Problem 1

Use the stock returns in DailyReturn.csv for this problem. DailyReturn.csv contains returns for 100 large US stocks and as well as the ETF, SPY which tracks the S&P500.

Create a routine for calculating an exponentially weighted covariance matrix. If you have a package that calculates it for you, verify that it calculates the values you expect. This means you still have to implement it.

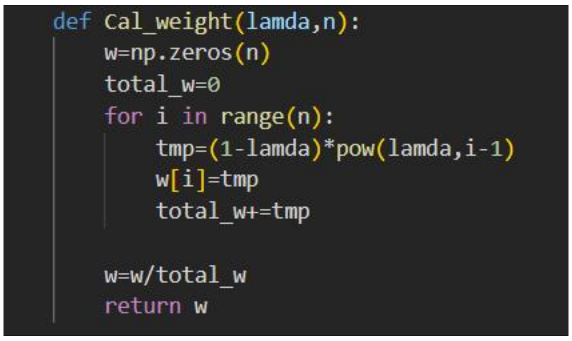
Vary λ ∈ (0, 1). Use PCA and plot the cumulative variance explained by each eigenvalue for each λ chosen.

What does this tell us about values of λ and the effect it has on the covariance matrix?

Answer:

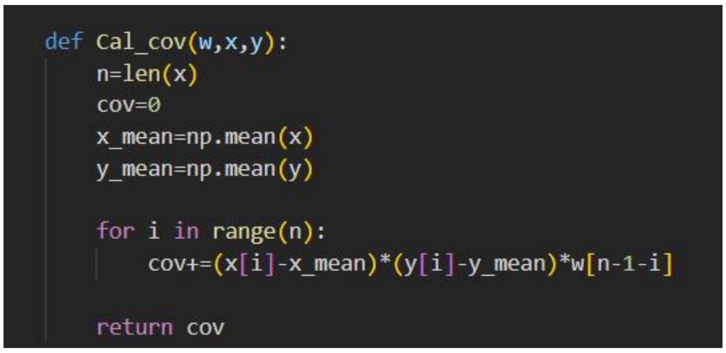
①The routine for calculating an exponentially weighted covariance matrix

<1> calculate the weight for different λ:

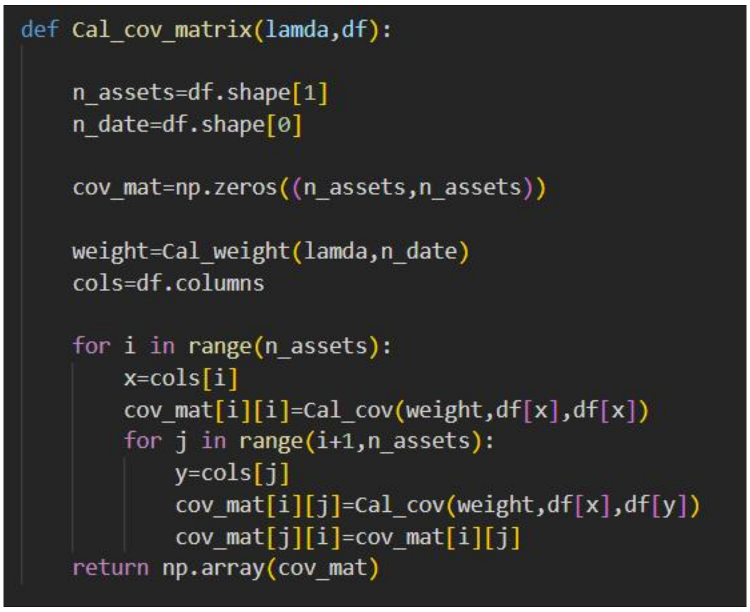


```
def Cal_weight(lamda,n):
    w=np.zeros(n)
    total_w=0
    for i in range(n):
        tmp=(1-lamda)*pow(lamda,i-1)
        w[i]=tmp
        total_w+=tmp

    w=w/total_w
    return w
```

<2>calculate the covariance matrix:

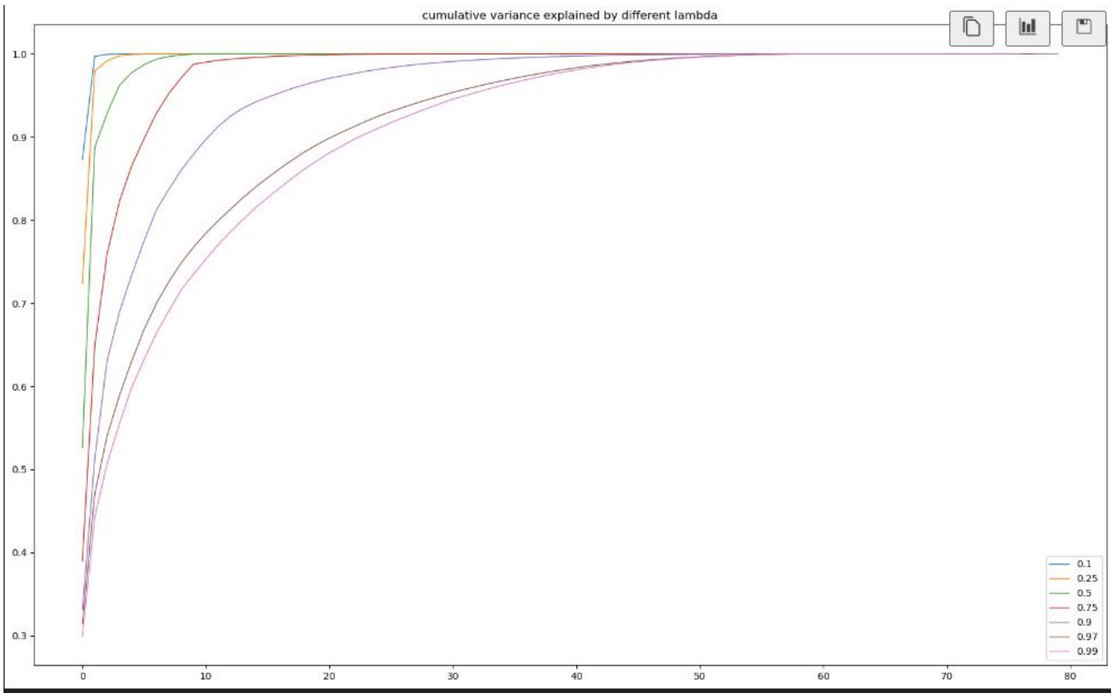


```
def Cal_cov(w,x,y):
    n=len(x)
    cov=0
    x_mean=np.mean(x)
    y_mean=np.mean(y)

    for i in range(n):
        cov+=(x[i]-x_mean)*(y[i]-y_mean)*w[n-1-i]

    return cov
```



```python
def Cal_cov_matrix(lamda,df):

    n_assets=df.shape[1]
    n_date=df.shape[0]

    cov_mat=np.zeros((n_assets,n_assets))

    weight=Cal_weight(lamda,n_date)
    cols=df.columns

    for i in range(n_assets):
        x=cols[i]
        cov_mat[i][i]=Cal_cov(weight,df[x],df[x])
        for j in range(i+1,n_assets):
            y=cols[j]
            cov_mat[i][j]=Cal_cov(weight,df[x],df[y])
            cov_mat[j][i]=cov_mat[i][j]
    return np.array(cov_mat)
```

