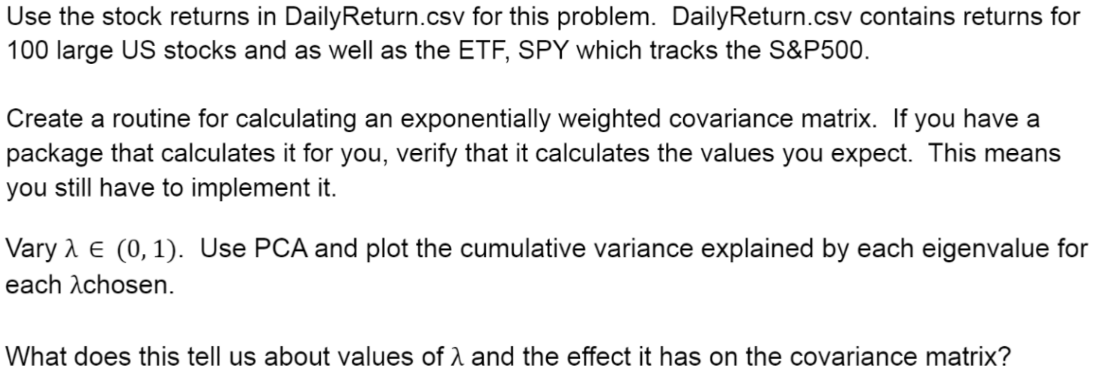
Problem 1



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 λchosen.

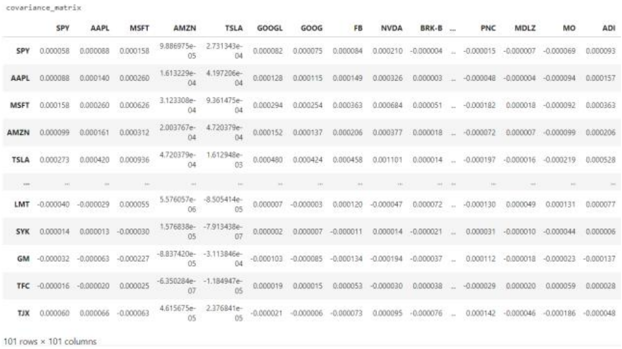
What does this tell us about values of λ and the effect it has on the covariance matrix?

In the codes for this problem, I wrote two functions – one called “populateWeights”, and the other called “exwCovMat”: the first function is similar to the one used as example by professor in class, calculating a list/vector of exponential weights and normalizing them for later use. The other function is where calculations of variances and covariances actually take place; it employs those two equations included in notes for week 3 to compute variances and covariances and constructs a covariance matrix with derived numbers.

After reading data from the provided .csv file, I called those two functions defined in the program to generate a reverse vector of exponential weights (since the order of dates used in the data file is ascending – the first date is the farthest away from now) and an exponentially weighted covariance matrix. Then, I employed eigh(), included in numpy, to compute eigenvalues and eigenvectors of the resulting matrix (there are negative eigenvalues regardless of values of lamda, but they are typically in the range of (-1e-8, 0), so I assume that they are zero in this case for convenience), and I summed all eigenvalues and stored the result in a variable called e_sum. Followingly, I removed all eigenvalues that are positive but smaller than 1e-8 to reduce dimensionalities via PCA, and started computing cumulative percent of variance that could explained by each eigenvalue and by first K eigenvalues, which were eventually graphed into two separate plots. Results and graphs are shown below with three different lamdas:

a. Lamda = 0.5

Exponentially Weighted Covariance Matrix (part of the matrix due to too many items):

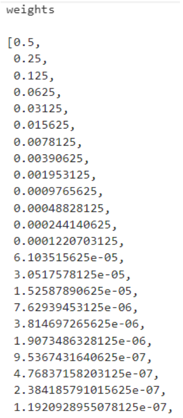


covariance_matrix

	SPY	AAPL	MSFT	AMZN	TSLA	GOOGL	GOOG	FB	NVDA	BRK-B	...	PNC	MDLZ	MO	ADI
SPY	0.000058	0.000088	0.000158	9.886975e-05	2.731343e-04	0.000082	0.000075	0.000084	0.000210	-0.000004	...	-0.000015	-0.000007	-0.000069	0.000093
AAPL	0.000088	0.000140	0.000260	1.613229e-04	4.197206e-04	0.000128	0.000115	0.000149	0.000326	0.000003	...	-0.000048	-0.000004	-0.000094	0.000157
MSFT	0.000158	0.000260	0.000626	3.123308e-04	9.361475e-04	0.000294	0.000254	0.000363	0.000684	0.000051	...	-0.000182	0.000018	-0.000092	0.000363
AMZN	0.000099	0.000161	0.000312	2.003767e-04	4.720379e-04	0.000152	0.000137	0.000206	0.000377	0.000018	...	-0.000072	0.000007	-0.000099	0.000206
TSLA	0.000273	0.000420	0.000936	4.720379e-04	1.612948e-03	0.000480	0.000424	0.000458	0.001101	0.000014	...	-0.000197	-0.000016	-0.000219	0.000528
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
LMT	-0.000040	-0.000029	0.000055	5.576057e-06	-8.505414e-05	0.000007	-0.000003	0.000120	-0.000047	0.000072	...	-0.000130	0.000049	0.000131	0.000077
SYK	0.000014	0.000013	-0.000030	1.576838e-05	-7.913438e-07	0.000002	0.000007	-0.000011	0.000014	-0.000021	...	0.000031	-0.000010	-0.000044	0.000006
GM	-0.000032	-0.000063	-0.000227	-8.837420e-05	-3.113846e-04	-0.000103	-0.000085	-0.000134	-0.000194	-0.000037	...	0.000112	-0.000018	-0.000023	-0.000137
TFC	-0.000016	-0.000020	0.000025	-6.350284e-07	-1.184947e-05	0.000019	0.000015	0.000053	-0.000030	0.000038	...	-0.000029	0.000020	0.000059	0.000028
TJX	0.000060	0.000066	-0.000063	4.615675e-05	2.376841e-05	-0.000021	-0.000006	-0.000073	0.000095	-0.000076	...	0.000142	-0.000046	-0.000186	-0.000048

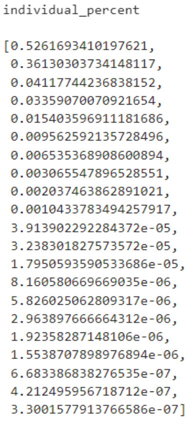
101 rows × 101 columns

Vector of Exponential Weights (from nearest period to farthest period) (part of the vector due to too many items):



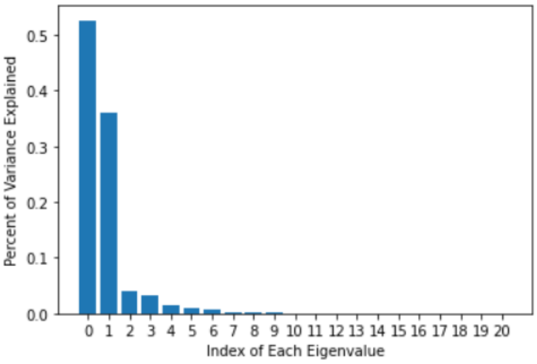
```
weights

[0.5,
 0.25,
 0.125,
 0.0625,
 0.03125,
 0.015625,
 0.0078125,
 0.00390625,
 0.001953125,
 0.0009765625,
 0.00048828125,
 0.000244140625,
 0.0001220703125,
 6.103515625e-05,
 3.0517578125e-05,
 1.52587890625e-05,
 7.62939453125e-06,
 3.814697265625e-06,
 1.9073486328125e-06,
 9.5367431640625e-07,
 4.76837158203125e-07,
 2.384185791015625e-07,
 1.1920928955078125e-07,
```

Percent of Variance Explained by Each Eigenvalue (that is greater than 1e-8):

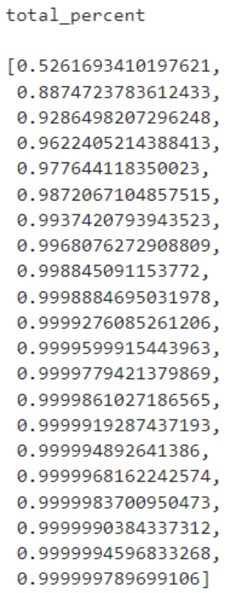


```
individual_percent

[0.5261693410197621,
 0.36130303734148117,
 0.04117744236838152,
 0.03359070070921654,
 0.015403596911181686,
 0.009562592135728496,
 0.006535368908600894,
 0.003065547896528551,
 0.002037463862891021,
 0.0010433783494257917,
 3.913902292284372e-05,
 3.238301827573572e-05,
 1.7950593590533686e-05,
 8.160580669669035e-06,
 5.826025062809317e-06,
 2.963897666664312e-06,
 1.92358287148106e-06,
 1.5538707898976894e-06,
 6.683386838276535e-07,
 4.212495956718712e-07,
 3.3001577913766586e-07]
```

Graph of Percent of Variance Explained by Each Eigenvalue (that is greater than 1e-8):

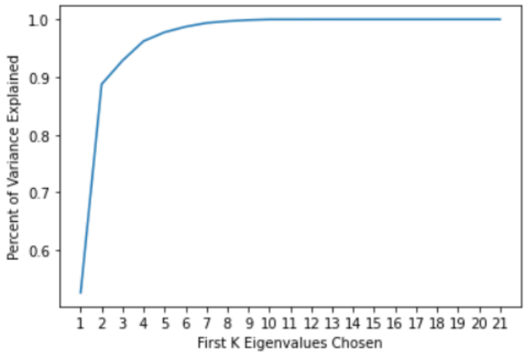


Percent of Variance Explained by First 1 to K Eigenvalues (that is greater than 1e-8):

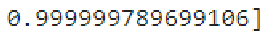


```
total_percent

[0.5261693410197621,
 0.8874723783612433,
 0.9286498207296248,
 0.9622405214388413,
 0.977644118350023,
 0.9872067104857515,
 0.9937420793943523,
 0.9968076272908809,
 0.998845091153772,
 0.9998884695031978,
 0.9999276085261206,
 0.9999599915443963,
 0.9999779421379869,
 0.9999861027186565,
 0.9999919287437193,
 0.999994892641386,
 0.9999968162242574,
 0.9999983700950473,
 0.9999990384337312,
 0.9999994596833268,
 0.999999789699106]
```

Graph of Percent of Variance Explained by First 1 to K Eigenvalues (that is greater than 1e-8):



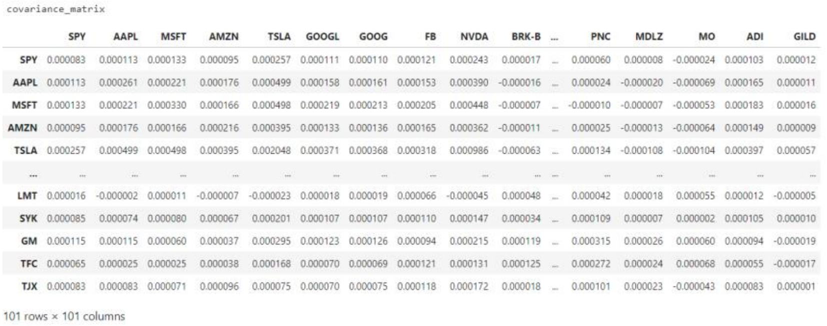
Total Percent of Variance Explained by All Eigenvalues Chosen:



```
0.999999789699106]
```

b. Lamda = 0.94

Exponentially Weighted Covariance Matrix (part of the matrix due to too many items):

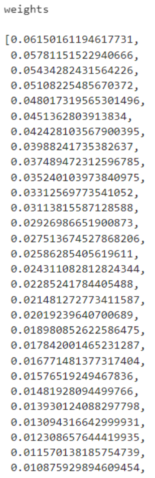


```
covariance_matrix
```

	SPY	AAPL	MSFT	AMZN	TSLA	GOOGL	GOOG	FB	NVDA	BRK-B	...	PNC	MDLZ	MO	ADI	GILD
SPY	0.000083	0.000113	0.000133	0.000095	0.000257	0.000111	0.000110	0.000121	0.000243	0.000017	...	0.000060	0.000008	-0.000024	0.000103	0.000012
AAPL	0.000113	0.000261	0.000221	0.000176	0.000499	0.000158	0.000161	0.000153	0.000390	-0.000016	...	0.000024	-0.000020	-0.000069	0.000165	0.000011
MSFT	0.000133	0.000221	0.000330	0.000166	0.000498	0.000219	0.000213	0.000205	0.000448	-0.000007	...	-0.000010	-0.000007	-0.000053	0.000183	0.000016
AMZN	0.000095	0.000176	0.000166	0.000216	0.000395	0.000133	0.000136	0.000165	0.000362	-0.000011	...	0.000025	-0.000013	-0.000064	0.000149	0.000009
TSLA	0.000257	0.000499	0.000498	0.000395	0.002048	0.000371	0.000368	0.000318	0.000986	-0.000063	...	0.000134	-0.000108	-0.000104	0.000397	0.000057
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
LMT	0.000016	-0.000002	0.000011	-0.000007	-0.000023	0.000018	0.000019	0.000066	-0.000045	0.000048	...	0.000042	0.000018	0.000055	0.000012	-0.000005
SYK	0.000085	0.000074	0.000080	0.000067	0.000201	0.000107	0.000107	0.000110	0.000147	0.000034	...	0.000109	0.000007	0.000002	0.000105	0.000010
GM	0.000115	0.000115	0.000060	0.000037	0.000295	0.000123	0.000126	0.000094	0.000215	0.000119	...	0.000315	0.000026	0.000060	0.000094	-0.000019
TFC	0.000065	0.000025	0.000025	0.000038	0.000168	0.000070	0.000069	0.000121	0.000131	0.000125	...	0.000272	0.000024	0.000068	0.000055	-0.000017
TJX	0.000083	0.000083	0.000071	0.000096	0.000075	0.000070	0.000075	0.000118	0.000172	0.000018	...	0.000101	0.000023	-0.000043	0.000083	0.000001

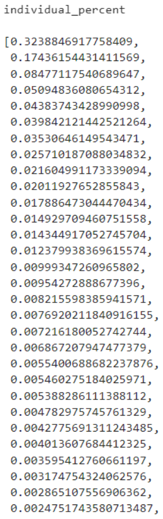
101 rows × 101 columns

Vector of Exponential Weights (from nearest period to farthest period) (part of the vector due to too many items):



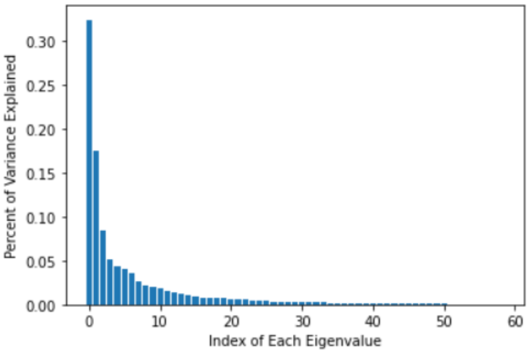
```
weights

[0.06150161194617731,
 0.05781151522940666,
 0.05434282431564226,
 0.05108225485670372,
 0.048017319565301496,
 0.0451362803913834,
 0.042428103567900395,
 0.03988241735382637,
 0.037489472312596785,
 0.035240103973840975,
 0.03312569773541052,
 0.03113815587128588,
 0.02926986651900873,
 0.027513674527868206,
 0.02586285405619611,
 0.024311082812824344,
 0.02285241784405488,
 0.021481272773411587,
 0.02019239640700689,
 0.018980852622586475,
 0.017842001465231287,
 0.016771481377317404,
 0.01576519249467836,
 0.01481928094499766,
 0.013930124088297798,
 0.013094316642999931,
 0.012308657644419935,
 0.011570138185754739,
 0.010875929894609454,
```

Percent of Variance Explained by Each Eigenvalue (that is greater than 1e-8) (part of the vector due to too many items):

