Week03 Project Report

Problem 1

Problem

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 λchosen.

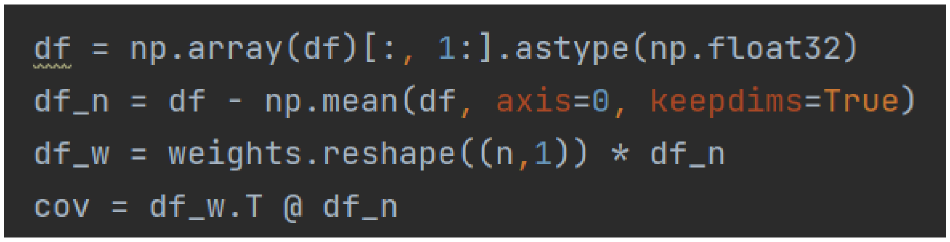
What does this tell us about values of λ and the effect it has on the covariance matrix?

Answer

1.Calculate exponentially weighted covariance matrix

To calculate the exponentially weighted covariance matrix for the stock returns, I wrote a function `exp_w_cov(df, lambd)` by using parameters of daily return data and a λ. I first calculated the array of `weights`, one weight for the data of one day. Then I used the weights to calculate the weighted covariance matrix.

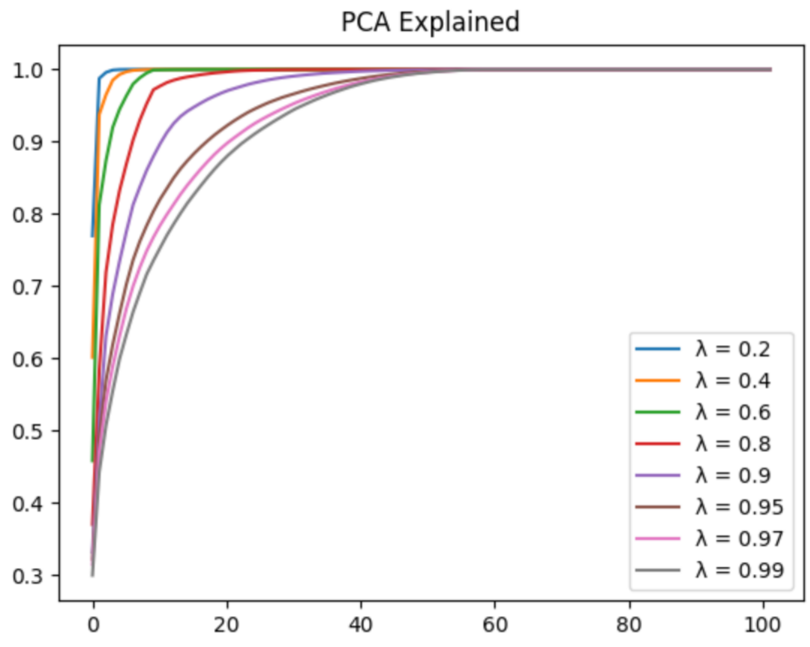
I tried to use `for` loops to do this first, by iterating columns twice and row once, but it took me a few minutes to get the results for the total eight λs. Then I tried to optimize my algorithm by calculating only half of the covariance matrix, because it is symmetric. But this still took me a long time. Finally, I used `numpy.array` to do matrix calculations, instead of loops, which took me a few seconds to get the answer. This was a great optimization of the speed. Here is the code which used matrix calculation.



```
df = np.array(df)[:, 1:].astype(np.float32)
df_n = df - np.mean(df, axis=0, keepdims=True)
df_w = weights.reshape((n,1)) * df_n
cov = df_w.T @ df_n
```

2.Use PCA and plot the cumulative variance explained

By varying λ, setting it to 0.2, 0.4, 0.6, 0.8, 0.9, 0.95, 0.97, 0.99, I used PCA and plot the cumulative variance explained by each eigenvalue for each λ chosen. Here is the plot.



From this plot, we can see that when λ gets larger, the line will be smoother. Because larger λ means that we give less weight to the nearest return data, which will make the covariance matrix less concentrated, and more data will have explanation power. And when we use PCA decomposition, we will find that the data information will be provided by data in a longer time, which makes the increase of the line smaller.

In addition, when λ gets larger, the beginning point of the line gets smaller, and increases smoother, this means that the first few eigenvalues explain less of the total variance. And the weights for the dates before, though getting smaller, are more balanced. That is to say, more data are considered.

By the way, when calculating the eigenvalues of the covariance matrix, I found that if λ is too small, for example it is 0.2, then most of the eigenvalues will be negative, and will be set to 0 when using PCA, which do not have any explanation of the total variance.

Problem 2

Problem

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.

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

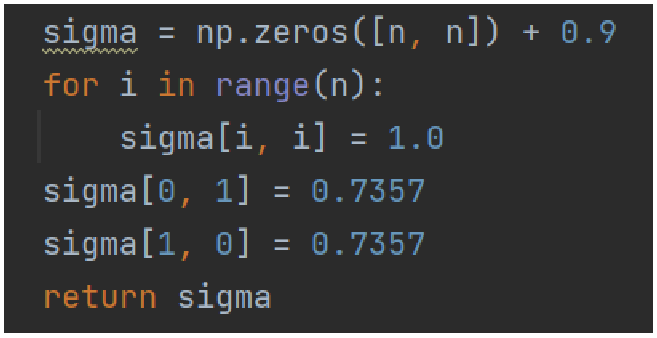
Compare the results of both using the Frobenius Norm. Compare the run time between the two. How does the run time of each function compare as N increases?