② the cumulative variance explained by each eigenvalue for each λ chosen.



③What does this tell us about values of λ and the effect it has on the covariance matrix?

- If the λ is small, it means that we values less on the old data, but values more on the recent data. Vice versa, If the λ is larger, it means that we put higher weight on the old data comparing to the weight when λ is small .

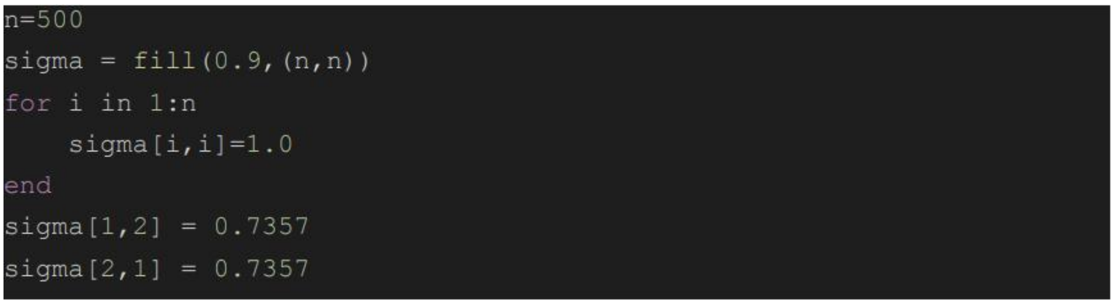
And according to the graph we plot, it shows that when λ becomes larger, it explained fewer percent, as the weight is more extreme, majorly focusing on recent data.

Problem 2

Copy the chol_psd(), and near_psd() functions from the course repository – implement in your programming language of choice. These are core functions you will need throughout the remainder of the class.

Implement Higham's 2002 nearest psd correlation function.

Generate a non-psd correlation matrix that is 500x500. You can use the code I used in class:



```
n=500
sigma = fill(0.9,(n,n))
for i in 1:n
    sigma[i,i]=1.0
end
sigma[1,2] = 0.7357
sigma[2,1] = 0.7357
```

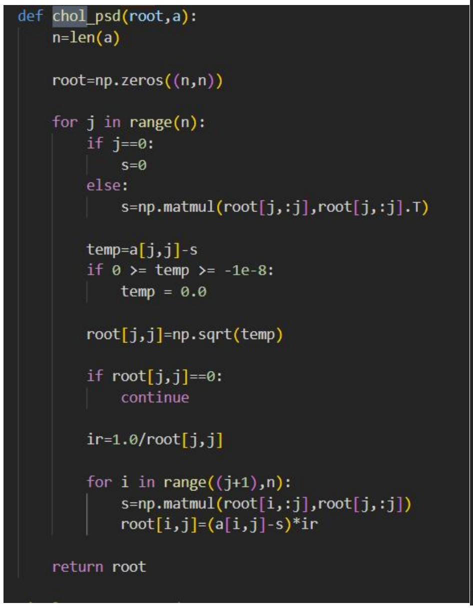
Use near_psd() and Higham's method to fix the matrix. Confirm the matrix is now PSD.

Compare the results of both using the Frobenius Norm. Compare the run time between the two. How does the run time of each function compare as N increases?

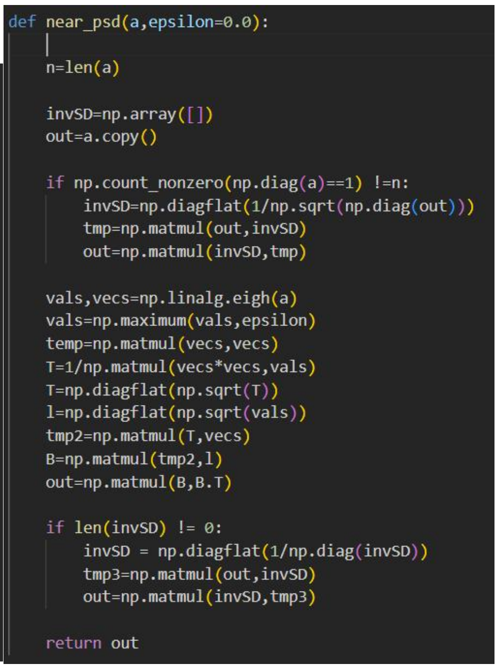
Based on the above, discuss the pros and cons of each method and when you would use each. There is no wrong answer here, I want you to think through this and tell me what you think.

Answer:

①implement chol_psd() & near_psd()

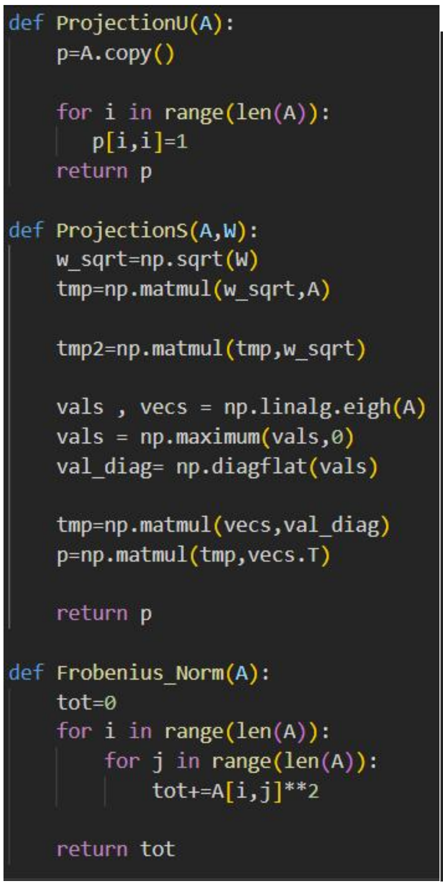


```
def chol_psd(root,a):
    n=len(a)

    root=np.zeros((n,n))

    for j in range(n):
        if j==0:
            s=0
        else:
            s=np.matmul(root[j,:j],root[j,:j].T)

        temp=a[j,j]-s
        if 0 >= temp >= -1e-8:
            temp = 0.0

        root[j,j]=np.sqrt(temp)

        if root[j,j]==0:
            continue

        ir=1.0/root[j,j]

        for i in range((j+1),n):
            s=np.matmul(root[i,:j],root[j,:j])
            root[i,j]=(a[i,j]-s)*ir

    return root
```

