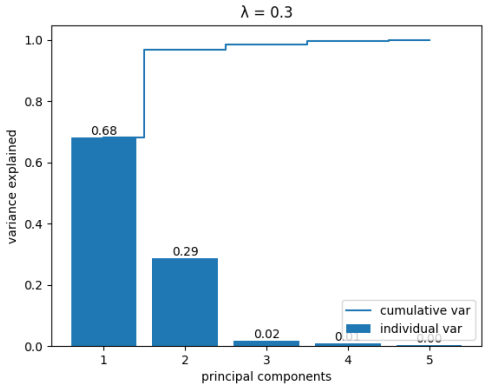
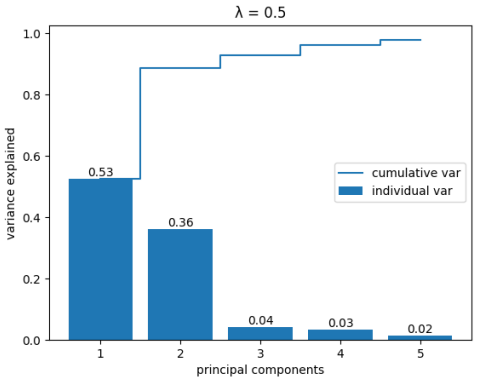
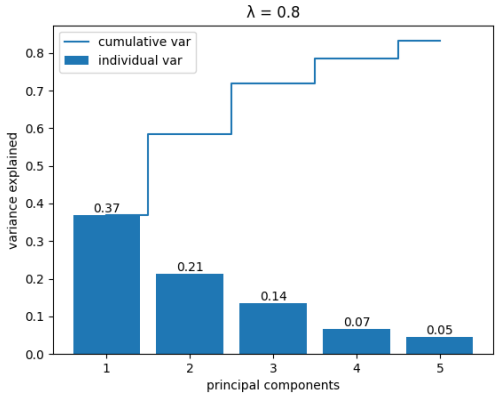
Project ----- Week03
Tina Zhao

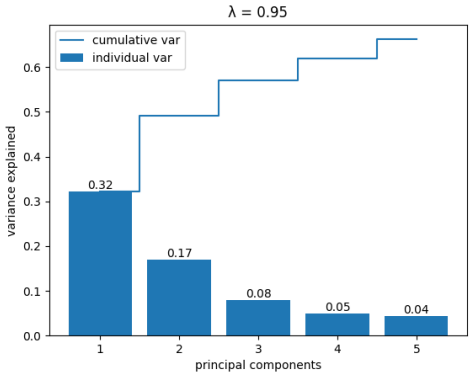
Problem 1

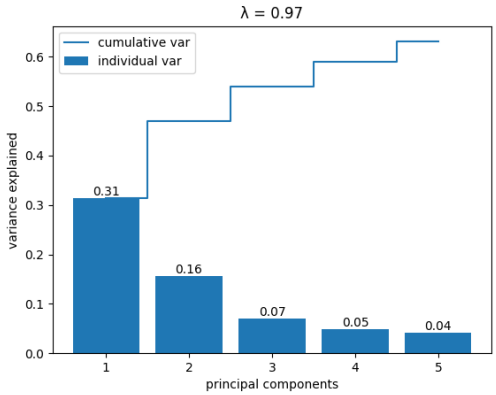
Generating exponentially weighted covariance matrix with different λ and use them to do PCA, plotting the explaining ratio of the first 5 principal components.

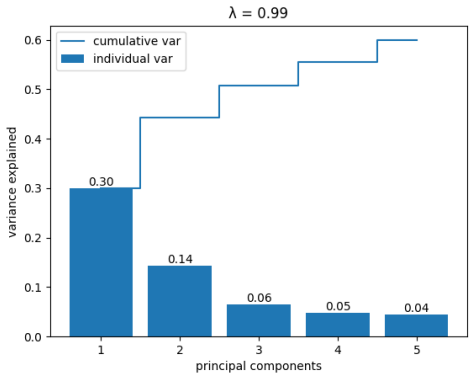












It can be noticed that as λ gets larger, the first few values explain less of the total variance.

A possible explanation for the observed result:

A smaller λ means we are giving more weight to the latest data point, making the information in the latest data much more important and captured than data in the past when calculating statistic data. This causes the "information" in our covariance matrix more "concentrated". As a result, when we decompose the covariance matrix using PCA, we will see that a larger proportion of the variance explained by the first few key components.