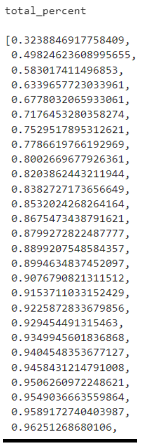


```
individual_percent

[0.3238846917758409,
 0.17436154431411569,
 0.08477117540689647,
 0.05094836080654312,
 0.04383743428990998,
 0.039842121442521264,
 0.03530646149543471,
 0.025710187088034832,
 0.021604991173339094,
 0.02011927652855843,
 0.017886473044470434,
 0.014929709460751558,
 0.014344917052745704,
 0.012379938369615574,
 0.00999347260965802,
 0.00954272888677396,
 0.008215598385941571,
 0.0076920211840916155,
 0.007216180052742744,
 0.006867207947477379,
 0.0055400688682237876,
 0.005460275184025971,
 0.005388286111388112,
 0.004782975745761329,
 0.0042775691311243485,
 0.004013607684412325,
 0.003595412760661197,
 0.003174754324062576,
 0.002865107556906362,
 0.0024751743580713487,
```

Graph of Percent of Variance Explained by Each Eigenvalue (that is greater than 1e-8):

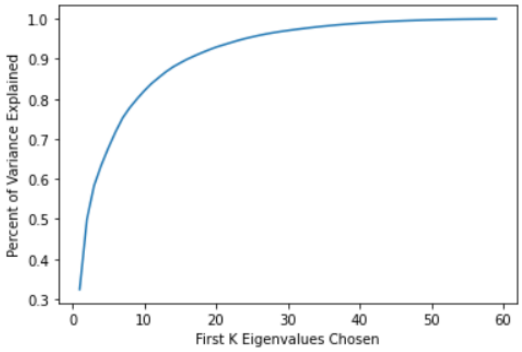


Percent of Variance Explained by First 1 to K Eigenvalues (that is greater than 1e-8) (part of the vector due to too many items):

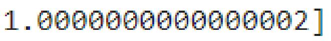


```
total_percent

[0.3238846917758409,
 0.49824623608995655,
 0.583017411496853,
 0.6339657723033961,
 0.6778032065933061,
 0.7176453280358274,
 0.7529517895312621,
 0.7786619766192969,
 0.8002669677926361,
 0.8203862443211944,
 0.838272717365649,
 0.8532024268264164,
 0.8675473438791621,
 0.8799272822487777,
 0.8899207548584357,
 0.8994634837452097,
 0.9076790821311512,
 0.9153711033152429,
 0.9225872833679856,
 0.929454491315463,
 0.9349945601836868,
 0.9404548353677127,
 0.9458431214791008,
 0.9506260972248621,
 0.9549036663559864,
 0.9589172740403987,
 0.96251268680106,
```

Graph of Percent of Variance Explained by First 1 to K Eigenvalues (that is greater than 1e-8):



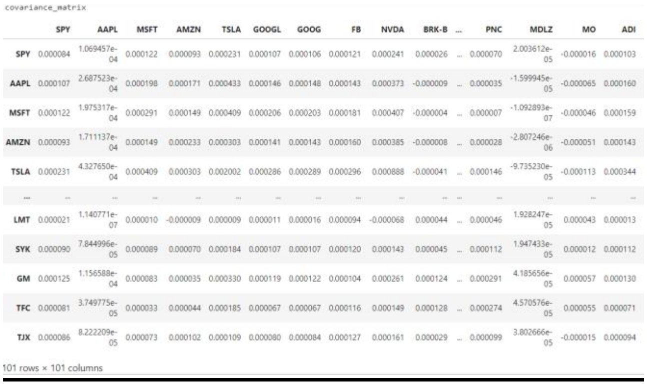
Total Percent of Variance Explained by All Eigenvalues Chosen:



```
1.0000000000000002]
```

c. **Lamda = 0.97**

Exponentially Weighted Covariance Matrix (part of the matrix due to too many items):

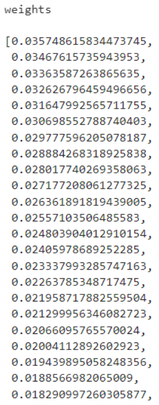


covariance_matrix

	SPY	AAPL	MSFT	AMZN	TSLA	GOOGL	GOOG	FB	NVDA	BRK-B	...	PNC	MDLZ	MO	ADI
SPY	0.000084	1.069457e-04	0.000122	0.000093	0.000231	0.000107	0.000106	0.000121	0.000241	0.000026	...	0.000070	2.003612e-05	-0.000016	0.000103
AAPL	0.000107	2.687523e-04	0.000198	0.000171	0.000433	0.000146	0.000148	0.000143	0.000373	-0.000009	...	0.000035	-1.599945e-05	-0.000065	0.000160
MSFT	0.000122	1.975317e-04	0.000291	0.000149	0.000409	0.000206	0.000203	0.000181	0.000407	-0.000004	...	0.000007	-1.092893e-07	-0.000046	0.000159
AMZN	0.000093	1.711137e-04	0.000149	0.000233	0.000303	0.000141	0.000143	0.000160	0.000385	-0.000008	...	0.000028	-2.807246e-06	-0.000051	0.000143
TSLA	0.000231	4.327650e-04	0.000409	0.000303	0.002002	0.000286	0.000289	0.000296	0.000888	-0.000041	...	0.000146	-9.735230e-05	-0.000113	0.000344
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
LMT	0.000021	1.140771e-07	0.000010	-0.000009	0.000009	0.000011	0.000016	0.000094	-0.000068	0.000044	...	0.000046	1.928247e-05	0.000043	0.000013
SYK	0.000090	7.844996e-05	0.000089	0.000070	0.000184	0.000107	0.000107	0.000120	0.000143	0.000045	...	0.000112	1.947433e-05	0.000012	0.000112
GM	0.000125	1.156588e-04	0.000083	0.000035	0.000330	0.000119	0.000122	0.000104	0.000261	0.000124	...	0.000291	4.185656e-05	0.000057	0.000130
TFC	0.000081	3.749775e-05	0.000033	0.000044	0.000185	0.000067	0.000067	0.000116	0.000149	0.000128	...	0.000274	4.570576e-05	0.000055	0.000071
TJX	0.000086	8.222209e-05	0.000073	0.000102	0.000109	0.000080	0.000084	0.000127	0.000161	0.000029	...	0.000099	3.802666e-05	-0.000015	0.000094

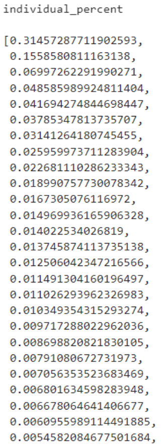
101 rows × 101 columns

Vector of Exponential Weights (from nearest period to farthest period) (part of the vector due to too many items):



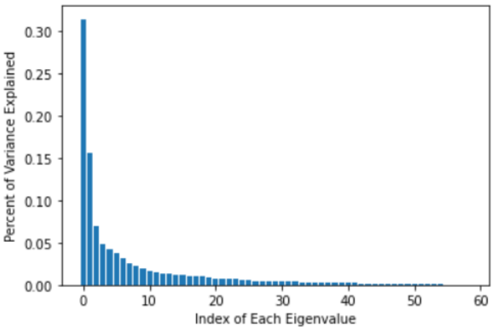
```
weights

[0.035748615834473745,
 0.03467615735943953,
 0.03363587263865635,
 0.032626796459496656,
 0.031647992565711755,
 0.030698552788740403,
 0.029777596205078187,
 0.028884268318925838,
 0.028017740269358063,
 0.027177208061277325,
 0.026361891819439005,
 0.02557103506485583,
 0.024803904012910154,
 0.02405978689252285,
 0.023337993285747163,
 0.02263785348717475,
 0.021958717882559504,
 0.021299956346082723,
 0.02066095765570024,
 0.02004112892602923,
 0.019439895058248356,
 0.0188566982065009,
 0.018290997260305877,
```

Percent of Variance Explained by Each Eigenvalue (that is greater than 1e-8) (part of the matrix due to too many items):

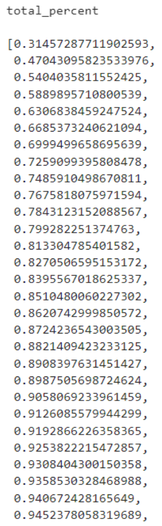


```
individual_percent

[0.31457287711902593,
 0.1558580811163138,
 0.06997262291990271,
 0.048585989924811404,
 0.041694274844698447,
 0.03785347813735707,
 0.03141264180745455,
 0.025959973711283904,
 0.022681110286233343,
 0.018990757730078342,
 0.0167305076116972,
 0.014969936165906328,
 0.014022534026819,
 0.013745874113735138,
 0.012506042347216566,
 0.011491304160196497,
 0.011026293962326983,
 0.010349354315293274,
 0.009717288022962036,
 0.008698820821830105,
 0.00791080672731973,
 0.007056353523683469,
 0.006801634598283948,
 0.006678064641406677,
 0.0060955989114491885,
 0.0054582084677501684,
```

Graph of Percent of Variance Explained by Each Eigenvalue (that is greater than 1e-8):

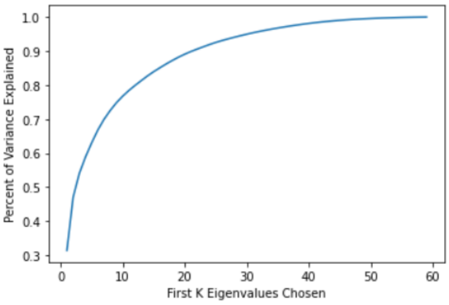


Percent of Variance Explained by First 1 to K Eigenvalues (that is greater than 1e-8) (part of the matrix due to too many items):

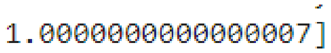


```
total_percent

[0.31457287711902593,
 0.47043095823533976,
 0.5404035811552425,
 0.5889895710800539,
 0.6306838459247524,
 0.6685373240621094,
 0.6999499658695639,
 0.7259099395808478,
 0.7485910498670811,
 0.7675818075971594,
 0.7843123152088567,
 0.799282251374763,
 0.813304785401582,
 0.8270506595153172,
 0.8395567018625337,
 0.8510480060227302,
 0.8620742999850572,
 0.8724236543003505,
 0.8821409423233125,
 0.8908397631451427,
 0.8987505698724624,
 0.9058069233961459,
 0.9126085579944299,
 0.9192866226358365,
 0.9253822215472857,
 0.9308404300150358,
 0.9358530328468988,
 0.940672428165649,
 0.9452378058319689,
```

Graph of Percent of Variance Explained by First 1 to K Eigenvalues (that is greater than 1e-8):



Total Percent of Variance Explained by All Eigenvalues Chosen:

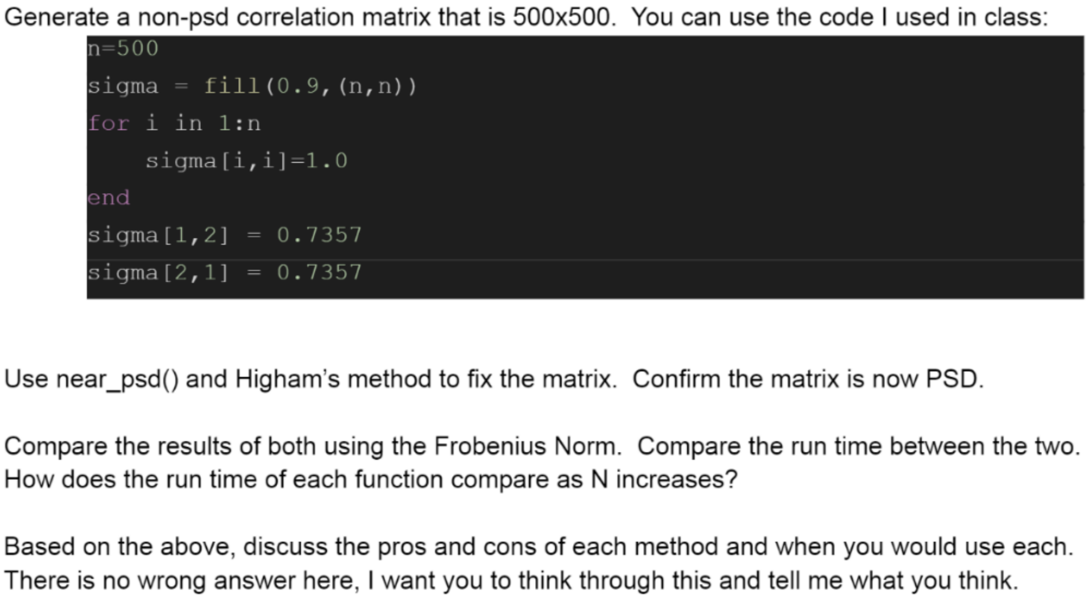


```
1.0000000000000007]
```

Conclusion

From results above, we could notice that as the larger lamda is, the smaller weight of nearest data is; in other words, the higher lamda is, the more data we would take into account in our analysis, which represents slower decays in weights. Comparing to lower ones, higher lamdas are expected to explain a higher percent of variance. For example, when lamda = 0.5, we could explain approximately 99.99997897% of variance after conducting a PCA, while nearly 100.00000000% of variance could be explained when lamda = 0.97 (note that there could be floating point errors or some other errors leading a percentage that is greater than 1 when lamda = 0.97). What's more, when the smoothing parameter, lamda, gets larger, the graph of percent of variance explained by first 1 to K eigenvalues becomes smoother. Upon lamda's effects on covariance matrices, as lamda increases (the optimal value derived by J.P. Morgan is about 0.97), it gives a more reliable forecast of the variance that comes closest to realized variance rate. To be blatant, as lamda increases, covariance matrices it generated become more accurate and similar as realized.

Problem 2



Generate a non-psd correlation matrix that is 500x500. You can use the code I used in class:

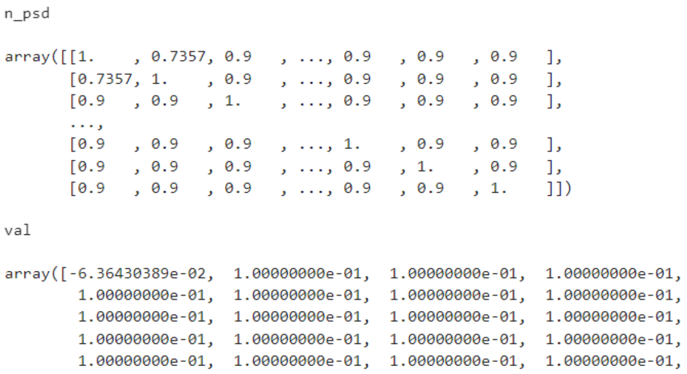
```
n=500
sigma = fill(0.9,(n,n))
for i in 1:n
    sigma[i,i]=1.0
end
sigma[1,2] = 0.7357
sigma[2,1] = 0.7357
```

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.

