# Response

## Problem 1

#### Calculate exponentially weighted covariance matrix

Exponentially weight covariance puts more weights to recent data. Weights exponentially decrease from the newest to the lowest.

### 1. Generate exponential weights

Using the formula below, I generate some exponential weights based on the given lambda.

$$Weight_{t-i} = (1-\lambda)\lambda^{i-1}$$

$$We\hat{i}ght_{t-i} = \frac{w_{t-i}}{\sum_{j=1}^{n} w_{t-j}}$$

```
def exponentialWeights(size, lam):
    weight = []
    totalWeight = 0
    for i in range(size):
        weight.append((1-lam)*(lam**(i+1)))
        totalWeight += weight[i]
    for i in range(size):
        weight[i] = weight[i]/totalWeight
    return weight
```

### 2. Generate exponential covariance for X and Y

Given weights, I calculated the covariance between X and Y.

$$Cov_{xy} = w_n * (x_1 - \bar{x})(y_1 - \bar{y}) + w_{n-1} * (x_2 - \bar{x})(y_2 - \bar{y}) + .. + w_1 * (x_n - \bar{x})(y_n - \bar{y})$$

```
def expCovForPair(weight, x, y):
    xMean = np.mean(x)
    yMean = np.mean(y)
    cov = 0
    for i in range(len(weight)):
        cov += weight[len(weight)-i-1]*(x[i] - xMean) * (y[i] - yMean)
    return cov
```

## 3. Generate exponential covariance matrix

Simply call the expCovForPair to fill in the covariance matrix.

```
def expCovForFrame(data, lam):
    weight = exponentialWeights(len(data), lam)
    covMatrix = pd.DataFrame(np.zeros((len(data.columns),
len(data.columns))))
    for i in range(len(data.columns)):
        x = data.iloc[:, i]
        covMatrix.iloc[i, i] = expCovForPair(weight, x, x)
        for j in range(i+1, len(data.columns)):
            y = data.iloc[:, j]
            covMatrix.iloc[i, j] = expCovForPair(weight, x, y)
            covMatrix.iloc[j, i] = covMatrix.iloc[i, j]
    return covMatrix
```

