**Problem1**

import numpy as np

import pandas as pd

from sklearn.decomposition import PCA

import matplotlib.pyplot as plt

# 读取数据

df = pd.read\_csv(r'/DailyReturn.csv')

file\_path = r'/DailyReturn.csv'

data = pd.read\_csv(file\_path)

def exponentially\_weighted\_covariance(X, lambd):

    T, N = X.shape

    weights = np.array([lambd \*\* (T - t) for t in range(1, T + 1)])

    weights /= weights.sum()  # Normalize weights

    # Compute weighted mean

    mu = np.average(X, axis=0, weights=weights)

    # Compute weighted deviations

    X\_centered = X - mu

    weighted\_X = X\_centered \* np.sqrt(weights[:, np.newaxis])

    # Compute covariance matrix

    Sigma = np.dot(weighted\_X.T, weighted\_X)

    return Sigma

# Lambda values to test

lambdas = [0.99999999, 0.99, 0.9, 0.7, 0.5, 0.3, 0.1, 0.01]

cumulative\_variances = {}

for lambd in lambdas:

    Sigma = exponentially\_weighted\_covariance(data.values, lambd)

    # Perform PCA on the covariance matrix

    pca = PCA()

    pca.fit(Sigma)

    cum\_var\_explained = np.cumsum(pca.explained\_variance\_ratio\_)

    cumulative\_variances[lambd] = cum\_var\_explained

# Plotting

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

for lambd, cum\_var in cumulative\_variances.items():

    plt.plot(range(1, len(cum\_var) + 1), cum\_var, label=f'λ = {lambd}')

plt.title('Cumulative Variance Explained by Principal Components')

plt.xlabel('Number of Components')

plt.ylabel('Cumulative Variance Explained')

plt.legend()

plt.grid(True)

plt.show()

**Problem2**

import numpy as np

import time

def chol\_psd(A, tol=1e-8):

    n = A.shape[0]

    root = np.zeros\_like(A)

    for i in range(n):

        for j in range(i+1):

            s = np.dot(root[i, :j], root[j, :j])

            if i == j:

                d = A[i, i] - s

                if d > tol:

                    root[i, j] = np.sqrt(d)

                else:

                    root[i, j] = 0.0

            else:

                if root[j, j] > tol:

                    root[i, j] = (A[i, j] - s) / root[j, j]

                else:

                    root[i, j] = 0.0

    return root

def near\_psd(A, epsilon=0):

    A\_sym = (A + A.T) / 2

    eigenvalues, eigenvectors = np.linalg.eigh(A\_sym)

    eigenvalues[eigenvalues < epsilon] = epsilon

    A\_psd = eigenvectors @ np.diag(eigenvalues) @ eigenvectors.T

    return A\_psd

def higham\_nearest\_psd(A, max\_iterations=100, tol=1e-8):

    n = A.shape[0]

    Y = A.copy()

    delta\_S = np.zeros\_like(A)

    for k in range(max\_iterations):

        R = Y - delta\_S

        X = project\_symmetric(R)

        eigenvalues, eigenvectors = np.linalg.eigh(X)

        eigenvalues[eigenvalues < 0] = 0

        X\_psd = eigenvectors @ np.diag(eigenvalues) @ eigenvectors.T

        delta\_S = X\_psd - R

        Y = project\_unit\_diagonal(X\_psd)

        if np.linalg.norm(Y - X\_psd, ord='fro') < tol:

            break

    return Y

def project\_symmetric(A):

    return (A + A.T) / 2

def project\_unit\_diagonal(A):

    A\_new = A.copy()

    np.fill\_diagonal(A\_new, 1)

    return A\_new

def is\_psd(A):

    eigenvalues = np.linalg.eigvalsh(A)

    return np.all(eigenvalues >= -1e-8)

def generate\_non\_psd\_correlation\_matrix(n):

    A = np.random.randn(n, n)

    A = (A + A.T) / 2

    np.fill\_diagonal(A, 1)

