Department Of Robotics and Mechatronics Engineering

University Of Dhaka

Laboratory Report

Course Code: 4212

Course Name: Digital Image Processing Lab

Lab Report No: 02

Lab Group No: Null

Experiment Name: Image Transformation Operation

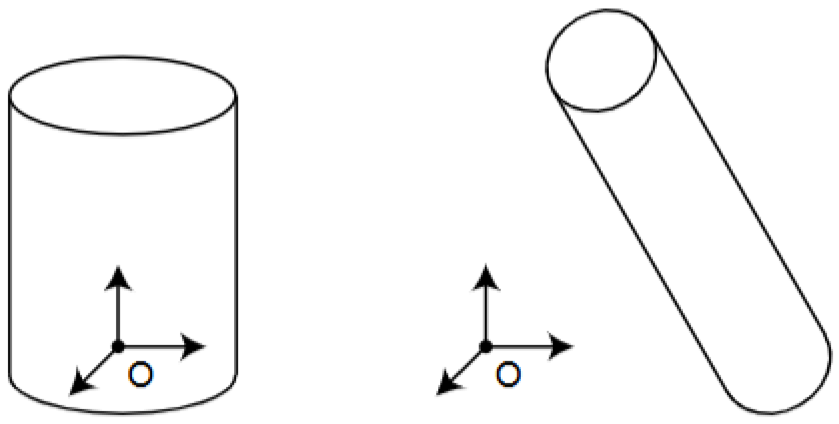
Experiment Date: 01 May, 2025.

Date Of Submission: 02 May, 2025.

Submitted To – Dr. Mehedi Hasan Associate Professor Department of Robotics and Mechatronics Engineering Faculty of Engineering and Technology	**Authored By –** • Mirza Afnan Islam (AE-172-018)

Introduction:

Image resizing (also called image scaling or resampling) is the process of changing the dimensions of a digital image by either increasing (upscaling) or decreasing (downscaling) its size. This is a fundamental operation in image processing and computer vision. In the image resizing process the dimension of an image is altered to meet specific requirements in image processing and computer vision tasks. Image resizing involves changing the number of pixels in the image, which affects both its visual size, and the amount of data needed to represent it. When an image is resized, either new pixels are created (during enlargement) or some pixels are removed (during reduction). The key challenge in resizing is determining how to map the pixel values from the original image to the resized image in a way that maintains visual quality. Bilinear interpolation is an extension of linear interpolation to a two-dimensional space. It approximates the value at a certain point within a grid by sampling the coordinates with values of four other grid points. The outstanding thing about the interpolation is that this is done in one direction (x or y) and then in the other direction, making it bilinear. Bicubic Interpolation is an interpolation technique used to upscale or downscale images by calculating pixel values based on the surrounding pixels. It estimates the color or intensity of new pixels based on a weighted average of neighboring pixels, resulting in a smoother and more accurate representation of the image. By using a 4×4 pixels grid around the target pixel, Bicubic Interpolation considers more information than simpler interpolation methods like Nearest Neighbor or Bilinear Interpolation. The additional data allows for a more precise estimation of pixel values, leading to higher-quality and better-detailed images. An affine transformation is a type of geometric transformation that preserves lines and parallelism. It allows for the scaling, rotation, translation (shifting), and shearing of an image. The figure below shows an example of what we mean. On the left, a cylinder has been built in a convenient place, and to a convenient size. Because of the requirements of a scene, it is first scaled to be longer and thinner than its original design, rotated to a desired orientation in space, and then moved to a desired position (i.e. translated). The set of operations providing for all such transformations, are known as the affine transforms. The affine include translations and all linear transformations, like scale, rotate, and shear.



Original cylinder model

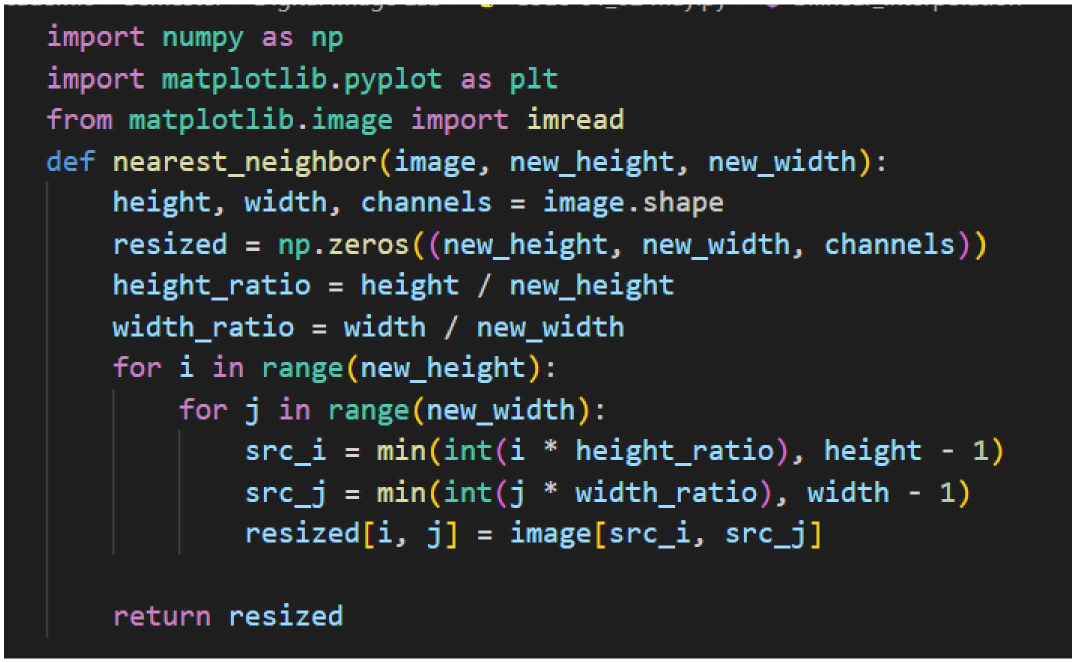
Transformed cylinder. It has been scaled, rotated, and translated

