IMPLEMENTATION OF SVD
(SINGULAR VALUE DECOMPOSITION) FOR IMAGE BLURRING AND DEBLURRING

Group 32

NAME	ENROLLMENT NO.
Sameep Vani	AU1940049
Kavya Patel	AU1940144
Kashvi Gandhi	AU1940175
Kairavi Shah	AU1940177

1. ABSTRACT:

Image blurring/deblurring is the process of removing blur within the image. To deblur or blur, thus, we use linear algebra concepts. An image can be considered as a matrix. Suppose that the camera resolution has about 65,536 pixels (256^2). Now this can differ on the basis of the resolutions offered by different cameras and may include some millions of pixels also.

After capturing an image, the image is sent in the form of a matrix signal. Thus, the image signal can be considered as $Ax + e = b$ where A is the matrix containing the blur, x is the exact image captured, e is the error that can be caused either by machine (insertion of unwanted noise signal) or by human (capturing image while moving) and b is the resultant vector that is the final image that we get.

Idea: - To retrieve the image, we need the exact value of x. This means that multiplying A^{-1}, we get $x = A^{-1}b – A^{-1}e$. Now here, we can not estimate the value of $A^{-1}e$ as it could be varying. But we can assume that x is approximately $A^{-1}b$ provided that $A^{-1}e$ is small in comparison to $A^{-1}b$. Hence using Singular value decomposition, we can obtain x and hence deblur and blur an image.

2. KEYWORDS

1. QR decomposition.
2. Eigenvectors and Eigenvalues.
3. Transpose.
4. Normalised Vectors.
5. Singular Value Decomposition

3. INTRODUCTION:

Our project focuses on the concept of SVD (Singular Value Decomposition) for image blurring and deblurring. The main idea behind doing this project is not only to learn the theoretical method of SVD but also to implement it in the form of a code that is used to blur/deblur an image using a minimum number of inbuilt libraries. In the real world, there can be various sources that lead to a blurred image such as:

1. Physical sources (moving object, defocused lens etc.)
2. Measurement errors.
3. Discretization errors.
4. Rounding errors.

In our project the main challenge is how to devise effective and reliable algorithms for performing blurring and deblurring of images. This project could be easily made using in built libraries but we had to make the code with minimum or zero in built libraries and so we had to make all the functionality on our own from scratch.

Our assumption is that we get errors minimum in our code and get the most efficient result. Although we faced errors such as whenever we were trying to access the column of a vector,our algorithm returned it in the form of a row because of which the remaining part of the code failed. Thus, to resolve it, we had to use a reshape function in order to get a column vector as a column.

Atlast, the whole python code for SVD implementation for image blurring and deblurring all in running and upto date mode and shows no errors.

4. APPROACH:

To blur or deblur an image, there needs to be generated a formula that represents the blurring and deblurring process. Let b be the image that we get after capturing it. The general formula for this is **Ax + e = b**, where A is a matrix that represents the blurred/deblurred image, x is the image we wish to attain, e is the error or the noise. Here, x, e and b are scalar quantities and A represents a matrix. But the range of e could be small to considerable level. So it is not possible to estimate the error/noise and thus the original image cannot be covered with this mathematical expression. So we assume that as compared to Ax, the value of e is insignificant and hence can be ignored. The advantage of this approach is that now we can easily get the original matrix x using the concepts of Linear algebra. Now to access x, the equation becomes x = A^{-1} (b - e) or $A^{-1}b$ – $A^{-1}e$. Now as discussed above, e is to be ignored, thus the final equation becomes $A^{-1}e$.

Blurring and deblurring using a general linear model

We have assumed that the blur/deblur is linear,the effect of inverted noise can be reduced while keeping the value of A^{-1} b almost the same using some tools.SVD is one of the most popular image blurring and deblurring methods which needs a user input and using that it determines how much noise is reduced. The more you reduce the inverted noise, more information from the image is lost. The SVD in accordance with the frequency of the signal, decomposes the image. With these decompositions, we can dampen the inverted noise so as to make an image that is to its highest quality possible.

How the SVD work ?

The Singular Value Decomposition (SVD) is a matrix decomposition which permits us to isolate our picture into an amount of pictures. The connection between its contribution to the final image and singular value is best portrayed outwardly.

