# **Lecture 10**

## ***Learning outcomes:***

- ➢ *Solving Engineering Problems with Python:*
- ▪ *Case studies: heat transfer, structural analysis, fluid mechanics, etc*
- ▪ *Solving differential equations with SciPy*
- ▪ *Engineering optimization problems*

## *Case studies: heat transfer, structural analysis, fluid mechanics*

## Solving Engineering Problems with Python: Case Studies

- Python is a powerful programming language that has become an essential tool for solving engineering problems due to its versatility, rich ecosystem of libraries, and ease of use.

- Here are some case studies showcasing how Python can be applied to real-world engineering problems in heat transfer, structural analysis, and fluid mechanics.

## Heat Transfer Analysis

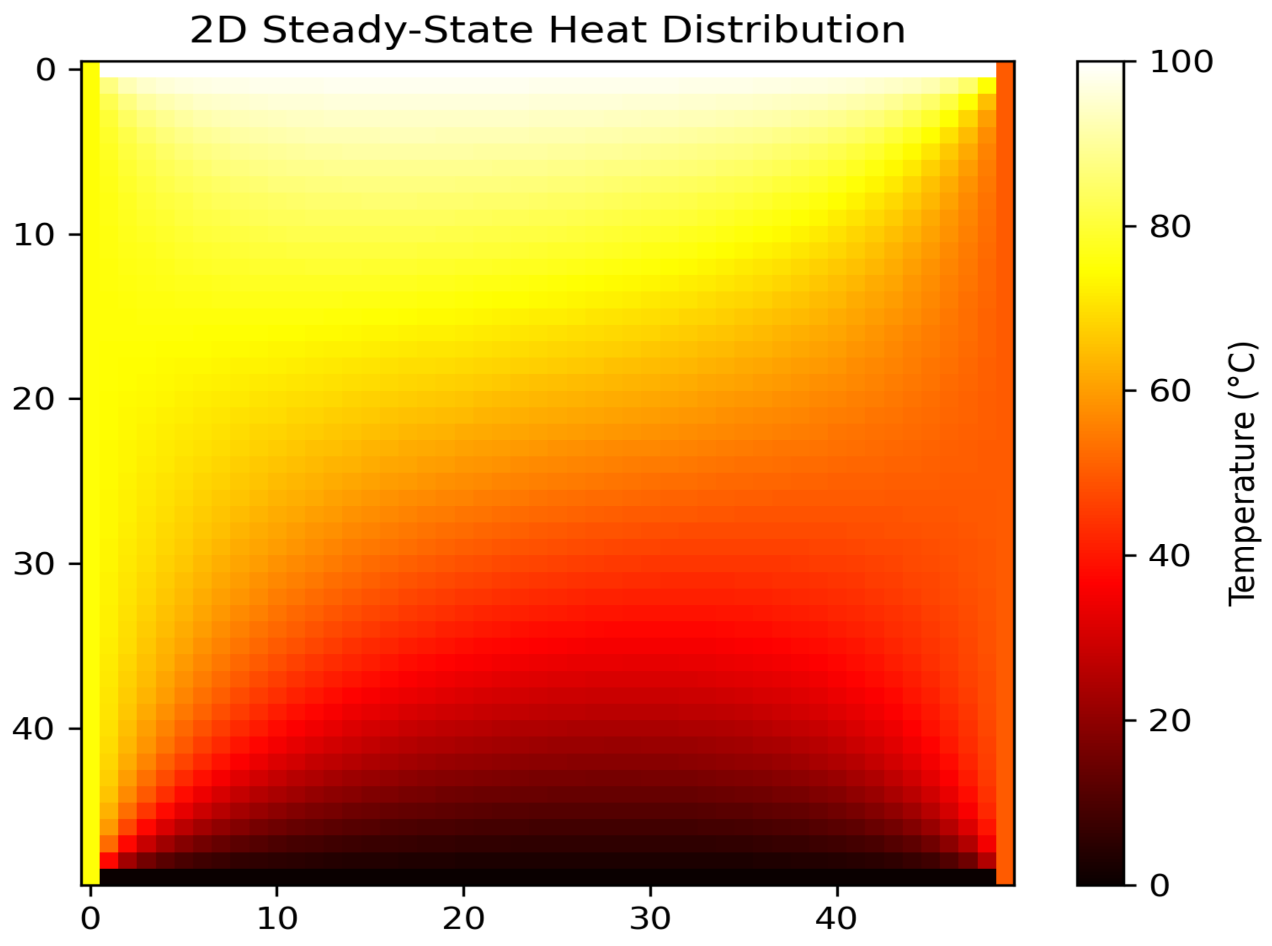
- Problem: Analyze the steady-state heat distribution in a 2D plate using the finite difference method (FDM).
- Python Implementation:

➢Use NumPy for numerical computations.

