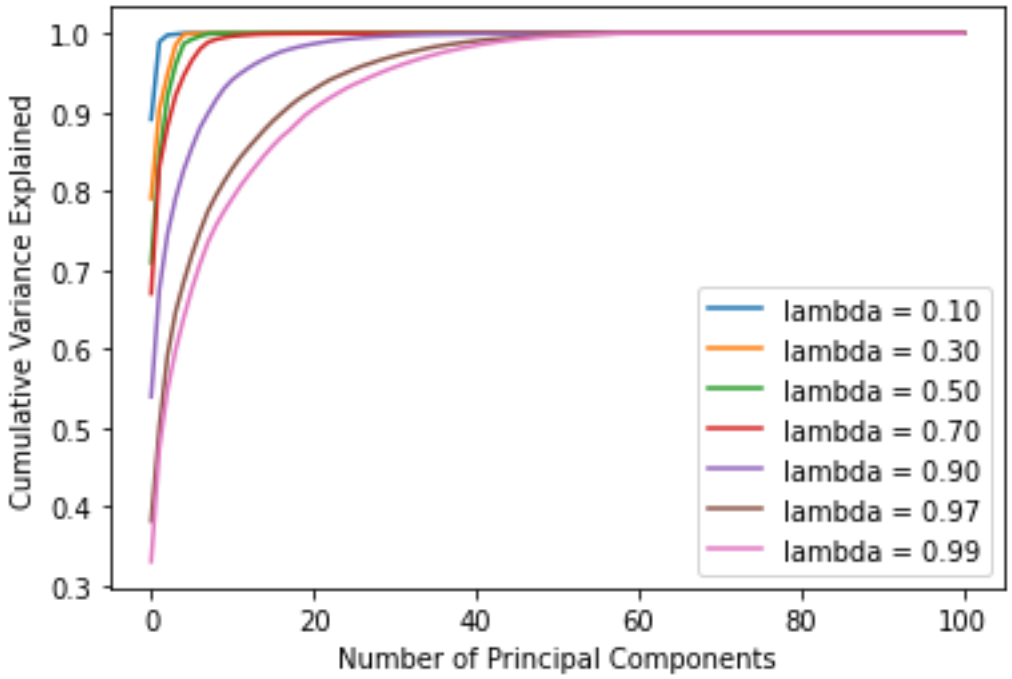
Problem 1

To calculate the exponentially weighted covariance matrix, I firstly initialized a covariance matrix with zeros. Then I looped through each asset: calculate the weights, calculate the mean of each asset, and calculate the weighted variance. The exponentially weighted covariance matrix with a lambda of 0.97 is:

```
[[8.41106909e-05 1.06945662e-04 1.21760871e-04 ... 1.25484463e-04
  8.11331555e-05 8.61130395e-05]
 [1.06945662e-04 2.68752303e-04 1.97531665e-04 ... 1.15658764e-04
  3.74977522e-05 8.22220854e-05]
 [1.21760871e-04 1.97531665e-04 2.91157502e-04 ... 8.30278956e-05
  3.31844912e-05 7.34713753e-05]
 ...
 [1.25484463e-04 1.15658764e-04 8.30278956e-05 ... 7.47889224e-04
  2.68371109e-04 2.00639601e-04]
 [8.11331555e-05 3.74977522e-05 3.31844912e-05 ... 2.68371109e-04
  3.08241679e-04 8.21009546e-05]
 [8.61130395e-05 8.22220854e-05 7.34713753e-05 ... 2.00639601e-04
  8.21009546e-05 2.62692778e-04]]
```

Following that, I set the lambdas to a list of values, and plotted the cumulative variance explained by each eigenvalue for each lambda chosen:



With a smaller lambda, each eigenvalue is forced to explain as little variance as possible, so the cumulative variance explained immediately reached to 1.0. If we provide the model with a larger lambda, we are taking in more eigenvalues, which means that we are using a more diverse portfolio with a larger number stock. So, the model would need more principal components to explain all the cumulative variance.

