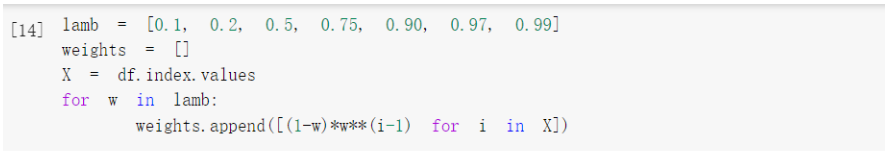
Fintech545 Project Week3

Kaiye Chen

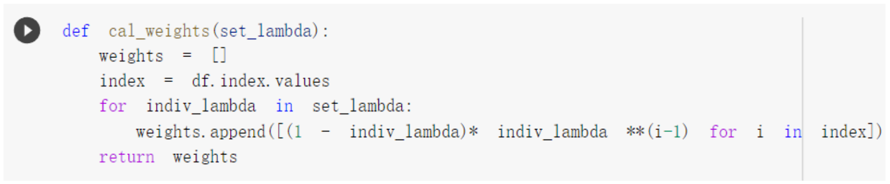
September 2022

1 Problem1

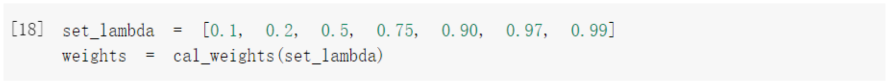
1.1 Calculate exponential weights



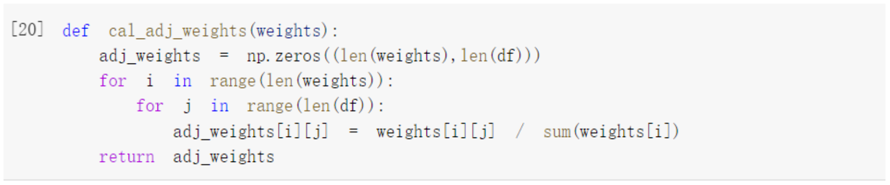
```
[14] lamb = [0.1, 0.2, 0.5, 0.75, 0.90, 0.97, 0.99]
     weights = []
     X = df.index.values
     for w in lamb:
         weights.append([(1-w)*w**(i-1) for i in X])
```



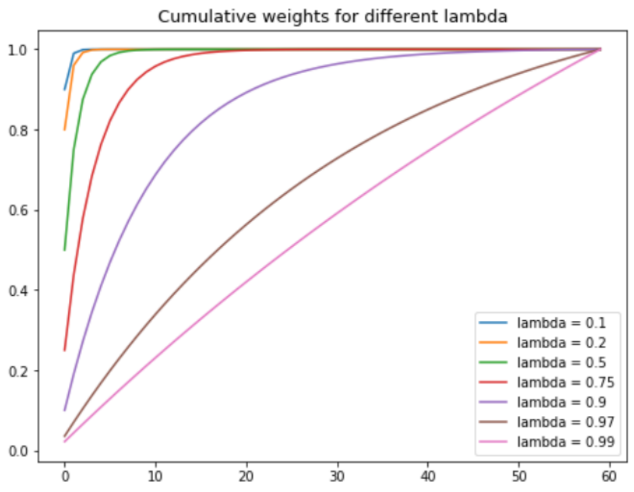
```
def cal_weights(set_lambda):
    weights = []
    index = df.index.values
    for indiv_lambda in set_lambda:
        weights.append([(1 - indiv_lambda)* indiv_lambda **(i-1) for i in index])
    return weights
```



```
[18] set_lambda = [0.1, 0.2, 0.5, 0.75, 0.90, 0.97, 0.99]
     weights = cal_weights(set_lambda)
```

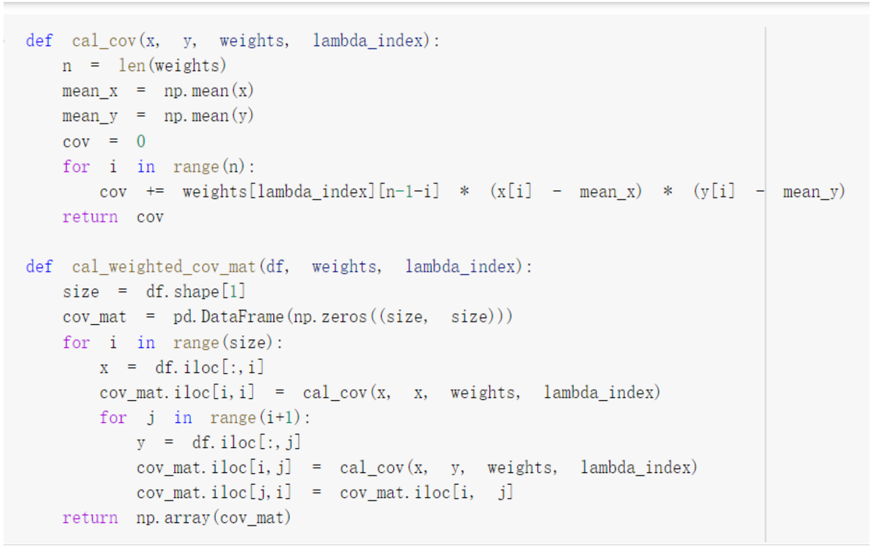


```
[20] def cal_adj_weights(weights):
         adj_weights = np.zeros((len(weights),len(df)))
         for i in range(len(weights)):
             for j in range(len(df)):
                 adj_weights[i][j] = weights[i][j] / sum(weights[i])
         return adj_weights
```



Here is my method to implement exponential weights. In order to be more conveniently used in the future, I defined all my work this week as functions. The result plot shows that with lambda increases, the portion gives to current day become lower, so the line converge to 1 with a lower speed.