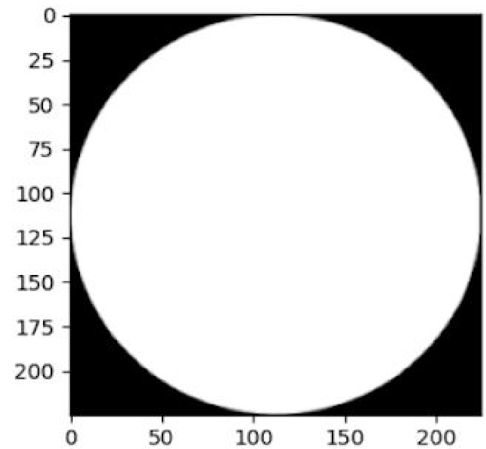


Image 1- Original picture

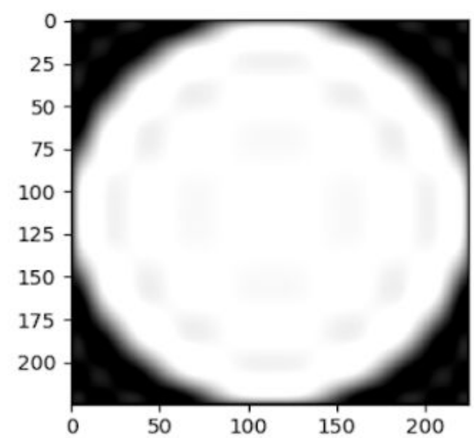


Image 2 - Image with singular value 5

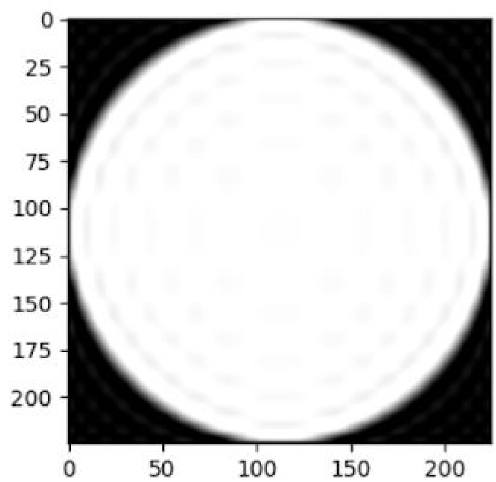


Image 3 - Image with singular value 10

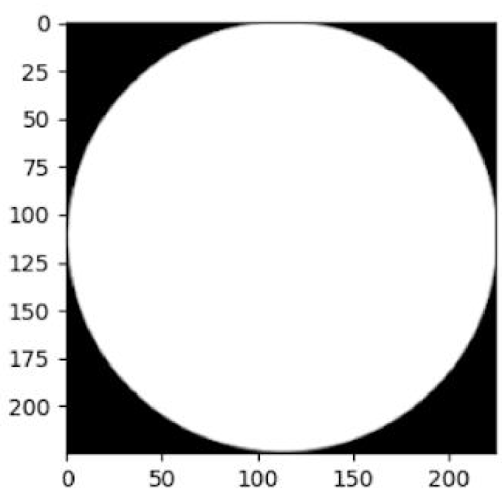


Image 3 - Image with singular value 200

The image on the left is the original image, whereas the image on the right represents the new image with SVD implemented. The higher singular values contribute lots of information to the image since they have lower frequency while the lower singular values have their values close to zero have higher frequency and they contribute very less information to the image.This shows that the higher-frequency singular values have granulated details, but they also contribute a lot to the inverted noise which we wish to dampen.Using SVD, we can identify the portion of image that needs to be dampened or eliminated so as to reduce the inverted noise.In majority instances, SVD assumes a zero boundary condition.

SVD Background :We are given a m x n matrix A, theSVD can be given by the following formula $\mathbf{A} = \mathbf{U}\,\mathbf{\Sigma}\,V^T$. Here, as we know **U** and **V** are orthogonal matrices and **Σ** represents a diagonal matrix. The elements of this diagonal matrix, $_i$ are nonnegative and appear in decreasing order. A is equal to the sum of rank one matrices. A = $_1 * u_1 * v_1^T + {}_2 * u_2 * v_2^T + \ldots + {}_n * u_n * v_n^T$. Inorder to calculate the SVD, we first need to compute $A^T A$.The singular values of A are the square root of the eigenvalues of $A^T A$, which are denoted by $_i$. The columns u_i of the matrix **U** are the normalized eigenvectors of $A^T A$, and the columns of **V**are given by $\boldsymbol{A}\,u_i \;/_i$. If in a matrix that has a many singular values, the smaller values can be removed since they contribute very less to the overall image. Thus, inverted noise can be damped by removing some smaller singular values.

