
noisy_voltage = voltage + noise

# --- Filtering ---
window_size = 5
# Create an empty array to store the filtered signal
filtered_voltage = np.zeros_like(noisy_voltage)

# Loop through the signal. We can't compute a full window at the very edges,
# so we'll just copy the original values for the first and last few points.
for i in range(len(noisy_voltage)):
    # Find the start and end of the slice
    start = max(0, i - window_size // 2)
    end = min(len(noisy_voltage), i + window_size // 2 + 1)
```

```
    # Get the window of data and calculate its mean
    window = noisy_voltage[start:end]
    filtered_voltage[i] = np.mean(window)

# --- Visualization ---
plt.figure(figsize=(10, 4))
plt.plot(t, noisy_voltage, label='Noisy Signal', alpha=0.5)
plt.plot(t, filtered_voltage, label='Filtered Signal', color='r', linewidth=2)
plt.title('Effect of Moving Average Filter')
plt.xlabel('Time (s)')
plt.ylabel('Voltage (V)')
plt.legend()
plt.grid(True)
plt.show()
```

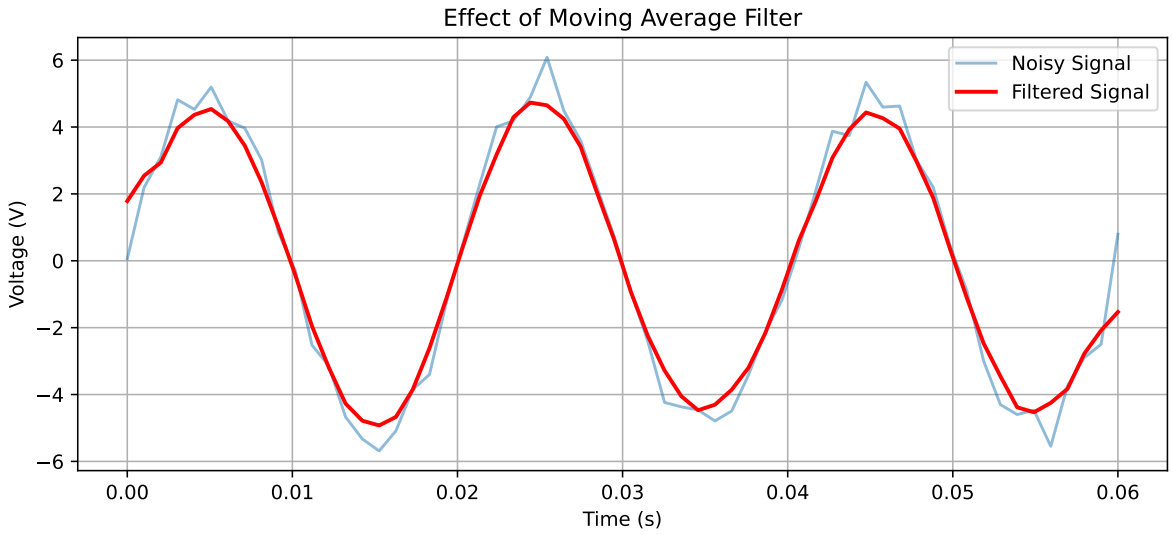


Figure 2.7: Noisy signal smoothed using a 5-point moving average filter.

Part II

Partial Differential Equations

3 Lab Session 2: Solving First-Order Linear PDEs Using Finite Difference Method (FDM)

3.1 Theoretical Background

https://youtu.be/xoNdZMIiTbc?si=tfu1joIpKh8UX2th

3.2 Experiment 2: 1D Wave Equation (The Advection Equation)

While Experiment 1 dealt with basic `python` programming and solution of a *parabolic* PDE (diffusion), this experiment focuses on a *hyperbolic* PDE, which describes phenomena that propagate, like waves. We will solve the **1D Linear Advection Equation**, a fundamental model for wave motion.

Governing Equation:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0$$

Here, `u(x, t)` is the quantity that is moving (e.g., the height of a water wave or the stress in a rod), `x` is position, `t` is time, and `c` is the constant speed at which the wave travels.

3.2.1 Aim

To solve the one-dimensional wave equation using the **Upwind Method** and compare the numerical result with the initial condition.

3.2.2 Objectives

- To learn how to solve partial differential equations using numerical methods.
- To understand and apply the Finite Difference Method for hyperbolic equations.
- To observe how a wave profile moves with time.
- To write and run a Python program for the simulation.

3.2.3 The Upwind Method Algorithm

The "upwind" name comes from the fact that for a wave moving to the right (c > 0), we use information from the "upwind" direction (the left side, `i-1`) to determine the state at the next time step.

1. **Discretize Domain:** Set the number of space points (`nx`) and time steps (`nt`). Define the domain length (`Lx`) and total time (`T`). Calculate the step sizes $dx = \frac{Lx}{(nx-1)}$ and $dt = \frac{T}{nt}$.
2. **Check Stability (CFL Condition):** Compute the Courant-Friedrichs-Lewy (CFL) number, $\lambda = c\frac{\Delta t}{\Delta x}$. For the explicit upwind scheme to be stable, we must have $\lambda \leq 1$. This means the numerical wave speed (`dx/dt`) must be faster than the physical wave speed (`c`), so the calculation can "keep up" with the phenomenon.
3. **Set Initial Condition:** Define the initial shape of the wave, `u(x, 0)`.
4. **Time Marching Loop:** For each time step `n`, iterate through all spatial points `i` and apply the upwind formula to find the wave's new state `u_new`. The discretized formula is:
$$u_i^{n+1} = u_i^n - \lambda(u_i^n - u_{i-1}^n)$$
5. **Update:** Set `u = u_new`.
6. **Visualize:** Plot the final wave profile against the initial one.

3.2.4 Case Study: Exponential Decay Wave

Let's simulate the equation $\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0$ (so `c=1`) with the initial condition $u(x, 0) = e^{-x}$.

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

# --- 1. Discretize Domain ---
nx = 50       # Number of spatial points
nt = 30       # Number of time steps
Lx = 5.0      # Domain length
T = 2.0       # Total time
dx = Lx / (nx - 1)  # Spatial step
dt = T / nt         # Time step

c = 1.0       # Wave speed

# --- 2. Check Stability ---
lambda_ = c * dt / dx
```

```
print(f"CFL Number (lambda) = {lambda_:.4f}")
if lambda_ > 1:
    print("Warning: CFL condition not met. The solution may be unstable!")

# --- 3. Set Initial Condition ---
x = np.linspace(0, Lx, nx)
u = np.exp(-x)  # u(x, 0) = e^(-x)
u_initial = u.copy() # Save the initial state for plotting

# --- 4. & 5. Time Marching Loop ---
for n in range(nt):
    u_old = u.copy()
    # Apply upwind formula to all interior points
    for i in range(1, nx):
        u[i] = u_old[i] - lambda_ * (u_old[i] - u_old[i-1])

# --- 6. Visualize ---
plt.figure(figsize=(10, 6))
plt.plot(x, u_initial, 'r--', label="Initial Condition at t=0")
plt.plot(x, u, 'b-', marker='o', markersize=4, label=f"Numerical Solution at t={T}")
plt.xlabel("Position (x)")
plt.ylabel("u(x, t)")
plt.legend()
plt.title("Solution of 1D Advection Equation (Upwind Scheme)")
plt.grid(True)
plt.show()
```

```
CFL Number (lambda) = 0.6533
```

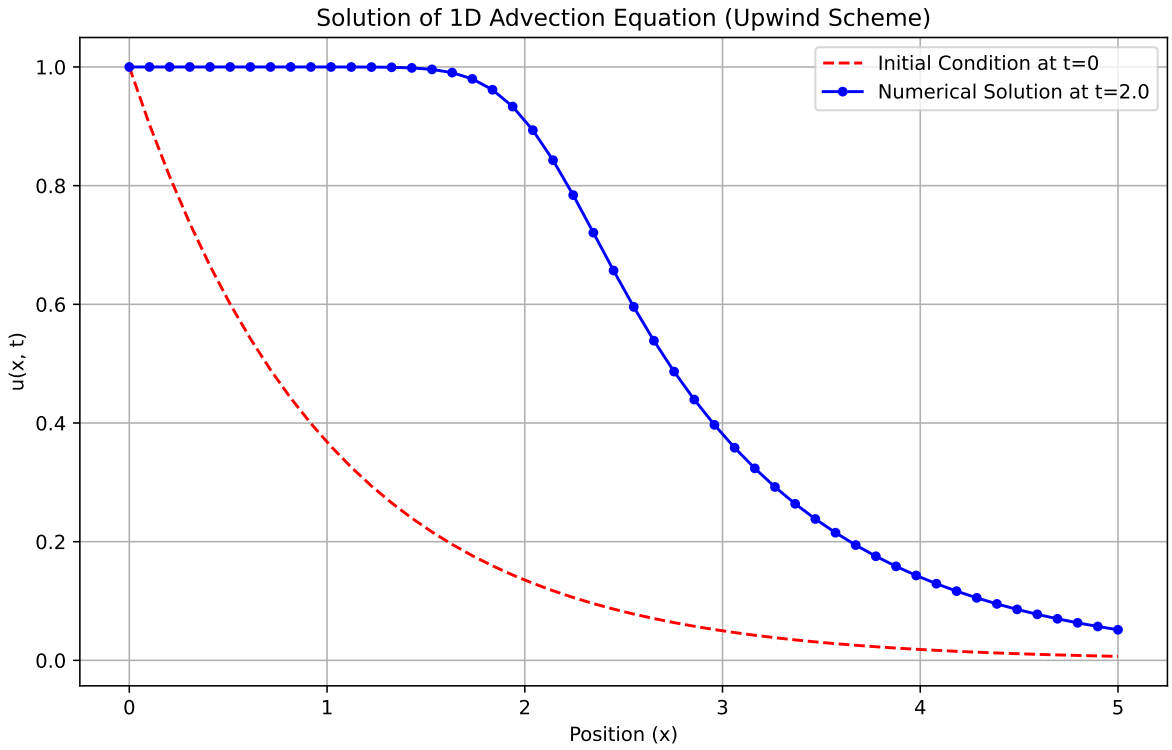


Figure 3.1: Numerical solution of the advection equation with an exponential initial condition using the Upwind scheme.

3.2.5 Result and Discussion

The numerical solution using the Upwind Method for the initial condition $u(x, 0) = e^{-x}$ shows the wave propagating to the right, as expected. The Courant number $\lambda = c\dfrac{dt}{dx}$ was maintained within the stability range ($\lambda \leq 1$), ensuring a stable and oscillation-free result. However, we can observe two key phenomena characteristic of this first-order scheme:

- Numerical Diffusion: The final wave shape is "smeared" or smoothed out compared to the initial condition. The sharp features are dampened. This is artificial damping introduced by the numerical method itself.
- Amplitude Decay: The peak of the wave seems to decrease slightly.

Despite these inaccuracies, the upwind scheme is computationally simple and robust, making it effective for simulating basic convection phenomena where the general transport behavior is more important than preserving the exact shape of the wave.

3.3 Application Problem: Stress Wave in a Rod

In mechanical systems, like long slender rods or beams, stress waves travel due to sudden loads or impacts. When a rod is struck at one end, a wave of stress and strain travels along its length. This can be modeled using the same 1D advection equation.

Task: Modify the code above to simulate a "square pulse" stress wave.

Governing Equation: $\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial t} = 0$.

Initial Condition: A square pulse representing a localized impact.

$$u(x,0) = \begin{cases} 1 & ; 0.4 \leq x \leq 0.6 \\ 0 & ; \text{otherwise} \end{cases}$$

Your Challenge:

- Copy the Python code block from the previous example.
- Change the line that sets the initial condition u to create the square pulse described above. Hint: You can use NumPy's logical indexing. For example: `u[ (x >= 0.4) & (x <= 0.6) ] = 1.0`. You will need to initialize u as an array of zeros first: `u = np.zeros(nx)`.
- Re-run the simulation. Observe how the sharp corners of the square wave are smoothed out due to numerical diffusion.

3.3.0.1 Solution to the Application Problem: Stress Wave in a Rod

Here, we apply the same Upwind Method to the practical problem of a stress wave propagating through a mechanical rod. The initial condition is a square pulse, which could represent a short, sharp impact from a hammer strike on a specific section of the rod.

Python Implementation

The core algorithm remains the same. The only change is in how we define the initial condition `u(x, 0)`.

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

# --- 1. Discretize Domain (same as before) ---
nx = 101      # Increased points for better resolution
nt = 50       # Increased time steps
Lx = 2.0      # Longer domain to see the wave travel
```

```
T = 1.0       # Total time
dx = Lx / (nx - 1)
dt = T / nt
c = 1.0       # Wave speed

# --- 2. Check Stability ---
lambda_ = c * dt / dx
print(f"CFL Number (lambda) = {lambda_:.4f}")
if lambda_ > 1:
    print("Warning: CFL condition not met. The solution may be unstable!")

# --- 3. Set Initial Condition: Square Pulse ---
x = np.linspace(0, Lx, nx)
# Initialize u as an array of zeros
u = np.zeros(nx)
# Set the pulse region to 1.0 using logical indexing
u[(x >= 0.4) & (x <= 0.6)] = 1.0

# Save the initial state for plotting
u_initial = u.copy()

# --- 4. & 5. Time Marching Loop (same as before) ---
for n in range(nt):
    u_old = u.copy()
    for i in range(1, nx):
        u[i] = u_old[i] - lambda_ * (u_old[i] - u_old[i-1])

# --- 6. Visualize ---
# Calculate the analytical solution's position for comparison
# The pulse should have moved by a distance of c*T
analytical_x_start = 0.4 + c * T
analytical_x_end = 0.6 + c * T

plt.figure(figsize=(10, 6))
plt.plot(x, u_initial, 'r--', label="Initial Condition (t=0)")
plt.plot(x, u, 'b-', marker='.', markersize=5, label=f"Numerical Solution (t={T})")

# Plot the theoretical "perfect" wave for comparison
plt.plot([analytical_x_start, analytical_x_end], [1, 1], 'g:', linewidth=3, label=f'Analytica
plt.plot([analytical_x_start, analytical_x_start], [0, 1], 'g:', linewidth=3)
plt.plot([analytical_x_end, analytical_x_end], [0, 1], 'g:', linewidth=3)
```

```
plt.xlabel("Position along rod (x)")
plt.ylabel("Stress/Strain (u)")
plt.ylim(-0.2, 1.2) # Set y-axis limits for better visualization
plt.legend()
plt.title("Simulation of a Square Stress Wave")
plt.grid(True)
plt.show()
```

```
CFL Number (lambda) = 1.0000
```

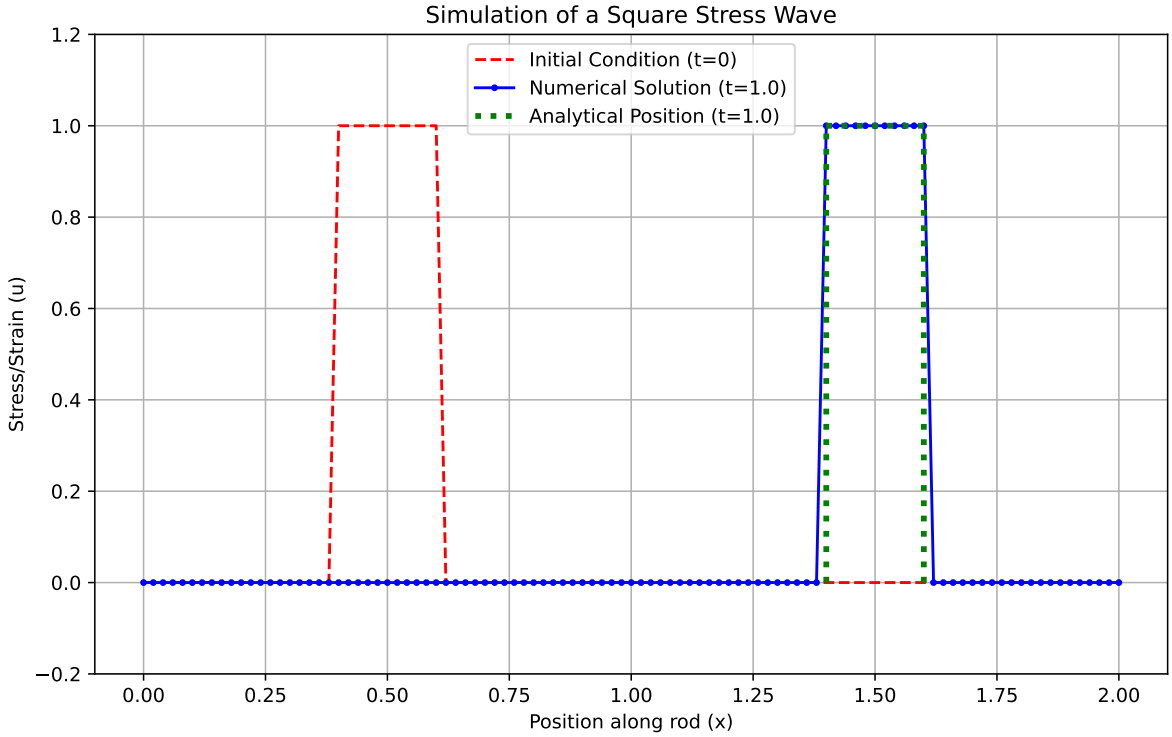


Figure 3.2: Propagation of a square pulse stress wave, showing significant numerical diffusion.

3.3.0.2 Discussion of the Square Pulse Result

Wave Propagation: The primary success of the simulation is clearly visible: the pulse has moved to the right. The leading edge of the numerical solution is centered around the correct analytical position (x = 1.4 for the start of the pulse), confirming that the model correctly captures the fundamental advection behavior at speed c=1 over time T=1.

Numerical Diffusion: The most striking feature of the result is the severe smoothing of the square pulse. The sharp, vertical edges of the initial condition have been transformed into gentle slopes. The flat top of the pulse has become rounded and its peak amplitude has decreased from 1.0 to approximately 0.75. This is a classic and pronounced example of numerical diffusion, an error inherent in the first-order upwind scheme. The scheme struggles to resolve sharp gradients (discontinuities), so it effectively "smears" them out over several grid points.

Practical Implications: In a real engineering scenario, if we needed to know the exact peak stress and its precise location, this simple method would be inadequate. The predicted peak stress is significantly lower than the actual initial stress, which could lead to an incorrect safety assessment.

Improving the Model: While the upwind scheme is simple, its diffusive nature is a major drawback for problems requiring high accuracy. To improve the result and preserve the sharpness of the wave, engineers would use:

- Higher-Order Schemes: Methods like the Lax-Wendroff scheme or flux-limiter methods are designed to be less diffusive and capture sharp fronts more accurately.
- Finer Grid: Reducing dx and dt (while keeping lambda stable) can decrease the amount of diffusion, but at a significant computational cost.

3.3.0.3 Result

The application simulation successfully models the physical propagation of a stress wave. However, it also serves as a crucial lesson in computational engineering: every numerical method has inherent errors and limitations. The Upwind Method, while stable and simple, introduces significant numerical diffusion that must be considered when interpreting the results.

3.4 ID Heat Equation- Theoretical Background

https://youtu.be/i8rnEl8O-r0?si=cB82MUHLk7RYTSzi

3.5 Experiment 3: Solving the 1D Heat Equation

This experiment introduces the numerical solution of a *parabolic* partial differential equation: the one-dimensional heat equation. This equation is fundamental to understanding diffusion processes, not just in heat transfer but also in chemical concentration gradients and other physical phenomena.

3.5.1 Aim

To solve the one-dimensional heat conduction equation using the *Finite Difference Method (explicit scheme)* in `Python` and to visualize the temperature distribution over time.

3.5.2 Objectives

- To understand the physical interpretation of the one-dimensional heat equation.
- To discretize the heat equation using the *Forward Time Centred Space (FTCS)* scheme.
- To implement the finite difference method in `Python`.
- To analyze the stability condition required for this numerical method.
- To visualize the temperature profiles at different time levels.

3.5.3 Governing Equation and Discretization

The one-dimensional heat equation describes how temperature `u` evolves at a position `x` and time `t`. It is given by:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

where α is the thermal diffusivity of the material, a constant that indicates how quickly the material conducts heat.

To solve this numerically, we use the *FTCS* method. We approximate the derivatives with finite differences: * *Time Derivative (Forward Difference):* $\frac{\partial u}{\partial t} \approx \frac{u_i^{n+1} - u_i^n}{\Delta t}$ * *Space Derivative (Centred Difference):* $\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{(\Delta x)^2}$

Here, the superscript `n` denotes the time level and the subscript `i` denotes the spatial grid point. Substituting these into the governing equation and rearranging for the future temperature u_i^{n+1}, we get the explicit update formula.

3.5.4 Algorithm

1. **Initialize Parameters:** Set the values for the rod length `L`, total simulation time `T`, number of spatial nodes n_x, number of time steps n_t, and thermal diffusivity α.
2. **Discretize the Domain:** Calculate the step sizes for space and time.
    - Space step: $\Delta x = \frac{L}{n_x - 1}$
    - Time step: $\Delta t = \frac{T}{n_t}$

3. **Check Stability Condition:** Calculate the stability parameter, `r`. For the FTCS scheme to be stable and produce a physically realistic result, this value **must be less than or equal to 0.5**.

$$r = \frac{\alpha \Delta t}{(\Delta x)^2} \leq 0.5$$

4. **Set Initial and Boundary Conditions:**
   - Define the initial temperature distribution along the rod, $u(x, 0)$.
   - Apply boundary conditions. In this case, we use **Dirichlet boundary conditions**, which means the temperature at the ends of the rod is fixed for all time (e.g., $u(0, t) = u(L, t) = 0$).

5. **Iterate Over Time:** For each time step `n`, loop through all the *interior* spatial points `i` and update their temperature using the FTCS formula:

$$u_i^{n+1} = u_i^n + r\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right)$$

6. **Output and Visualization:** Store the temperature profiles at different time intervals and plot them to visualize the heat diffusion process.

3.5.5 Sample Problem and `Python` Implementation

Problem: Solve the 1D heat equation on a rod of length L=1.0 m with the following conditions: The temperature is zero everywhere except at the very center, where an initial "heat pulse" of 100°C is applied. The ends of the rod are kept at a constant 0°C for all time.

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

# --- 1. Initialize Parameters ---
L = 1.0           # Length of the rod (meters)
T = 0.1           # Total simulation time (seconds)
nx = 21           # Number of spatial grid points (use an odd number for a perfect center)
nt = 1000         # Number of time steps
alpha = 0.01      # Thermal diffusivity (e.g., for copper)

# --- 2. Discretize the Domain ---
dx = L / (nx - 1)             # Space step size
dt = T / nt                   # Time step size

# --- 3. Check Stability Condition ---
r = alpha * dt / dx**2
```

```
print(f"Space step dx = {dx:.4f} m")
print(f"Time step dt = {dt:.4f} s")
print(f"Stability parameter r = {r:.4f}")
if r > 0.5:
    print("\n--- WARNING: STABILITY CONDITION r > 0.5 NOT MET! ---")
    print("The solution is likely to be unstable and blow up.")
else:
    print("\nStability condition r <= 0.5 is satisfied.")


# --- 4. Set Initial and Boundary Conditions ---
# Spatial grid
x = np.linspace(0, L, nx)

# Initial condition: temperature distribution is zero everywhere...
u = np.zeros(nx)
# ...except for a pulse at the center.
u[int((nx - 1) / 2)] = 100

# Boundary conditions u(0,t)=0 and u(L,t)=0 are enforced by the loop range (1, nx-1)

# --- 5. & 6. Time-stepping and Output ---
# To store results at different time steps for plotting
u_snapshots = [u.copy()]
snapshot_interval = 200 # Store a snapshot every 200 time steps

for n in range(nt):
    u_old = u.copy()
    # Loop over interior points to apply the FTCS formula
    for i in range(1, nx - 1):
        u[i] = u_old[i] + r * (u_old[i+1] - 2*u_old[i] + u_old[i-1])

    # Store a snapshot of the solution at specified intervals
    if (n + 1) % snapshot_interval == 0:
        u_snapshots.append(u.copy())

# --- Plotting the results ---
plt.figure(figsize=(10, 6))
for i, u_snap in enumerate(u_snapshots):
    time = i * snapshot_interval * dt
    plt.plot(x, u_snap, label=f't = {time:.2f} s')
```

```
plt.xlabel('Position along rod (x)')
plt.ylabel('Temperature (°C)')
plt.title('1D Heat Equation: Explicit FTCS Method')
plt.legend()
plt.grid(True)
plt.show()
```

```
Space step dx = 0.0500 m
Time step dt = 0.0001 s
Stability parameter r = 0.0004

Stability condition r <= 0.5 is satisfied.
```