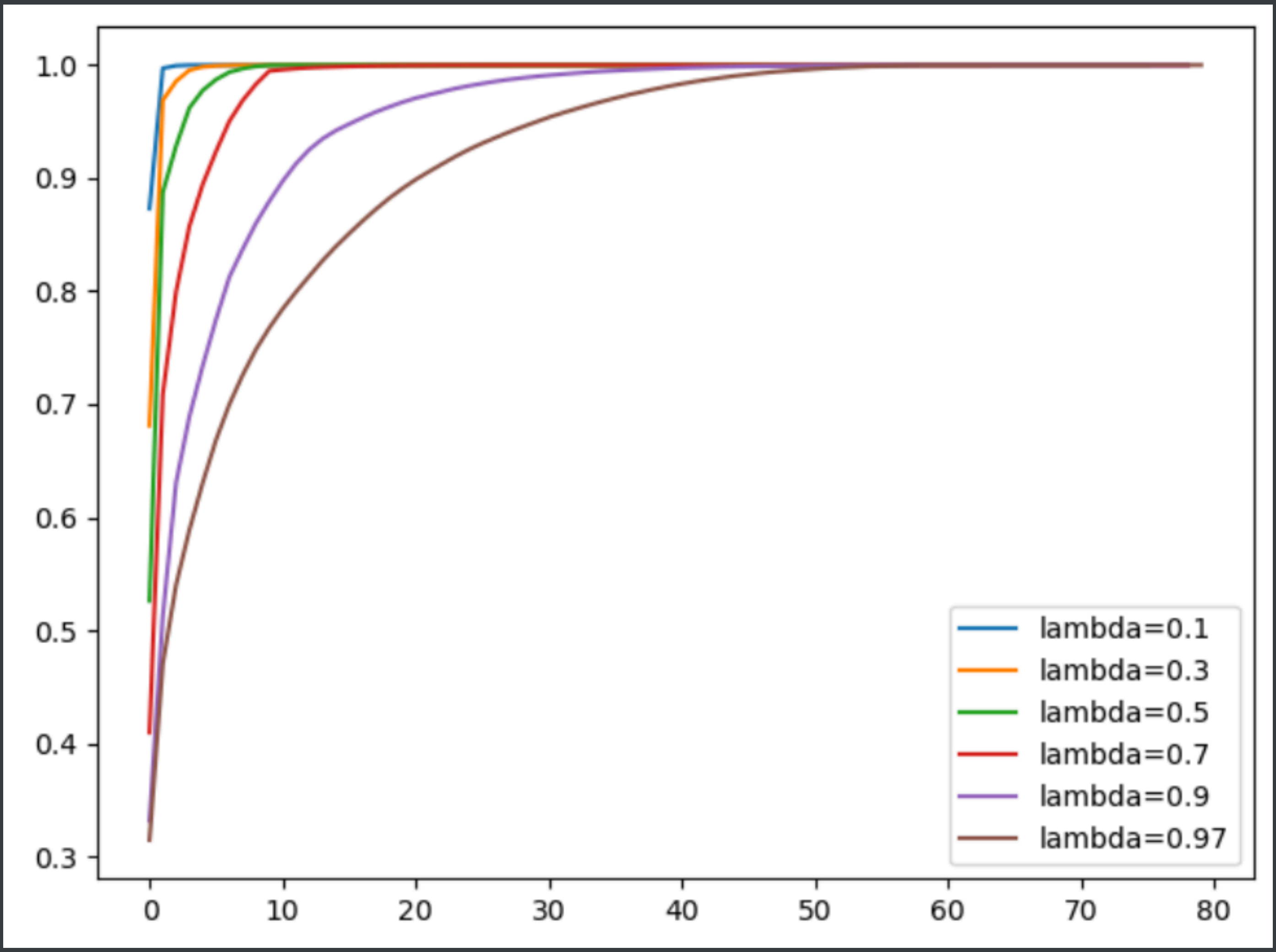
## Use PCA and plot the cumulative variance explained by each eigenvalue

I first use the eigh built-in function to calculate the eigenvalues and eigenvectors of the covariance matrix. Then, I cumulate the eigenvalues from the largest to the smallest to calculate the explanation power.

$$PctExplained(K) = \frac{\sum_{i=1}^{k} L_i}{\sum_{j=i}^{m} L_j}$$

```
def pcaExplained(covMatrix):
    eigenvalue, eigenvector = numpy.linalg.eigh(covMatrix)
    eigenSum = np.sum(eigenvalue)
    totalEigen = 0
    cumEigen = []
    for i in range(len(eigenvalue)-1, -1, -1):
        if eigenvalue[i] < 0:
            break
        totalEigen += eigenvalue[i]
        cumEigen.append(totalEigen/eigenSum)
    return cumEigen
```

The percentages explained given different different lambdas are shown below:



The larger the lambda, the slower the explanation power converges to 1.

## Problem 2

### Implement chol_psd() and near_psd()

#### CholeskyPSD

Following the formula below, I construct the Cholesky root.

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{32} \\ 0 & 0 & L_{33} \end{pmatrix}$$

$$= \begin{pmatrix} L_{11}^2 & & \text{(symmetric)} \\ L_{21}L_{11} & L_{21}^2 + L_{22}^2 & \\ L_{31}L_{11} & L_{31}L_{21} + L_{32}L_{22} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{pmatrix},$$