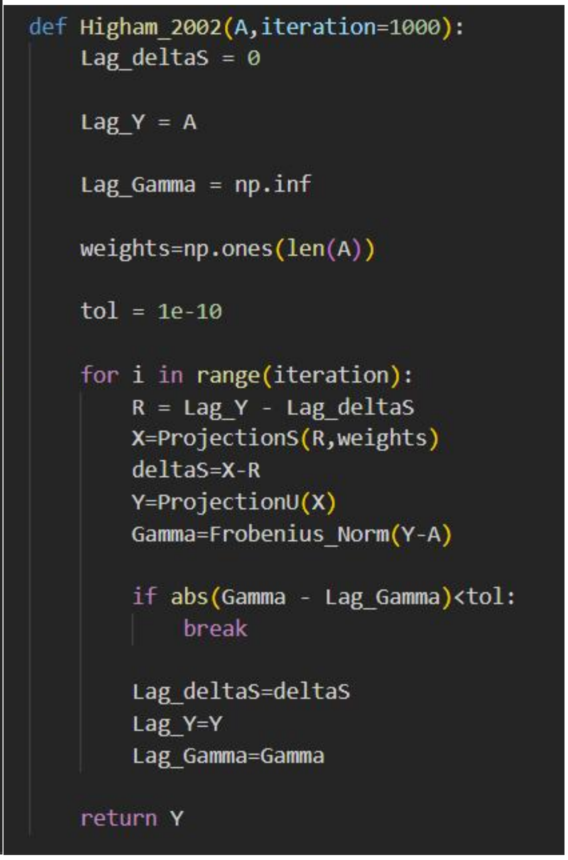


```
def near_psd(a,epsilon=0.0):

    n=len(a)

    invSD=np.array([])
    out=a.copy()

    if np.count_nonzero(np.diag(a)==1) !=n:
        invSD=np.diagflat(1/np.sqrt(np.diag(out)))
        tmp=np.matmul(out,invSD)
        out=np.matmul(invSD,tmp)

    vals,vecs=np.linalg.eigh(a)
    vals=np.maximum(vals,epsilon)
    temp=np.matmul(vecs,vecs)
    T=1/np.matmul(vecs*vecs,vals)
    T=np.diagflat(np.sqrt(T))
    l=np.diagflat(np.sqrt(vals))
    tmp2=np.matmul(T,vecs)
    B=np.matmul(tmp2,l)
    out=np.matmul(B,B.T)

    if len(invSD) != 0:
        invSD = np.diagflat(1/np.diag(invSD))
        tmp3=np.matmul(out,invSD)
        out=np.matmul(invSD,tmp3)

    return out
```

②Implement Higham's 2002

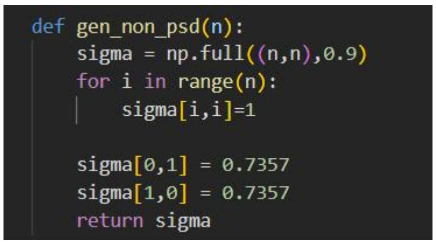


```python
def ProjectionU(A):
    p=A.copy()

    for i in range(len(A)):
        p[i,i]=1
    return p

def ProjectionS(A,W):
    w_sqrt=np.sqrt(W)
    tmp=np.matmul(w_sqrt,A)

    tmp2=np.matmul(tmp,w_sqrt)

    vals , vecs = np.linalg.eigh(A)
    vals = np.maximum(vals,0)
    val_diag= np.diagflat(vals)

    tmp=np.matmul(vecs,val_diag)
    p=np.matmul(tmp,vecs.T)

    return p

def Frobenius_Norm(A):
    tot=0
    for i in range(len(A)):
        for j in range(len(A)):
            tot+=A[i,j]**2

    return tot
```



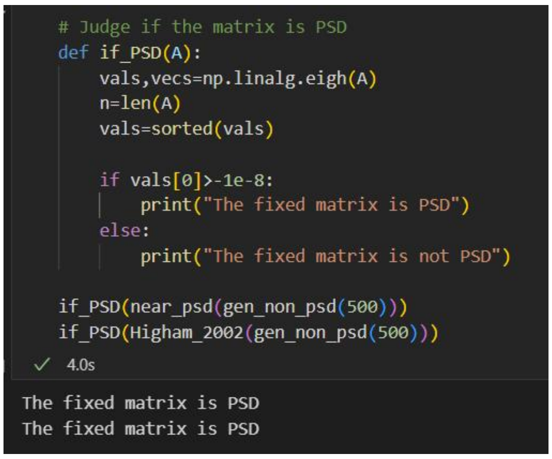
```python
def Higham_2002(A,iteration=1000):
    Lag_deltaS = 0

    Lag_Y = A

    Lag_Gamma = np.inf

    weights=np.ones(len(A))

    tol = 1e-10

    for i in range(iteration):
        R = Lag_Y - Lag_deltaS
        X=ProjectionS(R,weights)
        deltaS=X-R
        Y=ProjectionU(X)
        Gamma=Frobenius_Norm(Y-A)

        if abs(Gamma - Lag_Gamma)<tol:
            break

        Lag_deltaS=deltaS
        Lag_Y=Y
        Lag_Gamma=Gamma

    return Y
```

③generate non-psd：



```python
def gen_non_psd(n):
    sigma = np.full((n,n),0.9)
    for i in range(n):
        sigma[i,i]=1

    sigma[0,1] = 0.7357
    sigma[1,0] = 0.7357
    return sigma
```

④Use near_psd() and Higham's method to fix the matrix. Confirm the matrix is now PSD

