Response for Project Week 3

Course Code: Fintech545

Course Title: Quantitative Risk Management

Student Name: Wanglin (Steve) Cai

Student Net ID: WC191

Problem 1

1.1 Implementation of Exponentially Weighted Covariance Matrix

In this question, at first it asks us to create a routine for calculating an exponentially weighted covariance matrix.

I implemented this method by using the given datafile DailyReturn.csv as an example. At first, we need to look at some basic statistics of this dataset:

1) This dataset contains 60 rows (data points) and 102 columns (variables).
2) The first variable is date, which is from 2021/10/21 to 2022/1/14.
3) The other variables are stock daily returns from different stocks.

	Unnamed: 0	SPY	AAPL	MSFT	AMZN	TSLA	GOOGL	GOOG	FB	NVDA	...	PN
0	2021/10/21	0.002608	0.001474	0.010897	0.005842	0.032571	0.000825	0.002566	0.003228	0.026648	...	-0.00032
1	2021/10/22	-0.001036	-0.005285	-0.005149	-0.028955	0.017539	-0.030443	-0.029104	-0.050515	0.001498	...	0.01574
2	2021/10/25	0.005363	-0.000336	-0.003332	-0.004551	0.126616	-0.000869	0.001068	0.012569	0.019361	...	0.00222
3	2021/10/26	0.000900	0.004575	0.006426	0.016775	-0.006274	0.013543	0.006478	-0.039186	0.066952	...	-0.00226
4	2021/10/27	-0.004430	-0.003148	0.042114	0.004864	0.019078	0.049595	0.048367	-0.011368	-0.010762	...	-0.01462

5 rows × 102 columns

Fig 1. Basic Statistics of the Dataset

Then, according to the simple exponential smoothing model from the notes, the variance for tomorrow is updated based on today's forecast variance and today's deviation from the mean.

$$\sigma_t^2 = \lambda\sigma_{t-1}^2 + (1 - \lambda)\left(x_{t-1} - \bar{x}\right)^2$$

Fig 2. Simple Exponential Smoothing Model

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

$$\widehat{w_{t-i}} = \frac{w_{t-i}}{\sum_{j=1}^{n} w_{t-j}}$$

Fig 3. Normalization of the Weights

After normalization of the weight, I can calculate the weighted covariance matrix by using the formula below:

$$\widehat{cov(x,y)} = \sum_{i=1}^{n} w_{t-i}\left(x_{t-i} - \bar{x}\right)\left(y_{t-i} - \bar{y}\right)$$

Fig 4. Formula for Calculating Estimators for Exponentially Weighted Covariance Matrix

Using the dataset as an example, the calculated weighted covariance sub-matrix by this method is shown below.

```
wcov = weightedcov(np.asmatrix(DR.iloc[:,1:102]), L = 0.97)
pd.DataFrame(wcov).iloc[:5,:5]
```

✓ 0.0s

	0	1	2	3	4
0	0.000004	0.000015	0.000019	0.000017	0.000027
1	0.000015	0.000173	0.000056	0.000097	0.000082
2	0.000019	0.000056	0.000177	0.000056	0.000083
3	0.000017	0.000097	0.000056	0.000228	-0.000106
4	0.000027	0.000082	0.000083	-0.000106	0.002008

Fig 5. Example of Weighted Covariance Sub-Matrix (First Five Variables)

1.2 PCA and Plot of Cumulative Variance Explained by Eigenvalues

By using PCA formula from the notes, we can plot the cumulative variance explained by each eigenvalue with different input parameter, which is lambda.

The plot is shown below. In this plot, I set lambda to be 0.1, 0.3, 0.5, 0.7, 0.9 ad 0.97. By gradually increasing the lambda, the cumulative variance explained curve should be different.