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

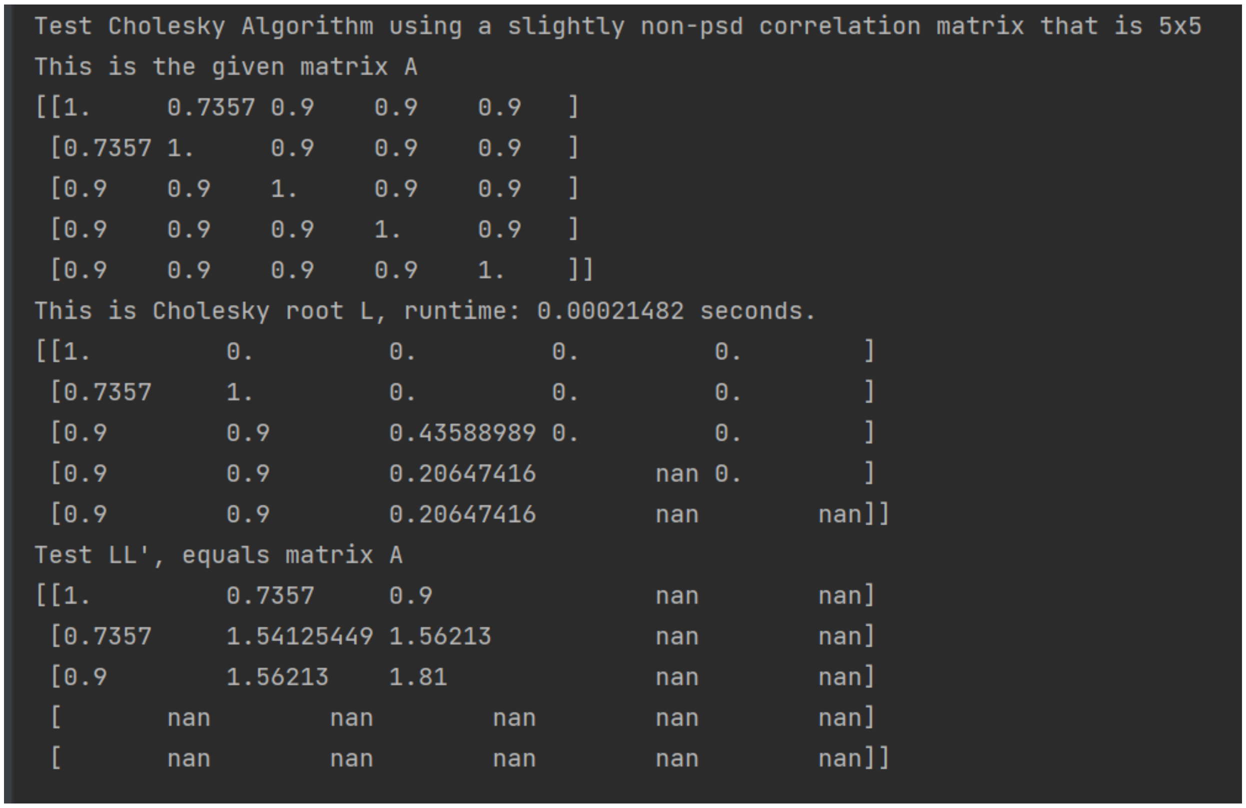
1.Implement `chol_psd`, `near_psd` and `higham_psd`

Implement `chol_psd` and test Cholesky Algorithm using a slightly non-psd correlation matrix that is 5x5, n = 5.



```
sigma = np.zeros([n, n]) + 0.9
for i in range(n):
    sigma[i, i] = 1.0
sigma[0, 1] = 0.7357
sigma[1, 0] = 0.7357
return sigma
```

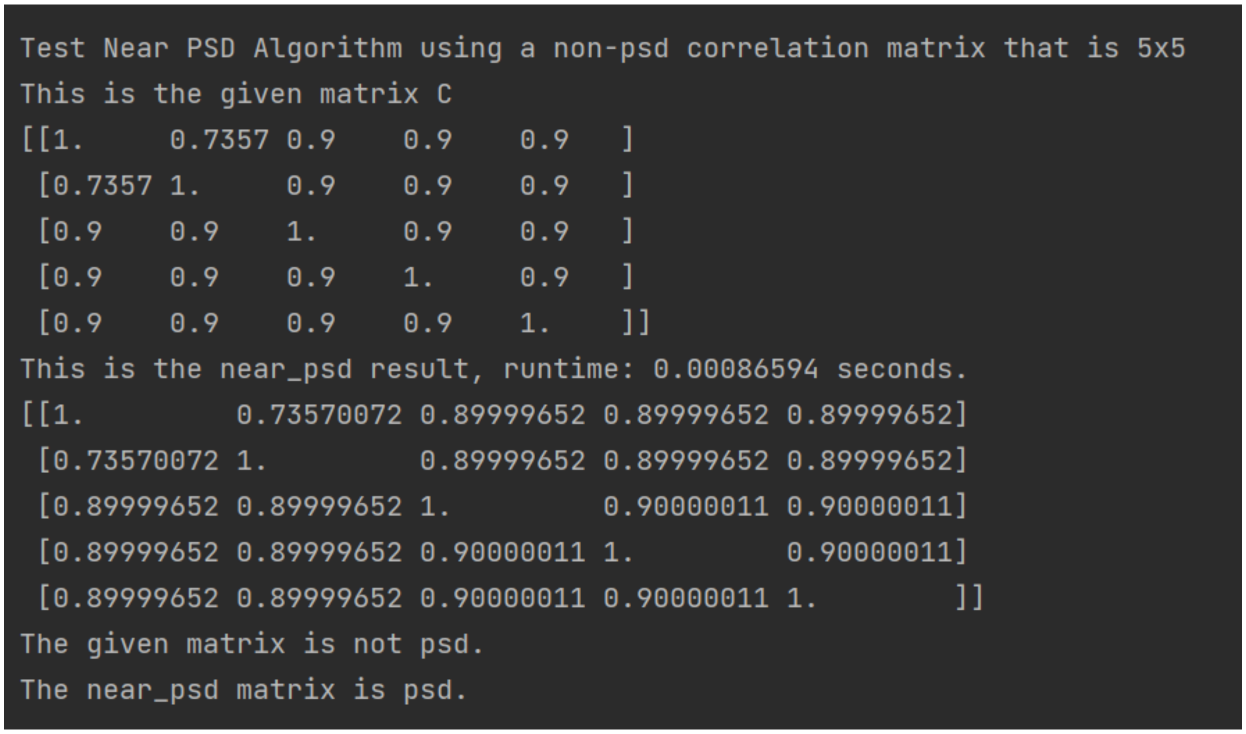
The result is as follow.



```
Test Cholesky Algorithm using a slightly non-psd correlation matrix that is 5x5
This is the given matrix A
[[1.     0.7357 0.9    0.9    0.9    ]
 [0.7357 1.     0.9    0.9    0.9    ]
 [0.9    0.9    1.     0.9    0.9    ]
 [0.9    0.9    0.9    1.     0.9    ]
 [0.9    0.9    0.9    0.9    1.     ]]
This is Cholesky root L, runtime: 0.00021482 seconds.
[[1.         0.         0.         0.         0.        ]
 [0.7357     1.         0.         0.         0.        ]
 [0.9        0.9        0.43588989 0.         0.        ]
 [0.9        0.9        0.20647416        nan 0.        ]
 [0.9        0.9        0.20647416        nan        nan]]
Test LL', equals matrix A
[[1.         0.7357     0.9               nan        nan]
 [0.7357     1.54125449 1.56213           nan        nan]
 [0.9        1.56213    1.81              nan        nan]
 [       nan        nan        nan        nan        nan]
 [       nan        nan        nan        nan        nan]]
```

