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 $\lambda \in (0, 1)$. 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?

The function for calculating an exponentially weighted covariance matrix will take time series of asset returns (x: asset returns, y: time) and lambda as inputs. The detailed steps and explanations are as follows.

1. Calculate a weight vector which will be a function of time. This can be done according to

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

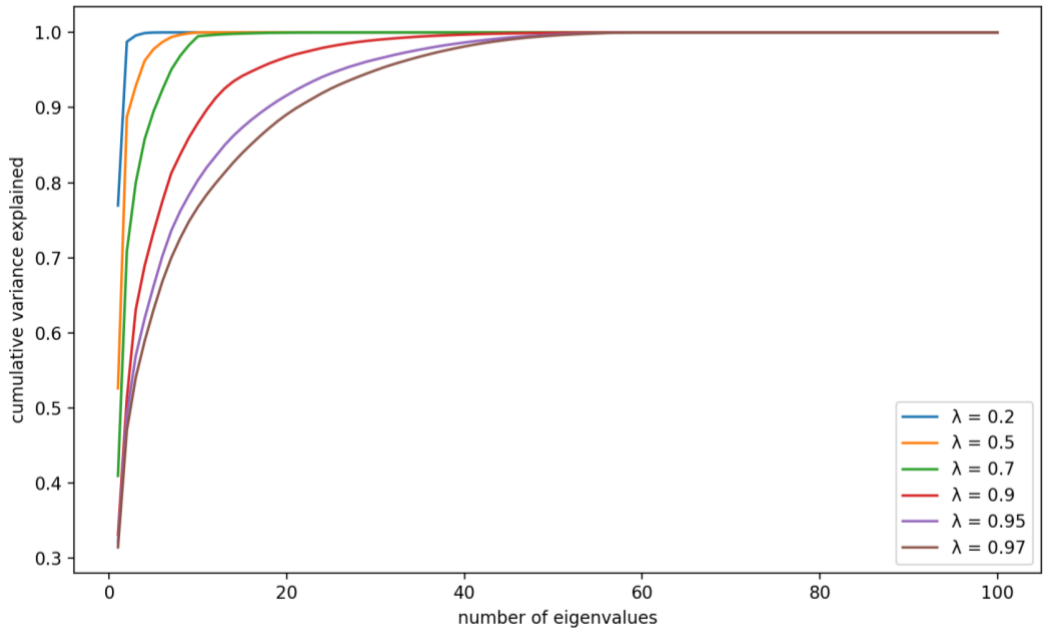
   Weights need to be normalized so that they sum to 1. The same weight will be applied to all asset returns at the same moment (e.g., day) in time.
2. To calculate covariances, the returns for each asset need to be adjusted by subtracting the mean return for that asset.

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

3. Given the above formula, the adjusted returns and the normalized weights, the covariance matrix can be obtained by
   a. The t * t matrix with the weight vector on the diagonal and 0 elsewhere multiplies the t *A adjusted return matrix, where A is the number of assets.
   b. The A* t transposed adjusted return matrix multiplies the product from a.

To plot the cumulative variance explained vs. the number of eigenvalues for each λ chosen, we can create a function that takes the number of eigenvalues and a covariance matrix as inputs and returns the cumulative variance explained. We achieve this by decomposing the covariance matrix into a series of eigenvectors and eigenvalues. Eigenpairs with larger eigenvalues explain higher percentages of variances. The cumulative variance explained can by calculated using eigenvalues considered divided by total eigenvalues. We vary λ values to create different exponentially weighted matrices.

The resulted plot is as below.



λ determines the weight given to the more recent observations. The smaller the value, the recent observations are weighed more heavily. As the above plot suggests, a smaller λ has a larger percentage of variance explained for the same number of eigenvalues taken into account, while a larger λ has a smaller percentage of variance explained, as the weights are more equally distributed across observations at different times.

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. 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.