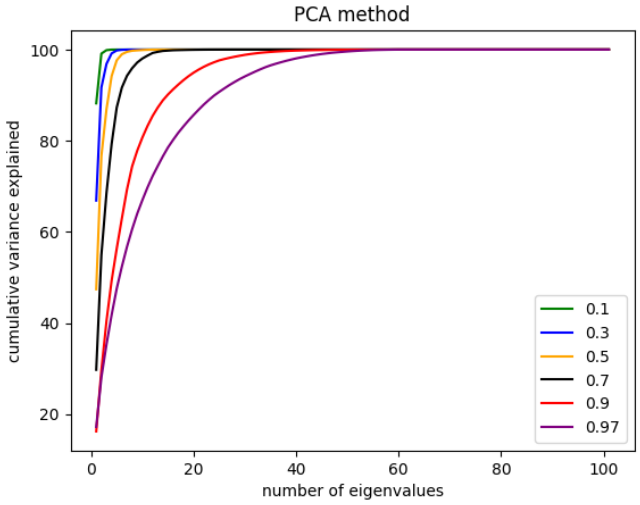

Fig 6. Plot of Cumulative Variance Explained Curve with Different Lambda

Since the row of the dataset is 60, so the positive eigenvalues cannot exceed 60. This is why the cumulative variance explained are the same for different lambda after 60 eigenvalues.

From the plot, I can also see that when lambda is small, the curve is ‘steep’, which means that cumulative variance explained approach to 1 after few values. This means that the covariance can be explained by a small number of eigenvalues. As the lambda becomes larger and larger, the curve becomes more ‘flat’, more and more eigenvalues are needed to keep the cumulative variance explained at the same level.

Problem 2

2.1 Implementation of chol_psd() and near_psd() functions

From the notes and the code provided, I can implement these two functions using Python. To test these two functions, I need to two different matrices. For the chol-psd(), I can utilize the weighted covariance matrix generated in the first problem. For the near_psd(), I can generate N*N matrix which is not PSD by the given code. Then, I can use these matrices to test the two functions.

```
[[1.    , 0.7357, 0.9   , ..., 0.9   , 0.9   , 0.9   ],
 [0.7357, 1.    , 0.9   , ..., 0.9   , 0.9   , 0.9   ],
 [0.9   , 0.9   , 1.    , ..., 0.9   , 0.9   , 0.9   ],
 ...,
 [0.9   , 0.9   , 0.9   , ..., 1.    , 0.9   , 0.9   ],
 [0.9   , 0.9   , 0.9   , ..., 0.9   , 1.    , 0.9   ],
 [0.9   , 0.9   , 0.9   , ..., 0.9   , 0.9   , 1.    ]])
```

Fig 7. Non-PSD Matrix Generated

```
root_matrix = chol_psd(wcov)
np.dot(root_matrix,root_matrix.T)

✓ 0.1s

ray([[ 3.86288511e-06,  1.52059856e-05,  1.93204344e-05, ...,
      -1.42056816e-05, -1.16327578e-05,  5.01017431e-07],
     [ 1.52059856e-05,  1.72836814e-04,  5.57443294e-05, ...,
       -4.88781235e-05, -4.03577438e-05,  4.27103200e-06],
     [ 1.93204344e-05,  5.57443294e-05,  1.77344032e-04, ...,
       -7.96223664e-05, -7.90683802e-05, -1.86135008e-05],
     ...,
     [-1.42056816e-05, -4.88781235e-05, -7.96223664e-05, ...,
        4.87878044e-04,  3.39439295e-05,  5.03572341e-05],
     [-1.16327578e-05, -4.03577438e-05, -7.90683802e-05, ...,
        3.39439295e-05,  1.47587815e-04, -1.94320736e-05],
     [ 5.01017431e-07,  4.27103200e-06, -1.86135008e-05, ...,
        5.03572341e-05, -1.94320736e-05,  1.85826826e-04]])
```

```
wcov

✓ 0.1s

ray([[ 3.86288511e-06,  1.52059856e-05,  1.93204344e-05, ...,
      -1.42056816e-05, -1.16327578e-05,  5.01017431e-07],
     [ 1.52059856e-05,  1.72836814e-04,  5.57443294e-05, ...,
       -4.88781235e-05, -4.03577438e-05,  4.27103200e-06],
     [ 1.93204344e-05,  5.57443294e-05,  1.77344032e-04, ...,
       -7.96223664e-05, -7.90683802e-05, -1.86135008e-05],
     ...,
     [-1.42056816e-05, -4.88781235e-05, -7.96223664e-05, ...,
        4.87878044e-04,  3.39439295e-05,  5.03572341e-05],
     [-1.16327578e-05, -4.03577438e-05, -7.90683802e-05, ...,
        3.39439295e-05,  1.47587815e-04, -1.94320736e-05],
     [ 5.01017431e-07,  4.27103200e-06, -1.86135008e-05, ...,
        5.03572341e-05, -1.94320736e-05,  1.85826826e-04]])
```

