import numpy as np

# --------------a--------------

#Jacobi method implementation

def jacobi(A, b, x0, max\_iter=100, omega=1.0):

    D = np.diag(np.diag(A))

    R = A - D

    D\_inv = np.linalg.inv(D)

    x = x0.copy()

    errors = []

    conv\_factors = []

    for k in range(max\_iter):

        x\_new = x + omega \* (D\_inv @ (b - A @ x))

        error = np.linalg.norm(A @ x\_new - b)

        errors.append(error)

        if k > 0:

            conv\_factors.append(error / errors[k - 1])

        x = x\_new

    return x, errors, conv\_factors

#gauss\_seidel method implementation

def gauss\_seidel(A, b, x0, max\_iter=100):

    L = np.tril(A)

    U = A - L

    x = x0.copy()

    errors = []

    conv\_factors = []

    for k in range(max\_iter):

        x\_new = np.linalg.solve(L, b - U @ x)

        error = np.linalg.norm(A @ x\_new - b)

        errors.append(error)

        if k > 0:

            conv\_factors.append(error / errors[k - 1])

        x = x\_new

    return x, errors, conv\_factors

#steepest\_descent method implementation

def steepest\_descent(A, b, x0, max\_iter=100):

    x = x0.copy()

    errors = []

    conv\_factors = []

    for k in range(max\_iter):

        r = b - A @ x

        alpha = r @ r / (r @ A @ r)

        x = x + alpha \* r

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

        errors.append(error)

        if k > 0:

            conv\_factors.append(error / errors[k - 1])

    return x, errors, conv\_factors

#conjugate\_gradient method implementation

def conjugate\_gradient(A, b, x0, max\_iter=100):

    x = x0.copy()

    r = b - A @ x

    p = r.copy()

    rs\_old = r @ r

    errors = []

    conv\_factors = []

    for k in range(max\_iter):

        Ap = A @ p

        alpha = rs\_old / (p @ Ap)

        x = x + alpha \* p

        r = r - alpha \* Ap

        rs\_new = r @ r

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

        errors.append(error)

        if k > 0:

            conv\_factors.append(error / errors[k - 1])

        if np.sqrt(rs\_new) < 1e-10:

            break

        p = r + (rs\_new / rs\_old) \* p

        rs\_old = rs\_new

    return x, errors, conv\_factors

# --------------b--------------

import scipy.sparse as sparse

import matplotlib.pyplot as plt

# create a random sparse matrix A

n = 256

A\_rand = sparse.random(n, n, density=5/n, format='csr')

v = np.random.rand(n)

V = sparse.spdiags(v, 0, n, n, format='csr')

A = A\_rand.transpose() @ V @ A\_rand + 0.1 \* sparse.eye(n)

A = A.toarray()

b = np.random.randn(n)

x0 = np.zeros(n)

# Run the methods

x\_jacobi, err\_jacobi, conv\_jacobi = jacobi(A, b, x0, omega=0.1)

x\_gs, err\_gs, conv\_gs = gauss\_seidel(A, b, x0)

x\_sd, err\_sd, conv\_sd = steepest\_descent(A, b, x0)

x\_cg, err\_cg, conv\_cg = conjugate\_gradient(A, b, x0)

errors = [

    (err\_jacobi, "Jacobi"),

    (err\_gs, "Gauss-Seidel"),

    (err\_sd, "Steepest Descent"),

    (err\_cg, "Conjugate Gradient")

]

convs = [

    (conv\_jacobi, "Jacobi"),

    (conv\_gs, "Gauss-Seidel"),

    (conv\_sd, "Steepest Descent"),

    (conv\_cg, "Conjugate Gradient")

]

# plot the errors

for err, name in errors:

    plt.figure(figsize=(8, 5))

    plt.semilogy(err)

    plt.title(f"Residual Norm: ||Ax(k) - b|| for {name}")

    plt.xlabel("Iteration")

    plt.ylabel("Residual Norm (log scale)")

    plt.grid(True)

    plt.tight\_layout()

    plt.show()

# plot the convergence factors

for conv, name in convs:

    plt.figure(figsize=(8, 5))

    plt.plot(conv)

    plt.title(f"Convergence Factor for {name}")

    plt.xlabel("Iteration")

    plt.ylabel("||Ax(k) - b|| / ||Ax(k-1) - b||")

    plt.grid(True)

    plt.tight\_layout()

    plt.show()

