Chapter 24

Characteristic Boundary-Value Problems

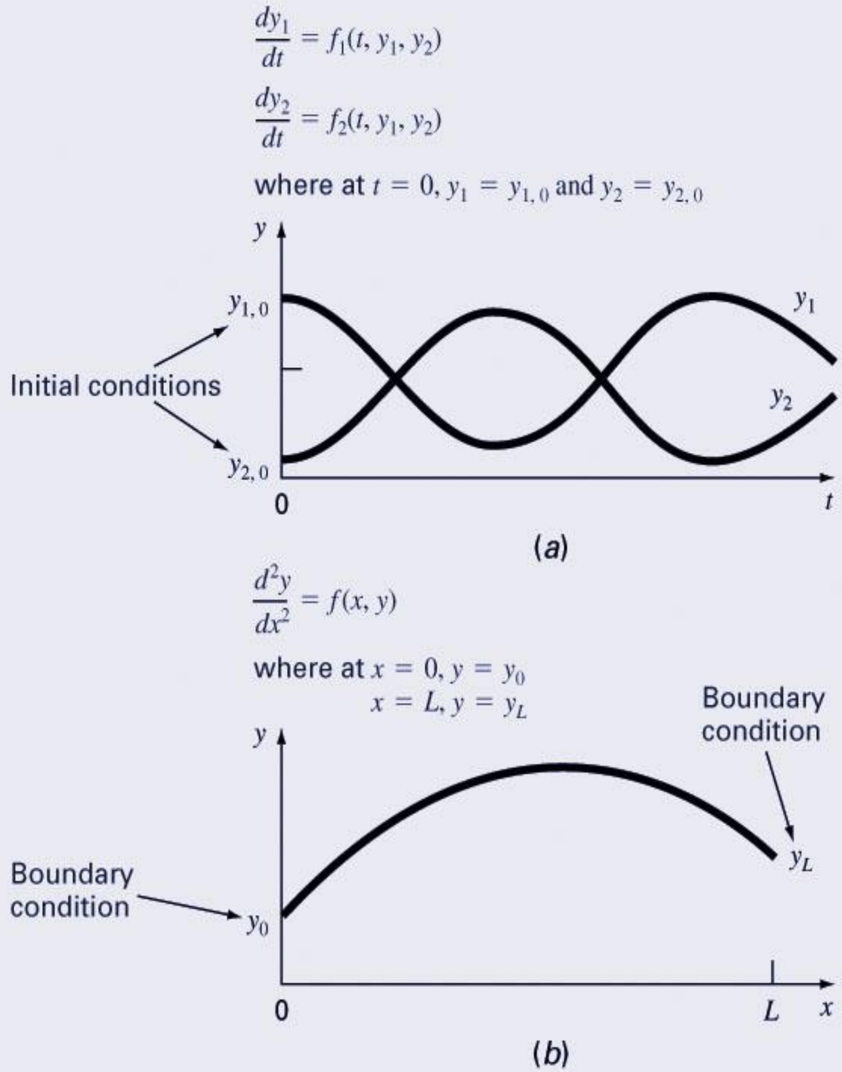
Chapter Objectives

- Understanding the difference between initial-value and boundary-value problems.

- Knowing how to implement the finite-difference method

Boundary-Value Problems

- *Boundary-value problems* are those where conditions are not known at a single point but rather are given at different values of the independent variable.
- Boundary conditions may include values for the variable or values for derivatives of the variable.



Boundary Conditions

- *Dirichlet boundary conditions* are those where a fixed value of a variable is known at a particular location.

- *Neumann boundary conditions* are those where a derivative is known at a particular location.

Finite-Difference Methods

- The simplest numerical methods are finite-difference approaches.

- In these techniques, finite differences are substituted for the derivatives in the original equation, transforming a linear differential equation into a set of simultaneous algebraic equations.

