SANT LONGOWAL INSTITUTE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF MATHEMATICS

ADVANCED NUMERICAL ANALYSIS LAB

(MDMA-621)

SUBMITTED BY: *SUBMITTED TO:*

INDEX

S.NO.	Experiments	Page no.	Date	signature

Experiment 1: Jacobi Method For Symmetric Matrices

The Jacobi method is an iterative algorithm for determining all eigenvalues of a real symmetric matrix. The method successively applies similarity transformations using orthogonal (rotation) matrices to the original matrix, targeting the largest off-diagonal element in each iteration to zero it out. Over successive iterations, the matrix converges to a diagonal form, where the diagonal elements approximate the eigenvalues of the original matrix.

Given a real symmetric matrix A, the Jacobi method constructs a sequence of orthogonally similar matrices:

$$A^{(k+1)} = J_k^T A^{(k)} J_k$$

where J_k is a Jacobi rotation matrix designed to annihilate the largest off-diagonal element in $A^{(k)}$. The process is repeated until all off-diagonal elements are less than a specified tolerance.

The diagonal elements of the resulting matrix are the eigenvalues of A:

Given the real symmetric matrix

$$A = \begin{bmatrix} 4 & -2 & 2 \\ -2 & 2 & -4 \\ 2 & -4 & 3 \end{bmatrix}$$

Use the Jacobi method to compute all its eigenvalues.

The following code implements the solution:

```
import numpy as np

A = np.array([[4, -2, 2],
              [-2, 2, -4],
              [2, -4, 3]], dtype=float)

n = len(A)
D = A.copy()
max_iter = 100
tol = 1e-10

for k in range(max_iter):
    # Find largest off-diagonal element in D
    max_val = 0.0
    for i in range(n):
        for j in range(i+1, n):
            if abs(D[i][j]) > abs(max_val):
                max_val = D[i][j]
                p = i
                q = j

    # Convergence check
    if abs(max_val) < tol:
        break

    # Compute rotation angle
    if D[p][p] == D[q][q]:
        theta = np.pi / 4
```

```python
    else:
        theta = 0.5 * np.arctan(2 * D[p][q] / (D[q][q] - D[p][p]))

    cos = np.cos(theta)
    sin = np.sin(theta)

    # Create rotation matrix J (only p and q rows/cols change)
    J = np.eye(n)
    J[p][p] = cos
    J[q][q] = cos
    J[p][q] = sin
    J[q][p] = -sin

    # Apply similarity transformation D =  J   D J
    D = J.T @ D @ J

# Eigenvalues are diagonal of D
for i in range(n):
    print(f"Eigenvalue {i+1} =", D[i][i])
```

OUTPUT:

```
Eigenvalue 1 = 2.1777644018132922
Eigenvalue 2 = -1.5379171033705512
Eigenvalue 3 = 8.36015270155726
```

These are the required eigenvalues of the given symmetric matrix A. The Eigenvectors can be determined from the matrix multiplication of all rotation matrices, its columns represent the eigenvectors corresponding to each eigenvalue.

Experiment 2: Givens Method for Symmetric Matrices

Tridiagonalization via Givens Rotations

For a real symmetric matrix A, eigenvalue computations are simplified by first reducing A to a symmetric tridiagonal matrix T through orthogonal similarity transformations. Givens rotations are used to systematically zero out subdiagonal elements while preserving symmetry. Each Givens rotation is an orthogonal matrix that acts only on two rows and columns, annihilating selected entries below the first subdiagonal.

Sturm Sequence and Bisection Method

Once the matrix is in tridiagonal form, the eigenvalues can be efficiently computed using the Sturm sequence property. For a given scalar λ, the Sturm sequence is a recurrence relation whose sign changes count the number of eigenvalues less than λ. The bisection method exploits this property: for each eigenvalue, an interval is repeatedly halved, and the Sturm count determines which subinterval contains the eigenvalue. This process continues until the eigenvalue is isolated within a desired tolerance. Given the real symmetric matrix

$$A = \begin{bmatrix} 4 & -2 & 2 \\ -2 & 2 & -4 \\ 2 & -4 & 3 \end{bmatrix}$$

Perform the following tasks:

1. Reduce A to tridiagonal form T using Givens rotations.
2. Compute all eigenvalues of T (and hence A) using the Sturm sequence property and the bisection method.

The following code implements the method:

```python
import numpy as np

# Input symmetric matrix
A = np.array([[4, -2, 2],
              [-2, 2, -4],
              [2, -4, 3]], dtype=float)

n = len(A)
T = A.copy()

# --- Step 1: Givens rotations to tridiagonal form ---
for j in range(n - 2):  # Loop over column
    for i in range(n - 1, j, -1):  # Eliminate below subdiagonal
        a = T[i - 1][j]
        b = T[i][j]
        if abs(b) < 1e-12:
            continue
        r = np.sqrt(a**2 + b**2)
        c = a / r
        s = -b / r

        # Apply rotation to rows i-1 and i
        for k in range(n):
            tau1 = T[i - 1][k]
            tau2 = T[i][k]
            T[i - 1][k] = c * tau1 - s * tau2
            T[i][k]     = s * tau1 + c * tau2

        # Apply rotation to columns i-1 and i
        for k in range(n):
            tau1 = T[k][i - 1]
            tau2 = T[k][i]
            T[k][i - 1] = c * tau1 - s * tau2
            T[k][i]     = s * tau1 + c * tau2

# Zero small entries to emphasize tridiagonal form
for i in range(n):
    for j in range(n):
        if abs(i - j) > 1 and abs(T[i][j]) < 1e-10:
            T[i][j] = 0.0

print("Tridiagonal matrix:")
print(np.round(T, 6))

# --- Step 2: Sturm sequence count for    ---
def sturm_count(T, lam):
    n = len(T)
    p = np.zeros(n)
    p[0] = T[0][0] - lam
    if p[0] == 0:
        p[0] = -1e-14
    count = 0
    if p[0] < 0:
        count += 1

    for i in range(1, n):
        a = T[i][i] - lam
        b = T[i][i - 1]
        p[i] = a * p[i - 1] - b**2 * (p[i - 2] if i > 1 else 1)
        if p[i] == 0:
            p[i] = -1e-14
        if p[i] * p[i - 1] < 0:
            count += 1

    return count

# --- Step 3: Bisection to find eigenvalues ---
def find_eigenvalues(T, lower, upper, tol=1e-10):
    eigenvalues = []
    for k in range(n):
        a = lower
        b = upper
        while b - a > tol:
            m = (a + b) / 2
            count = sturm_count(T, m)
            if count <= k:
                a = m
            else:
                b = m
        eigenvalues.append((a + b) / 2)
    return eigenvalues

# --- Step 4: Solve and print ---
eigvals = find_eigenvalues(T, lower=-10, upper=10)

print("\nEigenvalues:")
for i, val in enumerate(eigvals):
    print(f"Eigenvalue {i+1} =", val)
```

```
OUTPUT:

Tridiagonal matrix:
[[ 7.5       2.12132  -0.288675]
 [ 2.12132   3.       -0.408248]
 [-0.288675 -0.408248 -1.5     ]]

Eigenvalues:
Eigenvalue 1 = -1.5412184376691584
Eigenvalue 2 = 2.196577990507649
Eigenvalue 3 = 8.344640447139682
```