# --------------c--------------

At = A.T

A\_ls = At @ A

b\_ls = At @ b

x0 = np.zeros\_like(b)

# solving LS problem using the same methods

x\_jacobi\_ls, err\_jacobi\_ls, conv\_jacobi\_ls = jacobi(A\_ls, b\_ls, x0, omega=0.6)

x\_gs\_ls, err\_gs\_ls, conv\_gs\_ls = gauss\_seidel(A\_ls, b\_ls, x0)

x\_sd\_ls, err\_sd\_ls, conv\_sd\_ls = steepest\_descent(A\_ls, b\_ls, x0)

x\_cg\_ls, err\_cg\_ls, conv\_cg\_ls = conjugate\_gradient(A\_ls, b\_ls, x0)

errors\_ls = [

    (err\_jacobi\_ls, "Jacobi (LS)"),

    (err\_gs\_ls, "Gauss-Seidel (LS)"),

    (err\_sd\_ls, "Steepest Descent (LS)"),

    (err\_cg\_ls, "Conjugate Gradient (LS)")

]

convs\_ls = [

    (conv\_jacobi\_ls, "Jacobi (LS)"),

    (conv\_gs\_ls, "Gauss-Seidel (LS)"),

    (conv\_sd\_ls, "Steepest Descent (LS)"),

    (conv\_cg\_ls, "Conjugate Gradient (LS)")

]

# plotting the errors for LS problem

for err, name in errors\_ls:

    plt.figure(figsize=(8, 5))

    plt.semilogy(err)

    plt.title(f"Residual Norm: ||Ax(k) - b|| for {name}")

    plt.xlabel("Iteration")

    plt.ylabel("Residual Norm (log scale)")

    plt.grid(True)

    plt.tight\_layout()

    plt.show()

# plotting the convergence factors for LS problem

for conv, name in convs\_ls:

    plt.figure(figsize=(8, 5))

    plt.plot(conv)

    plt.title(f"Convergence Factor for {name}")

    plt.xlabel("Iteration")

    plt.ylabel("||Ax(k) - b|| / ||Ax(k-1) - b||")

    plt.grid(True)

    plt.tight\_layout()

    plt.show()

np.allclose(x\_cg, x\_cg\_ls, atol=1e-4)

import numpy as np

import matplotlib.pyplot as plt

# Define matrix A

A = np.array([

    [5, 4, 4, -1, 0],

    [3, 12, 4, -5, -5],

    [-4, 2, 6, 0, 3],

    [4, 5, -7, 10, 2],

    [1, 2, 5, 3, 10]

], dtype=float)

# Given vector b and initial guess x

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

x = np.zeros\_like(b)

# Number of iterations

max\_iter = 50

# Store residual norms

residuals = []

for k in range(max\_iter):

    r = b - A @ x               # residual

    Ar = A @ r                  # A \* r

    AtAr = A.T @ Ar             # A^T \* A \* r

    alpha = (r @ Ar) / (r @ AtAr)  # optimal step size

    x = x + alpha \* r           # update x

    res\_norm = np.linalg.norm(b - A @ x)  # residual norm

    residuals.append(res\_norm)

    print(f"Iteration {k+1}: Residual norm = {res\_norm:.2e}")

# Plotting residual norm vs iterations on a log scale

plt.figure(figsize=(8, 5))

plt.semilogy(range(1, max\_iter + 1), residuals, marker='o')

plt.title('GMRES(1) Residual Norm vs Iteration')

plt.xlabel('Iteration')

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

plt.grid(True)

plt.tight\_layout()

plt.show()

import numpy as np

import matplotlib.pyplot as plt

# Define the 10x10 Laplacian matrix L

L = np.array([

    [ 2, -1, -1,  0,  0,  0,  0,  0,  0,  0],

    [-1,  2, -1,  0,  0,  0,  0,  0,  0,  0],

    [-1, -1,  3, -1,  0,  0,  0,  0,  0,  0],

    [ 0,  0, -1,  5, -1,  0, -1,  0, -1, -1],

    [ 0,  0,  0, -1,  4, -1, -1, -1,  0,  0],

    [ 0,  0,  0,  0, -1,  3, -1, -1,  0,  0],

    [ 0,  0,  0, -1, -1, -1,  5, -1,  0, -1],

    [ 0,  0,  0,  0, -1, -1, -1,  4,  0, -1],

    [ 0,  0,  0, -1,  0,  0,  0,  0,  2, -1],

    [ 0,  0,  0, -1,  0,  0, -1, -1, -1,  4]