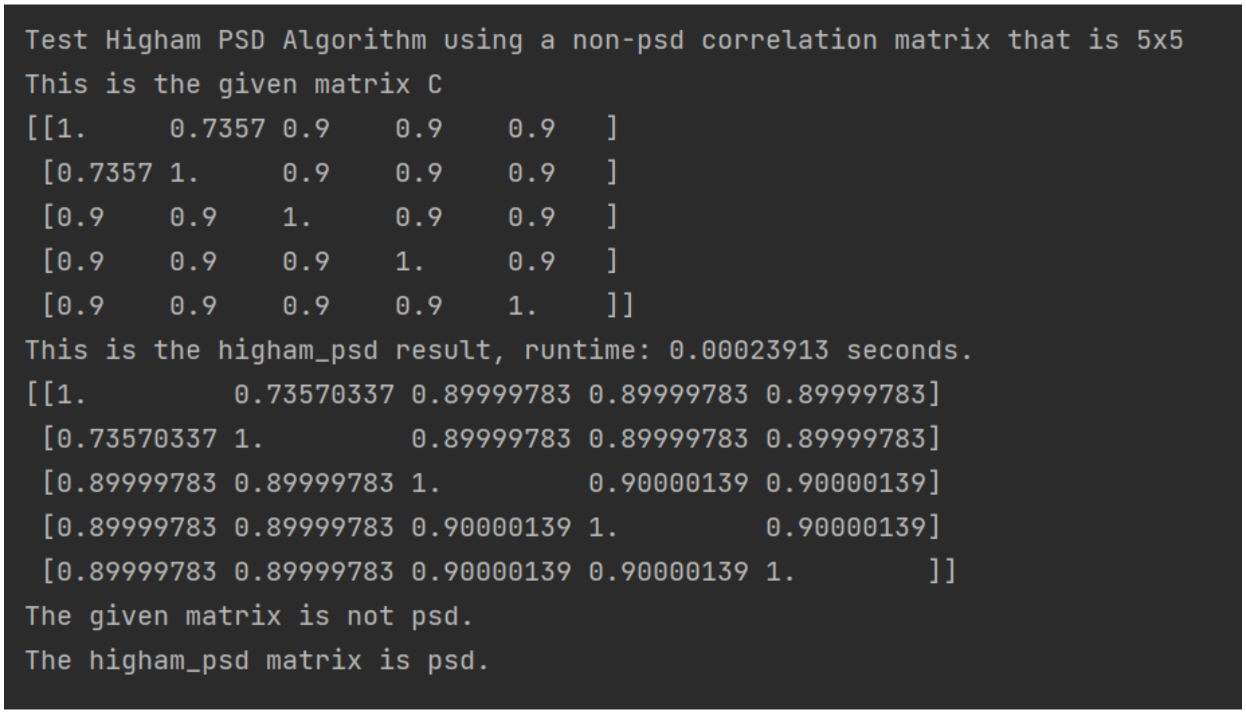
As we know, the Cholesky Algorithm requires the given matrix be symmetric and positive definite to get the root matrix, and will break when the given matrix is not positive semi definite. When I set the given matrix A slightly non-psd, it will not provide an ideal result, which is shown as the screenshot above. Though it is very fast, with only 0.0002 seconds for a 5x5 matrix, taking almost a quarter time of Near PSD Algorithm, the requirement of PSD is strict. And we often use the condition that the smallest eigenvalue is larger than `-1e-8` to ensure the given matrix to be PSD.

Implement `near_psd` and test Near PSD Algorithm using the same non-psd correlation matrix that is 5x5, n = 5. The result is as follow. It is confirmed that the result is psd by the condition that the smallest eigenvalue of the matrix is larger that `-1e-8`.



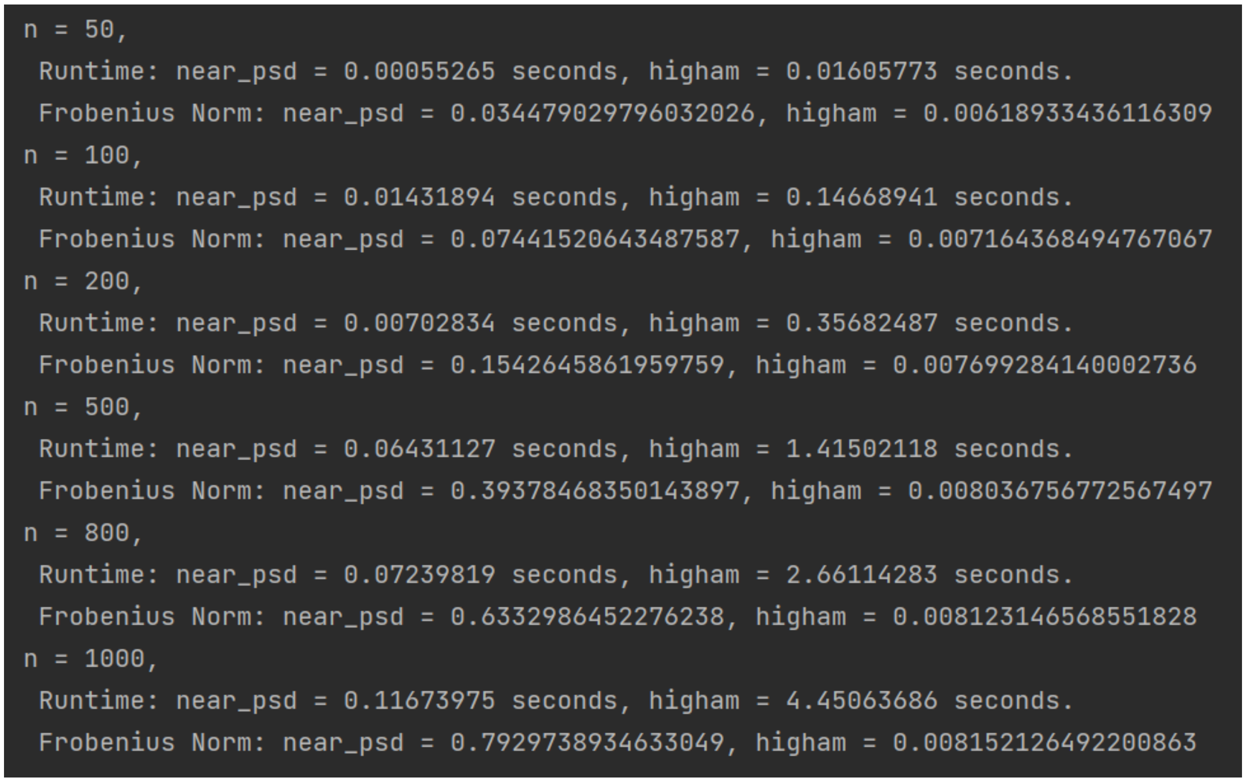
```
Test Near PSD Algorithm using a non-psd correlation matrix that is 5x5
This is the given matrix C
[[1.     0.7357 0.9    0.9    0.9   ]
 [0.7357 1.     0.9    0.9    0.9   ]
 [0.9    0.9    1.     0.9    0.9   ]
 [0.9    0.9    0.9    1.     0.9   ]
 [0.9    0.9    0.9    0.9    1.    ]]
This is the near_psd result, runtime: 0.00086594 seconds.
[[1.         0.73570072 0.89999652 0.89999652 0.89999652]
 [0.73570072 1.         0.89999652 0.89999652 0.89999652]
 [0.89999652 0.89999652 1.         0.90000011 0.90000011]
 [0.89999652 0.89999652 0.90000011 1.         0.90000011]
 [0.89999652 0.89999652 0.90000011 0.90000011 1.        ]]
The given matrix is not psd.
The near_psd matrix is psd.
```