Problem 2

Implement chol_psd() in python

Use the following matrix for testing.

```
n = 5
sigma = np.zeros([n, n]) + 0.9
for i in range(5):
    sigma[i, i] = 1.0

root = np.zeros([n, n])
```

```
>>> sigma
array([[1. , 1. , 0.9, 0.9, 0.9],
       [1. , 1. , 0.9, 0.9, 0.9],
       [0.9, 0.9, 1. , 0.9, 0.9],
       [0.9, 0.9, 0.9, 1. , 0.9],
       [0.9, 0.9, 0.9, 0.9, 1. ]])
```

Do the decomposition and check the result

```
chol_psd(root, sigma)
np.matmul(root, np.transpose(root))
```

```
>>> np.matmul(root, np.transpose(root))
array([[1. , 1. , 0.9, 0.9, 0.9],
       [1. , 1. , 0.9, 0.9, 0.9],
       [0.9, 0.9, 1. , 0.9, 0.9],
       [0.9, 0.9, 0.9, 1. , 0.9],
       [0.9, 0.9, 0.9, 0.9, 1. ]])
```

Implement near_psd() in python

Use the following matrix for testing.

```
n=500
sigma = np.zeros([n, n]) + 0.9
for i in range(n):
    sigma[i, i] = 1.0

#make the matrix non-definite
sigma[0,1] = 0.7357
sigma[1,0] = 0.7357
```

```
>>> sigma
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.    ]])
```

The matrix is not PSD

```
>>> sum(eigh(sigma)[0]>-1e-8)
499
```

Adjust the matrix with the function:

```
a = near_psd(sigma)
```

```
>>> 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.        ]])
```

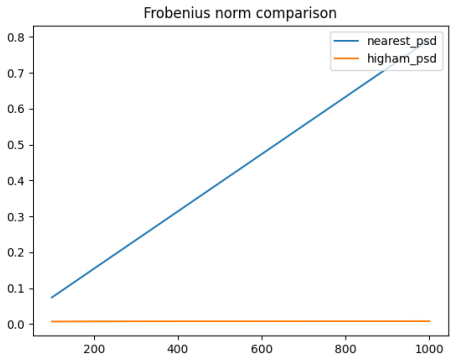
The adjusted matrix is now PSD

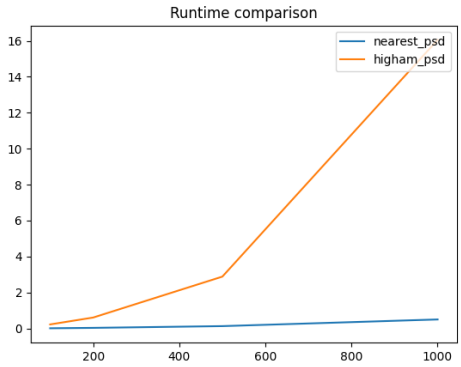
```
>>> sum(eigh(a)[0]>-1e-8)
500
```

Implement Higham's method in python

Compare the accuracy and runtime for the near_psd function and the Higham's method.

```
n =  100
       Runtime: nearest_psd:  0.015127897262573242 higham:  0.24402427673339844
Frobenius Norm: nearest_psd:  0.0744152064348294 higham:  0.007164373937354317
n =  200
       Runtime: nearest_psd:  0.02797412872314453 higham:  0.6522488594055176
Frobenius Norm: nearest_psd:  0.1542645861963699 higham:  0.007699288393155556
n =  500
       Runtime: nearest_psd:  0.1173563003540039 higham:  2.8030197620391846
Frobenius Norm: nearest_psd:  0.3937846835005644 higham:  0.008036762460431888
n =  1000
       Runtime: nearest_psd:  0.48015689849853516 higham:  12.01880168914795
Frobenius Norm: nearest_psd:  0.7929738934321975 higham:  0.008152133275712524
```





Comparing to Rebonato and Jackel's meghod, Higham's method results in a matrix that is much closer to the original non-psd matrix, especially as n growth, Higham's accuracy does not drop much while the near_psd method is gives a much higher Frobenius norm.

The trade off is also clear, the runtime for Higham's method is larger than Rebonato and Jackel's. And it shows a non-linear and higher growth rate in the runtime as n gets bigger.

Problem 3

Generating 4 different covariance matrices, and draw random samples from them with different explained ratio. Documenting the function's runtime and the Frobenius Norm between the covariance matrix of simulated data and the real data. (For calculating the norm, I first thought about using matching methods. Which means if the covariance matrix is generated using EW correlation or EW variance, I will calculate the covariance matrix of my simulated data using the same methods. But then I thought as the simulated data is randomly picked, it doesn't mean anything if I give data in the back more weights. So for the simulated data, I'm all using person correlation and simple variance.)

