Linear Algebra Computer Assignment

School of Electrical and Computer Engineering, University of Tehran

Fardin Abbasi 810199456

Fall 2022

Contents

1. Algorithm implementation

1.1. Lagrange Interpolation

$$P_3(x) = L_0(x)f_0 + L_1(x)f_1 + L_2(x)f_2 + L_3(x)f_3$$

$$L_0(x) = \frac{(x-0.6)(x-1.03)(x-1.39)}{-0.6 \times -1.03 \times -1.39} = 1 - 3.35704\,x + 3.51572\,x^2 - 1.16414\,x^3$$

$$L_1(x) = \frac{x(x-1.03)(x-1.39)}{0.6(0.6-1.03)(0.6-1.39)} = 7.02\,x - 11.86\,x^2 + 4.9\,x^3$$

$$L_2(x) = \frac{x(x-0.6)(x-1.39)}{1.03(1.03-0.6)(1.03-1.39)} = -5.56\,x + 13.27\,x^2 - 6.67\,x^3$$

$$L_3(x) = \frac{x(x-0.6)(x-1.03)}{1.39(1.39-0.6)(1.39-1.03)} = 1.545\,x - 4.075\,x^2 + 2.5\,x^3$$

$$P_3(x) = 2.718 \times L_0(x) + 0.797 \times L_1(x) + 0.368 \times L_2(x) + 0.597 \times L_3(x)$$

$$P_4(x) = L_0(x)f_0 + L_1(x)f_1 + L_2(x)f_2 + L_3(x)f_3 + L_4(x)f_4$$

$$L_0(x) = \frac{(x-0.6)(x-1.03)(x-1.39)(x-1.76)}{-0.6 \times -1.03 \times -1.39 \times -1.76} = 1 - 3.9\,x + 5.4\,x^2 - 3.16\,x^3 + 0.67\,x^4$$

$$L_1(x) = \frac{x(x-1.03)(x-1.39)(x-1.76)}{0.6(0.6-1.03)(0.6-1.39)(0.6-1.76)} = 10.67\,x - 24.1\,x^2 + 17.71\,x^3 - 4.24\,x^4$$

$$L_2(x) = \frac{x(x-0.6)(x-1.39)(x-1.76)}{1.03(1.03-0.6)(1.03-1.39)(1.03-1.76)} = -12.65\,x + 37.38\,x^2 - 32.32\,x^3 + 8.62\,x^4$$

$$L_3(x) = \frac{x(x-0.6)(x-1.03)(x-1.76)}{1.39(1.39-0.6)(1.39-1.03)(1.39-1.76)} = 7.45\,x - 23.88\,x^2 + 23.22\,x^3 - 6.85\,x^4$$

$$L_4(x) = \frac{x(x-0.6)(x-1.03)(x-1.39)}{1.76(1.76-0.6)(1.76-1.03)(1.76-1.39)} = -1.56\,x + 5.24\,x^2 - 5.5\,x^3 + 1.81\,x^4$$

$$P_4(x) = 2.718 \times L_0(x) + 0.797 \times L_1(x) + 0.368 \times L_2(x) + 0.597 \times L_3(x) + 1.712 \times L_4(x)$$

$$P_5(x) = L_0(x)f_0 + L_1(x)f_1 + L_2(x)f_2 + L_3(x)f_3 + L_4(x)f_4 + L_5(x)f_5$$

$$L_0(x) = \frac{(x-0.6)(x-1.03)(x-1.39)(x-1.76)(x-2.09)}{-0.6 \times -1.03 \times -1.39 \times -1.76 \times -2.09} = -x + 4.4\,x^2 - 7.3\,x^3 + 5.75\,x^4 - 2.17\,x^5 + 0.31\,x^6$$

$$L_1(x) = \frac{x(x-1.03)(x-1.39)(x-1.76)(x-2.09)}{0.6(0.6-1.03)(0.6-1.39)(0.6-1.76)(0.6-2.09)} = 15.04\,x - 41.18\,x^2 + 41.22\,x^3 - 17.9\,x^4 + 2.85\,x^5$$

$$L_2(x) = \frac{x(x-0.6)(x-1.39)(x-1.76)(x-2.09)}{1.03(1.03-0.6)(1.03-1.39)(1.03-1.76)(1.03-2.09)} = -24.88\,x + 85.4\,x^2 - 98.7\,x^3 + 47.36\,x^4 - 8.11\,x^5$$

$$L_3(x) = \frac{x(x-0.6)(x-1.03)(x-1.76)(x-2.09)}{1.39(1.39-0.6)(1.39-1.03)(1.39-1.76)(1.39-2.09)} = 19.43\,x - 71.58\,x^2 + 90.36\,x^3 - 46.84\,x^4 + 8.55\,x^5$$

$$L_4(x) = \frac{x(x-0.6)(x-1.03)(x-1.39)(x-2.09)}{1.76(1.76-0.6)(1.76-1.03)(1.76-1.39)(1.76-2.09)} = -9.86\,x + 37.83\,x^2 - 50.52\,x^3 + 28.07\,x^4 - 5.5\,x^5$$

$$L_5(x) = \frac{x(x-0.6)(x-1.03)(x-1.39)(x-1.76)}{2.09(2.09-0.6)(2.09-1.03)(2.09-1.39)(2.09-1.76)} = 1.98\,x - 7.77\,x^2 + 10.75\,x^3 - 6.26\,x^4 + 1.31\,x^5$$

$$P_5(x) = 2.718 \times L_0(x) + 0.797 \times L_1(x) + 0.368 \times L_2(x) + 0.597 \times L_3(x) + 1.712 \times L_4(x) + 2.718 \times L_5(x)$$

$$P_6(x) = L_0(x)f_0 + L_1(x)f_1 + L_2(x)f_2 + L_3(x)f_3 + L_4(x)f_4 + L_5(x)f_5 + L_6(x)f_6$$

$$L_0(x) = \frac{(x-0.6)(x-1.03)(x-1.39)(x-1.76)(x-2.09)(x-2.29)}{-0.6 \times -1.03 \times -1.39 \times -1.76 \times -2.09 \times 2.29} = 1 - 4.84\,x + 9.22\,x^2 - 8.95\,x^3 + 4.69\,x^4 - 1.27\,x^5 + 0.14\,x^6$$

$$L_1(x) = \frac{x(x-1.03)(x-1.39)(x-1.76)(x-2.09)(x-2.29)}{0.6(0.6-1.03)(0.6-1.39)(0.6-1.76)(0.6-2.09)(0.6-2.29)} = 20.1\,x - 63.79\,x^2 + 79.08\,x^3 - 47.98\,x^4 + 14.27\,x^5 - 1.67\,x^6$$

