Experiment – 1-2

Aim: Write a Python Program to find the roots of the equation by using Bisection Method.

Tools Used: Python 3.13.2

Program:

```
import sympy as sp

# Bisection method implementation
def bisection_method(func, a, b, tolerance=1e-7, max_iter=100):
    if func(a) * func(b) >= 0:
        print("The bisection method cannot be applied because the signs of func(a) and func(b) are not opposite.")
        return None, 0, 0

    iteration = 0
    while (b - a) / 2 > tolerance and iteration < max_iter:
        c = (a + b) / 2
        func_c = func(c)

        # Ternary operations to update bounds and check for root
        if func_c == 0:
            return c, iteration, (b - a) / 2
        a, b = (a, c) if func_c * func(a) < 0 else (c, b)
        iteration += 1

    error_bound = (b - a) / 2
    return (a + b) / 2, iteration, error_bound

# Function to parse user input into a sympy function
def parse_function(expression):
    x = sp.symbols('x')
    expr = sp.sympify(expression)
    return sp.lambdify(x, expr, "numpy")

# Function to find a suitable interval for the Bisection method
def find_interval_for_bisection(func, x_start=-10, x_end=10, step_size=0.1):
    a, b = x_start, x_start + step_size
    while b <= x_end:
        if func(a) * func(b) < 0:
            return a, b
        a, b = a + step_size, b + step_size

    print("No sign change found within the specified range.")
    return None, None

# Main program execution
if __name__ == "__main__":
    print("Welcome to the Bisection Method Solver")

    # Get the function from the user
    equation_str = input("Enter the equation function f(x): (e.g., 'x**3 + x**2 + x + 7'): ")
    func = parse_function(equation_str)

    # Menu system to choose interval
    print("\nChoose how to determine the interval:")
    print("1. Input interval manually")
    print("2. Automatically find interval")
    choice = input("Enter your choice (1 or 2): ").strip()

    if choice == '1':
        # User inputs the interval manually
        a = float(input("Enter the start of the interval (a): "))
        b = float(input("Enter the end of the interval (b): "))
        if func(a) * func(b) >= 0:
            print("The bisection method cannot be applied because the signs of f(a) and f(b) are not opposite.")
            exit()
    elif choice == '2':
        # Automatically find the interval
        a, b = find_interval_for_bisection(func)
        if a is None or b is None:
            print("Couldn't find a suitable interval for the Bisection Method.")
            exit()
        print(f"Found suitable interval for bisection method: [{a}, {b}]")
    else:
        print("Invalid choice. Exiting program.")
        exit()

    # Ask for tolerance
    tolerance = float(input("Enter the desired tolerance (e.g., 0.1e-3): ") or 0.1e-3)

    # Perform the Bisection Method
    root, iterations, error_bound = bisection_method(func, a, b, tolerance)

    # Display the results
    if root is not None:
        print(f"\nRoot found using Bisection Method: {root:.4f}")
```

Output:

```
Welcome to the Bisection Method Solver
Enter the equation function f(x): (e.g., 'x**3 + x**2 + x + 7'): x**3 + x**2 + x + 7

Choose how to determine the interval:
1. Input interval manually
2. Automatically find interval
Enter your choice (1 or 2): 2
Found suitable interval for bisection method: [-2.2000000000000197, -2.1000000000000196]
Enter the desired tolerance (e.g., 0.1e-3): 1e-6

Root found using Bisection Method: -2.1049
Number of iterations: 16
Error bound: 0.000001
```

Experiment – 3-4

Aim: Write a Python Program to find the roots of the equation by using Secant or Regula Falsi Method.

Tools Used: Python 3.13.2

Program:

```
import sympy as sp
import numpy as np

# Secant Method implementation
def secant_method(func, x0, x1, tolerance=1e-7, max_iter=100):
    iteration = 0
    while abs(x1 - x0) > tolerance and iteration < max_iter:
        try:
            x_new = x1 - func(x1) * (x1 - x0) / (func(x1) - func(x0))
        except ZeroDivisionError:
            print("Error: Division by zero. The method failed.")
            return None, iteration

        x0, x1 = x1, x_new
        iteration += 1
    return x1, iteration

# Regula Falsi Method implementation
def regula_falsi_method(func, a, b, tolerance=1e-7, max_iter=100):
    iteration = 0
    while (b - a) / 2 > tolerance and iteration < max_iter:
        c = b - func(b) * (b - a) / (func(b) - func(a))
        if func(c) == 0:
            return c, iteration
        a, b = (a, c) if func(c) * func(a) < 0 else (c, b)
        iteration += 1
    return (a + b) / 2, iteration

# Function to parse the equation into a sympy function
def parse_function(expression):
    x = sp.symbols('x')
    expr = sp.sympify(expression)
    return sp.lambdify(x, expr, "numpy")

# Function to find a suitable interval for the method
def find_interval_for_method(func, x_start=-10, x_end=10, step_size=0.1):
    a, b = x_start, x_start + step_size
    while b <= x_end:
        if func(a) * func(b) < 0:
            return a, b
        a, b = a + step_size, b + step_size
    print("No sign change found within the specified range.")
    return None, None

# Function to get initial guesses for methods
def smart_initial_guesses(func, x_start=-10, x_end=10, step_size=0.1):
    x0, x1 = None, None
    for x in np.arange(x_start, x_end, step_size):
        # Sign change detected using ternary-like logic
        if func(x) * func(x + step_size) < 0:
            x0, x1 = x, x + step_size
            break
    return (x0, x1) if x0 is not None and x1 is not None else (None, None)

# Display method options
def display_menu():
    print("\nChoose a method for root finding:")
    print("1. Secant Method")
    print("2. Regula Falsi Method")
    print("3. Exit")

if __name__ == "__main__":
    equation_str = input("Enter the equation function f(x): (e.g., 'x**3 + x**2 + x + 7'): ")

    func = parse_function(equation_str)

    # Find interval using ternary logic to check if it's found
    a, b = find_interval_for_method(func)
    if a is not None and b is not None:
        print(f"Found suitable interval: [{a}, {b}]")

        tolerance = float(input("Enter the desired tolerance (default is 0.1e-3): ") or 0.1e-3)

        while True:
            display_menu()
            method_choice = input("\nEnter the number corresponding to your choice: ").strip()

            if method_choice == '1':
                # Get initial guesses for Secant method using ternary check
                x0, x1 = smart_initial_guesses(func)
                if x0 is not None and x1 is not None:
                    root, iterations = secant_method(func, x0, x1, tolerance)
                    print(f"\nRoot found using Secant Method: {root:.4f}") if root is not None else print("Root not found.")
                    print(f"Number of iterations: {iterations}")

            elif method_choice == '2':
                root, iterations = regula_falsi_method(func, a, b, tolerance)
                print(f"\nRoot found using Regula Falsi Method: {root:.4f}") if root is not None else print("Root not found.")
                print(f"Number of iterations: {iterations}")

            elif method_choice == '3':
                print("Program successfully Terminated.")
                break

            else:
                print("Invalid choice, please try again.")

    else:
        print("Couldn't find a suitable interval for the method.")
```