LINEAR ALGEBRA CONCEPTS USED

QR Decomposition - It reduces a matrix into multiplication of two matrices as Q and R where Q is an orthogonal matrix and R is an upper triangular matrix. We used gram-schmidt orthogonalisation process to find the matrices.

```
for i in range(0, col):
        q[:, i] = A[:, i]
        for j in range (0, i):
          # For finding the projection of q[:, j]
            q[:, j] = getNormalisedVector(q[:, j])
            qTranspose = getTranspose(q[:, j].reshape(row, 1))
            r[j][i] = multiplyTwoMatricies(qTranspose,
A[:,i].reshape(row,1))

          # Main process for gram-schmidt orthogonal transformation
            q[:, i] = q[:, i] - multiplyScalarToVector(r[j][i],
q[:, j])
            q[:, j] = getNormalisedVector(q[:, j])
        r[i][i] = getNorm(q[:, i])
        q[:, i] = multiplyScalarToVector(1/r[i][i], q[:, i])
```

Eigenvalues:
Here the approach followed was to iterate through some number of times and find repeated QR decomposition of A matrix and with each iteration we change A to R*Q.
In the end, we get the a diagonal matrix in which the diagonal entries are the eigenvalues.

```
for i in range(0, np.size(A)):
        [Q, R] = findQR(A)
        A = multiplyTwoMatricies(R, Q)
    eValues = np.zeros(len(A))
    for i in range(0, len(A)):
        eValues[i] = A[i][i]
    return eValues
```

Singular Values Decomposition:

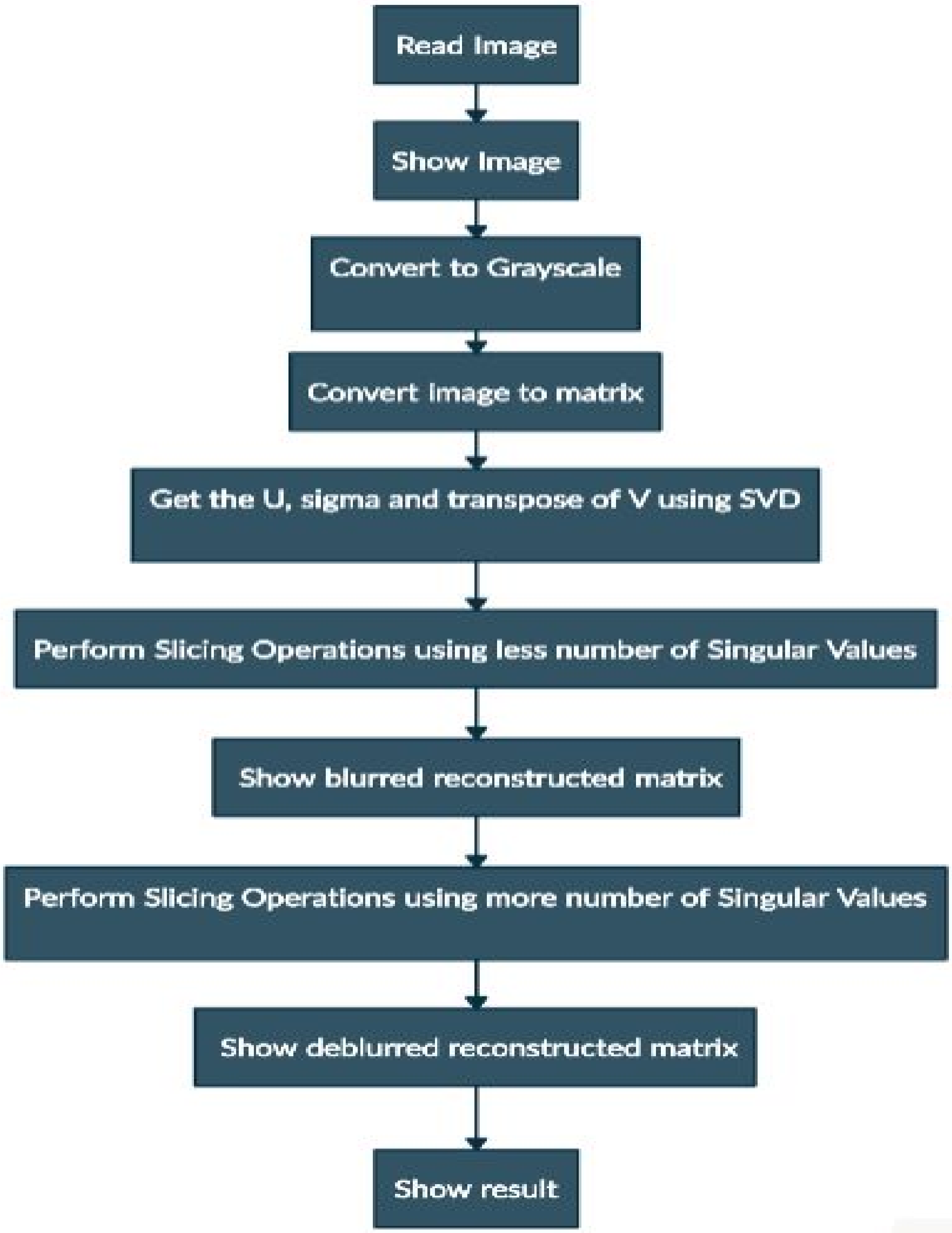
Here we receive eigenvalues, eigenvectors and singular values of matrix as parameters. Now using these, we can easily construct a V matrix as V matrix is just the normalised eigenvectors. SIGMA matrix can also be found easily as it is a diagonal matrix with its entries as singular values. The U matrix is tricky as we need to find AV[i]/sv[i]. The below code is self explanatory in order to find the U matrix.

```
U = np.zeros((n,n))
    V = np.zeros((m,m))
    sigma = np.zeros((n,m))

    # V
    for i in range(0, len(eValues)):
        V[:, i] = getNormalisedVector(eVectors[:, i])

    # Sigma
    for i in range(0, len(sv)):
        sigma[i][i] = sv[i]

    # U
    for i in range(0, m):
        # print('wait here also')
        temp = multiplyTwoMatricies(b, eVectors[:, i].reshape(m,
1)).reshape(1, m)
        U[:, i] = multiplyScalarToVector(1/sv[i], temp)
    return U, sigma, getTranspose(V)
```