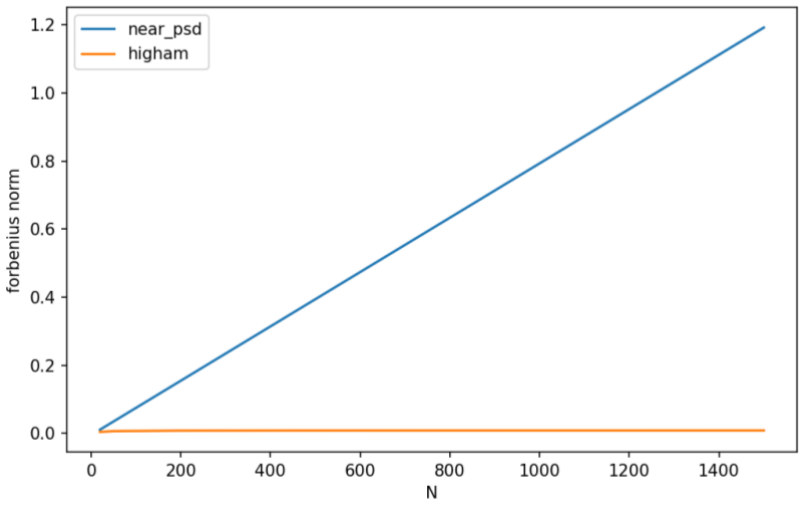
chol_psd() takes in a symmetric, PSD matrix and returns its Cholesky root. It can be implemented according to chol_psd() in week03.jl or following the algorithm detailed on page 3 of the notes. near_psd() takes in a non-PSD correlation or covariance matrix and returns a near, acceptable PSD covariance matrix. It can be implemented according to near_psd() in week03.jl or following the algorithm detailed on page 8 of the notes (Rebonato and Jackel's spectral decomposition). Higham's nearest PSD correlation function (2002) takes in a non-PSD correlation matrix and returns a near, PSD correlation matrix. It can be implemented following the algorithm detailed on page 9-10 of the notes.

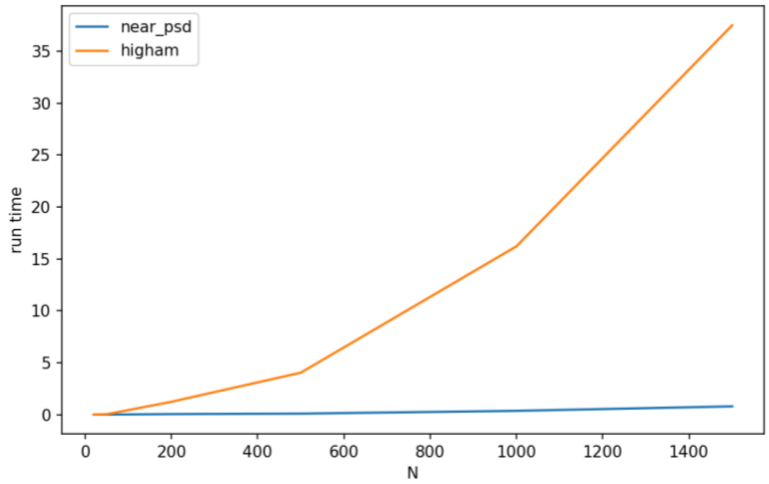
A 500x500 non-PSD correlation matrix was generated and then fixed through using near_psd() and Higham's method. The original matrix became PSD after the respective fixes.

As N increases, the results of the two fixes in terms of the Frobenius Norm are as below. We can tell that Higham's method returns a PSD correlation matrix that is extremely close to the non-PSD input, regardless of the size of the input matrix. The resulting matrix from near_psd(), however, becomes further and further from the input matrix as N increases. The Frobenius Norm seems to have a strong, positive, linear relationship with N.

The runtime of the fixes as N increases is also illustrated below. The plot shows that as N increases, the runtime of near_psd() remains slightly above 0 second. The runtime of Higham's algorithm, on the other hand, increases as the size of the input matrix becomes larger. More specifically, it seems that it increases at a faster rate as N increases.