$$L_2(x) = \frac{x(x-0.6)(x-1.39)(x-1.76)(x-2.09)(x-2.29)}{1.03(1.03-0.6)(1.03-1.39)(1.03-1.76)(1.03-2.09)(1.03-2.29)} = -45.3\,x + 175.3\,x^2 - 247.8\,x^3 + 164.8\,x^4 - 52.5\,x^5 + 6.4\,x^6$$

$$L_3(x) = \frac{x(x-0.6)(x-1.03)(x-1.76)(x-2.09)(x-2.29)}{1.39(1.39-0.6)(1.39-1.03)(1.39-1.76)(1.39-2.09)(1.39-2.29)} = 57.84\,x - 238.34\,x^2 + 362.05\,x^3 - 256.9\,x^4 + 86.3\,x^5 - 11.1\,x^6$$

$$L_4(x) = \frac{x(x-0.6)(x-1.03)(x-1.39)(x-2.09)(x-2.29)}{1.76(1.76-0.6)(1.76-1.03)(1.76-1.39)(1.76-2.09)(1.76-2.29)} = -42.83\,x + 183\,x^2 - 291.1\,x^3 + 217.7\,x^4 - 77.08\,x^5 + 10.42\,x^6$$

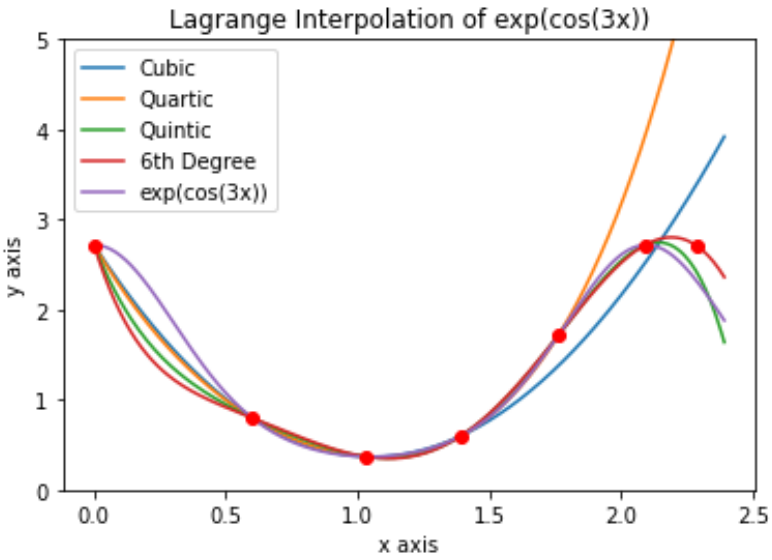
$$L_5(x) = \frac{x(x-0.6)(x-1.03)(x-1.39)(x-1.76)(x-2.29)}{2.09(2.09-0.6)(2.09-1.03)(2.09-1.39)(2.09-1.76)(2.09-2.29)} = 22.7\,x - 99.02\,x^2 + 162\,x^3 - 125.5\,x^4 + 46.36\,x^5 - 6.6\,x^6$$

$$L_6(x) = \frac{x(x-0.6)(x-1.03)(x-1.39)(x-1.76)(x-2.09)}{2.29(2.29-0.6)(2.29-1.03)(2.29-1.39)(2.29-1.76)(2.29-2.09)} = -4.33\,x + 19.06\,x^2 - 31.6\,x^3 + 25\,x^4 - 9.4\,x^5 + 1.37\,x^6$$

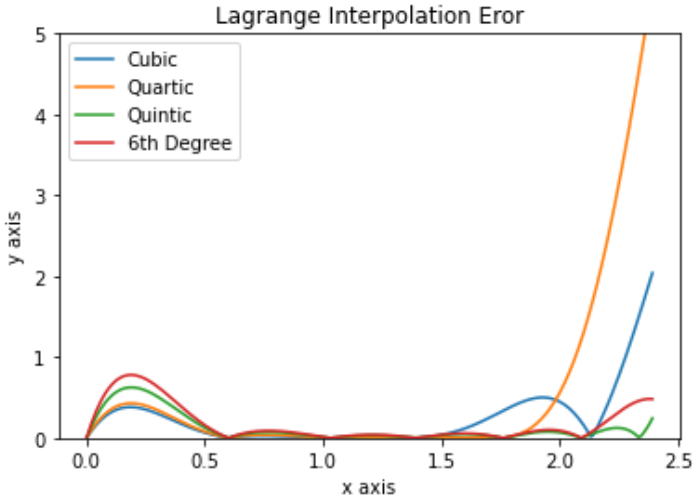
$$P_6(x) = 2.718 \times L_0(x) + 0.797 \times L_1(x) + 0.368 \times L_2(x) + 0.597 \times L_3(x) + 1.712 \times L_4(x) + 2.718 \times L_5(x) + 0.455 \times L_6(x)$$

1.1.2)

Lagrange polynomials for degrees of 3 to 6 are shown in addition to the main function:



1.1.3) Plot of absolute error is shown for different degrees:



1.1.4) Best estimation for n+1 points of data is n degree polynomial and in this case, the 6th degree has the lowest error.

LU Decomposition for Upper Hessenberg

1.2.1

```
def lu(A):
    [r,c] = np.shape(A)
    U = A.astype('float32')
    L = np.eye(r)
    for i in range(r-1):
        fac=U[i+1,i]/U[i,i]
        U[i+1,:]-=fac*U[i]
        L[i+1,i]=fac
    return L, U
H=np.matrix([[1,4,2,3],[3,4,1,7],[0,2,3,4],[0,0,1,3]])
lu(H)
```

```
(array([[ 1.       ,  0.       ,  0.       ,  0.       ],
        [ 3.       ,  1.       ,  0.       ,  0.       ],
        [ 0.       , -0.25     ,  1.       ,  0.       ],
        [ 0.       ,  0.       ,  0.5714286,  1.       ]]),
 matrix([[ 1.  ,  4.  ,  2.  ,  3.  ],
         [ 0.  , -8.  , -5.  , -2.  ],
         [ 0.  ,  0.  ,  1.75,  3.5 ],
         [ 0.  ,  0.  ,  0.  ,  1.  ]], dtype=float32))
```

1.2.2) Since the input matrix is upper Hessenberg, in the algorithm it only has to calculate the factor for just one row below so the computation cost is much lower!

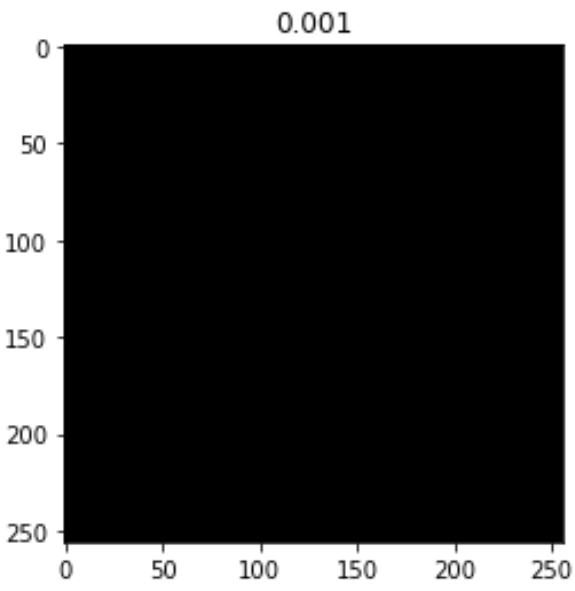
1.2.3) Since the given matrix is upper Hessenberg the computation cost is not relevant to the number of rows.