Output:

```
Enter the equation function f(x): (e.g., 'x**3 + x**2 + x + 7'): x**3 + x**2 + x + 7
Found suitable interval: [-2.2000000000000197, -2.1000000000000196]
Enter the desired tolerance (default is 0.1e-3): 0.1e-3

Choose a method for root finding:
1. Secant Method
2. Regula Falsi Method
3. Exit

Enter the number corresponding to your choice: 1

Root found using Secant Method: -2.1049
Number of iterations: 3

Choose a method for root finding:
1. Secant Method
2. Regula Falsi Method
3. Exit

Enter the number corresponding to your choice: 2

Root found using Regula Falsi Method: -2.1049
Number of iterations: 10

Choose a method for root finding:
1. Secant Method
2. Regula Falsi Method
3. Exit

Enter the number corresponding to your choice: 3
Program successfully Terminated.
```

Experiment – 5-6

Aim: Write a Python Program to find the roots of the equation by using Newton Raphson Method.

Tools Used: Python 3.13.2

Program:

```
import sympy as sp

def newton_raphson_method(func, func_prime, x0, tolerance=1e-7, max_iter=100):
    iteration = 0
    while iteration < max_iter:
        # Calculate the next approximation using Newton-Raphson formula
        try:
            x1 = x0 - func(x0) / func_prime(x0)
        except ZeroDivisionError:
            print("Error: Division by zero. The method failed.")
            return None, iteration

        if abs(x1 - x0) < tolerance:
            return x1, iteration

        x0 = x1
        iteration += 1

    print("Maximum iterations reached.")
    return x1, iteration

def parse_function(expression):
    x = sp.symbols('x')
    expr = sp.sympify(expression)
    func = sp.lambdify(x, expr, "numpy")
    return func

def find_initial_guess(func):
    # Sample points to evaluate the function
    sample_points = [-10, -5, 0, 5, 10]

    prev_value = func(sample_points[0])

    # Loop through points and find a sign change
    for point in sample_points[1:]:
        current_value = func(point)

        if prev_value * current_value < 0:  # Sign change detected
            return (point + sample_points[sample_points.index(point) - 1]) / 2  # Choose middle

        prev_value = current_value

    # If no sign change is detected, use 0 as the initial guess
    return 0

if __name__ == "__main__":
    # User inputs the equation for f(x)
    equation_str = input("Enter the equation function f(x): (e.g., 'x**3 + x**2 + x + 7'): ")

    # Parse the function f(x) from the input string
    func = parse_function(equation_str)

    # Compute the derivative of the function
    x = sp.symbols('x')
    expr = sp.sympify(equation_str)
    func_prime = sp.lambdify(x, sp.diff(expr, x), "numpy")

    tolerance = float(input("Enter the desired tolerance (default is 0.1e-3): ") or 0.1e-3)

    # Ask the user for the initial guess
    x0_input = input("Enter the initial guess x0 (leave empty for automated guess): ")

    if x0_input.strip():  # If the user enters a value
        x0 = float(x0_input)
    else:
        # Automatically determine the initial guess x0
        x0 = find_initial_guess(func)
        print(f"\nSmartly Automated initial guess x0: {x0}")

    # Apply Newton-Raphson Method
    root, iterations = newton_raphson_method(func, func_prime, x0, tolerance)

    if root is not None:
        print(f"\nRoot found using Newton-Raphson Method: {root:.4f}")
        print(f"Number of iterations: {iterations}")
```

Output:

```
Enter the equation function f(x): (e.g., 'x**3 + x**2 + x + 7'): x**3 + x**2 + x + 7
Enter the desired tolerance (default is 0.1e-3): 0.1e-3
Enter the initial guess x0 (leave empty for automated guess):

Smartly Automated initial guess x0: -2.5

Root found using Newton-Raphson Method: -2.1049
Number of iterations: 3
```

Experiment – 7-8

Aim: Write a Python Program to find the solution of a system of nonlinear equation by using Newton's method.