➢Visualize results with Matplotlib.

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

# Parameters
nx, ny = 50, 50  # Grid size
dx, dy = 1, 1    # Step size
tolerance = 1e-6

# Boundary conditions
T_top, T_bottom, T_left, T_right = 100, 0, 75, 50

# Initialize the grid
T = np.zeros((ny, nx))
T[0, :] = T_top
T[-1, :] = T_bottom
T[:, 0] = T_left
T[:, -1] = T_right
```

```
# Iterative solution (Jacobi method)
error = 1
while error > tolerance:
    T_new = T.copy()
    T_new[1:-1, 1:-1] = 0.25 * (T[1:-1, :-2] + T[1:-1, 2:] + T[:-2, 1:-1] + T[2:,
1:-1])
    error = np.max(np.abs(T_new - T))
    T = T_new

# Visualization
plt.imshow(T, cmap='hot', interpolation='nearest')
plt.colorbar(label='Temperature (°C)')
plt.title('2D Steady-State Heat Distribution')
plt.show()
```


2D Steady-State Heat Distribution
0
10
20
30
40
0
10
20
30
40
100
80
60
40
20
0
Temperature (°C)

## ❑ **Structural Analysis of Beams**

- One of the most common applications of Python in engineering is in the structural analysis of beams. Engineers often face the challenge of calculating shear forces and bending moments in beams subjected to various loads. Using Python, we can automate these calculations, allowing for quicker and more accurate results.
- **Example1:**

➢Insights from the Analysis

✓This code snippet demonstrates how to calculate and visualize the shear force and bending moment along a simply supported beam. By adjusting the parameters, engineers can quickly analyze different loading conditions and beam lengths, enhancing their understanding of structural behavior.

- **Code Snippet for Beam Analysis**

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

# Define beam parameters
length = 10  # length of the beam in meters
load = 5  # uniformly distributed load in kN/m

# Calculate reactions at supports
reaction_A = (load * length) / 2
reaction_B = reaction_A

# Calculate shear force and bending moment
x = np.linspace(0, length, 100)
shear_force = reaction_A - load * x
bending_moment = reaction_A * x - (load * x**2) / 2
```

```
# Plotting the results
plt.figure(figsize=(12, 6))
plt.subplot(2, 1, 1)
plt.plot(x, shear_force, label='Shear Force (kN)')
plt.title('Shear Force Distribution')
plt.xlabel('Length of Beam (m)')
plt.ylabel('Shear Force (kN)')
plt.grid()
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(x, bending_moment, label='Bending Moment (kNm)', color='orange')
plt.title('Bending Moment Distribution')
plt.xlabel('Length of Beam (m)')
plt.ylabel('Bending Moment (kNm)')
plt.grid()
plt.legend()

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

- Example2:

- Problem: Analyze a simply supported beam under a uniform load.
- Python Implementation:

- ✓Use NumPy for matrix operations.
- ✓Use SciPy for solving linear systems.

```
import numpy as np
from scipy.linalg import solve

# Beam properties
L = 10.0  # Length (m)
E = 200e9  # Young's modulus (Pa)
I = 1e-4  # Moment of inertia (m^4)
w = 1000  # Load (N/m)

# Number of elements and nodes
n_elements = 10
n_nodes = n_elements + 1
dx = L / n_elements

# Global stiffness matrix and load vector
K = np.zeros((n_nodes, n_nodes))
F = np.zeros(n_nodes)
```

```
# Assemble matrices
for i in range(n_elements):
    ke = (E * I / dx**3) * np.array([[12, 6*dx, -12, 6*dx],
                                     [6*dx, 4*dx**2, -6*dx, 2*dx**2],
                                     [-12, -6*dx, 12, -6*dx],
                                     [6*dx, 2*dx**2, -6*dx, 4*dx**2]])
    dofs = [i, i+1]
    for a in range(2):
        for b in range(2):
            K[dofs[a], dofs[b]] += ke[a, b]
    F[i:i+2] += w * dx / 2