$$\mathbf{L} = \begin{pmatrix} \sqrt{A_{11}} & 0 & 0 \\ A_{21}/L_{11} & \sqrt{A_{22} - L_{21}^2} & 0 \\ A_{31}/L_{11} & \left(A_{32} - L_{31}L_{21}\right)/L_{22} & \sqrt{A_{33} - L_{31}^2 - L_{32}^2} \end{pmatrix}$$

```
def CholeskyPSD(A):
    A = pd.DataFrame(A)
    L = pd.DataFrame(np.zeros((len(A), len(A))))
    for j in range(len(A)):
        diagMinus = 0
        if j > 0:
            diagMinus = (L.iloc[j, 0:j] * L.iloc[j, 0:j]).sum()
        tempDiag = A.iloc[j, j] - diagMinus
        if -1e-8 <= tempDiag <= 0:
            tempDiag = 0
        L.iloc[j, j] = np.sqrt(tempDiag)
        if -1e-8 <= tempDiag <= 1e-8:
            L.iloc[j, (j + 1):] = 0
            continue
        else:
            for i in range(j + 1, len(A)):
                offDiagMinus = 0
                if j > 0:
                    offDiagMinus = (L.iloc[i, 0:j] * L.iloc[j, 0:j]).sum()
                L.iloc[i, j] = (A.iloc[i, j] - offDiagMinus) / L.iloc[j,
j]
    return L
```

## NearPSD

The nearPSD function finds a near PSD correlation matrix for the non-PSD correlation matrix by replacing its negative eigenvalues with 0 and multiplying the eigenvector matrix with a scaling matrix.

$$CS = \Lambda S$$

$$\Lambda^{'} : \lambda_i^{'} = max(\lambda_i, 0)$$

$$B = \sqrt{T} S \sqrt{\Lambda^{'}}$$

$$BB^T = \hat{C}$$

```
def nearPSD(A):
    invSD, A = covToCorr(A)
    eigenvalue, eigenvector = numpy.linalg.eigh(A)
    for i in range(len(eigenvalue)):
        if eigenvalue[i] < 0:
            eigenvalue[i] = 0
    scalingMatrix = 1 / np.matmul((eigenvector * eigenvector), eigenvalue)
    scalingMatrix = np.diag(scalingMatrix)

    eigenvalueMatrix = np.diag(eigenvalue)
    factor = np.matmul(np.matmul(np.sqrt(scalingMatrix), eigenvector),
np.sqrt(eigenvalueMatrix))
    nearCorr = np.matmul(factor, np.transpose(factor))

    if invSD is not None:
        invSD = np.diag(1 / np.diag(invSD))
        nearCorr = np.matmul(np.matmul(invSD, nearCorr), invSD)
    return nearCorr
```

## Implement Higham's 2002 nearest psd correlation function

Higham algorithm finds the nearest PSD correlation matrix by projecting the initial matrix twice. The first projection replaces the diagonal matrix to be 1, and the second projection replaces the negative eigenvalues to be 0.

$$\Delta S_0 = 0, Y_0 = A, \gamma_0 = maxFloat$$

Loop from k = 1 to max Iterations

$$R_k = Y_{k-1} - \Delta S_{k-1}$$

$$X_k = P_S(R_k)$$

$$\Delta S_k = X_k - R_k$$

$$Y_k = P_U(X_k)$$

$$\gamma_k = \gamma(Y_k)$$

If

$$\gamma(Y_k) - \gamma(Y_{k-1}) < tolerance$$

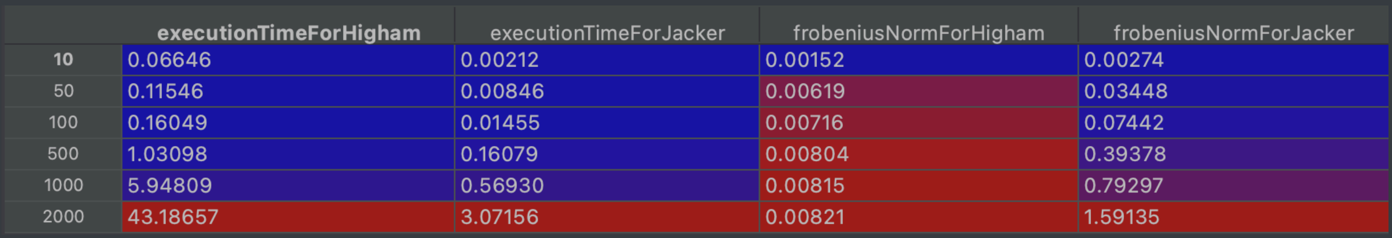
Then break.

```
def highamNearPSD(A, maxIterations, tolerance):
    gamma = float("inf")
    Y = A
    deltaS = pd.DataFrame(np.zeros((len(A), len(A))))
```

```
        for k in range(maxIterations):
            R = Y - deltaS
            X = secondProjection(R)
            deltaS = X - R
            Y = firstProjection(X)
            tempGamma = frobeniusNorm(Y - A)
            if np.abs(tempGamma - gamma) < tolerance:
                print("Convergence succeeds.")
                break
            gamma = tempGamma
        return pd.DataFrame(Y)
```