    A[0, 1] = 2

    A[1, 0] = 2

    return A

# Scaling analysis

n\_values = [100, 200, 300, 400, 500]

near\_psd\_times = []

higham\_psd\_times = []

for n in n\_values:

    print(f"\nMatrix size: {n}x{n}")

    A = generate\_non\_psd\_correlation\_matrix(n)

    print("Is original matrix PSD?", is\_psd(A))

    # near\_psd()

    start\_time = time.time()

    A\_near\_psd = near\_psd(A)

    near\_psd\_time = time.time() - start\_time

    near\_psd\_times.append(near\_psd\_time)

    print("Is near\_psd() matrix PSD?", is\_psd(A\_near\_psd))

    # Higham's method

    start\_time = time.time()

    A\_higham\_psd = higham\_nearest\_psd(A)

    higham\_psd\_time = time.time() - start\_time

    higham\_psd\_times.append(higham\_psd\_time)

    print("Is Higham's method matrix PSD?", is\_psd(A\_higham\_psd))

    # Frobenius norms

    near\_psd\_fro\_norm = np.linalg.norm(A - A\_near\_psd, ord='fro')

    higham\_psd\_fro\_norm = np.linalg.norm(A - A\_higham\_psd, ord='fro')

    print(f"Frobenius norm difference (near\_psd): {near\_psd\_fro\_norm}")

    print(f"Frobenius norm difference (Higham): {higham\_psd\_fro\_norm}")

    print(f"Runtime (near\_psd): {near\_psd\_time:.4f} seconds")

    print(f"Runtime (Higham's method): {higham\_psd\_time:.4f} seconds")

**Problem3**

import pandas as pd

import numpy as np

import time

from numpy.linalg import norm

# Load the data

df = pd.read\_csv(r'/DailyReturn.csv')

file\_path = r'/DailyReturn.csv'

data = pd.read\_csv(file\_path)

# Generate covariance matrices

def exponentially\_weighted\_covariance(X, lambd):

    T, N = X.shape

    weights = np.array([lambd \*\* (T - t) for t in range(1, T + 1)])

    weights /= weights.sum()

    mu = np.average(X, axis=0, weights=weights)

    X\_centered = X - mu

    weighted\_cov = (X\_centered \* weights[:, np.newaxis]).T @ X\_centered

    return weighted\_cov

def exponentially\_weighted\_variance(X, lambd):

    ew\_cov = exponentially\_weighted\_covariance(X, lambd)

    ew\_var = np.diag(ew\_cov)

    return ew\_var

def exponentially\_weighted\_correlation(X, lambd):

    ew\_cov = exponentially\_weighted\_covariance(X, lambd)

    ew\_std = np.sqrt(np.diag(ew\_cov))

    ew\_corr = ew\_cov / np.outer(ew\_std, ew\_std)

    ew\_corr = np.clip(ew\_corr, -1, 1)

    return ew\_corr

lambd = 0.97

data\_values = data.values

# Standard Pearson correlation and variance

pearson\_corr = data.corr()

standard\_variance = data.var()

# Exponentially weighted correlation and variance

ew\_variance = exponentially\_weighted\_variance(data\_values, lambd)

ew\_corr = exponentially\_weighted\_correlation(data\_values, lambd)

# Construct covariance matrices

def construct\_covariance(corr\_matrix, variance\_vector):

    std\_dev = np.sqrt(variance\_vector)

    cov\_matrix = corr\_matrix \* np.outer(std\_dev, std\_dev)

    return cov\_matrix

cov\_pearson\_var = construct\_covariance(pearson\_corr.values, standard\_variance.values)

cov\_pearson\_ew\_var = construct\_covariance(pearson\_corr.values, ew\_variance)

cov\_ew\_corr\_var = construct\_covariance(ew\_corr, standard\_variance.values)

cov\_ew\_corr\_ew\_var = construct\_covariance(ew\_corr, ew\_variance)