Implement `higham_psd` and test Higham PSD Algorithm using the same non-psd correlation matrix that is 5x5, n = 5. The result is as follow. It is confirmed that the result is psd by the condition that the smallest eigenvalue of the matrix is larger that `-1e-6`, because it cannot satisfy the tolerance value of `-1e-8`, which means that the psd found by the Higham Algorithm is not as strict as that found by the near_psd Algorithm. It sometimes cannot satisfy the requirement of Cholesky Algorithm, which often using the condition that the smallest eigenvalue is larger than `-1e-8`.

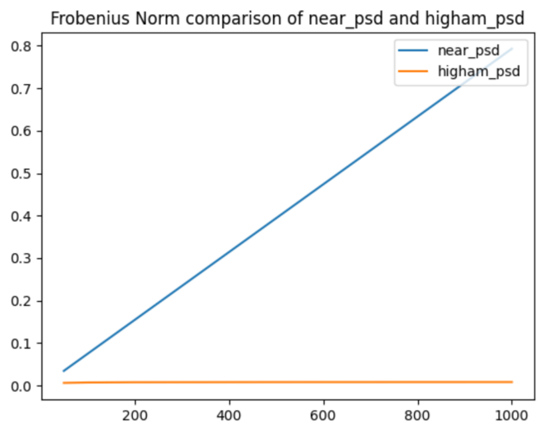


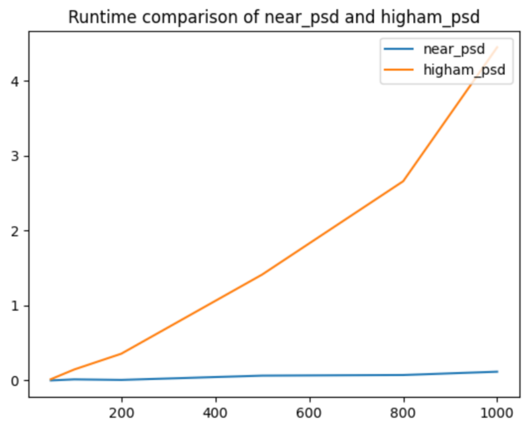
```
Test Higham PSD Algorithm using a non-psd correlation matrix that is 5x5
This is the given matrix C
[[1.     0.7357 0.9    0.9    0.9   ]
 [0.7357 1.     0.9    0.9    0.9   ]
 [0.9    0.9    1.     0.9    0.9   ]
 [0.9    0.9    0.9    1.     0.9   ]
 [0.9    0.9    0.9    0.9    1.    ]]
This is the higham_psd result, runtime: 0.00023913 seconds.
[[1.         0.73570337 0.89999783 0.89999783 0.89999783]
 [0.73570337 1.         0.89999783 0.89999783 0.89999783]
 [0.89999783 0.89999783 1.         0.90000139 0.90000139]
 [0.89999783 0.89999783 0.90000139 1.         0.90000139]
 [0.89999783 0.89999783 0.90000139 0.90000139 1.        ]]
The given matrix is not psd.
The higham_psd matrix is psd.
```

2.Compare the results of both using the Frobenius Norm



```
n = 50,
 Runtime: near_psd = 0.00055265 seconds, higham = 0.01605773 seconds.
 Frobenius Norm: near_psd = 0.034479029796032026, higham = 0.00618933436116309
n = 100,
 Runtime: near_psd = 0.01431894 seconds, higham = 0.14668941 seconds.
 Frobenius Norm: near_psd = 0.07441520643487587, higham = 0.007164368494767067
n = 200,
 Runtime: near_psd = 0.00702834 seconds, higham = 0.35682487 seconds.
 Frobenius Norm: near_psd = 0.1542645861959759, higham = 0.007699284140002736
n = 500,
 Runtime: near_psd = 0.06431127 seconds, higham = 1.41502118 seconds.
 Frobenius Norm: near_psd = 0.39378468350143897, higham = 0.008036756772567497
n = 800,
 Runtime: near_psd = 0.07239819 seconds, higham = 2.66114283 seconds.
 Frobenius Norm: near_psd = 0.6332986452276238, higham = 0.008123146568551828
n = 1000,
 Runtime: near_psd = 0.11673975 seconds, higham = 4.45063686 seconds.
 Frobenius Norm: near_psd = 0.7929738934633049, higham = 0.008152126492200863
```





Frobenius Norm can be thought as the distance between the psd result and the given non-psd matrix. Smaller Norm means nearer distance, less difference. As n increases, the Frobenius Norm of near_psd will increase linearly, which means a fast increasing error; while the Frobenius Norm of Higham PSD always remains near 0.0, with high accuracy. However, the runtime of Higham PSD will increase more than linearly, at about few times of the runtime of near_psd.

When considering PSD, we can choose near_psd to get faster result if the accuracy is not strictly required. However, if we want to get the nearest PSD matrix of the given matrix, Higham is definitely the better choice, though costing longer time. By the way, we need to pay attention to the PSD generated by Higham, because it sometimes cannot provide the result which can be used by the Cholesky Algorithm, which often using the condition that the smallest eigenvalue is larger than `-1e-8`.

Problem 3

Problem

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 $\lambda = 0.97$

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.