5.CODING AND SIMULATION:

Coding strategy:

1.Flowchart of our project



2.Types of modules:

1. PIL for image reading

2. Matplotlib for showing the result

3.numpy.linalg for checking the answers

3. Approach to code:

Following the procedure as described in flowchart, we first obtained the image, converted it into grayscale and obtained the corresponding matrix.

Then we implemented **Singular Value Decomposition**. To implement that we were required to find the U, SIGMA and transpose of V where U and V are the orthogonal matrix and SIGMA is a diagonal matrix. Thus, to find it, we are required to find **Eigenvalues and Eigenvectors**. For this we followed the **QR Algorithm.** Basic idea of QR Algorithm is to find repeated QR decomposition of the matrix of which we need to find the eigenvalues. But one thing to make sure is that, with every iteration the value of A matrix must be R*Q which we got from the previous QR Decomposition. Eventually, with sufficient number (usually m*n) of iterations, we get a diagonal matrix. The diagonal entries of the matrix are approximately equal to real eigenvalues. (precise upto 6th decimal place). Once we have eigenvectors and eigenvalues, we can easily get normalised vectors to obtain the matrix V. Then, to obtain a SIGMA matrix, we calculate the singular values and replace the diagonal elements of SIGMA with singular values. To obtain U, we easily get A*(Vi)/singular values[i].

Having **U, SIGMA and transpose of V**, we then perform slicing operations on all of them. So let us say, we take the first “r”. Let us denote them by Ur, (SIGMA)r and Vr.

Note: -

1. By taking “r” values, we are taking the first “r” values i.e the first “r” largest non-zero singular values.
2. Vr denotes the first “r” columns of TRANSPOSE OF V and not of V.