This is the required solution as per algorithm. Other than sturm sequence, Householder method can also be used to determine the eigenvalues which is much more computationally friendly.

Experiment 3: Successive Over-Relaxation Method

The Successive Over-Relaxation (SOR) method is an iterative technique for solving linear systems of the form $A\mathbf{x} = \mathbf{b}$, where A is a square matrix and $\mathbf{b}$ is a known vector. SOR is an extension of the Gauss-Seidel method, introducing a relaxation factor w ($0 < w < 2$) to potentially accelerate convergence.

In the SOR method, the solution vector $\mathbf{x}$ is updated iteratively as follows:

$$x_i^{(k+1)} = (1-w)x_i^{(k)} + \frac{w}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k+1 \text{ or } k)} \right)$$

where:

- $x_i^{(k+1)}$ is the updated value of the i-th variable at iteration $k+1$,
- w is the relaxation factor,
- a_{ij} are the elements of A,
- b_i is the i-th entry of $\mathbf{b}$.

For $w = 1$, the method reduces to Gauss-Seidel.
For $1 < w < 2$, over-relaxation is used to speed up convergence.

The method converges for strictly diagonally dominant or symmetric positive-definite matrices for suitable w. The process is repeated until the difference between successive iterates is below a specified tolerance. Given the system of linear equations:

$$\begin{cases} 4x_1 - x_2 = 15 \\ -x_1 + 4x_2 - x_3 = 10 \\ -x_2 + 4x_3 = 10 \end{cases}$$

or, equivalently,

$$A\mathbf{x} = \mathbf{b}, \quad \text{where} \quad A = \begin{bmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 15 \\ 10 \\ 10 \end{bmatrix}$$

Use the Successive Over-Relaxation (SOR) method with relaxation factor $w = 1.25$ to find the solution vector $\mathbf{x}$.

The following code implements the solution for the given problem.

```python
import numpy as np

def sor(A, b, w, tol=1e-6, max_iter=100):
    """
    Successive Over-Relaxation (SOR) Method to solve Ax = b
    Parameters:
        A : numpy array : Coefficient matrix
        b : numpy array : Right-hand side vector
        w : float : Relaxation factor (0 < w < 2)
        tol : float : Convergence tolerance
        max_iter : int : Maximum iterations
    Returns:
        x : numpy array : Solution vector
    """
    n = len(b)
    x = np.zeros(n)  # Initial guess (zero vector)

    for k in range(max_iter):
        x_old = x.copy()

        for i in range(n):
            sigma = sum(A[i, j] * x[j] for j in range(n) if j != i)
            x[i] = (1 - w) * x[i] + (w / A[i, i]) * (b[i] - sigma)

        # Check for convergence using infinity norm
        if np.linalg.norm(x - x_old, np.inf) < tol:
            print(f'SOR converged in {k+1} iterations.')
            return x

    print(f'SOR did not converge in {max_iter} iterations.')
    return x

# Example Usage
A = np.array([[4, -1, 0],
              [-1, 4, -1],
              [0, -1, 4]], dtype=float)  # Coefficient matrix

b = np.array([15, 10, 10], dtype=float)  # Right-hand side vector
w = 1.25  # Relaxation factor (1.0 is Gauss-Seidel, 1.25 is typical SOR)

x = sor(A, b, w)
print("Solution:", x)
```

```
OUTPUT:
SOR converged in 14 iterations.
Solution: [4.91071427 4.64285713 3.66071427]
```

This is the required solution of given system of equation.

Experiment 4: Newton-Raphson method

The Newton-Raphson method is an efficient iterative technique for finding solutions to systems of nonlinear equations. For a system of two equations in two variables:

$$\begin{cases} f(x,y) = 0 \\ g(x,y) = 0 \end{cases}$$

the method linearizes the system near a current guess (x_k, y_k) using the Jacobian matrix:

$$J(x,y) = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{bmatrix}$$

At each iteration, the update is given by:

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \end{bmatrix} - J(x_k, y_k)^{-1} \begin{bmatrix} f(x_k, y_k) \\ g(x_k, y_k) \end{bmatrix}$$

The process is repeated until the change in variables is less than a specified tolerance, indicating convergence to a solution. The method requires good initial guesses and the Jacobian to be non-singular at each step. Solve the following system of nonlinear equations using the Newton-Raphson method:

$$\begin{cases} x^2 + xy + y^2 = 7 \\ x^3 + y^3 = 9 \end{cases}$$

Start with the initial guess $x_0 = 1.5, y_0 = 0.5$ and iterate until the solution converges within a tolerance of 10^{-6}.

The following code implements the solution scheme:

```python
import numpy as np
import matplotlib.pyplot as mat
import sympy as sym

x,y = sym.symbols('x,y')
f = x**2+x*y+y**2-7
g = x**3+y**3-9

J = sym.Matrix([[sym.diff(f,x),sym.diff(f,y)],[sym.diff(g,x),sym.diff(g,y)]])

f = sym.lambdify((x, y), f, 'numpy')
g = sym.lambdify((x, y), g, 'numpy')
J_func = sym.lambdify((x, y), J, 'numpy')

def Newton_raphson(x0, y0, max_iteration=1000, tol=1e-6):
    x,y = x0,y0
    for i in range(max_iteration):
        initial_vector = np.array([[x],[y]])
        F = np.array([[f(x,y)],[g(x,y)]])
        J_inv = np.linalg.inv(J_func(x,y))
        solution_vector = initial_vector - J_inv@F
        print(solution_vector)
        if np.linalg.norm(solution_vector - initial_vector,ord=2) < tol:
            print("Convergence achieved")
            return solution_vector
        x,y = solution_vector.flatten()

x0,y0 = 1.5,0.5
solution = Newton_raphson(x0,y0)

print(f"x = {solution[0,0]:.10f}, y = {solution[1,0]:.10f}")
```

```
OUTPUT:

[[2.26754386]
 [0.9254386 ]]
[[2.03727117]
 [0.9644695 ]]
[[2.00125781]
 [0.99873662]]
[[2.00000158]
 [0.99999842]]
[[2.00000001]
 [1.00000002]]
[[2.00000000]
 [1.00000000]]
Convergence achieved
x = 2.0000000000, y = 1.0000000000
```

This is the required solution.