Fig 8. Comparison of Matrix for chol_psd()

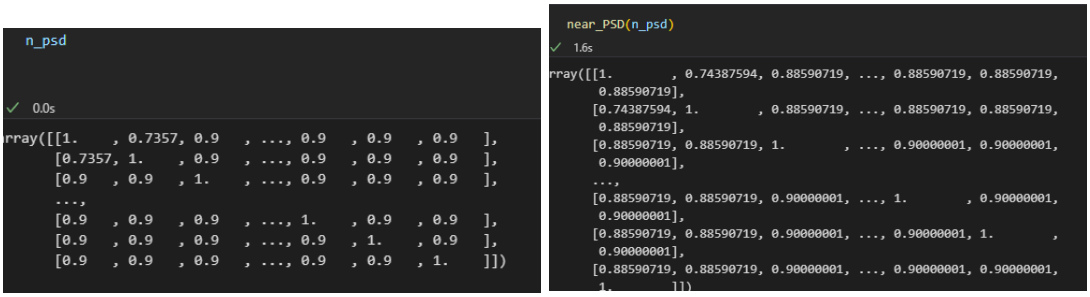

Fig 9. Comparison of Matrix for near_psd()

After generating the matrices and test these two functions, I get the results that we want. These two functions are implemented successfully.

2.2 Higham's 2002 Nearest PSD Function

The next step, I implemented the Higham() function. The function is implemented by following the notes. I also check the result of this function.

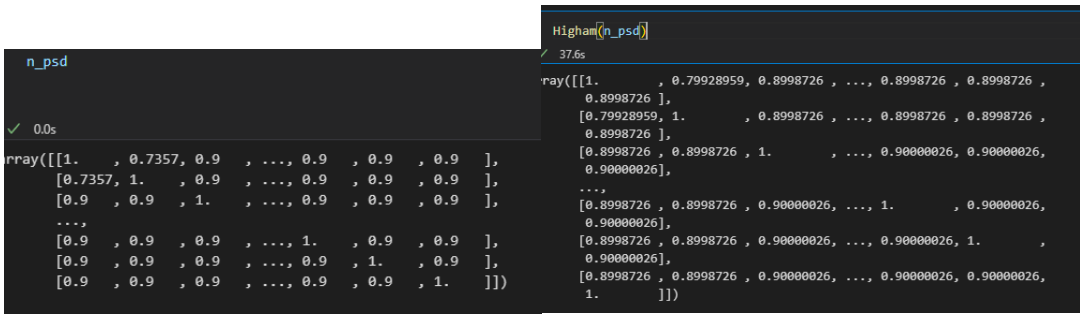

Fig 10. Comparison of Matrix for Higham()

By comparing the result, we can say the function is implemented successfully.

2.3 Confirm the Matrix is PSD

By checking the eigenvalues of the new matrix generated by near_psd() and Higham(), I found that the eigenvalues for them are all positive. Therefore, the matrix generated is PSD.