# Apply boundary conditions (fixed at one end)
K[0, :] = 0
K[:, 0] = 0
K[0, 0] = 1
F[0] = 0

# Solve for displacements
displacements = solve(K, F)
print("Node displacements:", displacements)
```

✓**Output**:

```
Node displacements: [ 0.00000000e+00  2.58838977e-06  1.43096061e-06  1.92904860e-06
  1.75824313e-06  1.71563639e-06  2.00005983e-06  1.28420406e-06
  2.90872933e-06 -7.07482285e-07  7.31122343e-06]
```

## ❑ **Thermal Analysis in Mechanical Systems**

▪ Another area where Python excels is in thermal analysis.

Engineers often need to model heat transfer in mechanical systems, which can be complex and time-consuming. Python libraries such as SciPy and NumPy can be utilized to solve differential equations that describe heat conduction.

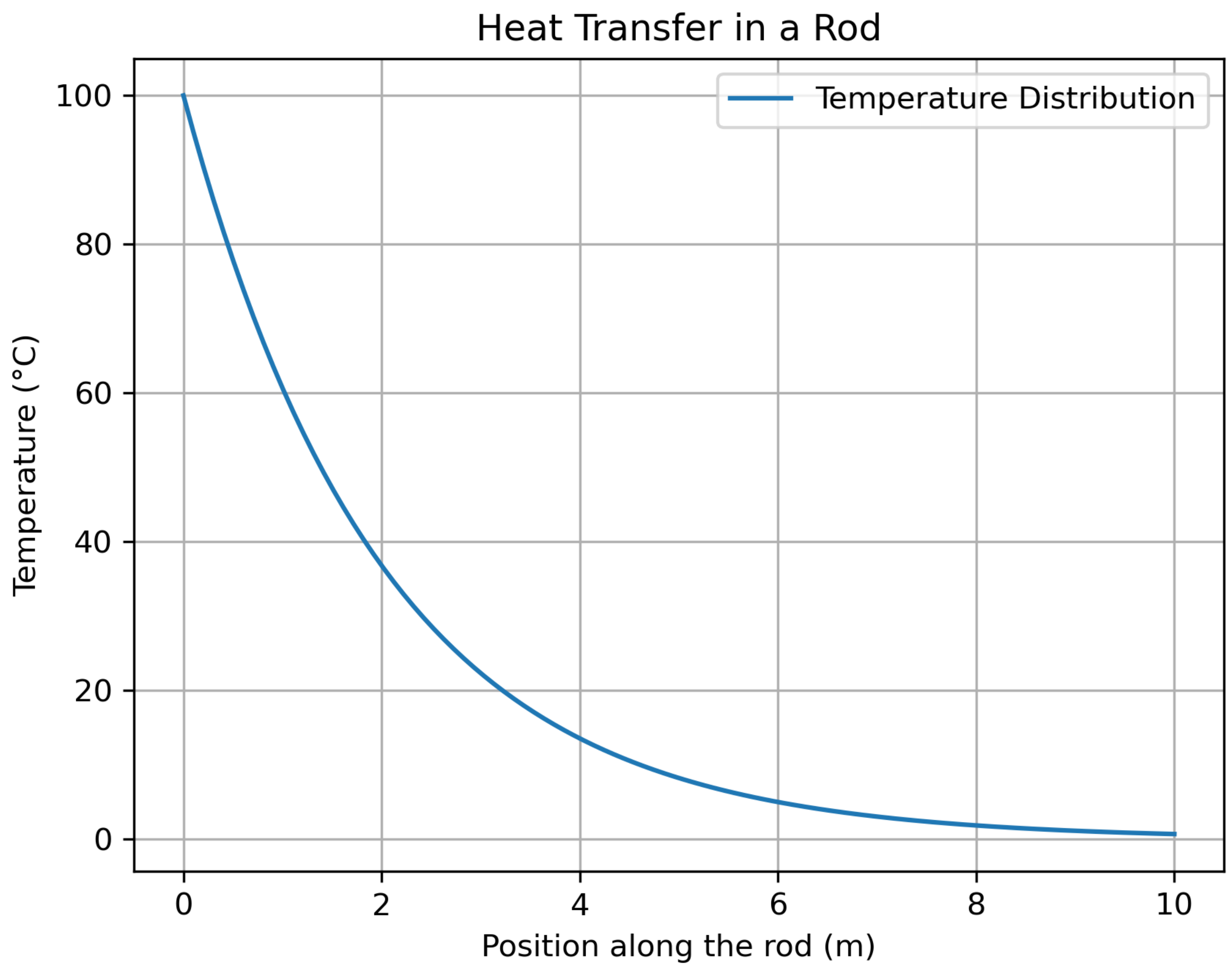
➢Understanding Thermal Dynamics

This example illustrates how to model the temperature distribution in a rod over time. By modifying the thermal properties and initial conditions, engineers can simulate various scenarios, aiding in the design of thermal systems.

- **Code Snippet for Heat Transfer**

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

# Define parameters
L = 10  # length of the rod in meters
T0 = 100  # initial temperature in Celsius
k = 0.5  # thermal conductivity

# Create a grid for the temperature distribution
x = np.linspace(0, L, 100)
T = T0 * np.exp(-k * x)
```

```
# Plotting the temperature distribution
plt.plot(x, T, label='Temperature Distribution')
plt.title('Heat Transfer in a Rod')
plt.xlabel('Position along the rod (m)')
plt.ylabel('Temperature (°C)')
plt.grid()
plt.legend()
plt.show()
```


Heat Transfer in a Rod
Temperature Distribution
Temperature (°C)
Position along the rod (m)
0
20
40
60
80
100
0
2
4
6
8
10

## Fluid Mechanics with Python

- Python is widely used in fluid mechanics to simulate and analyze fluid flow behavior in various engineering problems. This is facilitated by its powerful numerical and visualization libraries and specialized CFD tools. Here’s an overview of how Python can tackle problems in fluid mechanics, accompanied by a basic case study.
- Case Study: Flow Through a 2D Channel

➢Problem: Simulate the steady-state, incompressible flow of a viscous fluid through a 2D channel using the Navier-Stokes equations.

