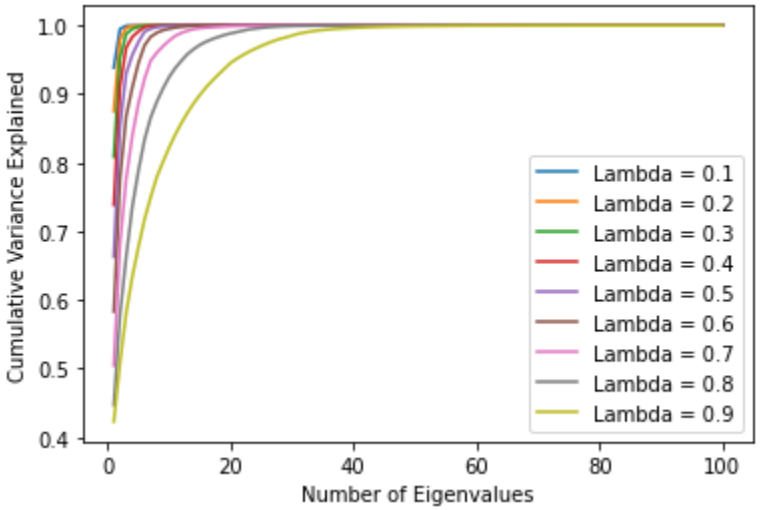
June Lee

Project 3

Problem 1)



Upon calculating the exponentially weighted covariance matrices for each of the λ values shown in the legend above, we obtain the plot shown above. The plot shows that a smaller λ value results in the first few eigenvalues explaining most of the total variance under PCA, while a larger λ value results in the amount of variance explained being spread out among additional eigenvalues. This is in line with expectations, as smaller λ values assign greater weights to recent observations and less weights to observations far into the past in comparison to larger λ values, resulting in the covariance matrix being mainly generated by recent movements in stock returns.

Problem 2)

Generating a non-PSD (500x500) matrix, using the code, we obtain the following matrix.

```
[[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.    ]]
Min Eigenvalue: (-0.06364303890475023+0j)
Original matrix is PSD:
False
```

This original matrix is not PSD as desired, with the minimum eigenvalue being negative. The test used to return “False” in the code is to check that all eigenvalues of the matrix are non-negative.

Inputting this matrix into near_psd(), we obtain the following matrix, which is indeed PSD as desired.

```
[[1.         0.74381947 0.88594237 ... 0.88594237 0.88594237 0.88594237]
 [0.74381947 1.         0.88594237 ... 0.88594237 0.88594237 0.88594237]
 [0.88594237 0.88594237 1.         ... 0.90000005 0.90000005 0.90000005]
 ...
 [0.88594237 0.88594237 0.90000005 ... 1.         0.90000005 0.90000005]
 [0.88594237 0.88594237 0.90000005 ... 0.90000005 1.         0.90000005]
 [0.88594237 0.88594237 0.90000005 ... 0.90000005 0.90000005 1.        ]]
near_psd output is PSD:
Min Eigenvalue: (1.0339602438125262e-14+0j)
True
```

Finally, inputting the original matrix into higham_near_psd(), with the tolerance parameter set to 10^{-16}, the weight matrix W set equal to the identity matrix I and the maximum number of iterations set to 500, we obtain the following matrix.