Fig 10. All eigenvalue is positive

2.4 The Frobenius Norm and Run Time

I choose N to be 50 to 500, with interval of 50 for each N. For the Frobenius Norm, it is the result of the difference between the input matrix and after its psd version. It is a scalar value that gives us information about how close the "near PSD matrix" is to the original input matrix in terms of magnitude. The smaller the Frobenius norm, the closer the two matrices are.

The result of the Frobenius Norm is below.

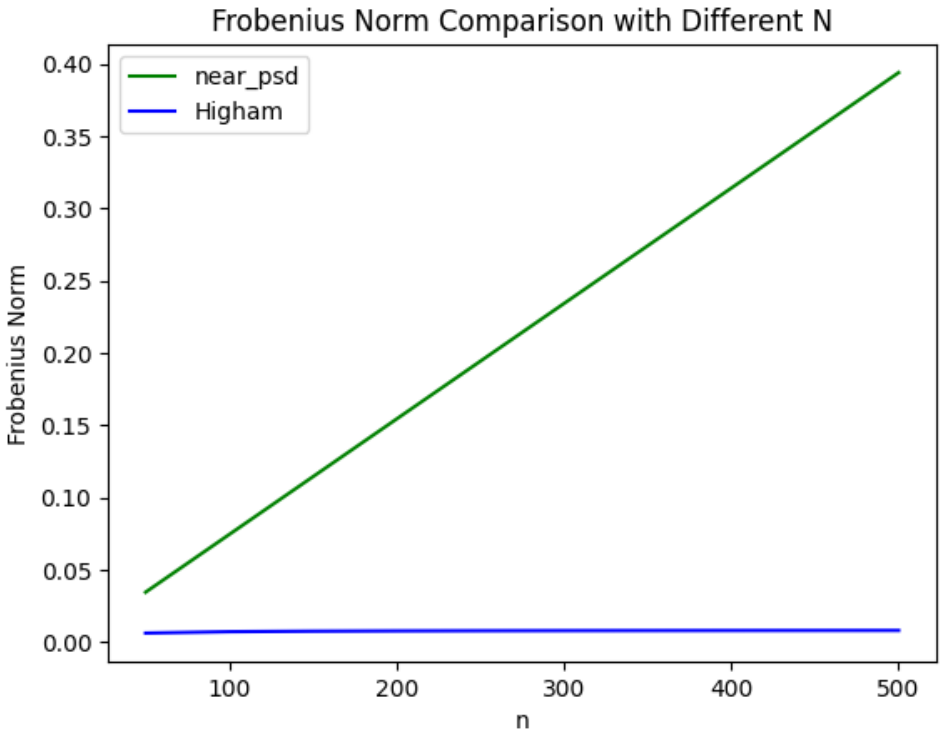

Fig 11. Result of the Norm Comparison with Different N

As N increases, the Frobenius norm of Higham method keeps at a low level, which means the accuracy keeps at high level. And the curve seems to be totally flat. While the Frobenius norm of near_psd method increase linearly with n, which means the accuracy decreases at a linear speed.

As for the run time, the result is also shown below.

