DEPARTMENT OF ROBOTICS & MECHATRONICS ENGINEERING

UNIVERSITY OF DHAKA

COURSE NAME: Digital Image Processing Lab

COURSE CODE: RME 4112

ASSINGMENT NO: 01

ASSIGNMENT ON: Bilinear Interpolation, Bi-cubic Interpolation, Affine Transformation

GROUP MEMBERS:

Humayra Rashid (SK-092-011)

Ipshita Ahmed Moon (SK-092-017)

Safayat Hasan (SH-092-021)

SUBMITTED TO: Dr. Md Mehedi Hasan
Lecturer,
Dept. of Robotics & Mechatronics Engineering,
University of Dhaka

DATE OF SUBMISSION: 24th January 2023

Theory:

The word interpolation means estimating unknown values using related known values. While doing image resizing we estimate the pixel value at any particular location using the known pixels. There are different kinds of interpolation such as nearest neighbor interpolation, bilinear interpolation, bi-cubic interpolation etc. So here we'll be implementing bilinear and bi-cubic interpolation.

In the later part we deal with the affine transformation. In this we convert our image into a matrix and do some mathematical operation. After that we get some image converted by affine tansformation.

Bilinear Interpolation:

At first we import math numpy for matrix calculation. Now we take cv2 for reading an image.

```
In [1]: import math
        import cv2
        import numpy as np
        img = cv2.imread("monalisa.jpg")
        print(img.shape)
```

Now, we take given_image, new_hight and new_width as 3 parameters for our function bilinear Interpolation. Using .shape method we get the dimensions of the given image. We create an empty array using resized of shape.

```
In [2]: def bilinear_Interpolation(given_img, new_hight, new_width):
            old_hight, old_width, c = given_img.shape

            #array for resizing
            resized = np.zeros((new_hight, new_width, c))
```

To fill in the empty array we iterate through the pixels using 2 for loops. We must map the coordinate values of the pixels in the new array back into the original image array to estimate their value. To obtain x and y values, multiply the coordinate values i,j by the scaling factors of the corresponding dimensions.

```
for i in range(new_hight):
    for j in range(new_width):

        x = i * hight_scaling_factor
        y = j * width_scaling_factor
```

We need the values of four neighboring pixels from the original image to estimate a pixel value using Bilinear Interpolation. As a result, we now compute the coordinate values for the four neighboring pixels.

```
#coordinate values for 4 surrounding pixels.
x_floor = math.floor(x)
x_ceil = min( old_hight - 1, math.ceil(x))
y_floor = math.floor(y)
y_ceil = min(old_width - 1, math.ceil(y))
```

We can calculate the pixel values (v1, v2, v3, and v4) of these four neighboring pixels using the coordinate values.

```
        v1 = given_img[x_floor, y_floor, :]
        v2 = given_img[x_ceil, y_floor, :]
        v3 = given_img[x_floor, y_ceil, :]
        v4 = given_img[x_ceil, y_ceil, :]
```