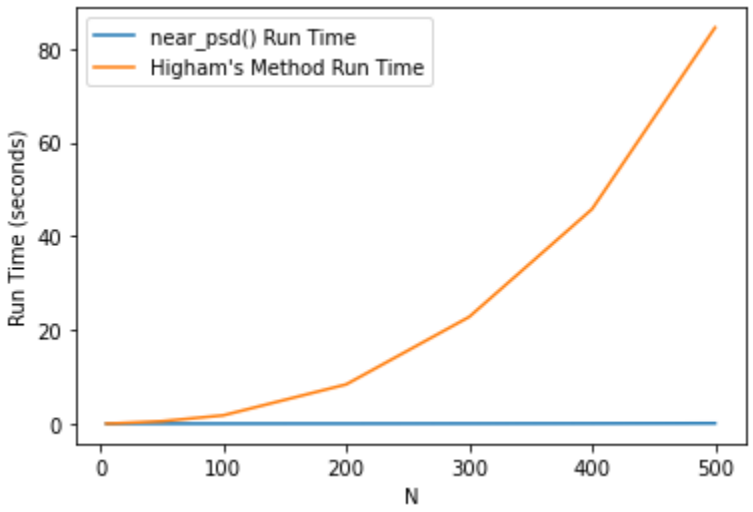
```
Reached Max Number of Iterations
[[1.         0.7985871  0.89974757 ... 0.89974757 0.89974757 0.89974757]
 [0.7985871  1.         0.89974757 ... 0.89974757 0.89974757 0.89974757]
 [0.89974757 0.89974757 1.         ... 0.90000101 0.90000101 0.90000101]
 ...
 [0.89974757 0.89974757 0.90000101 ... 1.         0.90000101 0.90000101]
 [0.89974757 0.89974757 0.90000101 ... 0.90000101 1.         0.90000101]
 [0.89974757 0.89974757 0.90000101 ... 0.90000101 0.90000101 1.        ]]
Number of Iterations: 500
Min Eigenvalue: (-3.0531133177191805e-16+0j)
Higham's method output is PSD:
False
```

Note that the output matrix is not PSD, as the minimum eigenvalue is negative. However, the absolute value of this minimum eigenvector is extremely small. If this value was significantly negative as mentioned in the notes, we would run the result on the algorithm again.

```
[[1.         0.74381947 0.88594237 ... 0.88594237 0.88594237 0.88594237]
 [0.74381947 1.         0.88594237 ... 0.88594237 0.88594237 0.88594237]
 [0.88594237 0.88594237 1.         ... 0.90000005 0.90000005 0.90000005]
 ...
 [0.88594237 0.88594237 0.90000005 ... 1.         0.90000005 0.90000005]
 [0.88594237 0.88594237 0.90000005 ... 0.90000005 1.         0.90000005]
 [0.88594237 0.88594237 0.90000005 ... 0.90000005 0.90000005 1.        ]]
near_psd output is PSD:
Min Eigenvalue: (1.0339602438125262e-14+0j)
True
norm = 0.6275226557678678
_____________________________________________________
Reached Max Number of Iterations
[[1.         0.7985871  0.89974757 ... 0.89974757 0.89974757 0.89974757]
 [0.7985871  1.         0.89974757 ... 0.89974757 0.89974757 0.89974757]
 [0.89974757 0.89974757 1.         ... 0.90000101 0.90000101 0.90000101]
 ...
 [0.89974757 0.89974757 0.90000101 ... 1.         0.90000101 0.90000101]
 [0.89974757 0.89974757 0.90000101 ... 0.90000101 1.         0.90000101]
 [0.89974757 0.89974757 0.90000101 ... 0.90000101 0.90000101 1.        ]]
Number of Iterations: 500
Min Eigenvalue: (-3.0531133177191805e-16+0j)
Higham's method output is PSD:
False
norm = 0.0896479963577577
```

In terms of the norm, the near_psd() yields a Frobenius norm of 0.6275, while the Higham's method yields a norm of 0.08975.

However, the runtime for Higham's method is significantly longer than that of near_psd(), especially as N increases.



In fact, the above plot indicates that the runtime for Higham's method increases exponentially as N increases, while the runtime for near_psd() remains low even as N increases. For the list of N values = [5, 10, 50, 100, 200, 300, 400, 500], we obtain the following corresponding runtimes (in seconds) for near_psd() and Higham's method, respectively.

```
['0.00', '0.00', '0.00', '0.00', '0.01', '0.02', '0.04', '0.07']
['0.02', '0.06', '0.68', '1.64', '8.31', '21.92', '45.72', '84.29']
```

As such, the pros and cons of each method are clear. While near_psd() may be less accurate than Higham's method in terms of its Frobenius norm, it always returns a PSD matrix and does so in a short amount of time, regardless of the size of N. On the other hand, Higham's method offers an output that is "closer" to the original matrix, but the output is not guaranteed to be PSD and the method suffers from exponentially increasing runtimes as N increases. As such, I would likely use near_psd() on larger datasets, for which the "closeness" to the original matrix is not of significant importance. On the other hand, if the closeness is indeed important and N is sufficiently small, I would use Higham's method.

