Lab 3 – ITITIU22240 – Đàm Nguyễn Trọng Lễ

Act 1:

import numpy as np

def gaussian\_elimination(A, b):

    n = len(A)

    aug = np.hstack((A, b.reshape(-1, 1))).astype(float)

    # Forward Elimination

    for i in range(n):

        pivot = aug[i, i]

        if pivot == 0:

            for k in range(i+1, n):

                if aug[k, i] != 0:

                    aug[[i, k]] = aug[[k, i]]

                    pivot = aug[i, i]

                    break

            else:

                raise ValueError("Matrix is singular.")

        for j in range(i+1, n):

            m = aug[j, i] / pivot

            aug[j] -= m \* aug[i]

    # Back Substitution

    x = np.zeros(n)

    for i in range(n-1, -1, -1):

        x[i] = (aug[i, -1] - np.dot(aug[i, i+1:n], x[i+1:n])) / aug[i, i]

    return x

# System of equations

A = np.array([[3, 1, -2],

              [2, -2, 4],

              [-1, 12, -1]], dtype=float)

b = np.array([1, -2, 0], dtype=float)

solution\_ge = gaussian\_elimination(A, b)

print("Gaussian Elimination Solution:", solution\_ge)

Gaussian Elimination Solution: [ 0. -0.04347826 -0.52173913]

Act 2:

import numpy as np

def jacobi\_method(A, b, x0=None, tol=1e-6, max\_iter=100):

    n = len(b)

    # Check for zero diagonal elements

    for i in range(n):

        if abs(A[i, i]) < 1e-10:

            raise ValueError(f"Zero diagonal element at position {i}. Jacobi method cannot proceed.")

    # Initial guess

    if x0 is None:

        x = np.zeros(n)

    else:

        x = x0.copy()

    # Iteration variables

    iterations = 0

    residual\_history = []

    # Extract diagonal elements

    D = np.diag(np.diag(A))

    R = A - D

    # Jacobi iteration

    while iterations < max\_iter:

        x\_old = x.copy()

        # Update x using the Jacobi formula: x = D^(-1) \* (b - R\*x)

        for i in range(n):

            sigma = 0

            for j in range(n):

                if j != i:

                    sigma += A[i, j] \* x\_old[j]

            x[i] = (b[i] - sigma) / A[i, i]

        # Calculate residual

        residual = np.linalg.norm(A @ x - b)

        residual\_history.append(residual)

        # Check convergence

        if np.linalg.norm(x - x\_old) < tol:

            break

        iterations += 1

    return x, iterations, residual\_history

def gauss\_seidel\_method(A, b, x0=None, tol=1e-6, max\_iter=100):

    n = len(b)

    # Check for zero diagonal elements

    for i in range(n):

        if abs(A[i, i]) < 1e-10:

            raise ValueError(f"Zero diagonal element at position {i}. Gauss-Seidel method cannot proceed.")

    # Initial guess

    if x0 is None:

        x = np.zeros(n)

    else:

        x = x0.copy()

    # Iteration variables

    iterations = 0

    residual\_history = []

    # Gauss-Seidel iteration

    while iterations < max\_iter:

        x\_old = x.copy()

        # Update x using the Gauss-Seidel formula

        for i in range(n):

            sigma = 0

            for j in range(n):

                if j != i:

                    sigma += A[i, j] \* x[j]  # Use updated values immediately

            x[i] = (b[i] - sigma) / A[i, i]

        # Calculate residual

        residual = np.linalg.norm(A @ x - b)

        residual\_history.append(residual)

        # Check convergence

        if np.linalg.norm(x - x\_old) < tol:

            break

        iterations += 1

    return x, iterations, residual\_history

import numpy as np

import time

# Define the system of equations

A = np.array([

    [3, 1, -2],

    [2, -2, 4],

    [-1, 12, -1]

])

b = np.array([1, -2, 0])

# Initial guess

x0 = np.zeros(3)