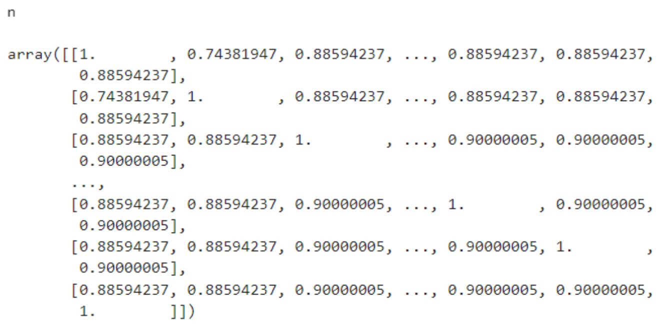
I wrote two main functions (there are two other functions defined at beginning for Cholesky decomposition, but they are not used in this problem) in my program – one called near_PSD(), the other called Higham() – both are methods used to derive nearest PSD correlation matrices. I first set up a 500 * 500 matrix in the way professor introduced in class, and it is not positive semi definite, since not all of its eigenvalues are nonnegative, as shown in the screenshot below:



```
n_psd

array([[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.    ]])

val

array([-6.36430389e-02,  1.00000000e-01,  1.00000000e-01,  1.00000000e-01,
        1.00000000e-01,  1.00000000e-01,  1.00000000e-01,  1.00000000e-01,
        1.00000000e-01,  1.00000000e-01,  1.00000000e-01,  1.00000000e-01,
        1.00000000e-01,  1.00000000e-01,  1.00000000e-01,  1.00000000e-01,
        1.00000000e-01,  1.00000000e-01,  1.00000000e-01,  1.00000000e-01,
```

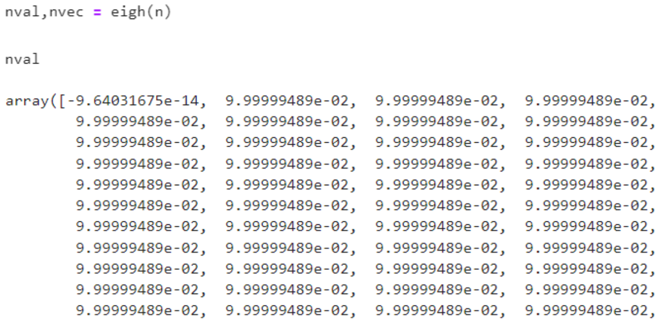
near_PSD()

Then, I called the near_PSD() function to compute the nearest PSD correlation matrix of the 500 * 500 matrix, and the resulting matrix, which is quite close to the original matrix, is:



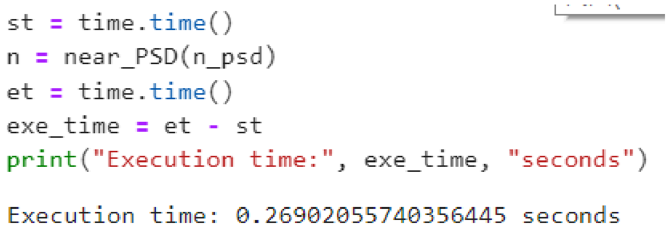
```
n

array([[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.        ]])
```

Checking eigenvalues of this matrix generated from near_PSD() function, we have:



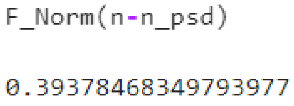
```
nval,nvec = eigh(n)

nval

array([-9.64031675e-14,  9.99999489e-02,  9.99999489e-02,  9.99999489e-02,
        9.99999489e-02,  9.99999489e-02,  9.99999489e-02,  9.99999489e-02,
        9.99999489e-02,  9.99999489e-02,  9.99999489e-02,  9.99999489e-02,
        9.99999489e-02,  9.99999489e-02,  9.99999489e-02,  9.99999489e-02,
        9.99999489e-02,  9.99999489e-02,  9.99999489e-02,  9.99999489e-02,
        9.99999489e-02,  9.99999489e-02,  9.99999489e-02,  9.99999489e-02,
        9.99999489e-02,  9.99999489e-02,  9.99999489e-02,  9.99999489e-02,
        9.99999489e-02,  9.99999489e-02,  9.99999489e-02,  9.99999489e-02,
        9.99999489e-02,  9.99999489e-02,  9.99999489e-02,  9.99999489e-02,
        9.99999489e-02,  9.99999489e-02,  9.99999489e-02,  9.99999489e-02,
        9.99999489e-02,  9.99999489e-02,  9.99999489e-02,  9.99999489e-02,
```

As shown above, the smallest eigenvalue is about -9e-14, which is larger than -1e-8 (the threshold I set for “effectively zero” is that the number’s absolute value must be smaller than 1e-8), so I see it as zero, making the resulting matrix a PSD correlation matrix (**note**: there are other ways to test if matrices are positive semi-definite or not; for example, there are functions defined in some packages for testing positive semi-definite; for another example, we could run matrices in Cholesky algorithm – if outputs are generated, input matrices are positive semi definite; however, these methods are not applicable here - resulting matrices may have negative eigenvalues (effectively zero ones) that would not pass any other tests – so I am determining through looking at its eigenvalues “manually”). I also recorded the time used to execute the near_PSD() function, and the result is about 0.269 seconds, as the screenshot shows below:



```
st = time.time()
n = near_PSD(n_psd)
et = time.time()
exe_time = et - st
print("Execution time:", exe_time, "seconds")

Execution time: 0.26902055740356445 seconds
```

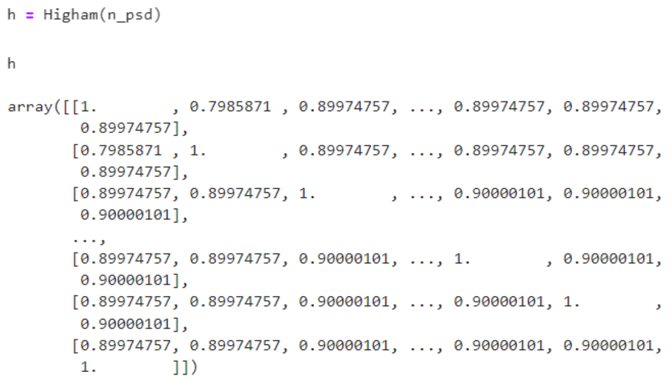
Besides with two main functions for calculations of nearest correlation matrices, I also define another function called F_Norm() to compute Frobenius Norm of a given matrix. I called this function upon the difference between resulting PSD correlation matrix from near_PSD() and original matrix, and the result is:



```
F_Norm(n-n_psd)

0.39378468349793977
```

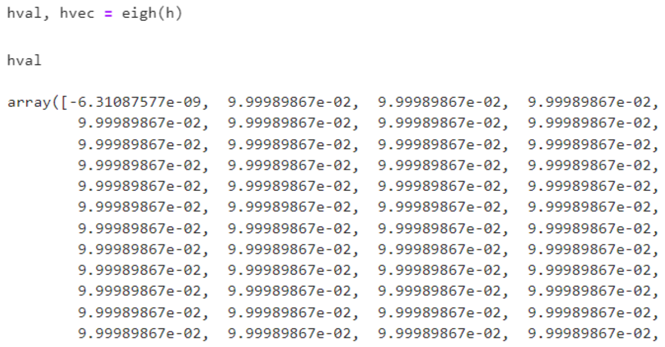
Higham()

After finishing implementing the near_PSD method, I starts executing the Higham method. I called Higham() function upon the original matrix, resulting in the matrix as shown below:



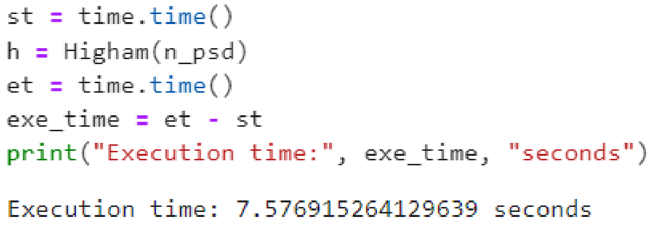
```
h = Higham(n_psd)

h

array([[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.        ]])
```

Similar to the result derived using near_PSD method, Higham method produces a matrix that is close to the original one. I further checked upon eigenvalues of the resulting matrix from Higham method to see if it is a PSD or not:



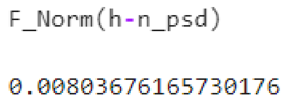
```
hval, hvec = eigh(h)

hval

array([-6.31087577e-09,  9.99989867e-02,  9.99989867e-02,  9.99989867e-02,
        9.99989867e-02,  9.99989867e-02,  9.99989867e-02,  9.99989867e-02,
        9.99989867e-02,  9.99989867e-02,  9.99989867e-02,  9.99989867e-02,
        9.99989867e-02,  9.99989867e-02,  9.99989867e-02,  9.99989867e-02,
        9.99989867e-02,  9.99989867e-02,  9.99989867e-02,  9.99989867e-02,
        9.99989867e-02,  9.99989867e-02,  9.99989867e-02,  9.99989867e-02,
        9.99989867e-02,  9.99989867e-02,  9.99989867e-02,  9.99989867e-02,
        9.99989867e-02,  9.99989867e-02,  9.99989867e-02,  9.99989867e-02,
        9.99989867e-02,  9.99989867e-02,  9.99989867e-02,  9.99989867e-02,
        9.99989867e-02,  9.99989867e-02,  9.99989867e-02,  9.99989867e-02,
        9.99989867e-02,  9.99989867e-02,  9.99989867e-02,  9.99989867e-02,
        9.99989867e-02,  9.99989867e-02,  9.99989867e-02,  9.99989867e-02,
```

As shown above in the screenshot, the smallest eigenvalue is about -6e-9, which is larger than -1e-8 (the threshold I set for "effectively zero" is that the number's absolute value must be smaller than 1e-8), so I see it as zero, making the resulting matrix a PSD correlation matrix (**note**: there are other ways to test if matrices are positive semi-definite or not; for example, there are functions defined in some packages for testing positive semi-definite; for another example, we could run matrices in Cholesky algorithm – if outputs are generated, input matrices are positive semi definite; however, these methods are not applicable here - resulting matrices may have negative eigenvalues (effectively zero ones) that would not pass any other tests – so I am determining through looking at its eigenvalues "manually"). I also recorded the time used to execute the Higham() function, and the result is about 7.577 seconds, which is way longer than the time near_PSD() takes, as the screenshot shows below:



```
st = time.time()
h = Higham(n_psd)
et = time.time()
exe_time = et - st
print("Execution time:", exe_time, "seconds")

Execution time: 7.576915264129639 seconds
```

Eventually, I ran F_Norm() function over the difference between resulting matrix from Higham() function and the original function, and the result is:



```
F_Norm(h-n_psd)

0.00803676165730176
```

Comparison of Frobenius Norm and Run Time between Two Methods

Upon Frobenius norm, as listed above, when employing near_PSD method, Frobenius norm of the difference between the nearest PSD matrix and the original matrix is about 0.39378, while Frobenius norm, using Higham method, of the difference is about 0.00804. Two Frobenius norms are quite different, showing that Higham method generates a more accurate and reliable result that is closer to the original matrix.

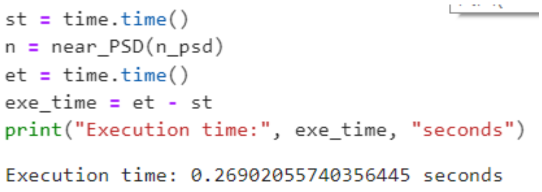
Upon run time, the execution time for near_PSD method is about 0.269 seconds, while it takes about 7.577 seconds to run Higham method, demonstrating that near_PSD method is more timesaving, regardless of quality of its outputs.

Relationship between N and Run Time & Relationship between N and Frob. Norm

Results above are generated when N = 500, and I am going to check run time and Frobenius norms of two methods when N = 1000, 5000 in the following part.

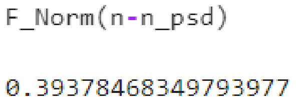
N = 500

Run time for near_PSD():



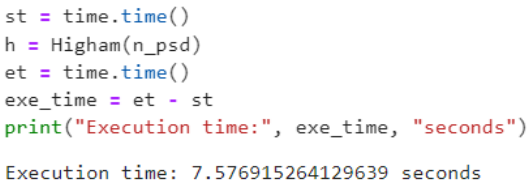
```
st = time.time()
n = near_PSD(n_psd)
et = time.time()
exe_time = et - st
print("Execution time:", exe_time, "seconds")

Execution time: 0.26902055740356445 seconds
```

Frobenius norm for near_PSD():



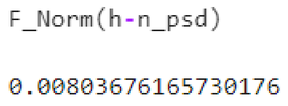
```
F_Norm(n-n_psd)

0.393784683497939777
```

Run time for Higham():



```
st = time.time()
h = Higham(n_psd)
et = time.time()
exe_time = et - st
print("Execution time:", exe_time, "seconds")

Execution time: 7.576915264129639 seconds
```

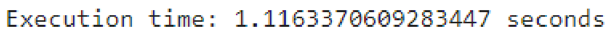
Frobenius norm for Higham():



```
F_Norm(h-n_psd)

0.00803676165730176
```

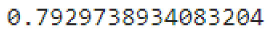
N = 1000

Run time for near_PSD():



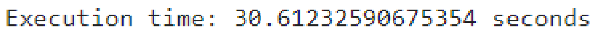
```
Execution time: 1.1163370609283447 seconds
```

Frobenius norm for near_PSD():



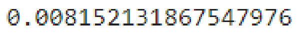
```
0.7929738934083204
```

Run time for Higham():



```
Execution time: 30.61232590675354 seconds
```

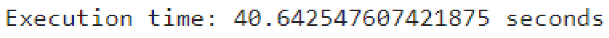
Frobenius norm for Higham():



```
0.008152131867547976
```

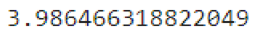
N = 5000

Run time for near_PSD():



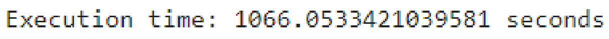
```
Execution time: 40.642547607421875 seconds
```

Frobenius norm for near_PSD():



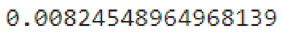
```
3.986466318822049
```

Run time for Higham():



```
Execution time: 1066.0533421039581 seconds
```

Frobenius norm for Higham():



```
0.00824548964968139
```

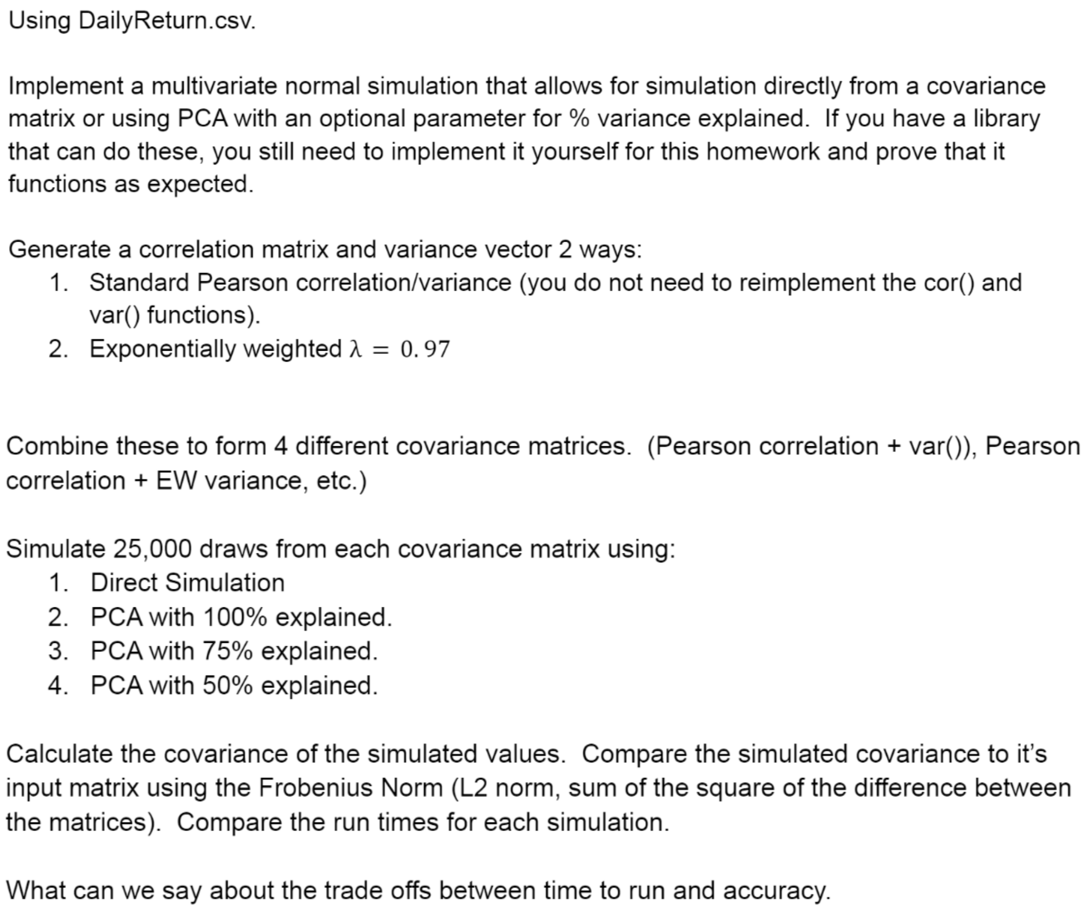
From results generated and shown above, as N increases, run time of both methods increase dramatically, and the execution time of Higham method is always larger than that of near_PSD method, especially when N is large, e.g. N = 5000. For another thing, as N increases, Frobenius norms of both methods also increase relatively. Increases in distances from the original matrix are more obvious while using near_PSD method – as N increases from 500 to 5000, the Frobenius norm increases from 0.39378 to 3.98647. Higham method follows the same routine that norms increase with N, but its norms increase at a much slower rate. This result demonstrates Higham method generates results that are more and more accurate as N increases.