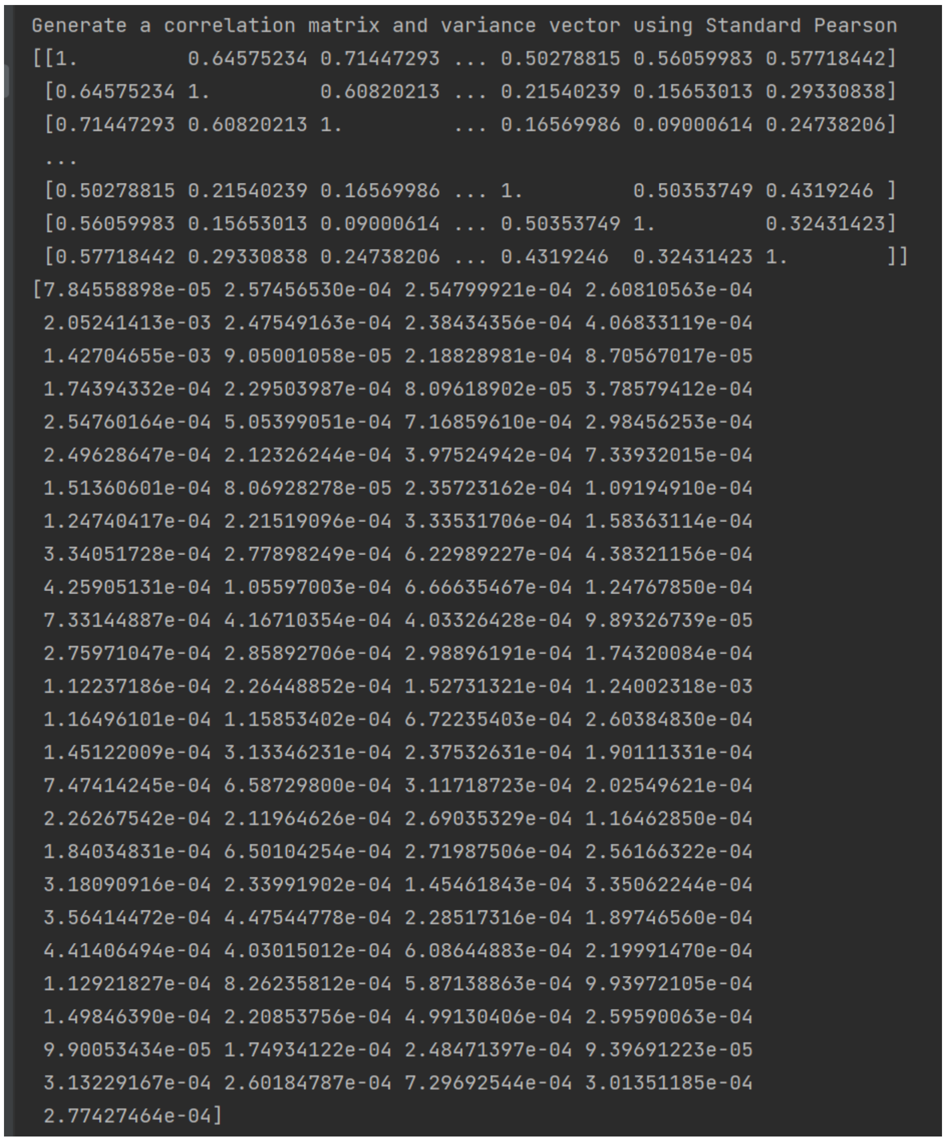
Calculate the covariance of the simulated values. Compare the simulated covariance to it's input matrix using the Frobenius Norm (L2 norm, sum of the square of the difference between the matrices). Compare the run times for each simulation.

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

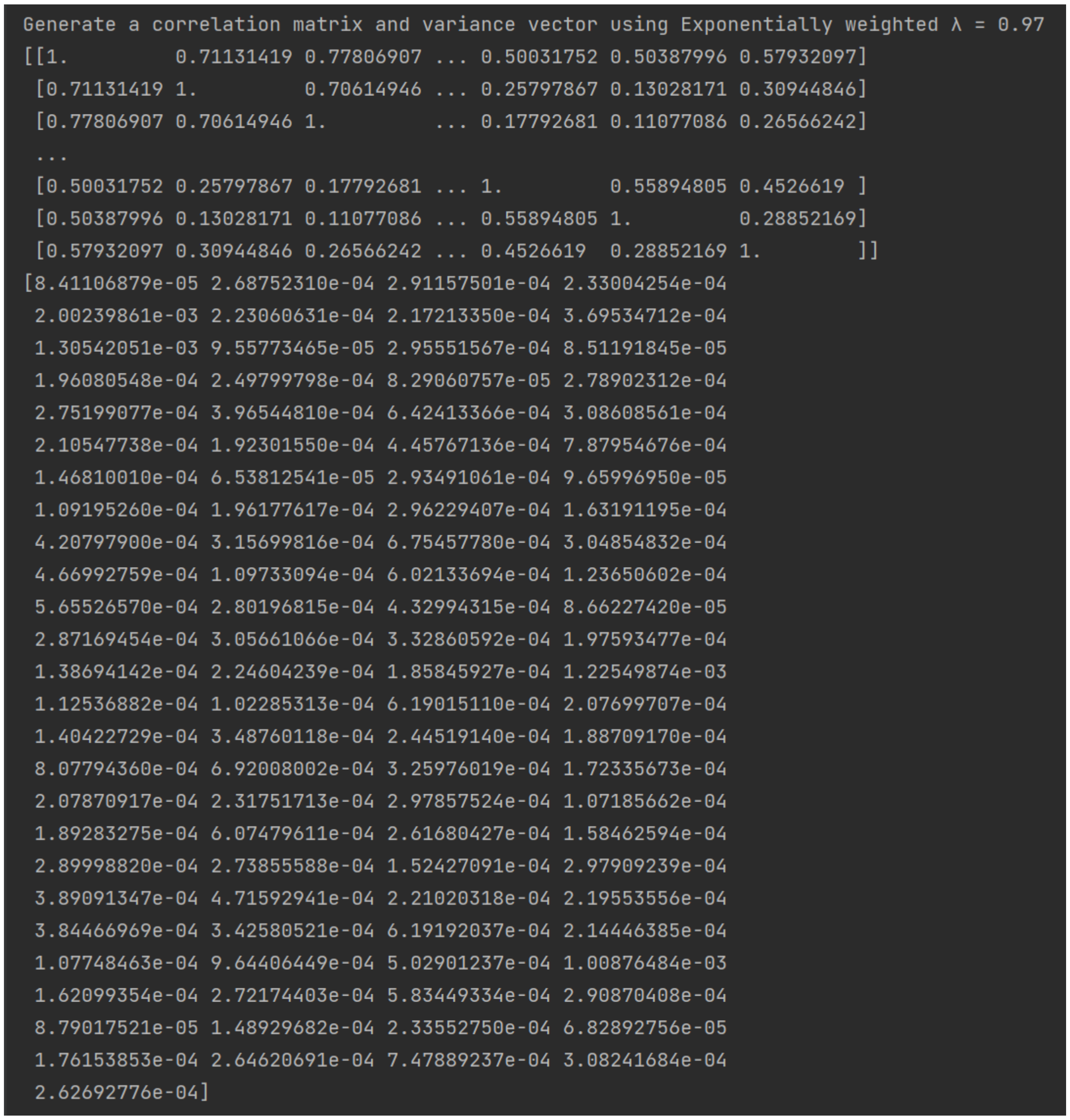
Answer

1.Generate a correlation matrix and variance vector 2 ways

For the Pearson correlation, there is function in `Pandas`, for the exponentially weighted correlation, I used the function in problem 1, with a matrix calculation of correlation.