SVD and FFT Compression

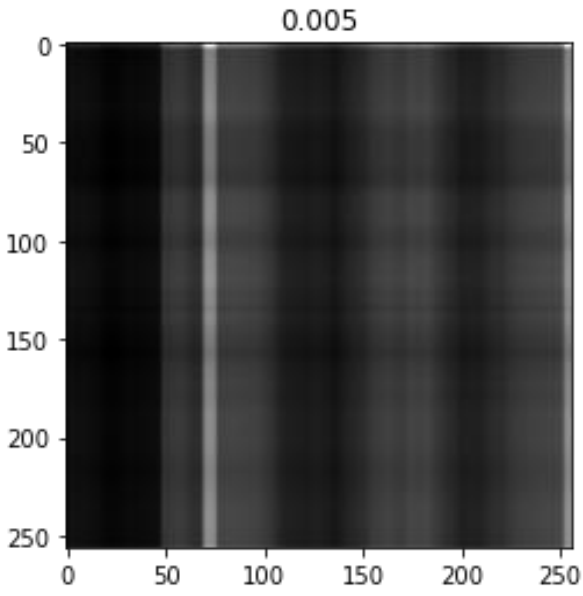
A)

SVD Compression:

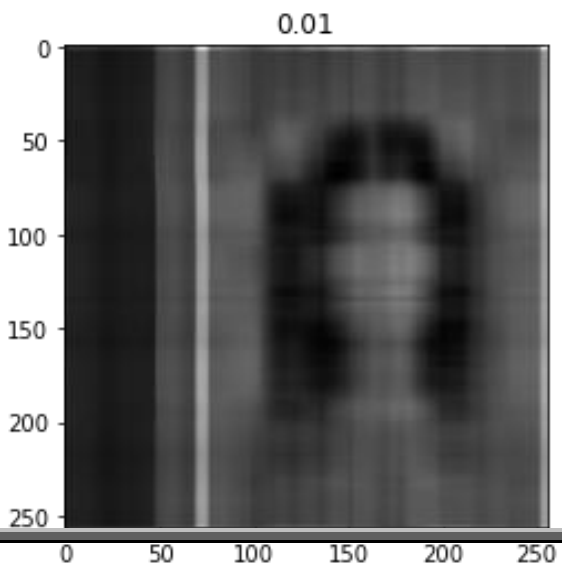
Ratio=0.1 %



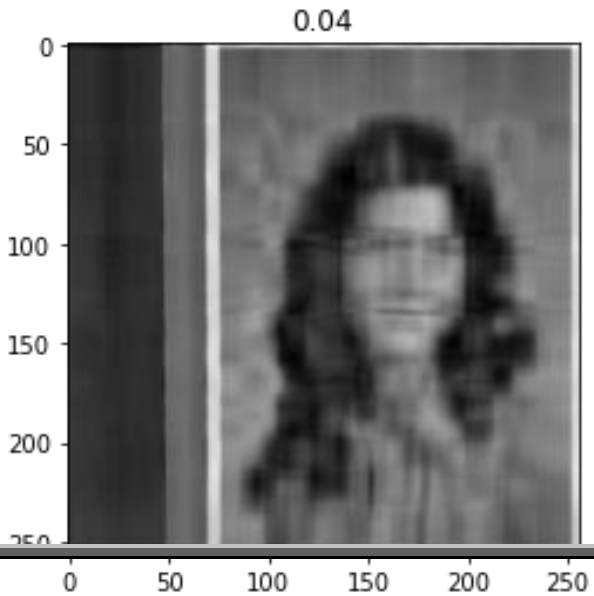
Ratio = 0.5 %



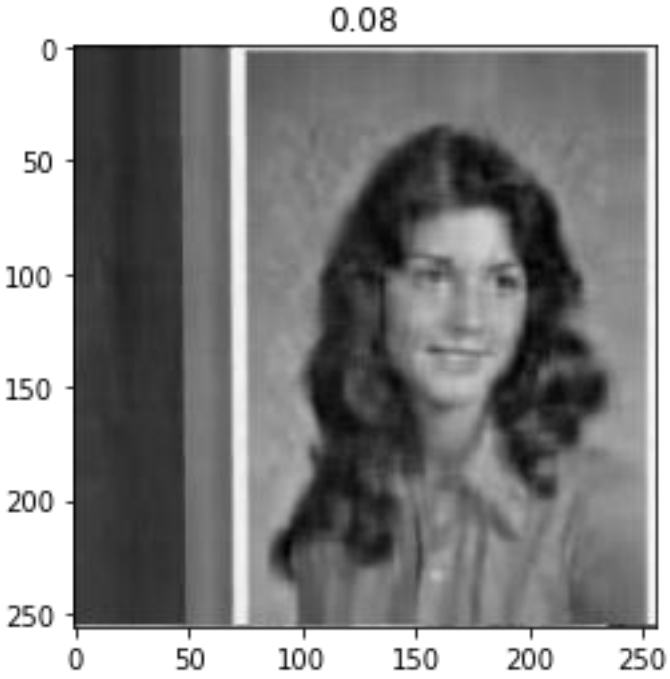
Ratio= 1%



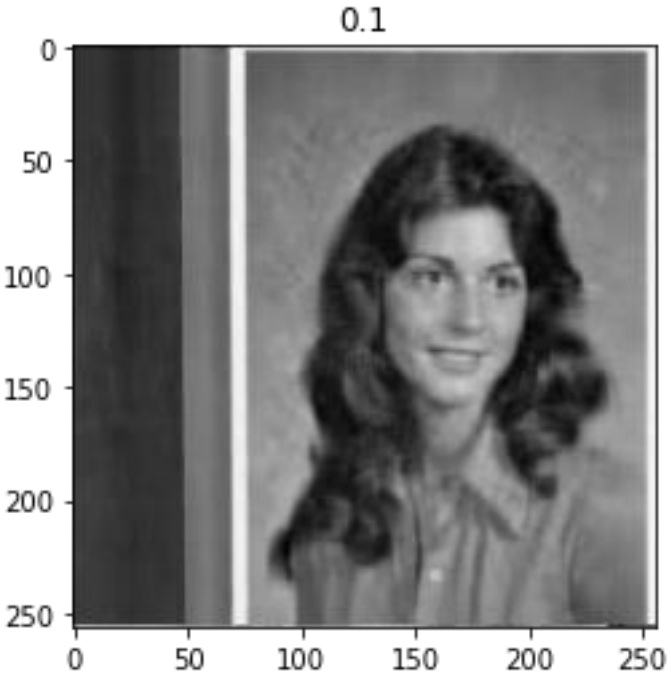
Ratio= 4%



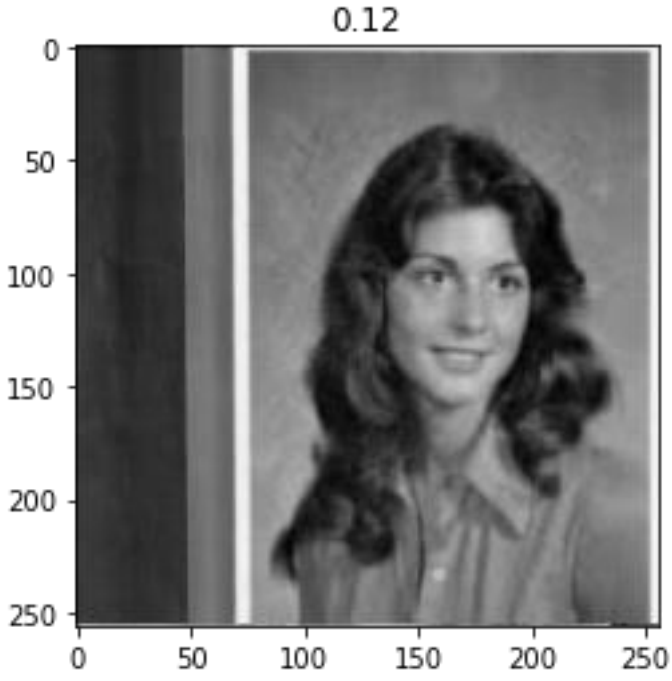
Ratio= 8%



Ratio= 10%



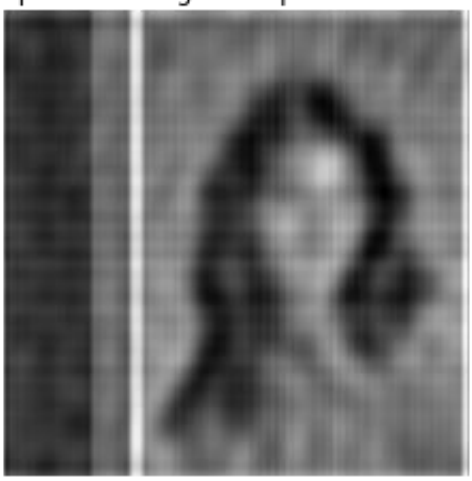
Ratio= 12%



FFT Compression

FFT Compressed image: comperession Rate = 0.1%

FFT Compressed image: comperession Rate = 0.5%



FFT Compressed image: comperession Rate = 1.0%

FFT Compressed image: comperession Rate = 4.0%

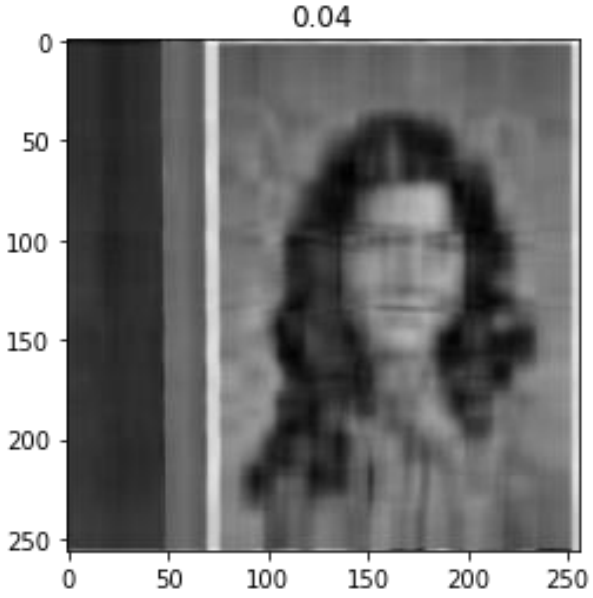
FFT Compressed image: comperession Rate = 8.0%

FFT Compressed image: comperession Rate = 10.0%

FFT Compressed image: comperession Rate = 12.0%

B)

The left image is compressed via the SVD method and the right one via the FFT method. As you can see the FFT method compresses much better than the SVD method.



FFT Compressed image: comperession Rate = 4.0%