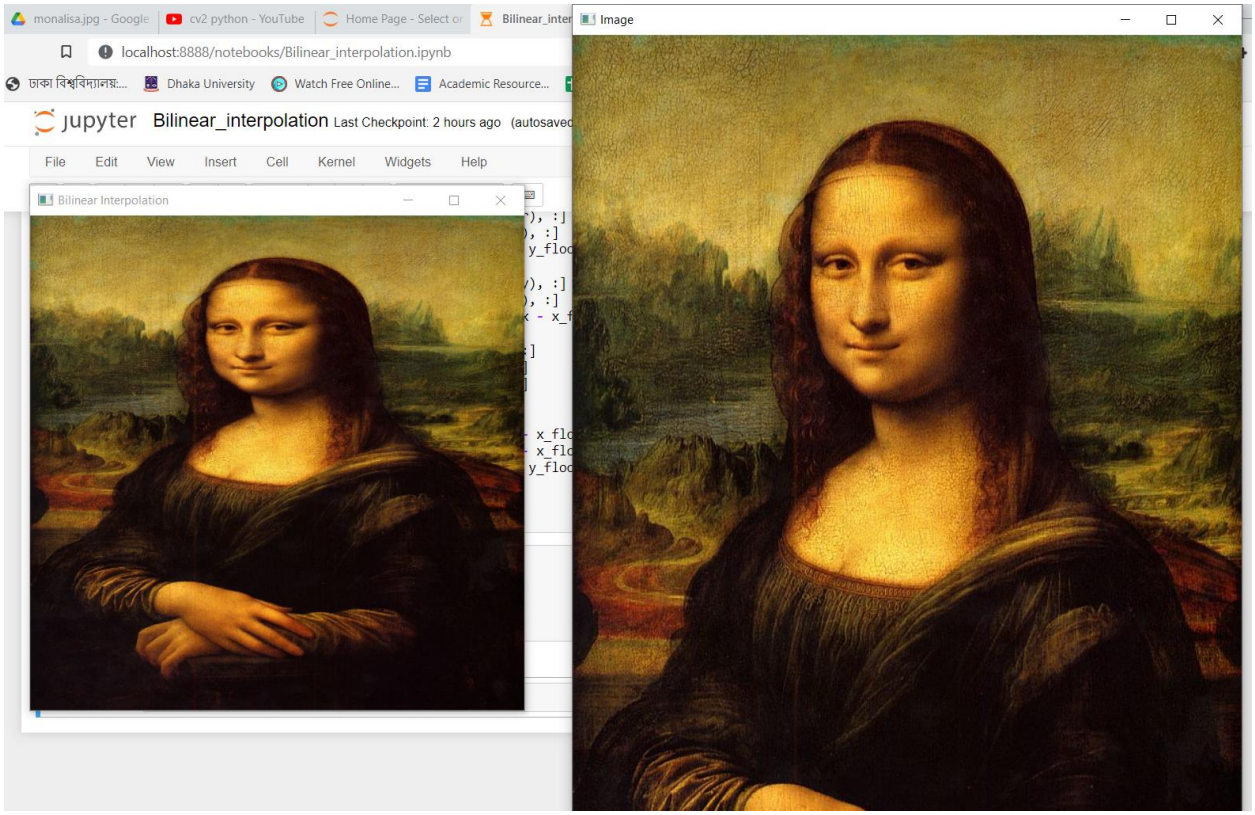
We estimate the pixel value q using neighboring pixel values and assign it to the pixel at coordinates (i,j) in the new image array.

```
            if (x_ceil == x_floor) and (y_ceil == y_floor):
                q = given_img[int(x), int(y), :]
            elif (x_ceil == x_floor):
                q1 = given_img[int(x), int(y_floor), :]
                q2 = given_img[int(x), int(y_ceil), :]
                q = q1 * (y_ceil - y) + q2 * (y - y_floor)
            elif (y_ceil == y_floor):
                q1 = given_img[int(x_floor), int(y), :]
                q2 = given_img[int(x_ceil), int(y), :]
                q = (q1 * (x_ceil - x)) + (q2 * (x - x_floor))
            else:
                v1 = given_img[x_floor, y_floor, :]
                v2 = given_img[x_ceil, y_floor, :]
                v3 = given_img[x_floor, y_ceil, :]
                v4 = given_img[x_ceil, y_ceil, :]

                q1 = v1 * (x_ceil - x) + v2 * (x - x_floor)
                q2 = v3 * (x_ceil - x) + v4 * (x - x_floor)
                q = q1 * (y_ceil - y) + q2 * (y - y_floor)

            resized[i,j,:] = q
    return resized.astype(np.uint8)
```

Finally, we call our image by function and get the output as required by bilinear interpolation.