1.2 Calculate covariance matrix



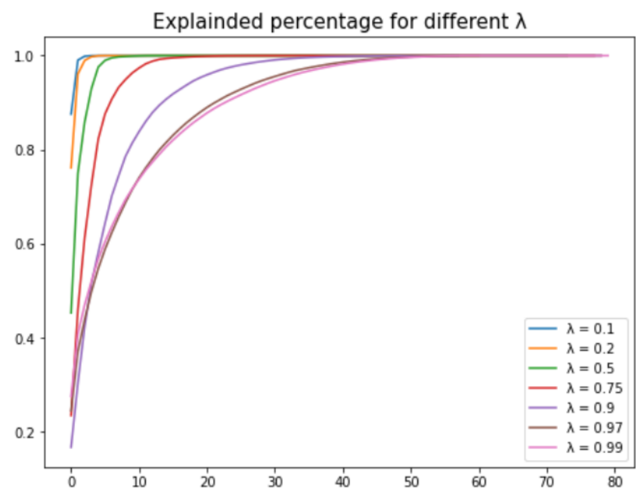
```
def cal_cov(x, y, weights, lambda_index):
    n = len(weights)
    mean_x = np.mean(x)
    mean_y = np.mean(y)
    cov = 0
    for i in range(n):
        cov += weights[lambda_index][n-1-i] * (x[i] - mean_x) * (y[i] - mean_y)
    return cov

def cal_weighted_cov_mat(df, weights, lambda_index):
    size = df.shape[1]
    cov_mat = pd.DataFrame(np.zeros((size, size)))
    for i in range(size):
        x = df.iloc[:,i]
        cov_mat.iloc[i,i] = cal_cov(x, x, weights, lambda_index)
        for j in range(i+1):
            y = df.iloc[:,j]
            cov_mat.iloc[i,j] = cal_cov(x, y, weights, lambda_index)
            cov_mat.iloc[j,i] = cov_mat.iloc[i, j]
    return np.array(cov_mat)
```

My method based on the logic that first calculate the covariance for each two component, then place them in accurate position of the covariance matrix.

1.3 Calculate PCA and results

```
def cal_PCA(cov_matrix):
    eigenvalue, eigenvector = np.linalg.eigh(cov_matrix)
    eigenvalue = sorted(eigenvalue, reverse = True)
    total_eigen = np.sum(eigenvalue)
    explained = 0
    explained_arr = []
    for i in range(0, len(eigenvalue)):
        if eigenvalue[i] < 0:
            break
        explained += eigenvalue[i]
        explained_arr.append(explained / total_eigen)
    return np.array(explained_arr)
```

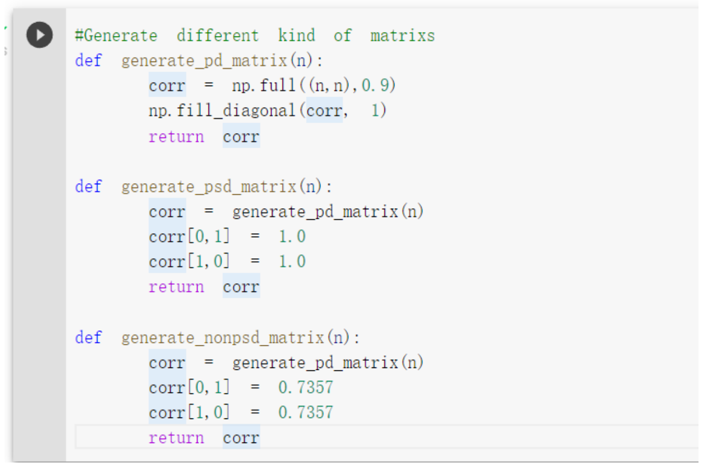
```
def plot_explained_ratio(df, weights, set_lambda):
    fig, pca_plot = plt.subplots(figsize=(8,6))
    for i in range(0, len(set_lambda)):
        cov_mat = cal_weighted_cov_mat(df, weights, i)
        PCA_results = cal_PCA(cov_mat)
        pca_plot.plot(PCA_results, label = f"λ = {set_lambda[i]}")
    pca_plot.legend(loc='best')
    pca_plot.set_title("Explainded percentage for different λ", fontsize=15)
    return
```



In this part, my method is based on very traditional Singular Value Decomposition(SVD) to get the eigenvalue and eigenvector. Then I sum and compare the explained percentage ratio. The comparing results shows that, similar to the result above, with lambda increases, the portion gives to current day become lower, so the line converge to 1 with a lower speed and the explained ratio for current day will also be comparatively lower.