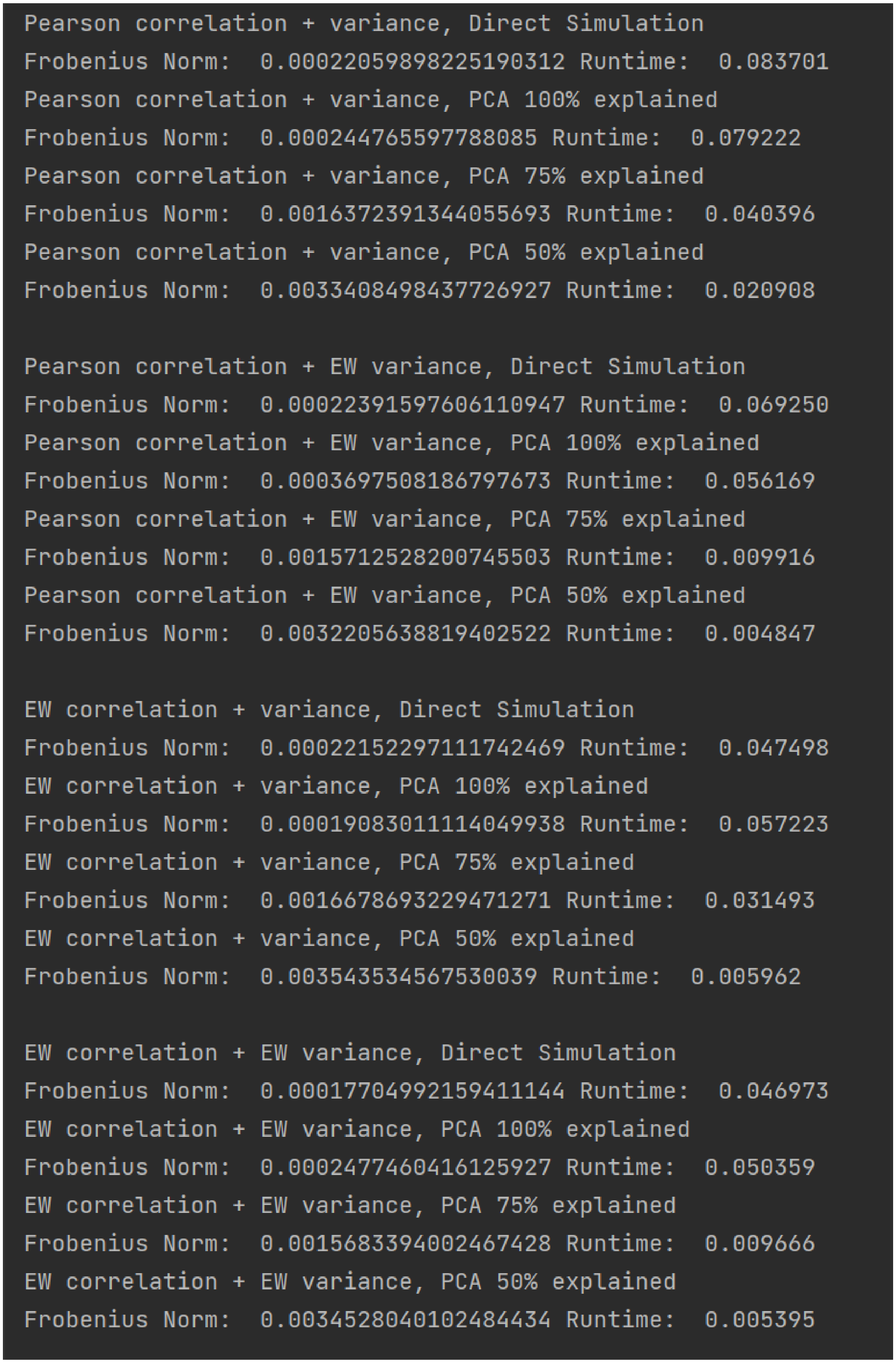
```
Generate a correlation matrix and variance vector using Standard Pearson
[[1.         0.64575234 0.71447293 ... 0.50278815 0.56059983 0.57718442]
 [0.64575234 1.         0.60820213 ... 0.21540239 0.15653013 0.29330838]
 [0.71447293 0.60820213 1.         ... 0.16569986 0.09000614 0.24738206]
 ...
 [0.50278815 0.21540239 0.16569986 ... 1.         0.50353749 0.4319246 ]
 [0.56059983 0.15653013 0.09000614 ... 0.50353749 1.         0.32431423]
 [0.57718442 0.29330838 0.24738206 ... 0.4319246  0.32431423 1.        ]]
[7.84558898e-05 2.57456530e-04 2.54799921e-04 2.60810563e-04
 2.05241413e-03 2.47549163e-04 2.38434356e-04 4.06833119e-04
 1.42704655e-03 9.05001058e-05 2.18828981e-04 8.70567017e-05
 1.74394332e-04 2.29503987e-04 8.09618902e-05 3.78579412e-04
 2.54760164e-04 5.05399051e-04 7.16859610e-04 2.98456253e-04
 2.49628647e-04 2.12326244e-04 3.97524942e-04 7.33932015e-04
 1.51360601e-04 8.06928278e-05 2.35723162e-04 1.09194910e-04
 1.24740417e-04 2.21519096e-04 3.33531706e-04 1.58363114e-04
 3.34051728e-04 2.77898249e-04 6.22989227e-04 4.38321156e-04
 4.25905131e-04 1.05597003e-04 6.66635467e-04 1.24767850e-04
 7.33144887e-04 4.16710354e-04 4.03326428e-04 9.89326739e-05
 2.75971047e-04 2.85892706e-04 2.98896191e-04 1.74320084e-04
 1.12237186e-04 2.26448852e-04 1.52731321e-04 1.24002318e-03
 1.16496101e-04 1.15853402e-04 6.72235403e-04 2.60384830e-04
 1.45122009e-04 3.13346231e-04 2.37532631e-04 1.90111331e-04
 7.47414245e-04 6.58729800e-04 3.11718723e-04 2.02549621e-04
 2.26267542e-04 2.11964626e-04 2.69035329e-04 1.16462850e-04
 1.84034831e-04 6.50104254e-04 2.71987506e-04 2.56166322e-04
 3.18090916e-04 2.33991902e-04 1.45461843e-04 3.35062244e-04
 3.56414472e-04 4.47544778e-04 2.28517316e-04 1.89746560e-04
 4.41406494e-04 4.03015012e-04 6.08644883e-04 2.19991470e-04
 1.12921827e-04 8.26235812e-04 5.87138863e-04 9.93972105e-04
 1.49846390e-04 2.20853756e-04 4.99130406e-04 2.59590063e-04
 9.90053434e-05 1.74934122e-04 2.48471397e-04 9.39691223e-05
 3.13229167e-04 2.60184787e-04 7.29692544e-04 3.01351185e-04
 2.77427464e-04]
```



```
Generate a correlation matrix and variance vector using Exponentially weighted λ = 0.97
[[1.         0.71131419 0.77806907 ... 0.50031752 0.50387996 0.57932097]
 [0.71131419 1.         0.70614946 ... 0.25797867 0.13028171 0.30944846]
 [0.77806907 0.70614946 1.         ... 0.17792681 0.11077086 0.26566242]
 ...
 [0.50031752 0.25797867 0.17792681 ... 1.         0.55894805 0.4526619 ]
 [0.50387996 0.13028171 0.11077086 ... 0.55894805 1.         0.28852169]
 [0.57932097 0.30944846 0.26566242 ... 0.4526619  0.28852169 1.        ]]
[8.41106879e-05 2.68752310e-04 2.91157501e-04 2.33004254e-04
 2.00239861e-03 2.23060631e-04 2.17213350e-04 3.69534712e-04
 1.30542051e-03 9.55773465e-05 2.95551567e-04 8.51191845e-05
 1.96080548e-04 2.49799798e-04 8.29060757e-05 2.78902312e-04
 2.75199077e-04 3.96544810e-04 6.42413366e-04 3.08608561e-04
 2.10547738e-04 1.92301550e-04 4.45767136e-04 7.87954676e-04
 1.46810010e-04 6.53812541e-05 2.93491061e-04 9.65996950e-05
 1.09195260e-04 1.96177617e-04 2.96229407e-04 1.63191195e-04
 4.20797900e-04 3.15699816e-04 6.75457780e-04 3.04854832e-04
 4.66992759e-04 1.09733094e-04 6.02133694e-04 1.23650602e-04
 5.65526570e-04 2.80196815e-04 4.32994315e-04 8.66227420e-05
 2.87169454e-04 3.05661066e-04 3.32860592e-04 1.97593477e-04
 1.38694142e-04 2.24604239e-04 1.85845927e-04 1.22549874e-03
 1.12536882e-04 1.02285313e-04 6.19015110e-04 2.07699707e-04
 1.40422729e-04 3.48760118e-04 2.44519140e-04 1.88709170e-04
 8.07794360e-04 6.92008002e-04 3.25976019e-04 1.72335673e-04
 2.07870917e-04 2.31751713e-04 2.97857524e-04 1.07185662e-04
 1.89283275e-04 6.07479611e-04 2.61680427e-04 1.58462594e-04
 2.89998820e-04 2.73855588e-04 1.52427091e-04 2.97909239e-04
 3.89091347e-04 4.71592941e-04 2.21020318e-04 2.19553556e-04
 3.84466969e-04 3.42580521e-04 6.19192037e-04 2.14446385e-04
 1.07748463e-04 9.64406449e-04 5.02901237e-04 1.00876484e-03
 1.62099354e-04 2.72174403e-04 5.83449334e-04 2.90870408e-04
 8.79017521e-05 1.48929682e-04 2.33552750e-04 6.82892756e-05
 1.76153853e-04 2.64620691e-04 7.47889237e-04 3.08241684e-04
 2.62692776e-04]
```