Tools Used: Python 3.13.2

Program:

```
import sympy as sp
import numpy as np
import random

def newton_raphson_method_system(funcs, jacobian, x0, tolerance=1e-7, max_iter=100):
    iteration = 0
    x0 = np.array(x0, dtype=float)
    while iteration < max_iter:
        F_values = np.array([float(f(*x0)) for f in funcs], dtype=float)
        J_matrix = np.array(jacobian(*x0), dtype=float)
        try:
            delta = np.linalg.solve(J_matrix, -F_values)
        except np.linalg.LinAlgError:
            print("Error: Jacobian matrix is singular. The method failed.")
            return None, iteration
        x1 = x0 + delta
        # Check for convergence based on the residual (F_values) instead of just delta
        if np.linalg.norm(F_values) < tolerance:
            return x1, iteration
        # Debug: Print the current guess values for each variable
        print(f"Iteration {iteration + 1}: " + ", ".join([f"x{i+1} = {x1[i]}" for i in range(len(x1))]))
        x0 = x1
        iteration += 1
    print("Maximum iterations reached.")
    return x1, iteration

def parse_system_of_equations(equations, vars):
    funcs = []
    for eq in equations:
        funcs.append(sp.lambdify(vars, sp.sympify(eq), "numpy"))
    return funcs

def parse_jacobian(equations, vars):
    jacobian = []
    for eq in equations:
        jacobian_row = []
        for var in vars:
            jacobian_row.append(sp.diff(sp.sympify(eq), var))
        jacobian.append(jacobian_row)
    jacobian_func = sp.lambdify(vars, jacobian, "numpy")
    return jacobian_func

def generate_initial_guess(num_vars):
    # Automatically generate a random initial guess in the range [-10, 10] for each variable
    return [random.uniform(-10, 10) for _ in range(num_vars)]

if __name__ == "__main__":
        num_equations = int(input("Enter the number of equations in the system [DEFAULT=2]: ") or 2)
        equations = []
        for i in range(num_equations):
            eq = input(f"Enter equation {i+1} (use variables x1, x2, ..., xn): ")
            equations.append(eq)

        vars = sp.symbols(f'x1:{num_equations + 1}')  # This generates x1, x2, ..., xn for n variables

        funcs = parse_system_of_equations(equations, vars)
        jacobian = parse_jacobian(equations, vars)

        # Initial guess
        x0_input = input(f"Enter the initial guess (e.g., 'x1, x2, ..., xn'): ")
        if x0_input.strip():
            x0 = [float(i) for i in x0_input.split(',')]
        else:
            x0 = generate_initial_guess(num_equations)
            print(f"Auto-generated initial guess: {x0}")

        tolerance = float(input("Enter the desired tolerance (e.g., 0.1e-3): ") or 0.1e-3)
        solution, iterations = newton_raphson_method_system(funcs, jacobian, x0, tolerance)

        if solution is not None:
            formatted_solution = [f"{sol:.6f}" for sol in solution]
            print(f"Solution found using Newton-Raphson Method: {', '.join(formatted_solution)}")
            print(f"Number of iterations: {iterations}")
```

Output:

```
Enter the number of equations in the system [DEFAULT=2]: 2
Enter equation 1 (use variables x1, x2, ..., xn): x1**2 + x2 - 4
Enter equation 2 (use variables x1, x2, ..., xn): x1 + x2**2 - 4
Enter the initial guess (e.g., 'x1, x2, ..., xn'): 1, 1
Enter the desired tolerance (e.g., 0.1e-3): 1e-6
Iteration 1: x1 = 1.6666666666666667, x2 = 1.6666666666666665
Iteration 2: x1 = 1.564102564102564, x2 = 1.5641025641025643
Iteration 3: x1 = 1.561554387641344, x2 = 1.5615543876413442
Iteration 4: x1 = 1.5615528128094318, x2 = 1.5615528128094316
Solution found using Newton-Raphson Method: 1.561553, 1.561553
Number of iterations: 4

Enter the number of equations in the system [DEFAULT=2]: 2
Enter equation 1 (use variables x1, x2, ..., xn): sin(x1) + x2 - 2
Enter equation 2 (use variables x1, x2, ..., xn): x1**2 + cos(x2) - 1
Enter the initial guess (e.g., 'x1, x2, ..., xn'):
Auto-generated initial guess: [6.462305257962548, 0.6446128080976372]
Enter the desired tolerance (e.g., 0.1e-3): 1e-6
Iteration 1: x1 = 3.439688635978612, x2 = 4.796093702144594
Iteration 2: x1 = 2.36441912810437, x2 = 1.265853332463239
Iteration 3: x1 = 1.1642318865945003, x2 = 0.44312108577168974
Iteration 4: x1 = 0.7698654920973509, x2 = 1.2374697654594335
Iteration 5: x1 = 0.8343094685116692, x2 = 1.25769028023047
Iteration 6: x1 = 0.8331470789898, x2 = 1.2599478990309005
Solution found using Newton-Raphson Method: 0.833147, 1.259948
Number of iterations: 6
```

Experiment – 9-10

Aim: Write a Python Program to find the solution of tri-diagonal system by using Gauss Thomas method.