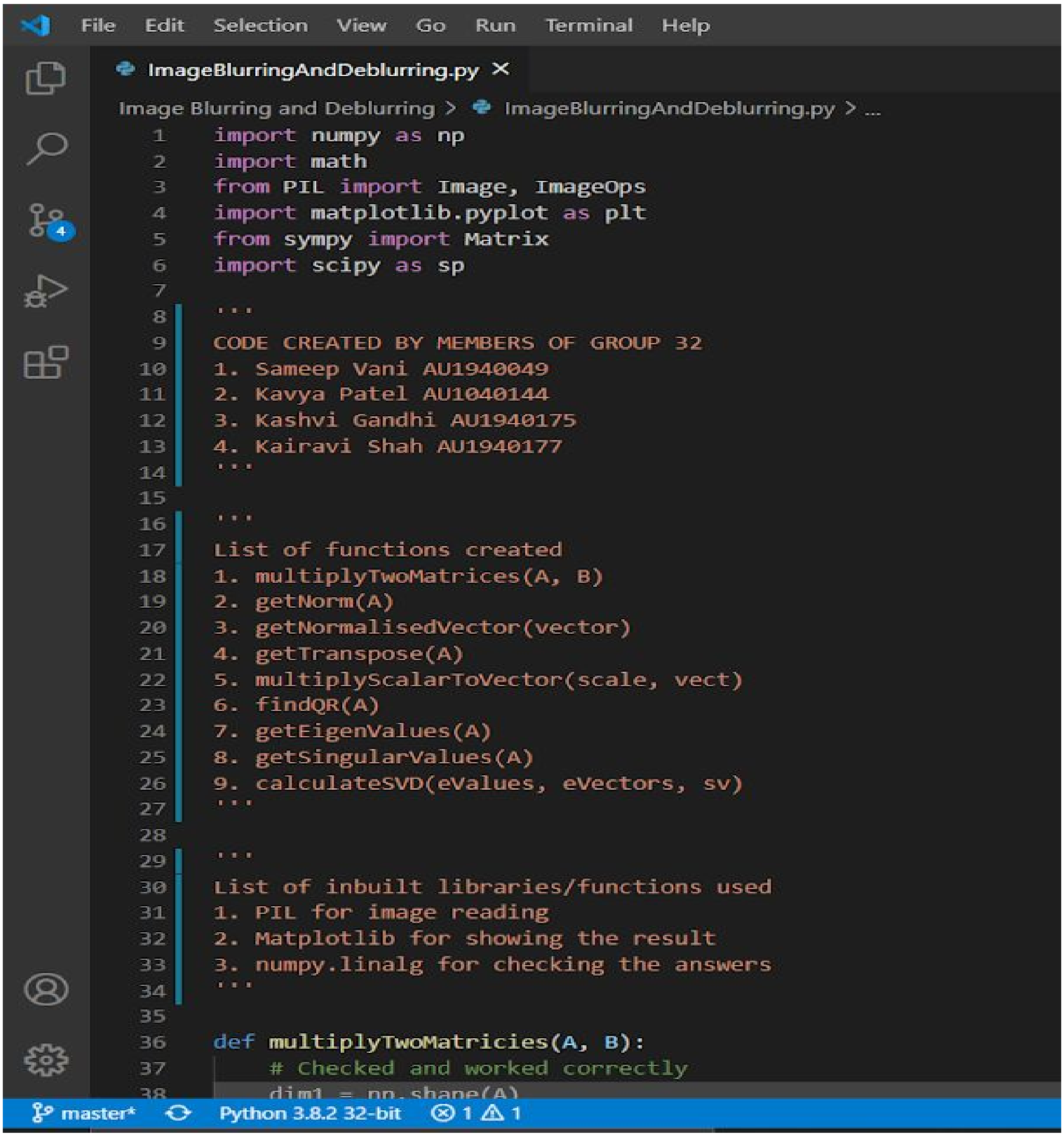
Then, we slice the values according to the value of r i.e we take the first r column of U, rxr matrix from SIGMA and first r columns of V again.

Moving further on, we find the approximated matrix which is formed by the matrix multiplication of Ur, (SIGMA)r and Vr and store the resultant approximation. Later, we show the matrix as an image using a python library called matplotlib.

Note: - When using matplotlib, we need to make sure that the grayscale image is read accurately and hence we need to pass additional arguments as parameters.

Repeating the above for 3 different numbers of singular values, we get the desired output.