The pros and cons of each method become apparent given the above discussion. near_psd() is fast yet less accurate, whereas Higham's method is much more accurate yet takes a lot longer to run especially as N becomes large. The specific risk management scenario and objectives dictate which to use. For example, in a real-time risk management scenario where quick decisions are required (e.g., high-frequency trading, real-time fraud detection), I would prefer near_psd(). On the other hand, in a situation where the objective is to estimate the long-term risk (e.g., when making a long-term investment strategy), I would prefer Higham's method. Some other scenarios I can think of in which I would vote for Higham's approach include stress testing and portfolio optimization.





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 $\lambda = 0.97$**

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 its input matrix using the Frobenius Norm (L2 norm, sum of the square of the difference between the matrices). Compare the runtimes for each simulation.
What can we say about the trade-offs between time to run and accuracy.

The direct simulation is based on the Cholesky factorization. We know that

$$X \sim N(\mu, \Sigma)$$

We can simulate X through $X = LZ + \mu$, where L can be obtained by using chol_psd().

For the method using PCA, as I mentioned previously, we decomposed the covariance matrix into a series of eigenvectors and eigenvalues. Eigenpairs with larger eigenvalues give us more important "principal components," which explain higher percentages of variances. We know that

$$C = S \Lambda S^T$$

Where S are the eigenvectors and Λ is a diagonal matrix of eigenvalues.

$$C = \left[S \Lambda^{\frac{1}{2}}\right]\left[\Lambda^{\frac{1}{2}} S^T\right]$$