In general, the FFT method compresses with much higher quality rather than the SVD method.

The reason is the difference between their algorithms which the FFT method allows low-pass and high-pass filtering with a great degree of accuracy.

Reference

C)

Principal Component Analysis (PCA) is a linear dimensionality reduction technique (algorithm) that transforms a set of correlated variables (p) into a smaller k (k<p) number of uncorrelated variables called ***principal components*** while keeping as much of the variability in the original data as possible.

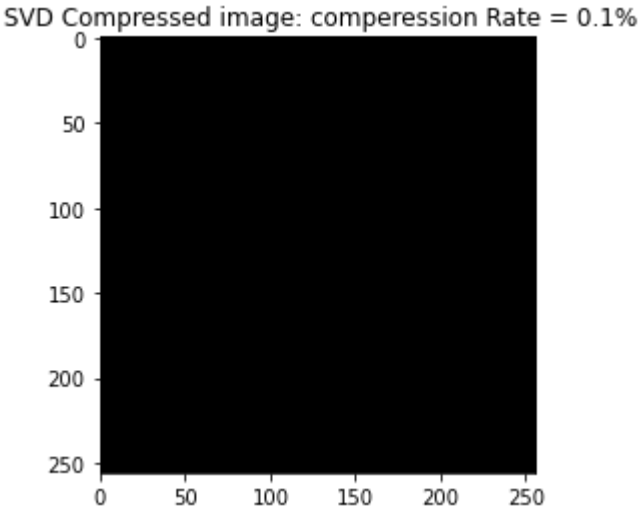
D)

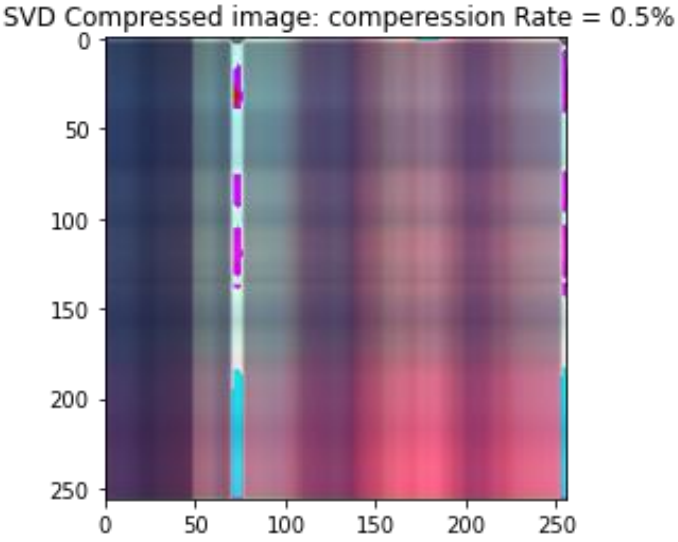
PCA has a very similar signal processing effect to that of SVD when applied to signal de-noising. The reason for this similarity was analyzed theoretically, and it is found that this is because the right singular vectors of the original matrix are just the eigenvectors of its covariance matrix, and this leads to the similarity between PCA and SVD in signal processing.