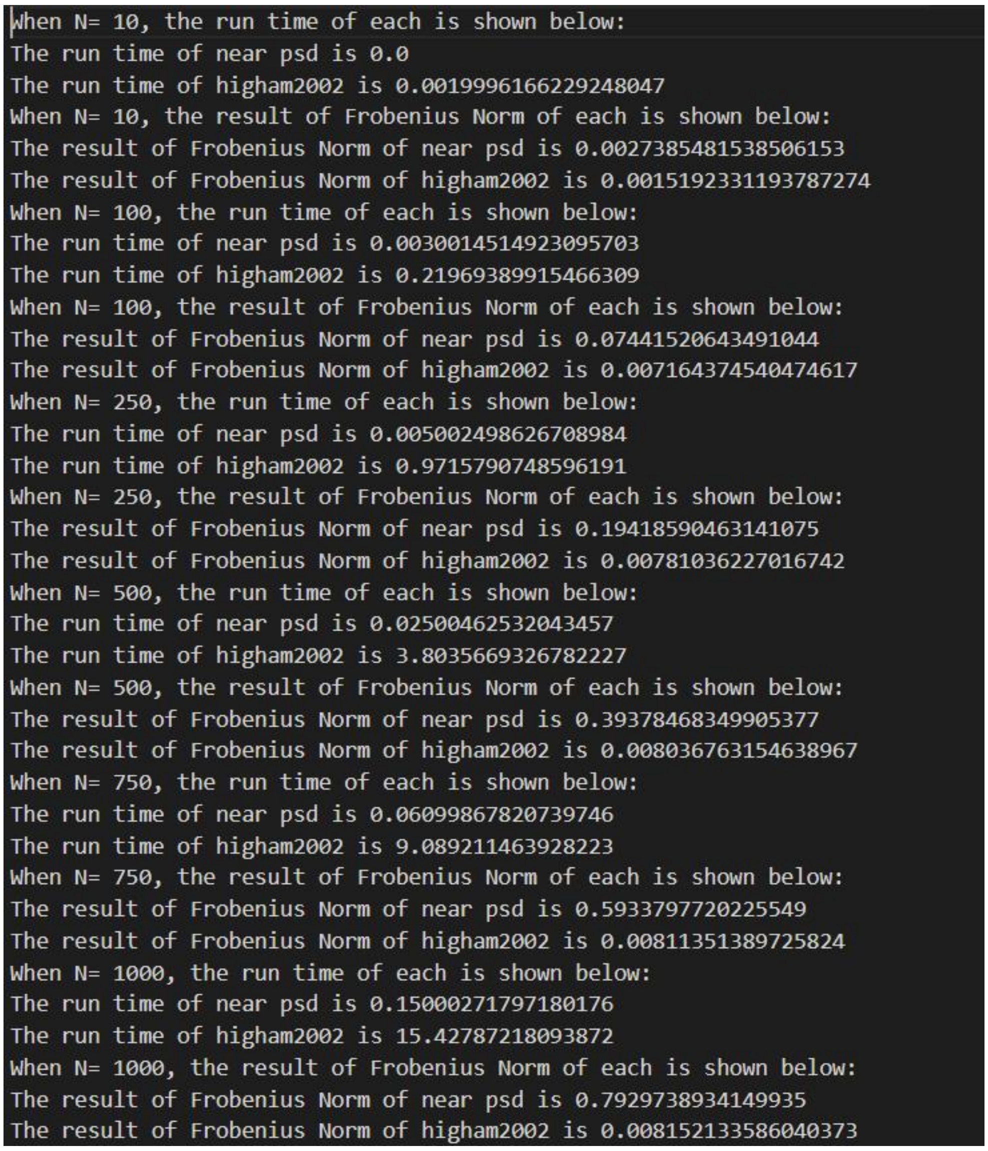


```python
# Judge if the matrix is PSD
def if_PSD(A):
    vals,vecs=np.linalg.eigh(A)
    n=len(A)
    vals=sorted(vals)

    if vals[0]>-1e-8:
        print("The fixed matrix is PSD")
    else:
        print("The fixed matrix is not PSD")

if_PSD(near_psd(gen_non_psd(500)))
if_PSD(Higham_2002(gen_non_psd(500)))
```

✓ 4.0s

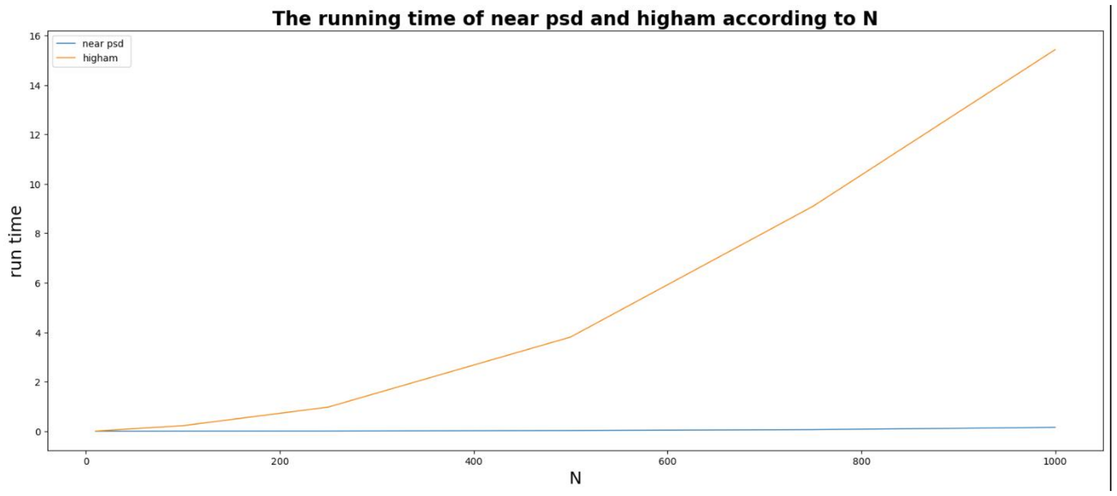
```
The fixed matrix is PSD
The fixed matrix is PSD
```