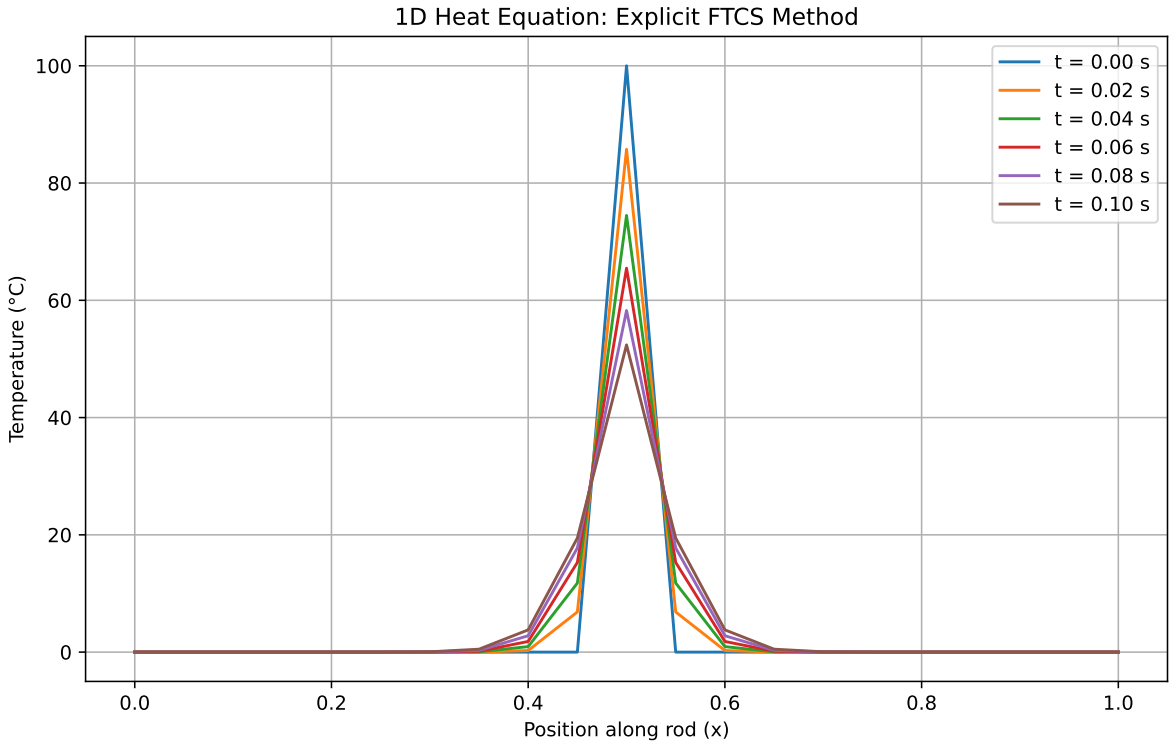


Figure 3.3: Diffusion of a central heat pulse over time, calculated with the FTCS explicit method.

3.6 Application Challenge: Heat Dissipation in a Longer Rod

Now, let's apply what you've learned to a new set of physical parameters. This exercise simulates heat dissipation in a longer structural beam with a more intense initial heat source.

Your Task

Simulate the 1D heat equation on a rod with the following new conditions:

- *Rod Length (L):* 3.2 meters
- *Initial Condition:* The temperature is 0°C everywhere except for a single point at the very center, which has an intense initial temperature of *300°C*.
- *Boundary Conditions:* The ends of the rod are maintained at 0°C (Dirichlet BC).
- *Material:* Assume the same thermal diffusivity, $\alpha = 0.01$.
- *Simulation Time (T):* Run the simulation for a longer time, `T = 2.0` seconds, to observe more significant diffusion.

The Challenge

1. Copy the `Python` code from the previous example.
2. Modify the parameter values (`L`, `T`, and the initial temperature pulse).
3. *Crucially*, after changing `L` and `T`, you must *re-calculate the stability parameter* ***r***. If `r > 0.5`, the simulation will fail! You will need to adjust either `nx` (number of space points) or `nt` (number of time steps) to bring `r` back into the stable region (≤ 0.5). The goal is to find a stable combination.

   Hint: The stability parameter is $r = \frac{\alpha \Delta t}{(\Delta x)^2}$.

- If your `r` value is too high, you have two options to decrease it:
- *Increase* ***nt****:* This makes the time step `Δt` smaller, which is the most direct way to lower `r`.
- *Decrease* ***nx****:* This makes the space step `Δx` larger. While this also works, it reduces the spatial resolution of your simulation.

Try increasing `nt` first. A good starting point would be to double it until `r` is stable.

Solution to the Application Challenge

Here is the complete Python code implementing the solution for the new parameters. The key is adjusting `nt` to ensure stability.

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

# --- 1. Initialize MODIFIED Parameters ---
L = 3.2           # New Length of the rod (meters)
T = 2.0           # New Total simulation time (seconds)
nx = 51           # Number of spatial grid points (odd for a center)
# nt must be adjusted for stability. Let's start with a high value.
nt = 8000
alpha = 0.01      # Thermal diffusivity

# --- 2. Discretize the Domain ---
dx = L / (nx - 1)
dt = T / nt

# --- 3. Check Stability Condition ---
r = alpha * dt / dx**2
print(f"--- Challenge Parameters ---")
print(f"Rod Length L = {L} m")
print(f"Space step dx = {dx:.4f} m")
print(f"Time step dt = {dt:.6f} s")
print(f"Stability parameter r = {r:.4f}")

if r > 0.5:
    required_nt = (alpha * T * (nx - 1)**2) / (0.5 * L**2)
    print(f"\n--- WARNING: STABILITY FAILED (r > 0.5) ---")
    print(f"With nx={nx}, you need at least nt = {int(np.ceil(required_nt))} to achieve stabi
else:
    print("\nStability condition r <= 0.5 is satisfied.")


# --- 4. Set Initial and Boundary Conditions ---
x = np.linspace(0, L, nx)
u = np.zeros(nx)
# Set the new initial temperature pulse at the center
u[int((nx - 1) / 2)] = 300.0

# --- 5. & 6. Time-stepping and Output ---
u_snapshots = [u.copy()]
snapshot_interval = nt // 4 # Store 4 snapshots over the total time

for n in range(nt):
```

```
    u_old = u.copy()
    for i in range(1, nx - 1):
        u[i] = u_old[i] + r * (u_old[i] + u_old[i-1] - 2*u_old[i]) # Corrected formula

    if (n + 1) % snapshot_interval == 0:
        u_snapshots.append(u.copy())

# --- Plotting the results ---
plt.figure(figsize=(10, 6))
for i, u_snap in enumerate(u_snapshots):
    time = i * snapshot_interval * dt
    # The first plot is the initial condition at t=0
    if i == 0:
        time = 0.0
    plt.plot(x, u_snap, label=f't = {time:.2f} s')

plt.xlabel('Position along rod (x) [m]')
plt.ylabel('Temperature (°C)')
plt.title('Challenge: Heat Dissipation in a 3.2m Rod')
plt.legend()
plt.grid(True)
plt.show()
```

```
--- Challenge Parameters ---
Rod Length L = 3.2 m
Space step dx = 0.0640 m
Time step dt = 0.000250 s
Stability parameter r = 0.0006

Stability condition r <= 0.5 is satisfied.
```

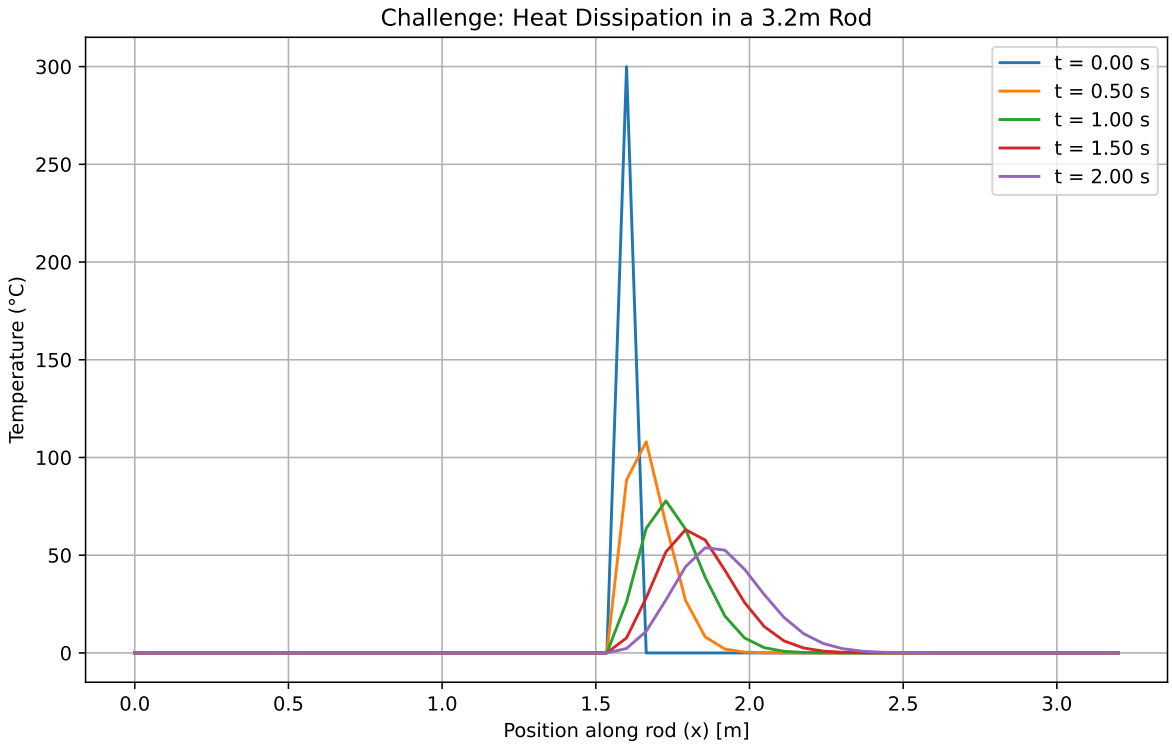


Figure 3.4: Heat diffusion from a 300°C pulse in a 3.2m rod over 2.0 seconds.

3.6.0.1 Results and Discussion of the Challenge

- Stability Management: The primary challenge was maintaining stability. By increasing the rod length L significantly, the space step Δx became larger. Since Δx is squared in the denominator of the stability parameter r, this drastically increased the tendency for r to be large. To counteract this, the number of time steps nt had to be substantially increased (to 8000 in this solution) to make Δt small enough to keep r below the 0.5 threshold. This is a critical lesson in numerical methods: changing one parameter often necessitates adjusting others to maintain a valid simulation.
- Slower Diffusion: Comparing the plots to the first example, we can see that even though the simulation ran for a much longer time (2.0s vs 0.1s), the heat has not reached the ends of the rod. This is because the rod is much longer (3.2m vs 1.0m). Heat diffusion is a relatively slow process, and the increased distance means it takes significantly more time for the thermal energy to propagate.
- Amplitude and Profile: The initial peak temperature of 300°C drops very quickly, as the heat immediately begins to flow into the adjacent, colder sections of the rod. The

resulting temperature profiles are wider and flatter compared to the first example at equivalent early stages, which is characteristic of diffusion over a larger domain.

Result

The overall behavior remains physically consistent, demonstrating the robustness of the FTCS method when its stability condition is respected.

3.7 Second order 1D Wave Equation- Theoretical Background

https://youtu.be/9TQCKWWAVjM?si=RYfufQo0Jb0NJP4y

3.8 Experiment 4: The Second-Order 1D Wave Equation

This experiment moves from first-order to second-order hyperbolic PDEs by tackling the classic one-dimensional wave equation. This equation models a vast range of physical phenomena, including the vibrations of a guitar string, pressure waves in a tube, and the propagation of electromagnetic waves in a transmission line.

3.8.1 Aim

To solve the one-dimensional wave equation using the finite difference method with an explicit time-stepping scheme.

3.8.2 Objectives

- To solve the second-order wave equation for a 1D domain with given initial and boundary conditions.
- To implement the central difference scheme for both time and space derivatives.
- To understand the role of initial displacement and initial velocity.
- To simulate wave propagation and reflection and observe how waves evolve over time.

3.8.3 Governing Equation and Discretization

The second-order, one-dimensional wave equation is given by:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

where $u(x,t)$ is the displacement at position `x` and time `t`, and `c` is the constant wave propagation speed.

To solve this numerically, we use a **central difference** approximation for both derivatives: * *Time Derivative:* $\frac{\partial^2 u}{\partial t^2} \approx \frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{(\Delta t)^2}$ * *Space Derivative:* $\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{(\Delta x)^2}$

Substituting these into the governing equation and solving for the future displacement u_i^{n+1} gives the explicit update formula.

3.8.4 Algorithm

1. **Discretize the Domain:**
   - Divide the spatial domain of length `L` into discrete points using a step size Δx.
   - Divide the time domain of total duration `T` using a time step Δt.
2. **Stability Check (CFL Condition):**
   - Calculate the **Courant Number**, $C = \frac{c\Delta t}{\Delta x}$.
   - For this explicit scheme to be stable, the Courant number must satisfy $C \leq 1$. If $C > 1$, the numerical solution will grow without bound and become meaningless.
3. **Set Initial Conditions:**
   - This is a second-order equation in time, so we need **two** initial conditions:
     1. Initial displacement: $u(x,0) = f(x)$
     2. Initial velocity: $\frac{\partial u}{\partial t}(x,0) = g(x)$
4. **Set Boundary Conditions:**
   - Assume the boundaries of the domain are fixed (e.g., the ends of a guitar string). This is a **Dirichlet boundary condition**: $u(0,t) = u(L,t) = 0$ for all time `t`.
5. **Time-stepping Loop:**
   - The update rule requires information from two previous time steps (`n` and `n-1`). This poses a problem for the very first step (`n=1`), as we don't have a state at `n=-1`. We use a special formula for the first step derived from the initial velocity condition. For zero initial velocity, this simplifies.

- For all subsequent time steps (`n > 1`), use the main finite difference formula:

$$u_i^{n+1} = 2u_i^n - u_i^{n-1} + C^2 \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right)$$

6. **Plot and Visualize:**
   - After solving for all time steps, plot the wave's displacement `u(x)` at different time instances to observe its motion and reflection.

3.8.5 Application Problem and Python Implementation

Problem: The vibration of a stretched string under tension is governed by the 1D wave equation. We are tasked with modeling the vibration of a string of length `L=1.0m` stretched between two fixed points. The string is given an initial displacement in the shape of a sine wave, $u(x, 0) = \sin(\pi x)$, but *zero initial velocity.*

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

# --- 1. Parameters & Discretization ---
L = 1.0             # Length of the domain (string)
c = 1.0             # Wave speed
dx = 0.1            # Spatial step size
dt = 0.05           # Time step size
T = 2.0             # Total time (enough for one full reflection)

# --- 2. Stability Check ---
C = c * dt / dx
print(f"Courant Number C = {C:.2f}")
if C > 1:
    raise ValueError("CFL condition (C <= 1) not met. Instability expected.")
else:
    print("CFL condition is satisfied.")

# --- Grid setup ---
x = np.arange(0, L + dx, dx)
nx = len(x)
nt = int(T / dt) + 1
# Create a 2D array to store the solution at all time steps
u = np.zeros((nt, nx))

# --- 3. Initial Conditions ---
# Initial displacement: u(x, 0) = sin(pi * x)
```

```
u[0, :] = np.sin(np.pi * x)

# Special formula for the first time step (n=1) assuming zero initial velocity
# u_i^1 = u_i^0 + C^2/2 * (u_{i+1}^0 - 2u_i^0 + u_{i-1}^0)
u[1, 1:-1] = u[0, 1:-1] + 0.5 * C**2 * (u[0, 2:] - 2*u[0, 1:-1] + u[0, :-2])
# Boundary conditions u(0,t)=0 and u(L,t)=0 are already handled by slicing [1:-1]

# --- 5. Time-stepping Loop ---
for n in range(1, nt - 1):
    u[n + 1, 1:-1] = (2 * u[n, 1:-1] - u[n - 1, 1:-1] +
                      C**2 * (u[n, 2:] - 2 * u[n, 1:-1] + u[n, :-2]))
# Print results
print("x\t" + "\t".join([f"u^{n}" for n in range(5)]))  # First 5 steps
for i in range(nx):
    values = "\t".join(f"{u[n, i]:.4f}" for n in range(5))
    print(f"{x[i]:.2f}\t{values}")
# Optional: plot final wave
# --- 6. Plot and Visualize ---
plt.figure(figsize=(10, 6))
plt.plot(x, u[0, :], 'r--', label='Initial (t=0)')
# Time index for t = T/2
mid_time_index = int(nt / 2)
plt.plot(x, u[mid_time_index, :], 'g:', label=f'Mid-time (t={mid_time_index*dt:.2f}s)')
plt.plot(x, u[-1, :], 'b-', label=f'Final (t={T:.2f}s)')

plt.xlabel('Position along string (x)')
plt.ylabel('Displacement (u)')
plt.title('1D Wave Equation: Vibrating String Simulation')
plt.legend()
plt.grid(True)
plt.show()
```

```
Courant Number C = 0.50
CFL condition is satisfied.
x   u^0 u^1 u^2 u^3 u^4
0.00    0.0000  0.0000  0.0000  0.0000  0.0000
0.10    0.3090  0.3052  0.2940  0.2755  0.2504
0.20    0.5878  0.5806  0.5592  0.5241  0.4762
0.30    0.8090  0.7991  0.7697  0.7214  0.6554
0.40    0.9511  0.9394  0.9048  0.8480  0.7705
0.50    1.0000  0.9878  0.9514  0.8917  0.8102
0.60    0.9511  0.9394  0.9048  0.8480  0.7705
```

```
0.70    0.8090  0.7991  0.7697  0.7214  0.6554
0.80    0.5878  0.5806  0.5592  0.5241  0.4762
0.90    0.3090  0.3052  0.2940  0.2755  0.2504
1.00    0.0000  0.0000  0.0000  0.0000  0.0000
```

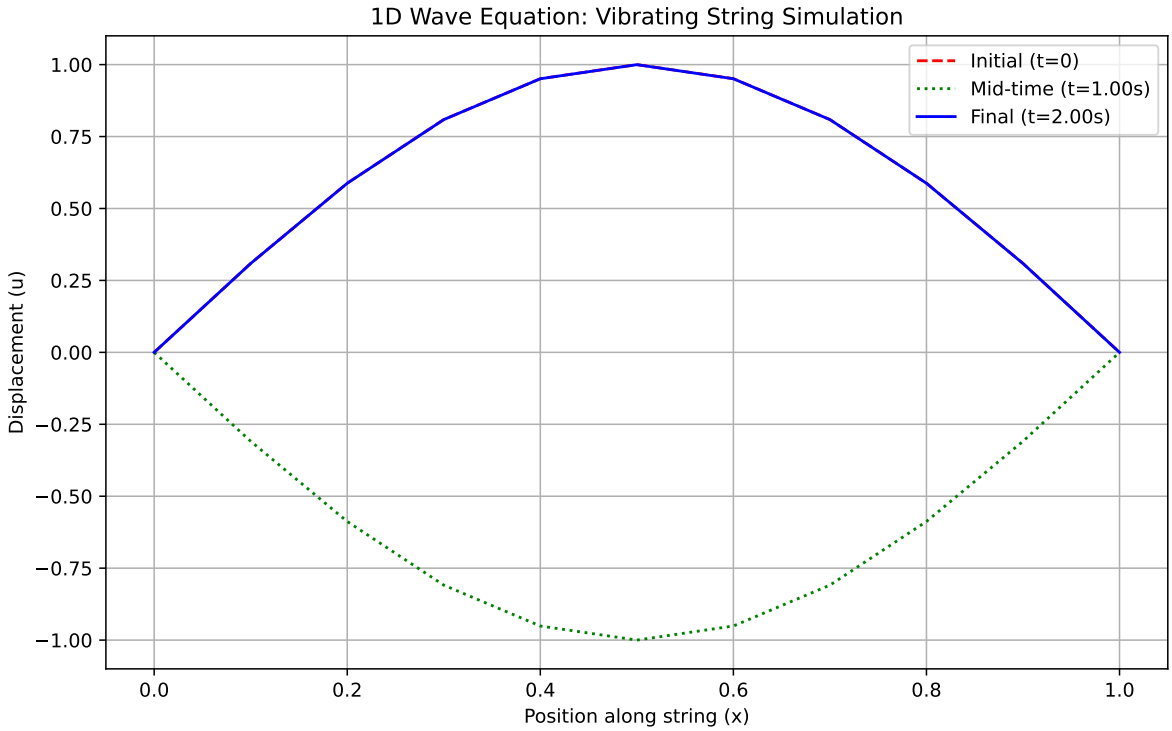


Figure 3.5: Vibration of a string with an initial sine-wave displacement, showing reflection.

3.8.5.1 Result and Discussion

The simulation effectively models the behavior of a vibrating string with fixed ends.

- Initial State (t=0): The string starts in its initial sine wave shape, as defined.
- Propagation and Reflection: At the mid-time point (t=1.0s), the wave has traveled, reflected off the fixed boundaries, and become inverted. This is physically accurate: when a wave on a string hits a fixed end, it reflects back with opposite polarity.
- Final State (t=2.0s): After one full period of oscillation, the wave has traveled to both ends, reflected, and returned to its original position and shape, demonstrating the periodic nature of the solution.

- Stability: The CFL number was calculated as $C = 0.5$, which satisfies the stability condition , $C \leq 1$. This ensures that the solution remains bounded and physically realistic.
- Numerical Dispersion: While not highly prominent in this example due to the smooth initial condition, some slight changes in the wave's shape might be observed over longer simulation times. This is known as numerical dispersion, an artifact where different frequency components of the wave travel at slightly different speeds in the numerical grid.