In [3]:
```
h,w = map(int,input().split(' '))
reshape = bilinear_Interpolation(img,h,w)
cv2.imshow("Bilinear Interpolation",reshape)
cv2.imshow("Image",img)
cv2.waitKey(0)
```

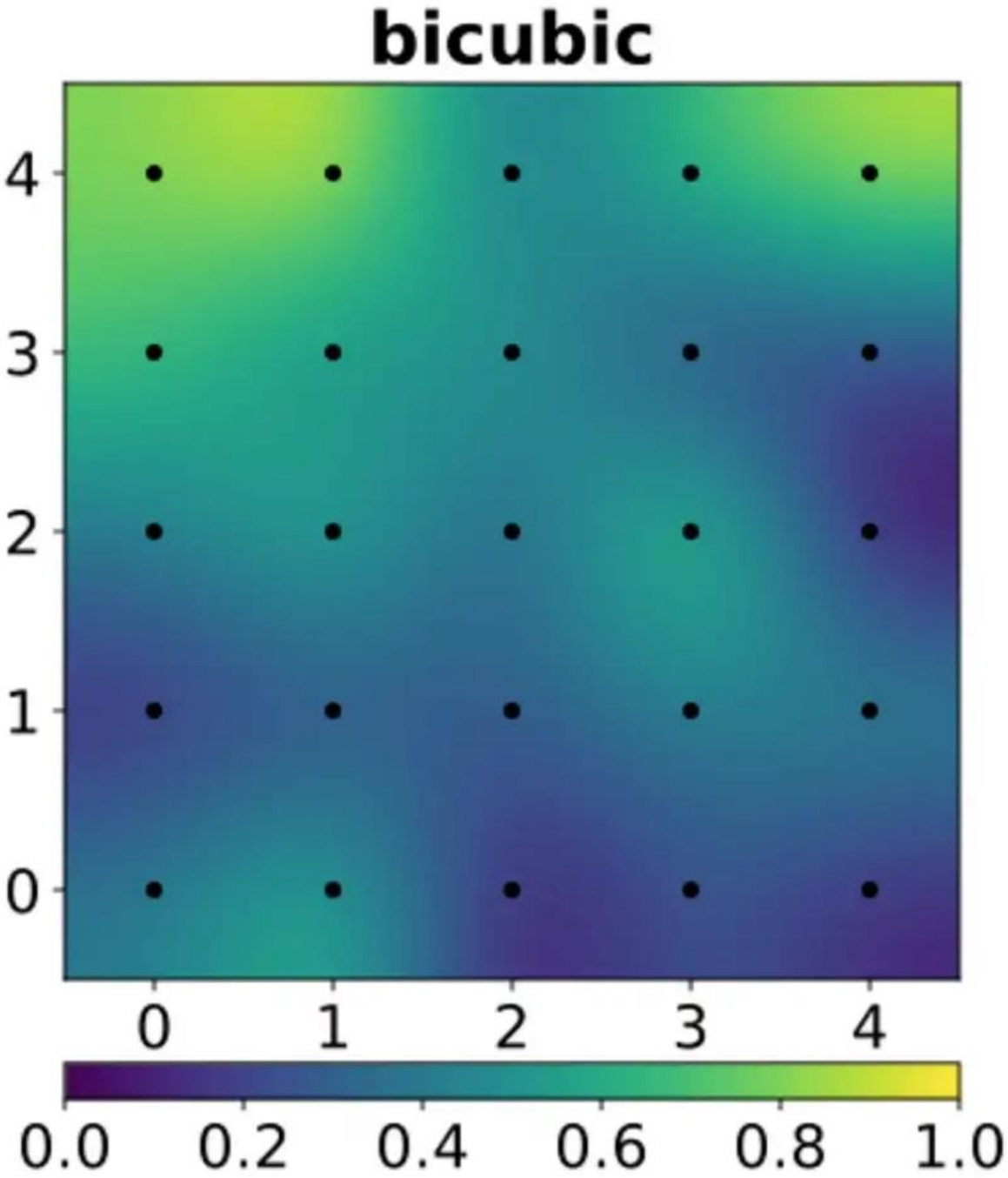
```
500 500
```



Bi-Cubic Interpolation:

Bicubic interpolation is a method of interpolating data points on a two-dimensional regular grid that is an extension of cubic interpolation. When speed is not an issue, bicubic interpolation is often preferred over bilinear or nearest-neighbor interpolation in image resampling. In contrast to bilinear interpolation, which only considers four pixels (22), bicubic interpolation considers sixteen pixels (44).

To maintain sharpness and details, each pixel must be approximated with its surrounding pixels to obtain the closest value. It is analogous to duplicating a pixel in order to fill the space created in the image by upscaling. As a result, the values must be similar to or the same as those of its nearest pixel neighbor. When upscaling, you must add 8 million new pixels to the image to fill the space. They only need to be precise enough to recreate the details while maintaining image clarity.



(Source Wikipedia)

This is a square (4,0) x (0,4) with each square representing a pixel. It has 25 pixels in total (5 x 5). The black dots represent the interpolated data, which totals 25 dots. Because the colors represent function values, we can see that they are not radially symmetric in this example. This enables smoother resampling with fewer image artifacts. Because of this, bicubic interpolation is frequently preferred over bilinear or nearest-neighbor interpolation, but it takes longer to process images.

Affine Transformation:

At first we take the Pillow, numpy, matplotlib.pyplot, scipy and skimage.

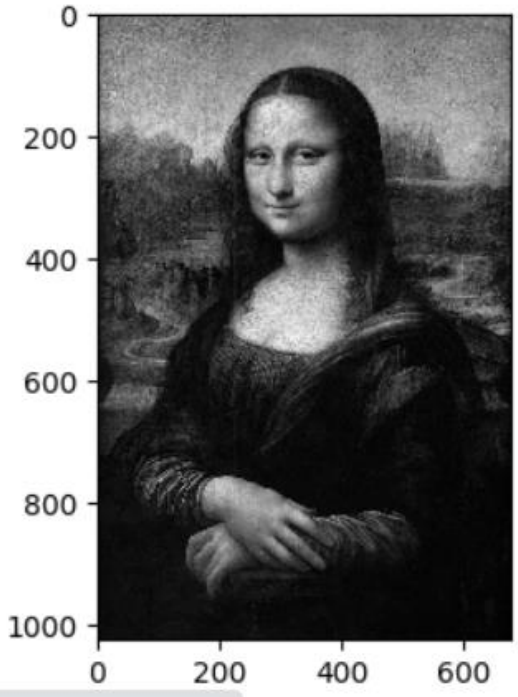
```
In [139]: from PIL import Image
          import numpy as np
          import matplotlib.pyplot as plt
          from scipy import ndimage as ndi
          from skimage.io import imread
```

Now we take the image and convert it into grey scale image which will be used for the translation and transformation later.

```
In [140]: img = imread('monalisa.jpg', as_gray = True)
          #w, h= img.shape
          #print(w, h)
```

Now we plot the image as graph to see the origin.

```
In [141]: plt.figure(dpi = 100)
          plt.imshow(img, cmap = 'gray')
          plt.show()
```