Conclusion

Each method has respective pros and cons. For near_PSD method, it is relatively easier to implement (code-writing part), and it is able to produce relatively reliable and accurate results with a little time. However, its accuracy and reliability decrease as original matrices have more items (N gets larger). For Higham method, it produces more accurate and reliable results, regardless of how large the original matrix is, while it usually takes longer to execute, especially when N is large, and it is difficult to implement under certain circumstances, like we are not setting weights to be identity matrix.

When we have a matrix that contains only a few items (N is small) that needs to be “transformed” into a positive semi-definite matrix, we could make use of both methods, since accuracy and speed are similar between them. If we are still pursuing accuracy of our result, we could take advantage of Higham method which would not take a lot more time than near_PSD method would. However, when we are dealing with matrices that have many rows and columns (N is big), we need to decide what to sacrifice. If we are pursuing speed, then we need to sacrifice accuracy through employing near_PSD method. Otherwise, we may spend a lot more time in computing nearest PSD matrices with Higham method.

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 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.

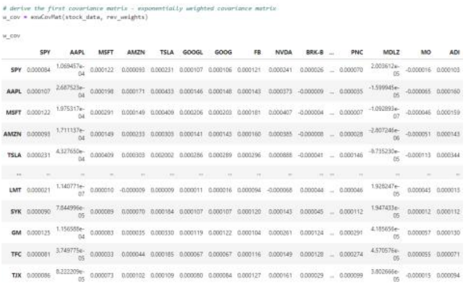
What can we say about the trade offs between time to run and accuracy.

For this problem, I copied codes I wrote for problem 1 & 2 and modified on them to meet my needs. As you may see, there are in total six functions defined in the program wrote for problem 3: "populateWeights" is used to calculate exponential weights; "exwCovMat" is to calculate the exponentially weighted covariance matrix; "exwVarMat" is to calculate the exponentially weighted variance matrix; "simulate_PCA" is to use PCA to simulate the system; "direct_simulate" is a function used to directly simulate a system without doing any other actions, such as conducting a PCA; "F_Norm" is used to calculate Frobenius norm of a given matrix.

After defining all functions that I may use, I start to set up things, primarily computing the list of exponential weights, that I would use later.

1. First Covariance Matrix

Then, I called "exwCovMat" function to derive the first covariance matrix – an exponentially weighted covariance matrix with lamda being 0.97 (part of the matrix due to too many items):



```
# derive the first covariance matrix - exponentially weighted covariance matrix
w_cov = ewCovMat(stock_data, rev_weights)
```

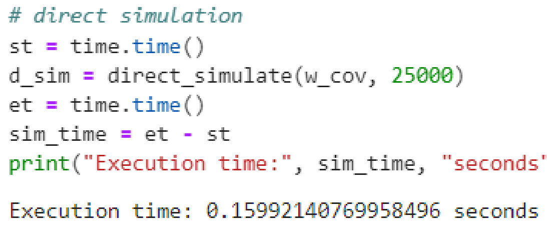
```
w_cov
```

	SPY	AAPL	MSFT	AMZN	TSLA	GOOGL	GOOG	FB	NVDA	BRK-B	...	PNC	MDLZ	MO	ADI
SPY	0.000084	1.069457e-04	0.000122	0.000093	0.000231	0.000107	0.000106	0.000121	0.000241	0.000026	...	0.000070	2.003612e-05	-0.000016	0.000103
AAPL	0.000107	2.687523e-04	0.000198	0.000171	0.000433	0.000146	0.000148	0.000143	0.000373	-0.000009	...	0.000035	-1.599945e-05	-0.000065	0.000160
MSFT	0.000122	1.975317e-04	0.000291	0.000149	0.000409	0.000206	0.000203	0.000181	0.000407	-0.000004	...	0.000007	-1.092893e-07	-0.000046	0.000159
AMZN	0.000093	1.711137e-04	0.000149	0.000233	0.000303	0.000141	0.000143	0.000160	0.000385	-0.000008	...	0.000028	-2.807246e-06	-0.000051	0.000143
TSLA	0.000231	4.327650e-04	0.000409	0.000303	0.002002	0.000286	0.000289	0.000296	0.000888	-0.000041	...	0.000146	-8.735230e-05	-0.000113	0.000344
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
LMT	0.000021	1.140771e-07	0.000010	-0.000009	0.000009	0.000011	0.000016	0.000094	-0.000068	0.000044	...	0.000046	1.928247e-05	0.000043	0.000015
SYK	0.000090	7.844996e-05	0.000089	0.000070	0.000184	0.000107	0.000107	0.000120	0.000143	0.000045	...	0.000112	1.947433e-05	0.000012	0.000112
GM	0.000125	1.156588e-04	0.000083	0.000035	0.000330	0.000119	0.000122	0.000104	0.000261	0.000124	...	0.000291	4.185658e-05	0.000057	0.000130
TFC	0.000081	3.749775e-05	0.000033	0.000044	0.000185	0.000067	0.000067	0.000116	0.000149	0.000128	...	0.000274	4.570576e-05	0.000055	0.000071
TJX	0.000086	8.222209e-05	0.000073	0.000102	0.000109	0.000080	0.000084	0.000127	0.000161	0.000029	...	0.000099	3.802666e-05	-0.000015	0.000094

With the first covariance matrix, I start running simulations over the matrix:

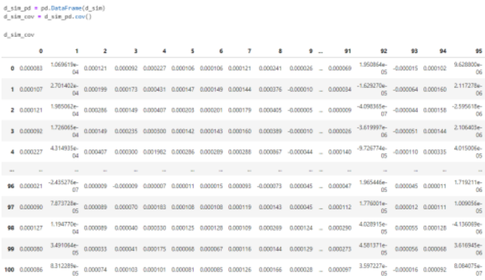
a. Direct Simulation

I called "direct_simulate" function with number of draws set to be 25,000, and timed this simulation process to check runtime, the runtime for direct simulation over exponentially weighted covariance matrix is:



```
# direct simulation
st = time.time()
d_sim = direct_simulate(w_cov, 25000)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")

Execution time: 0.15992140769958496 seconds
```

Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):

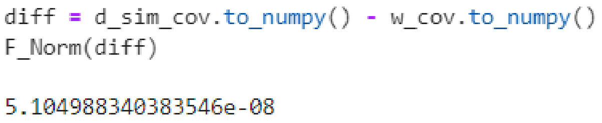


```
d_sim_pd = pd.DataFrame(d_sim)
d_sim_cov = d_sim_pd.cov()
```

```
d_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95
0	0.000083	1.069619e-04	0.000121	0.000092	0.000227	0.000106	0.000106	0.000121	0.000241	0.000026	...	0.000069	1.950864e-05	-0.000015	0.000102	9.628800e-06
1	0.000107	2.701402e-04	0.000199	0.000173	0.000431	0.000147	0.000149	0.000144	0.000376	-0.000010	...	0.000034	-1.629270e-05	-0.000064	0.000160	2.117278e-06
2	0.000121	1.985062e-04	0.000286	0.000149	0.000407	0.000203	0.000201	0.000179	0.000405	-0.000005	...	0.000009	-4.098365e-07	-0.000044	0.000158	-2.595618e-06
3	0.000092	1.726065e-04	0.000149	0.000235	0.000300	0.000142	0.000143	0.000160	0.000389	-0.000010	...	0.000026	-3.619997e-06	-0.000051	0.000144	2.106403e-06
4	0.000227	4.314935e-04	0.000407	0.000300	0.001982	0.000286	0.000289	0.000288	0.000867	-0.000044	...	0.000140	-9.726774e-05	-0.000110	0.000335	4.015006e-05
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000021	-2.435276e-07	0.000009	-0.000009	0.000007	0.000011	0.000015	0.000093	-0.000073	0.000045	...	0.000047	1.965446e-05	0.000045	0.000011	1.719211e-06
97	0.000090	7.873728e-05	0.000089	0.000070	0.000183	0.000108	0.000108	0.000119	0.000143	0.000045	...	0.000112	1.776001e-05	0.000012	0.000111	1.009056e-05
98	0.000127	1.194770e-04	0.000089	0.000040	0.000330	0.000125	0.000128	0.000109	0.000269	0.000124	...	0.000290	4.028915e-05	0.000055	0.000128	-4.136069e-06
99	0.000080	3.491064e-05	0.000033	0.000041	0.000175	0.000068	0.000067	0.000116	0.000144	0.000129	...	0.000273	4.581371e-05	0.000056	0.000068	3.616945e-06
100	0.000086	8.312289e-05	0.000074	0.000103	0.000101	0.000081	0.000085	0.000126	0.000166	0.000028	...	0.000097	3.597227e-05	-0.000016	0.000092	8.084075e-07

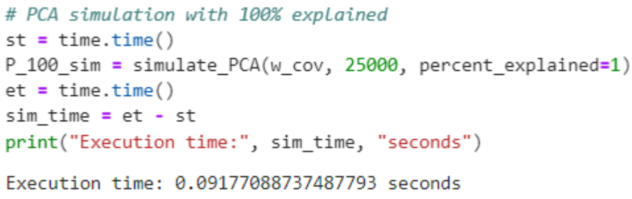
And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:



```
diff = d_sim_cov.to_numpy() - w_cov.to_numpy()
F_Norm(diff)

5.104988340383546e-08
```

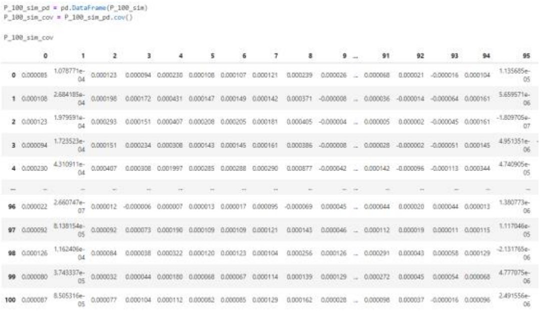
b. PCA with 100% Explained

I called "simulate_PCA" function with number of draws set to be 25,000 and "percent_explained" set to be 1, and timed this simulation process to check runtime, the runtime for direct simulation over exponentially weighted covariance matrix is:



```
# PCA simulation with 100% explained
st = time.time()
P_100_sim = simulate_PCA(w_cov, 25000, percent_explained=1)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")

Execution time: 0.09177088737487793 seconds
```

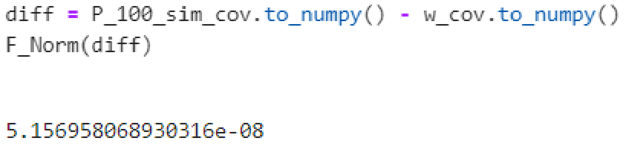
Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):



```
P_100_sim_pd = pd.DataFrame(P_100_sim)
P_100_sim_cov = P_100_sim_pd.cov()

P_100_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95
0	0.000085	1.078771e-04	0.000123	0.000094	0.000230	0.000108	0.000107	0.000121	0.000239	0.000026	...	0.000068	0.000021	-0.000016	0.000104	1.135685e-05
1	0.000108	2.684185e-04	0.000198	0.000172	0.000431	0.000147	0.000149	0.000142	0.000371	-0.000008	...	0.000036	-0.000014	-0.000064	0.000161	5.659571e-06
2	0.000123	1.979591e-04	0.000293	0.000151	0.000407	0.000208	0.000205	0.000181	0.000405	-0.000004	...	0.000005	0.000002	-0.000045	0.000161	-1.809705e-07
3	0.000094	1.723523e-04	0.000151	0.000234	0.000308	0.000143	0.000145	0.000161	0.000386	-0.000008	...	0.000028	-0.000002	-0.000051	0.000145	4.951351e-06
4	0.000230	4.310911e-04	0.000407	0.000308	0.001997	0.000285	0.000288	0.000290	0.000877	-0.000042	...	0.000142	-0.000096	-0.000113	0.000344	4.740905e-05
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000022	2.660747e-07	0.000012	-0.000006	0.000007	0.000013	0.000017	0.000095	-0.000069	0.000045	...	0.000044	0.000020	0.000044	0.000013	1.380773e-06
97	0.000092	8.138154e-05	0.000092	0.000073	0.000190	0.000109	0.000109	0.000121	0.000143	0.000046	...	0.000112	0.000019	0.000011	0.000115	1.117046e-05
98	0.000126	1.162406e-04	0.000084	0.000038	0.000322	0.000120	0.000123	0.000104	0.000256	0.000126	...	0.000291	0.000043	0.000058	0.000129	-2.131765e-06
99	0.000080	3.743337e-05	0.000032	0.000044	0.000180	0.000068	0.000067	0.000114	0.000139	0.000129	...	0.000272	0.000045	0.000054	0.000068	4.777075e-06
100	0.000087	8.505316e-05	0.000077	0.000104	0.000112	0.000082	0.000085	0.000129	0.000162	0.000028	...	0.000098	0.000037	-0.000016	0.000096	2.491556e-06

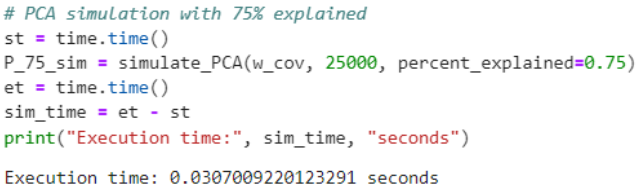
And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:



```
diff = P_100_sim_cov.to_numpy() - w_cov.to_numpy()
F_Norm(diff)


5.156958068930316e-08
```

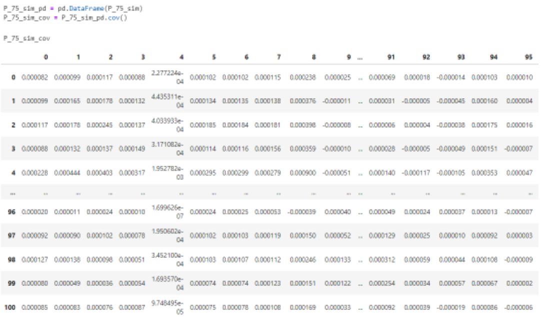
c. PCA with 75% Explained

I called "simulate_PCA" function with number of draws set to be 25,000 and "percent_explained" set to be 0.75, and timed this simulation process to check runtime, the runtime for direct simulation over exponentially weighted covariance matrix is:



```
# PCA simulation with 75% explained
st = time.time()
P_75_sim = simulate_PCA(w_cov, 25000, percent_explained=0.75)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")

Execution time: 0.0307009220123291 seconds
```