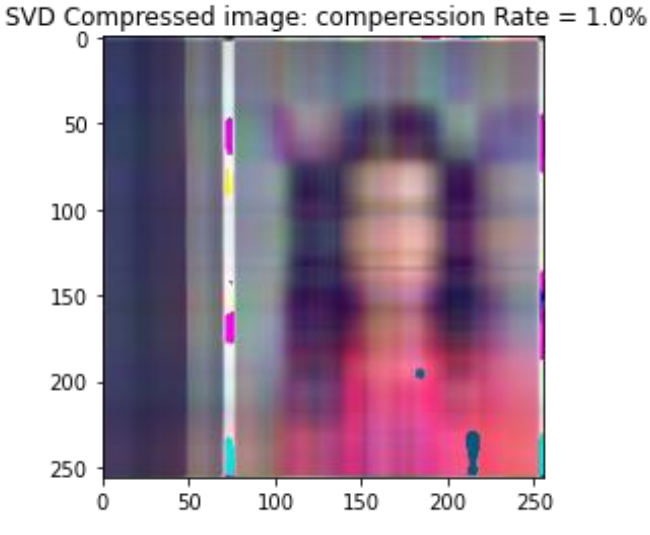
Reference

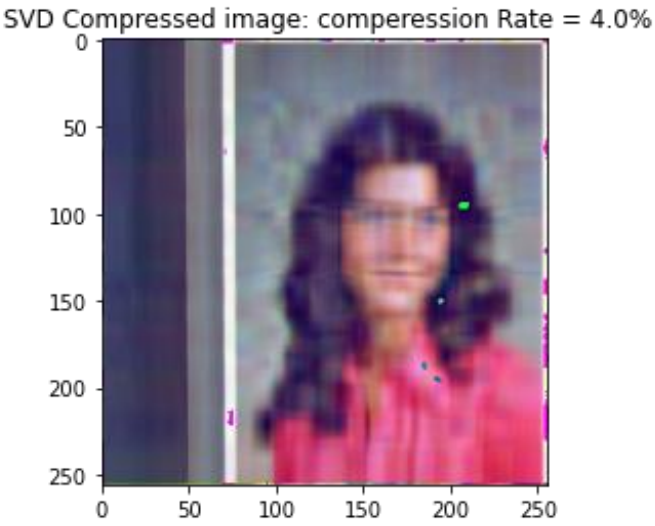
E)

SVD method for color image: Separate the image in RGB lines then compress each line separately.

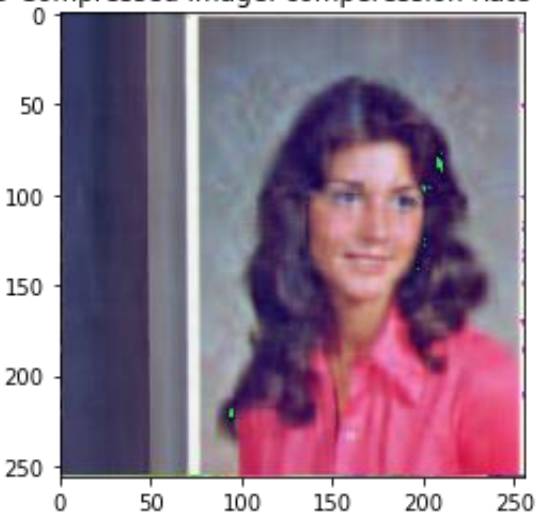




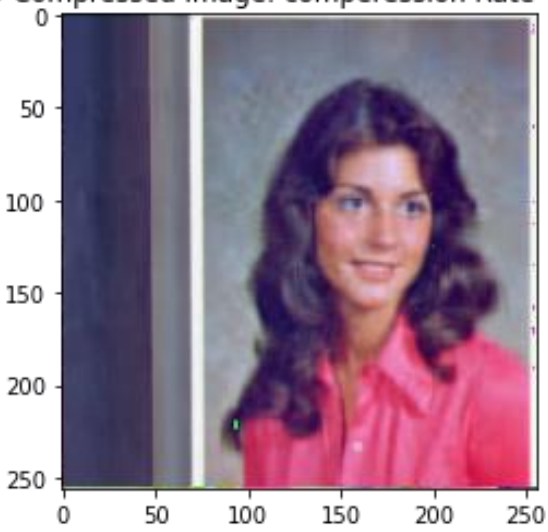




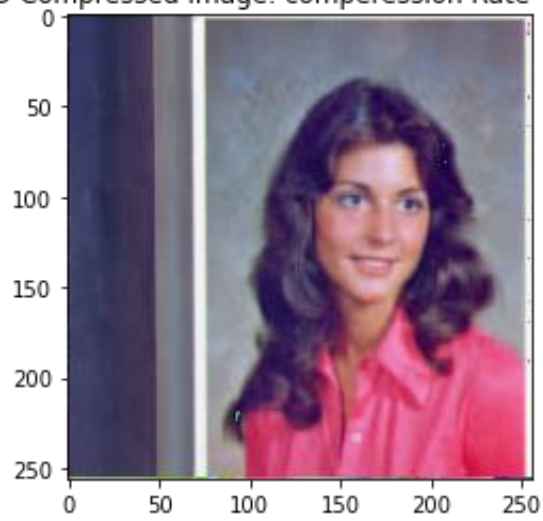
SVD Compressed image: comperession Rate = 8.0%