Experiment 5: Iterative Method

The general iteration scheme, also known as the fixed point iteration method, is an iterative technique to solve equations of the form $f(x) = 0$. The equation is first rewritten as $x = g(x)$, where $g(x)$ is the iteration function. Starting from an initial guess x_0, the method generates a sequence using:

$$x_{n+1} = g(x_n)$$

The process is repeated until the difference $|x_{n+1} - x_n|$ is less than a specified tolerance. For the method to converge, the iteration function $g(x)$ should satisfy $|g'(x)| < 1$ in the neighborhood of the root.

Algorithm:

1. Rewrite $f(x) = 0$ as $x = g(x)$.
2. Choose an initial guess x_0.
3. Compute $x_{n+1} = g(x_n)$ iteratively.
4. Stop when $|x_{n+1} - x_n|$ is sufficiently small.

```python
import numpy as np

def f1(x, y):
    return x**3 + y**2 - 4

def f2(x, y):
    return x * y - 1

def jacobian(x, y):
    J = np.array([[2*x, 2*y],
                  [y, x]])
    return J

x0 = 1.0  # initial guess for x
y0 = 10  # initial guess for y

tol = 1e-10
max_iter = 100

for i in range(max_iter):
    F = np.array([f1(x0, y0), f2(x0, y0)])
    J = jacobian(x0, y0)

    # Solve J * delta = -F
    try:
        delta = np.linalg.solve(J, -F)
    except np.linalg.LinAlgError:
        print("Jacobian is singular.")
        break

    x0 += delta[0]
    y0 += delta[1]

    if np.linalg.norm(delta, ord=2) < tol:
        print(f"Converged in {i+1} iterations.")
        break

print("Solution:")
print(f"x = {x0}")
print(f"y = {y0}")
```

```
OUTPUT:
Converged in 10 iterations.
Solution:
x = 0.5084220866051653
y = 1.9668697059901497
```

This is the required solution.

Experiment 6: Gauss-Legendre Method

Gauss-Legendre quadrature is a highly accurate numerical integration technique for approximating definite integrals of the form:

$$I = \int_a^b f(x)\,dx$$

It is based on evaluating the integrand at specific points (called nodes) and weighting these values appropriately. For the n-point Gauss-Legendre quadrature, the integral is approximated as:

$$I \approx \sum_{i=1}^{n} w_i f(x_i)$$

where x_i are the roots of the n-th Legendre polynomial mapped to $[a, b]$, and w_i are the corresponding weights. To apply Gauss-Legendre quadrature on an interval $[a, b]$, a change of variables is used to map the interval to $[-1, 1]$:

$$x = \frac{b-a}{2}t + \frac{b+a}{2}$$

where $t \in [-1, 1]$. The integral then becomes:

$$I = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b-a}{2}t + \frac{b+a}{2}\right) dt$$

The method is exact for all polynomials of degree up to $2n-1$.

Common Gauss-Legendre Quadrature Rules:

- **1-point:** $t_1 = 0$, $w_1 = 2$
- **2-point:** $t_{1,2} = \pm 1/\sqrt{3}$, $w_{1,2} = 1$
- **3-point:** $t_{1,2,3} = 0, \pm\sqrt{3/5}$, $w_{1,3} = 5/9$, $w_2 = 8/9$

Evaluate the integral

$$I = \int_1^2 \frac{2x}{1+x^4}\,dx$$

using the Gauss-Legendre quadrature method with 1-point, 2-point, and 3-point formulas. Compare the results with the exact value of the integral. The following code is the implementation of the solution:

```python
import numpy as np

a,b =1,2

def f(xk):
    t=xk
    #Transformation from [1,2] to [-1,1]
    x=((b-a)/2)*t + ((b+a)/2)
    f=2*x/(1+pow(x,4))
    return f
```

```

#Jacobian of mapping from [1,2] to [-1,1]
J = (b-a)/2
#1 point formula
I1 = J*2*f(0)
print("I1=",I1)

#2 Point Formula
I2 = J*(f(-1/np.sqrt(3)) + f(1/np.sqrt(3)))
print("\nI2=",I2)

#3 point Formula
I3 = J*(1/9*(5*f(-np.sqrt(3/5)) + 8*f(0) + 5*f(np.sqrt(3/5))))
print("\nI3=",I3)

#exact solution
I =np.arctan(4)- np.pi/4
print("\nI=",I)
```

```
OUTPUT:
I1= 0.4948453608247423

I2= 0.5433755145601464

I3= 0.5405910903505368

I= 0.5404195002705843
```

The definite integral converges this solution which matches well with the exact solution with varying degree of errors. The most closest one is obviously three-point formula.

Experiment 7: Romberg Integration

Romberg integration is a powerful numerical technique for approximating definite integrals, combining the trapezoidal rule with Richardson extrapolation to achieve high accuracy. The method begins by applying the composite trapezoidal rule with increasingly finer subintervals to estimate the integral:

$$I \approx T(h) = \frac{h}{2}\left[f(a) + 2\sum_{k=1}^{N-1} f(a+kh) + f(b)\right]$$

where h is the step size and N is the number of subintervals.