Tools Used: Python 3.13.2

Program:

```
import numpy as np
from sympy import symbols, Matrix
from tabulate import tabulate

def get_input(n, mode="manual"):
    """Get user input or generate random coefficients for a tridiagonal system."""
    if mode == "manual":
        print("\nEnter the coefficients for the tridiagonal matrix:")
        a = np.array([float(input(f"  a[{i+1}] (below main diagonal in row {i+2}): ")) for i in range(n-1)])
        b = np.array([float(input(f"  b[{i+1}] (main diagonal in row {i+1}): ")) for i in range(n)])
        c = np.array([float(input(f"  c[{i+1}] (above main diagonal in row {i+1}): ")) for i in range(n-1)])
        d = np.array([float(input(f"  d[{i+1}] (right-hand side for row {i+1}): ")) for i in range(n)])
    else:
        a, c = np.random.randint(1, 10, n-1), np.random.randint(1, 10, n-1)
        b, d = np.random.randint(10, 20, n), np.random.randint(10, 50, n)
        print("\nGenerated random coefficients successfully!")

    return a, b, c, d

def display_matrix(n, a, b, c, d):
    """Display the system as a tridiagonal matrix."""
    matrix = [[b[i] if i == j else (a[j] if i == j+1 else (c[i] if i+1 == j else 0)) for j in range(n)] for i
in range(n)]
    table = [row + [d[i]] for i, row in enumerate(matrix)]
    headers = ["a", "b", "c"][:n] + ["d"]
    print("\nTridiagonal System Representation:\n")
    print(tabulate(table, headers=headers, tablefmt="fancy_grid"))

def symbolic_representation(n, a, b, c, d):
    """Show symbolic representation of the equations."""
    x = symbols(f'x1:{n+1}')
    equations = [f"{a[i-1]}*{x[i-1]} + {b[i]}*{x[i]} + {c[i]}*{x[i+1]} = {d[i]}" if 0 < i < n-1 else
                 (f"{b[i]}*{x[i]} + {c[i]}*{x[i+1]} = {d[i]}" if i == 0 else f"{a[i-1]}*{x[i-1]} + {b[i]}*
{x[i]} = {d[i]}")
                 for i in range(n)]
    print("\nSystem of Equations:\n" + "\n".join([f"  {eq}" for eq in equations]))

def gauss_thomas(n, a, b, c, d):
    """Solve the tridiagonal system using the Gauss-Thomas method."""
    # Step 1: Forward Elimination
    c_star, d_star = np.zeros(n-1), np.zeros(n)
    c_star[0], d_star[0] = c[0] / b[0], d[0] / b[0]
    for i in range(1, n-1):
        denominator = b[i] - a[i-1] * c_star[i-1]
        c_star[i] = c[i] / denominator
        d_star[i] = (d[i] - a[i-1] * d_star[i-1]) / denominator
    d_star[n-1] = (d[n-1] - a[n-2] * d_star[n-2]) / (b[n-1] - a[n-2] * c_star[n-2])
    print("\nForward Elimination Steps:\n")
    for i in range(n-1):
        print(f"  c*[{i+1}] = {c_star[i]:.6f},  d*[{i+1}] = {d_star[i]:.6f}")
    print(f"  d*[{n}] = {d_star[n-1]:.6f}")

    # Step 2: Backward Substitution
    x = np.zeros(n)
    x[n-1] = d_star[n-1]
    for i in range(n-2, -1, -1):
        x[i] = d_star[i] - c_star[i] * x[i+1]
    print("\nBackward Substitution Steps:\n")
    for i in range(n-1, -1, -1):
        print(f"  x[{i+1}] = {x[i]:.6f}")
    return x

if __name__ == "__main__":
    print("Solving a tridiagonal system using the Gauss-Thomas (Thomas) method.\n")
    n = int(input("Enter the number of equations (n): "))
    choice = input("\nChoose input method (1: Manual, 2: Random): ")
    a, b, c, d = get_input(n, mode="manual" if choice == "1" else "random")
    display_matrix(n, a, b, c, d)
    symbolic_representation(n, a, b, c, d)
    solution = gauss_thomas(n, a, b, c, d)
    print("\nFinal Solution:")
    print(tabulate([[f"x[{i+1}]", f"{solution[i]:.6f}"] for i in range(n)], tablefmt="fancy_grid"))
```

Output:

```
Enter the number of equations (n): 4

Choose input method (1: Manual, 2: Random): 2
Generated random coefficients successfully!

Tridiagonal System Representation:
```

	a	b	c	d
17	7	0	0	22
1	17	9	0	25
0	6	19	2	14
0	0	1	15	45

```
System of Equations:
  17*x1 + 7*x2 = 22
  1*x1 + 17*x2 + 9*x3 = 25
  6*x2 + 19*x3 + 2*x4 = 14
  1*x3 + 15*x4 = 45

Forward Elimination Steps:
  c*[1] = 0.411765,  d*[1] = 1.294118
  c*[2] = 0.542553,  d*[2] = 1.429078
  c*[3] = 0.127027,  d*[3] = 0.344595
  d*[4] = 3.002453

Backward Substitution Steps:
  x[4] = 3.002453
  x[3] = -0.036798
  x[2] = 1.449043
  x[1] = 0.697453

Final Solution:
```

x[1]	0.697453
x[2]	1.44904
x[3]	-0.036798
x[4]	3.00245

Experiment – 11-12

Aim: Write a Python Program to find the solution of a system of equations by using Jacobi/Gauss-Seidel method.