Finite-Difference Example (cont)

- This is a matrix eigenvalue equation which can be diagonalized using the Python function **eig**

Finite Difference: Example

Convert the Schrodinger equation

$$-\frac{d^2y}{dx^2} = k^2y$$

for infinite well of width 1 and boundary conditions $y(0) = 0,\ y(1) = 0$ into a difference equation, using

$$y'' = (y_{i+1} - 2y_i + y_{i-1})/h^2$$

To get

$$-(y_{i+1} - 2y_i + y_{i-1}) = h^2k^2y_i$$

Finite Difference: Example

Consider $x_i = ih, \; i = 0,1,2, \ldots N+1$

The boundary conditions are then

$$y_0 = 0, \qquad y_{N+1} = 0$$

The equations at the internal points are

Finite-Difference (Cont.)

$$i = 1 \qquad -(y_2 - 2y_1 + y_0) = h^2k^2y_1$$
$$i = 2 \qquad -(y_3 - 2y_2 + y_1) = h^2k^2y_2$$
$$\vdots$$
$$i = N \qquad -(y_{N+1} - 2y_N + y_{N-1}) = h^2k^2y_N$$

Using the boundary conditions we have the matrix equation

Finite-Difference: (Cont.)

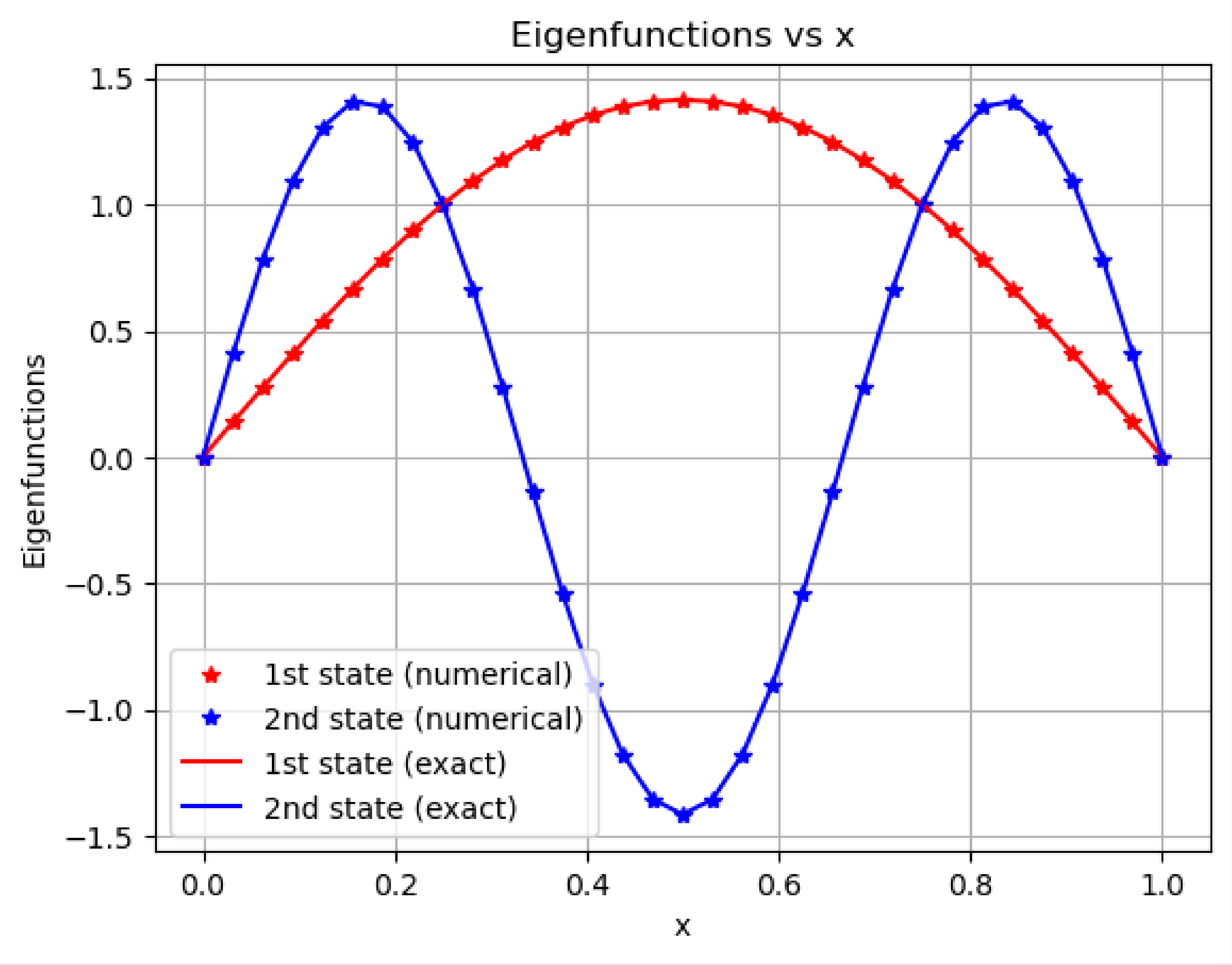
$$\begin{bmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & & & \vdots \\ 0 & -1 & 2 & & & \\ \vdots & & & & & \vdots \\ \vdots & & & & 2 & -1 \\ 0 & 0 & \dots & \cdots & -1 & 2 \end{bmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ \\ \vdots \\ y_N \end{pmatrix} = \lambda \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ \\ \vdots \\ y_N \end{pmatrix}$$