# Solve using Jacobi method

try:

    start\_time = time.time()

    jacobi\_solution, jacobi\_iterations, jacobi\_residuals = jacobi\_method(A, b, x0, tol=1e-6, max\_iter=100)

    jacobi\_time = time.time() - start\_time

    print("Solution using Jacobi Method:")

    print(f"x = {jacobi\_solution[0]:.6f}")

    print(f"y = {jacobi\_solution[1]:.6f}")

    print(f"z = {jacobi\_solution[2]:.6f}")

    print(f"Iterations: {jacobi\_iterations}")

    print(f"Time taken: {jacobi\_time:.6f} seconds")

    # Verify the solution

    residuals = A @ jacobi\_solution - b

    print("\nResiduals (should be close to zero):")

    print(residuals)

    print(f"Sum of absolute residuals: {np.sum(np.abs(residuals)):.10f}")

except ValueError as e:

    print(f"Error in Jacobi method: {e}")

print("\n" + "-"\*50 + "\n")

# Solve using Gauss-Seidel method

try:

    start\_time = time.time()

    gs\_solution, gs\_iterations, gs\_residuals = gauss\_seidel\_method(A, b, x0, tol=1e-6, max\_iter=100)

    gs\_time = time.time() - start\_time

    print("Solution using Gauss-Seidel Method:")

    print(f"x = {gs\_solution[0]:.6f}")

    print(f"y = {gs\_solution[1]:.6f}")

    print(f"z = {gs\_solution[2]:.6f}")

    print(f"Iterations: {gs\_iterations}")

    print(f"Time taken: {gs\_time:.6f} seconds")

    # Verify the solution

    residuals = A @ gs\_solution - b

    print("\nResiduals (should be close to zero):")

    print(residuals)

    print(f"Sum of absolute residuals: {np.sum(np.abs(residuals)):.10f}")

except ValueError as e:

    print(f"Error in Gauss-Seidel method: {e}")

Solution using Jacobi Method:

x = 152037631471235787279170060043344368399862504976157604200389908889600.000000

y = 609249400623943066746320893886785503457799604761999743285360026714112.000000

z = 1439937617137894804434883223145369737036781708505748756630518781444096.000000

Iterations: 100

Time taken: 0.007468 seconds

Residuals (should be close to zero):

[-1.81451294e+69 4.84532693e+69 5.71901756e+69]

Sum of absolute residuals: 12378857428362489808880535780326095361012656983198616323892066488680448.0000000000

--------------------------------------------------

Solution using Gauss-Seidel Method:

x = 1574640304958144207361292941958555174996325193539543824347174316819564493671883899795642271322081354484542506980229080273361143162959461709030359040.000000

y = 6508362969162576616780929997946205116457246623702464598957317882148218541431750454294624175144538715220723581934607490819491343509282119412441153536.000000

z = 76525715324992770047934914745163526844953908521326677785113428371443494487099019931002700543171226865971642746952003860802447434583396656664226562048.000000

Iterations: 100

Time taken: 0.004297 seconds

Residuals (should be close to zero):

[-1.41819147e+149 2.96235416e+149 0.00000000e+000]

Sum of absolute residuals: 438054562737510783672036954942667854292260528731515460295107708140423777906783240943182829069440077227497523259228834239253762256603408098506032807936.0000000000

Act 3:

import numpy as np

import time

import matplotlib.pyplot as plt

# Define the system of equations

A = np.array([

    [3, 1, -2],

    [2, -2, 4],

    [-1, 12, -1]

])

b = np.array([1, -2, 0])

# Initial guess for iterative methods

x0 = np.zeros(3)

# Comparative analysis

methods = []

solutions = []

iterations\_list = []

times = []

residuals\_list = []

# 1. Gaussian Elimination

try:

    start\_time = time.time()

    ge\_solution = gaussian\_elimination(A, b)

    ge\_time = time.time() - start\_time

    methods.append("Gaussian Elimination")

    solutions.append(ge\_solution)

    iterations\_list.append(1)  # Direct method, only 1 "iteration"

    times.append(ge\_time)

    residuals\_list.append(np.linalg.norm(A @ ge\_solution - b))

except ValueError as e:

    print(f"Error in Gaussian Elimination: {e}")