Tools Used: Python 3.13.2

Program:

```
import numpy as np
import sympy as sp
from tabulate import tabulate

def is_diagonally_dominant(A):
    return all(abs(A[i, i]) >= sum(abs(A[i, j]) for j in range(len(A)) if j != i) for i in
range(len(A)))
def generate_random_system(n, min_val=1, max_val=10):
    while True:
        A = np.random.uniform(min_val, max_val, (n, n))
        b = np.random.uniform(min_val, max_val, n)
        for i in range(n):
            A[i, i] = sum(abs(A[i, j]) for j in range(n) if j != i) + np.random.uniform(1,
max_val)
        if is_diagonally_dominant(A):
            return A, b
def format_matrix(A, b=None):
    table = [list(np.round(A[i], 5)) + ([round(b[i], 5)] if b is not None else []) for i
in range(len(A))]
    headers = [f"x{i+1}" for i in range(len(A))] + (["b"] if b is not None else [])
    return tabulate(table, headers=headers, tablefmt="fancy_grid")
def get_float_input(prompt, default):
    val = input(prompt)
    return float(val) if val.strip() != "" else default
def get_int_input(prompt, default):
    val = input(prompt)
    return int(val) if val.strip() != "" else default
def solve_system(A, b, method, tol=1e-6, max_iter=100):
    # Solve Ax = b using Jacobi or Gauss-Seidel method with detailed output.
    n, x = len(b), np.zeros(len(b))
    print(f"\nSolving using {method} Method...")
    headers = ["Iteration"] + [f"x{i+1}" for i in range(n)] + ["Error"]
    iterations = []
    if method == "Jacobi":
        D = np.diag(A)
        R = A - np.diagflat(D)
        for k in range(max_iter):
            x_new = (b - np.dot(R, x)) / D
            error = np.linalg.norm(x_new - x, ord=np.inf)
            iterations.append([k+1] + list(np.round(x_new, 5)) + [f"{error:.6e}"])
            if error < tol:
                break
            x = x_new
    elif method == "Gauss-Seidel":
        for k in range(max_iter):
            x_new = np.copy(x)
            for i in range(n):
                s1, s2 = np.dot(A[i, :i], x_new[:i]), np.dot(A[i, i+1:], x[i+1:])
                x_new[i] = (b[i] - s1 - s2) / A[i, i]
            error = np.linalg.norm(x_new - x, ord=np.inf)
            iterations.append([k+1] + list(np.round(x_new, 5)) + [f"{error:.6e}"])
            if error < tol:
                break
            x = x_new
    print("\nIteration Process:")
    print(tabulate(iterations, headers=headers, tablefmt="fancy_grid"))
    if error >= tol:
        print("\nIterative method did not converge within the max iterations.")
    return x_new
def solve_linear_system():
    print("\nSystem of Linear Equations Solver")
    print("1. Manually input the coefficient matrix and vectors.")
    print("2. Randomly generate a diagonally dominant system.")

    choice = input("Choose an option (1 or 2): ").strip()
    if choice not in {"1", "2"}:
        print("Invalid choice! Exiting.")
        return
    n = get_int_input("\nEnter the number of variables: ", 3)
    if n <= 0:
        print("Error: Number of variables must be positive!")
        return
    if choice == "1":
        print("\nEnter the coefficient matrix row by row (space-separated):")
        A = np.array([list(map(float, input(f"Row {i+1}: ").split())) for i in range(n)])
        b = np.array(list(map(float, input("\nEnter the right-hand side vector (space-
separated): ").split())))
    else:
        A, b = generate_random_system(n)
        print("\nGenerated System of Equations:")
        print(format_matrix(A, b))
    if np.any(np.diag(A) == 0):
        print("Error: Matrix contains zero diagonal elements. Cannot proceed.")
        return
    tol = get_float_input("\nEnter tolerance (default 1e-6): ", 1e-6)
    max_iter = get_int_input("Enter max iterations (default 100): ", 100)
    while True:
        print("\nChoose a method to solve:")
        print("1. Jacobi Method")
        print("2. Gauss-Seidel Method")
        method_choice = input("Enter choice (1 or 2): ").strip()
        method = {"1": "Jacobi", "2": "Gauss-Seidel"}.get(method_choice)
        if not method:
            print("Invalid method choice! Exiting.")
            return
        solution = solve_system(A, b, method, tol, max_iter)
        print("\n Final Approximate Solution (Iterative Method):")
        print(tabulate([solution], headers=[f"x{i+1}" for i in range(n)],
tablefmt="fancy_grid"))
        print("\n Direct Method Solution (Verification):")
        try:
            x_direct = np.linalg.solve(A, b)
            print(tabulate([x_direct], headers=[f"x{i+1}" for i in range(n)],
tablefmt="fancy_grid"))
        except np.linalg.LinAlgError:
            print("Direct solution failed due to a singular or ill-conditioned matrix.")
if __name__ == "__main__":
    solve_linear_system()
```

Output:

```
System of Linear Equations Solver
1. Manually input the coefficient matrix and vectors.
2. Randomly generate a diagonally dominant system.
Choose an option (1 or 2): 2

Enter the number of variables: 3

Generated System of Equations:
```

x1	x2	x3	b
12.9419	4.24417	1.35821	8.99958
5.12734	10.5559	1.89894	6.71284
3.4446	5.71926	12.0189	5.31848

```
Enter tolerance (default 1e-6): 0.1e-3
Enter max iterations (default 100):

Choose a method to solve:
1. Jacobi Method
2. Gauss-Seidel Method
Enter choice (1 or 2): 2

Solving using Gauss-Seidel Method...

Iteration Process:
```

Iteration	x1	x2	x3	Error
1	0.69538	0.29816	0.10873	0.695385
2	0.5871	0.33272	0.11533	0.108289
3	0.57424	0.33634	0.11729	0.0128552
4	0.57285	0.33667	0.11754	0.00139428
5	0.57271	0.33669	0.11757	0.000132522
6	0.5727	0.33669	0.11757	9.69046e-06

```
Final Approximate Solution (Iterative Method):
```

x1	x2	x3
0.572705	0.336691	0.117569

```
Direct Method Solution (Verification):
```

x1	x2	x3
0.572704	0.33669	0.117569

Experiment – 13-14

Aim: Write a Python Program to find the cubic spine interpolating function.