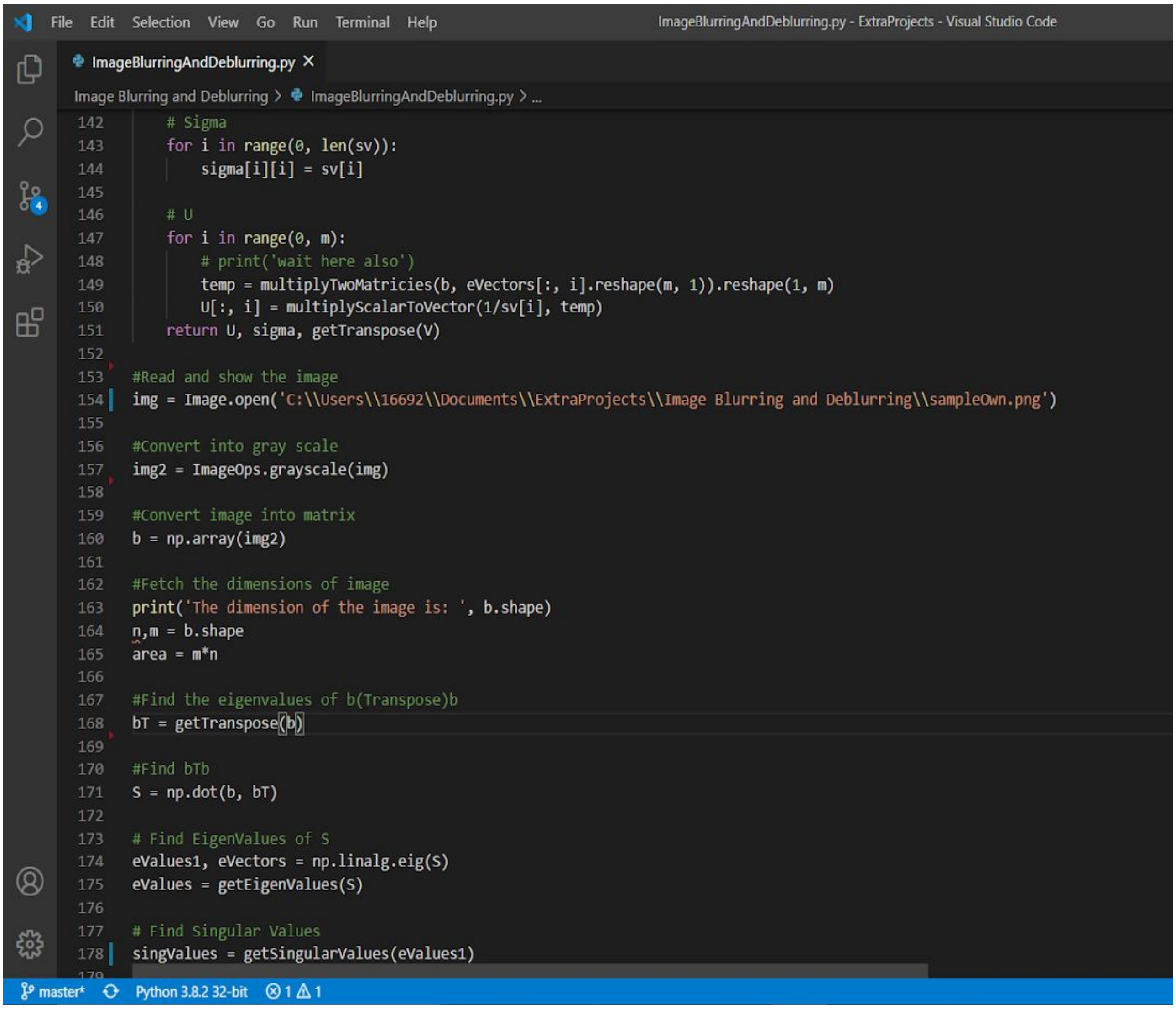
4. Figures of code:



```python
def multiplyTwoMatricies(A, B):
    # Checked and worked correctly
    dim1 = np.shape(A)
    size1 = np.size(A)
    m1 = dim1[0]
    n1 = size1//m1

    dim2 = np.shape(B)
    size2 = np.size(B)
    m2 = dim2[0]
    n2 = size2//m2

    result = np.zeros((m1, n2))
    for i in range(0, m1):
        for j in range(0, n2):
            for k in range(0, m2):
                result[i][j] += (A[i][k] * B[k][j])
    return result


def getNorm(A):
    # Checked and Works Fine
    norm = 0
    for i in range(0, len(A)):
        norm += A[i]*A[i]
    return math.sqrt(norm)


def getNormalisedVector(vector):
    # Checked and Works fine
    length = getNorm(vector)
    for i in range(0, len(vector)):
        vector[i] = vector[i]/length
    return vector

def getTranspose(A):
    # Checked and Works fine
```

```python
def getTranspose(A):
    # Checked and Works fine
    dim3 = np.shape(A)
    size3 = np.size(A)
    temp1 = dim3[0]
    temp2 = size3//dim3[0]

    result = np.zeros((temp2, temp1))
    for i in range(0, temp1):
        for j in range(0, temp2):
            result[j][i] = A[i][j]
    return result

def multiplyScalarToVector(scale, vect):
    # Checked and Works fine
    for i in range(0, len(vect)):
        vect[i] = scale*vect[i]
    return vect

def findQR(A):
    # Checked and Works fine
    dim4 = np.shape(A)
    size4 = np.size(A)
    row = dim4[0]
    col = size4//row

    q = np.zeros((row, col))
    r = np.zeros((col, col))
    for i in range(0, col):
        q[:, i] = A[:, i]
        for j in range (0, i):
            q[:, j] = getNormalisedVector(q[:, j])
            qTranspose = getTranspose(q[:, j].reshape(row, 1))
            r[j][i] = multiplyTwoMatricies(qTranspose, A[:, i].reshape(row, 1))
            q[:, i] = q[:, i] - multiplyScalarToVector(r[j][i], q[:, j])
            q[:, j] = getNormalisedVector(q[:, j])
        r[i][i] = getNorm(q[:, i])
```

```python
#Only For Square symmetric matrix
def getEigenValues(A):
    #Checked and Works Fine
    for i in range(0, np.size(A)):
        [Q, R] = findQR(A)
        A = multiplyTwoMatricies(R, Q)
    eValues = np.zeros(len(A))
    for i in range(0, len(A)):
        eValues[i] = A[i][i]
    return eValues

def getSingularValues(A):
    # Checked and Works fine
    result = np.zeros(len(A))
    for i in range(0, len(A)):
        if(A[i] < 0):
            A[i] = (-1) * A[i]
        result[i] = math.sqrt(A[i])