], dtype=float)

# Define b = [1, -1, 1, -1, ..., 1]

b = np.array([1 if i % 2 == 0 else -1 for i in range(10)], dtype=float)

# Jacobi method

def jacobi\_iteration(L, b, x0, tol=1e-5, max\_iter=1000):

    D = np.diag(np.diag(L))

    D\_inv = np.linalg.inv(D)

    x = x0

    residuals = []

    for \_ in range(max\_iter):

        x\_new = x + D\_inv @ (b - L @ x)

        res = np.linalg.norm(b - L @ x\_new)

        residuals.append(res)

        if res < tol:

            break

        x = x\_new

    return x, residuals

# Run the iteration

x0 = np.zeros(10)

x\_approx, residuals = jacobi\_iteration(L, b, x0)

# Compute convergence factors

convergence\_factors = [residuals[i] / residuals[i - 1] for i in range(1, len(residuals))]

# Plot residuals and convergence factors

plt.figure(figsize=(12, 5))

# Left: residual norm (log scale)

plt.subplot(1, 2, 1)

plt.semilogy(residuals, marker='o')

plt.title("Jacobi Residual Norm")

plt.xlabel("Iteration")

plt.ylabel("Residual Norm (log scale)")

plt.grid(True)

# Right: convergence factor

plt.subplot(1, 2, 2)

plt.plot(range(1, len(residuals)), convergence\_factors, marker='x')

plt.title("Convergence Factor per Iteration")

plt.xlabel("Iteration")

plt.ylabel("Convergence Factor")

plt.grid(True)

plt.tight\_layout()

plt.show()

print("convergece factor at last iteration: " + str(convergence\_factors[-1]))

import numpy as np

import matplotlib.pyplot as plt

from scipy.linalg import block\_diag

# Define the 10x10 Laplacian matrix L

L = np.array([

    [ 2, -1, -1,  0,  0,  0,  0,  0,  0,  0],

    [-1,  2, -1,  0,  0,  0,  0,  0,  0,  0],

    [-1, -1,  3, -1,  0,  0,  0,  0,  0,  0],

    [ 0,  0, -1,  5, -1,  0, -1,  0, -1, -1],

    [ 0,  0,  0, -1,  4, -1, -1, -1,  0,  0],

    [ 0,  0,  0,  0, -1,  3, -1, -1,  0,  0],

    [ 0,  0,  0, -1, -1, -1,  5, -1,  0, -1],

    [ 0,  0,  0,  0, -1, -1, -1,  4,  0, -1],

    [ 0,  0,  0, -1,  0,  0,  0,  0,  2, -1],

    [ 0,  0,  0, -1,  0,  0, -1, -1, -1,  4]

], dtype=float)

# Define the alternating right-hand side vector b

b = np.array([1 if i % 2 == 0 else -1 for i in range(10)], dtype=float)

# Define the damping parameter

omega = 0.7

# Create block preconditioner M = block\_diag(M1, M2)

M1 = L[0:3, 0:3]

M2 = L[3:, 3:]

M\_inv = np.zeros\_like(L)

# Place M1^{-1} and M2^{-1} into full M^{-1}

M\_inv[0:3, 0:3] = np.linalg.inv(M1)

M\_inv[3:, 3:] = np.linalg.inv(M2)

# Preconditioned Jacobi iteration using block M

def block\_preconditioned\_jacobi(L, b, M\_inv, omega=0.7, tol=1e-5, max\_iter=1000):

    x = np.zeros\_like(b)

    residuals = []

    for \_ in range(max\_iter):

        r = b - L @ x

        x\_new = x + omega \* (M\_inv @ r)

        res = np.linalg.norm(b - L @ x\_new)

        residuals.append(res)

        if res < tol:

            break

        x = x\_new

    return x, residuals

# Run the block preconditioned Jacobi

x0 = np.zeros(10)

x\_approx, residuals = block\_preconditioned\_jacobi(L, b, M\_inv, omega=omega)