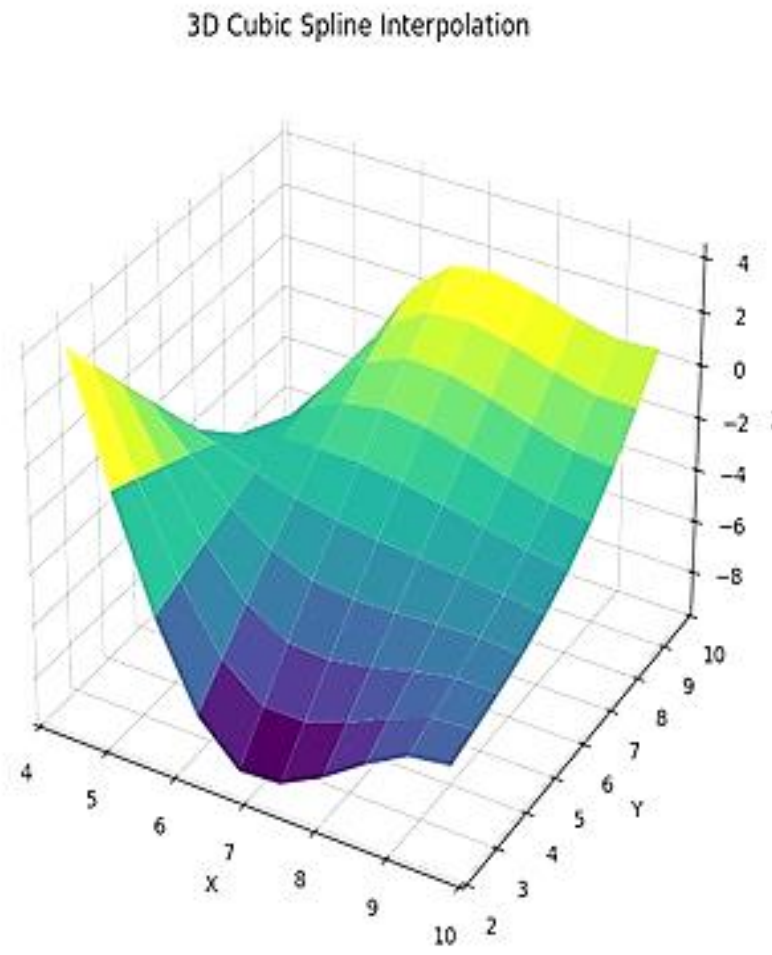
Tools Used: Python 3.13.2

Program:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def cubic_spline_3d(x, y, z, query_points_x, query_points_y):
    X, Y = np.meshgrid(query_points_x, query_points_y)
    def cubic_spline_1d(x_vals, y_vals, query_points):
        n = len(x_vals) - 1
        h = np.diff(x_vals)
        A = np.zeros((n + 1, n + 1))
        b = np.zeros(n + 1)
        A[0, 0] = 1
        A[n, n] = 1
        for i in range(1, n):
            A[i, i - 1] = h[i - 1]
            A[i, i] = 2 * (h[i - 1] + h[i])
            A[i, i + 1] = h[i]
            b[i] = 3 * ((y_vals[i + 1] - y_vals[i]) / h[i] - (y_vals[i] - y_vals[i - 1]) / h[i - 1])
        M = np.linalg.solve(A, b)
        def evaluate_spline(xi):
            for i in range(n):
                if x_vals[i] <= xi <= x_vals[i + 1]:
                    a = y_vals[i]
                    b = (y_vals[i + 1] - y_vals[i]) / h[i] - h[i] * (2 * M[i] + M[i + 1]) / 3
                    c = M[i] / 2
                    d = (M[i + 1] - M[i]) / (3 * h[i])
                    dx = xi - x_vals[i]
                    return a + b * dx + c * dx**2 + d * dx**3
            raise ValueError("Query point out of bounds.")

        return np.array([evaluate_spline(xi) for xi in query_points])
    interpolated_z = np.array([cubic_spline_1d(x, z_row, query_points_x) for z_row in z])
    interpolated_z_final = np.array([cubic_spline_1d(y, interpolated_z[:, i], query_points_y) for i in range(len(query_points_x))])
    return X, Y, interpolated_z_final
def get_input():
    while True:
        try:
            print("\nChoose how to provide input:")
            print("1. Enter data points manually")
            print("2. Use randomized data points (optimal range)")
            choice = int(input("Enter your choice (1 or 2): "))
            if choice == 1:
                n = int(input("Enter the number of data points (>= 3): "))
                if n < 3:
                    print("You must have at least 3 data points for spline interpolation!")
                    continue
                x = []
                y = []
                z = []
                print("Enter the x, y and z coordinates:")
                for i in range(n):
                    xi = float(input(f"x[{i}]: "))
                    yi = float(input(f"y[{i}]: "))
                    zi = float(input(f"z[{i}]: "))
                    x.append(xi)
                    y.append(yi)
                    z.append(zi)
                x = np.array(x)
                y = np.array(y)
                z = np.array(z)
            elif choice == 2:
                n = int(input("Enter the number of data points (>= 3): "))
                if n < 3:
                    print("You must have at least 3 data points for spline interpolation!")
                    continue
                x = np.sort(np.random.uniform(0, 10, n))  # Random x in range [0, 10]
                y = np.sort(np.random.uniform(0, 10, n))  # Random y in range [0, 10]
                z = np.random.uniform(-10, 10, (n, n))    # Random 2D z values in range [-10, 10]
                print("\nGenerated random data points:")
                for i in range(n):
                    for j in range(n):
                        print(f"x[{i}] = {x[i]:.2f}, y[{j}] = {y[j]:.2f}, z[{i},{j}] = {z[i,j]:.2f}")
            else:
                print("Invalid choice! Please enter 1 or 2.")
                continue
            return x, y, z
        except ValueError as e:
            print(f"Invalid input: {e}. Please try again.")
def main():
    while True:
        print("\n=== 3D Cubic Spline Interpolation ===")
        x, y, z = get_input()
        print("\nEnter the number of query points for interpolation (leave empty for default): ")
        user_input = input()
        if user_input.strip() == "":
            num_query_points = max(10, len(x) * 3)
            print(f"Using default number of query points: {num_query_points}")
        else:
            try:
                num_query_points = int(user_input)
                if num_query_points <= 0:
                    print("Number of query points must be greater than 0. Using default value.")
                    num_query_points = max(10, len(x) * 3)
            except ValueError:
                print("Invalid input! Using default number of query points.")
                num_query_points = max(10, len(x) * 3)