# 2. Jacobi Method

try:

    start\_time = time.time()

    jacobi\_solution, jacobi\_iterations, jacobi\_residuals = jacobi\_method(A, b, x0, tol=1e-6, max\_iter=100)

    jacobi\_time = time.time() - start\_time

    methods.append("Jacobi Method")

    solutions.append(jacobi\_solution)

    iterations\_list.append(jacobi\_iterations)

    times.append(jacobi\_time)

    residuals\_list.append(np.linalg.norm(A @ jacobi\_solution - b))

except ValueError as e:

    print(f"Error in Jacobi method: {e}")

# 3. Gauss-Seidel Method

try:

    start\_time = time.time()

    gs\_solution, gs\_iterations, gs\_residuals = gauss\_seidel\_method(A, b, x0, tol=1e-6, max\_iter=100)

    gs\_time = time.time() - start\_time

    methods.append("Gauss-Seidel Method")

    solutions.append(gs\_solution)

    iterations\_list.append(gs\_iterations)

    times.append(gs\_time)

    residuals\_list.append(np.linalg.norm(A @ gs\_solution - b))

except ValueError as e:

    print(f"Error in Gauss-Seidel method: {e}")

# Print comparison table

print("Comparative Analysis of Methods")

print("-" \* 80)

print(f"{'Method':<20} {'Iterations':<12} {'Time (s)':<12} {'Residual':<12} {'Solution'}")

print("-" \* 80)

for i, method in enumerate(methods):

    sol\_str = f"x={solutions[i][0]:.4f}, y={solutions[i][1]:.4f}, z={solutions[i][2]:.4f}"

    print(f"{method:<20} {iterations\_list[i]:<12} {times[i]:<12.6f} {residuals\_list[i]:<12.6e} {sol\_str}")

# Plot convergence for iterative methods

plt.figure(figsize=(10, 6))

if len(jacobi\_residuals) > 0:

    plt.semilogy(range(len(jacobi\_residuals)), jacobi\_residuals, 'b-o', label='Jacobi Method')

if len(gs\_residuals) > 0:

    plt.semilogy(range(len(gs\_residuals)), gs\_residuals, 'r-s', label='Gauss-Seidel Method')

plt.xlabel('Iteration')

plt.ylabel('Residual (log scale)')

plt.title('Convergence Comparison of Iterative Methods')

plt.grid(True)

plt.legend()

plt.tight\_layout()

# Add a horizontal line for Gaussian Elimination residual

if len(residuals\_list) > 0 and "Gaussian Elimination" in methods:

    ge\_idx = methods.index("Gaussian Elimination")

    plt.axhline(y=residuals\_list[ge\_idx], color='g', linestyle='-', label='Gaussian Elimination')

    plt.legend()

plt.show()

A graph with a line and a line

AI-generated content may be incorrect.

Discuss:

1. Gaussian Elimination:

Advantages: Direct method, typically fast for small systems, provides exact solution

Disadvantages: Higher computational complexity (O(n³)), less efficient for large systems,susceptible to round-off errors, requires complete matrix recalculation for changes

2. Jacobi Method:

Advantages: Simple to implement, parallelizable, works well for diagonally dominant matrices

Disadvantages: Generally slower convergence, may not converge for all matrices,requires more iterations than Gauss-Seidel

3. Gauss-Seidel Method:

Advantages: Faster convergence than Jacobi, uses less memory, good for sparse matrices

Disadvantages: Less parallelizable, may still not converge for all matrices, still iterative and potentially slower than direct methods for small systems

Act 4:

import numpy as np

import matplotlib.pyplot as plt

import time

# Define the system: Ax = b

A = np.array([

    [5, -2, 3],

    [2, 5, -1],

    [1, 3, 5]

])

b = np.array([10, 4, 8])

# For verification, calculate the exact solution using numpy's linear algebra solver

exact\_solution = np.linalg.solve(A, b)

print("Exact solution (for verification):")

print(f"x = {exact\_solution[0]:.6f}")

print(f"y = {exact\_solution[1]:.6f}")

print(f"z = {exact\_solution[2]:.6f}")

# Check if matrix is diagonally dominant (helps with convergence analysis)

diag\_dominant = True

for i in range(len(A)):

    if abs(A[i, i]) <= sum(abs(A[i, j]) for j in range(len(A)) if j != i):

        diag\_dominant = False

        break

print(f"\nMatrix is diagonally dominant: {diag\_dominant}")

print("Note: Diagonal dominance generally ensures convergence of iterative methods.\n")

def jacobi\_method(A, b, x0=None, tol=1e-6, max\_iter=1000):

    n = len(b)

    # Check for zero diagonal elements

    for i in range(n):

        if abs(A[i, i]) < 1e-10:

            raise ValueError(f"Zero diagonal element at position {i}. Jacobi method cannot proceed.")