The effect of varying lambda on the covariance matrix is indirect. The covariance

matrix is used to calculate the eigenvalues and eigenvectors that define the principal components. Hence, by altering the magnitude of lambda, we are changing the eigenvalues needed to represent the data. A larger lambda could make the covariance matrix smoother and more stable. In general, lambda is used to control the selection of components and the amount of variance explained by each component, rather than directly modifying the covariance matrix.

Problem 2

The chol_psd function takes in a PSD matrix and return a lower-triangular matrix which allows us to easily manipulate the eigenvalues and analyze data. I firstly defined a function named chol_psd that takes a square matrix A as input and returns the lower triangular matrix root. The number of columns in the matrix A is stored in the variable n. A zero matrix with the same shape as A is created and stored in the root variable.

Then the code loops over the columns of the matrix A, from 0 to n-1. Within the loop, a variable s is initialized to 0.0. If the current column index i is greater than 0, s is updated to the dot product of the first i rows of the ith column of root with itself. A variable temp is calculated as the difference between the ith diagonal element of A and s. If temp is close to 0 (within a tolerance of 1e-8), it is set to 0.0. The square root of temp is then stored in the ith diagonal element of root. If the ith diagonal element of root is 0.0, this means that temp is 0 and A has a zero eigenvalue.

The code sets all elements in the ith column of root to 0.0 for columns i+1 to n-1. If the ith diagonal element of root is not 0.0, the code updates the off-diagonal elements in the ith column of root. The code calculates a factor ir as the inverse of the ith diagonal element of root. The off-diagonal elements are updated using the formula root[j][i] = (A[j][i] - s) * ir where j is the row index and ranges from i+1 to n-1.

Here is an example output of chol_psd:

```
n = 5
a = np.ones((n,n)) * 0.9
for i in range(n):
    a[i,i] = 1.0
a[0,1] = 1.0
a[1,0] = 1.0

chol_psd(a)
#nearest_psd(a)
```

```
array([[1.        , 0.        , 0.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        , 0.        , 0.        ],
       [0.9       , 0.        , 0.43588989, 0.        , 0.        ],
       [0.9       , 0.        , 0.20647416, 0.38388595, 0.        ],
       [0.9       , 0.        , 0.20647416, 0.12339191, 0.36351459]])
```

The near_psd function takes a matrix "a" and an optional argument "epsilon" and returns a positive semi-definite matrix "out". If the input matrix is not positive semi-definite, the function attempts to produce a nearby positive semi-definite matrix by adjusting its eigenvalues.

The function starts by initializing the number of rows in the input matrix and setting the output matrix equal to the input matrix. If the input matrix is a covariance matrix, the code calculates the correlation matrix by dividing each element on the diagonal by the square root of the corresponding element in the diagonal.

Then I calculate the singular value decomposition of the input or correlation matrix and obtain the eigenvectors and eigenvalues. The eigenvalues a then replaced with the maximum of each eigenvalue and the epsilon value (which is a small positive number). The code then calculates the T matrix and the B matrix, which are used to adjust the input matrix so that it becomes positive semi-definite.

Finally, the function returns the positive semi-definite matrix. If the input matrix was a covariance matrix, the function restores the variance by multiplying the positive semi-definite matrix by the inverse of the square root of the diagonal of the original covariance matrix.

Here is an example output of near_psd:

```
[44] #near_psd test
     n = 500
     a = np.ones((n,n)) * 0.9
     for i in range(n):
         a[i,i] = 1.0
     a[0,1] = 0.7357
     a[1,0] = 0.7357

     near_psd(a)
```

```
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.        ]])
```

The higham algorithm is also used to find the nearest positive semi-definite matrix. The first step is to compute the eigenvalues and eigenvectors of the input matrix A using the eigh function from numpy.linalg. The code then checks if there are any negative eigenvalues in the eigenvalue array eigvals. If there are, it computes the square root of the eigenvalues, which are positive and stores it in the matrix D.

Next, I compute a matrix V as the product of the eigenvectors and the square root of D. The code then updates the input matrix A to be equal to V times its transpose V.T. This is the core step of the algorithm, as it ensures that the new matrix A has all its eigenvalues positive.

