Project Week03: Leslie Dees

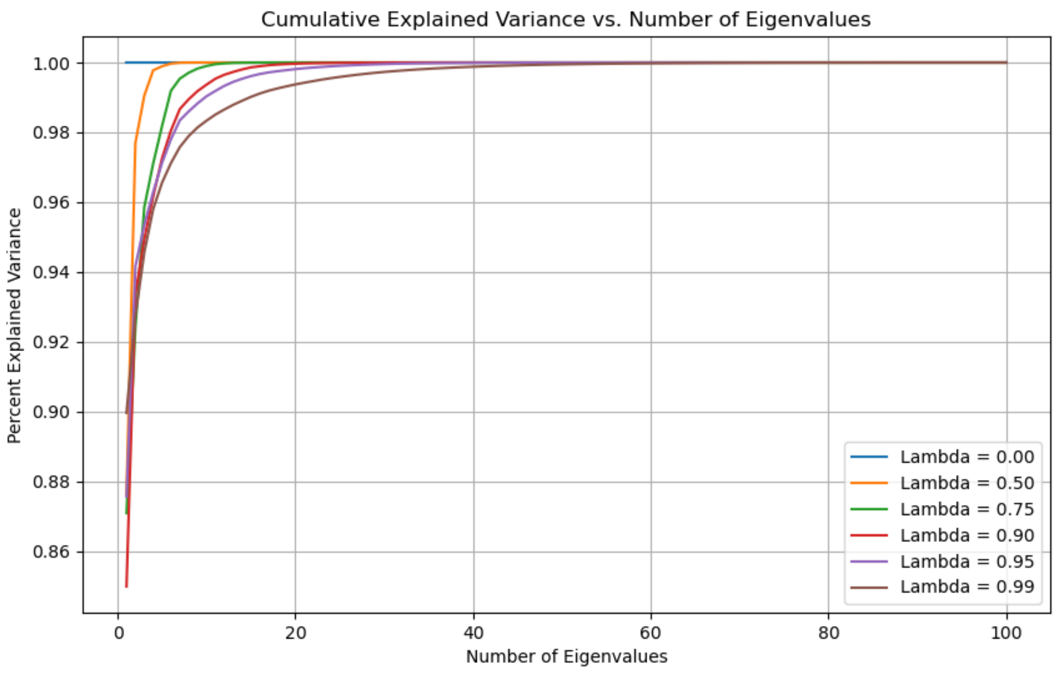
NOTE: Methods used in the code are referenced specifically in fin_package3.py. These will be relocated to fin_package.py later in the semester once consolidation is completed.

Problem 1:

Create a routine for calculating exponentially weighted covariance matrix

I created a function *calculate_ewma_covariance_matrix(df, lambda)* that takes in a dataframe and a lambda value to create an exponentially weighted covariance matrix. I followed the formula in the notes for the normalized covariance matrix in order to do this. The results are shown in p1_work.py.

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



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

What I can see from this plot is that as lambda increases to 1.0, The greater number of eigenvalues are required to fully represent the explained variance based on the PCA. In terms of lambda, it in essence controls the weights allocated to prior historical values. As lambda increases, the percentage explained per N decreases, where N is the number of eigenvalues. If lambda is small then the cumulative weight of explained variance increases to 100% very quickly. Effectively, few observations account for almost all of the information since the lower lambda values much more heavily favor recent historical data. In terms of the effect on the covariance matrix, the effective information contained in it is directly related to lambda. The larger that lambda gets, the more information is contained in the covariance matrix based on more historical relationships.

Problem 2:

Reimplement chol_psd() and near_psd() functions.

Both functions were adequately reproduced inside of fin_package_3. I required the use of *np.linalg.eigh()* to calculate the eigenvalues due to the non-psd nature of the original matrix.

Implement Higham's 2002 nearest psd correlation function

Higham's 2002 algorithm was reproduced in fin_package_3 as well, but the assumption of a symmetric matrix was required in order to use the *np.linalg.eigh* function to obtain eigenvalues. Within my algorithm reproduced in my package I commented on lines that were directly correlating to the sections of the algorithm detailed in the notes.

Confirm the matrix is now PSD using both the class and Higham's functions

I confirmed that the matrix is now PSD using 2 methods. Firstly, I created a function called *is_psd* which checks that all eigenvalues are positive. If they are not, then it prints the eigenvalues that are not. I did see some issue with this however, where in several runs for both the Higham and class function, the output would show a very small, complex, and negative number such as -4.23250844e-15+0.j.