$$B = \left[S \Lambda^{\frac{1}{2}}\right]$$

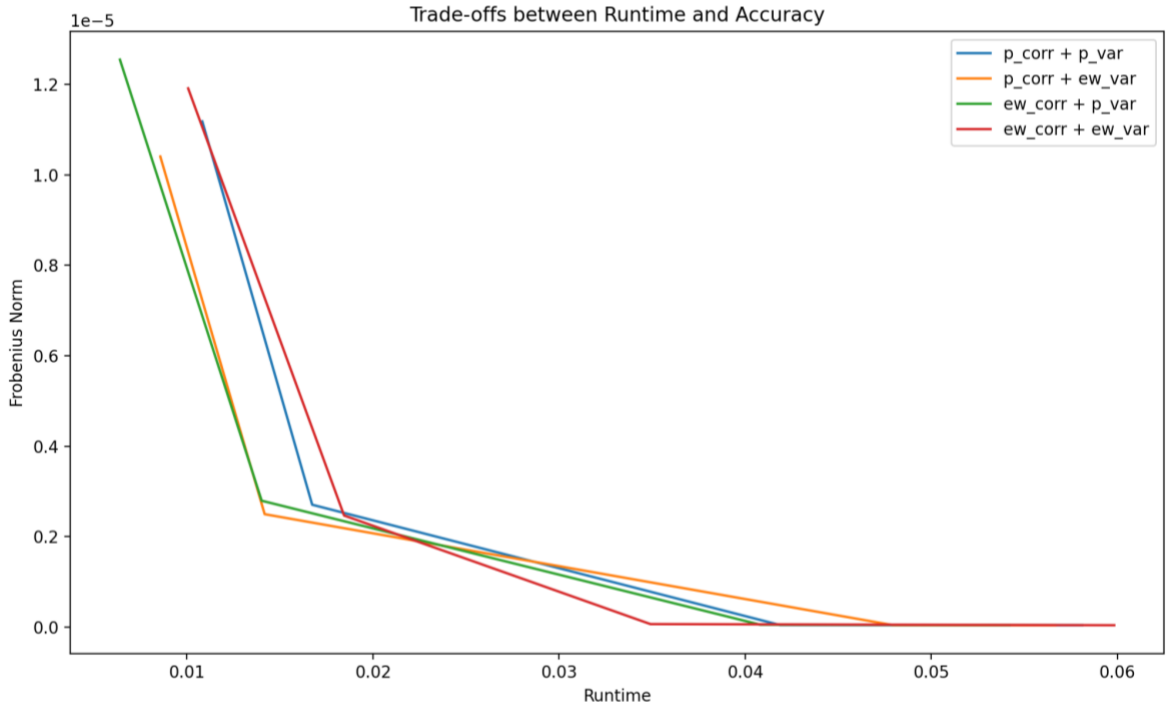
$$\Rightarrow C = BB^T$$

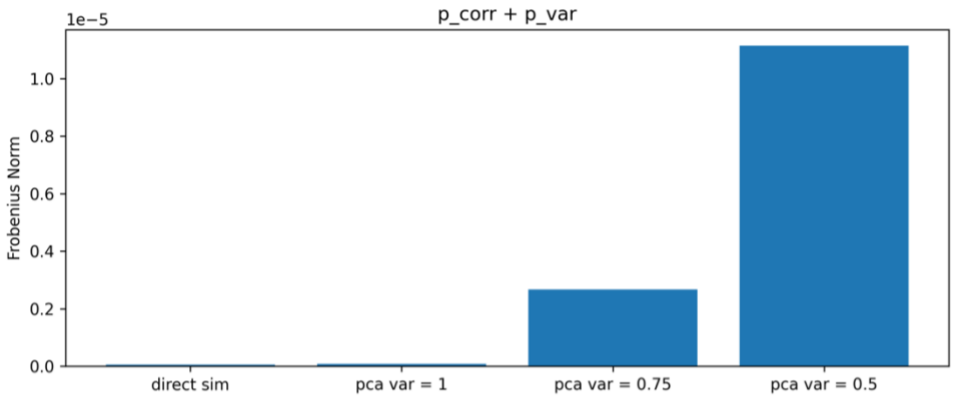
Given the above, we start with the eigenpair that has the largest eigenvalue and observe how the variance explained changes as more eigenvalues get included. Once the variance explained reaches the variance we specify, we know what S and Λ should be. Then by the calculation above, we obtain B, which provides an alternative to the Cholesky factorization L.

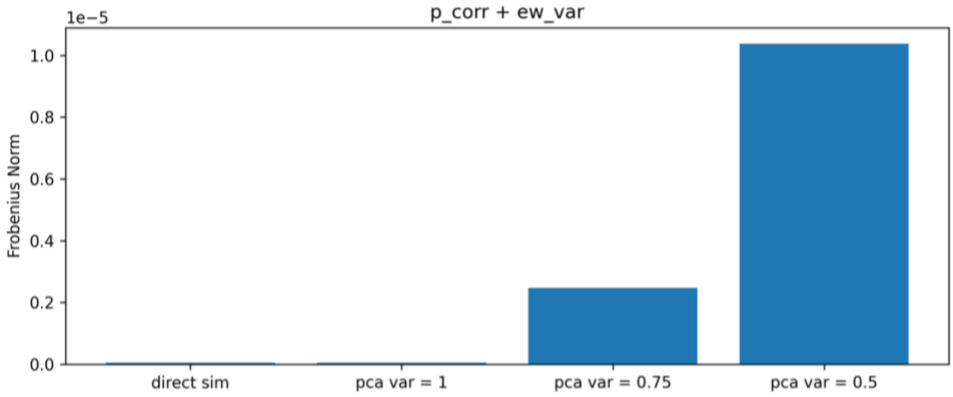
Next, we generate two correlation matrices and two variance vectors through the standard Pearson approach and the exponentially weighted approach with $\lambda = 0.97$ (Pearson correlation,