⑤ Compare the results of both using the Frobenius Norm

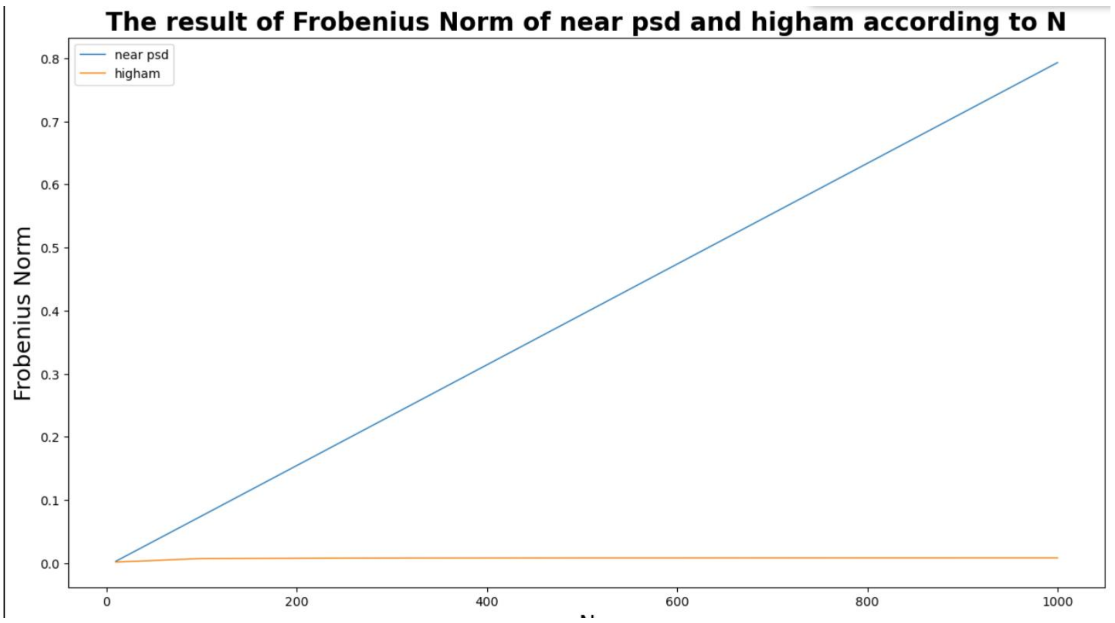


```
When N= 10, the run time of each is shown below:
The run time of near psd is 0.0
The run time of higham2002 is 0.0019996166229248047
When N= 10, the result of Frobenius Norm of each is shown below:
The result of Frobenius Norm of near psd is 0.0027385481538506153
The result of Frobenius Norm of higham2002 is 0.0015192331193787274
When N= 100, the run time of each is shown below:
The run time of near psd is 0.0030014514923095703
The run time of higham2002 is 0.21969389915466309
When N= 100, the result of Frobenius Norm of each is shown below:
The result of Frobenius Norm of near psd is 0.07441520643491044
The result of Frobenius Norm of higham2002 is 0.007164374540474617
When N= 250, the run time of each is shown below:
The run time of near psd is 0.005002498626708984
The run time of higham2002 is 0.9715790748596191
When N= 250, the result of Frobenius Norm of each is shown below:
The result of Frobenius Norm of near psd is 0.19418590463141075
The result of Frobenius Norm of higham2002 is 0.00781036227016742
When N= 500, the run time of each is shown below:
The run time of near psd is 0.02500462532043457
The run time of higham2002 is 3.8035669326782227
When N= 500, the result of Frobenius Norm of each is shown below:
The result of Frobenius Norm of near psd is 0.39378468349905377
The result of Frobenius Norm of higham2002 is 0.008036763154638967
When N= 750, the run time of each is shown below:
The run time of near psd is 0.06099867820739746
The run time of higham2002 is 9.089211463928223
When N= 750, the result of Frobenius Norm of each is shown below:
The result of Frobenius Norm of near psd is 0.5933797720225549
The result of Frobenius Norm of higham2002 is 0.00811351389725824
When N= 1000, the run time of each is shown below:
The run time of near psd is 0.15000271797180176
The run time of higham2002 is 15.42787218093872
When N= 1000, the result of Frobenius Norm of each is shown below:
The result of Frobenius Norm of near psd is 0.7929738934149935
The result of Frobenius Norm of higham2002 is 0.008152133586040373
```

The graph of run time：



The graph of Frobenius Norm



The run time graph shows that when N becomes larger, the running time difference between the near_psd and Higham-2002 is significantly becoming larger. Higham's running time grows fast when N is increasing, but the running time of near_psd is stable, remaining at a low time cost.

The Frobenius Norm graph, shows that when N becomes larger, the running time difference between the near_psd and Higham-2002 is also significantly becoming larger. However, Higham's Frobenius Norm is stable, remaining at a low level, while near_psd's Frobenius norm grows fast when N is increasing, which is almost significantly linear with N.

To sum up: near_psd costs little time but produces higher Frobenius norm, while higham produces lower Frobenius Norm, but cost longer time. The difference becomes more significant when N is increasing.

⑥ the pros and cons of each method and when you would use each
<1> near_psd:
Pros:
Cost less time;
Cons:
Cause higher Frobenius Norm
<2> Higham:
Pros:
Cause little Frobenius Norm
Cons:
Cost longer time;

For me, when N is small(N<100), I would like to choose Higham as the run time difference is not significant. However, when N is large, I have to make a decision between the significant time cost and the loss comparing to the original matrix. It depends on a lot of conditions. If I need the fixed matrix is close to the original one, I will still choose Higham. If I have a time limitation, then I may choose near_psd as it runs faster.

Problem 3
Using DailyReturn.csv.

Implement a multivariate normal simulation that allows for simulation directly from a covariance matrix or using PCA with an optional parameter for % variance explained. If you have a library that can do these, you still need to implement it yourself for this homework and prove that it functions as expected.

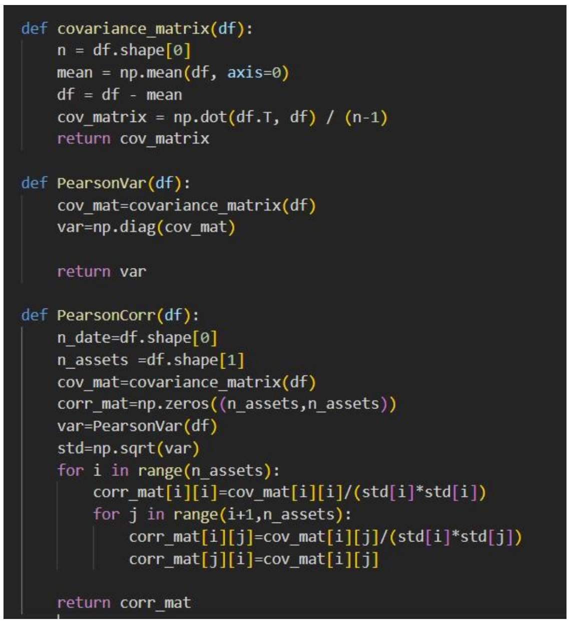
Generate a correlation matrix and variance vector 2 ways:
1. Standard Pearson correlation/variance (you do not need to reimplement the cor() and var() functions).
2. Exponentially weighted λ = 0. 97
Combine these to form 4 different covariance matrices. (Pearson correlation + var()), Pearson correlation + EW variance, etc.)