Romberg integration constructs a triangular table $R_{i,j}$, where:

- $R_{i,0}$ is the trapezoidal rule estimate with 2^i subintervals,
- Higher-order estimates are computed recursively by Richardson extrapolation:

$$R_{i,j} = \frac{4^j R_{i,j-1} - R_{i-1,j-1}}{4^j - 1}$$

Each step increases the order of accuracy, and the value in the lower right corner of the table provides a highly accurate approximation of the integral. Use the Romberg integration method to approximate the value of the integral

$$I = \int_0^{\pi} \left(\sin(x) - 4x^7 + 5\sqrt{45x^5}\right) dx$$

Construct the Romberg table up to four levels ($n = 4$) and report the most accurate approximation obtained. The following code is the implementation of above algorithm:

```python
import numpy as np

def f(x):
    return np.sin(x)-4*x**7+5*np.sqrt(45*x**5)

a, b = 0, np.pi
n = 4
R = np.zeros((n, n))

for i in range(n):
    N = 2**i
    h = (b - a) / N
    x = np.linspace(a, b, N + 1)
    y = f(x)
    R[i, 0] = h * (0.5 * y[0] + y[1:-1].sum() + 0.5 * y[-1])

for j in range(1, n):
    for i in range(j, n):
        R[i, j] = (4**j * R[i, j-1] - R[i-1, j-1]) / (4**j - 1)

# Set printing options for consistent formatting
np.set_printoptions(precision=10, floatmode='fixed', suppress=False)

print("Romberg Integration Table:")
print(R)
print("\nApproximated integral =", R[n-1, n-1])
```

```
OUTPUT:
Romberg Integration Table:
[[-18055.4       0.0000000000      0.0000000000      0]
 [ -9011.4  -5996.8132359779      0.0000000000      0]
 [ -5532.8  -4373.3468400092  -4265.1157469446      0]
 [ -4552.8  -4226.1932270914  -4216.3829862302  -4215.6]]

Approximated integral = -4215.609450345814
```

This is the required numerical solution of the given definite integral, the last term is last column gives that value which is converged through richardson extrapolation.

Experiment 8: Milne-Simpson Predictor Corrector Method

Milne's method is a multistep predictor-corrector technique for numerically solving ordinary differential equations (ODEs) of the form

$$\frac{dy}{dx} = f(x, y), \quad y(x_0) = y_0.$$

It uses previously computed values to predict and then iteratively correct the next value of y.

Predictor (Milne's formula):

$$y_{n+1}^{(p)} = y_{n-3} + \frac{4h}{3}\left[2f_n - f_{n-1} + 2f_{n-2}\right]$$

where h is the step size, $f_k = f(x_k, y_k)$, and $y_{n+1}^{(p)}$ is the predicted value.

Corrector (Milne-Simpson formula):

$$y_{n+1}^{(c)} = y_{n-1} + \frac{h}{3}\left[f_{n+1}^{(p)} + 4f_n + f_{n-1}\right]$$

where $f_{n+1}^{(p)} = f(x_{n+1}, y_{n+1}^{(p)})$.

The corrector step can be repeated iteratively until the difference between successive corrected values is within a desired tolerance. Since Milne's method is a multistep method, it requires initial values at several points, which are typically obtained using a single-step method such as Euler's or Runge-Kutta.

Use Milne's predictor-corrector method to numerically solve the initial value problem:

$$\frac{dy}{dx} = x^3 + yx, \quad y(0) = 1$$

with a step size $h = 0.1$. Use Euler's method to generate the first three values, and then apply Milne's method to compute subsequent values.

```python
import numpy as np

def f(x, y):
    return x**3 + y * x

# Initial condition
x0 = 0
y0 = 1
h = 0.1
y_values = []

# Step 1: Generate first 4 values using Euler or RK4 (you used Euler here)
for i in range(3):  # This gives y1, y2, y3
    y_next = y0 + h * f(x0, y0)
    y_values.append(y_next)
    y0 = y_next
    x0 = x0 + h

# Unpack to named variables
```

```python
y1, y2, y3 = np.array(y_values).flatten()
x1 = 0.1
x2 = 0.2
x3 = 0.3

j = 0
for j in range(6):
    x4 = x3 + h

    # Milne Predictor
    y_p = y0 + (4 * h / 3) * (2*f(x3, y3) - f(x2, y2) + 2*f(x1, y1))

    # Milne Corrector (iterative)
    for i in range(100):
        y_c = y2 + (h / 3) * (f(x4, y_p) + 4*f(x3, y3) + f(x2, y2))
        if abs(y_c - y_p) < 1e-13:
            print(f"Final Corrected value for x{j+4}:", y_c)
            break
        y_p = y_c

    # Shift values for next step
    y0 = y1
    y1 = y2
    y2 = y3
    y3 = y_c

    x1 = x2
    x2 = x3
    x3 = x4
```

```
OUTPUT:
Final Corrected value for x4: 1.0784575135135135
Final Corrected value for x5: 1.1313872075125975
Final Corrected value for x6: 1.2186354702665299
Final Corrected value for x7: 1.3230046060673684
Final Corrected value for x8: 1.475844461888508
Final Corrected value for x9: 1.6652557967117336
```

This the required solution of the given initial value problem.

Experiment 9: Adam-Bashforth Method

The Adams-Bashforth-Moulton method is a multistep predictor-corrector scheme for numerically solving ordinary differential equations (ODEs) of the form

$$\frac{dy}{dx} = f(x, y), \qquad y(x_0) = y_0.$$

It uses previously computed points to predict and then correct the next value of y.

Predictor (Adams-Bashforth 4-step explicit formula):

$$y^{(p)}_{n+1} = y_n + \frac{h}{24}\left[55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3}\right]$$

where $f_k = f(x_k, y_k)$ and h is the step size.

Corrector (Adams-Moulton 4-step implicit formula):

$$y^{(c)}_{n+1} = y_n + \frac{h}{24}\left[9f^{(p)}_{n+1} + 19f_n + 5f_{n-1} + f_{n-2}\right]$$

where $f^{(p)}_{n+1} = f(x_{n+1}, y^{(p)}_{n+1})$.

The corrector step may be repeated iteratively until the difference between successive corrected values is within a specified tolerance.

Since this is a multistep method, initial values at several points are required. These are typically generated using a single-step method such as Euler's or Runge-Kutta.