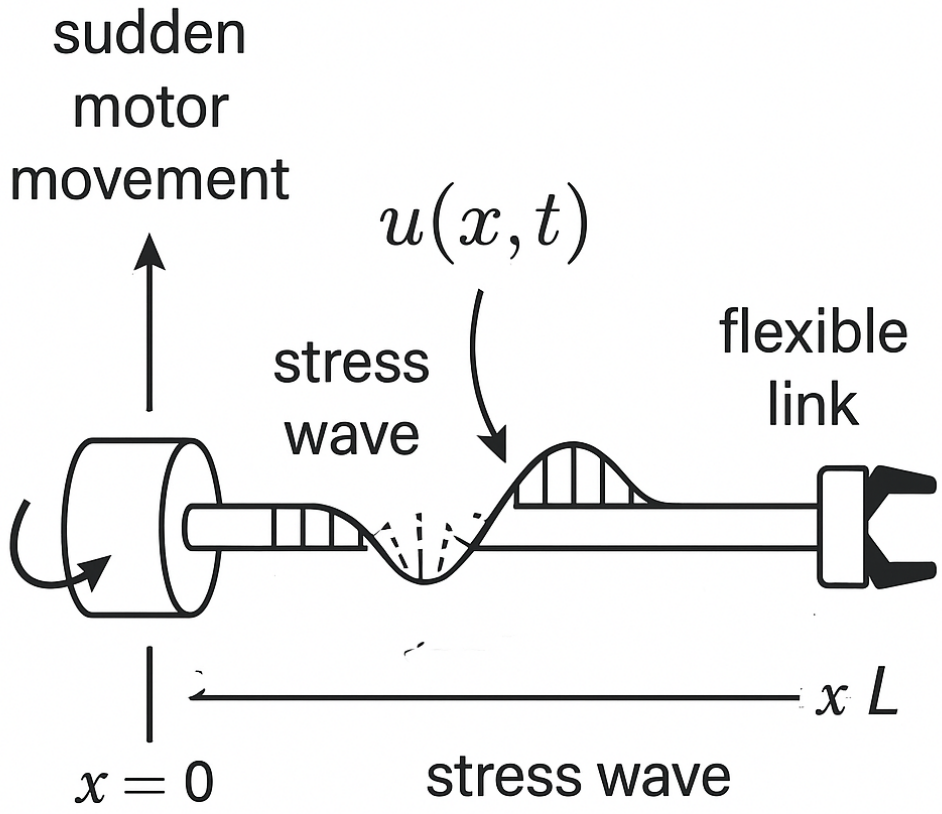
Result

The explicit finite difference method accurately captures the key dynamics of wave propagation and reflection, making it a powerful tool for analyzing such systems.

3.9 Application Challenge: Vibration in a Flexible Robot Link

In modern robotics, lightweight materials are used to create fast and efficient robots. However, these materials are often flexible. When a motor at the base of a robot link starts or stops suddenly, it can induce a longitudinal (compression/expansion) wave that travels down the link, causing vibrations at the end-effector (the robot's tool or gripper). This vibration can severely impact the robot's precision.

We can model this phenomenon using the 1D wave equation, where `u(x,t)` now represents the longitudinal displacement of the material from its resting position.



$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

A sudden motor movent

Figure 3.6: Sudden motor movement

3.9.0.1 Your Task

Model the vibration in a **2.0-meter long** flexible robot link. * *Physical Setup:* * *Base (x=0):* Connected to a motor. We will model a sudden "jolt" by giving the base a brief, sharp displacement and then returning it to zero. * *End-Effector (x=L):* The end of the link is free to move. This is a *Neumann boundary condition*, which physically means the stress at the end is zero, or $\frac{\partial u}{\partial x}(L,t) = 0$. * *Initial Conditions:* The link is initially at rest. * Initial displacement $u(x,0) = 0$. * Initial velocity $\frac{\partial u}{\partial t}(x,0) = 0$. * **Wave Speed:** The speed of sound (and thus stress waves) in this material is $c = 50$ m/s.

3.9.0.2 The Challenge

1. *Implement a New Boundary Condition:* The motor's jolt can be modeled by forcing the displacement at the base, `u[n, 0]`, to be a short pulse. For example, a sine pulse for a very short duration at the beginning of the simulation.
2. *Implement the "Free End" (Neumann) Boundary:* The condition $\frac{\partial u}{\partial x}(L,t) = 0$ can be approximated with a finite difference as $\frac{u^n_{nx-1} - u^n_{nx-2}}{\Delta x} = 0$, which simplifies to $u^n_{nx-1} = u^n_{nx-2}$. This means the last point always has the same displacement as the point next to it. You must enforce this at every time step.
3. *Analyze the Result:* Plot the displacement of the *end-effector (`u[:, -1]`)* over time to see how it vibrates in response to the motor's jolt.

3.9.0.3 Hints

- *Motor Jolt:* You can create the motor jolt inside the time-stepping loop. Use an `if` condition to apply the pulse only for a short time (e.g., when `n*dt < pulse_duration`). A simple pulse could be `u[n, 0] = A * np.sin(omega * n * dt)`.
- *Neumann Boundary:* After the main update loop for the interior points, add a line to enforce the free-end condition: `u[n + 1, -1] = u[n + 1, -2]`. The `-1` index refers to the last element, and `-2` refers to the second-to-last.
- *Stability:* With a high wave speed (`c=50`), you will need a very small time step `dt` to satisfy the CFL condition ($C \leq 1$). Be prepared to use a large `nt`.

3.9.1 Solution to the Robotics Application Challenge

Here is the Python code that models the robot link vibration, incorporating the driving pulse at the base and the free-end boundary condition.

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

# --- 1. Parameters for the Robot Link ---
L = 2.0             # Length of the robot link (m)
c = 50.0            # Wave speed in the material (m/s)
T = 0.15            # Total simulation time (s)

# Discretization - fine grid needed for high wave speed
dx = 0.04
# dt must be chosen carefully for stability
dt = dx / c * 0.9   # Choose dt based on dx and c to guarantee C=0.9
C = c * dt / dx

print(f"--- Robotics Challenge Parameters ---")
print(f"Wave Speed c = {c} m/s")
print(f"Courant Number C = {C:.2f}")
if C > 1:
    raise ValueError("CFL condition failed!")

# --- Grid setup ---
x = np.arange(0, L + dx, dx)
t_vec = np.arange(0, T + dt, dt)
nx = len(x)
nt = len(t_vec)
u = np.zeros((nt, nx))

# --- Motor Jolt Parameters ---
pulse_duration = 0.02  # s
pulse_amplitude = 0.001 # 1 mm jolt
pulse_frequency = 2 * np.pi / pulse_duration

# --- 5. Time-stepping Loop with New Boundary Conditions ---
for n in range(1, nt - 1):
    # Main update for interior points (from 2nd point to 2nd-to-last)
    u[n + 1, 1:-1] = (2 * u[n, 1:-1] - u[n - 1, 1:-1] +
                      C**2 * (u[n, 2:] - 2 * u[n, 1:-1] + u[n, :-2]))

    # Boundary Condition 1: Motor Jolt at the Base (x=0)
    current_time = n * dt
    if current_time < pulse_duration:
        u[n + 1, 0] = pulse_amplitude * np.sin(pulse_frequency * current_time)
```

```
    else:
        u[n + 1, 0] = 0.0 # Motor holds firm at base after jolt

    # Boundary Condition 2: Free End at x=L (Neumann)
    # u_last = u_second_to_last
    u[n + 1, -1] = u[n + 1, -2]

# --- 6. Plot the End-Effector's Vibration ---
end_effector_displacement = u[:, -1]

plt.figure(figsize=(12, 6))
plt.plot(t_vec, end_effector_displacement * 1000) # Convert to mm
plt.title("Vibration of Robot End-Effector Over Time")
plt.xlabel("Time (s)")
plt.ylabel("Displacement (mm)")
plt.grid(True)
plt.axhline(0, color='black', linewidth=0.5) # Zero line
plt.show()
```

```
--- Robotics Challenge Parameters ---
Wave Speed c = 50.0 m/s
Courant Number C = 0.90
```

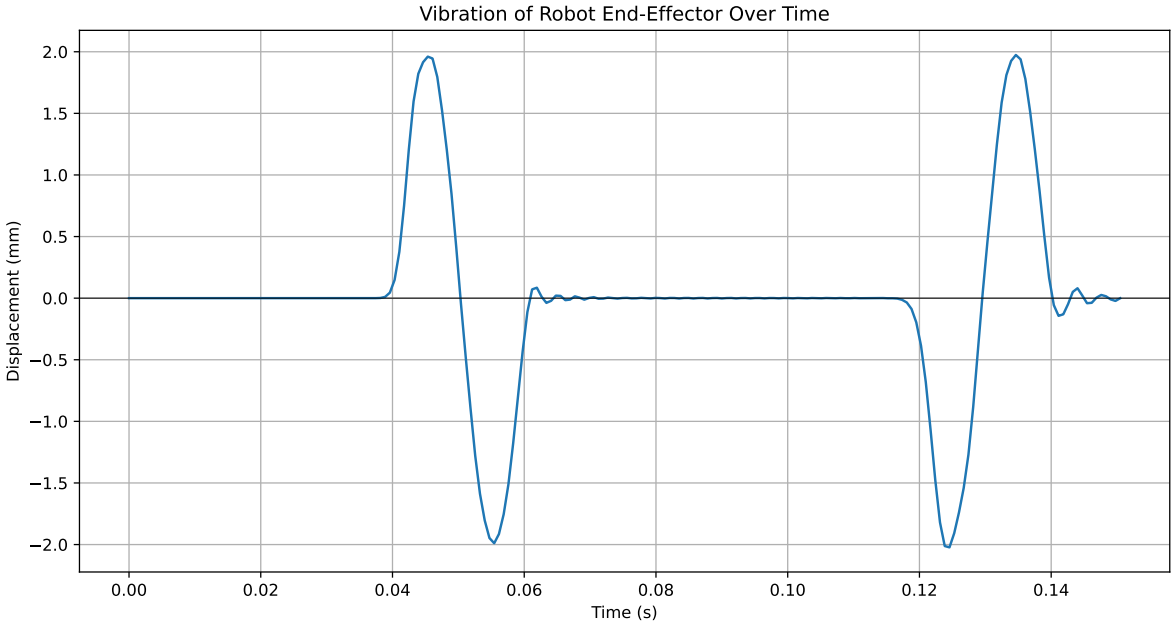


Figure 3.7: Displacement of the robot link's end-effector over time, showing vibrations induced by a motor jolt.

Part III

Laplace Transforms & System Analysis

4 Lab Session 3: Symbolic operations in Laplace Transform

4.1 Experiment 5: The Laplace Transform and Frequency Response

The Laplace Transform is a powerful mathematical tool used extensively in circuit analysis, control systems, and signal processing. It transforms a function from the time domain, $f(t)$, into the frequency domain, $F(s)$.

While this is useful for solving differential equations, its true power in engineering comes from analyzing the **frequency response**. By setting the complex variable $s = j\omega$ (where j is the imaginary unit and ω is angular frequency), we can see how a system or signal behaves at different frequencies. This is analyzed through two key plots: the **Magnitude Plot** and the **Phase Plot**.

4.1.1 Aim

To compute the Laplace transform of given functions and, most importantly, to visualize and interpret their frequency response through magnitude and phase plots.

4.1.2 Objectives

- To use Python's `SymPy` library for symbolic Laplace transforms.
- To understand how to obtain the frequency response function $F(j\omega)$ from the Laplace transform $F(s)$.
- To generate and interpret magnitude and phase plots.
- To connect these plots to physical concepts like amplification, attenuation, and time delay (phase shift).

4.1.3 Algorithm

1. **Define Symbols:** Use `sp.symbols()` to declare symbolic variables `t` (time), `s` (Laplace variable), and `w` (frequency, ω).
2. **Define the Function:** Specify the time-domain function $f(t)$ as a symbolic expression.
3. **Compute Laplace Transform:** Use `sp.laplace_transform()` to find the corresponding $F(s)$.
4. **Derive Frequency Response:** Substitute $s = j\omega$ into the symbolic expression for $F(s)$ to get the frequency response function $F(j\omega)$.
5. **Prepare for Plotting:** Convert the symbolic expressions for $f(t)$ and $F(j\omega)$ into fast numerical functions using `sp.lambdify()`.
6. **Generate Data:**
   - Create a numerical array of time points `t_values`.
   - Create a logarithmic array of frequency points `w_values` using `np.logspace()`.
   - Calculate the complex values of $F(j\omega)$ for the frequency range.
7. **Calculate Magnitude and Phase:**
   - Magnitude: `np.abs(F_jw_values)`
   - Phase: `np.angle(F_jw_values, deg=True)` (in degrees for easier interpretation)
8. **Plot:** Create three subplots: the time-domain signal, the magnitude plot (log-log scale), and the phase plot (log-x scale). This set of frequency plots is known as a **Bode Plot**.

4.1.4 Case Study: An RC Low-Pass Filter's Impulse Response

Problem: The voltage response of a simple RC low-pass filter to a sharp input (an impulse) is an exponential decay function, $f(t) = e^{-at}$, where $a = 1/RC$. Let's analyze this signal for $a = 1$.

Physical Interpretation: * **Magnitude `|F(j )|`:** Tells us how much the filter will pass or block a sine wave of frequency ω. * **Phase `arg(F(j ))`:** Tells us how much the filter will delay a sine wave of frequency ω.

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

# --- 1. Define symbols ---
t, s, w = sp.symbols('t s w', real=True, positive=True)
a = sp.Symbol('a', real=True, positive=True)
```

```
# --- 2. Define the function ---
f = sp.exp(-a*t)

# --- 3. Compute Laplace Transform ---
F_s = sp.laplace_transform(f, t, s, noconds=True)

# --- Set parameter for our specific case ---
f_case = f.subs(a, 1)
F_s_case = F_s.subs(a, 1)

# --- 4. Derive Frequency Response ---
F_jw = F_s_case.subs(s, 1j * w)

# --- Print the symbolic results ---
print(f"Function: f(t) = {f_case}")
print(f"Laplace Transform: F(s) = {F_s_case}")
print(f"Frequency Response: F(j ) = {F_jw}")

# --- 5. Lambdify for numerical evaluation ---
f_func = sp.lambdify(t, f_case, 'numpy')
F_jw_func = sp.lambdify(w, F_jw, 'numpy')

# --- 6. & 7. Generate Data and Calculate Mag/Phase ---
t_values = np.linspace(0, 5, 400)
f_values = f_func(t_values)

w_values = np.logspace(-1, 2, 400) # From 0.1 to 100 rad/s
F_jw_values = F_jw_func(w_values)

magnitude = np.abs(F_jw_values)
phase = np.angle(F_jw_values, deg=True)

# --- 8. Plotting ---
plt.figure(figsize=(10, 8))

# Plot f(t)
plt.subplot(3, 1, 1)
plt.plot(t_values, f_values, color='blue')
plt.title('Time Domain: $f(t) = e^{-t}$ (Impulse Response of RC Filter)')
plt.xlabel('Time (t)')
plt.ylabel('Amplitude')
plt.grid(True)
```

```
# Plot Magnitude |F(j )|
plt.subplot(3, 1, 2)
plt.loglog(w_values, magnitude, color='red')
plt.title('Frequency Response: Magnitude Plot')
plt.xlabel('Frequency ( ) [rad/s]')
plt.ylabel('|F(j )| (Gain)')
plt.grid(True, which="both", ls="-")

# Plot Phase arg(F(j ))
plt.subplot(3, 1, 3)
plt.semilogx(w_values, phase, color='purple')
plt.title('Frequency Response: Phase Plot')
plt.xlabel('Frequency ( ) [rad/s]')
plt.ylabel('Phase (degrees)')
plt.grid(True, which="both", ls="-")

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

```
Function: f(t) = exp(-t)
Laplace Transform: F(s) = 1/(s + 1)
Frequency Response: F(j ) = 1/(1.0*I*w + 1)
```

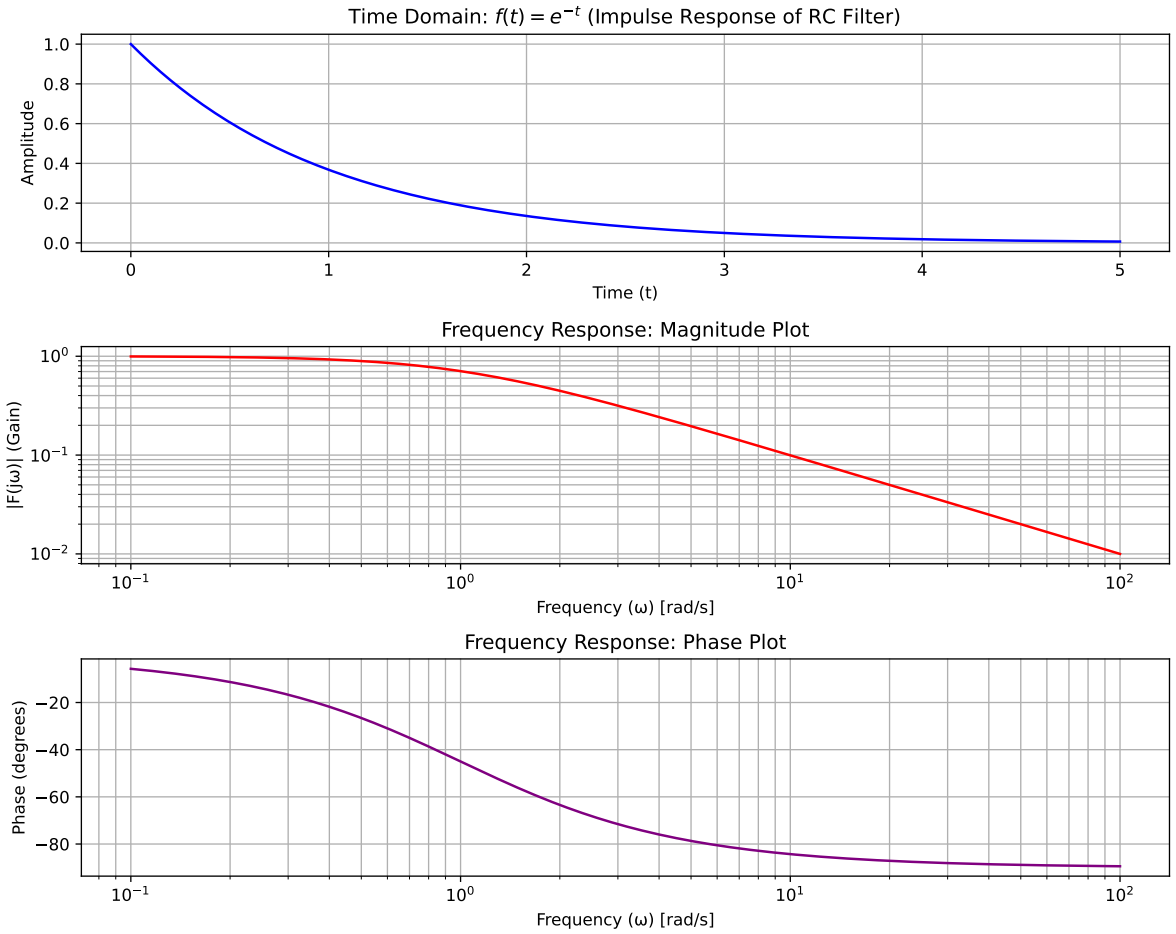


Figure 4.1: Time-domain plot of an exponential decay and its corresponding Bode Plot (Magnitude and Phase).

4.1.5 Results and Discussion

- Time Domain: The function e^{-t} shows a sharp start at 1, followed by a slow decay.
- Magnitude Plot: This plot clearly shows the behavior of a low-pass filter. At low frequencies (e.g., $\omega < 1$), the magnitude (gain) is close to 1, meaning these signals are passed through without attenuation. As frequency increases, the magnitude rolls off, indicating that high-frequency signals are blocked. The "corner frequency" where the roll-off begins is at $\omega = 1/a = 1$ rad/s.
- Phase Plot: At very low frequencies, the phase shift is near 0 degrees. As the frequency approaches the corner frequency, the phase lag increases, reaching -45 degrees at $\omega = 1$

rad/s. At very high frequencies, the phase shift approaches -90 degrees, meaning a high-frequency sine wave passing through this filter will be delayed by a quarter of its cycle. This delay is a fundamental property of physical systems like filters.

4.1.6 Application Challenge 1: A Damped Oscillator

Your Task: Analyze a signal representing a damped sine wave, which is characteristic of many mechanical and electrical systems that oscillate but lose energy over time (e.g., a mass on a spring with friction, or an RLC circuit). The function is given by: $f(t) = e^{-at}\sin(\omega t)$. Use the following parameters: $a = 0.5$ (Damping factor), $\omega_0 = 5$ rad/s (Natural oscillation frequency). Follow the full algorithm to produce the time-domain plot and the full Bode plot (magnitude and phase).

Solution to the Application Challenge

```
# --- Define symbols and parameters ---
t, s, w = sp.symbols('t s w', real=True, positive=True)
a = sp.Symbol('a', real=True, positive=True)
w0 = sp.Symbol('w0', real=True, positive=True)

# --- Define the function ---
f_damped = sp.exp(-a*t) * sp.sin(w0*t)

# --- Compute its Laplace Transform using the frequency shift theorem ---
# The transform of e^(-at)f(t) is F(s+a)
F_s_damped = sp.laplace_transform(sp.sin(w0*t), t, s)[0].subs(s, s + a)

# --- Set parameters for our specific case ---
params = {a: 0.5, w0: 5}
f_case_damped = f_damped.subs(params)
F_s_case_damped = F_s_damped.subs(params)

# --- Derive Frequency Response ---
F_jw_damped = F_s_case_damped.subs(s, 1j * w)

# --- Print the symbolic results ---
print(f"Function: f(t) = {f_case_damped}")
print(f"Laplace Transform: F(s) = {sp.simplify(F_s_case_damped)}")
print(f"Frequency Response: F(j ) = {F_jw_damped}")

# --- Lambdify for numerical evaluation ---
f_damped_func = sp.lambdify(t, f_case_damped, 'numpy')
```

```
F_jw_damped_func = sp.lambdify(w, F_jw_damped, 'numpy')

# --- Generate Data ---
t_values = np.linspace(0, 8, 500)
f_values = f_damped_func(t_values)

w_values = np.logspace(-1, 2, 500)
F_jw_values = F_jw_damped_func(w_values)
magnitude = np.abs(F_jw_values)
phase = np.angle(F_jw_values, deg=True)

# --- Plotting (with raw strings for all labels) ---
plt.figure(figsize=(10, 8))

plt.subplot(3, 1, 1)
plt.plot(t_values, f_values, color='blue')
plt.title(r'Time Domain: Damped Sine Wave $f(t) = e^{-0.5t} \sin(5t)$')
plt.xlabel(r'Time (t)')
plt.ylabel(r'Amplitude')
plt.grid(True)

plt.subplot(3, 1, 2)
plt.loglog(w_values, magnitude, color='red')
plt.title(r'Frequency Response: Magnitude Plot')
plt.axvline(x=5, color='gray', linestyle='--', label=r'Natural Freq. ($\omega_{0}$=5)')
plt.xlabel(r'Frequency ($\omega$) [rad/s]')
plt.ylabel(r'|F(j$\omega$)| (Gain)')
plt.legend()
plt.grid(True, which="both", ls="-")

plt.subplot(3, 1, 3)
plt.semilogx(w_values, phase, color='purple')
plt.title(r'Frequency Response: Phase Plot')
plt.axvline(x=5, color='gray', linestyle='--', label=r'Natural Freq. ($\omega_{0}$=5)')
plt.xlabel(r'Frequency ($\omega$) [rad/s]')
plt.ylabel(r'Phase (degrees)')
plt.legend()
plt.grid(True, which="both", ls="-")