Simulate 25,000 draws from each covariance matrix using:
1. Direct Simulation
2. PCA with 100% explained.
3. PCA with 75% explained.
4. PCA with 50% explained.

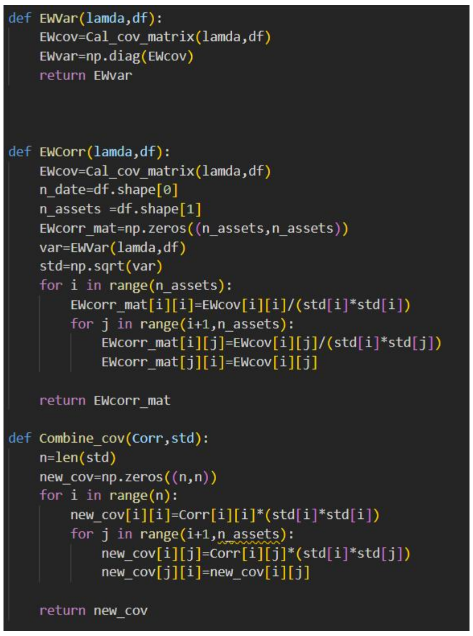
Calculate the covariance of the simulated values. Compare the simulated covariance to it's input matrix using the Frobenius Norm (L2 norm, sum of the square of the difference between the matrices). Compare the run times for each simulation.

What can we say about the trade offs between time to run and accuracy.

Answer:
①Generate a correlation matrix and variance vector 2 ways:

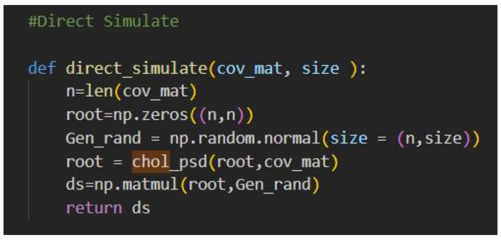


```python
def covariance_matrix(df):
    n = df.shape[0]
    mean = np.mean(df, axis=0)
    df = df - mean
    cov_matrix = np.dot(df.T, df) / (n-1)
    return cov_matrix

def PearsonVar(df):
    cov_mat=covariance_matrix(df)
    var=np.diag(cov_mat)

    return var

def PearsonCorr(df):
    n_date=df.shape[0]
    n_assets =df.shape[1]
    cov_mat=covariance_matrix(df)
    corr_mat=np.zeros((n_assets,n_assets))
    var=PearsonVar(df)
    std=np.sqrt(var)
    for i in range(n_assets):
        corr_mat[i][i]=cov_mat[i][i]/(std[i]*std[i])
        for j in range(i+1,n_assets):
            corr_mat[i][j]=cov_mat[i][j]/(std[i]*std[j])
            corr_mat[j][i]=cov_mat[i][j]

    return corr_mat
```

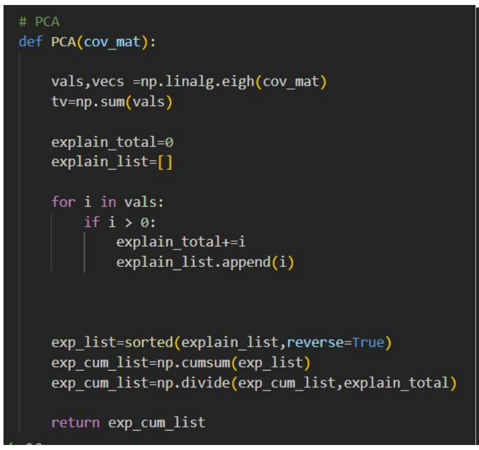


```python
def EWVar(lamda,df):
    EWcov=Cal_cov_matrix(lamda,df)
    EWvar=np.diag(EWcov)
    return EWvar


def EWCorr(lamda,df):
    EWcov=Cal_cov_matrix(lamda,df)
    n_date=df.shape[0]
    n_assets =df.shape[1]
    EWcorr_mat=np.zeros((n_assets,n_assets))
    var=EWVar(lamda,df)
    std=np.sqrt(var)
    for i in range(n_assets):
        EWcorr_mat[i][i]=EWcov[i][i]/(std[i]*std[i])
        for j in range(i+1,n_assets):
            EWcorr_mat[i][j]=EWcov[i][j]/(std[i]*std[j])
            EWcorr_mat[j][i]=EWcov[i][j]

    return EWcorr_mat

def Combine_cov(Corr,std):
    n=len(std)
    new_cov=np.zeros((n,n))
    for i in range(n):
        new_cov[i][i]=Corr[i][i]*(std[i]*std[i])
        for j in range(i+1,n_assets):
            new_cov[i][j]=Corr[i][j]*(std[i]*std[j])
            new_cov[j][i]=new_cov[i][j]

    return new_cov
```

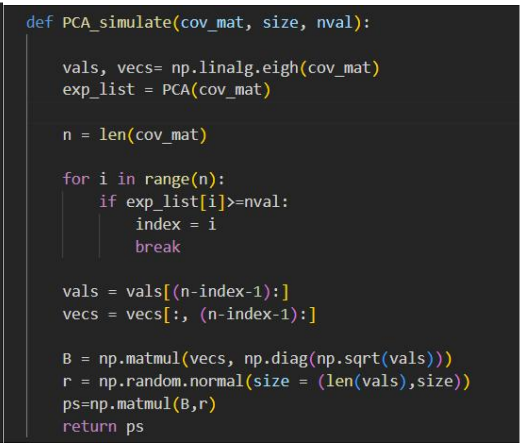
② Direct simulate & PCA simulate



```python
#Direct Simulate

def direct_simulate(cov_mat, size ):
    n=len(cov_mat)
    root=np.zeros((n,n))
    Gen_rand = np.random.normal(size = (n,size))
    root = chol_psd(root,cov_mat)
    ds=np.matmul(root,Gen_rand)
    return ds
```