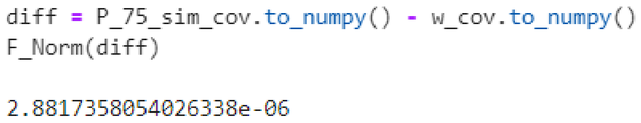
Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):



```
P_75_sim_pd = pd.DataFrame(P_75_sim)
P_75_sim_cov = P_75_sim_pd.cov()

P_75_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95
0	0.000082	0.000099	0.000117	0.000088	2.277224e-04	0.000102	0.000102	0.000115	0.000238	0.000025	...	0.000069	0.000018	-0.000014	0.000103	0.000010
1	0.000099	0.000165	0.000178	0.000132	4.435311e-04	0.000134	0.000135	0.000138	0.000376	-0.000011	...	0.000031	-0.000005	-0.000045	0.000160	0.000004
2	0.000117	0.000178	0.000245	0.000137	4.033933e-04	0.000185	0.000184	0.000181	0.000398	-0.000008	...	0.000006	0.000004	-0.000038	0.000175	0.000016
3	0.000088	0.000132	0.000137	0.000149	3.171082e-04	0.000114	0.000116	0.000156	0.000359	-0.000010	...	0.000028	-0.000005	-0.000049	0.000151	-0.000007
4	0.000228	0.000444	0.000403	0.000317	1.952782e-03	0.000295	0.000299	0.000279	0.000900	-0.000051	...	0.000140	-0.000117	-0.000105	0.000353	0.000047
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000020	0.000011	0.000024	0.000010	1.699626e-07	0.000024	0.000025	0.000053	-0.000039	0.000040	...	0.000049	0.000024	0.000037	0.000013	-0.000007
97	0.000092	0.000090	0.000102	0.000078	1.950602e-04	0.000102	0.000103	0.000119	0.000150	0.000052	...	0.000129	0.000025	0.000010	0.000092	0.000003
98	0.000127	0.000138	0.000098	0.000051	3.452100e-04	0.000103	0.000107	0.000112	0.000246	0.000133	...	0.000312	0.000059	0.000044	0.000108	-0.000009
99	0.000080	0.000049	0.000036	0.000054	1.693570e-04	0.000074	0.000074	0.000123	0.000151	0.000122	...	0.000254	0.000034	0.000057	0.000067	0.000002
100	0.000085	0.000083	0.000076	0.000087	9.748495e-05	0.000075	0.000078	0.000108	0.000169	0.000033	...	0.000092	0.000039	-0.000019	0.000086	-0.000006

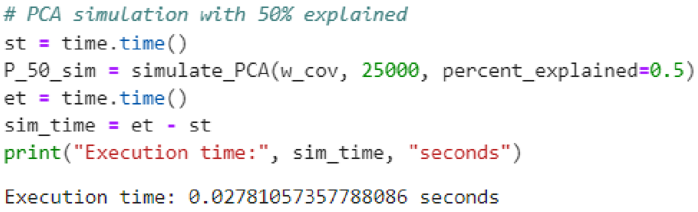
And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:



```
diff = P_75_sim_cov.to_numpy() - w_cov.to_numpy()
F_Norm(diff)

2.8817358054026338e-06
```

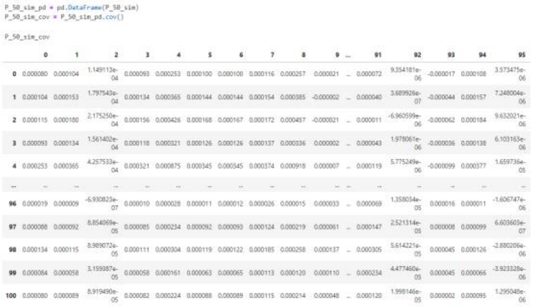
d. PCA with 50% Explained

I called "simulate_PCA" function with number of draws set to be 25,000 and "percent_explained" set to be 0.5, and timed this simulation process to check runtime, the runtime for direct simulation over exponentially weighted covariance matrix is:



```
# PCA simulation with 50% explained
st = time.time()
P_50_sim = simulate_PCA(w_cov, 25000, percent_explained=0.5)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")

Execution time: 0.02781057357788086 seconds
```

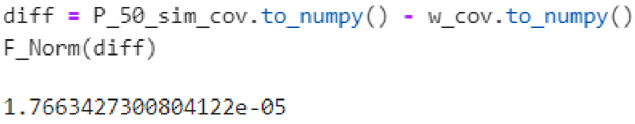
Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):



```
P_50_sim_pd = pd.DataFrame(P_50_sim)
P_50_sim_cov = P_50_sim_pd.cov()

P_50_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95
0	0.000080	0.000104	1.149113e-04	0.000093	0.000253	0.000100	0.000100	0.000116	0.000257	0.000021	...	0.000072	9.354181e-06	-0.000017	0.000108	3.573475e-06
1	0.000104	0.000153	1.797543e-04	0.000134	0.000365	0.000144	0.000144	0.000154	0.000385	-0.000002	...	0.000040	3.689926e-07	-0.000044	0.000157	7.248004e-06
2	0.000115	0.000180	2.175250e-04	0.000156	0.000426	0.000168	0.000167	0.000172	0.000457	-0.000021	...	0.000011	-6.960599e-06	-0.000062	0.000184	9.632021e-06
3	0.000093	0.000134	1.561402e-04	0.000118	0.000321	0.000126	0.000126	0.000137	0.000336	0.000002	...	0.000043	1.978061e-06	-0.000036	0.000138	6.103163e-06
4	0.000253	0.000365	4.257533e-04	0.000321	0.000875	0.000345	0.000345	0.000374	0.000918	0.000007	...	0.000119	5.775249e-06	-0.000099	0.000377	1.659736e-05
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000019	0.000009	-6.930823e-07	0.000010	0.000028	0.000011	0.000012	0.000026	0.000015	0.000033	...	0.000069	1.358034e-05	0.000016	0.000011	-1.606747e-06
97	0.000088	0.000092	8.854069e-05	0.000085	0.000234	0.000092	0.000093	0.000124	0.000219	0.000061	...	0.000147	2.521314e-05	0.000008	0.000099	6.603603e-07
98	0.000134	0.000115	8.989072e-05	0.000111	0.000304	0.000119	0.000122	0.000185	0.000258	0.000137	...	0.000305	5.614221e-05	0.000045	0.000126	-2.880206e-06
99	0.000084	0.000058	3.159387e-05	0.000058	0.000161	0.000063	0.000065	0.000113	0.000120	0.000110	...	0.000234	4.477460e-05	0.000045	0.000066	-3.923328e-06
100	0.000080	0.000089	8.919490e-05	0.000082	0.000224	0.000088	0.000089	0.000115	0.000214	0.000048	...	0.000120	1.998146e-05	0.000002	0.000095	1.295048e-06

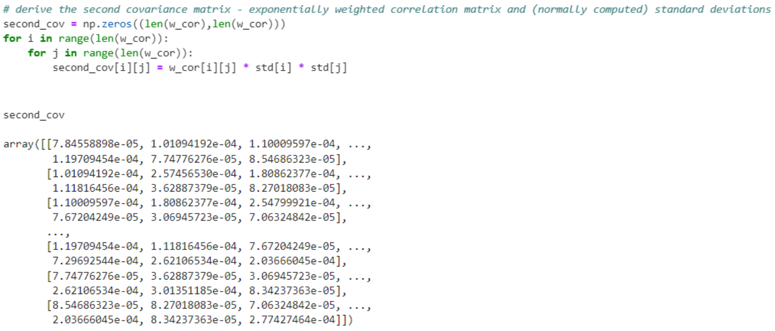
And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:



```
diff = P_50_sim_cov.to_numpy() - w_cov.to_numpy()
F_Norm(diff)

1.7663427300804122e-05
```

2. Second Covariance Matrix

For the second covariance matrix, I used exponentially weighted correlation matrix and (normally computed) standard deviations – the exponentially weighted correlation matrix was generated beforehand with exponentially weighted covariance matrix and exponentially weighted standard deviations (part of the matrix due to too many items):

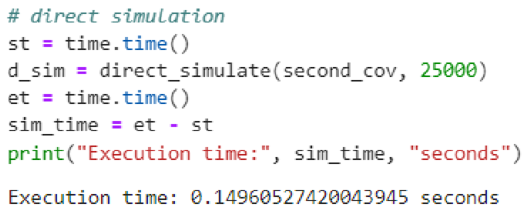


```
# derive the second covariance matrix - exponentially weighted correlation matrix and (normally computed) standard deviations
second_cov = np.zeros((len(w_cor),len(w_cor)))
for i in range(len(w_cor)):
    for j in range(len(w_cor)):
        second_cov[i][j] = w_cor[i][j] * std[i] * std[j]


second_cov

array([[7.84558898e-05, 1.01094192e-04, 1.10009597e-04, ...,
        1.19709454e-04, 7.74776276e-05, 8.54686323e-05],
       [1.01094192e-04, 2.57456530e-04, 1.80862377e-04, ...,
        1.11816456e-04, 3.62887379e-05, 8.27018083e-05],
       [1.10009597e-04, 1.80862377e-04, 2.54799921e-04, ...,
        7.67204249e-05, 3.06945723e-05, 7.06324842e-05],
       ...,
       [1.19709454e-04, 1.11816456e-04, 7.67204249e-05, ...,
        7.29692544e-04, 2.62106534e-04, 2.03666045e-04],
       [7.74776276e-05, 3.62887379e-05, 3.06945723e-05, ...,
        2.62106534e-04, 3.01351185e-04, 8.34237363e-05],
       [8.54686323e-05, 8.27018083e-05, 7.06324842e-05, ...,
        2.03666045e-04, 8.34237363e-05, 2.77427464e-04]])
```

With the second covariance matrix, I start running simulations over the matrix:

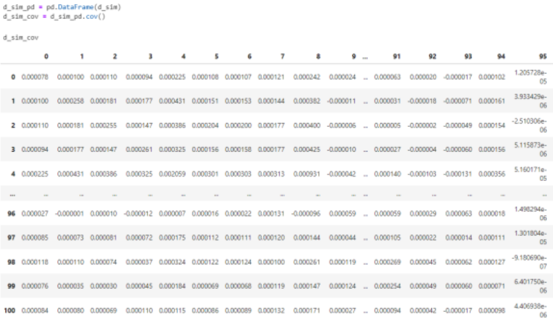
a. Direct Simulation

I called “direct_simulate” function with number of draws set to be 25,000, and timed this simulation process to check runtime, the runtime for direct simulation over the second matrix is:



```
# direct simulation
st = time.time()
d_sim = direct_simulate(second_cov, 25000)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")

Execution time: 0.14960527420043945 seconds
```

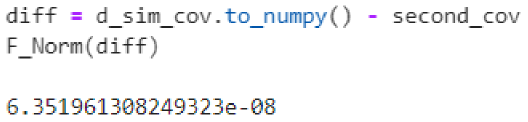
Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):



```
d_sim_pd = pd.DataFrame(d_sim)
d_sim_cov = d_sim_pd.cov()

d_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95
0	0.000078	0.000100	0.000110	0.000094	0.000225	0.000108	0.000107	0.000121	0.000242	0.000024	...	0.000063	0.000020	-0.000017	0.000102	1.205728e-05
1	0.000100	0.000258	0.000181	0.000177	0.000431	0.000151	0.000153	0.000144	0.000382	-0.000011	...	0.000031	-0.000018	-0.000071	0.000161	3.933429e-06
2	0.000110	0.000181	0.000255	0.000147	0.000386	0.000204	0.000200	0.000177	0.000400	-0.000006	...	0.000005	-0.000002	-0.000049	0.000154	-2.510306e-06
3	0.000094	0.000177	0.000147	0.000261	0.000325	0.000156	0.000158	0.000177	0.000425	-0.000010	...	0.000027	-0.000004	-0.000060	0.000156	5.115873e-06
4	0.000225	0.000431	0.000386	0.000325	0.002059	0.000301	0.000303	0.000313	0.000931	-0.000042	...	0.000140	-0.000103	-0.000131	0.000356	5.160171e-05
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000027	-0.000001	0.000010	-0.000012	0.000007	0.000016	0.000022	0.000131	-0.000096	0.000059	...	0.000059	0.000029	0.000063	0.000018	1.498294e-06
97	0.000085	0.000073	0.000081	0.000072	0.000175	0.000112	0.000111	0.000120	0.000144	0.000044	...	0.000105	0.000022	0.000014	0.000111	1.301804e-05
98	0.000118	0.000110	0.000074	0.000037	0.000324	0.000122	0.000124	0.000100	0.000261	0.000119	...	0.000269	0.000045	0.000062	0.000127	-9.180690e-07
99	0.000076	0.000035	0.000030	0.000045	0.000184	0.000069	0.000068	0.000119	0.000147	0.000124	...	0.000254	0.000049	0.000060	0.000071	6.401750e-06
100	0.000084	0.000080	0.000069	0.000110	0.000115	0.000086	0.000089	0.000132	0.000171	0.000027	...	0.000094	0.000042	-0.000017	0.000098	4.406938e-06

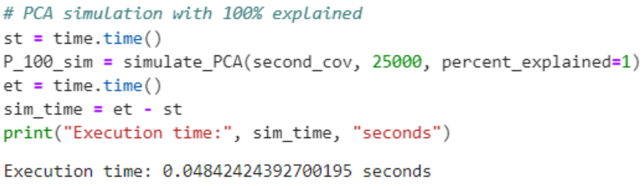
And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:



```
diff = d_sim_cov.to_numpy() - second_cov
F_Norm(diff)

6.351961308249323e-08
```

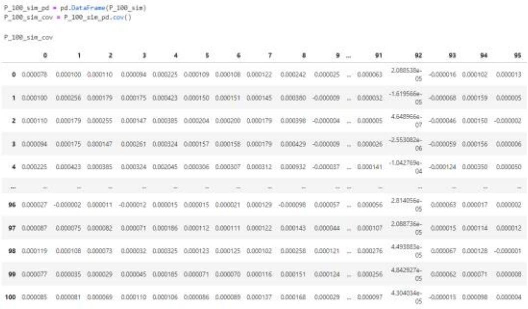
b. PCA with 100% Explained

I called "simulate_PCA" function with number of draws set to be 25,000 and "percent_explained" set to be 1, and timed this simulation process to check runtime, the runtime for direct simulation over the second covariance matrix is:



```
# PCA simulation with 100% explained
st = time.time()
P_100_sim = simulate_PCA(second_cov, 25000, percent_explained=1)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")

Execution time: 0.04842424392700195 seconds
```

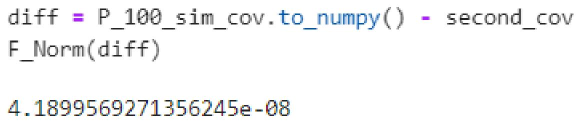
Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):



```
P_100_sim_pd = pd.DataFrame(P_100_sim)
P_100_sim_cov = P_100_sim_pd.cov()

P_100_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95
0	0.000078	0.000100	0.000110	0.000094	0.000225	0.000109	0.000108	0.000122	0.000242	0.000025	...	0.000063	2.088538e-05	-0.000016	0.000102	0.000013
1	0.000100	0.000256	0.000179	0.000175	0.000423	0.000150	0.000151	0.000145	0.000380	-0.000009	...	0.000032	-1.619566e-05	-0.000068	0.000159	0.000005
2	0.000110	0.000179	0.000255	0.000147	0.000385	0.000204	0.000200	0.000179	0.000398	-0.000004	...	0.000005	4.648966e-07	-0.000046	0.000150	-0.000002
3	0.000094	0.000175	0.000147	0.000261	0.000324	0.000157	0.000158	0.000179	0.000429	-0.000009	...	0.000026	-2.553082e-06	-0.000059	0.000156	0.000006
4	0.000225	0.000423	0.000385	0.000324	0.002045	0.000306	0.000307	0.000312	0.000932	-0.000037	...	0.000141	-1.042769e-04	-0.000124	0.000350	0.000050
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000027	-0.000002	0.000011	-0.000012	0.000015	0.000015	0.000021	0.000129	-0.000098	0.000057	...	0.000056	2.814056e-05	0.000063	0.000017	0.000002
97	0.000087	0.000075	0.000082	0.000071	0.000186	0.000112	0.000111	0.000122	0.000143	0.000044	...	0.000107	2.088736e-05	0.000015	0.000114	0.000012
98	0.000119	0.000108	0.000073	0.000032	0.000325	0.000123	0.000125	0.000102	0.000258	0.000121	...	0.000276	4.493883e-05	0.000067	0.000128	-0.000001
99	0.000077	0.000035	0.000029	0.000045	0.000185	0.000071	0.000070	0.000116	0.000151	0.000124	...	0.000256	4.842927e-05	0.000062	0.000071	0.000008
100	0.000085	0.000081	0.000069	0.000110	0.000106	0.000086	0.000089	0.000137	0.000168	0.000029	...	0.000097	4.304034e-05	-0.000015	0.000098	0.000004