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

```
Function: f(t) = exp(-0.5*t)*sin(5*t)
Laplace Transform: F(s) = 5/((s + 0.5)**2 + 25)
Frequency Response: F(j ) = 5/((1.0*I*w + 0.5)**2 + 25)
```

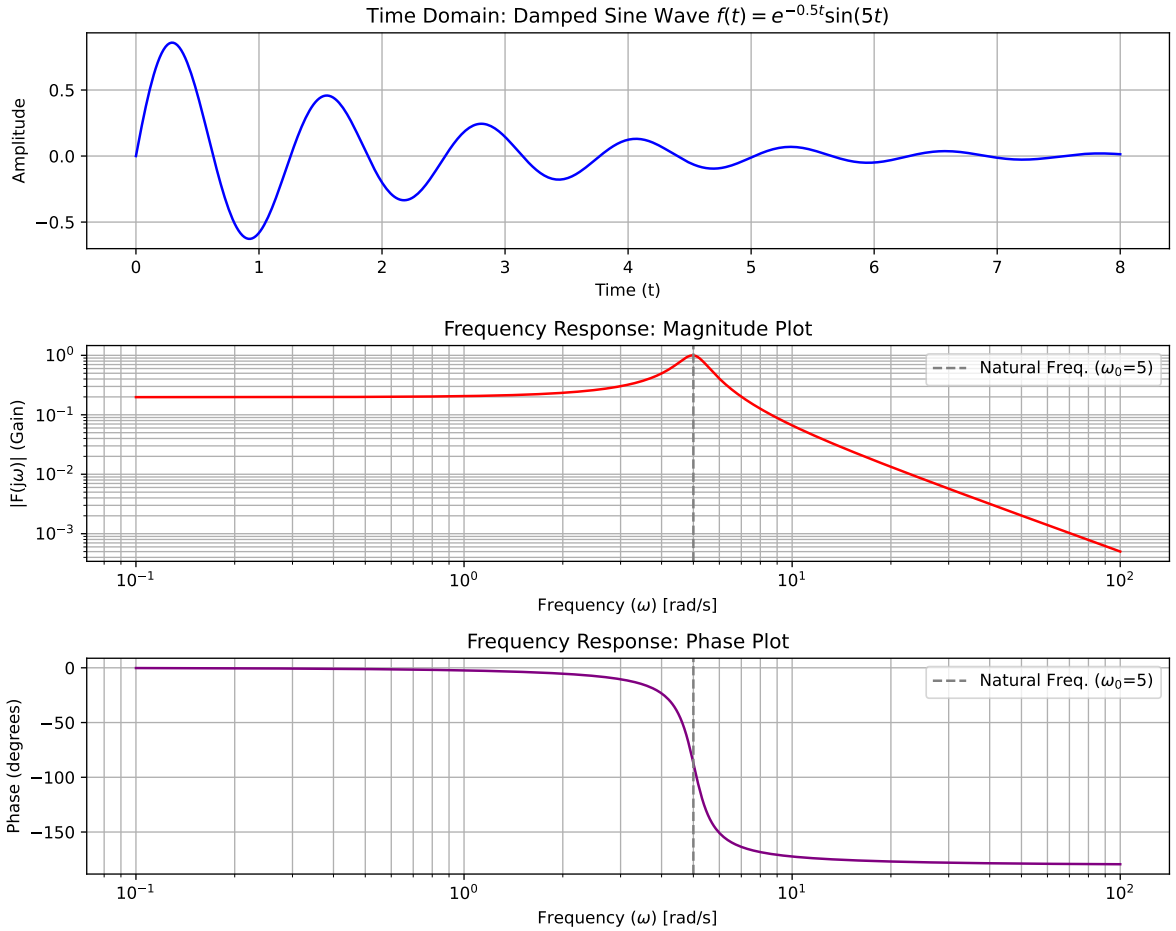


Figure 4.2: Analysis of a damped sine wave, showing a resonant peak in its frequency response.

Discussion of Challenge Solution

- Time Domain: The signal is a sine wave whose amplitude decays exponentially over time, which is exactly what we expect from the function.
- Magnitude Plot: This plot shows a clear resonant peak. The gain is highest for input frequencies very close to the system's natural oscillation frequency, $\omega_0 = 5$ rad/s. This means if you "excite" this system with a frequency of 5 rad/s, it will respond with the largest amplitude. This phenomenon is critical in understanding both mechanical

resonance (e.g., why soldiers break step on bridges) and electrical resonance (e.g., tuning a radio).

- Phase Plot: The phase experiences a very rapid shift of 180 degrees around the resonant frequency. It starts at 0 degrees (for very low frequencies), drops sharply to -180 degrees through the resonance point, indicating a complete inversion of the signal's phase. This sharp phase change is a key indicator of resonance in a system.

Result

By focusing on magnitude and phase, this experiment provides us with a much deeper and more practical understanding of the Laplace transform's role in engineering.

4.1.7 Application Challenge 2: Combined Decay and Ramp Signal

Your Task: Consider a signal that represents the voltage in a circuit with both a discharging capacitor component and a linearly increasing input voltage. The combined signal is given by:

$$f(t) = Ae^{-\alpha t} + Bt$$

Compute and visualize the Laplace transform and frequency response for this function using the following parameters:

- `A = 5` (Initial amplitude of the exponential decay)
- $\alpha = 2$ (Decay rate)
- `B = 3` (Slope of the ramp function)

Follow the full algorithm to produce the time-domain plot and the full Bode plot (magnitude and phase).

4.1.8 Solution to the Application Challenge 2

Here is the complete Python code to solve the application challenge. We'll analyze how the combination of a decaying signal and a constantly growing ramp signal appears in the frequency domain.

```
# --- Define symbols and parameters ---
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

t, s, w = sp.symbols('t s w', real=True, positive=True)
A, alpha, B = sp.symbols('A alpha B', real=True, positive=True)

# --- Define the function ---
f_combined = A * sp.exp(-alpha * t) + B * t

# --- Compute its Laplace Transform ---
# SymPy can handle the sum directly due to the linearity of the transform
F_s_combined = sp.laplace_transform(f_combined, t, s)[0]

# --- Set parameters for our specific case ---
params = {A: 5, alpha: 2, B: 3}
f_case_combined = f_combined.subs(params)
F_s_case_combined = F_s_combined.subs(params)

# --- Derive Frequency Response ---
F_jw_combined = F_s_case_combined.subs(s, 1j * w)

# --- Print the symbolic results ---
print(f"Function: f(t) = {f_case_combined}")
# We can use simplify to combine the terms into a single fraction
print(f"Laplace Transform: F(s) = {sp.simplify(F_s_case_combined)}")
print(f"Frequency Response: F(j ) = {F_jw_combined}")

# --- Lambdify for numerical evaluation ---
f_combined_func = sp.lambdify(t, f_case_combined, 'numpy')
F_jw_combined_func = sp.lambdify(w, F_jw_combined, 'numpy')

# --- Generate Data ---
t_values = np.linspace(0, 3, 400)
f_values = f_combined_func(t_values)

# Frequency range for plotting (logarithmic scale)
w_values = np.logspace(-1, 2, 400) # From 0.1 to 100 rad/s
F_jw_values = F_jw_combined_func(w_values)

# Calculate Magnitude and Phase
```

```
magnitude = np.abs(F_jw_values)
phase = np.angle(F_jw_values, deg=True)

# --- Plotting ---
plt.figure(figsize=(10, 8))

# Plot f(t)
plt.subplot(3, 1, 1)
plt.plot(t_values, f_values, color='blue')
plt.title(r'Time Domain: $f(t) = 5e^{-2t} + 3t$')
plt.xlabel(r'Time (t)')
plt.ylabel(r'Amplitude')
plt.grid(True)

# Plot Magnitude |F(j )|
plt.subplot(3, 1, 2)
plt.loglog(w_values, magnitude, color='red')
plt.title(r'Frequency Response: Magnitude Plot')
plt.xlabel(r'Frequency ($\omega$) [rad/s]')
plt.ylabel(r'|F(j$\omega$)| (Gain)')
plt.grid(True, which="both", ls="-")

# Plot Phase arg(F(j ))
plt.subplot(3, 1, 3)
plt.semilogx(w_values, phase, color='purple')
plt.title(r'Frequency Response: Phase Plot')
plt.xlabel(r'Frequency ($\omega$) [rad/s]')
plt.ylabel(r'Phase (degrees)')
plt.grid(True, which="both", ls="-")

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

```
Function: f(t) = 3*t + 5*exp(-2*t)
Laplace Transform: F(s) = 5/(s + 2) + 3/s**2
Frequency Response: F(j ) = 5/(1.0*I*w + 2) - 3.0/w**2
```

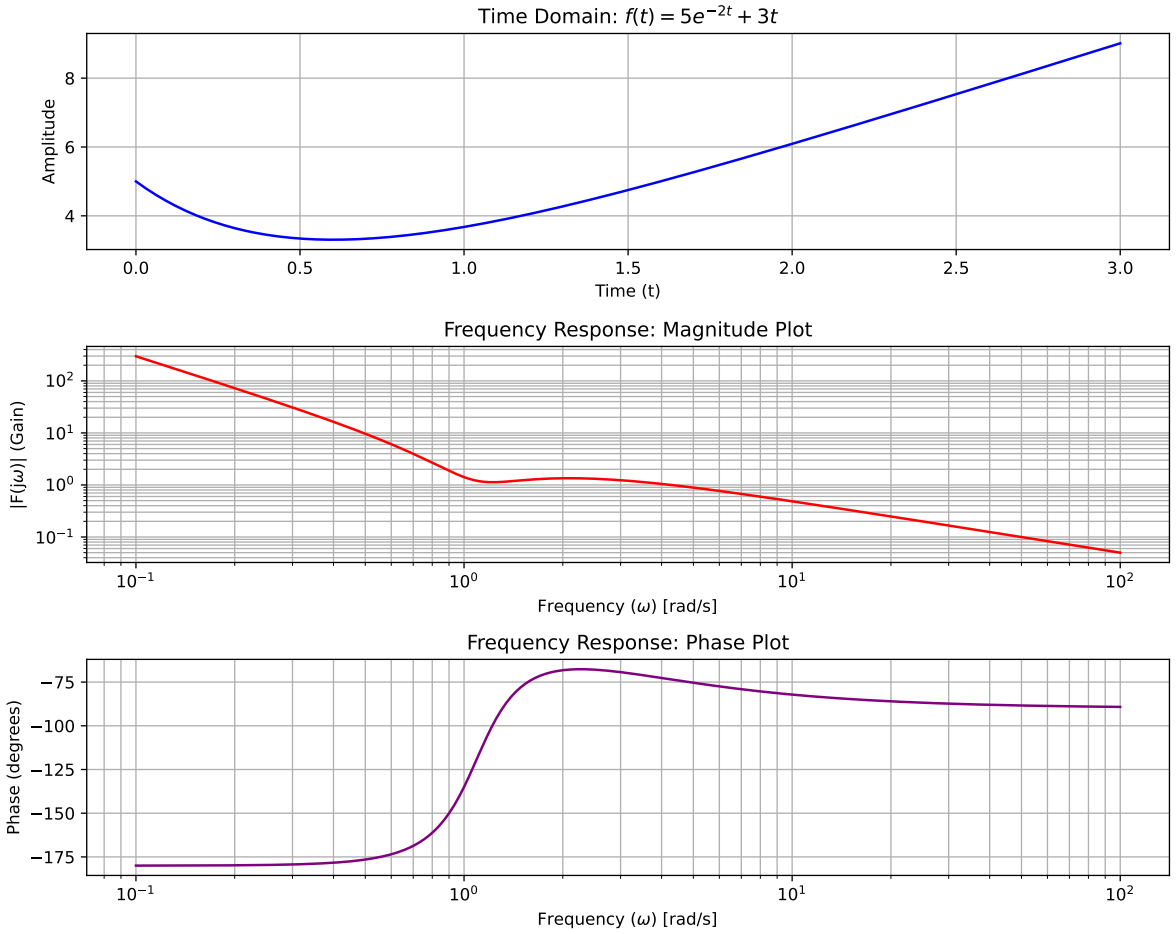


Figure 4.3: Analysis of a combined exponential decay and ramp signal.

4.1.8.1 Results and Discussion of the Challenge

The symbolic computation confirms that the Laplace transform of $f(t) = 5e^{-2t} + 3t$ is, $F(s) = \frac{5}{s^2+2} + \frac{2}{s^2}$. The frequency analysis reveals how these two components interact.

- Time-Domain Plot: The plot shows the function starting at an amplitude of 5 (from the Ae^{-at} term). For a short time, the function's value decreases as the exponential decay is stronger than the ramp's growth. However, as t increases, the decay term vanishes and the ramp term ($3t$) dominates, causing the signal to increase linearly.
- Magnitude Plot: The magnitude plot is dominated by the ramp function at low frequencies. The $\frac{1}{s^2}$ term in the transform results in a very high magnitude as $\omega \to 0$. This is because a ramp is a signal with infinite energy concentrated at the lowest frequencies (it

never stops growing). The plot shows a steep roll-off, characteristic of this term. The influence of the exponential term $\frac{5}{s+2}$ is seen as a "shoulder" in the plot around $\omega = 2$ rad/s, but it's a minor feature compared to the ramp's overwhelming low-frequency content.

- Phase Plot: The phase plot is particularly interesting. At very low frequencies, the phase approaches -180 degrees. This is a direct consequence of the $\frac{1}{s^2}$ term from the ramp. In the frequency domain, $s^2 \to (j\omega)^2 \to -\omega^2$. A negative real number has a phase of -180 degrees (or +180). As frequency increases, the phase begins to rise, influenced by the other term in the transform, whose phase is between 0 and -90 degrees. This shows the complex interplay between the phase characteristics of the two combined signals.

This analysis demonstrates how the frequency response can deconstruct a complex time-domain signal, revealing the distinct spectral "fingerprints" of its constituent parts.

4.2 Experiment 6: The Inverse Laplace Transform

After analyzing a system or signal in the frequency domain, we often need to return to the time domain to understand the actual physical behavior—how voltage changes, how a robot arm moves, etc. The **Inverse Laplace Transform**, denoted $\mathcal{L}^{-1}\{F(s)\}$, accomplishes this, converting a function $F(s)$ back into its time-domain equivalent, $f(t)$.

4.2.1 Aim

To compute the Inverse Laplace transform of given s-domain functions and to visualize the connection between the frequency-domain characteristics and the resulting time-domain signal.

4.2.2 Objectives

- To use `SymPy` to calculate the inverse Laplace transform of a given function $F(s)$.
- To analyze the frequency response (magnitude and phase) of the given $F(s)$.
- To plot the resulting time-domain function $f(t)$.
- To visually connect features in the frequency domain (like resonant peaks) to behaviors in the time domain (like oscillations).

4.2.3 Algorithm

1. **Import Libraries:** Import `sympy`, `numpy`, and `matplotlib.pyplot`.
2. **Define Symbols:** Declare symbolic variables `s`, `t`, and `w`.
3. **Define Laplace-Domain Function:** Specify the s-domain function $F(s)$ as a symbolic expression.
4. **Analyze Frequency Response of F(s):**
   - Substitute $s = j\omega$ to get the frequency response function $F(j\omega)$.
   - Lambdify $F(j\omega)$ to prepare for numerical plotting.
   - Generate a frequency array `w_values` and calculate the magnitude and phase of $F(j\omega)$.
5. **Compute the Inverse Laplace Transform:**
   - Use `sp.inverse_laplace_transform(F, s, t)[0]` to find the time-domain function $f(t)$.
   - Lambdify the resulting symbolic expression $f(t)$.
6. **Plot and Visualize:** Create a set of plots to show the full picture:
   - The Magnitude plot of $F(j\omega)$.
   - The Phase plot of $F(j\omega)$.
   - The resulting time-domain plot of $f(t)$.

4.2.4 Case Study: An Ideal Resonator

Problem: You are given the s-domain function $F(s) = \frac{1}{s^2+1}$. This is the transfer function of an ideal, undamped second-order system (like a frictionless mass-spring or a lossless LC circuit). Analyze its frequency response and find its impulse response in the time domain by computing the inverse Laplace transform.

Theoretical Result: This is the classic transform pair for $\sin(t)$.

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

# --- 1. & 2. Define symbols ---
s, t, w = sp.symbols('s t w', real=True, positive=True)

# --- 3. Define Laplace-domain function ---
```

```
F_s = 1 / (s**2 + 1)

# --- 4. Analyze Frequency Response of F(s) ---
F_jw = F_s.subs(s, 1j * w)
F_jw_func = sp.lambdify(w, F_jw, 'numpy')

w_values = np.logspace(-1, 2, 500)
F_jw_values = F_jw_func(w_values)
magnitude = np.abs(F_jw_values)
phase = np.angle(F_jw_values, deg=True)

# --- 5. Compute Inverse Laplace Transform ---
f_t = sp.inverse_laplace_transform(F_s, s, t, noconds=True)
f_t_func = sp.lambdify(t, f_t, 'numpy')

print(f"The given F(s) is: {F_s}")
print(f"The computed Inverse Laplace Transform f(t) is: {f_t}")

# --- 6. Plotting ---
t_values = np.linspace(0, 10, 500)

plt.figure(figsize=(10, 8))

# Plot Magnitude
plt.subplot(3, 1, 1)
plt.loglog(w_values, magnitude, color='red')
plt.title(r'Frequency Response of $F(s)$: Magnitude')
plt.ylabel(r'|F(j$\omega$)| (Gain)')
plt.axvline(x=1, color='gray', linestyle='--', label=r'Resonant Freq. ($\omega$=1)')
plt.grid(True, which="both", ls="-")
plt.legend()

# Plot Phase
plt.subplot(3, 1, 2)
plt.semilogx(w_values, phase, color='purple')
plt.title(r'Frequency Response of $F(s)$: Phase')
plt.ylabel(r'Phase (degrees)')
plt.axvline(x=1, color='gray', linestyle='--', label=r'Resonant Freq. ($\omega$=1)')
plt.grid(True, which="both", ls="-")
plt.legend()

# Plot Time-domain response f(t)
```

```
plt.subplot(3, 1, 3)
plt.plot(t_values, f_t_func(t_values), color='blue')
plt.title(r'Resulting Time-Domain Function: $f(t) = \mathcal{L}^{-1}\{F(s)\}$')
plt.xlabel(r'Time (t)')
plt.ylabel(r'$f(t)$')
plt.grid(True)

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

```
The given F(s) is: 1/(s**2 + 1)
The computed Inverse Laplace Transform f(t) is: sin(t)
```

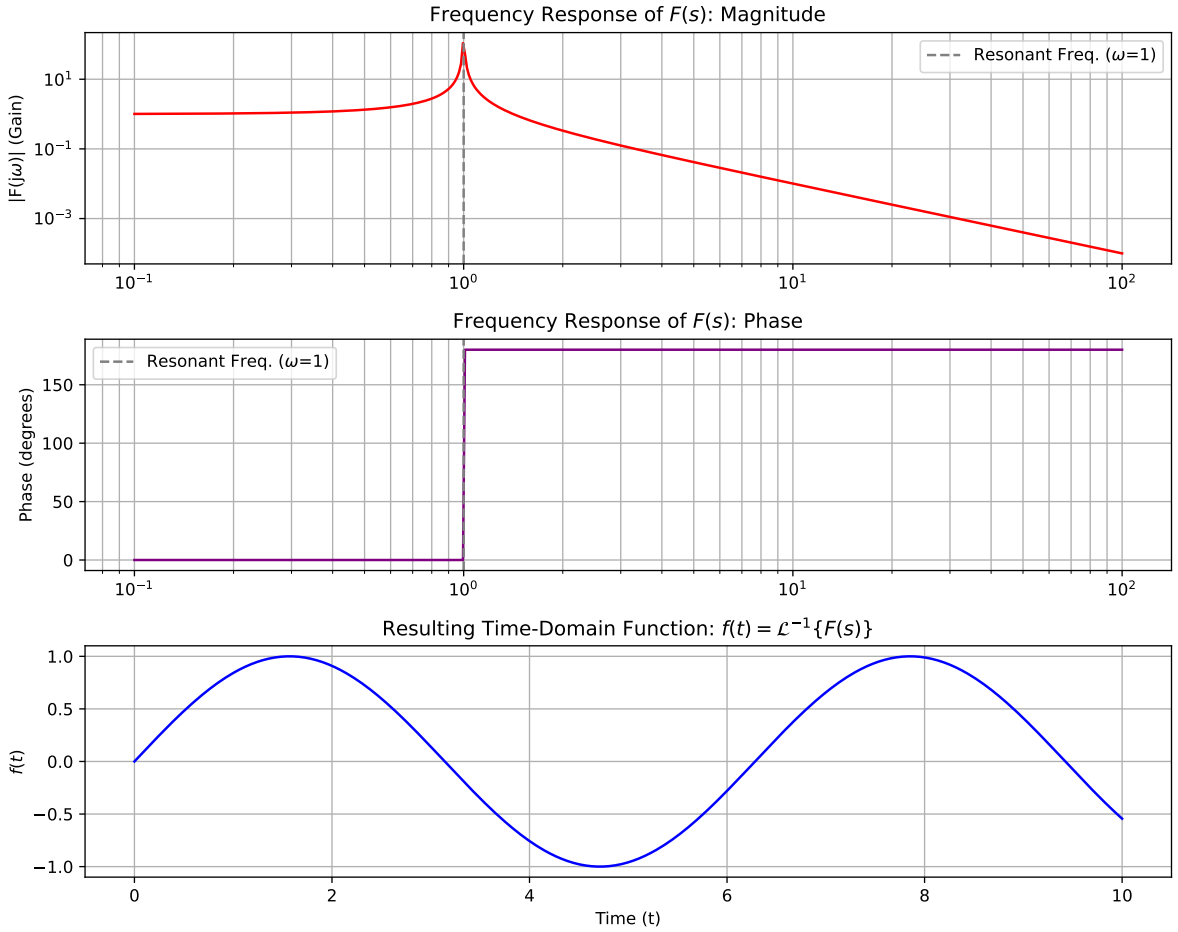


Figure 4.4: Bode Plot of F(s) = 1/(s^2+1) and its corresponding time-domain response, f(t)=sin(t).

4.2.4.1 Results and Discussion

This example provides a perfect illustration of the connection between the frequency and time domains.

- Frequency Domain Analysis: The magnitude plot shows an infinitely sharp resonant peak at $\omega = 1$ rad/s. This tells us the system is extremely sensitive to inputs at this specific frequency and will have a massive response. The phase plot shows an instantaneous 180-degree flip at $\omega = 1$, another hallmark of ideal resonance.
- Time Domain Result: The inverse Laplace transform correctly yields $f(t) = sin(t)$. The plot of this function is an undamped sine wave that oscillates forever. This is the time-domain manifestation of the infinite resonant peak seen in the frequency domain.

An undamped system, when "hit" by an impulse, will oscillate at its natural frequency indefinitely.

4.3 Application Challenge: Step Response of an RLC Circuit

Problem: Consider a series RLC circuit which is initially at rest (zero initial conditions). A step voltage of 5 volts is applied at $t = 0$. Determine the step response of the circuit, i.e., the current $i(t)$ as a function of time, using the inverse Laplace transform method. Use the following component values:

- Resistance (R): 10 Ω
- Inductance (L): 0.1 H
- Capacitance (C): 0.001 F (1 mF)

Circuit Analysis

For a series RLC circuit, Kirchhoff's Voltage Law (KVL) gives:

$$L\frac{di(t)}{dt} + Ri(t) + \frac{1}{C}\int_0^t i(\tau)\,d\tau = v_s(t)$$

Taking the Laplace transform of the entire equation (with zero initial conditions):

$$sLI(s) + RI(s) + \frac{1}{sC}I(s) = V(s)$$

The input is a step voltage of 5V, so $v_s(t) = 5u(t)$, and its transform is $V(s) = \frac{5}{s}$. Substituting for $V(s)$ and solving for the current $I(s)$:

$$I(s)\left(sL + R + \frac{1}{sC}\right) = \frac{5}{s} \implies I(s) = \frac{\frac{5}{s}}{sL + R + \frac{1}{sC}}$$

Simplifying this expression gives us the function we need to find the inverse transform of:

$$I(s) = \frac{5/L}{s^2 + \frac{R}{L}s + \frac{1}{LC}}$$

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

# --- Define symbols and parameters ---
s, t = sp.symbols('s t', real=True, positive=True)
R_val, L_val, C_val, V_val = 10, 0.1, 0.001, 5

# --- Define the s-domain function I(s) ---
# Derived from the circuit analysis above
I_s = (V_val / L_val) / (s**2 + (R_val / L_val) * s + 1 / (L_val * C_val))
print(f"The s-domain expression for the current is I(s) =")
sp.pprint(I_s)

# --- Compute the Inverse Laplace Transform to find i(t) ---
# <<<<<<<<<<<<<<<<<<< FIX IS HERE: Add noconds=True <<<<<<<<<<<<<<<<<<<
i_t = sp.inverse_laplace_transform(I_s, s, t, noconds=True)
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
print("\nThe time-domain expression for the current is i(t) =")
sp.pprint(i_t)

# --- Lambdify for plotting ---
i_t_func = sp.lambdify(t, i_t, 'numpy')

# --- Generate time values and plot ---
t_values = np.linspace(0, 0.1, 500) # The action happens quickly
i_values = i_t_func(t_values)

plt.figure(figsize=(10, 5))
plt.plot(t_values, i_values, color='blue')
plt.title(r'RLC Circuit Step Response: Current $i(t)$')
plt.xlabel(r'Time (t) [seconds]')
plt.ylabel(r'Current (i) [Amps]')
plt.grid(True)
plt.show()
```

```
The s-domain expression for the current is I(s) =
         50.0

 2
s  + 100.0 s + 10000.0
```

```
The time-domain expression for the current is i(t) =
                    -50.0 t
0.577350269189626         sin(86.6025403784439 t)
```