SVD Compressed image: comperession Rate = 10.0%

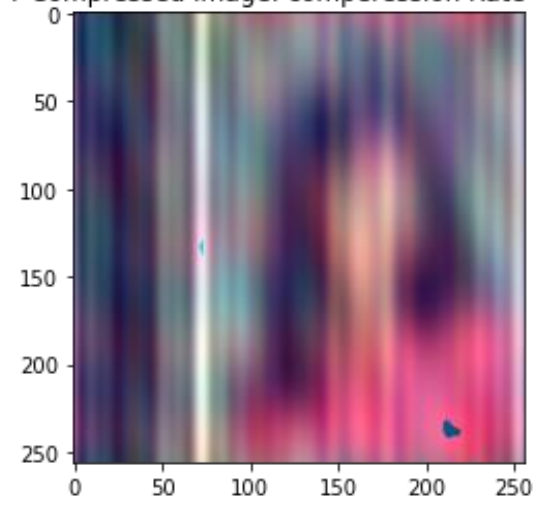


SVD Compressed image: comperession Rate = 12.0%

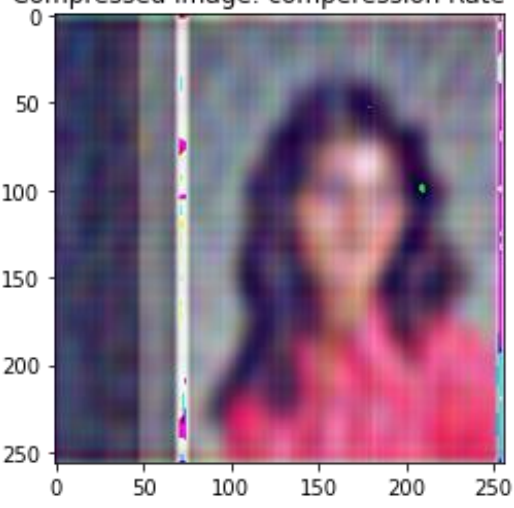


FFT method for color image

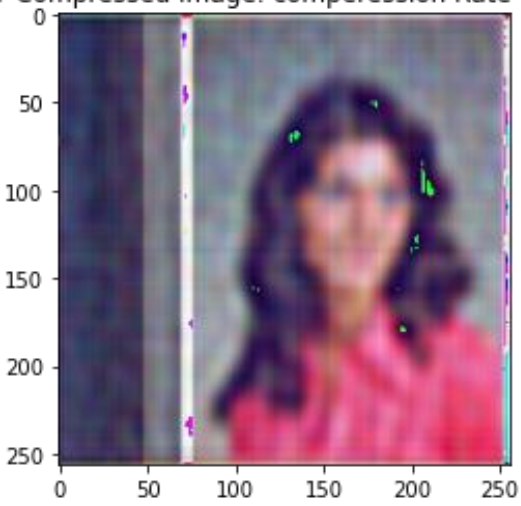
FFT Compressed image: comperession Rate = 0.1%



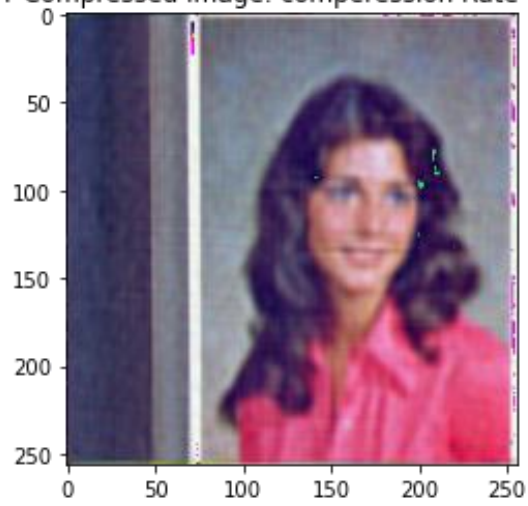
FFT Compressed image: comperession Rate = 0.5%



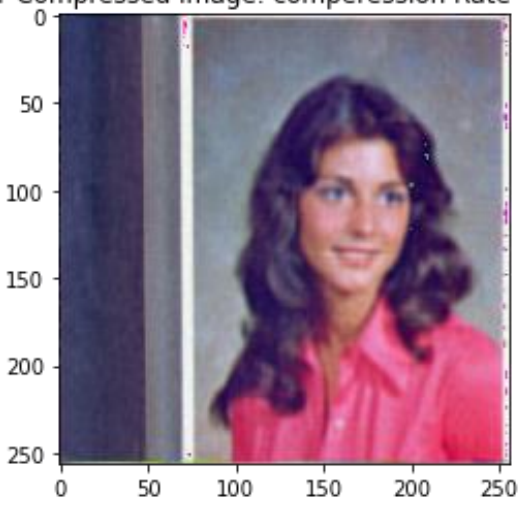
FFT Compressed image: comperession Rate = 1.0%



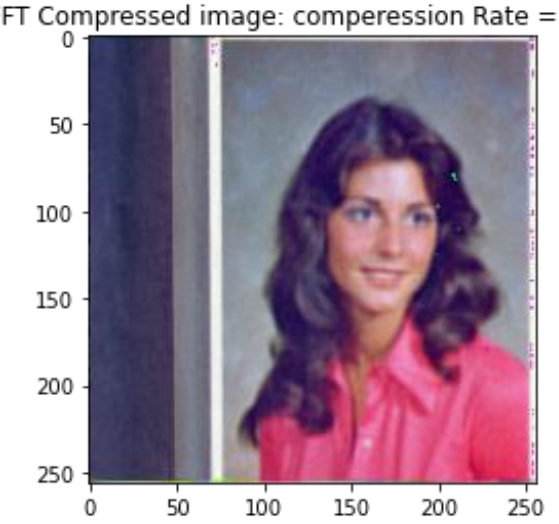
FFT Compressed image: comperession Rate = 4.0%



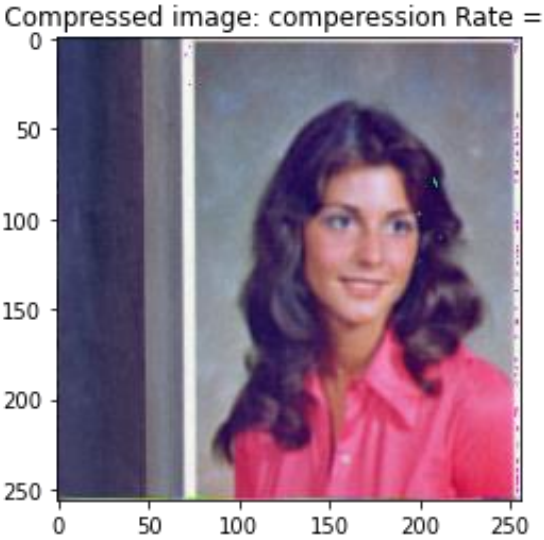
FFT Compressed image: comperession Rate = 8.0%