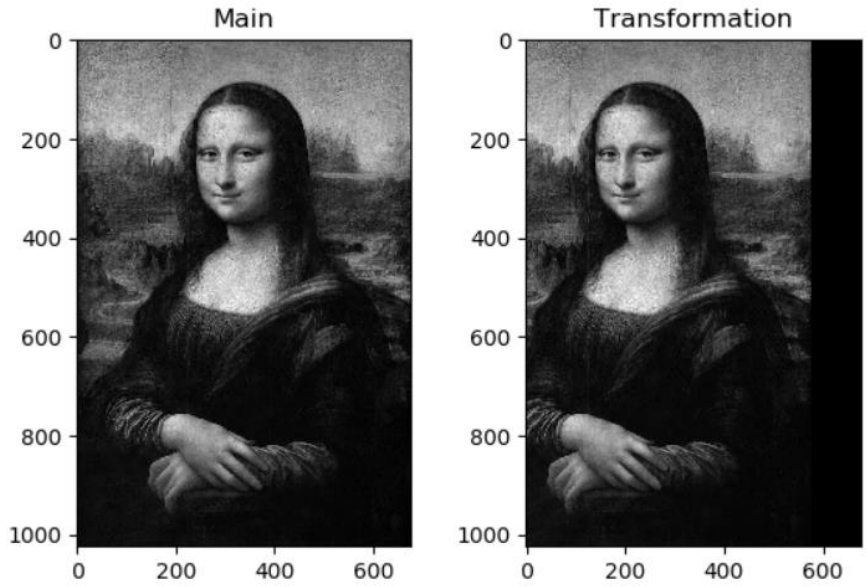


Now we will try to see the main image and transformed image in the subplot with no of rows as 1 and no of columns as 2 and dpi as 100.

```
In [142]: def show_images(main_image, transformed_image):
              figure, axes = plt.subplots(nrows=1, ncols = 2, dpi=100)

              axes[0].set_title('Main')
              axes[0].imshow(main_image, cmap = 'gray')

              axes[1].set_title('Transformation')
              axes[1].imshow(transformed_image, cmap= 'gray')


              figure.tight_layout()
              plt.show()
```

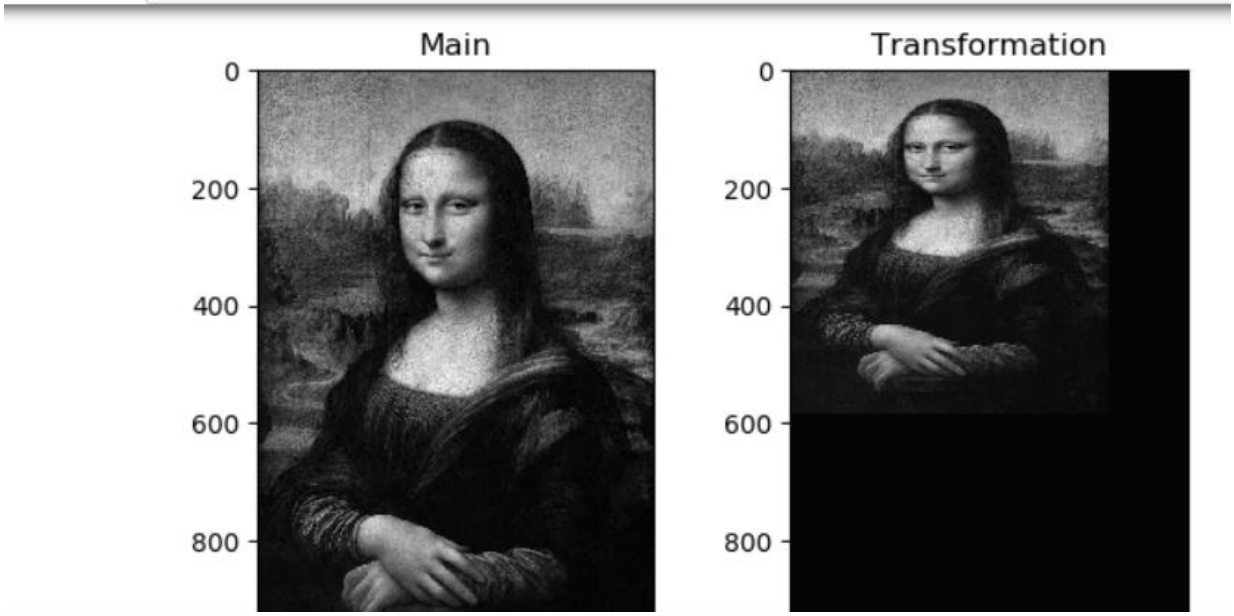
Now we will translate the image towards x-axis by multiplying with the $\begin{bmatrix} 1 & 0 & X \\ 0 & 1 & Y \\ 0 & 0 & 1 \end{bmatrix}$ matrix.

```
In [143]: translation = 100
          translation_matrix = np.array([[1,0,0],
                                         [0,1,translation],
                                         [0,0,1]])
          image = ndi.affine_transform(img, translation_matrix)

          show_images(img, image)
```