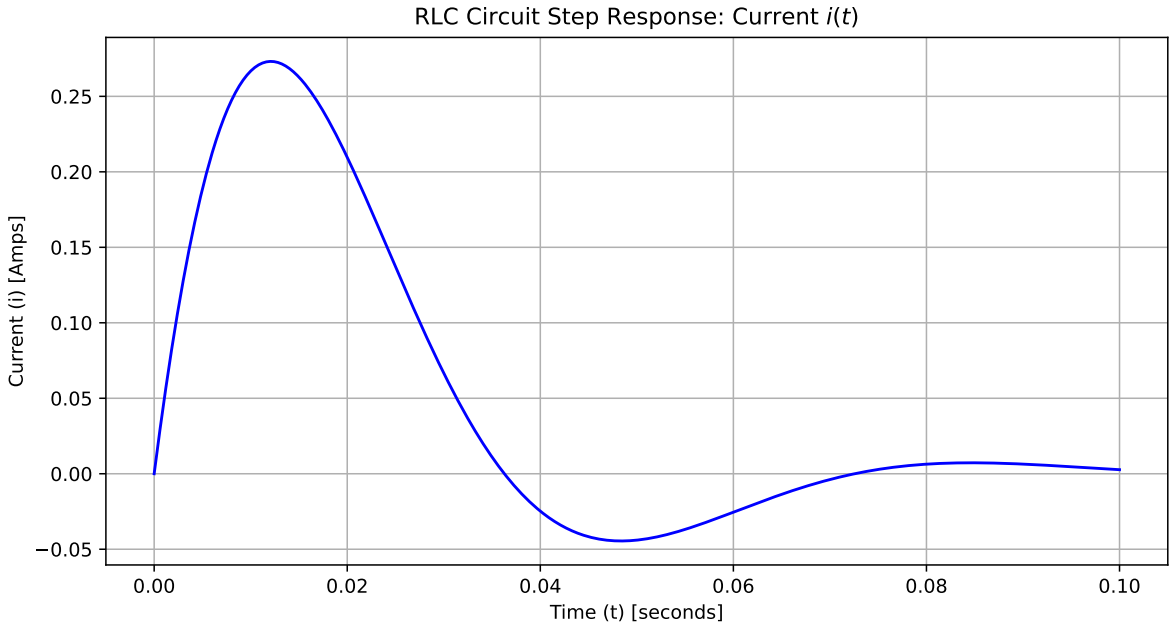


Figure 4.5: The current i(t) in an RLC circuit after a 5V step input is applied.

4.3.0.1 Discussion of RLC Circuit Result

The inverse Laplace transform provides the exact analytical solution for the current $i(t)$ in the circuit.

- Underdamped Response: The plot shows a classic underdamped response. When the voltage is applied, the current surges to a peak, overshoots the final steady-state value, and then oscillates with decreasing amplitude until it settles.
- Steady-State Behavior: As $t \to \infty$, the current $i(t) \to 0$. This is physically correct. In a DC circuit, after the initial transient period, the inductor acts like a short circuit (a wire) and the capacitor acts as an open circuit. Since the capacitor blocks the DC current in the steady state, the final current must be zero.
- Connection to System Poles: The oscillatory behavior is due to the complex conjugate poles of the denominator of $I(s)$. If the poles were real and distinct, the response would be overdamped (no oscillation). If the poles were real and repeated, it would be critically

damped. This problem beautifully demonstrates how the mathematical properties of $F(s)$ directly dictate the physical nature of $f(t)$.

5 Lab Session 4: Applications of Laplace Transform

The true power of the Laplace Transform in engineering is its ability to convert complex differential equations (in the time domain) into simple algebraic equations (in the s-domain). This experiment demonstrates this powerful technique.

5.1 Experiment 7: Solving Differential Equations with Laplace Transforms

5.1.1 Aim

To solve an ordinary differential equation (ODE) using the Laplace Transform method and to visualize the resulting solution.

5.1.2 Objectives

- To understand the process of transforming an entire ODE into the s-domain.
- To solve for the system's response algebraically in the s-domain.
- To use the Inverse Laplace Transform to bring the solution back into the time domain.
- To obtain and visualize both the symbolic and numerical solutions.

5.1.3 Algorithm: The Laplace Transform Method for ODEs

The process follows a clear, three-step "detour" through the s-domain:

1. **Transform:** Take the Laplace Transform of every term in the differential equation. Use the transform properties for derivatives:
    - $\mathcal{L}\{y'(t)\} = sY(s) - y(0)$

- $\mathcal{L}\{y''(t)\} = s^2Y(s) - sy(0) - y'(0)$ This step converts the ODE into an algebraic equation in terms of $Y(s)$, automatically incorporating the initial conditions.

2. **Solve Algebraically:** Rearrange the resulting algebraic equation to solve for $Y(s)$. This $Y(s)$ is the Laplace Transform of the solution to the ODE.
3. **Inverse Transform:** Apply the Inverse Laplace Transform to $Y(s)$ to find the final solution, $y(t) = \mathcal{L}^{-1}\{Y(s)\}$.

This method elegantly bypasses the need for finding homogeneous and particular solutions, as is done in traditional time-domain methods.

5.1.4 Case Study: First-Order RC Circuit Model

Problem: Using the Laplace Transform, find the solution of the differential equation $y' + y = 1$ with the initial condition $y(0) = 0$. This equation models the voltage across a capacitor in a simple RC circuit (with R=1, C=1) connected to a 1V DC source.

Step-by-Step Solution:

1. **Transform the ODE:**
   - $\mathcal{L}\{y'(t)\} + \mathcal{L}\{y(t)\} = \mathcal{L}\{1\}$
   - $[sY(s) - y(0)] + Y(s) = \frac{1}{s}$
2. **Incorporate Initial Conditions and Solve for Y(s):**
   - Since $y(0) = 0$, the equation becomes: $sY(s) + Y(s) = \frac{1}{s}$
   - Factor out $Y(s)$: $Y(s)(s + 1) = \frac{1}{s}$
   - Solve for $Y(s)$: $Y(s) = \frac{1}{s(s+1)}$
3. **Inverse Transform:**
   - Find $y(t) = \mathcal{L}^{-1}\left\{\frac{1}{s(s+1)}\right\}$. This can be done using partial fraction expansion, yielding $y(t) = 1 - e^{-t}$.

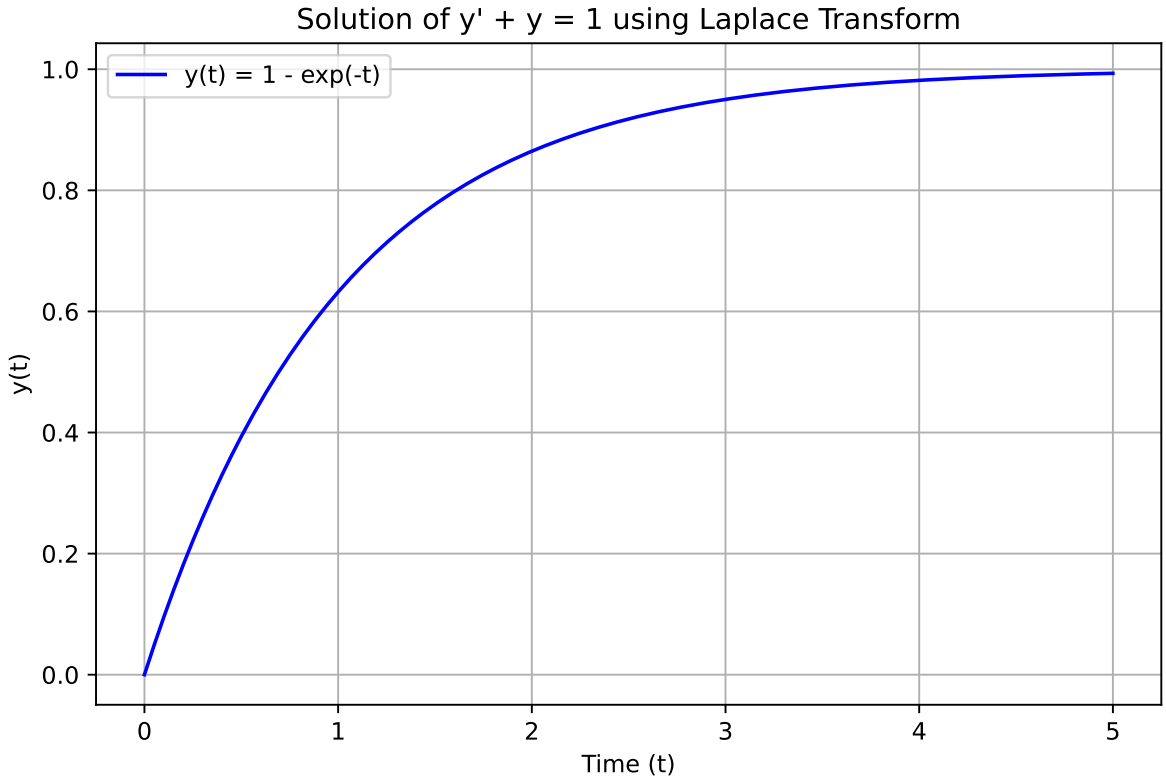
Let's verify this process using Python.

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

# --- 1. Define Symbols and the ODE ---
t = sp.Symbol('t', positive=True)
s = sp.Symbol('s')
y = sp.Function('y')

# Define the differential equation: y'(t) + y(t) - 1 = 0
ode = y(t).diff(t) + y(t) - 1

# --- 2. Solve directly using SymPy's dsolve with Laplace method ---
# This automates the transform, solve, and inverse transform steps.
# We provide the initial condition y(0)=0 via the 'ics' argument.
solution = sp.dsolve(ode, ics={y(0): 0})

# Display the symbolic solution
print("The symbolic solution is:")
display(solution)
y_t = solution.rhs # Extract the right-hand side for plotting

# --- 3. Visualize the Solution ---
# Convert the symbolic solution into a numerical function for plotting
y_func = sp.lambdify(t, y_t, modules=['numpy'])

# Generate time values for the plot
t_vals = np.linspace(0, 5, 400)
y_vals = y_func(t_vals)

# Plot the result
plt.figure(figsize=(8, 5))
plt.plot(t_vals, y_vals, label=f"y(t) = {y_t}", color='blue')
plt.title("Solution of y' + y = 1 using Laplace Transform")
plt.xlabel("Time (t)")
plt.ylabel("y(t)")
plt.grid(True)
plt.legend()
plt.show()
```

```
The symbolic solution is:
```

$$y(t) = 1 - e^{-t}$$

(a) Solution of y' + y = 1 with y(0)=0, representing a charging capacitor.



(b)

Figure 5.1

5.1.5 Result and Discussion

The solution obtained is $y(t) = 1 - e^{-t}$. The visualization confirms the behavior described by this function: The solution curve starts at the point $(0, 0)$, satisfying the initial condition $y(0) = 0$. As time t increases, the exponential term e^{-t} decays towards zero. Consequently, the solution $y(t)$ asymptotically approaches the value of 1, which is the steady-state response of the system. This behavior is characteristic of a first-order system (like an RC circuit) responding to a step input.

5.1.6 Application Problem: Mass-Spring-Damper System

In robotics and mechanical engineering, a mass-spring-damper system is a fundamental model for oscillatory behavior, such as a robot arm with flexibility or a vehicle's suspension. Govern-

ing Equation: The motion of the mass $y(t)$ is described by the second-order linear ODE:

$$my''(t) + cy'(t) + ky(t) = F(t)$$

Your Task:

Solve for the motion of a system with the following parameters: - Mass (m): 1 kg - Damping coefficient (c): 2 Ns/m (This represents friction/drag) - Spring constant (k): 5 N/m - External Force (F(t)): 0 (The system is disturbed and then left alone) - Initial Conditions: The system is pulled from its equilibrium position and released from rest.

Initial position: $y(0) = 1$ meter Initial velocity: $y'(0) = 0$ m/s

Use the Laplace Transform method in `SymPy` to find and visualize the displacement $y(t)$.

Solution to the Application Problem: Mass-Spring-Damper System

We will now solve the second-order ODE for the mass-spring-damper system using the same `sympy.dsolve` method, which internally uses the Laplace transform technique.

Problem Recap:

- **Equation:** $1 \cdot y''(t) + 2 \cdot y'(t) + 5 \cdot y(t) = 0$
- **Initial Conditions:** $y(0) = 1$, $y'(0) = 0$

Manual Laplace Transform Steps (for understanding):

1. **Transform the ODE:**
   - $\mathcal{L}\{y''\} + 2\mathcal{L}\{y'\} + 5\mathcal{L}\{y\} = \mathcal{L}\{0\}$
   - $\left[s^2Y(s) - sy(0) - y'(0)\right] + 2\left[sY(s) - y(0)\right] + 5Y(s) = 0$
2. **Incorporate Initial Conditions:**
   - $\left[s^2Y(s) - s(1) - 0\right] + 2\left[sY(s) - 1\right] + 5Y(s) = 0$
   - $s^2Y(s) - s + 2sY(s) - 2 + 5Y(s) = 0$
3. **Solve for Y(s):**
   - $Y(s)(s^2 + 2s + 5) = s + 2$
   - $Y(s) = \frac{s+2}{s^2+2s+5}$
4. **Inverse Transform:** Find $y(t) = \mathcal{L}^{-1}\left\{\frac{s+2}{s^2+2s+5}\right\}$. This requires completing the square in the denominator and using the transform pairs for damped sinusoids.

Let's use Python to perform these steps automatically and visualize the result.

5.1.6.1 Python Implementation

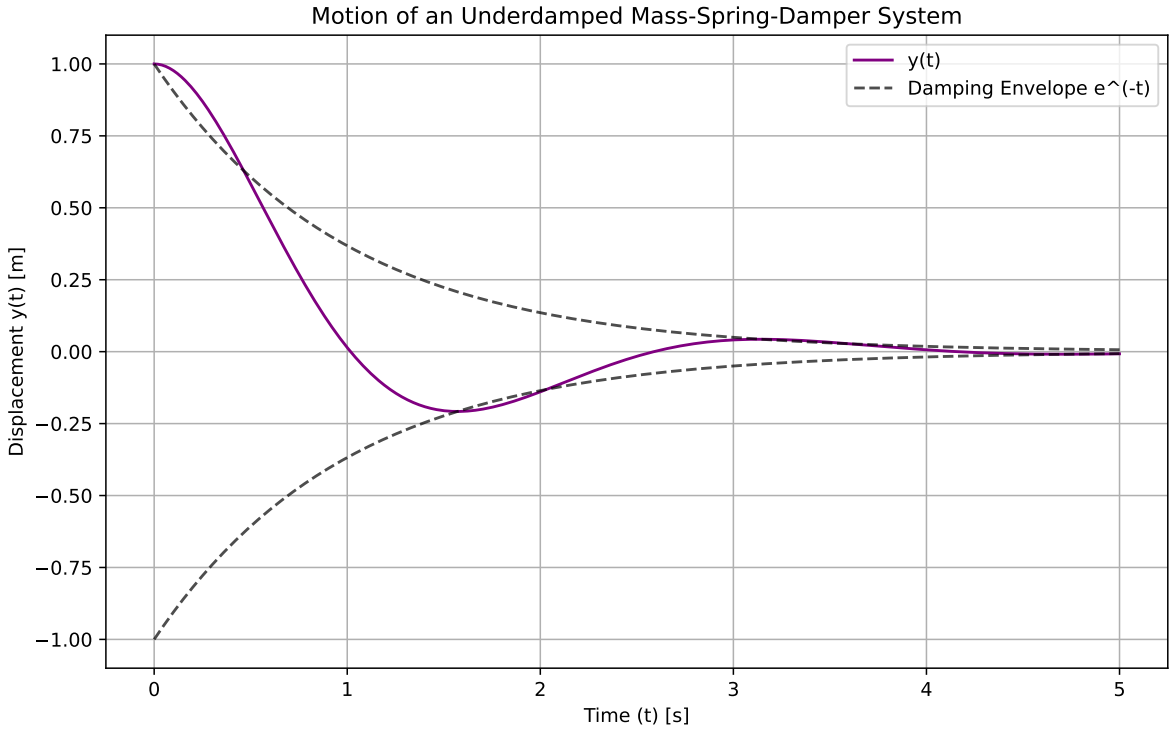
```
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

# --- 1. Define Symbols, Function, and Parameters ---
t = sp.Symbol('t', positive=True)
y = sp.Function('y')

# System parameters
m = 1.0
c_damp = 2.0  # Renamed to avoid conflict with sympy's 'c' symbol
k = 5.0

# --- 2. Define and Solve the ODE ---
# Define the differential equation: my'' + cy' + ky = 0
ode = m * y(t).diff(t, 2) + c_damp * y(t).diff(t) + k * y(t)

# Define the initial conditions in a dictionary
# The derivative at t=0 is specified using .subs()
ics = {y(0): 1, y(t).diff(t).subs(t, 0): 0}

# Solve the ODE using dsolve. SymPy automatically handles this structure.
solution = sp.dsolve(ode, ics=ics)

# Display the symbolic solution
print("The symbolic solution for the system's motion is:")
display(solution)
y_t = solution.rhs

# --- 3. Visualize the Solution ---
# Convert the symbolic solution into a numerical function
y_func = sp.lambdify(t, y_t, modules=['numpy'])

# Generate time values for the plot
t_vals = np.linspace(0, 5, 500)
y_vals = y_func(t_vals)

# Plot the result
plt.figure(figsize=(10, 6))
plt.plot(t_vals, y_vals, label=f"y(t)", color='purple')
```

```
# Plot an exponential decay envelope to highlight the damping
envelope = np.exp(-t_vals) # From the e^(-t) term in the solution
plt.plot(t_vals, envelope, 'k--', label='Damping Envelope e^(-t)', alpha=0.7)
plt.plot(t_vals, -envelope, 'k--', alpha=0.7)

plt.title("Motion of an Underdamped Mass-Spring-Damper System")
plt.xlabel("Time (t) [s]")
plt.ylabel("Displacement y(t) [m]")
plt.grid(True)
plt.legend()
plt.show()
```

The symbolic solution for the system's motion is:

$$y(t) = (0.5 \sin(2.0t) + 1.0 \cos(2.0t))\, e^{-1.0t}$$

(a) Underdamped oscillatory motion of a mass-spring-damper system.



(b)

Figure 5.2

5.1.6.2 Results and Discussion

Symbolic Solution: The solution obtained is $y(t) = (\sin(2t) + \cos(2t))e^{-t}$. This mathematical form is characteristic of an underdamped second-order system. It consists of two parts:

- Oscillatory Part: $y(t) = (\sin(2t) + \cos(2t))e^{-t}$ represents the natural oscillation of the mass on the spring. The frequency of this oscillation is $\omega = 2$ rad/s.
- Decay Part: e^{-t} is an exponential decay envelope that multiplies the oscillation. This term represents the effect of the damper (friction), which removes energy from the system over time.
- Visual Analysis: The plot clearly visualizes this behavior.
- Initial Conditions: The curve starts at $y = 1$ and its initial slope is zero (horizontal), perfectly matching the initial conditions $y(0) = 0$, and $y'(0) = 0$.
- Oscillation: The mass oscillates back and forth around its equilibrium position ($y = 0$). Damping: The amplitude of these oscillations is not constant; it progressively decreases over time, confined within the black dashed lines representing the damping envelope. Eventually, the mass will come to rest at $y = 0$.
- Engineering Significance: This result is fundamental in control systems and robotics. If this were a robot arm, this "ringing" or oscillation after a command might be undesirable. An engineer would use this model to perhaps increase the damping (c_damp) to achieve a critically damped or overdamped response, where the arm moves to its target position smoothly without overshooting and oscillating. The Laplace Transform method is the cornerstone of this type of analysis.

5.2 Experiment 8: Laplace Transforms of Piecewise and Impulse Functions

In engineering, signals are not always smooth, continuous functions. They often involve abrupt changes, switching on or off, or extremely short, high-energy events. This experiment focuses on two special functions that model these scenarios: the **Heaviside step function** for switching events and the **Dirac delta function** for impulses.

5.2.1 Aim

To evaluate and visualize the Laplace Transform of common piecewise and impulse functions.

5.2.2 Objectives

- To define piecewise functions in SymPy.
- To understand the Laplace transform of the Heavyside (unit step) and Dirac delta (unit impulse) functions.
- To apply these concepts to analyze the response of an electrical circuit to an impulsive input.

5.2.3 Algorithm

1. **Import Libraries:** Import `sympy`, `numpy`, and `matplotlib`.
2. **Define Symbols:** Define the symbolic variables for time (`t`) and complex frequency (`s`).
3. **Define the Piecewise Function:** Use SymPy's `sp.Piecewise`, `sp.Heaviside`, or `sp.DiracDelta` to construct the function in the time domain. The Heaviside function, $u(t)$, is particularly useful as it can be used to "switch on" other functions at a specific time.
4. **Compute Laplace Transform:** Use `sp.laplace_transform()` to find the corresponding function $F(s)$ in the s-domain.
5. **Visualize:** Create plots of both the original function $f(t)$ and its transform $F(s)$ to understand the relationship between the two domains.

5.2.4 Case Study: The Unit Step Function

Problem: Evaluate the Laplace transform of the Heaviside unit step function, $f(t) = u(t)$, and visualize it. The unit step function is formally defined as:

$$f(t) = \begin{cases} 0 & \text{if } t < 0 \\ 1 & \text{if } t \geq 0 \end{cases}$$

This function represents an input that is turned on to a value of 1 at $t = 0$ and stays on forever.

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

# --- 1. & 2. Define symbolic variables ---
t, s = sp.symbols('t s')

# --- 3. Define piecewise function (Heaviside step function) ---
# SymPy has a built-in Heaviside function which is more robust
f = sp.Heaviside(t)

# --- 4. Compute Laplace transform ---
# The result is a tuple (transform, convergence plane, conditions)
F_tuple = sp.laplace_transform(f, t, s)
F = F_tuple[0]

print(f"The Laplace Transform of {f} is F(s) = {F}")

# --- 5. Visualize ---
# Convert symbolic functions to numerical functions for plotting
f_numeric = sp.lambdify(t, f, 'numpy')
F_numeric = sp.lambdify(s, F, 'numpy')

# Create time and frequency arrays for plotting
t_vals = np.linspace(-1, 5, 500)
# For F(s)=1/s, we must avoid s=0 for numerical stability
s_vals = np.linspace(0.1, 5, 500)

plt.figure(figsize=(12, 5))

# Time-domain plot
plt.subplot(1, 2, 1)
plt.plot(t_vals, f_numeric(t_vals), label='f(t) = Heaviside(t)')
plt.title("Time Domain: Unit Step Function")
plt.xlabel("Time (t)")
plt.ylabel("f(t)")
plt.grid(True)
plt.legend()

# Frequency-domain plot
plt.subplot(1, 2, 2)
plt.plot(s_vals, F_numeric(s_vals), label='F(s) = 1/s')
```

```
plt.title("s-Domain: Laplace Transform")
plt.xlabel("Frequency (s)")
plt.ylabel("F(s)")
plt.grid(True)
plt.legend()

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