2.Simulate 25,000 draws from each covariance matrix

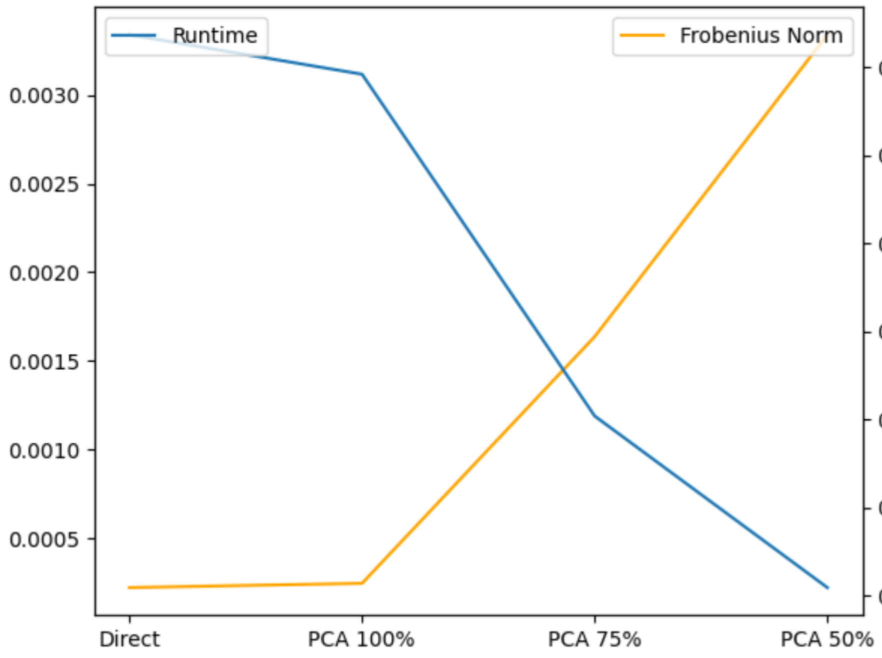
By generating four covariance matrix, I calculated the Frobenius Norm between the simulated matrix and the original matrix and compared the runtime of each method.