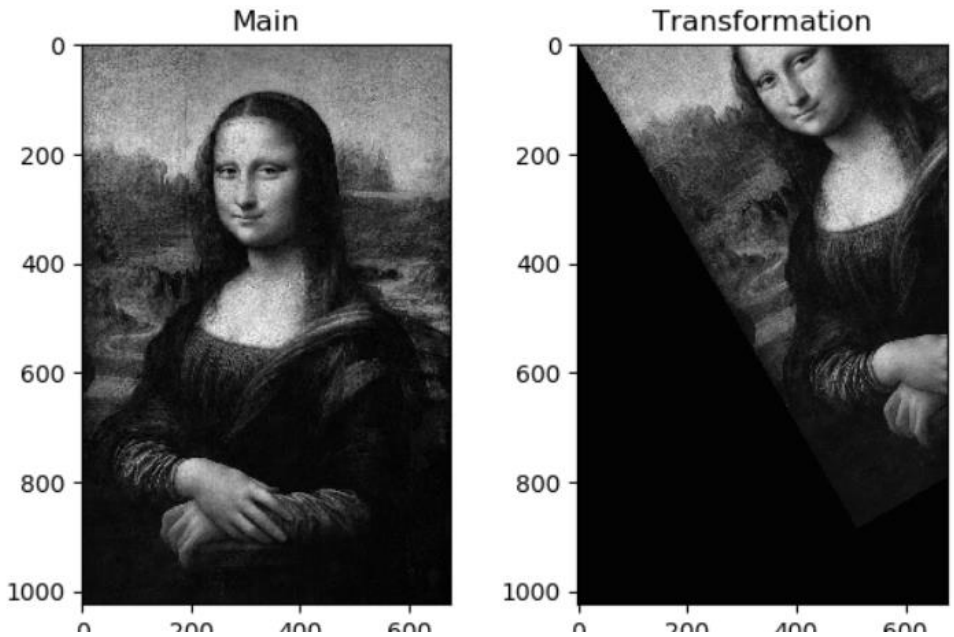
Now we will scale the image towards x-axis and y-axis by multiplying with the $\begin{bmatrix} W & 0 & 0 \\ 0 & H & 0 \\ 0 & 0 & 1 \end{bmatrix}$ matrix.

```
In [144]: #1.75 along x axis and 1.25 along y axis
          sx, sy = 1.75, 1.25
          scale_matrix = np.array([[sx,0,0],[0,sy,0],[0,0,1]])
          img1 = ndi.affine_transform(img, scale_matrix)
          show_images(img, img1)
```



Now while rotating the image we take the angle as pi/6. We now rotate the image by multiplying with the transformation matrix $\begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$

```
In [145]: theta = np.pi/6 #30 degree
          #counterclockwise rotation
          rotate_matrix = np.array([[np.cos(theta),np.sin(theta),0],[-np.sin(theta), np.co
          img2 = ndi.affine_transform(img, rotate_matrix)
          show_images(img, img2)
```



Now for doing the shear in both the x and y axis we multiply with the transformation matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ tan\theta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In [146]:
```
lambdaa = 0.3
shear_matrix = np.array([[1,lambdaa,0],[lambdaa,1,0],[0,0,1]])
img3 = ndi.affine_transform(img, shear_matrix)
show_images(img, img3)
```