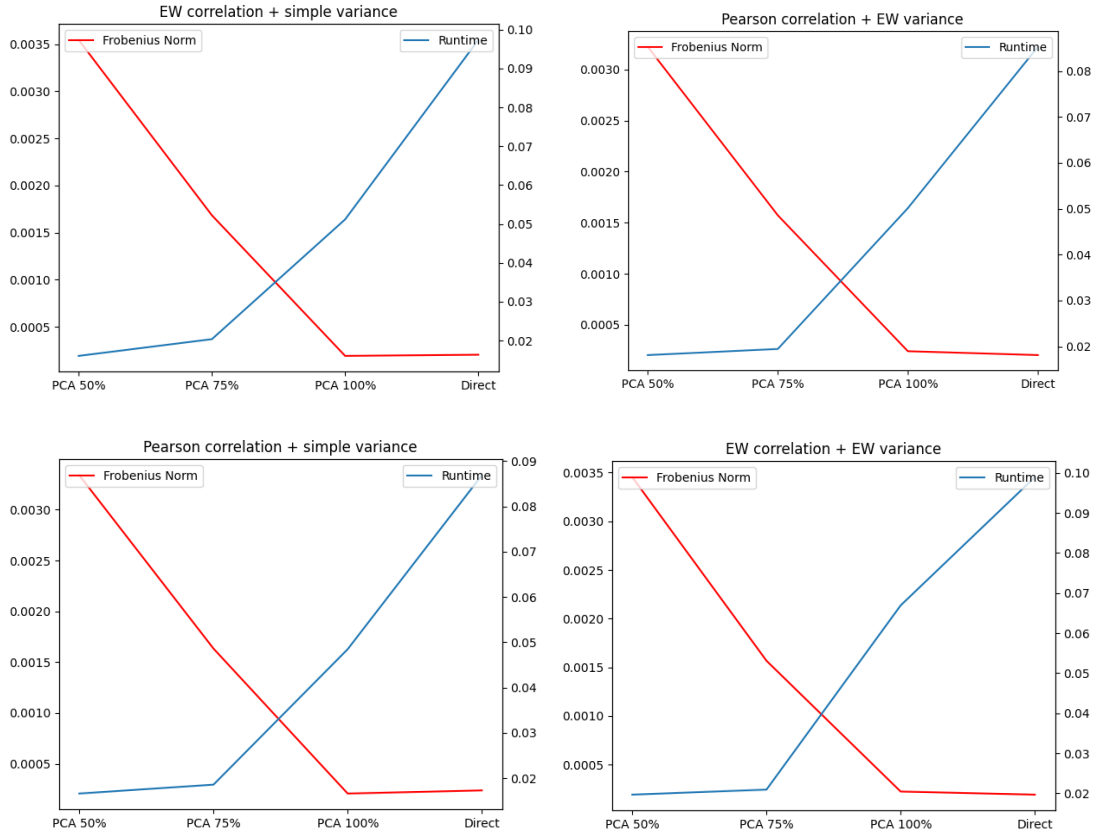
Below is the reported result.

```
--------Pearson correlation + simple variance-------
distance from real covariance(50% explained):  0.003338176649647851
runtime (50% explained): 0.016621828079223633
distance from real covariance(75% explained):  0.0016336442115893997
runtime (75% explained): 0.018581151962280273
distance from real covariance(100% explained):  0.00020766955547365463
runtime (100% explained): 0.04848051071166992
distance from real covariance(direct):  0.0002383645301513912
runtime (direct): 0.08688688278198242
--------Pearson correlation + EW variance-------
distance from real covariance(50% explained):  0.003222800640781367
runtime (50% explained): 0.01812124252319336
distance from real covariance(75% explained):  0.0015721489892915298
runtime (75% explained): 0.01944875717163086
distance from real covariance(100% explained):  0.00024084177113002859
runtime (100% explained): 0.05012941360473633
distance from real covariance(direct):  0.0002031490779350886
runtime (direct): 0.08527541160583496
--------EW correlation + simple variance-------
distance from real covariance(50% explained):  0.0035444391315474
runtime (50% explained): 0.016039133071899414
distance from real covariance(75% explained):  0.001683858167319884
runtime (75% explained): 0.020343542098999023
distance from real covariance(100% explained):  0.0001926484639393386
runtime (100% explained): 0.051154375076293945
distance from real covariance(direct):  0.00020502259467118633
runtime (direct): 0.0973355770111084
--------EW correlation + EW variance-------
distance from real covariance(50% explained):  0.00345383230610893
runtime (50% explained): 0.019693613052368164
distance from real covariance(75% explained):  0.001567638157728435
runtime (75% explained): 0.02095174789428711
distance from real covariance(100% explained):  0.00022370937166578953
runtime (100% explained): 0.06691908836364746
distance from real covariance(direct):  0.00019201033755592288
runtime (direct): 0.09891819953918457
```

It is obvious that a higher explained ratio means a smaller distance between the covariance matrix of the simulation and the real data with the cost of a longer runtime. In

addition, PCA has a higher efficiency than direct simulation even when targeting 100% explanation.

Plotting the results to see more clearly:



There exists clear tradeoff between runtime and accuracy. On the other hand, the increase of accuracy (shows in the drop of the Frobenius norm) is around the same amount when changing PCA explained ratio from 50% to 75% and from 75% to 100%. But the runtime increases for the later one is much more significant than the previous. Maybe boosting accuracy from low to a moderate value is more time efficient than from a moderate accuracy to a relative high accuracy.