    return result

def calculateSVD(eValues, eVectors, sv):
    # m = 3
    # n = 3
    U = np.zeros((n,n))
    V = np.zeros((m,m))
    sigma = np.zeros((n,m))

    # V
    for i in range(0, len(eValues)):
        V[:, i] = getNormalisedVector(eVectors[:, i])

    # Sigma
    for i in range(0, len(sv)):
        sigma[i][i] = sv[i]

    # U
    for i in range(0, m):
```




```python
    # Sigma
    for i in range(0, len(sv)):
        sigma[i][i] = sv[i]

    # U
    for i in range(0, m):
        # print('wait here also')
        temp = multiplyTwoMatricies(b, eVectors[:, i].reshape(m, 1)).reshape(1, m)
        U[:, i] = multiplyScalarToVector(1/sv[i], temp)
    return U, sigma, getTranspose(V)

#Read and show the image
img = Image.open('C:\\Users\\16692\\Documents\\ExtraProjects\\Image Blurring and Deblurring\\sampleOwn.png')

#Convert into gray scale
img2 = ImageOps.grayscale(img)

#Convert image into matrix
b = np.array(img2)

#Fetch the dimensions of image
print('The dimension of the image is: ', b.shape)
n,m = b.shape
area = m*n

#Find the eigenvalues of b(Transpose)b
bT = getTranspose(b)

#Find bTb
S = np.dot(b, bT)

# Find EigenValues of S
eValues1, eVectors = np.linalg.eig(S)
eValues = getEigenValues(S)

# Find Singular Values
singValues = getSingularValues(eValues1)
```

```python
eValues1, eVectors = np.linalg.eig(S)
eValues = getEigenValues(S)

# Find Singular Values
singValues = getSingularValues(eValues1)

# Compute SVD
UCheck, sigmaCheck, VTCheck = np.linalg.svd(b)
U, Sigma, VT = calculateSVD(eValues, eVectors, singValues)

# Blurring an image by taking small number of singular values(say 20)
k = 1

# Performing Slicing operations
resultantBlurredMatrixApproximated = U[:,:k] @ Sigma[0:k,:k] @ VT[:k,:]

# Deblurring an image by taking large number of singular values (say 1000)
k = 10
resultantDeblurredMatrixApproximated = U[:,:k] @ Sigma[0:k,:k] @ VT[:k,:]

# Performing Slicing operations
resultantDeblurredMatrixApproximated = U[:,:k] @ Sigma[0:k,:k] @ VT[:k,:]

# Final Image
k = 1000
resultantDeblurredMatrixApproximatedFinal = U[:,:k] @ Sigma[0:k,:k] @ VT[:k,:]

# Show Result/Output
f, axes = plt.subplots(2,2)
plt.suptitle('Results')
axes[0][0].imshow(img)
axes[0][1].imshow(resultantBlurredMatrixApproximated, cmap='gray',vmin=0, vmax=255)
axes[1][0].imshow(resultantDeblurredMatrixApproximated, cmap='gray',vmin=0, vmax=255)
axes[1][1].imshow(resultantDeblurredMatrixApproximatedFinal, cmap='gray',vmin=0, vmax=255)
plt.show()
```

