```python
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
```

```python
# --------------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
```

```python
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)
```

```python
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()
```

```python
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]))
```

```python
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]))
```

```python
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)
```