Use the Adams-Bashforth-Moulton predictor-corrector method to numerically solve the initial value problem:

$$\frac{dy}{dx} = x^3 + yx, \qquad y(0) = 1$$

with a step size $h = 0.1$. Use Euler's method to compute the first three values, then apply the Adams-Bashforth 4-step predictor and Adams-Moulton 4-step corrector to compute subsequent values.
This is the following code to implement the scheme:

```python
import numpy as np

# Define the differential equation dy/dx = f(x, y)
def f(x, y):
    f = x**3 + y * x
    return f

# Initialize storage for the first few y-values
y_values = []

# Initial conditions
x0 = 0
y0 = 1
h = 0.1  # Step size

# Generate y1, y2, y3 using simple Euler's method
for i in range(3):
    y_next = y0 + h * f(x0, y0)  # Euler step
    y_values.append(y_next)      # Store the computed y value
    y0 = y_next                  # Update y0 for next iteration
    x0 = x0 + h                  # Move to next x value

# Unpack initial values for Adams-Bashforth-Moulton method
y1, y2, y3 = np.array(y_values).flatten()

# Predictor-Corrector loop (6 additional steps)
for j in range(6):
    # Predictor: Adams-Bashforth 4-step explicit formula
    y_p = y3 + (h / 24) * (
        55 * f(x0 + (j + 3) * h, y3)
        - 59 * f(x0 + (j + 2) * h, y2)
        + 37 * f(x0 + (j + 1) * h, y1)
        - 9  * f(x0 + j * h, y0)
    )

    # Corrector loop: Adams-Moulton 4-step implicit formula
    for i in range(100):
        y_c = y3 + (h / 24) * (
            9  * f(x0 + (j + 4) * h, y_p)
            + 19 * f(x0 + (j + 3) * h, y3)
            - 5  * f(x0 + (j + 2) * h, y2)
            +      f(x0 + (j + 1) * h, y1)
        )

        # Print intermediate corrected value
        print(y_c)

        # Convergence check
        if abs(y_c - y_p) < 1e-13:
            print(f"Final Corrected value for x{j + 4}:", y_c)
            break

        # Update predictor with corrected value
        y_p = y_c

    # Shift y-values for next step (advance the solution window)
    y3 = y_c
    y2 = y3
    y1 = y2
    y0 = y1
```

```
OUTPUT:
Final Corrected value for x4: 1.1284715139067174
Final Corrected value for x5: 1.2594100330232136
Final Corrected value for x6: 1.4339765042335348
Final Corrected value for x7: 1.6648365228332072
Final Corrected value for x8: 1.968182374095253
Final Corrected value for x9: 2.3647436023894453
```

This is the required solution of the initial value problem.

Experiment 10: Explicit Finite Difference Scheme

The one-dimensional heat equation is given by

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

where $u(x,t)$ is the temperature at position x and time t.

The finite difference method discretizes both time and space. The spatial domain $[a,b]$ is divided into n grid points with spacing h, and time is advanced in steps of size Δt.

For the explicit scheme, the second spatial derivative is approximated by the central difference:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i-1}^n - 2u_i^n + u_{i+1}^n}{h^2}$$

and the time derivative is approximated by the forward difference:

$$\frac{\partial u}{\partial t} \approx \frac{u_i^{n+1} - u_i^n}{\Delta t}$$

The update formula for the interior grid points is:

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

where $r = \frac{\Delta t}{h^2}$.

Boundary conditions (Dirichlet) are applied by fixing u_0 and u_{n-1} at each time step.

This method is conditionally stable; for stability, $r \leq \frac{1}{2}$ is required.
Solve the one-dimensional heat equation

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

on the domain $x \in [0,1]$ with the initial condition

$$u(x,0) = \begin{cases} 1 + 2x, & 0 \leq x < 0.5 \\ 3 - 2x, & 0.5 \leq x \leq 1 \end{cases}$$

and Dirichlet boundary conditions $u(0,t) = u(1,t) = 0$. Use the finite difference method with $n = 6$ spatial grid points and time step $\Delta t = 0.02$ to compute the solution for two time steps. Plot the resulting temperature profile $u(x,t)$. The following code solves the given heat equation.

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

n = 6                    # number of spatial grid points
a, b = 0, 1
dt = 0.02                # time step
```

```python
t_final = 2 * dt            # run for 2 time steps
h = (b - a) / (n - 1)
x = np.linspace(a, b, n)

# Initial condition (piecewise linear)
u = np.zeros(n)
u[:n//2] = 1 + 2 * x[:n//2]
u[n//2:] = 3 - 2 * x[n//2:]

r = dt / h**2

# Construct A matrix (implicit time integration)
A = np.diag((1 - 2*r) * np.ones(n - 2)) + \
    np.diag(r * np.ones(n - 3), 1) + \
    np.diag(r * np.ones(n - 3), -1)

# Time stepping
t = 0
while t < t_final:
    t += dt
    v = u[1:-1]             # interior values
    R = A @ v
    R[0] += r * u[0]        # apply BC on left
    R[-1] += r * u[-1]      # apply BC on right
    u[1:-1] = R             # update solution (Dirichlet BCs: u[0], u[-1] stay 0)

# Plot result
plt.plot(x, u, 'o-', label='u(x,t)')
plt.xlabel("x")
plt.ylabel("u")
plt.title("1D Heat Equation (Explicit, 2 Steps)")
plt.grid(True)
plt.legend()
plt.show()
```

OUTPUT:

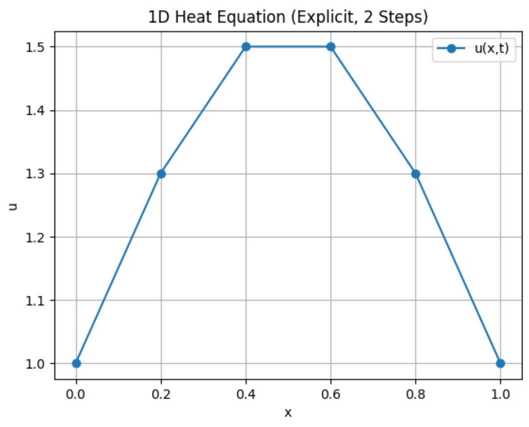

Figure 1: Numerical Solution of Heat Equation

Experiment 11: Five-Point Formula for solving Elliptic PDEs

The Laplace equation in two dimensions is given by

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

on a rectangular domain.

The finite difference method approximates the derivatives using values at grid points. Two common stencils are:

Standard 5-Point Stencil:

$$u_{i,j} = \frac{1}{4}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right)$$

This uses the four immediate neighbors (up, down, left, right) to update the value at (i, j).

Diagonal (9-Point) Stencil:

$$u_{i,j} = \frac{1}{20}\left[4(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1})......\right]$$

$$\left[..... + \left(u_{i+1,j+1} + u_{i+1,j-1} + u_{i-1,j+1} + u_{i-1,j-1}\right)\right]$$

This includes the four diagonal neighbors in addition to the standard four, providing a higher-order approximation.

Both methods iteratively update the interior points of the grid, holding the boundary values fixed (Dirichlet conditions). The process is repeated until the solution converges or for a fixed number of iterations.
Solve the two-dimensional Laplace equation

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

on a square domain with Dirichlet boundary conditions such that $u = 100$ on the top edge and $u = 0$ on the other edges.

Use both the standard 5-point finite difference method and the diagonal (9-point) method to compute the steady-state temperature distribution on a 100×100 grid. Compare the results by plotting the computed solutions. The following code implements the scheme.

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

N = 100 # grid size
u1 = np.zeros((N, N))  # standard 5-point
u2 = np.zeros((N, N))  # diagonal 5-point
max_iter = 500

# Apply non-zero boundary condition on top edge
u1[:, -1] = 100
```

```python
u2[:, -1] = 100

# Standard 5-point method
for _ in range(max_iter):
    for i in range(1, N-1):
        for j in range(1, N-1):
            u1[i, j] = 0.25 * (u1[i+1, j] + u1[i-1, j] + u1[i, j+1] + u1[i, j-1])

# Diagonal (9-point) method
for _ in range(max_iter):
    for i in range(1, N-1):
        for j in range(1, N-1):
            u2[i, j] = (1/20) * (
                4*(u2[i+1, j] + u2[i-1, j] + u2[i, j+1] + u2[i, j-1]) +
                (u2[i+1, j+1] + u2[i+1, j-1] + u2[i-1, j+1] + u2[i-1, j-1])
            )

# Plot both
plt.subplot(1, 2, 1)
plt.imshow(u1, cmap='hot')
plt.title("Standard 5-Point")

plt.subplot(1, 2, 2)
plt.imshow(u2, cmap='hot')
plt.title("Diagonal 5-Point")

plt.show()
```