## ❖Python Implementation: Lid-Driven Cavity Flow

- The lid-driven cavity problem is a classic benchmark for simulating 2D incompressible flow. A square cavity is filled with fluid, and the top wall moves at a constant velocity, inducing circulation within the cavity.

- Steps:

➢Discretize the domain using a uniform grid.

➢Solve the Navier-Stokes equations iteratively (using the SIMPLE algorithm or similar).

➢Visualize the velocity and pressure fields.

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

# Parameters
L = 1.0  # Length of cavity (m)
N = 50   # Number of grid points
Re = 100 # Reynolds number
nu = 1 / Re  # Kinematic viscosity

# Grid spacing
dx = L / (N - 1)
dy = L / (N - 1)

# Initialize velocity and pressure fields
u = np.zeros((N, N))  # x-velocity
v = np.zeros((N, N))  # y-velocity
p = np.zeros((N, N))  # Pressure
```

```
# Boundary conditions
u[-1, :] = 1  # Lid moving with velocity 1 (top wall)

# Time-stepping parameters
dt = 0.001  # Time step
nt = 500    # Number of time steps

# Helper function for Laplacian
def laplacian(field):
    return (np.roll(field, 1, axis=0) + np.roll(field, -1, axis=0) +
            np.roll(field, 1, axis=1) + np.roll(field, -1, axis=1) - 4 * field) / dx**2

# Time-stepping loop
for _ in range(nt):
    # Solve for velocity
    un = u.copy()
    vn = v.copy()

    u[1:-1, 1:-1] = (un[1:-1, 1:-1] +
                     dt * (-un[1:-1, 1:-1] * (un[1:-1, 1:-1] - un[1:-1, :-2]) / dx -
                           vn[1:-1, 1:-1] * (un[1:-1, 1:-1] - un[:-2, 1:-1]) / dy +
                           nu * laplacian(un)))
```

```
v[1:-1, 1:-1] = (vn[1:-1, 1:-1] +
                 dt * (-un[1:-1, 1:-1] * (vn[1:-1, 1:-1] - vn[1:-1, :-2]) / dx -
                      vn[1:-1, 1:-1] * (vn[1:-1, 1:-1] - vn[:-2, 1:-1]) / dy +
                      nu * laplacian(vn)))

# Apply boundary conditions
u[:, 0] = u[:, -1] = u[0, :] = v[:, 0] = v[:, -1] = v[0, :] = v[-1, :] = 0
u[-1, :] = 1  # Lid velocity