FFT Compressed image: comperession Rate = 10.0%

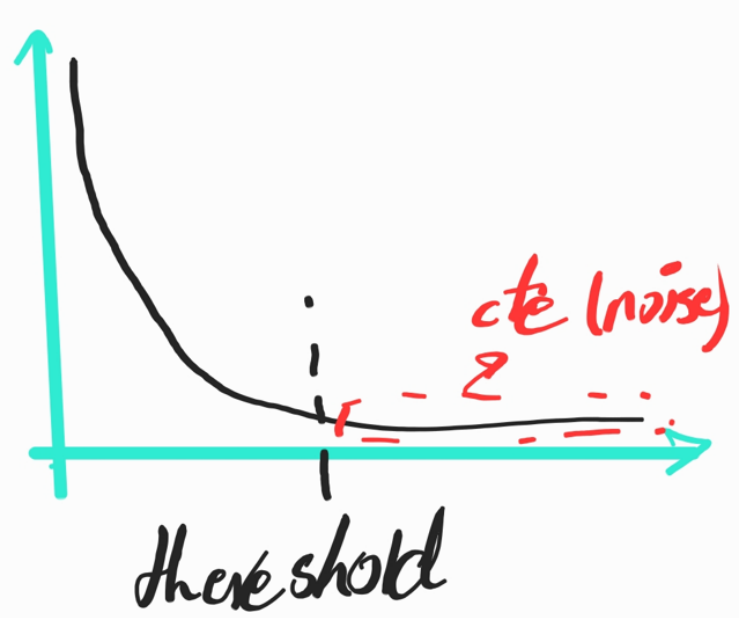


FFT Compressed image: comperession Rate = 12.0%



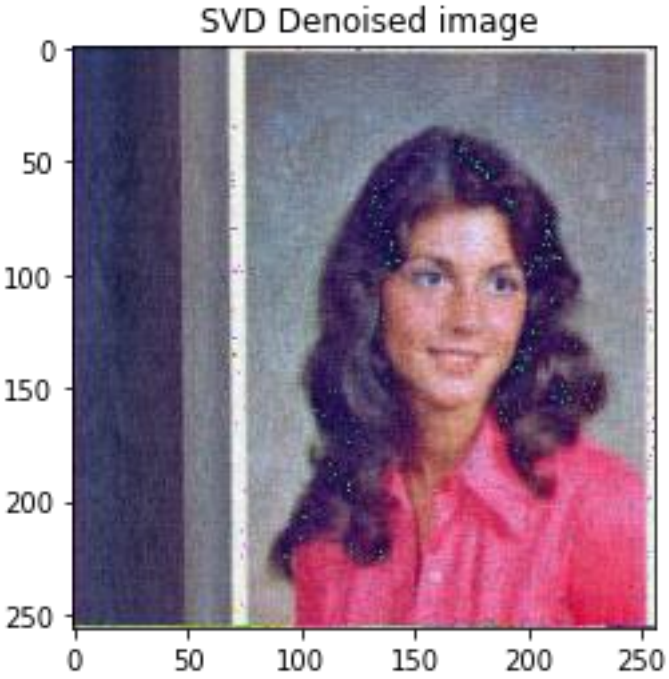
Both compression methods work for color images!

F) SVD Denoising

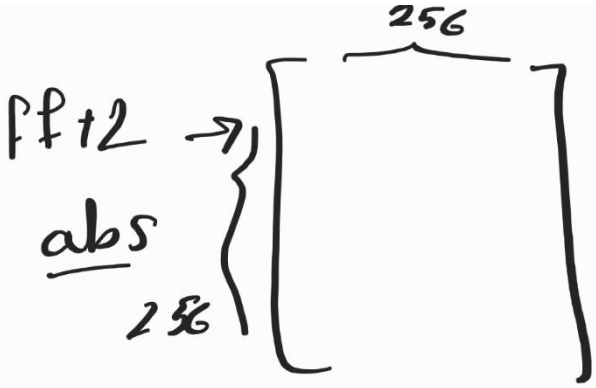


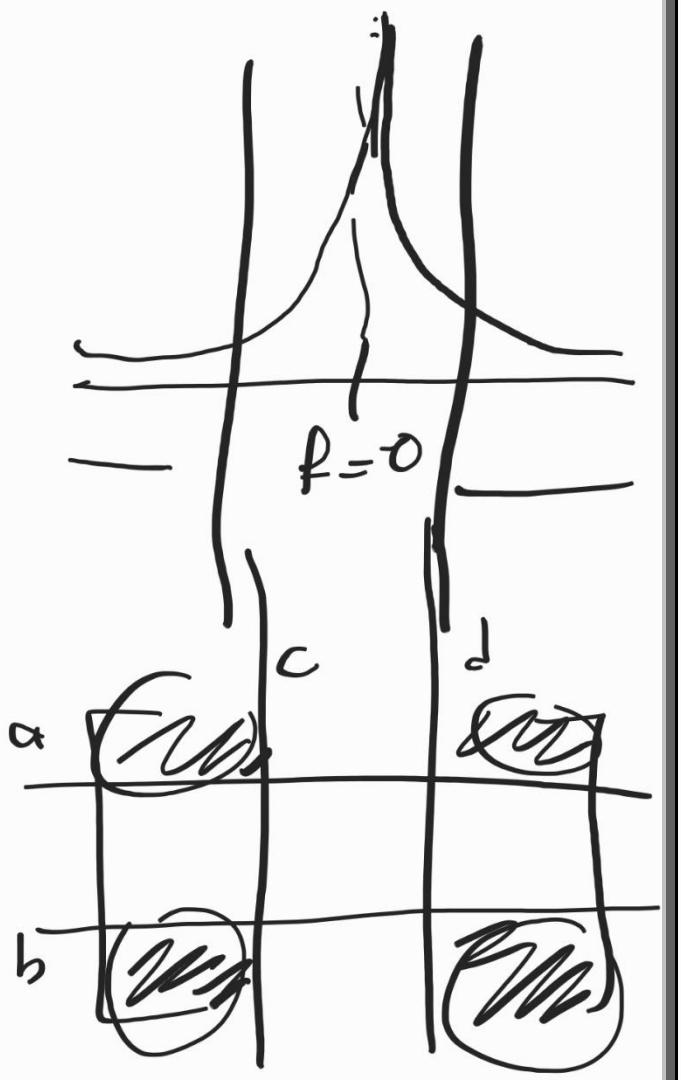
Noise in image is like a DC offset in its singular values, so by calculating the differences between two each singular values and cutting those singular values whose differences are less than the threshold, the noise will be eliminated.