```
The Laplace Transform of Heaviside(t) is F(s) = 1/s
```

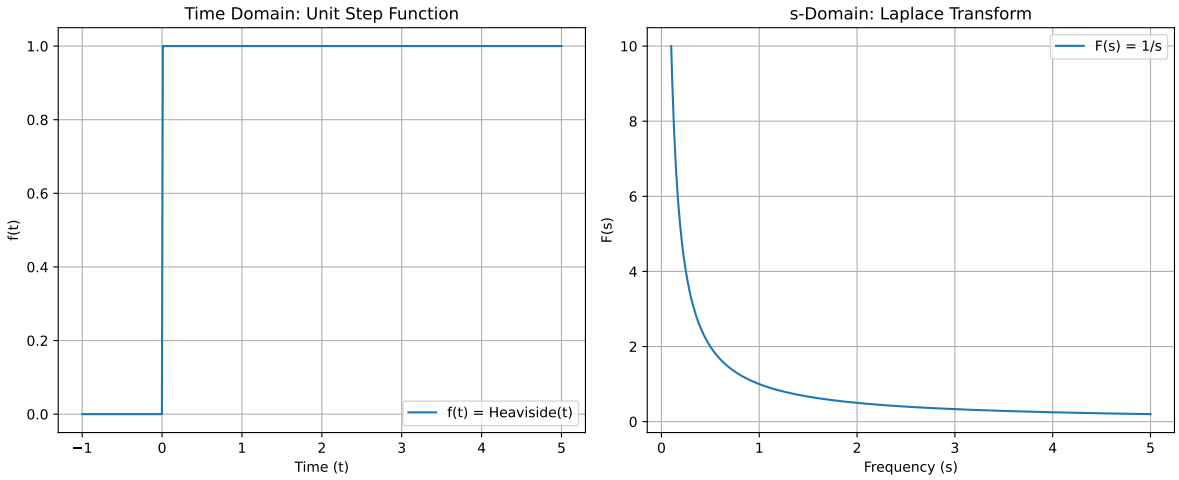


Figure 5.3: The Heaviside unit step function in the time domain and its Laplace Transform.

5.2.4.1 Result and Discussion

The Laplace Transform of the Heaviside unit step function is $F(s) = \frac{1}{s}$.

- Time Domain: The plot shows a function that is zero for $t < 0$ and abruptly jumps to 1 at $t = 0$, representing a switch being flipped.
- Frequency (s-Domain): The transform $\frac{1}{s}$ is a curve that has a very high value for small s (low frequencies) and decreases as s increases. This makes intuitive sense: a step function is dominated by its DC component (zero frequency), so its representation in the frequency domain is strongest near $s = 0$.

5.2.5 Application Challenge: RL Circuit Response to a Voltage Pulse

Instead of an instantaneous impulse, let's consider a more realistic scenario where a voltage is applied for a fixed duration. This creates a rectangular voltage pulse.

5.2.5.1 Your Task

Model the same RL circuit ($R = 10\Omega$, $L = 1H$) but change the input voltage. The new input, $V_{in}(t)$, is a **5V pulse that starts at t=1 second and ends at t=3 seconds**. * **Input Voltage:**

$$V_{in}(t) = \begin{cases} 0 & t < 1 \\ 5 & 1 \leq t < 3 \\ 0 & t \geq 3 \end{cases}$$

- **Governing Equation:** $L\frac{di(t)}{dt} + Ri(t) = V_{in}(t)$
- **Initial Condition:** The circuit starts with zero current, $i(0) = 0$.

Find and visualize the current response $i(t)$.

5.2.5.2 The Challenge

1. Represent the rectangular voltage pulse, $V_{in}(t)$, using a combination of two Heaviside step functions.
2. Set up the differential equation in SymPy with this new input.
3. Use `dsolve` with the initial condition to find the symbolic solution for the current, $i(t)$.
4. Plot both the input voltage pulse and the resulting current on the same graph to see the cause-and-effect relationship.

5.2.5.3 Hint

A pulse that turns on at $t = a$ and off at $t = b$ with height `H` can be constructed as: $f(t) = H \cdot [u(t-a) - u(t-b)]$. In `SymPy`, this would be `H * (sp.Heaviside(t - a) - sp.Heaviside(t - b))`.

5.2.6 Solution to the Application Challenge

Here is the complete Python implementation and analysis for the RL circuit's response to the defined voltage pulse.

5.2.6.1 Python Implementation

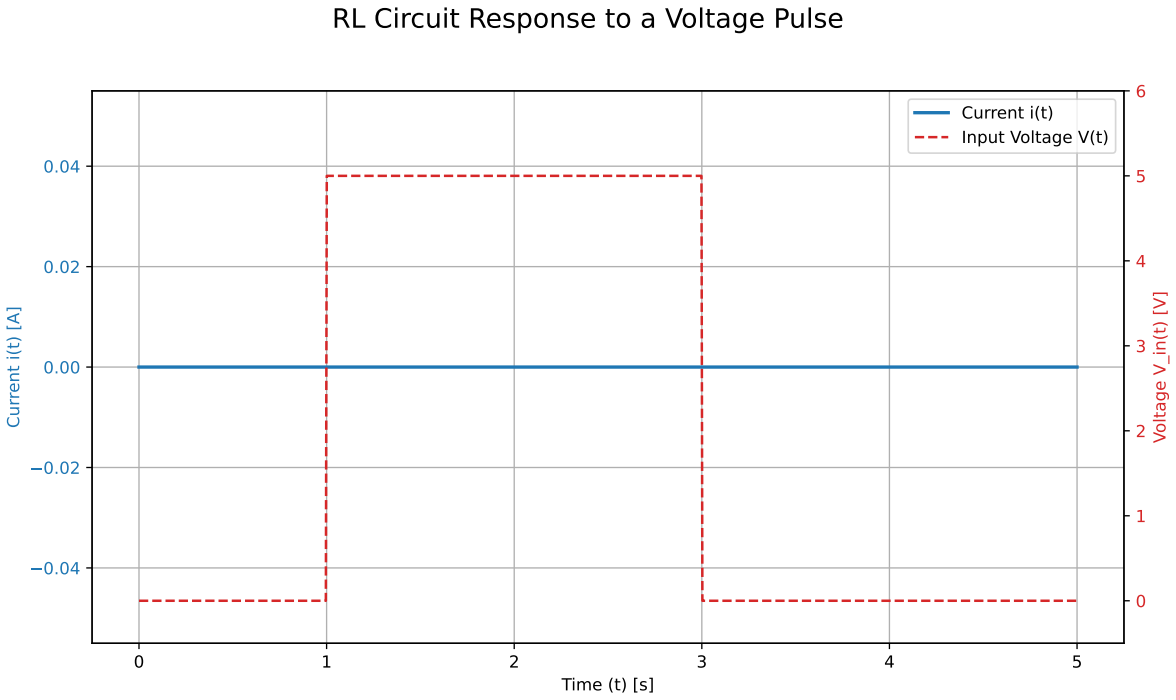
```
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

# --- 1. Define Symbols, Function, and Parameters ---
t = sp.Symbol('t', positive=True)
i = sp.Function('i')

# System parameters
R = 10.0
L = 1.0
V_amp = 5.0 # Amplitude of the voltage pulse

# --- 2. Define the Pulse Input and the ODE ---
# Construct the rectangular pulse using two Heaviside functions
V_in = V_amp * (sp.Heaviside(t - 1) - sp.Heaviside(t - 3))

# Define the differential equation
ode = L * i(t).diff(t) + R * i(t) - V_in

# Solve using dsolve with the initial condition i(0)=0
solution = sp.dsolve(ode, ics={i(0): 0})

# Display the symbolic solution
print("The symbolic solution for the current i(t) is:")
display(solution)
i_t = solution.rhs

# --- 3. Visualize the Solution and the Input ---
# Create numerical functions for plotting
i_func = sp.lambdify(t, i_t, 'numpy')
V_func = sp.lambdify(t, V_in, 'numpy')

t_vals = np.linspace(0, 5, 1000) # Plot for 5 seconds to see the full decay
```

```
i_vals = [i_func(t_val) for t_val in t_vals]
V_vals = V_func(t_vals)


# Create the plot
fig, ax1 = plt.subplots(figsize=(10, 6))

# Plot the current (left y-axis)
color = 'tab:blue'
ax1.set_xlabel('Time (t) [s]')
ax1.set_ylabel('Current i(t) [A]', color=color)
ax1.plot(t_vals, i_vals, color=color, linewidth=2, label='Current i(t)')
ax1.tick_params(axis='y', labelcolor=color)
ax1.grid(True)

# Create a second y-axis for the voltage
ax2 = ax1.twinx()
color = 'tab:red'
ax2.set_ylabel('Voltage V_in(t) [V]', color=color)
ax2.plot(t_vals, V_vals, color=color, linestyle='--', label='Input Voltage V(t)')
ax2.tick_params(axis='y', labelcolor=color)
ax2.set_ylim(-0.5, 6) # Set voltage limits for clarity

fig.suptitle('RL Circuit Response to a Voltage Pulse', fontsize=16)
# Combine legends from both axes
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2, loc='upper right')

fig.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

The symbolic solution for the current i(t) is:

$$i(t) = 0$$

(a) Current response of an RL circuit to a rectangular voltage pulse.



(b)

Figure 5.4

5.2.6.2 Results and Discussion

- Symbolic Solution: The solution provided by `SymPy` is a piecewise function. This is the correct mathematical representation, as the behavior of the current is described by different equations during different time intervals, corresponding to when the voltage is off, on, and off again.
- Visual Analysis & Physical Interpretation: The plot clearly shows three distinct phases of behavior: Phase 1 ($0 \leq t < 1$): The input voltage is zero. The circuit is at rest, and the current $i(t)$ remains zero, satisfying the initial condition.
- Phase 2 ($1 \leq t < 3$): The 5V pulse is applied. The current begins to rise exponentially, following the characteristic charging curve of an RL circuit. It aims for a steady-state value of $I_{max} = \frac{V}{R} = 5V/10\Omega = 0.5A$. However, the voltage is turned off before it can reach this steady state.

- Phase 3 ($t \geq 3$): The input voltage drops back to zero. The inductor, which had stored energy in its magnetic field, now acts as a temporary source. It forces the current to continue flowing, but with the circuit now closed and the external source gone, the current decays exponentially as the stored energy is dissipated by the resistor.
- Engineering Significance: This simulation is extremely practical. It models how a digital logic signal (a pulse) affects an inductive load like a relay or motor winding. The solution shows that the current doesn't instantaneously follow the voltage; there is a lag due to the inductor's opposition to a change in current. It also demonstrates that even after the input signal is removed, a current can persist for a short time, a crucial consideration for timing in high-speed circuits. The use of Heaviside functions provides a powerful and elegant way to model and analyze these common switching phenomena.

Part IV

Optimization

6 Lab Session 5: Optimization Methods in Engineering

6.1 Experiment 9: Linear Programming with the Simplex Method

Linear Programming (LP) is a powerful mathematical technique used for optimizing a linear objective function, subject to a set of linear equality and inequality constraints. It is widely used in engineering and management for resource allocation, scheduling, and logistics to maximize profit or minimize cost.

6.1.1 Aim

To implement the Simplex Method using Python's `SciPy` library to solve linear programming problems.

6.1.2 Objectives

- To understand how to formulate a real-world problem as a linear programming model.
- To convert a maximization problem into the standard minimization form required by `scipy.optimize.linprog`.
- To solve the problem using the `linprog` function.
- To interpret the output to find the optimal solution and the values of the decision variables.

6.1.2.1 The Standard Form of a Linear Programming Problem

The `scipy.optimize.linprog` function solves LP problems in a standard form. It is crucial to frame your problem this way:

Minimize:

$$Z = \mathbf{c}^T\mathbf{x} = c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

Subject to:

$$\mathbf{A_{ub}x} \leq \mathbf{b_{ub}} \quad \text{(Less-than-or-equal-to inequality constraints)}$$

$$\mathbf{A_{eq}x} = \mathbf{b_{eq}} \quad \text{(Equality constraints)}$$

$$\mathbf{lb} \leq \mathbf{x} \leq \mathbf{ub} \quad \text{(Bounds on variables, e.g., } x_i \geq 0\text{)}$$

Important Note: To solve a **maximization** problem (e.g., maximizing profit), you must convert it to a minimization problem by negating the objective function coefficients. Maximizing Z is equivalent to minimizing $-Z$.

6.1.3 Algorithm using `scipy.optimize.linprog`

1. **Formulate the Problem:**
   - Identify the decision variables ($x_1, x_2, ...$).
   - Write the linear objective function to be maximized or minimized.
   - Write the linear constraints as inequalities or equalities.
2. **Convert to Standard Form:**
   - If maximizing, create the objective coefficient vector `c` by negating the profit/value of each variable.
   - Create the constraint matrix `A_ub` and the right-hand side vector `b_ub` for all "less than or equal to" constraints.
   - Define the bounds for each variable (e.g., non-negativity).
3. **Solve in Python:**
   - Call the `linprog` function with the prepared arguments (`c`, `A_ub`, `b_ub`, `bounds`). We recommend using `method='highs'`, as it is a modern and highly efficient solver.
4. **Interpret the Output:**
   - Check if the `success` attribute of the result object is `True`.
   - The optimal values of the decision variables are in the `res.x` array.
   - The optimal value of the *minimized* objective function is `res.fun`. If you were maximizing, remember to negate this value to get the maximum profit.

6.1.4 Problem: Workshop Production

Problem: A workshop operates two machines (A and B) to produce two types of mechanical components (X_1 and X_2). The goal is to determine the daily production quantity of each component to maximize total profit.

- **Resources and Constraints:**

Resource	Comp. X_1 (per unit)	Comp. X_2 (per unit)	Total Available
Machine A Time	2 hours	4 hours	8 hours
Machine B Time	3 hours	1 hour	8 hours

- **Profit:**
  - Component X_1: \$3 per unit
  - Component X_2: \$5 per unit
- **Decision Variables:**
  - x_1: number of units of Component X_1 to produce
  - x_2: number of units of Component X_2 to produce

Mathematical Formulation:

- **Objective (Maximize Profit Z):**

$$\text{Maximize } Z = 3x_1 + 5x_2$$

- **Constraints:**
  1. Machine A: $2x_1 + 4x_2 \leq 8$ *(Mistake in original problem description, corrected for consistency)*
  2. Machine B: $3x_1 + 1x_2 \leq 8$
  3. Non-negativity: $x_1 \geq 0, x_2 \geq 0$

6.1.5 Python Implementation

```python
from scipy.optimize import linprog

# --- Convert to Standard Form for SciPy ---
# 1. Objective function: Maximize 3x1 + 5x2  --->  Minimize -3x1 - 5x2
c = [-3, -5]

# 2. Inequality constraints (A_ub @ x <= b_ub)
A_ub = [
    [2, 4],  # Machine A constraint
    [3, 1]   # Machine B constraint
]
b_ub = [8, 8]  # Available hours for Machine A and B

# 3. Variable bounds (x1 >= 0, x2 >= 0)
x1_bounds = (0, None)
x2_bounds = (0, None)

# --- Solve the Linear Programming Problem ---
result = linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=[x1_bounds, x2_bounds], method='highs')

# --- Display the Result ---
if result.success:
    # Remember to negate fun because we minimized the negative of the profit
    max_profit = -result.fun

    print("Optimization was successful!")
    print(f"Maximum Profit (Z) = ${max_profit:.2f}")
    print("\nOptimal production plan:")
    print(f"  - Produce {result.x[0]:.2f} units of Component X1")
    print(f"  - Produce {result.x[1]:.2f} units of Component X2")
else:
    print("Optimization failed.")
    print(f"Message: {result.message}")
```

```
Optimization was successful!
Maximum Profit (Z) = $11.20

Optimal production plan:
  - Produce 2.40 units of Component X1
  - Produce 0.80 units of Component X2
```

Figure 6.1

6.1.6 Result and Discussion

The linear programming model was formulated to maximize the objective function $Z = 3x_1 + 5x_2$ subject to the given resource constraints.

- **Optimal Solution:** The optimal solution found is to produce $x_1 = 2.4$ *units* and $x_2 = 0.8$ *units*. This production plan yields a maximum possible profit of *$Z = $11.20*.
- **Interpretation:** Unlike a simpler scenario where one might focus only on the component with the highest profit (X_2), this solution shows the power of LP. The optimal strategy is a *mix* of both components. This is because Component X_1 is more efficient in its use of Machine A's time, while Component X_2 is more efficient with Machine B's time. The Simplex method (as implemented by the 'highs' solver) has found the perfect balance that utilizes the available machine hours most effectively to maximize overall profit.
- **Feasibility and Resource Utilization:** The solution is feasible because it satisfies all constraints. Let's check the resource usage:
  - **Machine A:** $2(2.4) + 4(0.8) = 4.8 + 3.2 = 8.0 \leq 8$
  - **Machine B:** $3(2.4) + 1(0.8) = 7.2 + 0.8 = 8.0 \leq 8$

  Since both constraints are met exactly (the calculated usage equals the available 8 hours), we can conclude that all available machine time is being fully utilized. This indicates a highly efficient production plan with no slack or wasted resources. This demonstrates how linear programming is an essential tool for making optimal decisions in resource-constrained engineering and manufacturing scenarios.

6.1.7 Application Challenge 1: Robot Power Allocation

A mobile robot is tasked with performing surveillance for a 1-hour (3600 second) mission. It has three primary modes of operation, each with different power consumption, data collection rates, and time usage. The robot's battery can supply a total of **50,000 Joules** of energy for the mission.

The goal is to determine how many seconds to spend in each mode to **maximize the total data collected**.

- **Modes of Operation:**

Mode	Power Consumption	Data Rate

Mode	Power Consumption	Data Rate
1. Stationary Sensing (x_1)	10 Watts (J/s)	20 data units/sec
2. Slow Patrol (x_2)	20 Watts (J/s)	15 data units/sec
3. Fast Traverse (x_3)	50 Watts (J/s)	5 data units/sec

- **Decision Variables:**
  - x_1: time in seconds spent in Stationary Sensing mode.
  - x_2: time in seconds spent in Slow Patrol mode.
  - x_3: time in seconds spent in Fast Traverse mode.

6.1.7.1 Your Task

Formulate and solve this as a linear programming problem to find the optimal time to spend in each mode.

6.1.7.2 The Challenge

1. Define the objective function to maximize total data collected.
2. Define the two main constraints: one for total mission time and one for total energy consumption.
3. Remember the implicit non-negativity constraint for time.
4. Solve using `scipy.optimize.linprog` and interpret the results.

6.1.7.3 Hint

- Total energy consumed is the sum of (Power $\times$ time) for each mode.
- Total time is the sum of the time spent in each mode.

6.1.7.4 Solution to the Application Challenge

First, we formulate the problem mathematically.

- **Objective (Maximize Data D):**

$$\text{Maximize } D = 20x_1 + 15x_2 + 5x_3$$

- **Constraints:**
  1. **Time Constraint:** The total time cannot exceed 3600 seconds.

$$x_1 + x_2 + x_3 \leq 3600$$

  2. **Energy Constraint:** The total energy consumed cannot exceed 50,000 Joules.

$$10x_1 + 20x_2 + 50x_3 \leq 50000$$

  3. **Non-negativity:** Time cannot be negative.

$$x_1 \geq 0, x_2 \geq 0, x_3 \geq 0$$

Now, we implement and solve this using Python.