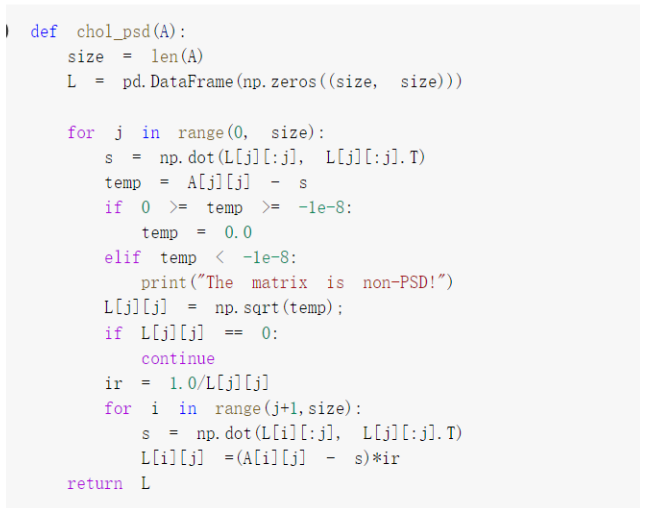
2 Problem2

2.1 Recurrent Functions

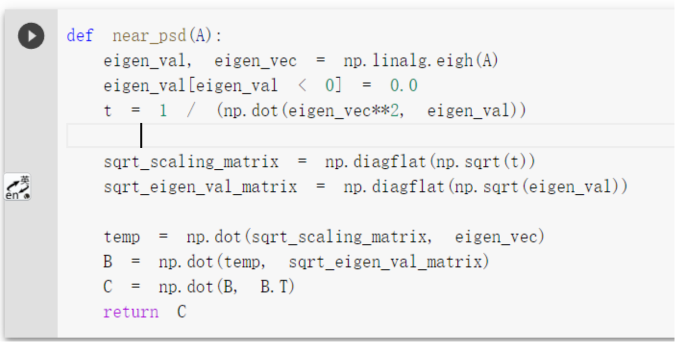


```
#Generate different kind of matrixs
def generate_pd_matrix(n):
        corr = np.full((n,n),0.9)
        np.fill_diagonal(corr, 1)
        return corr

def generate_psd_matrix(n):
        corr = generate_pd_matrix(n)
        corr[0,1] = 1.0
        corr[1,0] = 1.0
        return corr

def generate_nonpsd_matrix(n):
        corr = generate_pd_matrix(n)
        corr[0,1] = 0.7357
        corr[1,0] = 0.7357
        return corr
```

Firstly, I defined functions that generate different kinds of matrix in order for further usage.



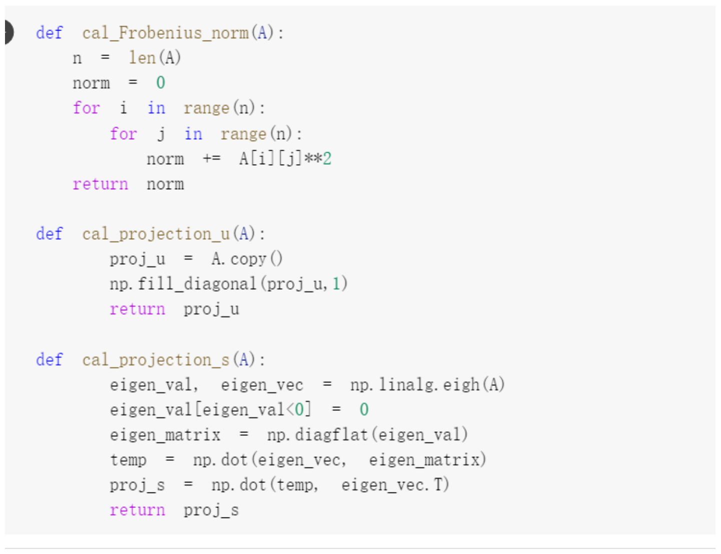
```
def chol_psd(A):
    size = len(A)
    L = pd.DataFrame(np.zeros((size, size)))

    for j in range(0, size):
        s = np.dot(L[j][:j], L[j][:j].T)
        temp = A[j][j] - s
        if 0 >= temp >= -1e-8:
            temp = 0.0
        elif temp < -1e-8:
            print("The matrix is non-PSD!")
        L[j][j] = np.sqrt(temp);
        if L[j][j] == 0:
            continue
        ir = 1.0/L[j][j]
        for i in range(j+1,size):
            s = np.dot(L[i][:j], L[j][:j].T)
            L[i][j] =(A[i][j] - s)*ir
    return L
```



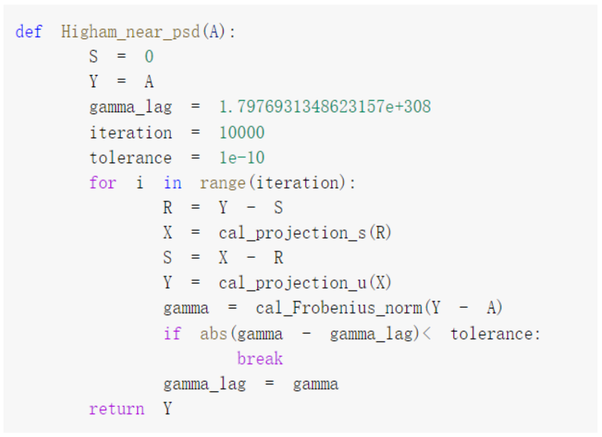
```
def near_psd(A):
    eigen_val, eigen_vec = np.linalg.eigh(A)
    eigen_val[eigen_val < 0] = 0.0
    t = 1 / (np.dot(eigen_vec**2, eigen_val))

    sqrt_scaling_matrix = np.diagflat(np.sqrt(t))
    sqrt_eigen_val_matrix = np.diagflat(np.sqrt(eigen_val))

    temp = np.dot(sqrt_scaling_matrix, eigen_vec)
    B = np.dot(temp, sqrt_eigen_val_matrix)
    C = np.dot(B, B.T)
    return C
```

In this part, I recurrent the *chol_psd*(), and *near_psd*() functions from the course repository using python.