To confirm the matrix is PSD after using nearPSD() and highamNearPSD(), I pass in the output matrix to CholeskyPSD(), and realize that there is no NAN values. This means that my output matrix is PSD.

## Compare the results of both using the Frobenius Norm and the run time



| | executionTimeForHigham | executionTimeForJacker | frobeniusNormForHigham | frobeniusNormForJacker |
|---|---|---|---|---|
| 10 | 0.06646 | 0.00212 | 0.00152 | 0.00274 |
| 50 | 0.11546 | 0.00846 | 0.00619 | 0.03448 |
| 100 | 0.16049 | 0.01455 | 0.00716 | 0.07442 |
| 500 | 1.03098 | 0.16079 | 0.00804 | 0.39378 |
| 1000 | 5.94809 | 0.56930 | 0.00815 | 0.79297 |
| 2000 | 43.18657 | 3.07156 | 0.00821 | 1.59135 |

Given an n, the execution time for Higham algorithm is always larger than that for Near PSD algorithm. As n increases, the execution time for Higham algorithm and Near PSD algorithm increase, but the execution time for Higham algorithm increases faster than the execution time for Near PSD. This is because the time complexity of Higham algorithm is n square, but the time complexity of Near PSD algorithm is n.

Given an n, the Frobenius norm for Higham algorithm is always smaller than that for Near PSD algorithm. As n increases, the Frobenius norm for Higham algorithm and Near PSD algorithm increase, but the Frobenius norm for Higham algorithm increases slower than the Frobenius norm for Near PSD.

### pros and cons of each method

#### Near PSD:

Pros: low time complexity, quick to run

Cons: relatively low accuracy

#### Higham:

Pros: high accuracy

Cons: high time complexity, slow to run

## Problem 3

I answered this problem in 5 steps:

1. Calculate the standard deviation, EW standard deviation, Pearson correlation, and EW Pearson correlation from the covariance matrix of 101 stocks.
2. Construct 4 different covariance matrices using the standard deviations and correlations matrices above.
3. Use CholeskyPSD or PCA to construct the root. In terms of PCA, small eigenvalues are omitted once the explanation power exceeds a certain level.
4. Substitute the root into the multinormal simulation process to calculate 25000 draws of random data using

$$X = LZ + \mu$$

$$Z_i \sim N(0,1)$$

$$\Sigma = LL^{'}$$

5. After simulating 25000 draws of data, calculate its covariance matrix and compare it with the 4 covariance matrices using Frobenius norm.

## Execution Time

```
           directSimulation    PCA100%     PCA75%     PCA50%
PCorr-PStd         3.470515  0.083920  0.035165  0.026061
PCorr-EStd         3.777353  0.132134  0.034197  0.139038
ECorr-PStd         3.855314  0.073272  0.030294  0.022285
ECorr-EStd         3.336744  0.082531  0.031152  0.024521
```

## Frobenius Norm

```
           directSimulation       PCA100%     PCA75%     PCA50%
PCorr-PStd     4.496674e-08  5.494022e-08  0.000003  0.000011
PCorr-EStd     5.075239e-08  6.556066e-08  0.000003  0.000013
ECorr-PStd     4.792984e-08  4.728301e-08  0.000002  0.000010
ECorr-EStd     4.541466e-08  4.721879e-08  0.000002  0.000012
```

The Execution time for direct simulation is the longest. The lower the explanation power, the shorter the execution time.

The accuracy for direct simulation and PCA with 100% explained has the highest accuracy. The lower the explanation power, the worse the accuracy.

Therefore, there is a tradeoff between execution time and accuracy. Generally, inaccurate results need less time.