Problem 3)

Below is the input matrix obtained using standard correlation matrix and variance vector, and the covariance matrices obtained under direct simulation, PCA with 100%, PCA with 75%, and PCA with 50%. The input matrices and the resulting covariance matrices for each method for the 3 remaining scenarios are available to view in my code.

Input Covariance Matrix

	0	1	2	...	97	98	99
0	0.000061	0.000062	0.000067	...	6.089304e-05	0.000069	0.000040
1	0.000062	0.000200	0.000086	...	2.287037e-05	0.000038	0.000034
2	0.000067	0.000086	0.000157	...	2.491099e-05	0.000029	0.000041
3	0.000096	0.000096	0.000140	...	3.642394e-05	0.000053	0.000058
4	0.000132	0.000186	0.000116	...	1.463149e-04	0.000192	0.000069
..	...	...	...	...	...	...	...
95	0.000002	0.000005	-0.000009	...	6.323148e-07	0.000013	0.000014
96	0.000042	0.000032	0.000040	...	5.856426e-05	0.000039	0.000035
97	0.000061	0.000023	0.000025	...	3.709486e-04	0.000181	0.000047
98	0.000069	0.000038	0.000029	...	1.808218e-04	0.000328	0.000039
99	0.000040	0.000034	0.000041	...	4.730133e-05	0.000039	0.000117

[100 rows x 100 columns]

Direct Sim

	0	1	2	...	97	98	99
0	0.000059	0.000061	0.000065	...	5.852513e-05	0.000068	0.000039
1	0.000061	0.000199	0.000083	...	1.837055e-05	0.000037	0.000032
2	0.000065	0.000083	0.000154	...	2.101145e-05	0.000027	0.000039
3	0.000093	0.000094	0.000137	...	3.014533e-05	0.000050	0.000057
4	0.000129	0.000182	0.000113	...	1.402369e-04	0.000185	0.000065
..	...	...	...	...	...	...	...
95	0.000001	0.000005	-0.000007	...	-1.840636e-07	0.000012	0.000014
96	0.000041	0.000031	0.000039	...	5.845120e-05	0.000038	0.000033
97	0.000059	0.000018	0.000021	...	3.719309e-04	0.000179	0.000047
98	0.000068	0.000037	0.000027	...	1.786541e-04	0.000325	0.000036
99	0.000039	0.000032	0.000039	...	4.716675e-05	0.000036	0.000117

[100 rows x 100 columns]

PCA with 100%

	0	1	2	...	97	98	99
0	0.000055	0.000049	0.000055	...	0.000067	0.000073	0.000033
1	0.000049	0.000043	0.000049	...	0.000059	0.000064	0.000029
2	0.000055	0.000049	0.000055	...	0.000067	0.000073	0.000033
3	0.000085	0.000075	0.000085	...	0.000104	0.000112	0.000051
4	0.000136	0.000120	0.000136	...	0.000166	0.000180	0.000082
..	...	...	...	...	...	...	...
95	-0.000004	-0.000004	-0.000004	...	-0.000005	-0.000005	-0.000002
96	0.000037	0.000033	0.000037	...	0.000045	0.000049	0.000022
97	0.000067	0.000059	0.000067	...	0.000082	0.000089	0.000040
98	0.000073	0.000064	0.000073	...	0.000089	0.000097	0.000044
99	0.000033	0.000029	0.000033	...	0.000040	0.000044	0.000020

[100 rows x 100 columns]