# Visualization
plt.quiver(u, v)
plt.title("Velocity Field")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

## Solving differential equations with SciPy

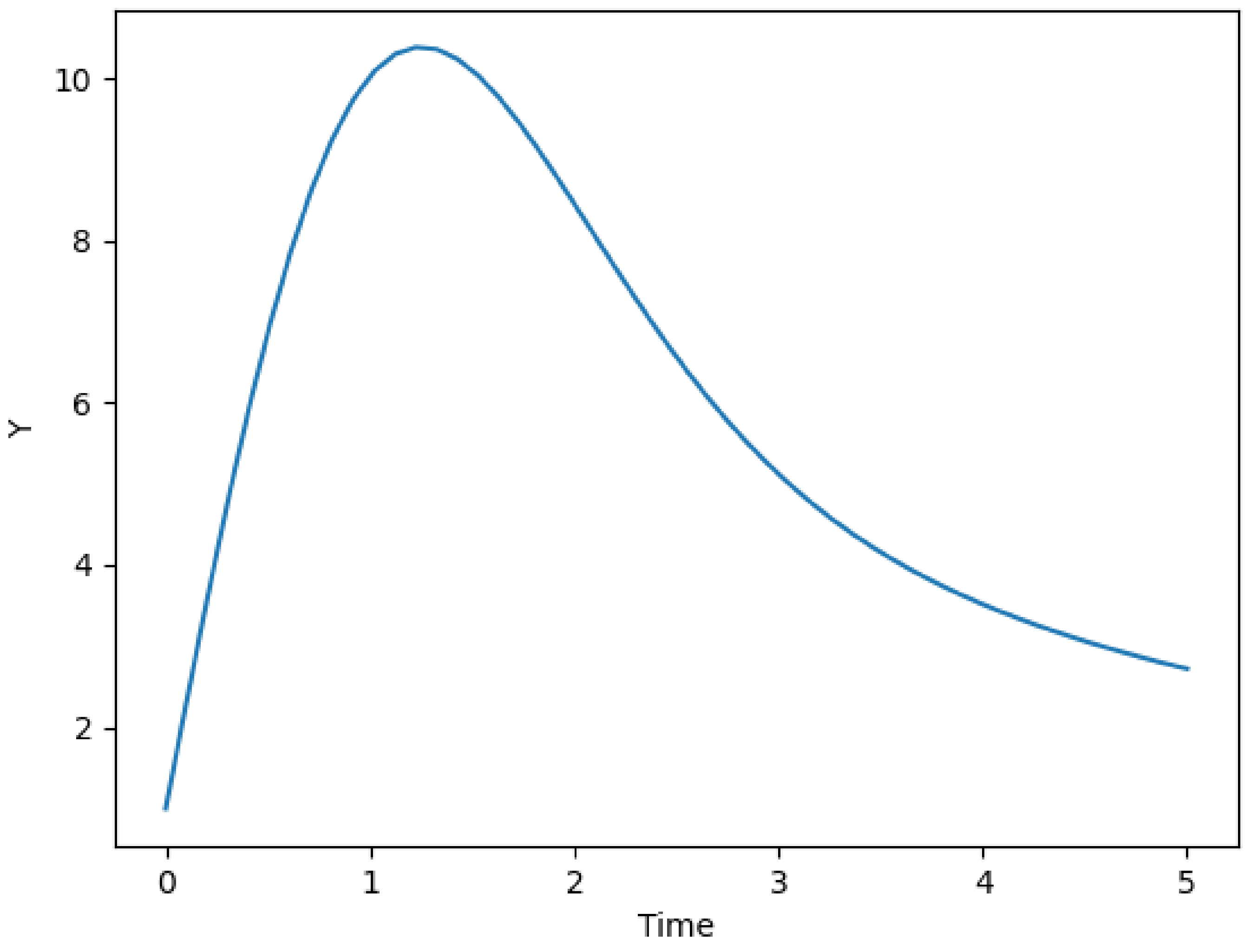
## Solve Differential Equations with ODEINT Function of SciPy module in Python

- ODE stands for Ordinary Differential Equation and refers to those kinds of differential equations that involve derivatives but no partial derivatives. In other words, we only consider one independent variable in these equations.
- Now let us solve some ODE with the help of the odeint function.

➢**Example 1:**

$$\frac{dy}{dt} = -yt + 13$$

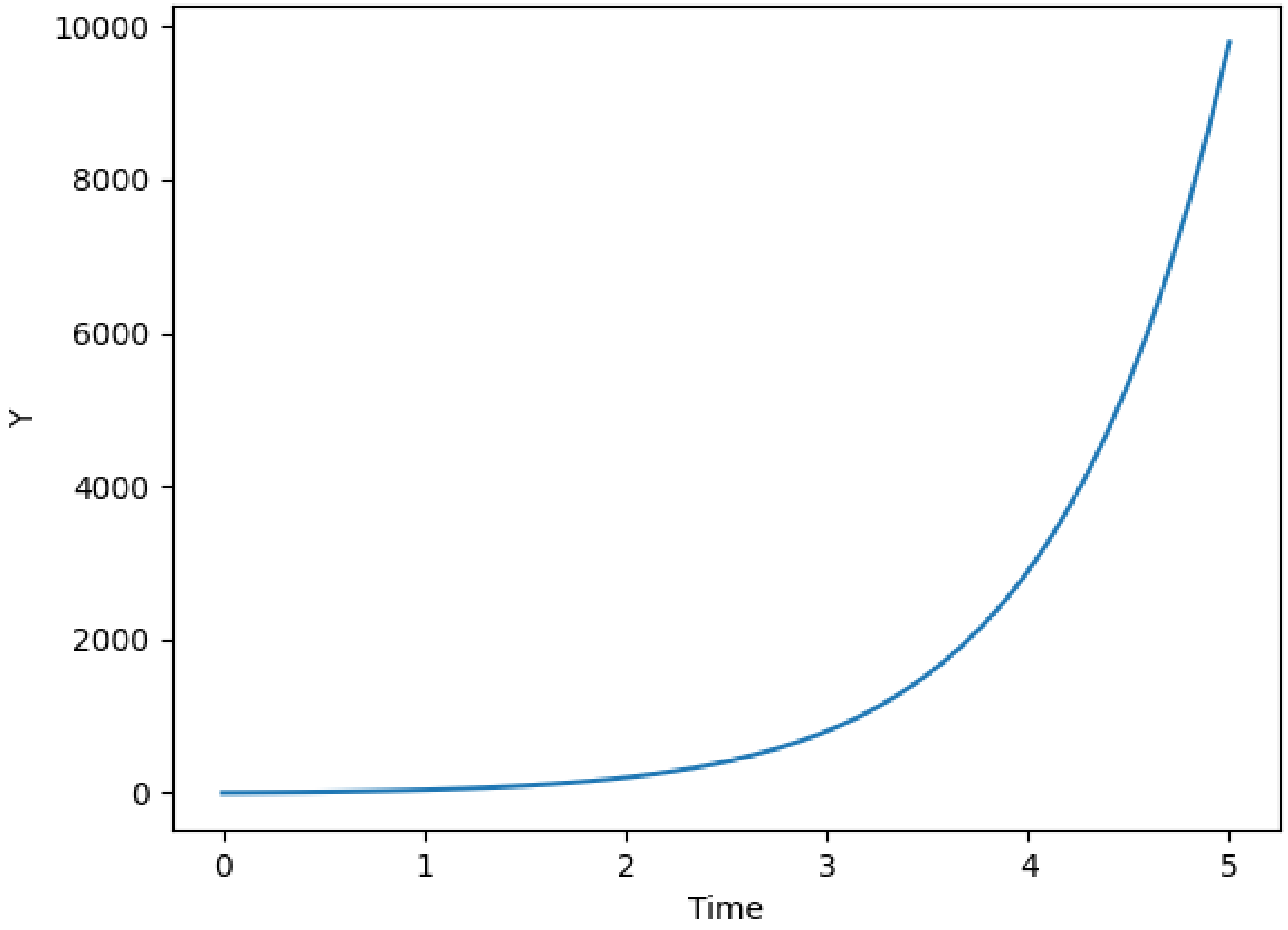
```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

def returns_dydt(y,t):
    dydt = -y * t + 13
    return dydt
# initial condition
y0 = 1
# values of time
t = np.linspace(0,5)
# solving ODE
y = odeint(returns_dydt, y0, t)
# plot results
plt.plot(t,y)
plt.xlabel("Time")
plt.ylabel("Y")
plt.show()
```