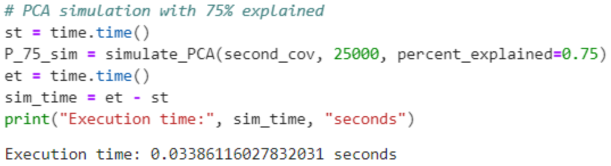
And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:



```
diff = P_100_sim_cov.to_numpy() - second_cov
F_Norm(diff)

4.1899569271356245e-08
```

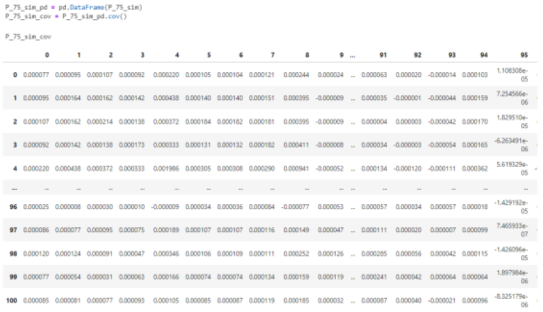
c. PCA with 75% Explained

I called “simulate_PCA” function with number of draws set to be 25,000 and “percent_explained” set to be 0.75, and timed this simulation process to check runtime, the runtime for direct simulation over the second matrix is:



```
# PCA simulation with 75% explained
st = time.time()
P_75_sim = simulate_PCA(second_cov, 25000, percent_explained=0.75)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")

Execution time: 0.03386116027832031 seconds
```

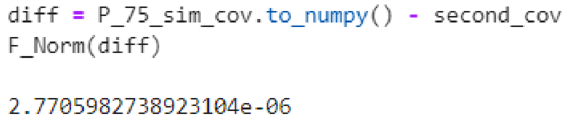
Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):



```
P_75_sim_pd = pd.DataFrame(P_75_sim)
P_75_sim_cov = P_75_sim_pd.cov()

P_75_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95
0	0.000077	0.000095	0.000107	0.000092	0.000220	0.000105	0.000104	0.000121	0.000244	0.000024	...	0.000063	0.000020	-0.000014	0.000103	1.108308e-05
1	0.000095	0.000164	0.000162	0.000142	0.000438	0.000140	0.000140	0.000151	0.000395	-0.000009	...	0.000035	-0.000001	-0.000044	0.000159	7.254566e-06
2	0.000107	0.000162	0.000214	0.000138	0.000372	0.000184	0.000182	0.000181	0.000395	-0.000009	...	0.000004	0.000003	-0.000042	0.000170	1.829510e-05
3	0.000092	0.000142	0.000138	0.000173	0.000333	0.000131	0.000132	0.000182	0.000411	-0.000008	...	0.000034	-0.000003	-0.000054	0.000165	-6.263491e-06
4	0.000220	0.000438	0.000372	0.000333	0.001986	0.000305	0.000308	0.000290	0.000941	-0.000052	...	0.000134	-0.000120	-0.000111	0.000362	5.619329e-05
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000025	0.000008	0.000030	0.000010	-0.000009	0.000034	0.000036	0.000084	-0.000077	0.000053	...	0.000057	0.000034	0.000057	0.000018	-1.429192e-05
97	0.000086	0.000077	0.000095	0.000075	0.000189	0.000107	0.000107	0.000116	0.000149	0.000047	...	0.000111	0.000020	0.000007	0.000099	7.465933e-07
98	0.000120	0.000124	0.000091	0.000047	0.000346	0.000106	0.000109	0.000111	0.000252	0.000126	...	0.000285	0.000056	0.000042	0.000115	-1.426096e-05
99	0.000077	0.000054	0.000031	0.000063	0.000166	0.000074	0.000074	0.000134	0.000159	0.000119	...	0.000241	0.000042	0.000064	0.000064	1.897984e-06
100	0.000085	0.000081	0.000077	0.000093	0.000105	0.000085	0.000087	0.000119	0.000185	0.000032	...	0.000087	0.000040	-0.000021	0.000096	-8.325179e-06

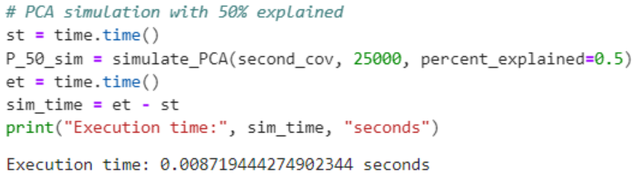
And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:



```
diff = P_75_sim_cov.to_numpy() - second_cov
F_Norm(diff)

2.7705982738923104e-06
```

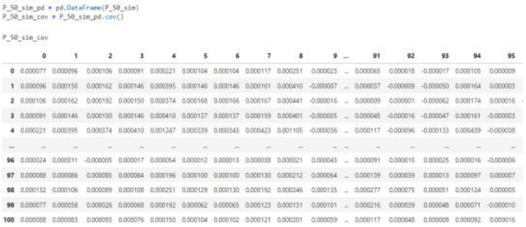
d. PCA with 50% Explained

I called “simulate_PCA” function with number of draws set to be 25,000 and “percent_explained” set to be 0.5, and timed this simulation process to check runtime, the runtime for direct simulation over the second matrix is:



```
# PCA simulation with 50% explained
st = time.time()
P_50_sim = simulate_PCA(second_cov, 25000, percent_explained=0.5)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")

Execution time: 0.008719444274902344 seconds
```

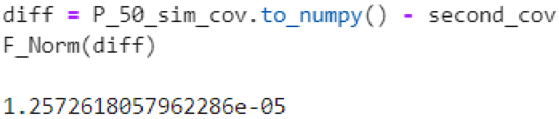
Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):



```
P_50_sim_pd = pd.DataFrame(P_50_sim)
P_50_sim_cov = P_50_sim_pd.cov()

P_50_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95
0	0.000077	0.000096	0.000106	0.000091	0.000221	0.000104	0.000104	0.000117	0.000251	0.000023	...	0.000065	0.000018	-0.000017	0.000105	0.000009
1	0.000096	0.000150	0.000162	0.000146	0.000395	0.000146	0.000146	0.000161	0.000410	-0.000007	...	0.000037	-0.000009	-0.000050	0.000164	0.000003
2	0.000106	0.000162	0.000192	0.000150	0.000374	0.000168	0.000166	0.000167	0.000441	-0.000016	...	0.000009	0.000001	-0.000062	0.000174	0.000016
3	0.000091	0.000146	0.000150	0.000146	0.000410	0.000137	0.000137	0.000159	0.000401	-0.000005	...	0.000045	-0.000016	-0.000047	0.000161	-0.000003
4	0.000221	0.000395	0.000374	0.000410	0.001247	0.000339	0.000343	0.000423	0.001105	-0.000036	...	0.000117	-0.000096	-0.000133	0.000439	-0.000038
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000024	0.000011	-0.000005	0.000017	0.000054	0.000012	0.000013	0.000038	0.000021	0.000043	...	0.000091	0.000015	0.000025	0.000016	-0.000006
97	0.000088	0.000086	0.000085	0.000084	0.000196	0.000100	0.000100	0.000130	0.000212	0.000064	...	0.000139	0.000039	0.000013	0.000097	0.000007
98	0.000132	0.000106	0.000089	0.000108	0.000251	0.000129	0.000130	0.000192	0.000246	0.000135	...	0.000277	0.000075	0.000051	0.000124	0.000005
99	0.000077	0.000058	0.000026	0.000068	0.000192	0.000062	0.000065	0.000123	0.000131	0.000101	...	0.000216	0.000039	0.000048	0.000071	-0.000010
100	0.000088	0.000083	0.000093	0.000076	0.000150	0.000104	0.000102	0.000121	0.000201	0.000059	...	0.000117	0.000048	0.000008	0.000092	0.000016

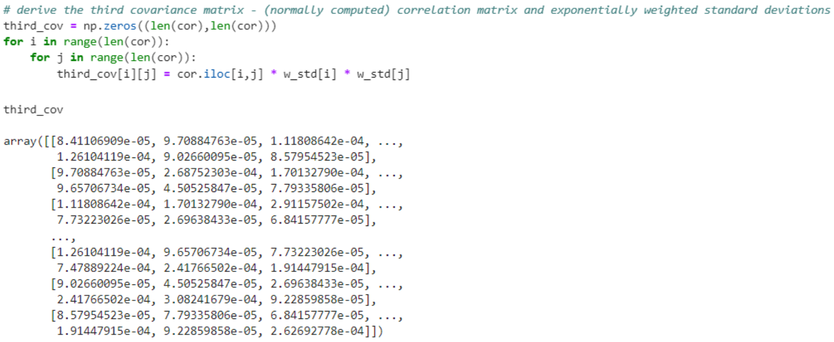
And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:



```
diff = P_50_sim_cov.to_numpy() - second_cov
F_Norm(diff)

1.2572618057962286e-05
```

3. Third Covariance Matrix

For the third covariance matrix, I used (normally computed) correlation matrix and exponentially weighted standard deviations (part of the matrix due to too many items):

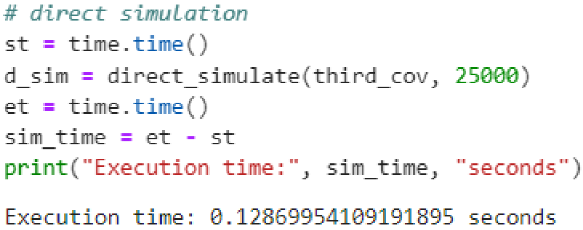


```
# derive the third covariance matrix - (normally computed) correlation matrix and exponentially weighted standard deviations
third_cov = np.zeros((len(cor),len(cor)))
for i in range(len(cor)):
    for j in range(len(cor)):
        third_cov[i][j] = cor.iloc[i,j] * w_std[i] * w_std[j]

third_cov

array([[8.41106909e-05, 9.70884763e-05, 1.11808642e-04, ...,
        1.26104119e-04, 9.02660095e-05, 8.57954523e-05],
       [9.70884763e-05, 2.68752303e-04, 1.70132790e-04, ...,
        9.65706734e-05, 4.50525847e-05, 7.79335806e-05],
       [1.11808642e-04, 1.70132790e-04, 2.91157502e-04, ...,
        7.73223026e-05, 2.69638433e-05, 6.84157777e-05],
       ...,
       [1.26104119e-04, 9.65706734e-05, 7.73223026e-05, ...,
        7.47889224e-05, 2.41766502e-04, 1.91447915e-04],
       [9.02660095e-05, 4.50525847e-05, 2.69638433e-05, ...,
        2.41766502e-04, 3.08241679e-04, 9.22859858e-05],
       [8.57954523e-05, 7.79335806e-05, 6.84157777e-05, ...,
        1.91447915e-04, 9.22859858e-05, 2.62692778e-04]])
```

With the third covariance matrix, I start running simulations over the matrix:

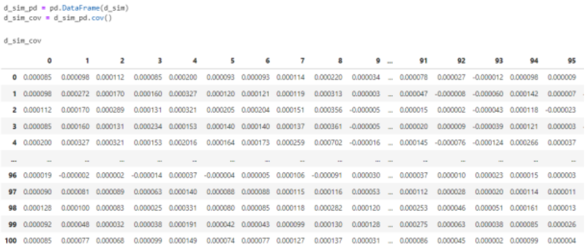
a. Direct Simulation

I called "direct_simulate" function with number of draws set to be 25,000, and timed this simulation process to check runtime, the runtime for direct simulation over the third matrix is:



```
# direct simulation
st = time.time()
d_sim = direct_simulate(third_cov, 25000)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")

Execution time: 0.12869954109191895 seconds
```

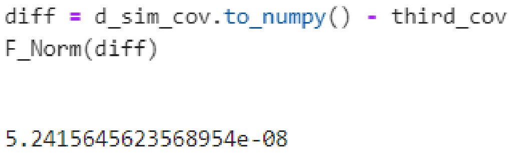
Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):



```
d_sim_pd = pd.DataFrame(d_sim)
d_sim_cov = d_sim_pd.cov()

d_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95
0	0.000085	0.000098	0.000112	0.000085	0.000200	0.000093	0.000093	0.000114	0.000220	0.000034	...	0.000078	0.000027	-0.000012	0.000098	0.000009
1	0.000098	0.000272	0.000170	0.000160	0.000327	0.000120	0.000121	0.000119	0.000313	0.000003	...	0.000047	-0.000008	-0.000060	0.000142	0.000007
2	0.000112	0.000170	0.000289	0.000131	0.000321	0.000205	0.000204	0.000151	0.000356	-0.000005	...	0.000015	0.000002	-0.000043	0.000118	-0.000023
3	0.000085	0.000160	0.000131	0.000234	0.000153	0.000140	0.000140	0.000137	0.000361	-0.000005	...	0.000020	0.000009	-0.000039	0.000121	0.000003
4	0.000200	0.000327	0.000321	0.000153	0.002016	0.000164	0.000173	0.000259	0.000702	-0.000016	...	0.000145	-0.000076	-0.000124	0.000266	0.000037
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000019	-0.000002	0.000002	-0.000014	0.000037	-0.000004	0.000005	0.000106	-0.000091	0.000030	...	0.000037	0.000010	0.000023	0.000015	0.000003
97	0.000090	0.000081	0.000089	0.000063	0.000140	0.000088	0.000088	0.000115	0.000116	0.000053	...	0.000112	0.000028	0.000020	0.000114	0.000011
98	0.000128	0.000100	0.000083	0.000025	0.000331	0.000080	0.000085	0.000118	0.000282	0.000120	...	0.000253	0.000046	0.000051	0.000161	0.000013
99	0.000092	0.000048	0.000032	0.000038	0.000191	0.000042	0.000043	0.000099	0.000130	0.000128	...	0.000275	0.000063	0.000038	0.000085	0.000026
100	0.000085	0.000077	0.000068	0.000099	0.000149	0.000074	0.000077	0.000127	0.000137	0.000031	...	0.000086	0.000045	0.000002	0.000099	0.000004

And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:

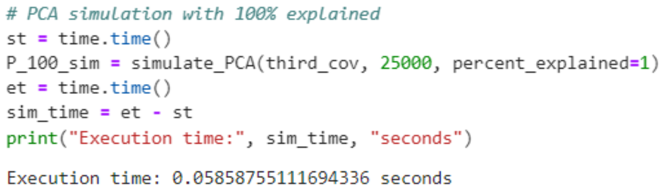


```
diff = d_sim_cov.to_numpy() - third_cov
F_Norm(diff)
```

```
5.2415645623568954e-08
```

b. PCA with 100% Explained

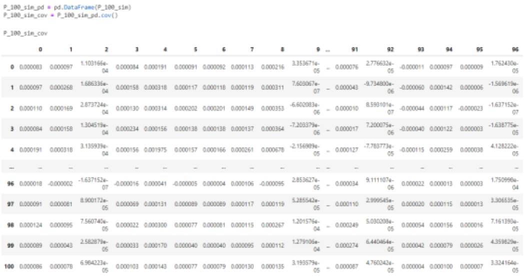
I called "simulate_PCA" function with number of draws set to be 25,000 and "percent_explained" set to be 1, and timed this simulation process to check runtime, the runtime for direct simulation over the third matrix is:



```
# PCA simulation with 100% explained
st = time.time()
P_100_sim = simulate_PCA(third_cov, 25000, percent_explained=1)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")
```

```
Execution time: 0.05858755111694336 seconds
```

Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):

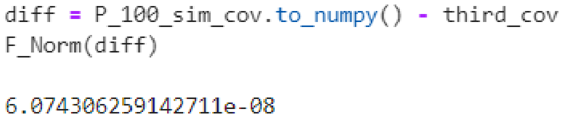


```
P_100_sim_pd = pd.DataFrame(P_100_sim)
P_100_sim_cov = P_100_sim_pd.cov()
```