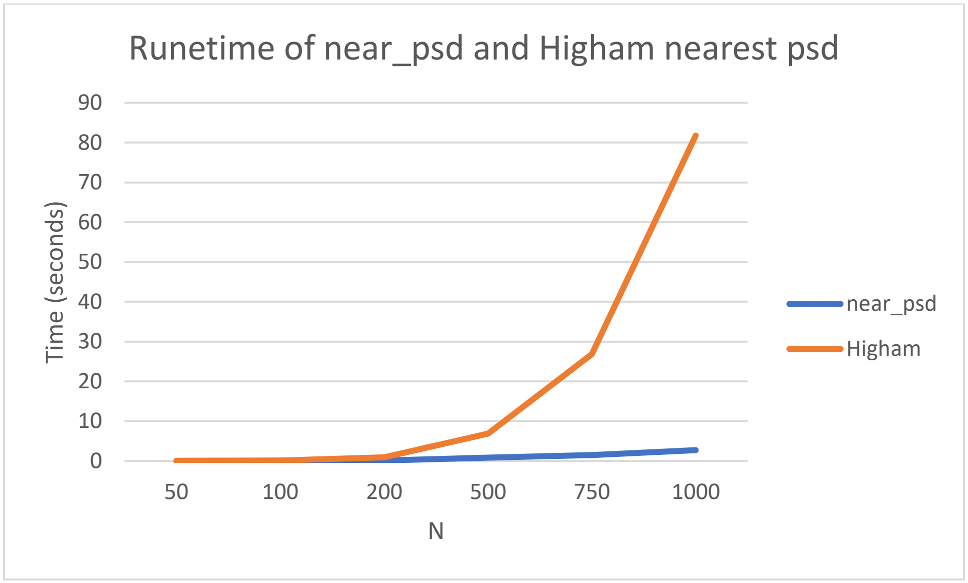
This output led me to test it directly in the Cholesky function that I reproduced. While the issues of the very small, complex, negative eigenvalue was captured in my personal function, I was able to run the *chol_psd()* without error. This validates that both of the methods adequately turn matrices PSD. This is observable in the code where I test N=500, and the output obtained displays a small, complex, and negative eigenvalue yet still is able to run in the Cholesky function without fail.

Compare the results using the Frobenius Norm and compare the runtime between the two. How does the runtime compare as N increases?

N	near_psd time (seconds)	higham_near_psd time (seconds)	Frobenius Norm
50	0.015	0.051	0.168
100	0.031	0.150	0.259
200	0.138	0.884	0.383
500	0.853	6.873	0.621

750	1.481	26.840	0.765
1000	2.715	81.77	0.889

I can see that as N increases, the Frobenius Norm between the two increases at a relatively linear scale. It is difficult to tell however which has the values that are best since the "true value" of the matrix cannot be determined in this PSD modification method. The drastic difference is the rate at which the Higham function increases in runtime compared to the *near_psd()* function. The Higham function appears to increase at a semi-exponential rate in runtime while the *near_psd()* function increases rather linearly.



Based on the above, discuss the pros and cons of each method and when you would use each.

Firstly, I can tell that in cases where my correlation matrix is very large I would prefer to not use Higham's method. The growth of the runtime being near-exponential would be extremely detrimental to working on datasets with a huge number of features. In cases where there are many features I would prefer to use the *near_psd()* method instead.

However, there is something to be said about the widespread use of Higham's function. I would like to believe that an iterative algorithm would be able to best optimize the PSD nature of the matrix, and so I would like to believe it to be a more accurate final result. In cases where the number of features is small, I would lean towards using Higham's function.

Problem 3:

Implement a multivariate normal simulation that allows for simulation directly from a covariance matrix or using PCA with an optional parameter for % variance explained.

This was implemented through the function *multivariate_normal_simulation()* in fin_package_3.py. I take in variables that either account for a Direct method or PCA method, and default to using a 1.0 or 100% explain ability for PCA unless otherwise indicated in the inputs.

Generate a correlation matrix and variance vector using Standard Pearson and Exponentially weighted lambda = 0.97.

Correlation matrix and variance vector for the standard Pearson approach were performed using the .corr() and .var() functions of pandas DataFrames. I utilized my own exponentially weighted covariance matrix function in order to determine that correlation matrix and variance vector.

Combine these two forms 4 different covariance matrices.

All combinations were performed inside of p3_work.py inside of Week03/Project

Simulate 25,000 draws from each 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 and compare them to the input matrix using the Frobenius norm and runtime.

1. **Direct Simulation**

Covariance Matrix	**Runtime (seconds**	**Frobenius Norm**
Pearson + var	0.23	0.9626
Pearson + EW var	0.27	0.4510
Exp Weighted + var	0.24	0.5158
Exp Weighted + EW var	0.27	0.5219

2. **PCA with 100% Explained**

Covariance Matrix	**Runtime (seconds**	**Frobenius Norm**
Pearson + var	0.10	0.7049
Pearson + EW var	0.12	0.5542
Exp Weighted + var	0.13	13.8998
Exp Weighted + EW var	0.11	13.8888

3. **PCA with 75% Explained**

Covariance Matrix	**Runtime (seconds**	**Frobenius Norm**
Pearson + var	0.11	3.4771
Pearson + EW var	0.09	3.4885
Exp Weighted + var	0.11	4.0868
Exp Weighted + EW var	0.12	4.1026

4. **PCA with 50% Explained**

Covariance Matrix	**Runtime (seconds**	**Frobenius Norm**
Pearson + var	0.10	7.4522
Pearson + EW var	0.12	7.4517
Exp Weighted + var	0.12	9.3027
Exp Weighted + EW var	0.13	9.3070

What can we say about the trade-offs between runtime and accuracy?

In general, the PCA explainability using the exponentially weighted covariances appears to be highly off in all cases. While I cannot find any reason for this in my code, I am worried that it could be a bug.

In terms of a trade-off though the Direct simulations took a larger runtime across the board. However, it was notable that they were also more accurate in every case other than the PCA with 100% explainability paired with the Pearson correlation matrices. In this particular case, accuracies were greater when the Pearson variance was used and slightly lower when the exponentially weighted variance vector was used. What this tells us though is that there appears to be a very direct relationship between greater accuracies and slower runtimes.

I also would like to note that the PCA analysis when explainability was less than 100% performed horribly in the Frobenius Norm cases across the board. I believe that this is due to key supporting features of the data being left out.