10
8
6
4
2
Y
0
1
2
3
4
5
Time

## Example 2:

$$\frac{dy}{dt} = 13\, e^{t} + y$$

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

def returns_dydt(y,t):
    dydt = 13 * np.exp(t) + y
    return dydt
# initial condition
y0 = 1
# values of time
t = np.linspace(0,5)
# solving ODE
y = odeint(returns_dydt, y0, t)
# plot results
plt.plot(t,y)
plt.xlabel("Time")
plt.ylabel("Y")
plt.show()
```



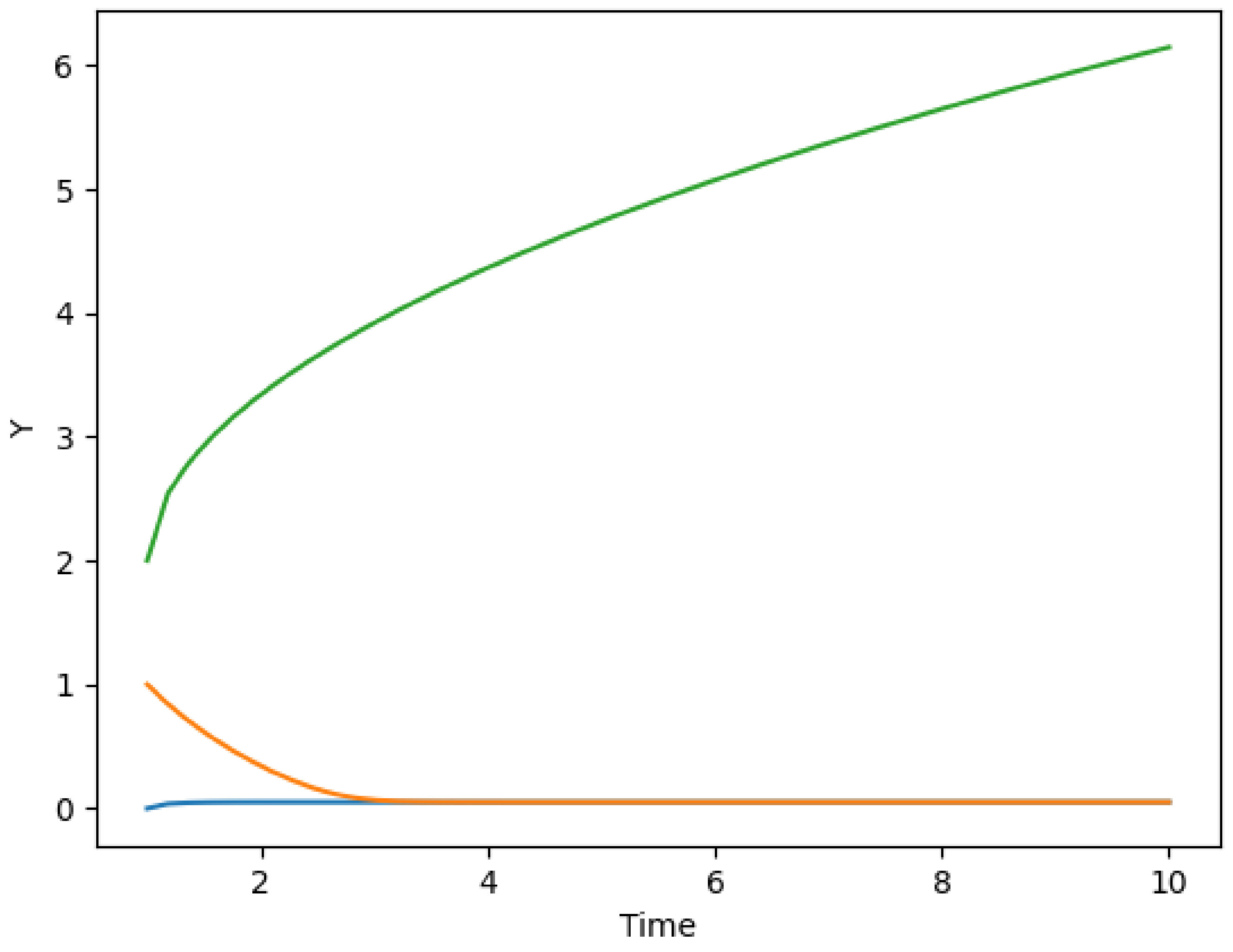
10000
8000
6000
4000
2000
0
Y
0
1
2
3
4
5
Time

## Example 3:

$$\frac{dy}{dt} = \frac{1-y}{1.95-y} - \frac{y}{0.05 + y}$$

- Let us try giving multiple initial conditions by making **y0** an array.

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
def returns_dydt(y,t):
    dydt = (1-y)/(1.95-y) - y/(0.05+y)
    return dydt
# initial conditions
y0 = [0, 1, 2]
# values of time
t = np.linspace(1,10)
# solving ODE
y = odeint(returns_dydt, y0, t)
# plot results
plt.plot(t,y)
plt.xlabel("Time")
plt.ylabel("Y")
plt.show()
```