TIME COMPLEXITY OF FUNCTIONS

1) multiplyTwoMatrices(A, B): $O(n^3)$
2) getNorm(A): $O(n)$
3) getNormalisedVector(vector): $O(n)$
4) getTranspose(A): $O(n^2)$
5) multiplyScalarToVector(scale, vect): $O(n)$
6) findQR(A): $O(n^5)$
7) getEigenValues(A): $O(n^4)$
8) getSingularValues(A): $O(n)$
9) calculateSVD(eValues, eVectors, sv): $O(n^4)$

TOTAL TIME COMPLEXITY : $O(n^5)$

5. Inferences:

1. Larger singular values contribute more to the final image.
2. Larger the number of singular values we include, the sharper the image becomes.

6. Conclusion:

As we reach the end of the image blurring and deblurring using SVD (Single Value Decomposition), we have received all the results successfully. The final image is blurred and deblurred with singular values of 5, 10 and 200 and can also be obtained for any singular value. So thus we have learnt how to implement the SVD for image blurring and deblurring not only theoretically but also practically using python code.

All the project work was equally divided among all four of us. Sameep led the group for this project and gave us the main flow of this project. We all started working respectively in our fields. Sameep started with the python code and Kashvi helped him along. Kavya and Kairavi started with the report and gathered all the information required for the report and then helped Sameep and Kashvi solve the error which incurred in the code. After the completion of the code, Sameep and Kashvi came along to finish and give the finishing touch to the report and presentation.

7. References:

[1] Sanborn, J. J. (2019, May). APPLYING LINEAR ALGEBRA TO IMAGE DEBLURRING (Tech.). Retrieved November 6, 2020, from http://msekce.karlin.mff.cuni.cz/~tuma/Aplikace15/prezentace_Hnetynkova.pdf.

[2]Hnˇetynkov´a, I. (2020). *Singular Value Decomposition - Applications in Image Processing*. Msekce.karlin.mff.cuni.cz. Retrieved 6 November 2020, from

https://academicarchive.snhu.edu/bitstream/handle/10474/3516/hon2019sanborn.pdf?sequence=1&isAllowed=y#:~:text=To%20fix%20these%20errors%2C%20one,represent%20a%20certain%20color%20intensity.&text=To%20deblur%20a%20photo%2C%20we,on%20concepts%20of%20linear%20algebra.

[3]Christian Hansen, P. (2020). *Image Deblurring in the Light of the Cosine Transform*. People.compute.dtu.dk. Retrieved 6 November 2020, from http://people.compute.dtu.dk/pcha/Talks/Light.pdf.

[4]Imm.dtu.dk. (2020). Retrieved 6 November 2020, from http://www.imm.dtu.dk/~pcha/HNO/chap1.pdf .

[5]C, I. (2020). *Implementing QR decomposition in C*. Stack Overflow. Retrieved 6 November 2020, from https://stackoverflow.com/questions/35834294/implementing-qr-decomposition-in-c.

[6]*QRalgorithm*. Pi.math.cornell.edu. (2020). Retrieved 6 November 2020, from http://pi.math.cornell.edu/~web6140/TopTenAlgorithms/QRalgorithm.html.

[7]Brownlee, J. (2020). *How to Calculate the SVD from Scratch with Python*. Machine Learning Mastery. Retrieved 6 November 2020, from https://machinelearningmastery.com/singular-value-decomposition-for-machine-learning/.

[8]Brunton, S. (2020). SVD: Image Compression [Python] [Video]. Retrieved 6 November 2020, from https://www.youtube.com/watch?v=H7qMMudo3e8&list=WL&index=2&t=364s.

[9]*The QR Method for Finding Eigenvalues*. Cstl-csm.semo.edu. (2020). Retrieved 6 November 2020, from http://cstl-csm.semo.edu/jwojdylo/MA345/Chapter6/qrmethod/qrmethod.pdf.

[10]Mikida, E. (2020). *The QR Algorithm for Finding Eigenvectors*. Cse.buffalo.edu. Retrieved 6 November 2020, from https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Eric-Mikida-Fall-2011.pdf.

[11]*Finding matrix eigenvectors using QR decomposition*. StackExchange. (2020). Retrieved 6 November 2020, from https://stats.stackexchange.com/questions/20643/finding-matrix-eigenvectors-using-qr-decomposition.

[12]Fekadie Anley, E. (2016). The QR Method for Determining All Eigenvalues of Real Square Matrices. *Pure And Applied Mathematics Journal*, *5*(4), 113. https://doi.org/10.11648/j.pamj.20160504.15