2.2 Implement Higham



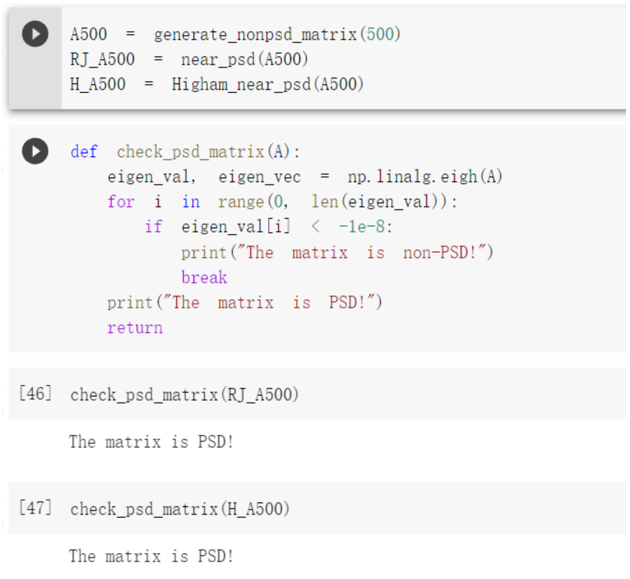
```
def cal_Frobenius_norm(A):
    n = len(A)
    norm = 0
    for i in range(n):
        for j in range(n):
            norm += A[i][j]**2
    return norm

def cal_projection_u(A):
        proj_u = A.copy()
        np.fill_diagonal(proj_u,1)
        return proj_u

def cal_projection_s(A):
        eigen_val, eigen_vec = np.linalg.eigh(A)
        eigen_val[eigen_val<0] = 0
        eigen_matrix = np.diagflat(eigen_val)
        temp = np.dot(eigen_vec, eigen_matrix)
        proj_s = np.dot(temp, eigen_vec.T)
        return proj_s
```



```python
def Higham_near_psd(A):
    S = 0
    Y = A
    gamma_lag = 1.7976931348623157e+308
    iteration = 10000
    tolerance = 1e-10
    for i in range(iteration):
        R = Y - S
        X = cal_projection_s(R)
        S = X - R
        Y = cal_projection_u(X)
        gamma = cal_Frobenius_norm(Y - A)
        if abs(gamma - gamma_lag)< tolerance:
            break
        gamma_lag = gamma
    return Y
```

In this part, I implement the Higham method based on the math in our slides, and I set the first gamma lag to the largest float in python.

2.3 Check the matrices are PSD or not



```python
A500 = generate_nonpsd_matrix(500)
RJ_A500 = near_psd(A500)
H_A500 = Higham_near_psd(A500)
```

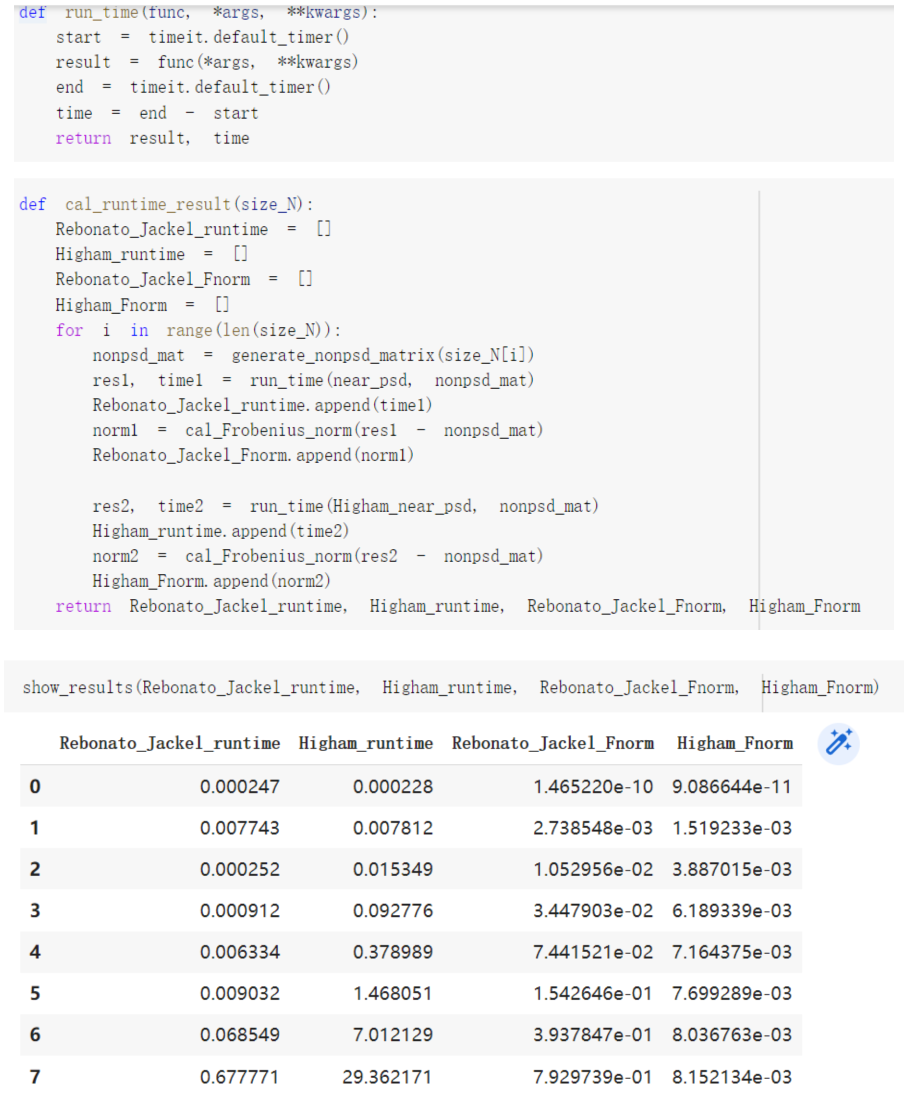
```python
def check_psd_matrix(A):
    eigen_val, eigen_vec = np.linalg.eigh(A)
    for i in range(0, len(eigen_val)):
        if eigen_val[i] < -1e-8:
            print("The matrix is non-PSD!")
            break
    print("The matrix is PSD!")
    return
```

```
[46] check_psd_matrix(RJ_A500)

The matrix is PSD!
```

```
[47] check_psd_matrix(H_A500)

The matrix is PSD!
```