# Simulation functions

def simulate\_direct(cov\_matrix, n\_samples):

    mean\_vector = np.zeros(cov\_matrix.shape[0])

    simulated\_data = np.random.multivariate\_normal(mean\_vector, cov\_matrix, size=n\_samples)

    return simulated\_data

def simulate\_pca(cov\_matrix, n\_samples, variance\_explained):

    eigenvalues, eigenvectors = np.linalg.eigh(cov\_matrix)

    idx = np.argsort(eigenvalues)[::-1]

    eigenvalues = eigenvalues[idx]

    eigenvectors = eigenvectors[:, idx]

    cum\_variance\_explained = np.cumsum(eigenvalues) / np.sum(eigenvalues)

    num\_components = np.searchsorted(cum\_variance\_explained, variance\_explained) + 1

    selected\_eigenvalues = eigenvalues[:num\_components]

    selected\_eigenvectors = eigenvectors[:, :num\_components]

    mean\_vector = np.zeros(cov\_matrix.shape[0])

    random\_samples = np.random.randn(n\_samples, num\_components)

    simulated\_data = random\_samples @ np.diag(np.sqrt(selected\_eigenvalues)) @ selected\_eigenvectors.T

    return simulated\_data

# Simulation parameters

n\_samples = 25000

variance\_explained\_levels = [1.0, 0.75, 0.50]

covariance\_matrices = {

    'pearson\_var': cov\_pearson\_var,

    'pearson\_ew\_var': cov\_pearson\_ew\_var,

    'ew\_corr\_var': cov\_ew\_corr\_var,

    'ew\_corr\_ew\_var': cov\_ew\_corr\_ew\_var

}

simulation\_results = {}

timings = {}

for cov\_name, cov\_matrix in covariance\_matrices.items():

    print(f"\nSimulating from covariance matrix: {cov\_name}")

    simulation\_results[cov\_name] = {}

    timings[cov\_name] = {}

    # Direct Simulation

    start\_time = time.time()

    sim\_data\_direct = simulate\_direct(cov\_matrix, n\_samples)

    end\_time = time.time()

    timings[cov\_name]['direct'] = end\_time - start\_time

    simulation\_results[cov\_name]['direct'] = sim\_data\_direct

    print(f"Direct simulation completed in {timings[cov\_name]['direct']:.2f} seconds.")

    # PCA with varying variance explained

    for variance\_explained in variance\_explained\_levels:

        start\_time = time.time()

        sim\_data\_pca = simulate\_pca(cov\_matrix, n\_samples, variance\_explained)

        end\_time = time.time()

        method\_name = f'pca\_{int(variance\_explained\*100)}'

        timings[cov\_name][method\_name] = end\_time - start\_time

        simulation\_results[cov\_name][method\_name] = sim\_data\_pca

        print(f"PCA simulation ({int(variance\_explained\*100)}% variance) completed in {timings[cov\_name][method\_name]:.2f} seconds.")

# Calculate covariance of simulated values

simulated\_covariances = {}

for cov\_name in covariance\_matrices.keys():

    simulated\_covariances[cov\_name] = {}

    for method\_name, sim\_data in simulation\_results[cov\_name].items():

        sim\_cov = np.cov(sim\_data, rowvar=False)

        simulated\_covariances[cov\_name][method\_name] = sim\_cov

# Compare simulated covariance to input covariance using Frobenius Norm

frobenius\_norms = {}

for cov\_name, input\_cov in covariance\_matrices.items():

    frobenius\_norms[cov\_name] = {}

    for method\_name, sim\_cov in simulated\_covariances[cov\_name].items():

        fro\_norm = norm(input\_cov - sim\_cov, 'fro')

        frobenius\_norms[cov\_name][method\_name] = fro\_norm

# Print timings

print("\nTimings (in seconds):")

for cov\_name, methods in timings.items():

    print(f"\nCovariance Matrix: {cov\_name}")

    for method\_name, timing in methods.items():

        print(f"  {method\_name}: {timing:.2f} seconds")

# Print Frobenius Norms

print("\nFrobenius Norms:")

for cov\_name, methods in frobenius\_norms.items():

    print(f"\nCovariance Matrix: {cov\_name}")

    for method\_name, fro\_norm in methods.items():

        print(f"  {method\_name}: {fro\_norm:.4f}")