PCA with 75%

	0	1	2	...	97	98	99
0	5.894871e-05	0.000056	0.000064	...	0.000061	0.000070	0.000039
1	5.562643e-05	0.000108	0.000076	...	0.000014	0.000038	0.000036
2	6.366390e-05	0.000076	0.000109	...	0.000021	0.000030	0.000039
3	9.387775e-05	0.000098	0.000140	...	0.000024	0.000056	0.000059
4	1.312646e-04	0.000190	0.000116	...	0.000149	0.000198	0.000069
..	...	...	...	...	...	...	...
95	1.381474e-07	-0.000002	-0.000014	...	0.000005	0.000015	0.000016
96	4.129218e-05	0.000028	0.000043	...	0.000055	0.000030	0.000040
97	6.088547e-05	0.000014	0.000021	...	0.000282	0.000190	0.000041
98	6.993803e-05	0.000038	0.000030	...	0.000190	0.000255	0.000038
99	3.851027e-05	0.000036	0.000039	...	0.000041	0.000038	0.000044

[100 rows x 100 columns]

PCA with 50%

	0	1	2	...	97	98	99
0	0.000059	0.000053	0.000063	...	0.000065	0.000072	0.000038
1	0.000053	0.000068	0.000068	...	0.000024	0.000032	0.000037
2	0.000063	0.000068	0.000097	...	0.000023	0.000028	0.000039
3	0.000095	0.000099	0.000141	...	0.000052	0.000060	0.000056
4	0.000130	0.000199	0.000117	...	0.000142	0.000194	0.000077
..	...	...	...	...	...	...	...
95	-0.000002	-0.000008	-0.000015	...	0.000012	0.000025	0.000006
96	0.000043	0.000043	0.000046	...	0.000039	0.000038	0.000039
97	0.000065	0.000024	0.000023	...	0.000230	0.000204	0.000044
98	0.000072	0.000032	0.000028	...	0.000204	0.000231	0.000044
99	0.000038	0.000037	0.000039	...	0.000044	0.000044	0.000032

[100 rows x 100 columns]

Upon calculating the Frobenius norm and the runtime (in seconds) for each input matrix (labeled Scenarios A through D) and for each method (direct, 100%, 75%, and 50%), we obtain the following results.

Frobenius Norm

	A	B	C	D
Direct	0.000199	0.651067	0.000221	0.000215
100%	0.004391	0.000227	0.000263	0.000217
75%	0.001086	0.001357	0.001609	0.001621
50%	0.002072	0.002654	0.003547	0.003547

Runtime

	A	B	C	D
Direct	1.515878	1.336894	1.259184	1.285785
100%	1.175948	1.302667	1.279193	1.300519
75%	1.204612	1.205484	1.204806	1.189280
50%	1.177627	1.186180	1.214752	1.192677

Note that the Frobenius Norm tends to increase as we move from Direct Simulation to PCA with X% explained. This is as expected, since all three PCA methods drop zero eigenvalues and corresponding eigenvectors (even for 100% explained) and additional eigenvalues and corresponding eigenvectors depending on desired %-explained parameter. As the desired %-explained parameter decreases, we remove additional eigenvalues and eigenvectors. Thus, the fact that 100% PCA has the lowest Frobenius Norm among the three PCA methods, 75% has the second lowest, and 50% has the highest, is as expected.

The fact that the PCA methods tend to have greater Frobenius Norms is compensated by the runtime. As less random standard normal variables need to be generated for the PCA methods (with a lower desired %-explained corresponding to less random variables we must generate), the fact that PCA with 50% explained has the lowest runtime, followed by 75%, 100%, and Direct Simulation is as expected. Thus, we see the tradeoff between accuracy and runtime, with higher accuracy corresponding to longer runtimes and lower accuracy corresponding to shorter runtimes. However, the difference in runtimes is quite small, especially compared to the previous trade-off between near_psd() and Higham's method, which could make

Direct Simulation and PCA 100% more attractive in comparison to PCA 75% and PCA 50% when considering the improved accuracy in general.

However, as seen in Scenario B, Direct Simulation is not always more accurate than the PCA methods. Since the input covariance matrix under Scenario B is not fully PSD (with previous test cases yielding extremely small negative eigenvalues), the PCA methods yield more accuracy in addition to shorter runtimes.