        query_points_x = np.linspace(min(x), max(x), num_query_points)
        query_points_y = np.linspace(min(y), max(y), num_query_points)
        X, Y, interpolated_z = cubic_spline_3d(x, y, z, query_points_x, query_points_y)
        print("\nInterpolation complete. Plotting 3D surface...")
        fig = plt.figure(figsize=(10, 7))
        ax = fig.add_subplot(111, projection='3d')
        ax.plot_surface(X, Y, interpolated_z, cmap='viridis', edgecolor='none')
        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_zlabel('Z')
        ax.set_title('3D Cubic Spline Interpolation')
        plt.show()
        repeat = input("\nDo you want to run the program again? (yes/no): ").strip().lower()
        if repeat not in ('yes', 'y'):
            print("Goodbye!")
            break
if __name__ == "__main__":
    main()
```

Output:

```
=== 3D Cubic Spline Interpolation ===

Choose how to provide input:
1. Enter data points manually
2. Use randomized data points (optimal range)
Enter your choice (1 or 2): 2
Enter the number of data points (>= 3): 3

Generated random data points:
x[0] = 4.30, y[0] = 2.41, z[0,0] = 4.24
x[0] = 4.30, y[1] = 5.69, z[0,1] = -6.48
x[0] = 4.30, y[2] = 9.53, z[0,2] = -8.91
x[1] = 8.62, y[0] = 2.41, z[1,0] = -9.45
x[1] = 8.62, y[1] = 5.69, z[1,1] = 0.03
x[1] = 8.62, y[2] = 9.53, z[1,2] = 1.59
x[2] = 9.67, y[0] = 2.41, z[2,0] = 6.87
x[2] = 9.67, y[1] = 5.69, z[2,1] = -7.72
x[2] = 9.67, y[2] = 9.53, z[2,2] = -7.58

Enter the number of query points for interpolation

Using default number of query points: 10

Interpolation complete. Plotting 3D surface...
```



Experiment – 15-16

Aim: Write a Python Program to evaluate the approximate value of finite integrals using Gaussian/Romberg integration.

Tools Used: Python 3.13.2

Program:

```
import random
from sympy import symbols, sympify, lambdify
from scipy.integrate import quad

def get_input_or_random(prompt, use_random=False, min_val=-10, max_val=10):
    return random.uniform(min_val, max_val) if use_random else get_user_input(prompt)
def get_user_input(prompt):
    while True:
        try:
            value = input(prompt).strip()
            if value == '': raise ValueError("Input cannot be empty.")
            return float(value)
        except ValueError as e:
            print(f"Invalid input. Please enter a valid number. Error: {e}")
def parse_function(function_str):
    try:
        x = symbols('x')
        func = sympify(function_str)
        return lambdify(x, func, "numpy")
    except Exception as e:
        print(f"Error parsing the function: {e}")
        return None
if __name__ == "__main__":
    print("1. Manually input the lower and upper limits.")
    print("2. Randomly generate the suitable lower and upper limits.\n")
    choice = input("Choose an option (1 or 2): ").strip()
    use_random = True if choice == '2' else False
    equation_str = input("Enter the equation function f(x): (e.g., 'x**3 + x**2 + x + 7'): ")
    func = parse_function(equation_str)
    if not func:
        print("Exiting due to invalid function.")
        exit()
    a = get_input_or_random("Enter the lower limit of integration (a): ", use_random)
    b = get_input_or_random("Enter the upper limit of integration (b): ", use_random)
    if use_random:
        print(f"Randomized limits used: a = {a:.3f}, b = {b:.3f}")
    elif a >= b:
        print("Invalid limits: The lower limit must be less than the upper limit. Exiting.")
        exit()
    while True:
        print("\nGaussian Quadrature and Romberg Integration")
        print("-----------------------------------------------")
        print("1. Perform Gaussian Quadrature Integration")
        print("2. Perform Romberg Integration")
        print("3. Exit")
        method = input("Enter your choice: ").strip()
        if method == '1':
            print("\nUsing Gaussian Quadrature Method...")
            try:
                result, _ = quad(func, a, b)
                print(f"Result using Gaussian Quadrature: {result:.3f}\n")
            except Exception as e:
                print(f"Error with integration: {e}")
        elif method == '2':
            print("\nUsing Romberg Integration Method (via quad with higher accuracy)...")
            try:
                result, _ = quad(func, a, b, epsrel=1e-8)
                print(f"Result using Romberg Integration: {result:.3f}\n")
            except Exception as e:
                print(f"Error with integration: {e}")
        elif method == '3':
            print("\nGoodBye.")
            print("-----------------------------------------------")
            break
        else:
            print("\n-----------------------------------------------")
            print("\tInvalid method selected! Try again...")
            print("-----------------------------------------------")
            continue
```

Output:

```
1. Manually input the lower and upper limits.
2. Randomly generate the suitable lower and upper limits.