In this part, I defined functions to check if after two methods' translation, the matrices will be PSD or not. I set the tolerance of eigen value to -1e-8. The results shows that two method will both produce PSD matrices.

2.4 Comparing methods

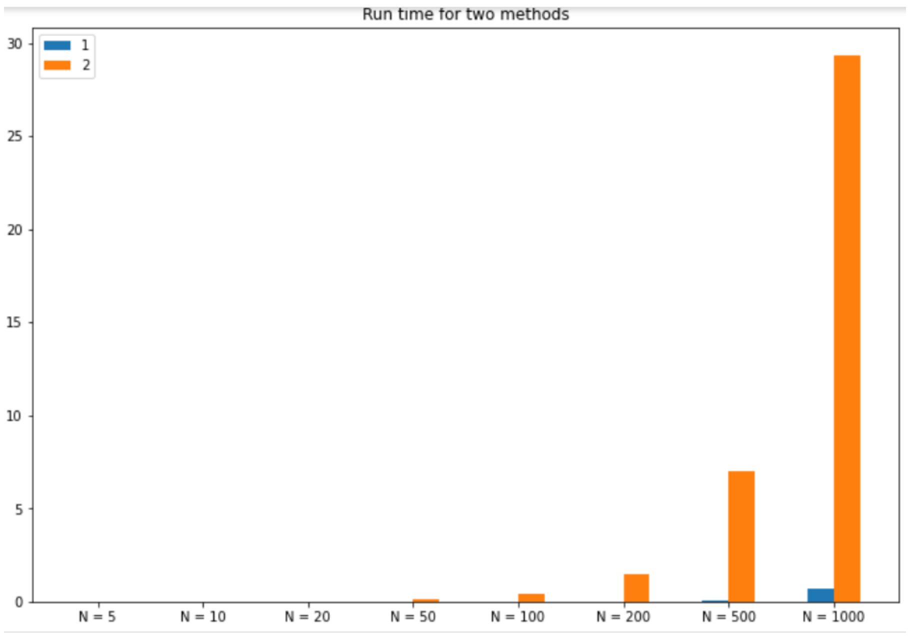


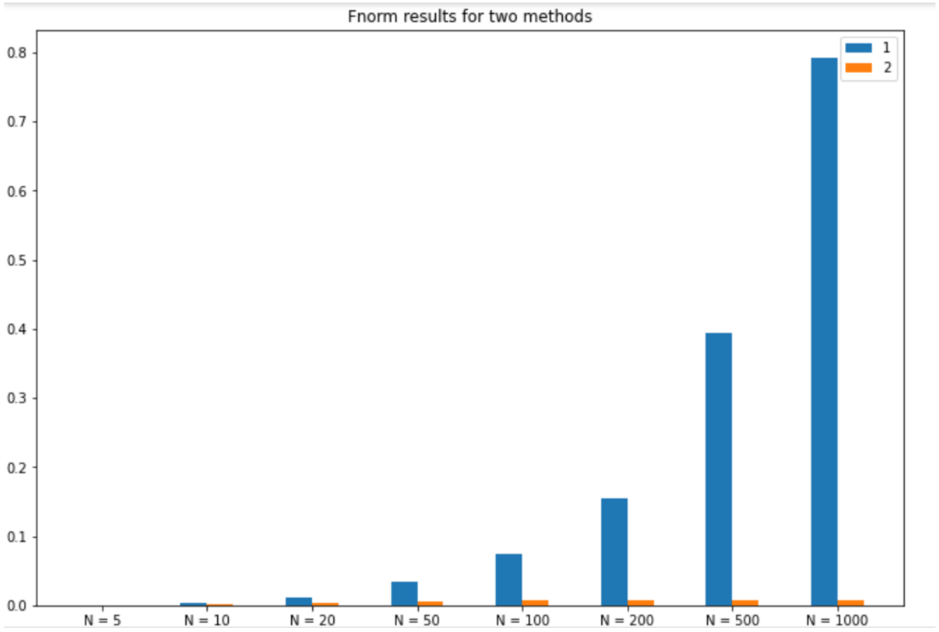
```
def run_time(func, *args, **kwargs):
    start = timeit.default_timer()
    result = func(*args, **kwargs)
    end = timeit.default_timer()
    time = end - start
    return result, time
```

```
def cal_runtime_result(size_N):
    Rebonato_Jackel_runtime = []
    Higham_runtime = []
    Rebonato_Jackel_Fnorm = []
    Higham_Fnorm = []
    for i in range(len(size_N)):
        nonpsd_mat = generate_nonpsd_matrix(size_N[i])
        res1, time1 = run_time(near_psd, nonpsd_mat)
        Rebonato_Jackel_runtime.append(time1)
        norm1 = cal_Frobenius_norm(res1 - nonpsd_mat)
        Rebonato_Jackel_Fnorm.append(norm1)

        res2, time2 = run_time(Higham_near_psd, nonpsd_mat)
        Higham_runtime.append(time2)
        norm2 = cal_Frobenius_norm(res2 - nonpsd_mat)
        Higham_Fnorm.append(norm2)
    return Rebonato_Jackel_runtime, Higham_runtime, Rebonato_Jackel_Fnorm, Higham_Fnorm
```

```
show_results(Rebonato_Jackel_runtime, Higham_runtime, Rebonato_Jackel_Fnorm, Higham_Fnorm)
```