    # Initial guess

    if x0 is None:

        x = np.zeros(n)

    else:

        x = x0.copy()

    # Iteration variables

    iterations = 0

    residual\_history = []

    error\_history = []  # Track difference from exact solution

    # Extract diagonal elements

    D = np.diag(np.diag(A))

    R = A - D

    # Jacobi iteration

    while iterations < max\_iter:

        x\_old = x.copy()

        # Update x using the Jacobi formula: x = D^(-1) \* (b - R\*x)

        for i in range(n):

            sigma = 0

            for j in range(n):

                if j != i:

                    sigma += A[i, j] \* x\_old[j]

            x[i] = (b[i] - sigma) / A[i, i]

        # Calculate residual

        residual = np.linalg.norm(A @ x - b)

        residual\_history.append(residual)

        # Calculate error compared to exact solution

        error = np.linalg.norm(x - exact\_solution)

        error\_history.append(error)

        # Check convergence

        if np.linalg.norm(x - x\_old) < tol:

            break

        iterations += 1

    return x, iterations, residual\_history, error\_history

def gauss\_seidel\_method(A, b, x0=None, tol=1e-6, max\_iter=1000):

    n = len(b)

    # Check for zero diagonal elements

    for i in range(n):

        if abs(A[i, i]) < 1e-10:

            raise ValueError(f"Zero diagonal element at position {i}. Gauss-Seidel method cannot proceed.")

    # Initial guess

    if x0 is None:

        x = np.zeros(n)

    else:

        x = x0.copy()

    # Iteration variables

    iterations = 0

    residual\_history = []

    error\_history = []  # Track difference from exact solution

    # Gauss-Seidel iteration

    while iterations < max\_iter:

        x\_old = x.copy()

        # Update x using the Gauss-Seidel formula

        for i in range(n):

            sigma = 0

            for j in range(n):

                if j != i:

                    sigma += A[i, j] \* x[j]  # Use updated values immediately

            x[i] = (b[i] - sigma) / A[i, i]

        # Calculate residual

        residual = np.linalg.norm(A @ x - b)

        residual\_history.append(residual)

        # Calculate error compared to exact solution

        error = np.linalg.norm(x - exact\_solution)

        error\_history.append(error)

        # Check convergence

        if np.linalg.norm(x - x\_old) < tol:

            break

        iterations += 1

    return x, iterations, residual\_history, error\_history

# Initial guess

x0 = np.zeros(3)

# Tolerance

tol = 1e-6

# Solve using Jacobi method

print("-" \* 50)

print("Jacobi Method:")

start\_time = time.time()

jacobi\_solution, jacobi\_iterations, jacobi\_residuals, jacobi\_errors = jacobi\_method(A, b, x0, tol=tol)

jacobi\_time = time.time() - start\_time

print(f"Solution: x = {jacobi\_solution[0]:.6f}, y = {jacobi\_solution[1]:.6f}, z = {jacobi\_solution[2]:.6f}")

print(f"Iterations: {jacobi\_iterations}")

print(f"Time taken: {jacobi\_time:.6f} seconds")

print(f"Final residual: {jacobi\_residuals[-1]:.6e}")

print(f"Difference from exact solution: {np.linalg.norm(jacobi\_solution - exact\_solution):.6e}")