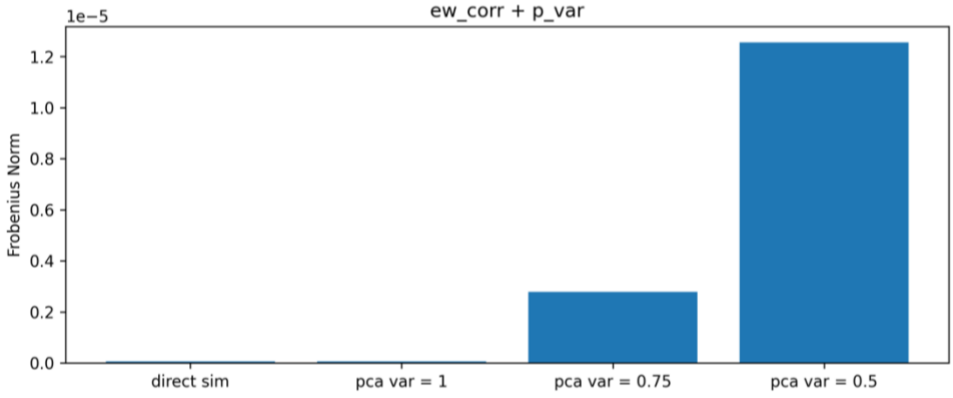
Pearson variance, exponentially weighted correlation, exponentially weighted variance). We then generate four covariance matrices given the two correlation matrices and the two variance vectors (four combinations in total). For each covariance matrix, we simulate 25,000 draws using direct simulation and PCA with 100%/75%/50% explained. Altogether we need to run 16 simulations. We run these simulations to compare the covariances of the simulated vs. the input (real) covariance. This can be evaluated through the Frobenius Norm. We also would like to compare the runtimes for each kind of simulation (direct vs. PCA with 100% vs. PCA with 75% vs. PCA with 50% variance explained).

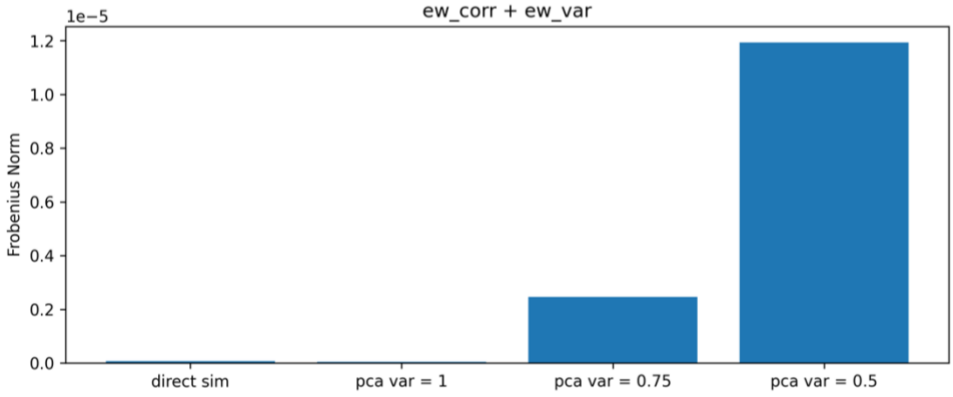
The simulation process is rather straightforward, using the functions written earlier that take a covariance matrix as the input. The results and discussions are as follows.