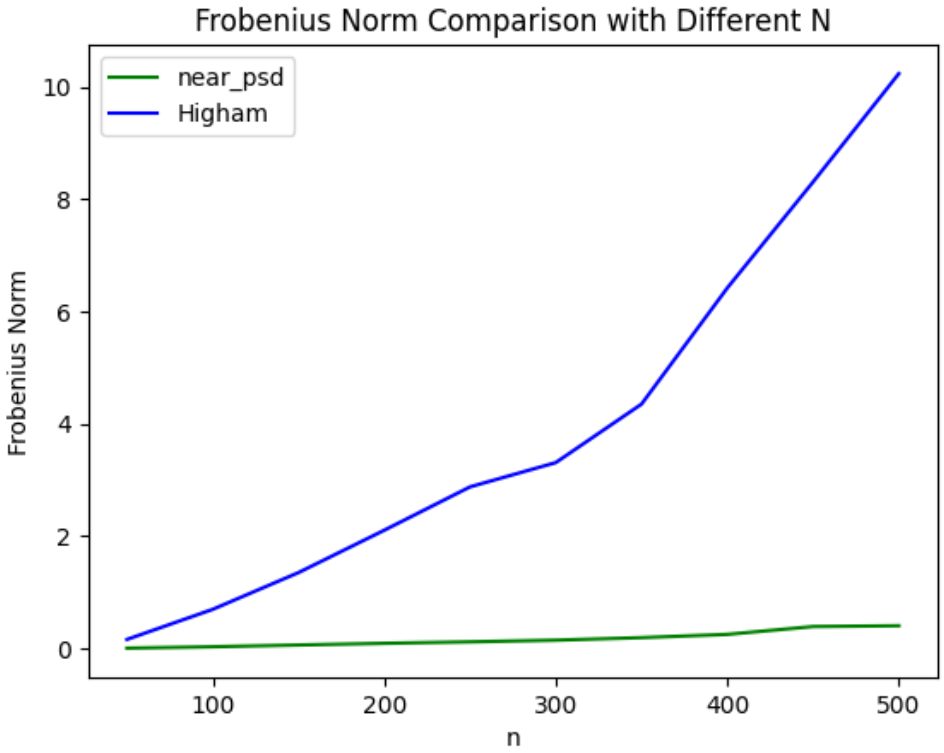

Fig 12. Result of the Run Time Comparison with Different N

For the run time of these two methods, the result behaves differently from the above result. As N increases, the run time of near_psd method keeps at a low level, which means the efficiency keeps at high level. While the run time of Higham method increase with n, which means the efficiency is relatively low as N becomes larger and larger.

2.5 Pros and Cons of the Two Methods

It is a tradeoff between efficiency and accuracy.

When we want the run time to be as small as possible such as we are doing high frequency trading, we probably want to choose near_psd() to get efficiency but sacrifice some accuracy. If we are doing some work which is not time sensitive and need more accuracy, Higham's method will likely be chosen in this case.

Method	Pros	Cons
Near_psd	Fast convergence: A good choice for large scale problems	Accuracy might be sacrificed a lot as N increases
Higham	Accuracy guaranteed even though N increases	Slower convergence: Less suitable for large scale problems.

Table 1. Pros and Cons of the Two Methods

Problem 3

3.1 Implementation of Multivariate Normal Simulation

According to the notes and the code provided, I built a function that implement the multivariate normal simulation. As we can see in the result below, when the number of simulations is 2000, the result matrix is pretty close to the original one. Therefore, it is safe to say that the function is as expected.

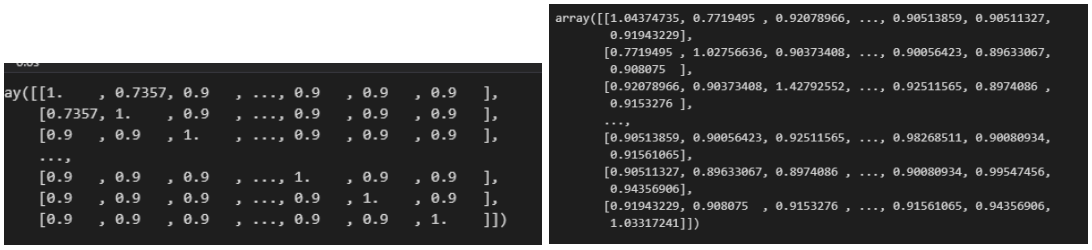

Fig 13. Comparison of Matrix for PCA Simulation with nsim=2000

3.2 Four Different Covariance Matrices

For this part, the four covariance matrices are:

1) Pearson correlation and var
2) Pearson correlation and EW variance
3) EW(correlation + correlation)
4) EW correlation + var

Next, we are going to simulate 25000 times using:

1 Direct Simulation

2 PCA with 100% Explained

3 PCA with 75% Explained

4 PCA with 50% Explained

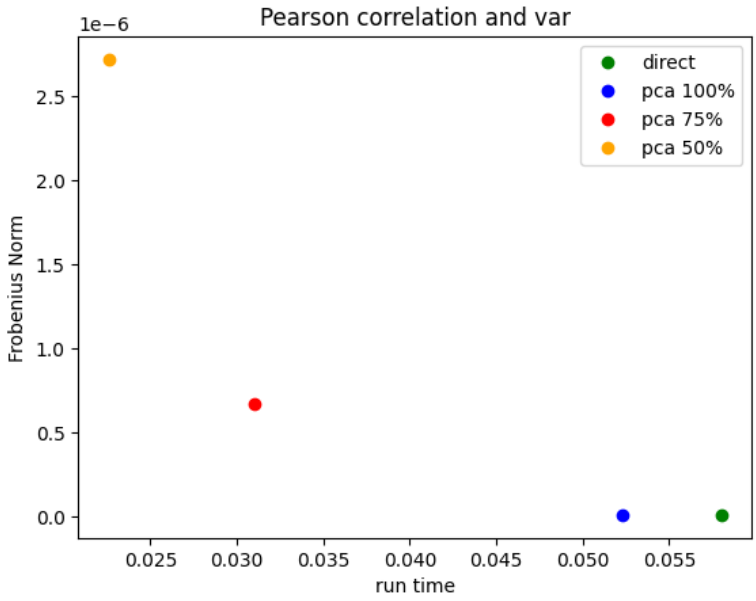

Fig 14. Comparison of Different Simulation Method Using Pearson Corr and Var

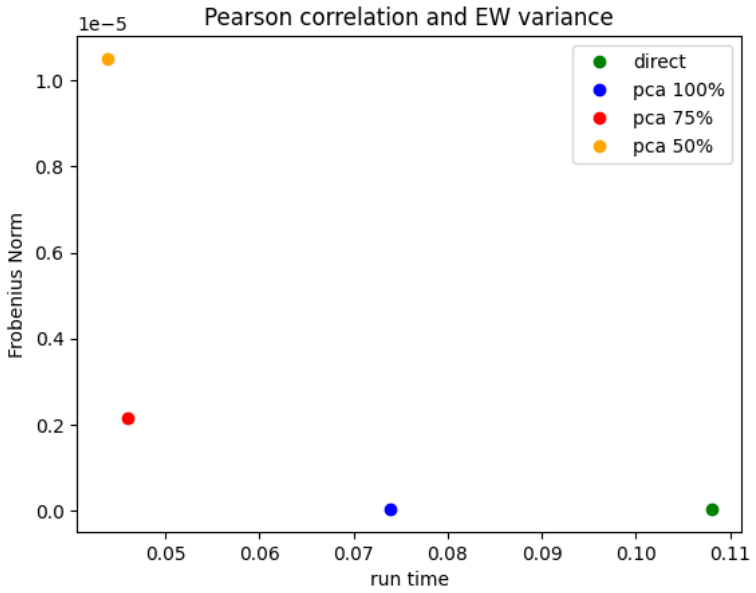

Fig 15. Comparison of Different Simulation Method Using Pearson correlation and EW variance