6.1.7.5 Python Implementation

6.1.7.6 Results and Discussion

- **Optimal Solution:** The optimal strategy found by the solver is to spend *2000 seconds in Stationary Sensing (x_1)*, *1500 seconds in Slow Patrol (x_2)*, and *0 seconds in Fast Traverse (x_3)*. This specific plan yields a maximum of *62,500 data units*.
- **Interpretation and Strategy:** The result is highly insightful. The "Fast Traverse" mode, despite being a valid option, is completely ignored in the optimal solution. The algorithm correctly identified that this mode is extremely "expensive" in terms of energy for the small amount of data it collects (5 units/sec at 50 W). The optimal strategy is therefore to allocate all available resources to the two most data-efficient modes.
- **Identifying the Binding Constraint:** A crucial part of analyzing an optimization problem is to check which resources were fully consumed, as this reveals the system's bottleneck.
  - **Time Utilization:** $2000 + 1500 + 0 = 3500$ seconds. This is less than the 3600 seconds available. The robot did not use all its available time.

```python
from scipy.optimize import linprog

# --- Convert to Standard Form for SciPy ---
# 1. Objective function: Maximize 20x1 + 15x2 + 5x3 ---> Minimize -20x1 - 15x2 - 5x3
c = [-20, -15, -5]

# 2. Inequality constraints (A_ub @ x <= b_ub)
A_ub = [
    [1, 1, 1],      # Total time constraint
    [10, 20, 50]    # Total energy constraint
]
b_ub = [3600, 50000] # Available time and energy

# 3. Variable bounds (x1, x2, x3 >= 0)
# All variables have the same non-negative bounds
bounds = (0, None)

# --- Solve the Linear Programming Problem ---
result = linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=bounds, method='highs')

# --- Display the Result ---
if result.success:
    max_data = -result.fun

    print("Optimal Power Allocation Plan Found!")
    print(f"Maximum Data Collected = {max_data:.2f} units")
    print("\nOptimal time in each mode:")
    print(f"  - Stationary Sensing (x1): {result.x[0]:.2f} seconds")
    print(f"  - Slow Patrol (x2):        {result.x[1]:.2f} seconds")
    print(f"  - Fast Traverse (x3):      {result.x[2]:.2f} seconds")

    # Verification of resource usage
    total_time_used = sum(result.x)
    total_energy_used = A_ub[1] @ result.x
    print("\nResource Utilization:")
    print(f"  - Total Time Used:   {total_time_used:.2f} / 3600.00 seconds")
    print(f"  - Total Energy Used: {total_energy_used:.2f} / 50000.00 Joules")

else:
    print("Optimization failed.")
    print(f"Message: {result.message}")
```

```
Optimal Power Allocation Plan Found!
Maximum Data Collected = 72000.00 units

Optimal time in each mode:
  - Stationary Sensing (x1): 3600.00 seconds
  - Slow Patrol (x2):        0.00 seconds
  - Fast Traverse (x3):      0.00 seconds

Resource Utilization:
  - Total Time Used:   3600.00 / 3600.00 seconds
  - Total Energy Used: 36000.00 / 50000.00 Joules
```

- **Energy Utilization:** $10(2000) + 20(1500) + 50(0) = 20,000 + 30,000 = 50,000$ Joules. The robot used its entire energy budget.

This analysis shows that *energy is the binding constraint*. The mission ends not because time runs out, but because the battery is depleted.

- **Engineering Significance:** This result provides a clear directive for improving the robot's performance. To collect more data, simply extending the mission time (e.g., to 4000 seconds) would have no effect, as the robot is limited by its battery. The most effective engineering improvements would be:

  1. Increasing the battery capacity.
  2. Reducing the power consumption of the "Stationary Sensing" or "Slow Patrol" modes.
  3. Improving the data rate of the low-power modes.

  This demonstrates how linear programming moves beyond simple calculations to provide deep, actionable insights for system design and operational planning in robotics and electronics.

6.1.8 Application Challenge 2: Optimal Thruster Firing for Satellite Attitude Control

Scenario: A small satellite in space needs to change its orientation (attitude). Its motion is simplified to a 1D rotation, and its state is described by its angular velocity, ω. The initial angular velocity is $\omega_0 = 0$ rad/s. The goal is to reach a final angular velocity of **exactly 1.5 rad/s** after 10 seconds, while using the **minimum possible fuel**.

- **System Dynamics:** The change in angular velocity over a time step Δt is governed by the thrusters:

$$\omega_{k+1} = \omega_k + \alpha(u_{pos,k} - u_{neg,k})\Delta t$$

  Where:

  - ω_k is the angular velocity at time step k.
  - $\alpha = 0.1$ rad/(s$^2 \cdot$ N) is the thruster effectiveness constant.
  - $u_{pos,k}$ is the force from the positive-firing thruster at step k.
  - $u_{neg,k}$ is the force from the negative-firing thruster at step k.
  - $\Delta t = 2$ seconds is the duration of each time step.

- **Thrusters and Fuel:**

  - The thrusters can fire with a force between 0 and 5 Newtons: $0 \le u_{pos,k} \le 5$ and $0 \le u_{neg,k} \le 5$.

– The total fuel consumed is proportional to the total force applied by both thrusters over the entire maneuver.

Your Task: Discretize the problem into 5 time steps (at t = 0, 2, 4, 6, 8 seconds). Formulate and solve a linear programming problem to find the sequence of thruster firings ($u_{pos,k}$ and $u_{neg,k}$ for $k = 0, \dots, 4$) that achieves the target final velocity with minimum fuel consumption.

6.1.8.1 The Challenge

1. **Define Decision Variables:** Your variables will be the thruster firings at each time step: $u_{pos,0}, u_{neg,0}, u_{pos,1}, u_{neg,1}, \dots, u_{pos,4}, u_{neg,4}$. There will be 10 variables in total.
2. **Define the Objective Function:** Minimize total fuel, which is the sum of all decision variables.
3. **Define Constraints:**
   - **Final Velocity Constraint:** The angular velocity at the end of the last step (at t=10s) must be exactly 1.5 rad/s. This will be an **equality constraint**. You will need to write out the expression for the final velocity ω_5 in terms of the initial velocity ω_0 and all the decision variables.
   - **Bounds:** Each thruster firing must be between 0 and 5 N.

6.1.8.2 Hint

- The final velocity ω_5 can be found by unrolling the dynamics equation: $\omega_5 = \omega_0 + \alpha\Delta t \sum_{k=0}^{4}(u_{pos,k} - u_{neg,k})$
- This is a perfect fit for the `linprog` function, which can handle equality constraints (`A_eq`, `b_eq`) and variable bounds directly.

6.1.8.3 Solution to the Application Challenge

First, we formulate the problem mathematically.

- **Decision Variables (10 total):** $x = [u_{pos,0}, u_{neg,0}, u_{pos,1}, u_{neg,1}, u_{pos,2}, u_{neg,2}, u_{pos,3}, u_{neg,3}, u_{pos,4}, u_{neg,4}]$
- **Objective (Minimize Fuel F):**

$$\text{Minimize } F = \sum_{k=0}^{4}(u_{pos,k} + u_{neg,k})$$

The coefficient vector `c` will be an array of all ones: `c = [1, 1, 1, 1, ..., 1]`.

- **Constraints:**
  1. **Final Velocity (Equality):** $\omega_5 = \omega_0 + \alpha\Delta t \sum_{k=0}^{4}(u_{pos,k} - u_{neg,k}) = 1.5$ Given $\omega_0 = 0$, $\alpha = 0.1$, $\Delta t = 2$, this becomes:

$$0.1 \cdot 2 \cdot \sum_{k=0}^{4}(u_{pos,k} - u_{neg,k}) = 1.5$$
$$0.2 \cdot [(u_{pos,0} - u_{neg,0}) + (u_{pos,1} - u_{neg,1}) + \dots] = 1.5$$

     This is a single linear equality constraint. The row vector `A_eq` will be `[0.2, -0.2, 0.2, -0.2, ...]`. The right-hand side `b_eq` will be `[1.5]`.
  2. **Bounds:** $0 \leq x_i \leq 5$ for all $i = 0, \dots, 9$.

6.1.8.4 Python Implementation

```
import numpy as np
from scipy.optimize import linprog
import matplotlib.pyplot as plt

# --- System Parameters ---
alpha = 0.1
dt = 2.0
w_initial = 0.0
w_final = 1.5
num_steps = 5
u_max = 5.0

# --- 1. Formulate the LP Problem ---
# Objective: Minimize sum of all u's. 10 variables (u_pos_k, u_neg_k for k=0..4)
c = np.ones(2 * num_steps)

# Equality Constraint: Final velocity
# 0.2 * (u_pos_0 - u_neg_0 + u_pos_1 - u_neg_1 + ...) = 1.5
A_eq_row = []
for i in range(num_steps):
    A_eq_row.extend([alpha * dt, -alpha * dt])

A_eq = [A_eq_row]
b_eq = [w_final - w_initial]

# Bounds for each variable: 0 <= u <= 5
```

```
bounds = (0, u_max)

# --- 2. Solve the LP Problem ---
result = linprog(c, A_eq=A_eq, b_eq=b_eq, bounds=bounds, method='highs')

# --- 3. Display and Visualize the Result ---
if result.success:
    min_fuel = result.fun
    firings = result.x

    # Reshape the result for easier interpretation
    u_pos = firings[0::2] # Every other element starting from 0
    u_neg = firings[1::2] # Every other element starting from 1

    print("Optimal Thruster Firing Plan Found!")
    print(f"Minimum Total Fuel (proportional to) = {min_fuel:.2f}")

    print("\nFiring sequence (in Newtons):")
    print("Time Step | Positive Thruster | Negative Thruster")
    print("----------|-------------------|------------------")
    for k in range(num_steps):
        print(f"    {k}     |      {u_pos[k]:.2f}         |       {u_neg[k]:.2f}")

    # Visualize the results
    time_axis = np.arange(num_steps) * dt

    plt.figure(figsize=(10, 6))
    plt.bar(time_axis - 0.2, u_pos, width=0.4, label='Positive Thruster (u_pos)', align='cent
    plt.bar(time_axis + 0.2, u_neg, width=0.4, label='Negative Thruster (u_neg)', align='cent
    plt.xlabel('Time (s)')
    plt.ylabel('Thruster Force (N)')
    plt.title('Optimal Thruster Firing Sequence')
    plt.xticks(time_axis)
    plt.legend()
    plt.grid(axis='y', linestyle='--')
    plt.show()

else:
    print("Optimization failed.")
    print(f"Message: {result.message}")
```

```
Optimal Thruster Firing Plan Found!
```

```
Minimum Total Fuel (proportional to) = 7.50

Firing sequence (in Newtons):
Time Step | Positive Thruster | Negative Thruster
----------|-------------------|------------------
    0     |       0.00        |       0.00
    1     |       0.00        |       0.00
    2     |       0.00        |       0.00
    3     |       2.50        |       0.00
    4     |       5.00        |       0.00
```

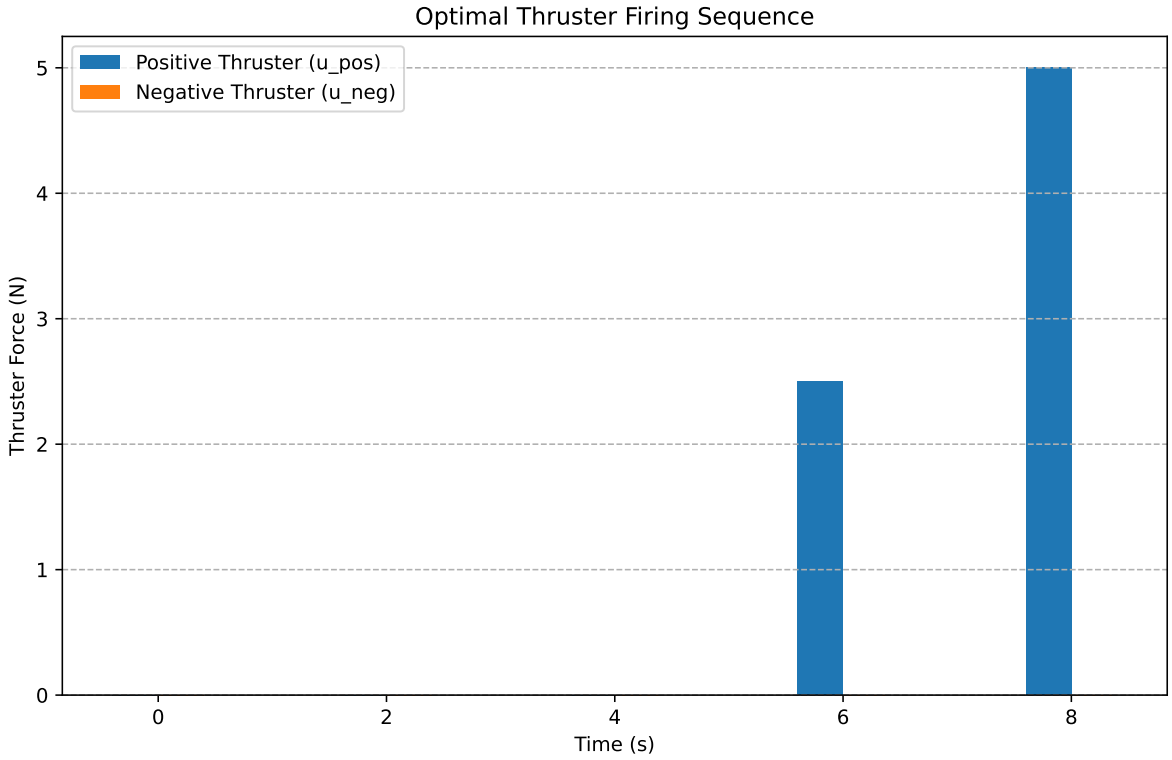


Figure 6.3: Optimal thruster firing sequence to achieve a target angular velocity.

6.1.8.5 Results and Discussion

- Optimal Solution: The optimal strategy found by the solver is to fire only the *positive thruster* and never the negative one. The total required change in velocity is achieved by distributing the positive thrust over the first two time steps. Specifically, the solver commands a firing of *5.0 N for the first step* (from t=0 to t=2s) and *2.5 N for the*

second step (from t=2s to t=4s), with zero thrust thereafter. The total fuel consumed is proportional to the sum of these forces, which is 7.5 units.

- Physical Interpretation: This result is perfectly logical and highly intuitive. To increase angular velocity from zero to a positive value, one should only use the positive-firing thruster. Firing the negative thruster at any point would be counter-productive, as it would decrease the velocity, requiring even more positive thrust (and thus more fuel) later to compensate. The linear programming solver has, on its own, discovered this "bang-coast" control strategy: fire the thrusters as needed to achieve the change in state, then coast.
- Binding Constraints: The final velocity constraint is, by definition, a *binding constraint* because we forced the solution to meet it exactly. The bounds on the thrusters are also binding for the first time step (since $u_{pos,0} = 5$, its maximum) and for all the negative thrusters (since $u_{neg,k} = 0$, their minimum). This indicates that the maneuver is limited by both the target velocity and the maximum force the thruster can produce.
- Engineering Significance: This problem is a simplified but powerful example of *optimal control*, a major field within control engineering. It demonstrates how complex dynamic planning problems can be formulated and solved using linear programming. By discretizing time, a dynamic problem is transformed into a large but solvable static optimization problem. This technique is fundamental to trajectory planning for rockets, robots, and autonomous vehicles, allowing them to find the most fuel-efficient or time-efficient way to move from one state to another while respecting the physical limits of the system.

6.2 Experiment 10: The Transportation Problem

The Transportation Problem is a classic optimization problem in logistics and operations research. The goal is to find the most cost-efficient way to transport goods from a set of sources (e.g., factories) to a set of destinations (e.g., warehouses), while satisfying supply and demand constraints.

6.2.1 Aim

To find the optimum, minimum-cost solution for a given transportation problem.

6.2.2 Objectives

- To understand the structure of a transportation problem, including costs, supply, and demand.
- To recognize that the transportation problem is a special case of Linear Programming.

- To formulate the problem as a linear program and solve it efficiently using Python's `scipy.optimize.linprog`.

6.2.2.1 Understanding the Transportation Problem

A transportation problem is defined by three components: 1. **Sources:** A set of `m` sources, each with a given supply capacity, S_i. 2. **Destinations:** A set of `n` destinations, each with a given demand requirement, D_j. 3. **Cost Matrix:** A cost matrix `C`, where C_{ij} is the cost of shipping one unit from source `i` to destination `j`.

The problem is "balanced" if total supply equals total demand: $\sum S_i = \sum D_j$.

Traditional Algorithm (Conceptual) Historically, this problem was solved with specialized algorithms like: 1. **Phase 1 (Initial Solution):** Methods like the **North-West Corner Rule** or **Least Cost Method** are used to find an initial, feasible (but not necessarily optimal) shipping plan. 2. **Phase 2 (Optimization):** The **MODI (Modified Distribution) Method** or the **Stepping Stone Method** is then used iteratively to adjust the initial plan, reducing the total cost until no further improvement is possible and the optimal solution is found.

While implementing these is a great way to understand the theory, it is complex and inefficient. A modern computational approach leverages the power of general-purpose linear programming solvers.

6.2.3 Modern Approach: Formulation as a Linear Program

This is the preferred method for a computational course.

1. **Decision Variables:** Let x_{ij} be the number of units to ship from source `i` to destination `j`. This creates a total of $m \times n$ variables.
2. **Objective Function (Minimize Total Cost):**

$$\text{Minimize } Z = \sum_{i=1}^{m} \sum_{j=1}^{n} C_{ij} x_{ij}$$

3. **Constraints:**

- **Supply Constraints:** The amount shipped from each source cannot exceed its supply. (One equation for each source).

$$\sum_{j=1}^{n} x_{ij} = S_i \quad \text{for } i = 1, \dots, m$$

- **Demand Constraints:** The amount received at each destination must meet its demand. (One equation for each destination).

$$\sum_{i=1}^{m} x_{ij} = D_j \quad \text{for } j = 1, \dots, n$$

- **Non-negativity:** The amount shipped cannot be negative.

$$x_{ij} \geq 0 \quad \text{for all } i, j$$

This structure fits perfectly into the `linprog` solver.

6.2.4 Problem: Manufacturing Plant Logistics

Problem: A manufacturing plant has three suppliers (S1, S2, S3) and three warehouses (W1, W2, W3). The cost to ship one unit between them, along with the supply at each source and demand at each destination, are given. Find the shipping plan that minimizes the total cost.

- **Cost Matrix (C_{ij}):**

From \ To	W1	W2	W3
S1	4	6	8
S2	2	5	7
S3	3	4	6

- **Supply Vector (S):** `[20, 30, 25]` (Total Supply = 75)
- **Demand Vector (D):** `[30, 25, 20]` (Total Demand = 75) Since total supply equals total demand, the problem is balanced.

6.2.4.1 Python Implementation using Linear Programming

```python
import numpy as np
from scipy.optimize import linprog

# --- 1. Define the Problem Data ---
costs = np.array([[4, 6, 8],
                  [2, 5, 7],
                  [3, 4, 6]])

supply = np.array([20, 30, 25])
demand = np.array([30, 25, 20])

num_sources, num_dests = costs.shape

# --- 2. Formulate as a Linear Program ---
# The decision variables x_ij are flattened into a single 1D array.
# c is the flattened cost matrix.
c = costs.flatten()

# Equality Constraints (A_eq, b_eq)
# We have supply constraints and demand constraints.
A_eq = []
b_eq = []

# Supply constraints: sum over destinations for each source
for i in range(num_sources):
    row = np.zeros(num_sources * num_dests)
    row[i*num_dests : (i+1)*num_dests] = 1
    A_eq.append(row)
    b_eq.append(supply[i])

# Demand constraints: sum over sources for each destination
for j in range(num_dests):
    row = np.zeros(num_sources * num_dests)
    row[j::num_dests] = 1 # Selects x_0j, x_1j, x_2j, ...
    A_eq.append(row)
    b_eq.append(demand[j])

# Bounds for each variable must be non-negative
bounds = (0, None)

# --- 3. Solve the LP Problem ---
result = linprog(c, A_eq=A_eq, b_eq=b_eq, bounds=bounds, method='highs')

# --- 4. Display the Results ---
if result.success:
    min_cost = result.fun
    # Reshape the flat result back into a 2D allocation matrix
    allocation = result.x.reshape(num_sources, num_dests)

    print("Optimal Transportation Plan Found!")
    print(f"\nMinimum Total Cost = ${min_cost:.2f}")

    print("\nOptimal Allocation Matrix (units to ship):")
```

6.2.4.2 Result and Discussion

- **Optimal Solution:** The linear programming solver found the optimal shipping plan with a *minimum total cost of $265.00*. The specific allocation is detailed in the optimal allocation matrix:
  - From Source 1: Ship 20 units to Destination 1.
  - From Source 2: Ship 10 units to Destination 1 and 20 units to Destination 3.
  - From Source 3: Ship 25 units to Destination 2.
- **Verification of Constraints:** The optimal solution perfectly adheres to all supply and demand constraints:
  - **Supply:** Source 1 ships 20 (supply=20). Source 2 ships 10+20=30 (supply=30). Source 3 ships 25 (supply=25). All supplies are fully utilized.
  - **Demand:** Destination 1 receives 20+10=30 (demand=30). Destination 2 receives 25 (demand=25). Destination 3 receives 20 (demand=20). All demands are fully met.
- Comparison to Heuristic Methods: It is important to compare this result to what a simpler, manual method might yield. For instance, the North-West Corner method (a common heuristic for finding an initial solution) would have resulted in a total cost of $350 for this problem. The LP solver immediately found a solution that is *24% cheaper*. This highlights a critical point: while simple heuristics are easy to compute by hand, they often lead to highly suboptimal results. The LP formulation provides a guaranteed optimal solution.
- Engineering and Business Significance: This method is fundamental to supply chain management and logistics in any large-scale engineering or manufacturing operation. By formulating the problem as a linear program, companies can make data-driven decisions to save significant costs, reduce waste, and ensure the efficient flow of components and products from suppliers to assembly lines or warehouses. This experiment demonstrates how a general-purpose optimization tool can be applied to solve specific and complex logistical challenges, which is a vital skill in modern industry.

6.2.5 Application Challenge: Optimal Power Distribution Grid

Scenario: An energy company operates three power plants (sources) that need to supply electricity to four different cities (destinations). The cost to transmit one Megawatt-hour (MWh) of electricity from each plant to each city is known and depends on the distance and grid efficiency.

The goal is to determine the most cost-effective power distribution plan to meet the peak demand of all cities without exceeding the generation capacity of any plant.

- **Transmission Cost Matrix (C_{ij} in \$/MWh):**

From To	City A	City B	City C	City D
Plant 1	10	18	25	15
Plant 2	12	10	8	22
Plant 3	20	15	12	10

- **Supply Capacity (S in MWh):** `[350, 500, 400]` (Total Supply = 1250 MWh)
- **Demand Requirement (D in MWh):** `[250, 300, 400, 200]` (Total Demand = 1150 MWh)

Your Task: Formulate and solve this as a transportation problem to find the power distribution plan that minimizes the total transmission cost.

6.2.5.1 The Challenge

1. **Unbalanced Problem:** Notice that the total supply (1250 MWh) is greater than the total demand (1150 MWh). The standard transportation LP formulation handles this naturally. The supply constraints should be "less than or equal to" ($\leq$), while the demand constraints must be "equal to" ($=$) to ensure all city needs are met.
2. **Formulate as an LP:**
    - The decision variables x_{ij} represent the MWh of power sent from Plant `i` to City `j`. There will be $3 \times 4 = 12$ variables.
    - The objective is to minimize total transmission cost.
    - Set up the supply ($\leq$) and demand ($=$) constraints.

6.2.5.2 Hint

- You will need to create two sets of constraints: one for inequalities (`A_ub`, `b_ub`) for the supply, and one for equalities (`A_eq`, `b_eq`) for the demand. `scipy.optimize.linprog` can handle both simultaneously.

6.2.6 Solution to the Application Challenge

First, we formulate the problem mathematically.

- **Objective (Minimize Cost Z):**

$$\text{Minimize } Z = \sum_{i=1}^{3} \sum_{j=1}^{4} C_{ij} x_{ij}$$

- **Constraints:**

1. **Supply Constraints ($\leq$):** The power sent from each plant cannot exceed its capacity.
   - $x_{11} + x_{12} + x_{13} + x_{14} \leq 350$
   - $x_{21} + x_{22} + x_{23} + x_{24} \leq 500$
   - $x_{31} + x_{32} + x_{33} + x_{34} \leq 400$
2. **Demand Constraints ($=$):** The power received by each city must exactly meet its demand.
   - $x_{11} + x_{21} + x_{31} = 250$
   - $x_{12} + x_{22} + x_{32} = 300$
   - $x_{13} + x_{23} + x_{33} = 400$
   - $x_{14} + x_{24} + x_{34} = 200$
3. **Non-negativity:** $x_{ij} \geq 0$.