The code repeats the process from step 2 to step 4, until all the eigenvalues of the matrix A are positive, or until a maximum number of iterations (num_iter) has been reached.

Finally, the code returns the symmetric part of the matrix A which is equal to (A + A.T) / 2.

Here is an example output of Higham function:

```
[46] #Higham  test
     nearest_psd_correlation(a,  10)
```

```
array([[1.03169433, 0.76739433, 0.89987276, ..., 0.89987276, 0.89987276,
        0.89987276],
       [0.76739433, 1.03169433, 0.89987276, ..., 0.89987276, 0.89987276,
        0.89987276],
       [0.89987276, 0.89987276, 1.00000051, ..., 0.90000051, 0.90000051,
        0.90000051],
       ...,
       [0.89987276, 0.89987276, 0.90000051, ..., 1.00000051, 0.90000051,
        0.90000051],
       [0.89987276, 0.89987276, 0.90000051, ..., 0.90000051, 1.00000051,
        0.90000051],
       [0.89987276, 0.89987276, 0.90000051, ..., 0.90000051, 0.90000051,
        1.00000051]])
```

Then I confirmed that these two algorithms work:

```
[47] #Confirm  PSD
     chol_psd(near_psd(a))
```

```
array([[ 1.00000000e+00,  0.00000000e+00,  0.00000000e+00, ...,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 7.43819469e-01,  6.68380578e-01,  0.00000000e+00, ...,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 8.85942373e-01,  3.39568795e-01,  3.15910028e-01, ...,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       ...,
       [ 8.85942373e-01,  3.39568795e-01, -6.35633859e-04, ...,
         2.58198824e-01,  0.00000000e+00,  0.00000000e+00],
       [ 8.85942373e-01,  3.39568795e-01, -6.35633859e-04, ...,
        -1.29099412e-01,  2.23606741e-01,  0.00000000e+00],
       [ 8.85942373e-01,  3.39568795e-01, -6.35633859e-04, ...,
        -1.29099412e-01, -2.23606741e-01,  0.00000000e+00]])
```

```
chol_psd(nearest_psd_correlation(a,  10))
```

```
array([[ 1.01572355e+00,  0.00000000e+00,  0.00000000e+00, ...,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 7.55514950e-01,  6.78889892e-01,  0.00000000e+00, ...,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 8.85942599e-01,  3.39568881e-01,  3.15910109e-01, ...,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       ...,
       [ 8.85942599e-01,  3.39568881e-01, -6.35634022e-04, ...,
         2.58198890e-01,  0.00000000e+00,  0.00000000e+00],
       [ 8.85942599e-01,  3.39568881e-01, -6.35634022e-04, ...,
        -1.29099445e-01,  2.23606798e-01,  0.00000000e+00],
       [ 8.85942599e-01,  3.39568881e-01, -6.35634022e-04, ...,
        -1.29099445e-01, -2.23606798e-01,  1.99764502e-06]])
```

Now I compare the accuracy and run-time of these two:

```
[49] #Frobenius  norm
     def  Frobenius(A):
         return  np.linalg.norm(A,  ord='fro')

     Frobenius(a)
```

```
450.10494610591377
```

```
[50] Frobenius(near_psd(a))
```

```
450.0494394140364
```

```
[51] Frobenius(nearest_psd_correlation(a,  10))
     #Even  when  n  is  large,  highman  is  still  more  accurate
```

```
450.10494160642054
```

```
#large N
n = 5000
a = np.ones((n,n)) * 0.9
for i in range(n):
    a[i,i] = 1.0
a[0,1] = 0.7357
a[1,0] = 0.7357
```

```
[53] #run time
import time
start = time.time()
near = near_psd(a)
node1 = time.time()
hig = nearest_psd_correlation(a, 10)
node2 = time.time()

print("Run time of near_psd is:", node1-start)
print("Run time of highman is:", node2-node1)
print("Their difference is:", node2-2*node1+start)
```

```
Run time of near_psd is: 76.70982480049133
Run time of highman is: 90.22025060653687
Their difference is: 13.510425806045532
```

As the results indicate, the Higham algorithm takes longer time to run but has a higher accuracy. The reason might be that it iterates the result over and over again to make sure the output is PSD. Here I set the iteration time to be 10, but if I encounter a more complex matrix, I might need more than 10 times of iteration and much longer time to run than near_psd. To conclude, the trade-off between near_psd and Higham is the accuracy and rum-time.