# Solve using Gauss-Seidel method

print("-" \* 50)

print("Gauss-Seidel Method:")

start\_time = time.time()

gs\_solution, gs\_iterations, gs\_residuals, gs\_errors = gauss\_seidel\_method(A, b, x0, tol=tol)

gs\_time = time.time() - start\_time

print(f"Solution: x = {gs\_solution[0]:.6f}, y = {gs\_solution[1]:.6f}, z = {gs\_solution[2]:.6f}")

print(f"Iterations: {gs\_iterations}")

print(f"Time taken: {gs\_time:.6f} seconds")

print(f"Final residual: {gs\_residuals[-1]:.6e}")

print(f"Difference from exact solution: {np.linalg.norm(gs\_solution - exact\_solution):.6e}")

# Comparison

print("-" \* 50)

print("Comparison:")

print(f"Jacobi iterations: {jacobi\_iterations}")

print(f"Gauss-Seidel iterations: {gs\_iterations}")

print(f"Iteration reduction: {jacobi\_iterations - gs\_iterations} ({(jacobi\_iterations - gs\_iterations)/jacobi\_iterations\*100:.1f}%)")

# Plot convergence

plt.figure(figsize=(12, 10))

# Plot 1: Residual convergence (log scale)

plt.subplot(2, 1, 1)

plt.semilogy(range(1, len(jacobi\_residuals)+1), jacobi\_residuals, 'b-o', label='Jacobi Residual', markersize=4)

plt.semilogy(range(1, len(gs\_residuals)+1), gs\_residuals, 'r-s', label='Gauss-Seidel Residual', markersize=4)

plt.axhline(y=tol, color='g', linestyle='--', label=f'Tolerance ({tol})')

plt.xlabel('Iterations')

plt.ylabel('Residual (log scale)')

plt.title('Convergence Comparison: Residual')

plt.grid(True)

plt.legend()

# Plot 2: Error convergence (log scale)

plt.subplot(2, 1, 2)

plt.semilogy(range(1, len(jacobi\_errors)+1), jacobi\_errors, 'b-o', label='Jacobi Error', markersize=4)

plt.semilogy(range(1, len(gs\_errors)+1), gs\_errors, 'r-s', label='Gauss-Seidel Error', markersize=4)

plt.xlabel('Iterations')

plt.ylabel('Error vs. Exact Solution (log scale)')

plt.title('Convergence Comparison: Error')

plt.grid(True)

plt.legend()

plt.tight\_layout()

plt.show()

# Print comparison table

print("\nComparative Analysis of Methods")

print("-" \* 80)

print(f"{'Method':<15} {'Iterations':<12} {'Time (s)':<12} {'Final Residual':<18} {'Error vs Exact'}")

print("-" \* 80)

print(f"{'Jacobi':<15} {jacobi\_iterations:<12} {jacobi\_time:<12.6f} {jacobi\_residuals[-1]:<18.6e} {np.linalg.norm(jacobi\_solution - exact\_solution):.6e}")

print(f"{'Gauss-Seidel':<15} {gs\_iterations:<12} {gs\_time:<12.6f} {gs\_residuals[-1]:<18.6e} {np.linalg.norm(gs\_solution - exact\_solution):.6e}")

Exact solution (for verification):

x = 1.527273

y = 0.400000

z = 1.054545

Matrix is diagonally dominant: False

Note: Diagonal dominance generally ensures convergence of iterative methods.

--------------------------------------------------

Jacobi Method:

Solution: x = 1.527272, y = 0.400000, z = 1.054546

Iterations: 25

Time taken: 0.001540 seconds

Final residual: 2.201831e-06

Difference from exact solution: 5.812493e-07

--------------------------------------------------

Gauss-Seidel Method:

Solution: x = 1.527273, y = 0.400000, z = 1.054545

Iterations: 11

Time taken: 0.000708 seconds

Final residual: 1.014150e-06

Difference from exact solution: 1.812592e-07

--------------------------------------------------

Comparison:

Jacobi iterations: 25

Gauss-Seidel iterations: 11

Iteration reduction: 14 (56.0%)

A graph with a line graph

AI-generated content may be incorrect.

A graph with a line going up

AI-generated content may be incorrect.

Comparative Analysis of Methods

--------------------------------------------------------------------------------

Method Iterations Time (s) Final Residual Error vs Exact

--------------------------------------------------------------------------------

Jacobi 25 0.001540 2.201831e-06 5.812493e-07

Gauss-Seidel 11 0.000708 1.014150e-06 1.812592e-07