# Compute convergence factors

convergence\_factors = [residuals[i] / residuals[i - 1] for i in range(1, len(residuals))]

# Plot results

plt.figure(figsize=(12, 5))

# Residual norm (log scale)

plt.subplot(1, 2, 1)

plt.semilogy(residuals, marker='o')

plt.title("Block Preconditioned Jacobi: Residual Norm")

plt.xlabel("Iteration")

plt.ylabel("Residual Norm (log scale)")

plt.grid(True)

# Convergence factor

plt.subplot(1, 2, 2)

plt.plot(range(1, len(residuals)), convergence\_factors, marker='x')

plt.title("Block Preconditioned Jacobi: Convergence Factor")

plt.xlabel("Iteration")

plt.ylabel("convergence factor")

plt.grid(True)

plt.tight\_layout()

plt.show()

print("convergece factor at last iteration: " + str(convergence\_factors[-1]))

import numpy as np

import matplotlib.pyplot as plt

# Original Laplacian matrix L

L = np.array([

    [ 2, -1, -1,  0,  0,  0,  0,  0,  0,  0],

    [-1,  2, -1,  0,  0,  0,  0,  0,  0,  0],

    [-1, -1,  3, -1,  0,  0,  0,  0,  0,  0],

    [ 0,  0, -1,  5, -1,  0, -1,  0, -1, -1],

    [ 0,  0,  0, -1,  4, -1, -1, -1,  0,  0],

    [ 0,  0,  0,  0, -1,  3, -1, -1,  0,  0],

    [ 0,  0,  0, -1, -1, -1,  5, -1,  0, -1],

    [ 0,  0,  0,  0, -1, -1, -1,  4,  0, -1],

    [ 0,  0,  0, -1,  0,  0,  0,  0,  2, -1],

    [ 0,  0,  0, -1,  0,  0, -1, -1, -1,  4]

], dtype=float)

# RHS vector b = [1, -1, 1, -1, ..., 1]

b = np.array([1 if i % 2 == 0 else -1 for i in range(10)], dtype=float)

perm = [0, 1, 2, 3, 7, 9, 4, 5, 6, 8]

L\_perm = L[np.ix\_(perm, perm)]

b\_perm = b[perm]

group1 = [0, 1, 2]

group2 = [7, 4, 5, 6]

group3 = [3, 8, 9]

groups = [group1, group2, group3]

# Construct M^{-1} from the block inverse

M\_inv = np.zeros\_like(L\_perm)

for group in groups:

    block = L\_perm[np.ix\_(group, group)]

    inv\_block = np.linalg.inv(block)

    for i, gi in enumerate(group):

        for j, gj in enumerate(group):

            M\_inv[gi, gj] = inv\_block[i, j]

# Block-Jacobi iteration

def run\_block\_jacobi(L, b, M\_inv, omega=0.7, tol=1e-5, max\_iter=100):

    x = np.zeros\_like(b)

    residuals = []

    for \_ in range(max\_iter):

        r = b - L @ x

        x\_new = x + omega \* (M\_inv @ r)

        res = np.linalg.norm(b - L @ x\_new)

        residuals.append(res)

        if res < tol:

            break

        x = x\_new

    return x, residuals

# Run the solver

x\_final, residuals = run\_block\_jacobi(L\_perm, b\_perm, M\_inv, omega=0.7)

convergence\_factors = [residuals[i] / residuals[i - 1] for i in range(1, len(residuals))]

# Plot results

plt.figure(figsize=(12, 5))

# Convergence factor

plt.subplot(1, 2, 1)

plt.plot(range(1, len(residuals)), convergence\_factors)

plt.title("Jacobi method with block preconditioner C and w = 0.7")

plt.xlabel("iterations")

plt.ylabel("convergence rate")

plt.grid(True)

plt.legend(["convergence rate"])

# Residual norm

plt.subplot(1, 2, 2)

plt.semilogy(residuals)

plt.title("Jacobi method with block preconditioner C and w = 0.7")

plt.xlabel("iterations")

plt.ylabel("residual")

plt.grid(True)

plt.legend(["residual"])

plt.tight\_layout()

plt.show()

# Print final solution

print("Final solution x:")

print(x\_final)