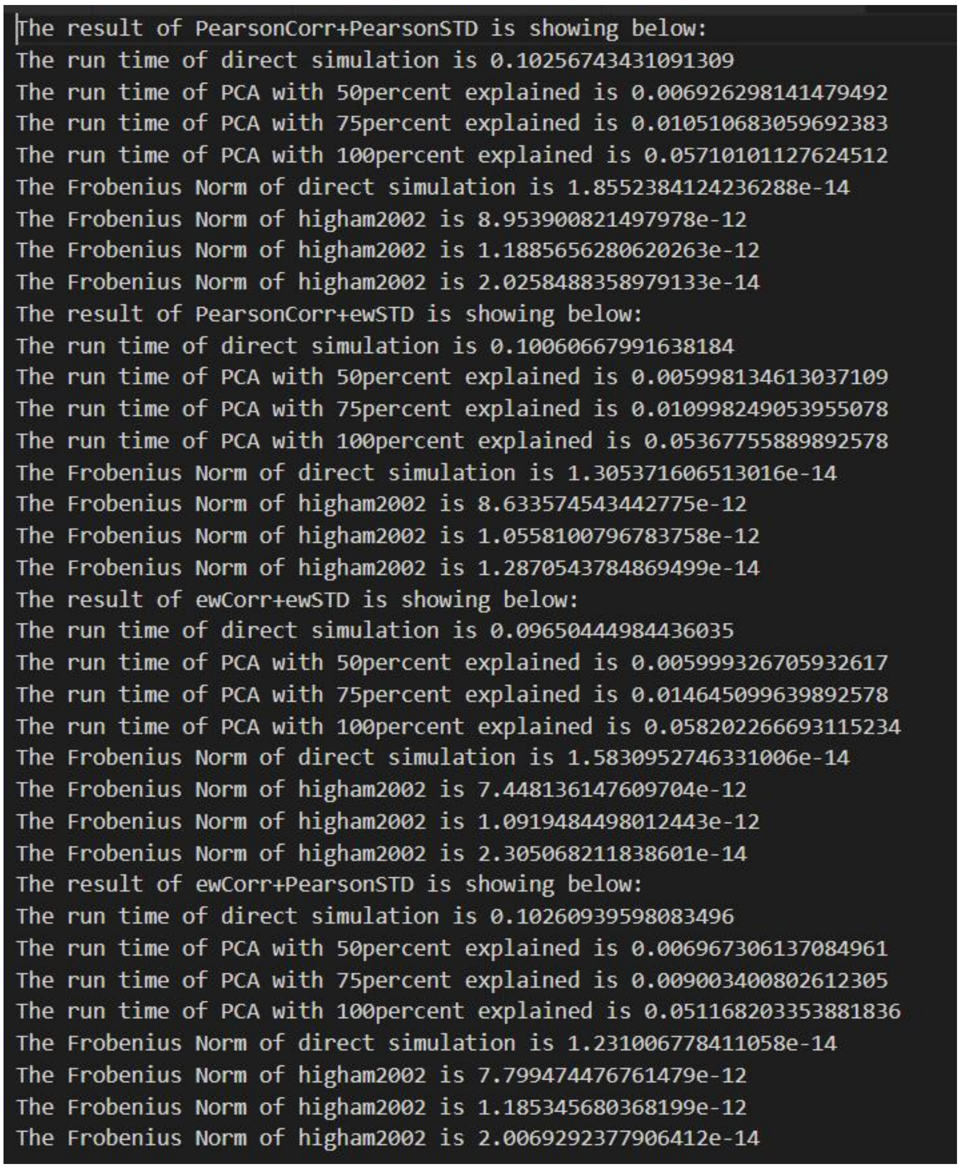


```python
# PCA
def PCA(cov_mat):

    vals,vecs =np.linalg.eigh(cov_mat)
    tv=np.sum(vals)

    explain_total=0
    explain_list=[]

    for i in vals:
        if i > 0:
            explain_total+=i
            explain_list.append(i)



    exp_list=sorted(explain_list,reverse=True)
    exp_cum_list=np.cumsum(exp_list)
    exp_cum_list=np.divide(exp_cum_list,explain_total)

    return exp_cum_list
```



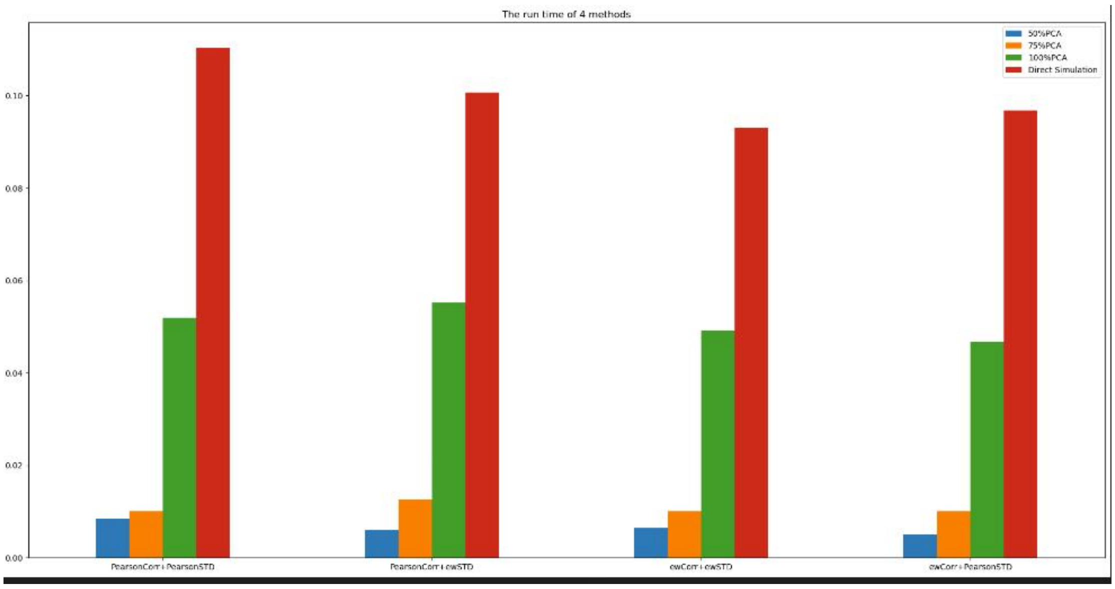
```python
def PCA_simulate(cov_mat, size, nval):

    vals, vecs= np.linalg.eigh(cov_mat)
    exp_list = PCA(cov_mat)

    n = len(cov_mat)

    for i in range(n):
        if exp_list[i]>=nval:
            index = i
            break

    vals = vals[(n-index-1):]
    vecs = vecs[:, (n-index-1):]

    B = np.matmul(vecs, np.diag(np.sqrt(vals)))
    r = np.random.normal(size = (len(vals),size))
    ps=np.matmul(B,r)
    return ps
```

③ Compare the simulated covariance to it’s input matrix using the Frobenius Norm. Compare the run times for each simulation.