OUTPUT:
This is the required heat map

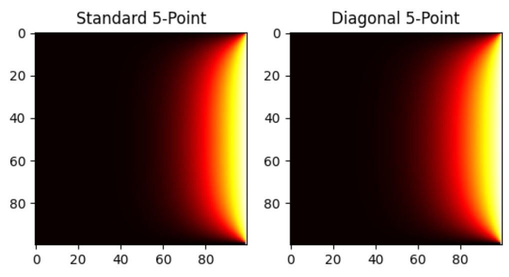

Figure 2: Required Heat Map

Experiment 12: Solving Parabolic PDEs using Crank-Nicolson Scheme

The Crank-Nicolson method is an implicit, second-order accurate finite difference scheme for solving parabolic partial differential equations such as the 1D heat equation:

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

on a domain $x \in [a, b]$, with given initial and boundary conditions.

The spatial domain is discretized into n grid points with spacing h, and time is advanced in steps of size Δt. The Crank-Nicolson scheme averages the explicit and implicit (forward and backward Euler) methods, leading to the update formula for interior points:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{1}{2}\left[\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h^2} + \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{h^2}\right]$$

This results in a tridiagonal system of equations for the new time level, which must be solved at each time step.

The Crank-Nicolson method is unconditionally stable and second-order accurate in both space and time.

Solve the 1D heat equation

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

for $x \in [0.5, 1.5]$ and $t \in [0, 0.25]$, with initial condition

$$u(x, 0) = \sin(\pi x)$$

and boundary conditions

$$u(0.5, t) = e^{-\pi^2 t}\sin(\pi \cdot 0.5), \qquad u(1.5, t) = e^{-\pi^2 t}\sin(\pi \cdot 1.5).$$

Use the Crank-Nicolson method with $n = 41$ spatial grid points and time step $\Delta t = 0.0001$. Plot the numerical and exact solutions at $t = 0.25$, and report the maximum error. The following is the required implementation of scheme:

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

# Parameters
n = 41   # number of discretized points
a = 0.5
b = 1.5
dt = 0.0001
t = 0
h = (b - a) / (n - 1)   # space step

# Initialize vectors and matrices
x = np.zeros(n)
u = np.zeros(n)
R = np.zeros(n-2)
v = np.zeros(n-2)
```

```python
A = np.zeros((n-2, n-2))
B = np.zeros((n-2, n-2))
er = np.zeros(n)
ex = np.zeros(n)

# x array
for i in range(n):
    x[i] = a + (i-1) * h

# Initial condition for u
for i in range(n):
    u[i] = np.sin(np.pi * x[i])

# Calculate the coefficient r
r = dt / (h ** 2)

# Set up the matrix A
A[0, 0] = 2 * (1 + r)
A[0, 1] = -r
for i in range(1, n-3):
    A[i, i] = 2 * (1 + r)
    A[i, i-1] = -r
    A[i, i+1] = -r
A[n-3, n-4] = -r
A[n-3, n-3] = 2 * (1 + r)

# Set up the matrix B
B[0, 0] = 2 * (1 - r)
B[0, 1] = r
for i in range(1, n-3):
    B[i, i] = 2 * (1 - r)
    B[i, i-1] = r
    B[i, i+1] = r
B[n-3, n-4] = r
B[n-3, n-3] = 2 * (1 - r)

# Time-stepping loop
count = 0
while t < 0.25:
    # Step 1: Store u values for the next time step
    v = u[1:n-1]

    # Step 2: Compute the right-hand side vector R
    R = np.dot(B, v)
    R[0] += r * (np.exp(-np.pi**2 * t) * np.sin(np.pi * a) + np.exp(-np.pi**2 * (t + dt)
    ) * np.sin(np.pi * a))
    R[n-3] += r * (np.exp(-np.pi**2 * t) * np.sin(np.pi * b) + np.exp(-np.pi**2 * (t +
    dt)) * np.sin(np.pi * b))

    # Step 3: Solve for R1 using matrix A
    R1 = np.linalg.solve(A, R)

    # Step 4: Update the solution u
    u[1:n-1] = R1
    u[0] = np.exp(-np.pi**2 * t) * np.sin(np.pi * a)
    u[n-1] = np.exp(-np.pi**2 * t) * np.sin(np.pi * b)

    # Increment time
    t += dt
    count += 1

    # Optional: Display progress every 10 iterations
    # if count % 10 == 0:
    #     print(f"Iteration {count}, Time {t:.4f}")

# Calculate error and exact solution
for i in range(n):
    er[i] = u[i] - np.exp(-np.pi**2 * t) * np.sin(np.pi * x[i])
    ex[i] = np.exp(-np.pi**2 * t) * np.sin(np.pi * x[i])