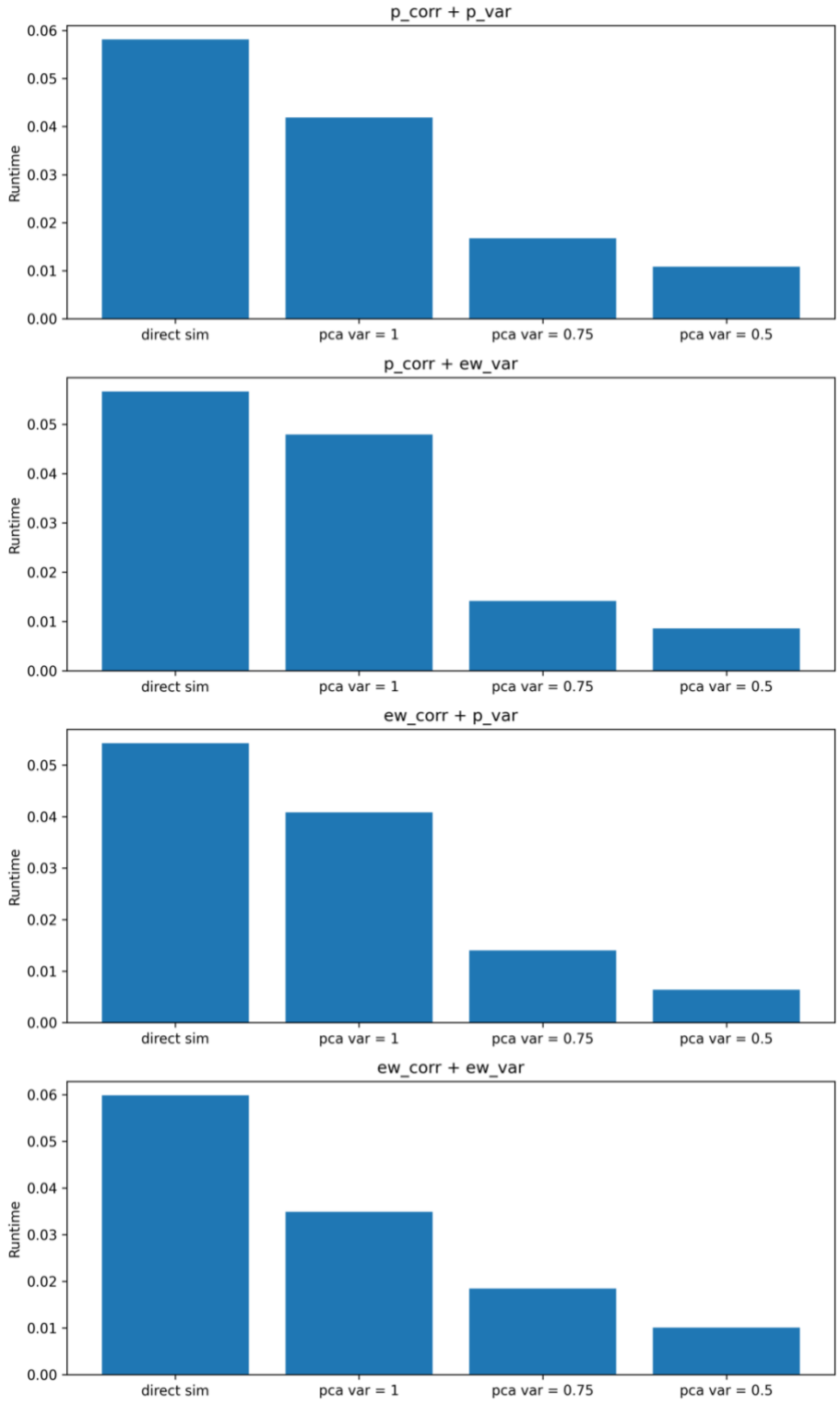



p_corr + p_var
1e−5
Frobenius Norm
0.0
0.2
0.4
0.6
0.8
1.0
direct sim
pca var = 1
pca var = 0.75
pca var = 0.5


p_corr + ew_var
1e−5
Frobenius Norm
0.0
0.2
0.4
0.6
0.8
1.0
direct sim
pca var = 1
pca var = 0.75
pca var = 0.5


ew_corr + p_var
1e−5
Frobenius Norm
0.0
0.2
0.4
0.6
0.8
1.0
1.2
direct sim
pca var = 1
pca var = 0.75
pca var = 0.5


ew_corr + ew_var
1e−5
Frobenius Norm
0.0
0.2
0.4
0.6
0.8
1.0
1.2
direct sim
pca var = 1
pca var = 0.75
pca var = 0.5



From the first plot, we observe a similar trend in all four covariance matrices — the smaller the sum of the squared differences between the input and the simulated, the longer it takes to run the

simulation. The second and the third groups of plots further validate this observation. The direct simulation and the PCA with 100% explained are the most accurate, the PCA with 75% explained comes the second and the PCA with 50% explained the last. On the other hand, the least accurate simulation takes the shortest amount of time to run. The direct simulation and the PCA with 100% explained simulation are close in their accuracy, but the Cholesky factorization involves a nested for loop that contributes to its higher time complexity.