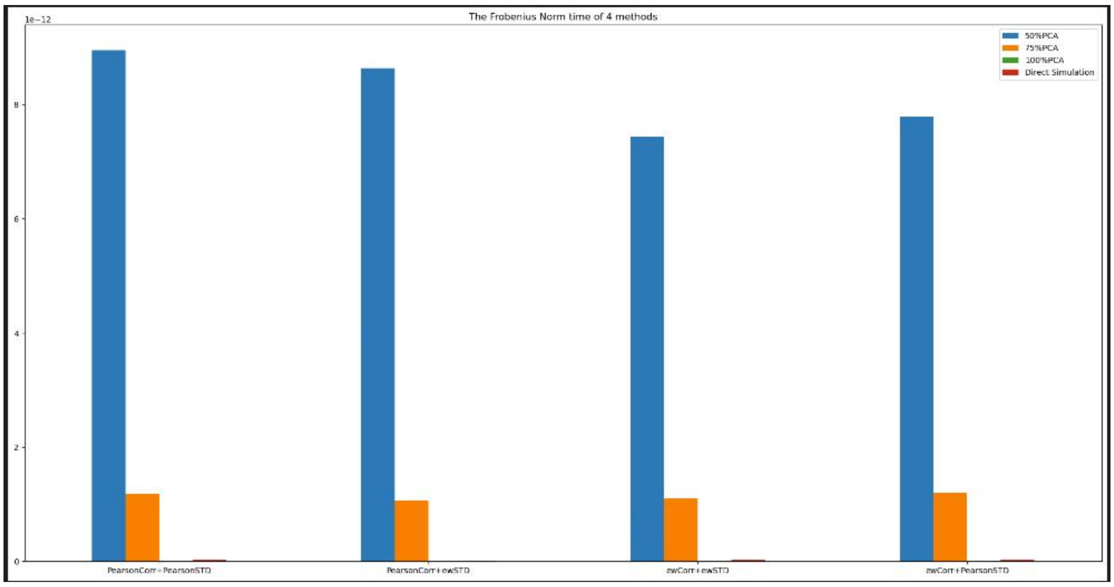


```
The result of PearsonCorr+PearsonSTD is showing below:
The run time of direct simulation is 0.10256743431091309
The run time of PCA with 50percent explained is 0.006926298141479492
The run time of PCA with 75percent explained is 0.010510683059692383
The run time of PCA with 100percent explained is 0.05710101127624512
The Frobenius Norm of direct simulation is 1.8552384124236288e-14
The Frobenius Norm of higham2002 is 8.953900821497978e-12
The Frobenius Norm of higham2002 is 1.1885656280620263e-12
The Frobenius Norm of higham2002 is 2.0258488358979133e-14
The result of PearsonCorr+ewSTD is showing below:
The run time of direct simulation is 0.10060667991638184
The run time of PCA with 50percent explained is 0.005998134613037109
The run time of PCA with 75percent explained is 0.010998249053955078
The run time of PCA with 100percent explained is 0.05367755889892578
The Frobenius Norm of direct simulation is 1.305371606513016e-14
The Frobenius Norm of higham2002 is 8.633574543442775e-12
The Frobenius Norm of higham2002 is 1.0558100796783758e-12
The Frobenius Norm of higham2002 is 1.2870543784869499e-14
The result of ewCorr+ewSTD is showing below:
The run time of direct simulation is 0.09650444984436035
The run time of PCA with 50percent explained is 0.005999326705932617
The run time of PCA with 75percent explained is 0.014645099639892578
The run time of PCA with 100percent explained is 0.058202266693115234
The Frobenius Norm of direct simulation is 1.5830952746331006e-14
The Frobenius Norm of higham2002 is 7.448136147609704e-12
The Frobenius Norm of higham2002 is 1.0919484498012443e-12
The Frobenius Norm of higham2002 is 2.305068211838601e-14
The result of ewCorr+PearsonSTD is showing below:
The run time of direct simulation is 0.10260939598083496
The run time of PCA with 50percent explained is 0.006967306137084961
The run time of PCA with 75percent explained is 0.009003400802612305
The run time of PCA with 100percent explained is 0.051168203353881836
The Frobenius Norm of direct simulation is 1.231006778411058e-14
The Frobenius Norm of higham2002 is 7.799474476761479e-12
The Frobenius Norm of higham2002 is 1.185345680368199e-12
The Frobenius Norm of higham2002 is 2.0069292377906412e-14
```

The graph of the run time of different methods



The graph of the Frobenius Norm of different methods



The run time graph shows that 50% **PCA simulation costs the shortest time**, following by 75% PCA simulation. The direct simulation cost the longest time. Among all PCA methods, 100% explained PCA cost the longest time.

The Frobenius Norm graph, shows that **50% PCA simulation produces the highest Frobenius Norm**, following by 75% PCA simulation. The direct simulation produces the lowest Frobenius Norm. Among all PCA methods, 100% explained PCA produces the lowest Frobenius Norm..

To sum up: direct simulation costs highest time but produces lowest Frobenius norm. Among PCA methods, with higher percent explained, the PCA will cost highest time but produces lowest Frobenius norm, which means higher accuracy.
Speed: the left is better
50%PCA > 75% PCA > 100% PCA > Direct Simulation
Accuracy: the left is better
Direct Simulation >   100% PCA > 75% PCA > 50%PCA

④What can we say about the trade offs between time to run and accuracy.
According to the results above, we could describe that the relationship between speed and accuracy is negative. It means that the method which is faster will be less accurate. However, such relationship is not very linear, as we can see a huge difference in Frobenius norm between 50%PCA and 75% PCA. The run time between these two methods is similar, while the difference in Frobenius norm is huge. So if we need to find a balance in speed and accuracy, 75% PCA is a good choice. In other cases, you could choose 50% if you only need run fast, or you could choose 100%PCA or direct simulation if you only care the accuracy.