# Find the maximum error
e1 = np.max(np.abs(er))

```

```
# Plot the results
plt.plot(x, ex, label="Exact Solution")
plt.plot(x, u, 'o', label="Numerical Solution")
plt.legend()
plt.show()
```

OUTPUT:

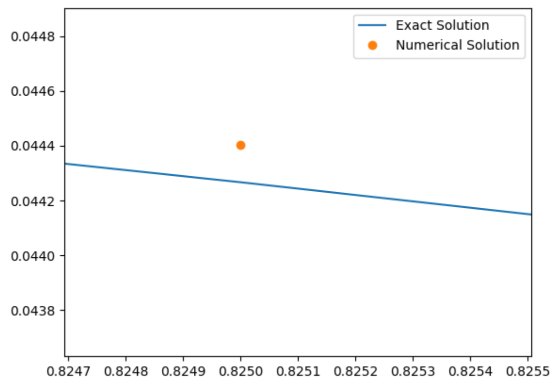

Figure 3: Error between exact and numerical solution

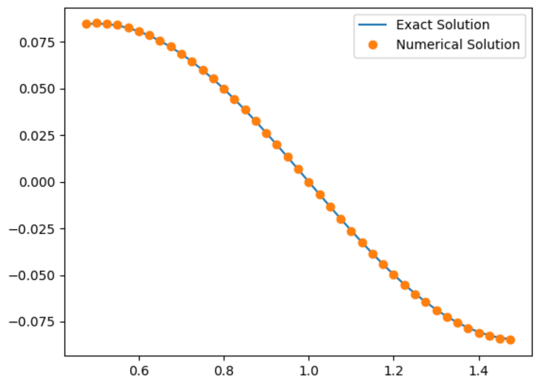

Figure 4: Numerical and Exact Solution

Experiment 13: Solving Parabolic PDEs using Dufort Scheme

The DuFort-Frankel scheme is an explicit finite difference method used to solve the one-dimensional heat equation:

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

where $u(x,t)$ is the temperature, α is the thermal diffusivity, and the domain is $x \in [0, L]$.

The spatial domain is discretized into n grid points with spacing Δx, and time is advanced in steps of size Δt. The DuFort-Frankel scheme updates the temperature at each interior point using:

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

where $\gamma = \frac{\alpha \Delta t}{(\Delta x)^2}$.

This method is conditionally stable and allows for larger time steps compared to the standard explicit scheme, but it introduces numerical diffusion and requires careful treatment of initial and boundary conditions.

Dirichlet boundary conditions are applied by fixing the temperature at the ends of the domain for all time steps.
Solve the one-dimensional heat equation

$$\frac{\partial u}{\partial t} = 0.01 \frac{\partial^2 u}{\partial x^2}$$

for $x \in [0, 10]$ and $t \in [0, 2]$, with initial condition

$$u(x, 0) = \sin\left(\frac{\pi x}{10}\right)$$

and Dirichlet boundary conditions $u(0,t) = u(10,t) = 0$.

Use the DuFort-Frankel scheme with $n = 101$ spatial grid points and $m = 500$ time steps. Plot the temperature profile at selected time steps. The following code implements the scheme.

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

# Parameters
L = 10.0          # Length of the domain
T = 2.0           # Total time
n = 101            # Number of spatial points
m = 500            # Number of time steps
alpha = 0.01       # Thermal diffusivity
dx = L / (n - 1)   # Spatial step
dt = T / m         # Time step

# Stability condition
gamma = alpha * dt / dx**2

# Initialize the temperature field
x = np.linspace(0, L, n)  # Spatial grid
u = np.zeros(n)             # Temperature array
```

```
u_new = np.zeros(n)         # Array for next time step

# Initial condition (e.g., a sine wave)
u[:] = np.sin(np.pi * x / L)

# Boundary conditions (Dirichlet)
u[0] = 0
u[-1] = 0

# Time-stepping loop (DuFort-Frankel scheme)
for t in range(1, m):
    # Apply DuFort-Frankel scheme to update temperature
    for i in range(1, n-1):
        u_new[i] = u[i] + gamma * (u[i-1] - 2 * u[i] + u[i+1])

    # Update old values
    u[:] = u_new[:]

    # Boundary conditions (fixed value)
    u[0] = 0
    u[-1] = 0

    # Plot at selected time steps (optional)
    if t % 50 == 0:
        plt.plot(x, u, label=f't = {t*dt:.2f}')

# Final plot
plt.xlabel('x')
plt.ylabel('Temperature (u)')
plt.title('DuFort-Frankel Scheme for Heat Equation')
plt.legend()
plt.show()
```

OUTPUT:

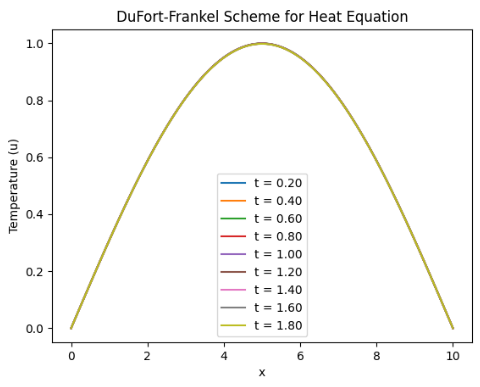

Figure 5: Numerical Solution by Dufort Scheme

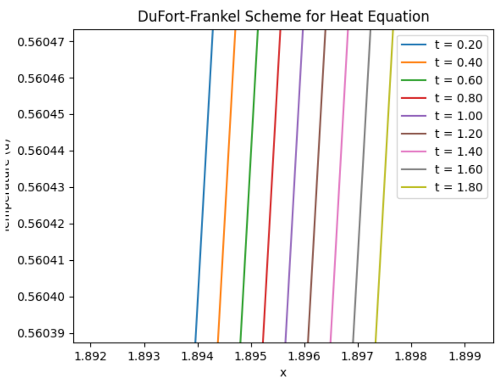

Figure 6: Time Variation

Experiment 14: Explicit Scheme to solve Hyperbolic PDEs

The 1D advection equation describes the transport of a quantity $u(x,t)$ at a constant speed c:

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

The explicit FTCS (Forward Time Centered Space) method discretizes this equation using: - Forward difference in time:

$$\frac{\partial u}{\partial t} \approx \frac{u_i^{n+1} - u_i^n}{\Delta t}$$

- Centered difference in space:

$$\frac{\partial u}{\partial x} \approx \frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x}$$

The update formula for interior points is:

$$u_i^{n+1} = u_i^n - \frac{c\Delta t}{2\Delta x}\left(u_{i+1}^n - u_{i-1}^n\right)$$

Stability: The method is conditionally stable under the CFL (Courant-Friedrichs-Lewy) condition:

$$\frac{c\Delta t}{\Delta x} \leq 1$$

However, the FTCS scheme is generally unstable for hyperbolic equations like the advection equation, even when the CFL condition is satisfied. This instability arises due to the lack of numerical diffusion in the scheme.
Solve the 1D advection equation

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

for $x \in [0,10]$ and $t \in [0,10]$, with: - Initial condition:

$$u(x,0) = e^{-(x-5)^2} \quad \text{(Gaussian pulse centered at } x = 5\text{)}$$

- Dirichlet boundary conditions:

$$u(0,t) = u(10,t) = 0$$

- Wave speed $c = 1.0$

Use the explicit FTCS method with $n = 101$ spatial grid points and $m = 500$ time steps. Plot the wave profile at selected time steps to observe the solution's behavior.