```
P_100_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95	96
0	0.000083	0.000097	1.103166e-04	0.000084	0.000191	0.000091	0.000092	0.000113	0.000216	3.353671e-05	...	0.000076	2.776632e-05	-0.000011	0.000097	0.000009	1.762430e-05
1	0.000097	0.000268	1.686336e-04	0.000158	0.000318	0.000117	0.000118	0.000119	0.000311	7.603067e-07	...	0.000043	-9.734800e-06	-0.000060	0.000142	0.000006	-1.569619e-06
2	0.000110	0.000169	2.873724e-04	0.000130	0.000314	0.000202	0.000201	0.000149	0.000353	-6.602083e-06	...	0.000010	8.593101e-07	-0.000044	0.000117	-0.000023	-1.637152e-07
3	0.000084	0.000158	1.304519e-04	0.000234	0.000156	0.000138	0.000138	0.000137	0.000364	-7.203379e-06	...	0.000017	7.200075e-06	-0.000040	0.000122	0.000003	-1.638775e-05
4	0.000191	0.000318	3.135939e-04	0.000156	0.001975	0.000157	0.000166	0.000261	0.000678	-2.156989e-05	...	0.000127	-7.783773e-05	-0.000115	0.000259	0.000038	4.128222e-05
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000018	-0.000002	-1.637152e-07	-0.000016	0.000041	-0.000005	0.000004	0.000106	-0.000095	2.853627e-05	...	0.000034	9.111107e-06	0.000022	0.000013	0.000003	1.750998e-04
97	0.000091	0.000081	8.900172e-05	0.000069	0.000131	0.000089	0.000089	0.000117	0.000119	5.285542e-05	...	0.000110	2.999545e-05	0.000020	0.000115	0.000013	3.306535e-05
98	0.000124	0.000095	7.560740e-05	0.000022	0.000300	0.000077	0.000081	0.000115	0.000267	1.201576e-04	...	0.000249	5.030208e-05	0.000054	0.000156	0.000016	7.161393e-05
99	0.000089	0.000043	2.582879e-05	0.000033	0.000170	0.000040	0.000040	0.000095	0.000112	1.279106e-04	...	0.000274	6.440464e-05	0.000042	0.000079	0.000026	4.359829e-05
100	0.000086	0.000078	6.984223e-05	0.000103	0.000143	0.000077	0.000079	0.000130	0.000135	3.193579e-05	...	0.000087	4.760242e-05	0.000004	0.000100	0.000007	3.324164e-05

And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:

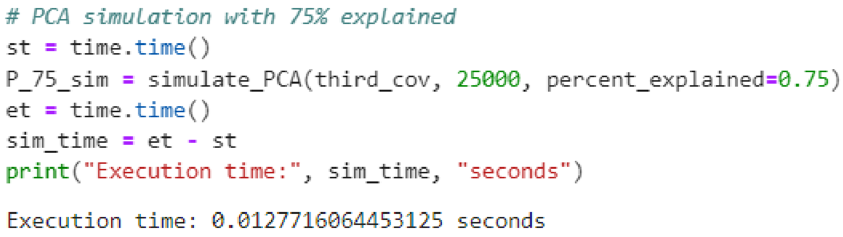


```
diff = P_100_sim_cov.to_numpy() - third_cov
F_Norm(diff)
```

```
6.074306259142711e-08
```

c. PCA with 75% Explained

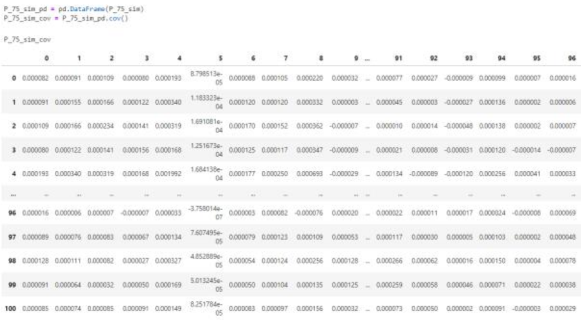
I called "simulate_PCA" function with number of draws set to be 25,000 and "percent_explained" set to be 0.75, and timed this simulation process to check runtime, the runtime for direct simulation over the third matrix is:



```
# PCA simulation with 75% explained
st = time.time()
P_75_sim = simulate_PCA(third_cov, 25000, percent_explained=0.75)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")
```

```
Execution time: 0.0127716064453125 seconds
```

Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):



```
P_75_sim_pd = pd.DataFrame(P_75_sim)
P_75_sim_cov = P_75_sim_pd.cov()
```

```
P_75_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95	96
0	0.000082	0.000091	0.000109	0.000080	0.000193	8.798513e-05	0.000088	0.000105	0.000220	0.000032	...	0.000077	0.000027	-0.000009	0.000099	0.000007	0.000016
1	0.000091	0.000155	0.000166	0.000122	0.000340	1.183323e-04	0.000120	0.000120	0.000332	0.000003	...	0.000045	0.000003	-0.000027	0.000136	0.000002	0.000006
2	0.000109	0.000166	0.000234	0.000141	0.000319	1.691081e-04	0.000170	0.000152	0.000362	-0.000007	...	0.000010	0.000014	-0.000048	0.000138	0.000002	0.000007
3	0.000080	0.000122	0.000141	0.000156	0.000168	1.251673e-04	0.000125	0.000117	0.000347	-0.000009	...	0.000021	0.000008	-0.000031	0.000120	-0.000014	-0.000007
4	0.000193	0.000340	0.000319	0.000168	0.001992	1.684138e-04	0.000177	0.000250	0.000693	-0.000029	...	0.000134	-0.000089	-0.000120	0.000256	0.000041	0.000033
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000016	0.000006	0.000007	-0.000007	0.000033	-3.758014e-07	0.000003	0.000082	-0.000076	0.000020	...	0.000022	0.000011	0.000017	0.000024	-0.000008	0.000069
97	0.000089	0.000076	0.000083	0.000067	0.000134	7.607495e-05	0.000079	0.000123	0.000109	0.000053	...	0.000117	0.000030	0.000005	0.000103	0.000002	0.000048
98	0.000128	0.000111	0.000082	0.000027	0.000327	4.852889e-05	0.000054	0.000124	0.000256	0.000128	...	0.000266	0.000062	0.000016	0.000150	0.000004	0.000078
99	0.000091	0.000064	0.000032	0.000050	0.000169	5.013245e-05	0.000050	0.000104	0.000135	0.000125	...	0.000259	0.000058	0.000046	0.000071	0.000022	0.000038
100	0.000085	0.000074	0.000085	0.000091	0.000149	8.251784e-05	0.000083	0.000097	0.000156	0.000032	...	0.000073	0.000050	0.000002	0.000091	-0.000003	0.000029

And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:

```
diff = P_75_sim_cov.to_numpy() - third_cov
F_Norm(diff)
```

```
2.810525441864881e-06
```

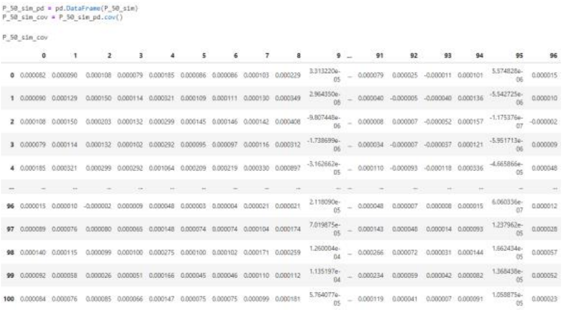
d. PCA with 50% Explained

I called "simulate_PCA" function with number of draws set to be 25,000 and "percent_explained" set to be 0.5, and timed this simulation process to check runtime, the runtime for direct simulation over the third matrix is:

```
# PCA simulation with 50% explained
st = time.time()
P_50_sim = simulate_PCA(third_cov, 25000, percent_explained=0.5)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")
```

```
Execution time: 0.011819839477539062 seconds
```

Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):



```
P_50_sim_pd = pd.DataFrame(P_50_sim)
P_50_sim_cov = P_50_sim_pd.cov()
```

```
P_50_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95	96
0	0.000082	0.000090	0.000108	0.000079	0.000185	0.000086	0.000086	0.000103	0.000229	3.313220e-05	...	0.000079	0.000025	-0.000011	0.000101	5.574828e-06	0.000015
1	0.000090	0.000129	0.000150	0.000114	0.000321	0.000109	0.000111	0.000130	0.000349	2.964350e-08	...	0.000040	-0.000005	-0.000040	0.000136	-5.542725e-06	0.000010
2	0.000108	0.000150	0.000203	0.000132	0.000299	0.000145	0.000146	0.000142	0.000408	-9.807448e-06	...	0.000008	0.000007	-0.000052	0.000157	-1.175376e-07	-0.000002
3	0.000079	0.000114	0.000132	0.000102	0.000292	0.000095	0.000097	0.000116	0.000312	-1.738699e-06	...	0.000034	-0.000007	-0.000037	0.000121	-5.951713e-06	0.000009
4	0.000185	0.000321	0.000299	0.000292	0.001064	0.000209	0.000219	0.000330	0.000897	-3.162662e-05	...	0.000110	-0.000093	-0.000118	0.000336	-4.665866e-05	0.000048
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000015	0.000010	-0.000002	0.000009	0.000048	0.000003	0.000004	0.000021	0.000021	2.118090e-05	...	0.000048	0.000007	0.000008	0.000015	6.060336e-07	0.000012
97	0.000089	0.000076	0.000080	0.000065	0.000148	0.000074	0.000074	0.000104	0.000174	7.019875e-05	...	0.000143	0.000048	0.000014	0.000093	1.237962e-05	0.000028
98	0.000140	0.000115	0.000099	0.000100	0.000275	0.000100	0.000102	0.000171	0.000259	1.260004e-04	...	0.000266	0.000072	0.000031	0.000144	1.662434e-05	0.000057
99	0.000092	0.000058	0.000026	0.000051	0.000166	0.000045	0.000046	0.000110	0.000112	1.135197e-04	...	0.000234	0.000059	0.000042	0.000082	1.368438e-05	0.000052
100	0.000084	0.000076	0.000085	0.000066	0.000147	0.000075	0.000075	0.000099	0.000181	5.764077e-05	...	0.000119	0.000041	0.000007	0.000091	1.058875e-05	0.000023

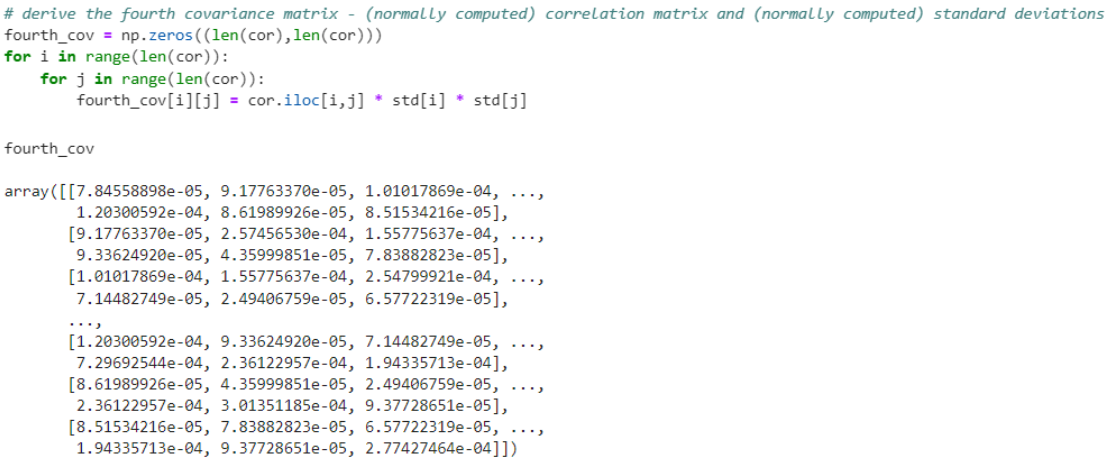
And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:

```
diff = P_50_sim_cov.to_numpy() - third_cov
F_Norm(diff)
```

```
1.3045028844069348e-05
```

4. Fourth Covariance Matrix

For the fourth covariance matrix, I used (normally computed) correlation matrix and (normally computed) standard deviations – which is the same as the covariance matrix of the original set of data (part of the matrix due to too many items):

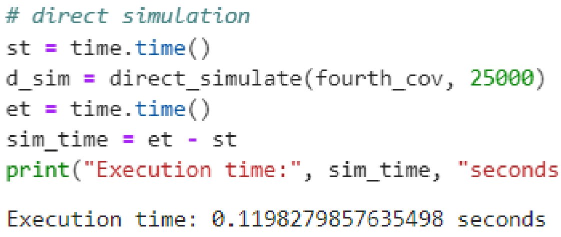


```
# derive the fourth covariance matrix - (normally computed) correlation matrix and (normally computed) standard deviations
fourth_cov = np.zeros((len(cor),len(cor)))
for i in range(len(cor)):
    for j in range(len(cor)):
        fourth_cov[i][j] = cor.iloc[i,j] * std[i] * std[j]

fourth_cov

array([[7.84558898e-05, 9.17763370e-05, 1.01017869e-04, ...,
        1.20300592e-04, 8.61989926e-05, 8.51534216e-05],
       [9.17763370e-05, 2.57456530e-04, 1.55775637e-04, ...,
        9.33624920e-05, 4.35999851e-05, 7.83882823e-05],
       [1.01017869e-04, 1.55775637e-04, 2.54799921e-04, ...,
        7.14482749e-05, 2.49406759e-05, 6.57722319e-05],
       ...,
       [1.20300592e-04, 9.33624920e-05, 7.14482749e-05, ...,
        7.29692544e-04, 2.36122957e-04, 1.94335713e-04],
       [8.61989926e-05, 4.35999851e-05, 2.49406759e-05, ...,
        2.36122957e-04, 3.01351185e-04, 9.37728651e-05],
       [8.51534216e-05, 7.83882823e-05, 6.57722319e-05, ...,
        1.94335713e-04, 9.37728651e-05, 2.77427464e-04]])
```

With the fourth covariance matrix, I start running simulations over the matrix:

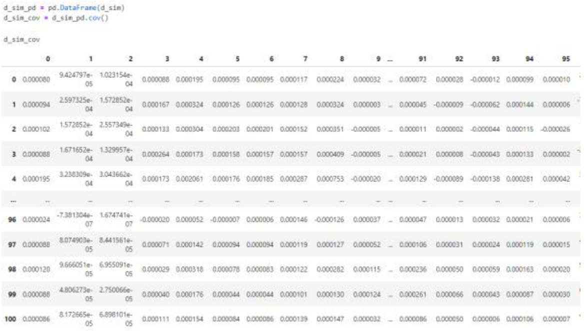
a. Direct Simulation

I called "direct_simulate" function with number of draws set to be 25,000, and timed this simulation process to check runtime, the runtime for direct simulation over the fourth matrix is:



```
# direct simulation
st = time.time()
d_sim = direct_simulate(fourth_cov, 25000)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")

Execution time: 0.1198279857635498 seconds
```