Choose an option (1 or 2): 2
Enter the equation function f(x): (e.g., 'x**3 + x**2 + x + 7'): x**3 + x**2 + x + 7
Randomized limits used: a = -8.896, b = -0.891

Gaussian Quadrature and Romberg Integration
-----------------------------------------------
1. Perform Gaussian Quadrature Integration
2. Perform Romberg Integration
3. Exit
Enter your choice: 1

Using Gaussian Quadrature Method...
Result using Gaussian Quadrature: -1314.469


Gaussian Quadrature and Romberg Integration
-----------------------------------------------
1. Perform Gaussian Quadrature Integration
2. Perform Romberg Integration
3. Exit
Enter your choice: 2

Using Romberg Integration Method (via quad with higher accuracy)...
Result using Romberg Integration: -1314.469


Gaussian Quadrature and Romberg Integration
-----------------------------------------------
1. Perform Gaussian Quadrature Integration
2. Perform Romberg Integration
3. Exit
Enter your choice: 3

GoodBye.
-----------------------------------------------
```

Experiment – 17-18

Aim: Write a Python Program to solve the boundary value problem using finite difference method.

Tools Used: Python 3.13.2

Program:

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

def get_input_or_random(prompt, use_random=False, min_val=-10, max_val=10):
    return (random.uniform(-1, 1) if 'boundary' in prompt.lower() else
            random.randint(10, 100) if 'grid points' in prompt.lower() else
            random.uniform(min_val, max_val)) if use_random else get_user_input(prompt)
def get_user_input(prompt):
    while True:
        try:
            value = input(prompt).strip()
            if value == '': raise ValueError("Input cannot be empty.")
            return float(value)
        except ValueError as e:
            print(f"Invalid input. Please enter a valid number. Error: {e}")
def solve_bvp(N, a=0, b=1, alpha=0, beta=0):
    h = (b - a) / (N + 1)
    x = np.linspace(a, b, N + 2)
    f = -np.pi**2 * np.sin(np.pi * x)
    f[0], f[-1] = alpha, beta
    A = np.diag(-2 * np.ones(N)) + np.diag(np.ones(N-1), 1) + np.diag(np.ones(N-1), -1)
    A /= h**2
    b = f[1:-1]
    y = np.linalg.solve(A, b)
    full_y = np.concatenate(([alpha], y, [beta]))
    return x, full_y

if __name__ == "__main__":
    print("1. Manually input the number of grid points (N) and boundary conditions.")
    print("2. Randomly generate the suitable number of grid points (N) and boundary conditions.\n")
    choice = int(input("Choose an option (1 or 2): "))
    if choice == 1:
        use_random = False
    elif choice == 2:
        use_random = True
    else:
        print("\nInvalid Choice Made! Try Again... \n")
    a = get_input_or_random("Enter the lower boundary (a): ", use_random)
    b = get_input_or_random("Enter the upper boundary (b): ", use_random)
    if a >= b:
        print("Invalid limits: The lower limit must be less than the upper limit. Exiting.")
        exit()
    alpha = get_input_or_random("Enter the value for the lower boundary condition (alpha): ", use_random)
    beta = get_input_or_random("Enter the value for the upper boundary condition (beta): ", use_random)
    N = int(get_input_or_random("Enter the number of interior grid points (N): ", use_random, 10, 100))
    print(f"\nValues used:")
    print(f"a (lower boundary): {a}")
    print(f"b (upper boundary): {b}")
    print(f"alpha (lower boundary condition): {alpha}")
    print(f"beta (upper boundary condition): {beta}")
    print(f"N (number of interior grid points): {N}")
    if use_random:
        print(f"Randomized values used: \na = {a:.3f}, b = {b:.3f}, alpha = {alpha:.3f}, beta = {beta:.3f}, N = {N}")
    while True:
        print("\nFinite Difference Method for Boundary Value Problem")
        print("-----------------------------------------------------")
        print("1. Solve the Boundary Value Problem using FDM")
        print("2. Exit")
        method = input("Enter your choice: ").strip()
        if method == '1':
            print("\nSolving the Boundary Value Problem using Finite Difference Method...")
            x, y = solve_bvp(N, a=a, b=b, alpha=alpha, beta=beta)
            print(f"Solution at grid points: {list(zip(x, y))}\n")
            plt.plot(x, y, label="FDM Solution")
            plt.xlabel("x")
            plt.ylabel("y(x)")
            plt.title("Solution of Boundary Value Problem using Finite Difference Method (FDM)")
            plt.grid(True)
            plt.legend()
            plt.show()
        elif method == '2':
            print("\nGoodBye.")
            print("-----------------------------------------------------")
            break
        else:
            print("\n-----------------------------------------------------")
            print("\tInvalid method selected! Try again...")
            print("-----------------------------------------------------")
            continue
```

Output:

```
1. Manually input the number of grid points (N) and boundary conditions.
2. Randomly generate the suitable number of grid points (N) and boundary conditions.

Choose an option (1 or 2): 2

Values used:
a (lower boundary): -0.17400654611264743
b (upper boundary): 0.27092642111540566
alpha (lower boundary condition): 0.9655245135503838
beta (upper boundary condition): -0.8775665505581238
N (number of interior grid points): 88
Randomized values used:
a = -0.174, b = 0.271, alpha = 0.966, beta = -0.878, N = 88

Finite Difference Method for Boundary Value Problem
--------------------------------------------------
1. Solve the Boundary Value Problem using FDM
2. Exit
Enter your choice: 1
```