Y
Time
0
1
2
3
4
5
6
2
4
6
8
10

## SciPy optimization (scipy.optimize)

- The scipy.optimize module is a versatile library in the SciPy ecosystem, offering a wide array of algorithms for both unconstrained and constrained optimization problems. It includes functions for finding the minima of scalar and multi-variable functions, solving root-finding problems, and fitting curves to data.
- Functions in this module include:

- ➢minimize(): A function used to minimize a scalar function of one or more variables.

```
from scipy.optimize import minimize

def objective_function(x):
    return x[0]**2 + x[1]**2
result = minimize(objective_function, [1, 1], method='BFGS')
print(result.x)
```

✓**Output**:

```
# Optimal solution
[-1.07505143e-08 -1.07505143e-08]
```

- root(): Finds the root of a vector function – extremely useful for solving systems of nonlinear equations.

```
from scipy.optimize import root

def equations(vars):
    x, y = vars
    return [x + 2*y - 3, x - y - 1]
result = root(equations, [0, 0])
print(result.x)
```

✓**Output**:

```
[1.66666667 0.66666667]
```

- ➢ curve_fit(): This function fits a curve to a set of data points and is useful for fitting data and parameter estimation.

```
from scipy.optimize import curve_fit
import numpy as np

def model(x, a, b):
    return a * np.exp(b * x)
x_data = np.array([1, 2, 3])
y_data = np.array([2.7, 7.4, 20.1])
params, covariance = curve_fit(model, x_data, y_data)
print(params)
```

✓ **Output**:

```
# Fitted parameters
[0.9981286  1.00089935]
```

## CVXPY

- CVXPY is a Python library designed for convex optimization problems. It enables users to define and solve these problems using a high-level, declarative syntax. It simplifies formulating complex optimization problems by allowing users to specify the objective function and constraints in a natural and readable way.

- CVXPY offers the following features:

➢Declarative syntax: CVXPY provides an intuitive way to model optimization problems using Python's mathematical expressions.

➢State-of-the-art solvers: It interfaces with advanced solvers like ECOS, SCS, and OSQP, which efficiently handle convex problems.

```
import cvxpy as cp
# Define variables
x = cp.Variable()
y = cp.Variable()
# Define constraints
constraints = [x + y == 1, x - y >= 2]
# Define the objective function
objective = cp.Minimize(x**2 + y**2)
# Formulate the problem
prob = cp.Problem(objective, constraints)
# Solve the problem
result = prob.solve()
print(f"Optimal value: {result}")
print(f"x: {x.value}, y: {y.value}")
```

✓ **Output**:

```
Optimal value: 2.5
x: 1.5, y: -0.5000000000000001
```

## Engineering optimization problems

## Engineering optimization problems

- SciPy provides powerful tools for solving engineering optimization problems, allowing engineers to find optimal solutions to complex design, process, or operational problems.