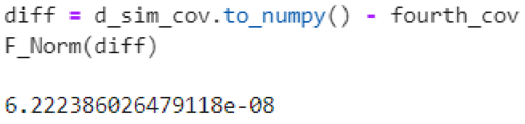
Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):



```
d_sim_pd = pd.DataFrame(d_sim)
d_sim_cov = d_sim_pd.cov()

d_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95
0	0.000080	9.424797e-05	1.023154e-04	0.000088	0.000195	0.000095	0.000095	0.000117	0.000224	0.000032	...	0.000072	0.000028	-0.000012	0.000099	0.000010
1	0.000094	2.597325e-04	1.572852e-04	0.000167	0.000324	0.000126	0.000126	0.000128	0.000324	0.000003	...	0.000045	-0.000009	-0.000062	0.000144	0.000006
2	0.000102	1.572852e-04	2.557349e-04	0.000133	0.000304	0.000203	0.000201	0.000152	0.000351	-0.000005	...	0.000011	0.000002	-0.000044	0.000115	-0.000026
3	0.000088	1.671652e-04	1.329957e-04	0.000264	0.000173	0.000158	0.000157	0.000157	0.000409	-0.000005	...	0.000021	0.000008	-0.000043	0.000133	0.000002
4	0.000195	3.238309e-04	3.043662e-04	0.000173	0.002061	0.000176	0.000185	0.000287	0.000753	-0.000020	...	0.000129	-0.000089	-0.000138	0.000281	0.000042
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000024	-7.381304e-07	1.674741e-07	-0.000020	0.000052	-0.000007	0.000006	0.000146	-0.000126	0.000037	...	0.000047	0.000013	0.000032	0.000021	0.000006
97	0.000088	8.074903e-05	8.441561e-05	0.000071	0.000142	0.000094	0.000094	0.000119	0.000127	0.000052	...	0.000106	0.000031	0.000024	0.000119	0.000015
98	0.000120	9.666051e-05	6.955091e-05	0.000029	0.000318	0.000078	0.000083	0.000122	0.000282	0.000115	...	0.000236	0.000050	0.000059	0.000163	0.000020
99	0.000088	4.806273e-05	2.750066e-05	0.000040	0.000176	0.000044	0.000044	0.000101	0.000130	0.000124	...	0.000261	0.000066	0.000043	0.000087	0.000030
100	0.000086	8.172665e-05	6.898101e-05	0.000111	0.000154	0.000084	0.000086	0.000139	0.000147	0.000032	...	0.000086	0.000050	0.000006	0.000106	0.000007

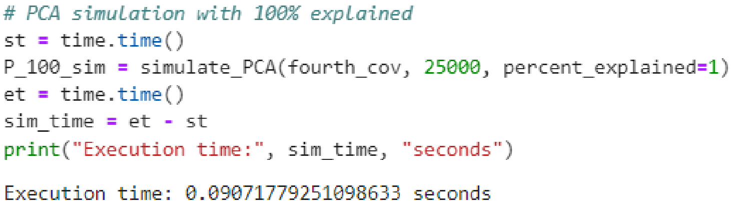
And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:



```
diff = d_sim_cov.to_numpy() - fourth_cov
F_Norm(diff)

6.222386026479118e-08
```

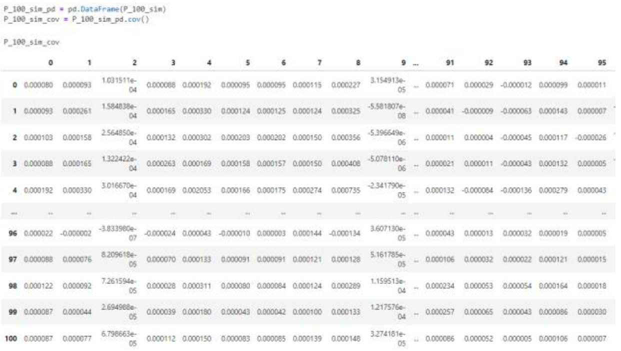
b. PCA with 100% Explained

I called "simulate_PCA" function with number of draws set to be 25,000 and "percent_explained" set to be 1, and timed this simulation process to check runtime, the runtime for direct simulation over the fourth matrix is:



```
# PCA simulation with 100% explained
st = time.time()
P_100_sim = simulate_PCA(fourth_cov, 25000, percent_explained=1)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")

Execution time: 0.09071779251098633 seconds
```

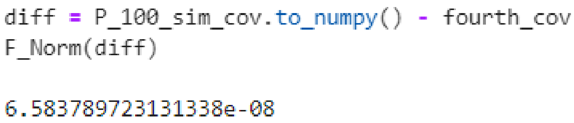
Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):



```
P_100_sim_pd = pd.DataFrame(P_100_sim)
P_100_sim_cov = P_100_sim_pd.cov()

P_100_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95
0	0.000080	0.000093	1.031511e-04	0.000088	0.000192	0.000095	0.000095	0.000115	0.000227	3.154913e-05	...	0.000071	0.000029	-0.000012	0.000099	0.000011
1	0.000093	0.000261	1.584838e-04	0.000165	0.000330	0.000124	0.000125	0.000124	0.000325	-5.581807e-08	...	0.000041	-0.000009	-0.000063	0.000143	0.000007
2	0.000103	0.000158	2.564850e-04	0.000132	0.000302	0.000203	0.000202	0.000150	0.000356	-5.396649e-06	...	0.000011	0.000004	-0.000045	0.000117	-0.000026
3	0.000088	0.000165	1.322422e-04	0.000263	0.000169	0.000158	0.000157	0.000150	0.000408	-5.078110e-06	...	0.000021	0.000011	-0.000043	0.000132	0.000005
4	0.000192	0.000330	3.016670e-04	0.000169	0.002053	0.000166	0.000175	0.000274	0.000735	-2.341790e-05	...	0.000132	-0.000084	-0.000136	0.000279	0.000043
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000022	-0.000002	-3.833980e-07	-0.000024	0.000043	-0.000010	0.000003	0.000144	-0.000134	3.607130e-05	...	0.000043	0.000013	0.000032	0.000019	0.000005
97	0.000088	0.000076	8.209618e-05	0.000070	0.000133	0.000091	0.000091	0.000121	0.000128	5.161785e-05	...	0.000106	0.000032	0.000022	0.000121	0.000015
98	0.000122	0.000092	7.261594e-05	0.000028	0.000311	0.000080	0.000084	0.000124	0.000289	1.159513e-04	...	0.000234	0.000053	0.000054	0.000164	0.000018
99	0.000087	0.000044	2.694988e-05	0.000039	0.000180	0.000043	0.000042	0.000100	0.000133	1.217576e-04	...	0.000257	0.000065	0.000043	0.000086	0.000030
100	0.000087	0.000077	6.798663e-05	0.000112	0.000150	0.000083	0.000085	0.000139	0.000148	3.274181e-05	...	0.000086	0.000052	0.000005	0.000106	0.000007

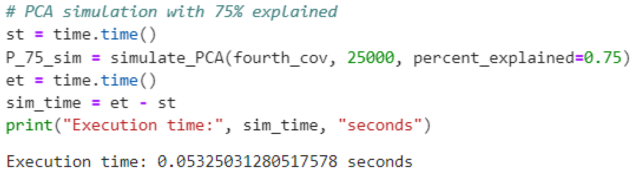
And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:



```
diff = P_100_sim_cov.to_numpy() - fourth_cov
F_Norm(diff)

6.583789723131338e-08
```

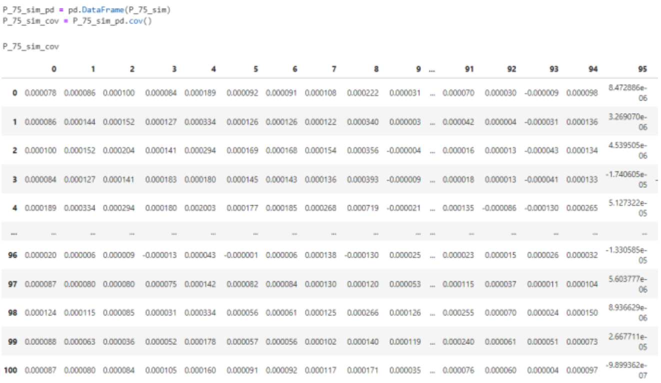
c. PCA with 75% Explained

I called "simulate_PCA" function with number of draws set to be 25,000 and "percent_explained" set to be 0.75, and timed this simulation process to check runtime, the runtime for direct simulation over the fourth matrix is:



```
# PCA simulation with 75% explained
st = time.time()
P_75_sim = simulate_PCA(fourth_cov, 25000, percent_explained=0.75)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")

Execution time: 0.05325031280517578 seconds
```

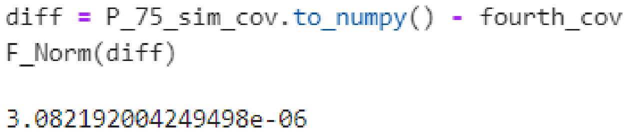
Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):



```
P_75_sim_pd = pd.DataFrame(P_75_sim)
P_75_sim_cov = P_75_sim_pd.cov()

P_75_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95
0	0.000078	0.000086	0.000100	0.000084	0.000189	0.000092	0.000091	0.000108	0.000222	0.000031	...	0.000070	0.000030	-0.000009	0.000098	8.472886e-06
1	0.000086	0.000144	0.000152	0.000127	0.000334	0.000126	0.000126	0.000122	0.000340	0.000003	...	0.000042	0.000004	-0.000031	0.000136	3.269070e-06
2	0.000100	0.000152	0.000204	0.000141	0.000294	0.000169	0.000168	0.000154	0.000356	-0.000004	...	0.000016	0.000013	-0.000043	0.000134	4.539505e-06
3	0.000084	0.000127	0.000141	0.000183	0.000180	0.000145	0.000143	0.000136	0.000393	-0.000009	...	0.000018	0.000013	-0.000041	0.000133	-1.740605e-05
4	0.000189	0.000334	0.000294	0.000180	0.002003	0.000177	0.000185	0.000268	0.000719	-0.000021	...	0.000135	-0.000086	-0.000130	0.000265	5.127322e-05
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000020	0.000006	0.000009	-0.000013	0.000043	-0.000001	0.000006	0.000138	-0.000130	0.000025	...	0.000023	0.000015	0.000026	0.000032	-1.330585e-05
97	0.000087	0.000080	0.000080	0.000075	0.000142	0.000082	0.000084	0.000130	0.000120	0.000053	...	0.000115	0.000037	0.000011	0.000104	5.603777e-06
98	0.000124	0.000115	0.000085	0.000031	0.000334	0.000056	0.000061	0.000125	0.000266	0.000126	...	0.000255	0.000070	0.000024	0.000150	8.936629e-06
99	0.000088	0.000063	0.000036	0.000052	0.000178	0.000057	0.000056	0.000102	0.000140	0.000119	...	0.000240	0.000061	0.000051	0.000073	2.667711e-05
100	0.000087	0.000080	0.000084	0.000105	0.000160	0.000091	0.000092	0.000117	0.000171	0.000035	...	0.000076	0.000060	0.000004	0.000097	-9.899362e-07

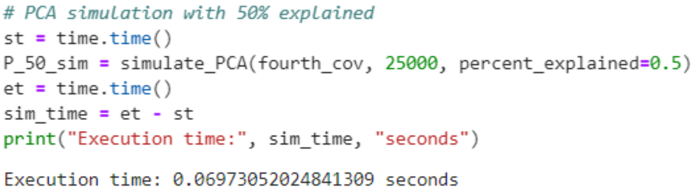
And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:



```
diff = P_75_sim_cov.to_numpy() - fourth_cov
F_Norm(diff)

3.082192004249498e-06
```

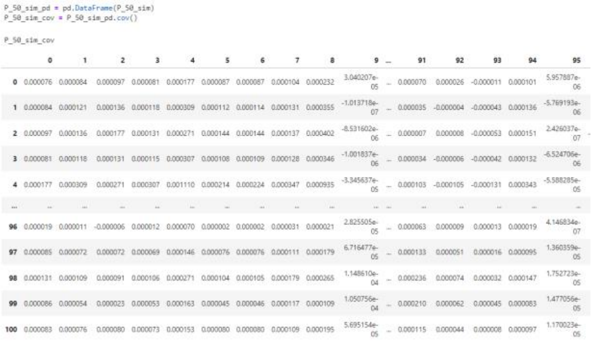
d. PCA with 50% Explained

I called "simulate_PCA" function with number of draws set to be 25,000 and "percent_explained" set to be 0.5, and timed this simulation process to check runtime, the runtime for direct simulation over the fourth matrix is:



```
# PCA simulation with 50% explained
st = time.time()
P_50_sim = simulate_PCA(fourth_cov, 25000, percent_explained=0.5)
et = time.time()
sim_time = et - st
print("Execution time:", sim_time, "seconds")

Execution time: 0.06973052024841309 seconds
```

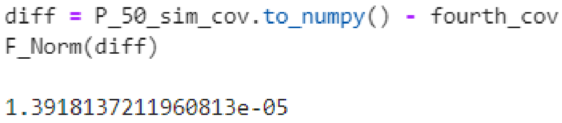
Then, I calculated the covariance matrix for the resulting matrix from the simulation (part of the matrix due to too many items):



```
P_50_sim_pd = pd.DataFrame(P_50_sim)
P_50_sim_cov = P_50_sim_pd.cov()

P_50_sim_cov
```

	0	1	2	3	4	5	6	7	8	9	...	91	92	93	94	95
0	0.000076	0.000084	0.000097	0.000081	0.000177	0.000087	0.000087	0.000104	0.000232	3.040207e-05	...	0.000070	0.000026	-0.000011	0.000101	5.957887e-06
1	0.000084	0.000121	0.000136	0.000118	0.000309	0.000112	0.000114	0.000131	0.000355	-1.013718e-07	...	0.000035	-0.000004	-0.000043	0.000136	-5.769193e-06
2	0.000097	0.000136	0.000177	0.000131	0.000271	0.000144	0.000144	0.000137	0.000402	-8.531602e-06	...	0.000007	0.000008	-0.000053	0.000151	2.426037e-07
3	0.000081	0.000118	0.000131	0.000115	0.000307	0.000108	0.000109	0.000128	0.000346	-1.001837e-06	...	0.000034	-0.000006	-0.000042	0.000132	-6.524706e-06
4	0.000177	0.000309	0.000271	0.000307	0.001110	0.000214	0.000224	0.000347	0.000935	-3.345637e-05	...	0.000103	-0.000105	-0.000131	0.000343	-5.588285e-05
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	0.000019	0.000011	-0.000006	0.000012	0.000070	0.000002	0.000002	0.000031	0.000021	2.825505e-05	...	0.000063	0.000009	0.000013	0.000019	4.146834e-07
97	0.000085	0.000072	0.000072	0.000069	0.000146	0.000076	0.000076	0.000111	0.000179	6.716477e-05	...	0.000133	0.000051	0.000016	0.000095	1.360359e-05
98	0.000131	0.000109	0.000091	0.000106	0.000271	0.000104	0.000105	0.000179	0.000265	1.148610e-04	...	0.000236	0.000074	0.000032	0.000147	1.752723e-05
99	0.000086	0.000054	0.000023	0.000053	0.000163	0.000045	0.000046	0.000117	0.000109	1.050756e-04	...	0.000210	0.000062	0.000045	0.000083	1.477056e-05
100	0.000083	0.000076	0.000080	0.000073	0.000153	0.000080	0.000080	0.000109	0.000195	5.695154e-05	...	0.000115	0.000044	0.000008	0.000097	1.170023e-05

And I computed the Frobenius norm to show the difference between the simulated covariance matrix and the input matrix:



```
diff = P_50_sim_cov.to_numpy() - fourth_cov
F_Norm(diff)

1.3918137211960813e-05
```

Comparison of Frobenius Norm and Runtime

Across four covariance matrices we generated through various ways, the pattern is generally the same: direct simulation takes longest time simulating systems and processes, while it results in covariance matrices that are nearest, or most accurate, comparing to input matrices; PCA with 100% explained comes next: it takes a little longer with sacrifice of little accuracy (note: in theory, direct simulation should be the same as PCA with 100% explained, but in my simulations, they are close but not the same – probable reasons could be: 1. floating points errors; 2. the way we check if required percent of variance explained is met); PCA with 75% explained is the second most efficient simulation, but its output more deviated from the input matrix from previous two simulations; PCA with 50% explained comes last – it takes the shortest time to generate the most deviated output. For example, when input matrix is exponentially weighted covariance matrix (the first of four matrices used in this problem), Frobenius norms and runtime of each simulation (in four decimal places) are shown below:

Attributes Type of Sim.	Runtime (s)	Frobenius Norm
Direct Simulation	0.1599	5.1050e-08
PCA with 100% Explained	0.0918	5.1570e-08
PCA with 75% Explained	0.0307	2.8817e-06
PCA with 50% Explained	0.0278	1.7663e-05

Conclusion

From results above, as percent of variance explained, lamda as defined in my programs, in PCA simulations increases, resulting matrices become less deviated from original ones (results are more accurate and reliable), but runtime of simulations also increases gradually. When direct simulation is used rather than PCA, results are generally more accurate but time-consuming than any PCA processes (though, in theory, direct simulations should be the same as PCA process with 100% of variance explained). Overall, we may conclude that there is a positive relationship between runtime and accuracy - it usually takes longer to generate more accurate simulations.