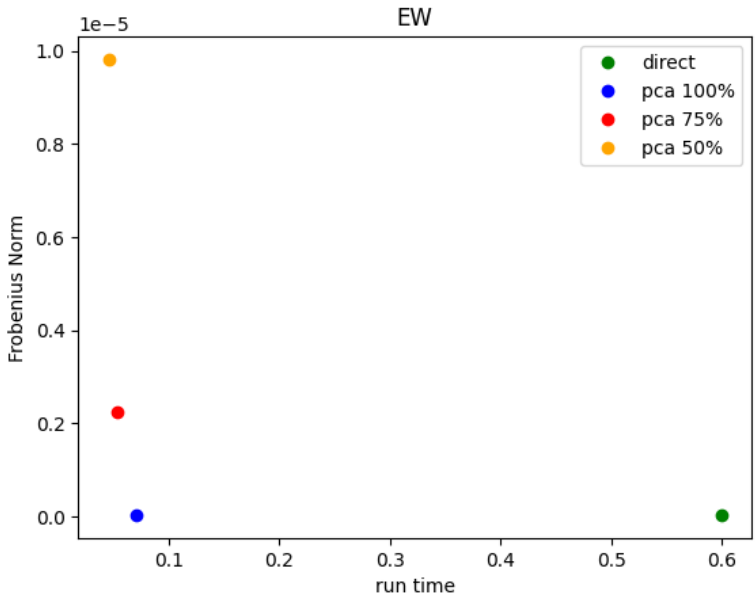

Fig 16. Comparison of Different Simulation Method Using EW(correlation + correlation)

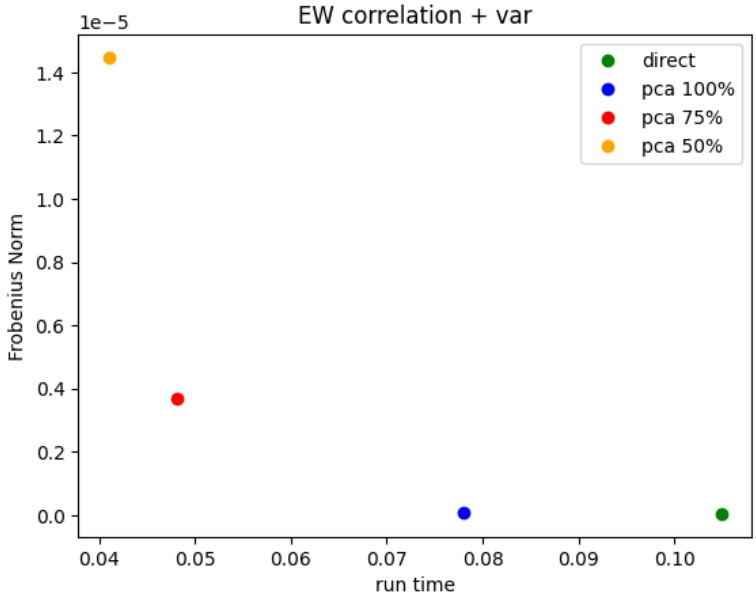

Fig 17. Comparison of Different Simulation Method Using EW correlation + var

First of all, when I compare the simulated covariance to its input matrix using the Frobenius Norm. I found that among these four different matrices, the original one, which is the Pearson correlation with Var one, has the lowest Frobenius Norm. This means that this matrix's accuracy is the best among these four matrices. Besides, the accuracy drops dramatically after PCA with 75% explained.

Besides, we can see that for different simulation method, the Frobenius Norm is

different. For direct method/PCA 100% explained, their norm is low since they don't drop many information compared with PCA 75% explained/50% explained. This makes sense/intuitive since more information means more accuracy.

As for the run time, the run time decreases as the level of % variance explained decrease for all matrices. This is also intuitive because with less information to compute, the run time should be faster.

For the tradeoffs, as the level of % variance explained decrease, the run time is shorter while the accuracy is decreased. The tradeoff here is the same problem we faced in Problem 2 (efficiency and accuracy). When we want the run time to be as small as possible such as we are doing high frequency trading, we probably want to get efficiency but sacrifice some accuracy. If we are doing some work which is not time sensitive and need more accuracy, a more accuracy method will be chosen accordingly.