Here $\lambda = h^2k^2$. Note that $h = \frac{1}{N+1}$

```python
import numpy as np

def tridiagmtx(l, d):
    """
    Generate a tridiagonal Hermitian matrix.

    Parameters:
    l : complex or float
        The subdiagonal and superdiagonal value (scalar).
    d : array-like
        The diagonal values (array).

    Returns:
    A : ndarray
        The resulting tridiagonal Hermitian matrix.
    """
    n = len(d)
    D = np.diag(d)  # Diagonal matrix
    L = np.zeros((n, n), dtype=complex)  # Lower triangular part
    U = np.zeros((n, n), dtype=complex)  # Upper triangular part

    for r in range(1, n):
        L[r, r - 1] = l
        U[r - 1, r] = np.conj(l)

    A = L + D + U
    return A
```

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

def tridiagmtx(l, d):
    n = len(d)
    D = np.diag(d)
    L = np.zeros((n, n), dtype=complex)
    U = np.zeros((n, n), dtype=complex)

    for r in range(1, n):
        L[r, r - 1] = l
        U[r - 1, r] = np.conj(l)

    A = L + D + U
    return A
def infinitewell():
    # Initialization
    l = -1
    d = 2 * np.ones(31)  # Change 20 to 31
    n = len(d)
    h = 1 / (n + 1)  # Well width = 1
    # Generate the tridiagonal matrix
    A = tridiagmtx(l, d)
    condition_number = np.linalg.cond(A)
    print("Condition number of A:", condition_number)
    # Eigen decomposition
    eigenvalues, eigenvectors = np.linalg.eigh(A)  # Use `eigh` for Hermitian matrix
    # Normalize eigenfunctions to ensure they match theoretical expectations
    eigenvectors = eigenvectors / np.sqrt(h)
    # Calculate energy values
    E = eigenvalues / h**2
    Ee = (np.arange(1, n + 1)**2) * np.pi**2  # Exact energy values

    # Calculate relative error
    Eror = np.abs(((Ee - E) / Ee) * 100)
    print(np.column_stack((E, Ee, Eror)))
```

```python
    # Choose eigenstates for plotting
    I1, I2 = 1, 3
    x = h * np.arange(0, n + 2)
    s1 = np.sign(eigenvectors[0, I1 - 1])
    s2 = np.sign(eigenvectors[0, I2 - 1])

    # Add boundary values (zero at x=0 and x=1)
    psi1 = np.concatenate(([0], s1 * eigenvectors[:, I1 - 1], [0]))
    psi2 = np.concatenate(([0], s2 * eigenvectors[:, I2 - 1], [0]))

    # Plot numerical eigenfunctions
    plt.plot(x, psi1, 'r*', label='I1 state (numerical)')
    plt.plot(x, psi2, 'b*', label='I2 state (numerical)')

    # Overlay exact eigenfunctions
    psige = np.sqrt(2) * np.sin(np.pi * I1 * x)
    psiee = np.sqrt(2) * np.sin(I2 * np.pi * x)
    plt.plot(x, psige, '-r', label='I1 state (exact)')
    plt.plot(x, psiee, '-b', label='I2 state (exact)')

    # Customize plot
    plt.grid()
    plt.legend()
    plt.title('Eigenfunctions vs x')
    plt.xlabel('x')
    plt.ylabel('Eigenfunctions')
    plt.show()

# Call the function
infinitewell()
```


Eigenfunctions vs x
Eigenfunctions
x
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.2
0.4
0.6
0.8
1.0
1st state (numerical)
2nd state (numerical)
1st state (exact)
2nd state (exact)