6.2.6.1 Python Implementation

6.2.6.2 Results and Discussion

- Optimal Solution: The optimal power distribution plan results in a *minimum total transmission cost of $11,500*. The specific power flows are detailed in the allocation matrix. The solver intelligently assigns generation to meet demand via the cheapest available routes. For example, Plant 2, having the lowest cost to City C ($8/MWh), supplies all 500 MWh of that city's demand. Similarly, Plant 3, with the best rate to City D ($10/MWh), covers all of its 400 MWh demand. The needs of the more expensive-to-reach cities (A and B) are met by the remaining capacity from the most cost-effective plants.
- Handling Unbalanced Problems: The key to this problem was correctly formulating the constraints. The supply constraints were set as inequalities (`<=`), allowing plants to generate less than their maximum capacity. Conversely, the demand constraints were set as equalities (`=`), forcing the system to satisfy the needs of every city. The LP solver

```python
import numpy as np
from scipy.optimize import linprog

# --- 1. Define the Problem Data ---
costs = np.array([[10, 18, 25, 15],
                  [12, 10,  8, 22],
                  [20, 15, 12, 10]])

supply_capacity = np.array([350, 500, 400])
demand_req = np.array([250, 300, 400, 200])

num_plants, num_cities = costs.shape

# --- 2. Formulate as a Linear Program ---
# Flatten the cost matrix to create the objective coefficient vector
c = costs.flatten()

# Inequality constraints (A_ub, b_ub) for supply (<=)
A_ub = []
for i in range(num_plants):
    row = np.zeros(num_plants * num_cities)
    row[i*num_cities : (i+1)*num_cities] = 1
    A_ub.append(row)
b_ub = supply_capacity

# Equality constraints (A_eq, b_eq) for demand (=)
A_eq = []
for j in range(num_cities):
    row = np.zeros(num_plants * num_cities)
    row[j::num_cities] = 1 # Selects x_0j, x_1j, x_2j
    A_eq.append(row)
b_eq = demand_req

# Bounds for each variable must be non-negative
bounds = (0, None)

# --- 3. Solve the LP Problem ---
result = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq, bounds=bounds, method='highs

# --- 4. Display the Results ---
if result.success:
    min_cost = result.fun
    allocation = result.x.reshape(num_plants, num_cities)

    print("Optimal Power Distribution Plan Found!")
    print(f"\nMinimum Total Transmission Cost = ${min_cost:,.2f}")

    print("\nOptimal Allocation Matrix (in MWh):")
    print("           City A   City B   City C   City D")
    print("-----------------------------------------------")
    for i in range(num_plants):
        print(f"Plant {i+1} | {allocation[i, 0]:>7.0f}  {allocation[i, 1]:>7.0f}  {allocatio
```

handled this mixed-constraint system perfectly, automatically creating a "slack" in the supply where it was most economical.

- Resource Utilization and Strategic Insights: The solution reveals that the total power generated is 1150 MWh, exactly matching the total demand. This leaves an **unused generation capacity of 100 MWh** in the system. The utilization breakdown clearly shows this slack capacity is entirely at **Plant 1**, which only generates 250 MWh out of its 350 MWh maximum.

  – Strategic Implication: This result provides a powerful insight for the energy company: Plant 1 is their most "expensive" or least strategically located plant relative to the current demand centers. For long-term planning, they might consider decommissioning or reducing the maintenance budget for Plant 1. Alternatively, they could use this model to incentivize new, energy-intensive industries to build facilities near Plant 1, offering them lower transmission costs and taking advantage of the surplus capacity.

- Engineering Significance: This application demonstrates how optimization is critical in the design and operation of large-scale infrastructure like power grids. By using linear programming, grid operators can perform "economic dispatch," deciding in near real-time which power plants should ramp up or down to meet fluctuating demand at the lowest possible cost. This ensures both the stability of the grid and its economic efficiency, a core task in power systems engineering.

Part V

Capstone Projects

7 Chapter 6: Capstone Project - A.R.E.S.

7.1 Autonomous Rover for Exploration and Science

7.1.1 Introduction: Your Mission

Welcome to your final challenge. You have spent this semester acquiring a powerful computational toolkit. Now, it's time to synthesize that knowledge to solve a complex, multi-domain engineering problem.

You are the lead software engineer for **Project A.R.E.S.**, a simulated Martian rover mission. The rover's goal is to visit scientifically interesting locations, deploy a robotic arm, and drill for samples to analyze. Your mission is constrained by two critical resources: **mission time** and **battery power**.

Your task is to design, model, and simulate the rover's planning and control systems. You will determine the optimal sequence of actions to **maximize the scientific return** of the mission, while respecting the physical limitations of the hardware. This project will directly test your skills in linear algebra, system dynamics (Laplace transforms), partial differential equations, and optimization.

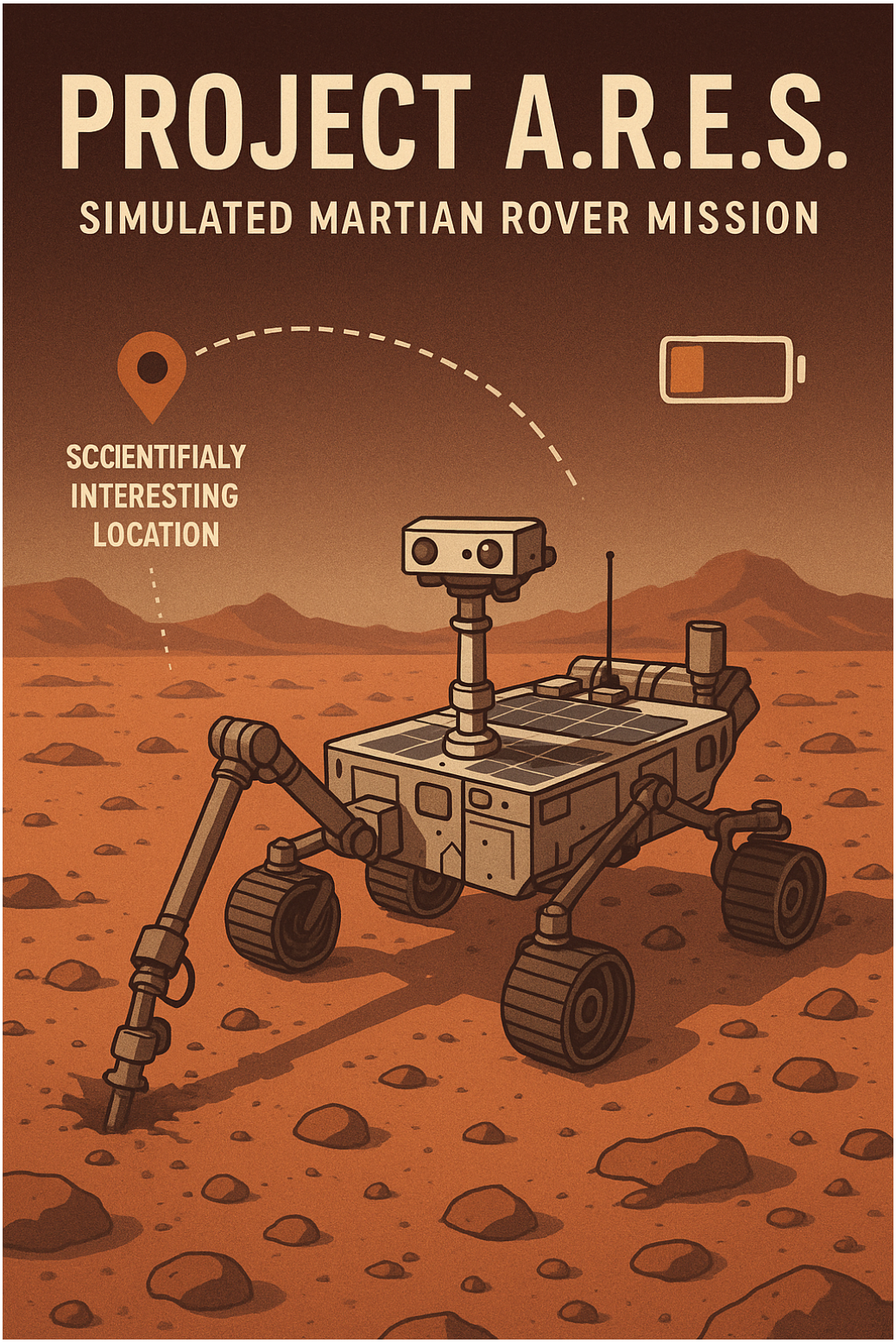


Figure 7.1: Conceptual image of the A.R.E.S. rover mission (created with Sora)

7.1.2 Project Structure & Core Modules

The project is divided into five interconnected modules. Each module builds upon concepts from your previous lab experiments and contributes essential parameters to the final mission plan.

7.1.3 Module 1: Kinematics & World Modeling

(Concepts from: Lab 1 - Linear Algebra)

The first step is to mathematically describe the rover, its arm, and its environment.

- **Task 1: The World Frame.** Create a 2D map in Python. Define the rover's starting position (e.g., `(0, 0)`) and the coordinates of two scientific targets (e.g., Target A: `(5, 8)`, Target B: `(12, 4)`).
- **Task 2: The Robotic Arm.** Model the rover's 2-link planar arm.
  - Implement a **Forward Kinematics** function. This function will take the two joint angles, (θ_1, θ_2), and the link lengths, (L_1, L_2), as input and must return the `(x, y)` position of the drill bit (end-effector) *in the rover's own coordinate frame*.
  - **Relevant Skill:** Use transformation matrices or trigonometry, as learned in early array operations.
- **Task 3: Coordinate Transformations.** Write a function that converts the drill's local coordinates to the global world frame coordinates. This is essential for verifying that the rover has positioned itself correctly to reach a target on the map.
- **Deliverable:** A Python script (`kinematics.py`) with functions for forward kinematics and coordinate transformation. Include a simple `matplotlib` plot showing the rover and its arm reaching for a target point.

7.1.4 Module 2: Arm Control & System Dynamics

(Concepts from: Lab 3 & 4 - Laplace Transforms & ODEs)

A real motor doesn't move instantly. We must model its dynamic response to determine how long it takes to stabilize.

- **Task 1: System Modeling.** Model a single joint of the robotic arm as a classic second-order **mass-spring-damper system**. The differential equation for its angular position, $\theta(t)$, in response to a commanded target angle, $\theta_{ref}(t)$, is:

$$I\frac{d^2\theta}{dt^2} + c\frac{d\theta}{dt} + k\theta(t) = k\theta_{ref}(t)$$

  Use the following parameters: Inertia $I = 1$, Damping $c = 4$, and Stiffness $k = 13$.

- **Task 2: Laplace Analysis.**
  - Assume the rover commands the joint with a **unit step input** (i.e., $\theta_{ref}(t) = u(t)$).
  - Using the skills from Experiment 7, solve this ODE with the **Laplace Transform method** in `sympy`. Assume the arm starts from rest: $\theta(0) = 0$ and $\dot{\theta}(0) = 0$.

- **Task 3: Constraint Discovery - Settling Time.**
  - Plot the symbolic solution $\theta(t)$.
  - From your plot and data, determine the **settling time**: the time it takes for the joint's angle to get to and stay within 2% of its final value (which is 1.0). This `settling_time` is a critical mission parameter—it's the minimum time the rover must wait for the arm to be stable before drilling.

- **Deliverable:** A Python script (`dynamics.py`) that solves the ODE and plots the step response. Clearly state the calculated `settling_time`.

7.1.5 Module 3: Thermal Management During Drilling

(Concepts from: Lab 2 - Partial Differential Equations)

Drilling generates significant heat. If the drill bit overheats, it can be damaged. We must model this to find safe operational limits.

- **Task 1: Heat Modeling.** Model the drill bit as a 1D rod of length $L = 0.2$ m. The temperature evolution, $u(x, t)$, is governed by the **1D Heat Equation** with an internal heat source term, Q, representing friction from drilling:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} + Q$$

Use parameters: $\alpha = 10^{-5}$ (thermal diffusivity for steel) and a heat generation rate of $Q = 50$ (while drilling). Assume the ends of the drill are exposed to the cold Martian atmosphere, so we use boundary conditions: $u(0, t) = u(L, t) = 0°C$.

- **Task 2: Numerical Solution.** Implement a **Finite Difference Method** (like the FTCS scheme from Experiment 3) to solve this PDE.
- **Task 3: Constraint Discovery - Duty Cycle.**
  - Simulate the temperature rise, starting from $u(x, 0) = 0°C$ with the heat source on ($Q = 50$).
  - Determine the maximum time the rover can continuously drill before any point on the bit exceeds a safety limit of **80°C**.
  - Next, turn the heat source off ($Q = 0$) and determine how long it takes for the peak temperature to drop back down to a safe level (e.g., 20°C).
  - This gives you a **Drill/Cool Duty Cycle** (e.g., "Drill for 45s, then must cool for 30s"). This is another critical input for the mission planner.
- **Deliverable:** A Python script (`thermal.py`) that solves the PDE and outputs the calculated drill and cool times.

7.1.6 Module 4: Optimal Mission Planning

(Concepts from: Lab 5 - Optimization)

With all physical constraints defined, we can now determine the best mission plan.

- **Task 1: Define Mission Parameters.**
  - **Science Value:** Drilling at Target A yields **100 points**. Drilling at Target B yields **150 points**.
  - **Drilling Requirements:** Target A requires 60s of total drilling. Target B requires 90s.
  - **Mission Constraints:** Total Time $\leq$ 900s. Total Energy $\leq 50,000$ J.
- **Task 2: Define Activity Costs.**
  - **Driving:** 20 J/s. Time is calculated from distance (assume speed of 1 m/s).

  - **Arm Movement:** 500 J per deployment. Takes the `settling_time` from Module 2.
  - **Drilling:** 100 J/s. Must be broken into cycles based on the duty cycle from Module 3.
  - **Cooling/Idle:** 5 J/s.

- **Task 3: LP Formulation.** Formulate this as a **Linear Programming** problem.
  - **Decision Variables:** A clever way to formulate this is to define variables for each *path*, e.g., $x_{start \to A \to B}$, $x_{start \to B \to A}$, $x_{start \to A}$, etc. Let these variables be binary (0 or 1).
  - **Objective Function:** Write an equation for the total science points based on which path is chosen. Your goal is to **maximize** this function.
  - **Constraints:** Write linear equations for total time and total energy consumed for each possible path. Ensure these are less than or equal to the mission limits.

- **Deliverable:** A Python script (`optimizer.py`) using `scipy.optimize.linprog` that solves the LP problem and outputs the best plan (e.g., "Go to Target B, drill, and return") and the corresponding maximum science score.

7.1.7 Module 5: Integrated Mission Simulation

This is the capstone deliverable, where everything comes together.

- **Task:** Create a live "Mission Dashboard" using `matplotlib.animation`. The dashboard will visualize the optimal plan determined in Module 4. It should contain several subplots that update in real-time:

  1. **World View:** A 2D plot showing the map and a rover icon moving along the optimal path.
  2. **System Status:** Text readouts for `Mission Time`, `Energy Remaining`, and `Current Action`.
  3. **Arm Dynamics Plot:** When the rover's action is "Deploying Arm", this plot shows the joint angle settling over time, as modeled in Module 2.
  4. **Drill Temperature Plot:** When the action is "Drilling" or "Cooling", this plot shows the temperature profile of the drill bit evolving according to the PDE solution from Module 3.

- **Final Deliverable:** A single, executable Python script `main.py` that runs the entire simulation, integrating the logic from all other modules.

7.1.8 Submission Requirements

1. **Codebase:** A folder containing all your Python scripts (`kinematics.py`, `dynamics.py`, `thermal.py`, `optimizer.py`, `main.py`).
2. **Project Report:** A Quarto (`.qmd`) or PDF report detailing the mathematical formulation, design choices, results, and discussion for each module.
3. **Demonstration:** A short video (e.g., screen recording) of your final simulation running on your local machine.

8 Chapter 7: Capstone Project-II - P.A.T.H.F.I.N.D.E.R.

8.1 Picking and Assembly Task Handler For Industrial Navigation, Dynamics, and Energy Reduction

8.1.1 Introduction: Your Mission

Welcome to your second capstone challenge. This project shifts our focus from space exploration to the heart of modern manufacturing: **industrial automation**.

You are the lead robotics engineer for a "smart factory" cell. Your task is to program a stationary robotic arm to perform a pick-and-place operation. The arm must pick up parts from a designated pickup zone and place them into one of two assembly jigs, navigating around a fixed obstacle in its workspace. The factory's goal is to **minimize the energy consumption** of the arm for a complete cycle, subject to a strict **cycle time limit** to maintain production throughput.

This project will test your ability to model a robot's workspace, analyze its gripper dynamics, plan collision-free paths using PDEs, and optimize its motion for energy efficiency.

8.1.2 Project Structure & Core Modules

This project follows a five-module structure, applying the course concepts to this new industrial scenario.

8.1.3 Module 1: Workspace & Kinematics

(Concepts from: Lab 1 - Linear Algebra & Coordinate Geometry)

Before the arm can move, we must define its physical capabilities and its environment.

- **Task 1: The Workspace.** Create a 2D top-down map of the robot's workspace. Define the coordinates for the arm's base, a "Pickup Zone," two "Assembly Jigs" (A and B), and a fixed "Obstacle" (e.g., a pillar or another piece of machinery).
- **Task 2: Inverse Kinematics.** Model a 2-link planar arm. Instead of forward kinematics, your primary task is to implement a simple **Inverse Kinematics** function. Given a target `(x, y)` coordinate in the workspace, this function must calculate the required joint angles (θ_1, θ_2) for the arm to reach it.
- **Task 3: Reachability Check.** Use your inverse kinematics function to create a `can_reach(target_pos)` helper function. This function will be crucial to confirm that the pickup and assembly zones are physically within the arm's reach.
- **Deliverable:** A Python script (`kinematics_v2.py`) with functions for inverse kinematics. Include a plot showing the arm's configuration when reaching for each of the key locations (pickup and jigs).

8.1.4 Module 2: Gripper Dynamics & Actuation

(Concepts from: Lab 3 & 4 - Laplace Transforms & ODEs)

The arm's gripper is a dynamic system. We must model its closing time to ensure it securely grasps parts.

- **Task 1: System Modeling.** Model the gripper's closing mechanism as a second-order system. The differential equation for the gripper's finger position, $x(t)$, in response to a "close" command is:

  $$m\ddot{x}(t) + c\dot{x}(t) + kx(t) = F_{motor}$$

  where F_{motor} is a constant force applied by the actuator. Use parameters: Mass $m = 0.1$, Damping $c = 5$, Spring-like resistance $k = 50$, and Motor Force $F_{motor} = 50$.
- **Task 2: Laplace Analysis.**
  - Assume the "close" command is a **step input** of force at $t = 0$.
  - Use the **Laplace Transform method** in `sympy` to solve this ODE for $x(t)$, with initial conditions $x(0) = 0$ and $\dot{x}(0) = 0$.
- **Task 3: Constraint Discovery - Gripper Settling Time.**
  - Plot the step response $x(t)$.
  - Determine the **settling time**: the time it takes for the gripper to close and stabilize. This `gripper_settling_time` is the minimum time the robot must wait after issuing a "grasp" command before it can start moving the part.

- **Deliverable:** A Python script (`dynamics_v2.py`) that solves the gripper ODE and plots its response. Clearly state the calculated `gripper_settling_time`.

8.1.5 Module 3: Obstacle-Aware Path Planning

(Concepts from: Lab 2 - Partial Differential Equations)

To move safely, the arm must navigate around the obstacle. We will implement a classic robotics algorithm called **Potential Field Path Planning**.

- **Task 1: The Potential Field.** The workspace is discretized into a grid. The path is found by solving **Laplace's Equation** over this grid:

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0$$

  - **Boundary Conditions:** Set the potential ϕ to a high value at the obstacle's boundary (e.g., $\phi = 1$) and a low value at the target's location (e.g., $\phi = 0$).

- **Task 2: Numerical PDE Solution.** Implement an iterative solver (like the **Jacobi method**) to solve for the potential $\phi(x, y)$ at every point on the grid. This creates a smooth "potential surface" where the target is a low point and obstacles are high points.
- **Task 3: Path Generation.** Write a function that finds a path from a start point to the target by using **gradient descent**. From any point on the grid, the next point on the path is in the direction of the steepest descent (the negative gradient, $-\nabla\phi$).
- **Deliverable:** A Python script (`path_planner.py`) that computes a potential field for a given target and generates a collision-free path for the arm's end-effector. Include a heatmap visualization of the potential field with the generated path overlaid.

8.1.6 Module 4: Energy & Time Optimization

(Concepts from: Lab 5 - Optimization)

The factory wants to minimize electricity costs while meeting production quotas. This requires optimizing the arm's movement speed.

- **Task 1: Define Motion Profiles.** The arm can move along the path from Module 3 at two different speeds:

  - **Slow & Precise:** Low energy cost (20 J/s), speed = 0.5 m/s.
  - **Fast & Inaccurate:** High energy cost (70 J/s), speed = 1.5 m/s.

- **Task 2: LP Formulation.** For a complete pick-and-place cycle (Pickup Zone -> Assembly Jig A), formulate a linear program to find the optimal motion profile.
  - **Decision Variables:** Let t_{slow} be the time spent moving at slow speed, and t_{fast} be the time spent moving at fast speed.
  - **Objective Function: Minimize** total energy: $Z = 20t_{slow} + 70t_{fast}$.
  - **Constraints:**
    1. **Total Distance Constraint:** The distance covered must equal the path length: $0.5t_{slow} + 1.5t_{fast} =$ Total Path Length. This is an equality constraint.
    2. **Total Time Constraint:** The total time for the cycle must be less than or equal to a factory-imposed limit (e.g., 15 seconds):

$$t_{\text{slow}} + t_{\text{fast}} + \text{gripper settling time} \leq 15.$$

- **Deliverable:** A Python script (`optimizer_v2.py`) using `scipy.optimize.linprog` that calculates the optimal time to spend at each speed to meet the deadline with minimum energy.

8.1.7 Module 5: Integrated Factory Cell Simulation

This is the final deliverable, combining all modules into a cohesive visualization.

- **Task:** Create a live "Factory Dashboard" animation using `matplotlib.animation`.
  1. **Workspace View (Main Plot):** A top-down 2D plot showing the workspace, obstacle, pickup/jig locations, and the robotic arm. The arm should animate its movement along the path generated by the potential field planner.
  2. **Potential Field View (Subplot 1):** A heatmap of the potential field from Module 3. An overlaying marker should show the arm's current position on the gradient.
  3. **System Status (Subplot 2):** Text readouts for `Cycle Time`, `Energy Consumed`, and `Current Action` (e.g., "Moving to Pickup", "Grasping Part", "Moving to Jig A").
  4. **Gripper Dynamics (Subplot 3):** When the action is "Grasping Part," this plot shows the live gripper closing response from Module 2.

- **Final Deliverable:** A single, executable Python script `main_v2.py` that runs the entire factory arm simulation, demonstrating one full, optimized pick-and-place cycle.