By choosing a better k in the code you can reduce the noise



G)FFT Denoising





fft2 → X_f

X_f(:, c:d) = 0

X_f(a:b, :) = 0

ifft

↓

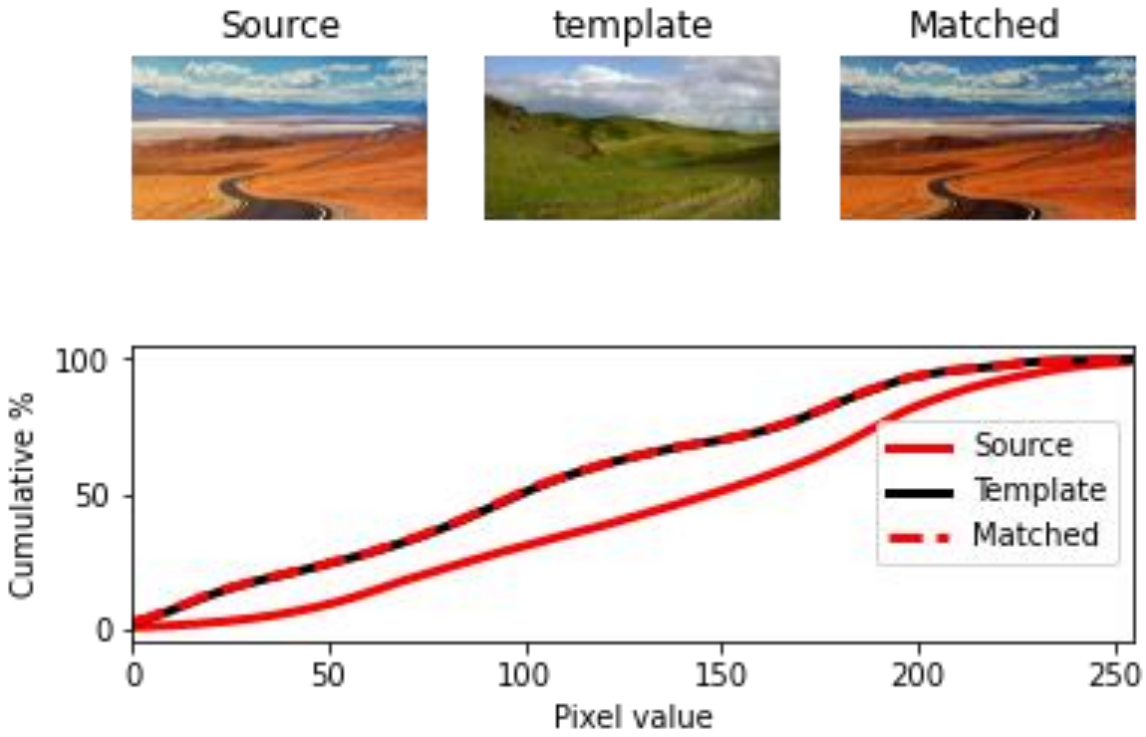
low-pass

In the code that I uploaded you can test the result by changing the thereshold.

H)

FFT denoising works better in image denoising because its algorithms depend on Fourier transform which is more accurate than SVD.

Histogram matching



In image processing, histogram matching or histogram specification is the transformation of an image so that its histogram matches a specified histogram

Modified Gram-Schmidt

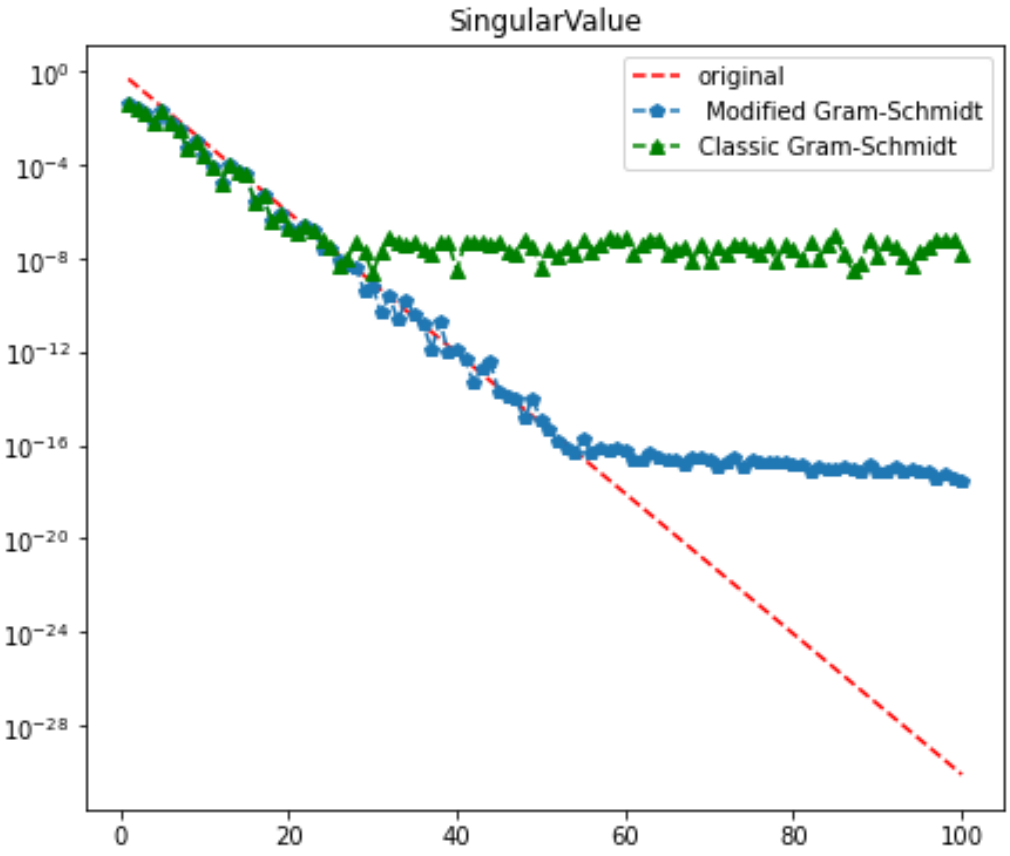
1.

```
def QR(A):
        r, c = A.shape
        Q = np.zeros((r, c),dtype=np.float64) # initialize matrix Q
        u = np.zeros((r, c),dtype=np.float64) # initialize matrix u
        u[:, 0] = copy.copy(A[:, 0])
        Q[:, 0] = u[:, 0] / np.linalg.norm(u[:, 0])
        for i in range(1, c):
                u[:, i] = A[:, i]
                for j in range(i):
                        u[:, i] -= np.dot(A[:, i] , Q[:, j]) * Q[:, j] # get each u vector
                Q[:, i] = u[:, i] / np.linalg.norm(u[:, i]) # compute each e vetor
  # QT=np.transpose(Q)
        R = np.zeros((r, c),dtype=np.float64)
        for i in range(n):
                for j in range(i, c):
                        R[i, j] = np.dot(A[:, j] , Q[:, i])
   #R=np.matmul(QT,A)
        return Q,R
```

2.

```
def QR_Modified_Decomposition(A):
        r, c = A.shape # get the shape of A
        Q = np.zeros((r, c),dtype=np.float64) # initialize matrix Q
  # u = np.zeros((n, m),dtype=np.float64) # initialize matrix u
        R = np.zeros((r, c),dtype=np.float64)
        u = copy.copy(A)
        for i in range(c):
                R[i,i]=np.linalg.norm(u[:, i])
                Q[:, i] = u[:, i] / np.linalg.norm(u[:, i])
                for j in range(i,n):
                        R[i,j]= np.dot(Q[:, i] , u[:, j])
                        u[:, j] -= (np.dot(Q[:, i] , u[:, j]))* Q[:, i] # get each u vector
        return Q,R
```

3.



As you can see, obviously, the modified Gram-Schmit converges more into original values than the classic method.

The modified method is much more stable in dealing with rounded values and is recommended to be used.