- These tools include functions for constrained, unconstrained, and nonlinear optimization.

➢Unconstrained Optimization

✓In unconstrained optimization, we minimize (or maximize) a function without any constraints.

**Example: Minimize**

```
from scipy.optimize import minimize
# Define the objective function
def objective(x):
    return x[0]**2 + 4*x[0] + 4
# Initial guess
x0 = [0]
# Perform optimization
result = minimize(objective, x0)
# Print results
print("Optimal value of x:", result.x)
print("Minimum value of the function:", result.fun)
```

✓**Output**:

```
Optimal value of x: [-2.00000002]
Minimum value of the function: 0.0
```

- ➢Constrained Optimization
- ✓In constrained optimization, the problem includes constraints such as equality or inequality conditions.

**Example: Minimize**

```
from scipy.optimize import minimize

# Define the objective function
def objective(x):
    return x[0]**2 + x[1]**2

# Define the constraint (equality)
def constraint(x):
    return x[0] + x[1] - 1
```

```
# Constraints dictionary
con = {'type': 'eq', 'fun': constraint}
# Initial guess
x0 = [0, 0]
# Perform optimization
result = minimize(objective, x0, constraints=con)
# Print results
print("Optimal values of x:", result.x)
print("Minimum value of the function:", result.fun)
```

✓**Output**:

```
Optimal values of x: [0.5 0.5]
Minimum value of the function: 0.5
```

➢Nonlinear Optimization

✓Nonlinear optimization is suitable for systems with nonlinear constraints or objective functions.

**Example: Minimize**

```
from scipy.optimize import minimize

# Define the objective function
def objective(x):
    return x[0]**2 + x[1]**2

# Define the constraint (inequality)
def constraint(x):
    return 1 - (x[0]**2 + x[1]**2)
```

```
# Constraints dictionary
con = {'type': 'ineq', 'fun': constraint}

# Initial guess
x0 = [0.5, 0.5]

# Perform optimization
result = minimize(objective, x0, constraints=con)

# Print results
print("Optimal values of x:", result.x)
print("Minimum value of the function:", result.fun)
```

✓ **Output**:

```
Optimal values of x: [-1.11022302e-16 -1.11022302e-16]
Minimum value of the function: 2.465190328815662e-32
```

➢ Optimization with Bounds

✓Bounded optimization restricts variables to specific ranges.

**Example: Minimize**

```
from scipy.optimize import minimize

# Define the objective function
def objective(x):
    return x[0]**2 + 4*x[0] + 4

# Define bounds
bounds = [(0, 2)]
```

```
# Initial guess
x0 = [1]

# Perform optimization
result = minimize(objective, x0, bounds=bounds)

# Print results
print("Optimal value of x:", result.x)
print("Minimum value of the function:", result.fun)
```

✓**Output**:

```
Optimal value of x: [0.]
Minimum value of the function: 4.0
```

- ➢Engineering Application: Designing a Beam
- ✓Problem: Minimize the weight of a beam subject to strength and deflection constraints.

```
from scipy.optimize import minimize

# Parameters
rho = 7850  # Density of steel (kg/m^3)
L = 2       # Length of the beam (m)
M = 1000    # Applied moment (N·m)
sigma_max = 250e6  # Max allowable stress (Pa)
delta_max = 0.005  # Max allowable deflection (m)
E = 210e9   # Modulus of elasticity (Pa)
```

```
# Objective function: Minimize weight
def objective(x):
    b, h = x
    return rho * b * h * L

# Constraints
def stress_constraint(x):
    b, h = x
    sigma = M / (1/6 * b * h**2)
    return sigma_max - sigma

def deflection_constraint(x):
    b, h = x
    delta = (4 * M * L**2) / (E * b * h**3)
    return delta_max - delta
```

```
# Bounds for variables
bounds = [(0.01, 0.1), (0.01, 0.1)]  # Width and height between 1cm and 10cm
# Initial guess
x0 = [0.05, 0.05]
# Constraints dictionary
constraints = [
    {'type': 'ineq', 'fun': stress_constraint},
    {'type': 'ineq', 'fun': deflection_constraint}
]
# Perform optimization
result = minimize(objective, x0, bounds=bounds, constraints=constraints)
# Print results
print("Optimal beam dimensions (b, h):", result.x)
print("Minimum weight of the beam:", result.fun, "kg")
```

✓**Output**:

```
Optimal beam dimensions (b, h): [0.01523812 0.09999978]
Minimum weight of the beam: 23.923789217445258 kg
```