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

# Parameters
```

```python
L = 10.0          # Length of the domain
T = 10.0           # Total time
n = 101            # Number of spatial points
m = 500            # Number of time steps
c = 1.0            # Wave speed
dx = L / (n - 1) # Spatial step size
dt = T / m         # Time step size

# Stability condition (CFL condition)
if c * dt / dx > 1:
    print("Warning: The scheme may be unstable, adjust dt or dx.")

# Initialize the wave function
x = np.linspace(0, L, n)  # Spatial grid
u = np.zeros(n)            # Wave function array
u_new = np.zeros(n)        # Array for next time step

# Initial condition: a Gaussian pulse
u[:] = np.exp(-(x - L/2)**2)

# Boundary conditions (fixed)
u[0] = 0
u[-1] = 0

# Time-stepping loop (Explicit scheme)
for t in range(1, m):
    # Apply the explicit scheme (Lax-Wendroff)
    for i in range(1, n-1):
        u_new[i] = u[i] - (c * dt / (2 * dx)) * (u[i+1] - u[i-1])

    # Update old values for next iteration
    u[:] = u_new[:]

    # Boundary conditions (fixed)
    u[0] = 0
    u[-1] = 0

    # Plot at selected time steps (optional)
    if t % 50 == 0:
        plt.plot(x, u, label=f't = {t*dt:.2f}')

# Final plot
plt.xlabel('x')
plt.ylabel('Wave Function (u)')
plt.title('1D Wave Equation using Explicit Scheme')
plt.legend()
plt.show()
```

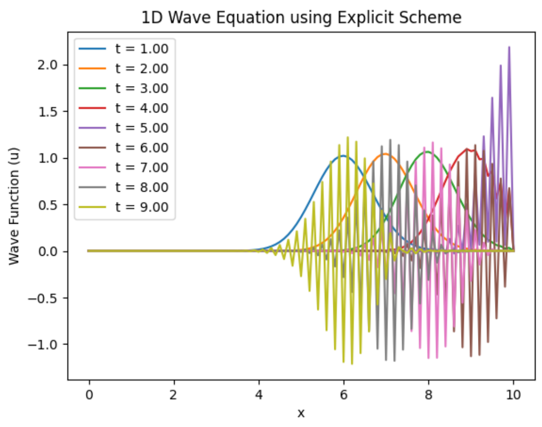

Figure 7: Time evolution of waves

Experiment 15: Galerkin Method

The finite element method (FEM) is a numerical technique for finding approximate solutions to boundary value problems for partial differential equations. The Galerkin method is a specific approach within FEM where the test functions are chosen to be the same as the basis (shape) functions.

Consider the 1D Poisson equation:

$$-\frac{d^2u}{dx^2} = f(x), \quad x \in (0,1)$$

with Dirichlet boundary conditions $u(0) = u(1) = 0$.

The domain is divided into n elements of size h. For each element, local stiffness matrices and load vectors are assembled:

$$K_e = \frac{1}{h}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, \quad F_e = \frac{h}{2}\begin{bmatrix} f(x_i) \\ f(x_{i+1}) \end{bmatrix}$$

For constant $f(x) = 1$, $F_e = \frac{h}{2}\begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

The global stiffness matrix K and load vector F are assembled by summing contributions from all elements. Dirichlet boundary conditions are enforced by modifying K and F, and the resulting linear system is solved for the nodal values of u.

Use the Galerkin finite element method to solve the boundary value problem

$$-\frac{d^2u}{dx^2} = 1, \quad x \in (0,1)$$

with boundary conditions $u(0) = u(1) = 0$.

Divide the domain into 4 equal elements, assemble the global stiffness matrix and load vector, apply boundary conditions, and solve for the nodal values of $u(x)$. Plot the approximate solution.

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

n = 4                  # number of elements
h = 1 / n              # element size
x = np.linspace(0, 1, n+1)  # nodes

K = np.zeros((n+1, n+1))  # stiffness matrix
F = np.zeros(n+1)         # load vector

for i in range(n):
    # local stiffness and force (constant f=1)
    K[i][i]     += 1/h
    K[i][i+1]   += -1/h
    K[i+1][i]   += -1/h
    K[i+1][i+1] += 1/h

    F[i]     += h/2
    F[i+1]   += h/2

# Apply boundary conditions: u(0) = u(1) = 0
K = K[1:-1, 1:-1]
```

```
F = F[1:-1]

u_inner = np.linalg.solve(K, F)
u = np.concatenate(([0], u_inner, [0]))

plt.plot(x, u, 'o-', label='Galerkin FEM')
plt.xlabel('x')
plt.ylabel('u(x)')
plt.title('Simple Galerkin FEM for -u" = 1')
plt.grid(True)
plt.legend()
plt.show()
```

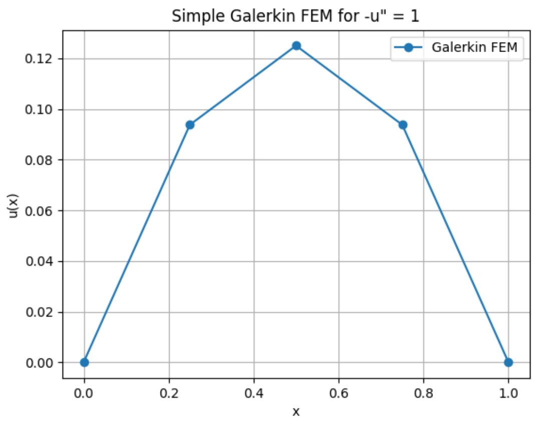

Figure 8: Numerical Solution of Poisson Equation