	Rebonato_Jackel_runtime	Higham_runtime	Rebonato_Jackel_Fnorm	Higham_Fnorm
0	0.000247	0.000228	1.465220e-10	9.086644e-11
1	0.007743	0.007812	2.738548e-03	1.519233e-03
2	0.000252	0.015349	1.052956e-02	3.887015e-03
3	0.000912	0.092776	3.447903e-02	6.189339e-03
4	0.006334	0.378989	7.441521e-02	7.164375e-03
5	0.009032	1.468051	1.542646e-01	7.699289e-03
6	0.068549	7.012129	3.937847e-01	8.036763e-03
7	0.677771	29.362171	7.929739e-01	8.152134e-03



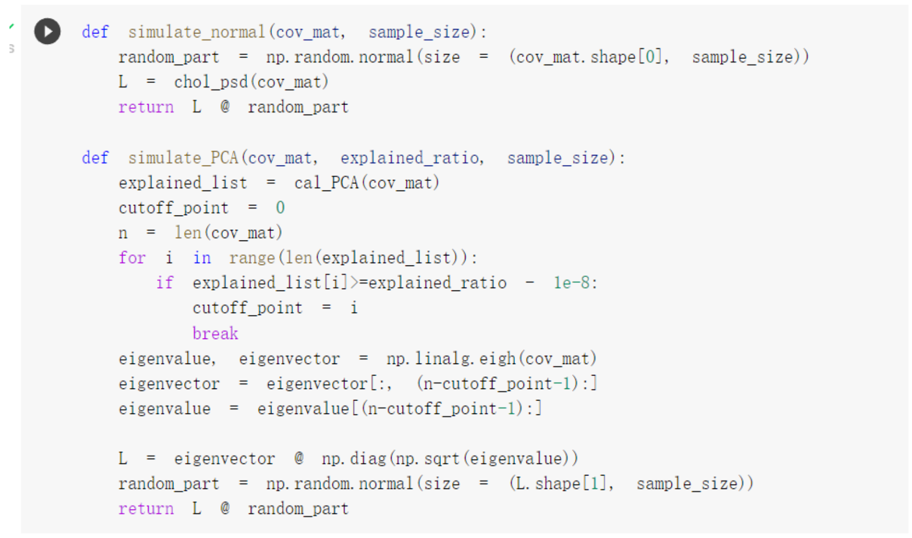


In this part, I used the package timeit to get the run time length of two methods. In the results, I first print the run time and Frobenius Norm as a dataframe. In this dataframe, we can see that the with the size of matrices N increases, the run time of Rebonato_Jackel are significantly lower than Higham.

However, the Frobenius Norm of Rebonato_Jackel are significantly higher. It indicates that there is a trade off between run time and Norm, in other words, we need to make a balance between computational cost and accuracy. Then in the plots we can see that with N increases, the run time and Fnorm increase as the N^2 speed, which may result from the 2-dimension structure of matrices.