Problem 3

To implement exponentially weighted variance, I firstly calculate the weighting for each row in the data using the formula (1 - lambd) * (lambd ** (n_rows - i - 1)). Then I calculate the variance of each column in the data using the weighted mean. For each column j, the mean of the column is calculated using np.mean(data.iloc[:, j]). The variance is calculated as the weighted sum of the squared deviations from the mean for each row i, using the formula weight[i] * deviation**2. The resulting variance for each column is stored in the variance_vector numpy array.

To implement exponentially weighted correlation, I want the code take in a covariance matrix and a variance vector. The first step is to calculate the standard deviation for each feature using the variance vector. Next, the exponential weighted correlation matrix is calculated as the ratio of the covariance matrix and the outer product of the standard deviation vector with itself. Finally, the calculated correlation matrix is returned as the output of the function.

Combining these two, I could get four different kinds of covariance matrix:

```
[ ]  #1
     lambda_value = 0.0
     var = exponential_weighted_variance(data, lambda_value)
     cor = data.corr()
```

```
[ ]  cov1 = generate_covariance_matrix(cor, var)
```

```
[ ]  #2
     lambda_value = 0.97
     var_weighted = exponential_weighted_variance(data, lambda_value)
     cor = data.corr()
```

```
[ ]  cov2 = generate_covariance_matrix(cor, var_weighted)
```

```
[ ]  #3
     lambda_value = 0.0
     var = np.var(data, axis=0)
     cor_weighted = exponential_weighted_correlation(ewcov, var)
```

```
[ ]  cov3 = generate_covariance_matrix(cor, var)
```

```
[ ]  #4
     lambda_value = 0.97
     var_weighted = exponential_weighted_variance(data, lambda_value)
     cor_weighted = exponential_weighted_correlation(ewcov, var_weighted)
```

```
[ ]  cov4 = generate_covariance_matrix(cor_weighted, var_weighted)
```

```
[130] Frobenius(np.cov(data,  rowvar=False))
```

```
0.011968397602104986
```

Then, simulating 25,000 draws from each covariance matrix using:

- Direct Simulation
  - This can be done by multiplying a PSD matrix with a standardized random data. Comparing their Frobenius norm:

```
[132] print(Frobenius(d_cov1))
      print(Frobenius(d_cov2))
      print(Frobenius(d_cov3))
      print(Frobenius(d_cov4))
```

```
0.01357834736756668
0.010063694016827716
0.011788343591917282
0.012796420482215224
```

- PCA with 100% explained.

```
print(Frobenius(pca100_cov1))
print(Frobenius(pca100_cov2))
print(Frobenius(pca100_cov3))
print(Frobenius(pca100_cov4))
```

```
0.013461787317309947
0.010071476963148546
0.01183368497059213
0.012803174926308173
```

- PCA with 75% explained.

```
print(Frobenius(pca75_cov1))
print(Frobenius(pca75_cov2))
print(Frobenius(pca75_cov3))
print(Frobenius(pca75_cov4))
```

```
0.01358175132489859
0.009913226482298901
0.011545249770079943
0.01245550914926372
```

- PCA with 50% explained.

```
print(Frobenius(pca50_cov1))
print(Frobenius(pca50_cov2))
print(Frobenius(pca50_cov3))
print(Frobenius(pca50_cov4))
```

```
0.012946793685218238
0.009659137643099057
0.01138038064107553
0.012285894366920498
```

- These three can be done by a simulate_pca function. It performs a simulation of a principal component analysis (PCA) of a given input matrix a. to be specific, the function performs an eigenvalue decomposition of the input matrix a and sorts the eigenvalues in decreasing order.

The run-time for all of these is really small. It takes less than one second to run each simulation. So, I would prefer using more eigenvalues or direct simulation with input of original variance and exponentially weighted correlation, because their Frobenius norms are closer to the original data.