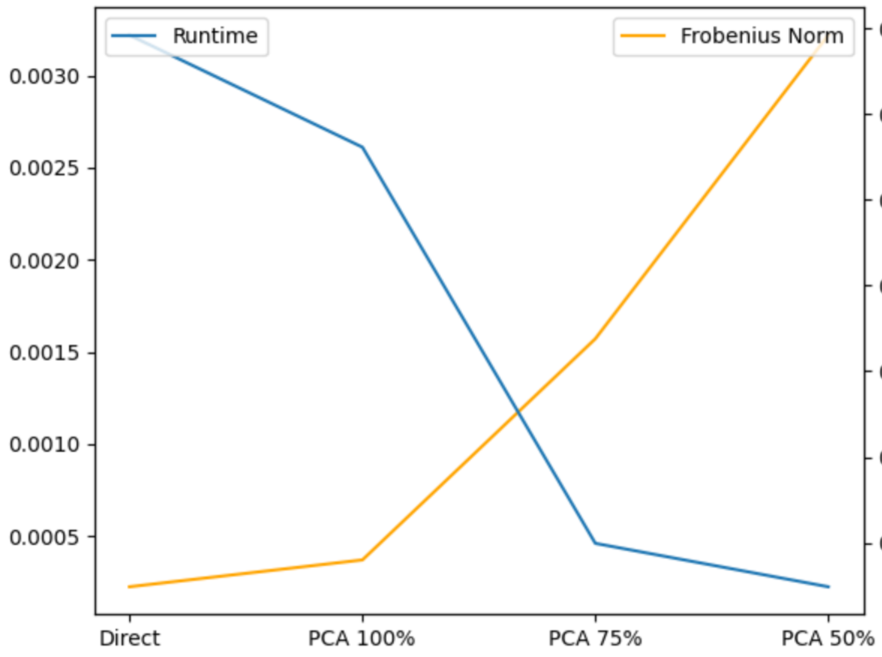
```
Pearson correlation + variance, Direct Simulation
Frobenius Norm:  0.00022059898225190312 Runtime:  0.083701
Pearson correlation + variance, PCA 100% explained
Frobenius Norm:  0.00024765597788085 Runtime:  0.079222
Pearson correlation + variance, PCA 75% explained
Frobenius Norm:  0.0016372391344055693 Runtime:  0.040396
Pearson correlation + variance, PCA 50% explained
Frobenius Norm:  0.0033408498437726927 Runtime:  0.020908

Pearson correlation + EW variance, Direct Simulation
Frobenius Norm:  0.00022391597606110947 Runtime:  0.069250
Pearson correlation + EW variance, PCA 100% explained
Frobenius Norm:  0.0003697508186797673 Runtime:  0.056169
Pearson correlation + EW variance, PCA 75% explained
Frobenius Norm:  0.0015712528200745503 Runtime:  0.009916
Pearson correlation + EW variance, PCA 50% explained
Frobenius Norm:  0.0032205638819402522 Runtime:  0.004847

EW correlation + variance, Direct Simulation
Frobenius Norm:  0.00022152297111742469 Runtime:  0.047498
EW correlation + variance, PCA 100% explained
Frobenius Norm:  0.00019083011114049938 Runtime:  0.057223
EW correlation + variance, PCA 75% explained
Frobenius Norm:  0.0016678693229471271 Runtime:  0.031493
EW correlation + variance, PCA 50% explained
Frobenius Norm:  0.003543534567530039 Runtime:  0.005962

EW correlation + EW variance, Direct Simulation
Frobenius Norm:  0.00017704992159411144 Runtime:  0.046973
EW correlation + EW variance, PCA 100% explained
Frobenius Norm:  0.0002477460416125927 Runtime:  0.050359
EW correlation + EW variance, PCA 75% explained
Frobenius Norm:  0.0015683394002467428 Runtime:  0.009666
EW correlation + EW variance, PCA 50% explained
Frobenius Norm:  0.0034528040102484434 Runtime:  0.005395
```

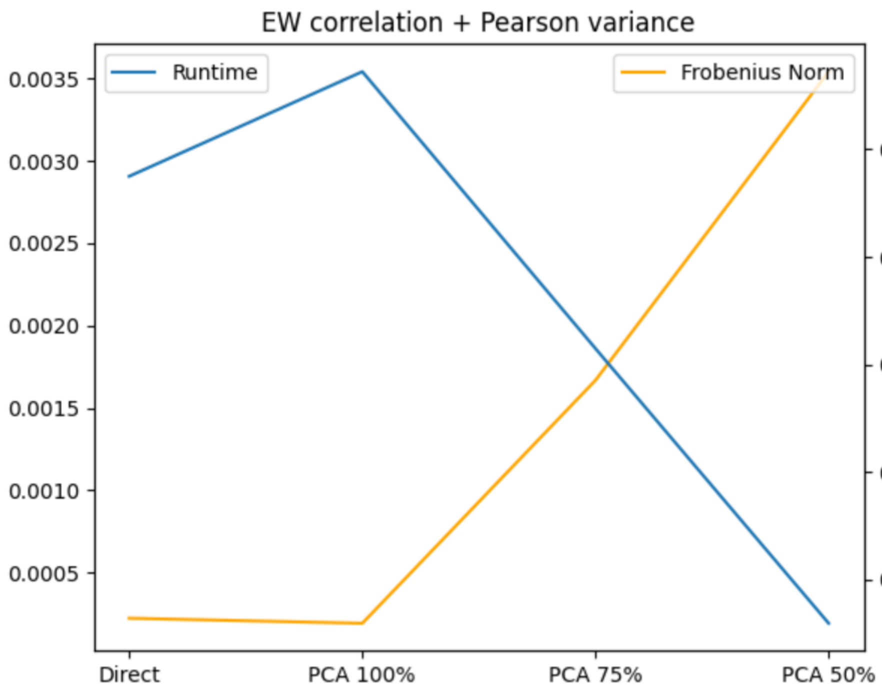
I drew plots to show the above results.

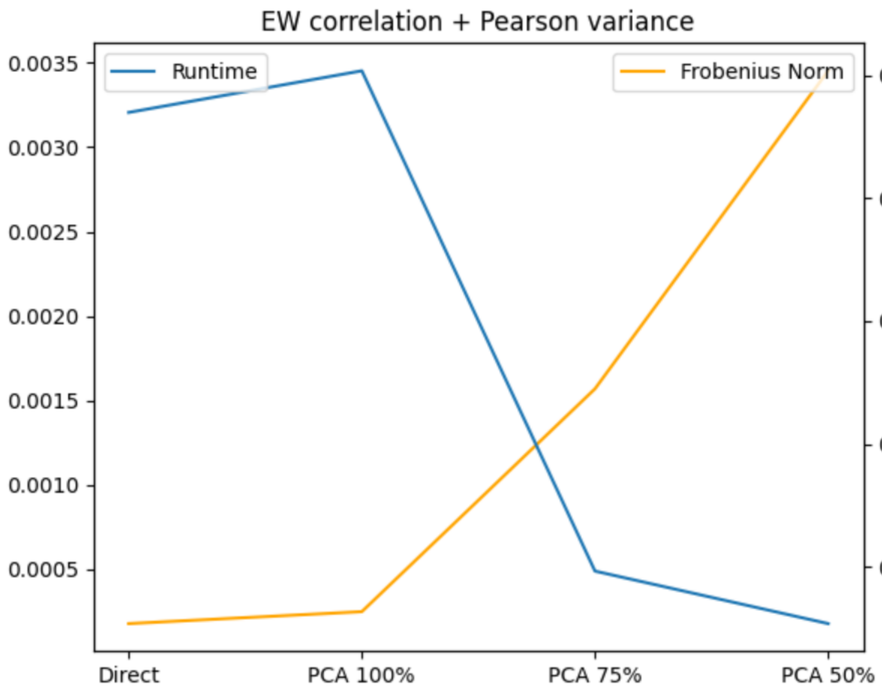
Pearson correlation + variance



Pearson correlation + EW variance







By the plots, we can see that the Frobenius Norm is the smallest by Direct Simulation, which is a little better than the PCA with 100% explained. As the percentage of PCA explained grows, the norm increase evidently, which means less accuracy. However, the runtime of Direct Simulation is long, which is longer than most PCA methods. If we need high accuracy, we can choose the Direct Simulation; otherwise, we can choose PCA with 75%-100% explained, because in this part, the Norm increases not too much but the runtime drops obviously.