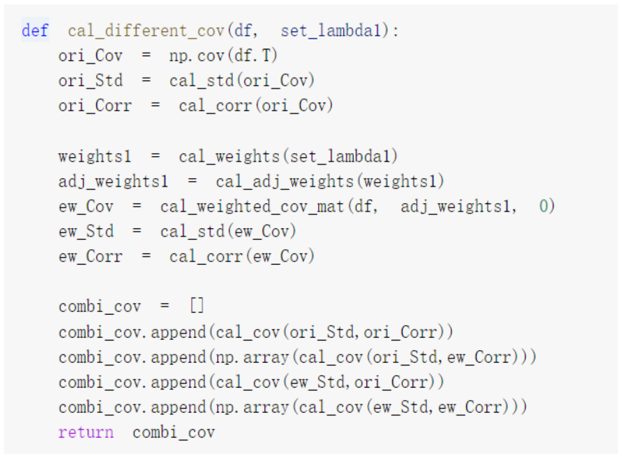
3 Problem3

3.1 Define simulation process

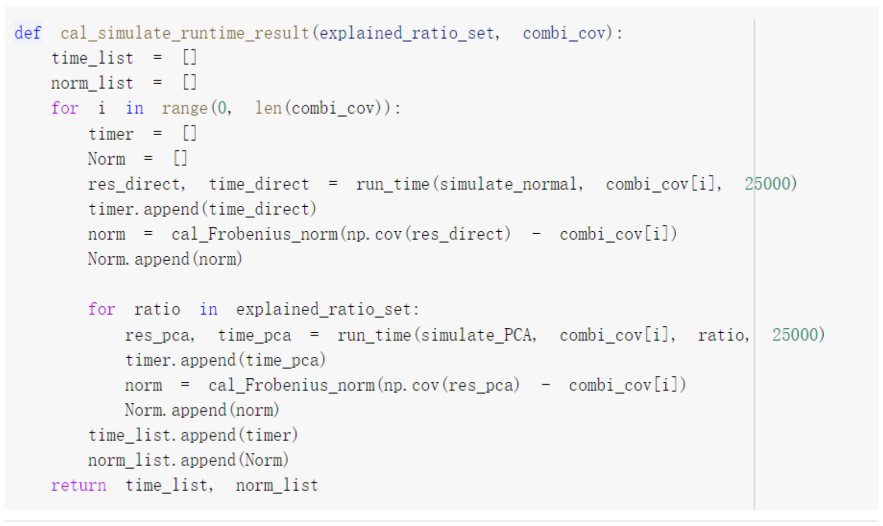


```python
def simulate_normal(cov_mat, sample_size):
    random_part = np.random.normal(size = (cov_mat.shape[0], sample_size))
    L = chol_psd(cov_mat)
    return L @ random_part

def simulate_PCA(cov_mat, explained_ratio, sample_size):
    explained_list = cal_PCA(cov_mat)
    cutoff_point = 0
    n = len(cov_mat)
    for i in range(len(explained_list)):
        if explained_list[i]>=explained_ratio - 1e-8:
            cutoff_point = i
            break
    eigenvalue, eigenvector = np.linalg.eigh(cov_mat)
    eigenvector = eigenvector[:, (n-cutoff_point-1):]
    eigenvalue = eigenvalue[(n-cutoff_point-1):]

    L = eigenvector @ np.diag(np.sqrt(eigenvalue))
    random_part = np.random.normal(size = (L.shape[1], sample_size))
    return L @ random_part
```

In this part, I used different covariance matrices as input to do the simulation. Based on the sample size, I used random function to generate a random normal matrix, then dot two matrix to get the simulation result.



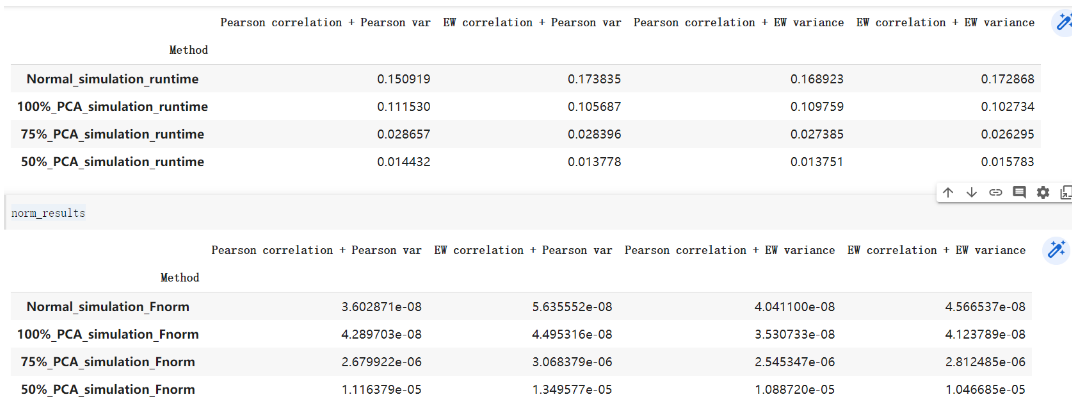
```python
def cal_different_cov(df, set_lambda1):
    ori_Cov = np.cov(df.T)
    ori_Std = cal_std(ori_Cov)
    ori_Corr = cal_corr(ori_Cov)

    weights1 = cal_weights(set_lambda1)
    adj_weights1 = cal_adj_weights(weights1)
    ew_Cov = cal_weighted_cov_mat(df, adj_weights1, 0)
    ew_Std = cal_std(ew_Cov)
    ew_Corr = cal_corr(ew_Cov)

    combi_cov = []
    combi_cov.append(cal_cov(ori_Std,ori_Corr))
    combi_cov.append(np.array(cal_cov(ori_Std,ew_Corr)))
    combi_cov.append(cal_cov(ew_Std,ori_Corr))
    combi_cov.append(np.array(cal_cov(ew_Std,ew_Corr)))
    return combi_cov
```



```python
def  cal_simulate_runtime_result(explained_ratio_set,  combi_cov):
    time_list  =  []
    norm_list  =  []
    for  i  in  range(0,  len(combi_cov)):
        timer  =  []
        Norm  =  []
        res_direct,  time_direct  =  run_time(simulate_normal,  combi_cov[i],  25000)
        timer.append(time_direct)
        norm  =  cal_Frobenius_norm(np.cov(res_direct)  -  combi_cov[i])
        Norm.append(norm)

        for  ratio  in  explained_ratio_set:
            res_pca,  time_pca  =  run_time(simulate_PCA,  combi_cov[i],  ratio,  25000)
            timer.append(time_pca)
            norm  =  cal_Frobenius_norm(np.cov(res_pca)  -  combi_cov[i])
            Norm.append(norm)
        time_list.append(timer)
        norm_list.append(Norm)
    return  time_list,  norm_list
```

In this part, I calculate different covariance matrices based on different combinations of weighted and normal deviation. Then I call the former timer function to calculate the run time and norm.

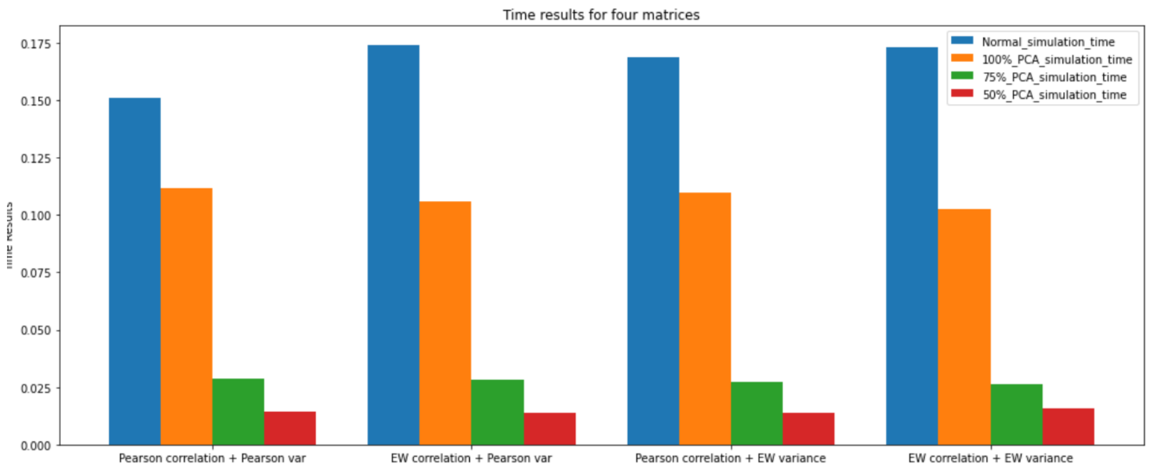
3.2 Simulation results

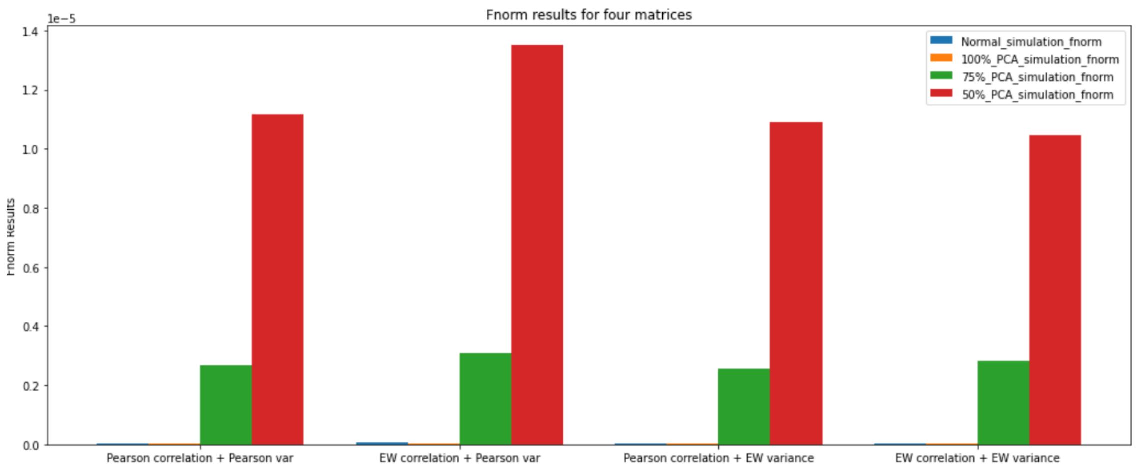


Method	Pearson correlation + Pearson var	EW correlation + Pearson var	Pearson correlation + EW variance	EW correlation + EW variance
Normal_simulation_runtime	0.150919	0.173835	0.168923	0.172868
100%_PCA_simulation_runtime	0.111530	0.105687	0.109759	0.102734
75%_PCA_simulation_runtime	0.028657	0.028396	0.027385	0.026295
50%_PCA_simulation_runtime	0.014432	0.013778	0.013751	0.015783

```
norm_results
```

Method	Pearson correlation + Pearson var	EW correlation + Pearson var	Pearson correlation + EW variance	EW correlation + EW variance
Normal_simulation_Fnorm	3.602871e-08	5.635552e-08	4.041100e-08	4.566537e-08
100%_PCA_simulation_Fnorm	4.289703e-08	4.495316e-08	3.530733e-08	4.123789e-08
75%_PCA_simulation_Fnorm	2.679922e-06	3.068379e-06	2.545347e-06	2.812485e-06
50%_PCA_simulation_Fnorm	1.116379e-05	1.349577e-05	1.088720e-05	1.046685e-05





In this final part, I show the final results of simulation. In this result, we can see very obvious negative relationship between run time and norm. This relationship strongly indicates that when we using PCA techniques to do dimension reduction, if we want to minimize the information loss, the computational cost will be really high. Instead, if we can increase our tolerance of information loss, we can save significant amount of computational power. It is nothing more than the traditional trade-off in statistics between accuracy and complexity.