Methodology and Code Explanation:

Code-01:

Initially we imported NumPy and Matplotlib library to read images. In this portion we have to do resizing using Nearest Neighbor Method, Bi-linear and bi-cubic interpolation.

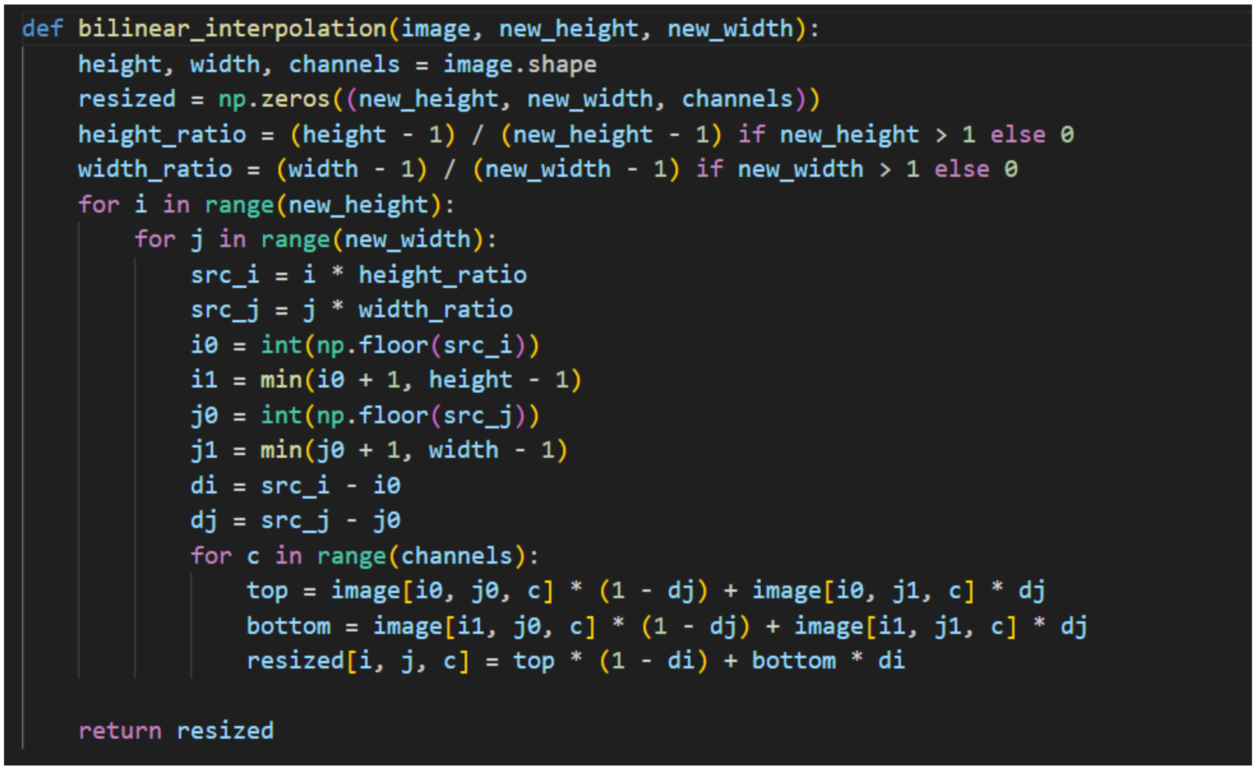
The first resizing method implemented is Nearest Neighbor Interpolation, which is a simple technique where each pixel in the resized image is assigned the value of the closest pixel in the original image. This method does not consider smooth transitions and can result in a blocky or pixelated appearance. Here, we compute the source pixel indices by scaling the target dimensions back to the original size, using simple rounding (int()), and directly copying the nearest pixel's color values.



```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.image import imread
def nearest_neighbor(image, new_height, new_width):
    height, width, channels = image.shape
    resized = np.zeros((new_height, new_width, channels))
    height_ratio = height / new_height
    width_ratio = width / new_width
    for i in range(new_height):
        for j in range(new_width):
            src_i = min(int(i * height_ratio), height - 1)
            src_j = min(int(j * width_ratio), width - 1)
            resized[i, j] = image[src_i, src_j]

    return resized
```

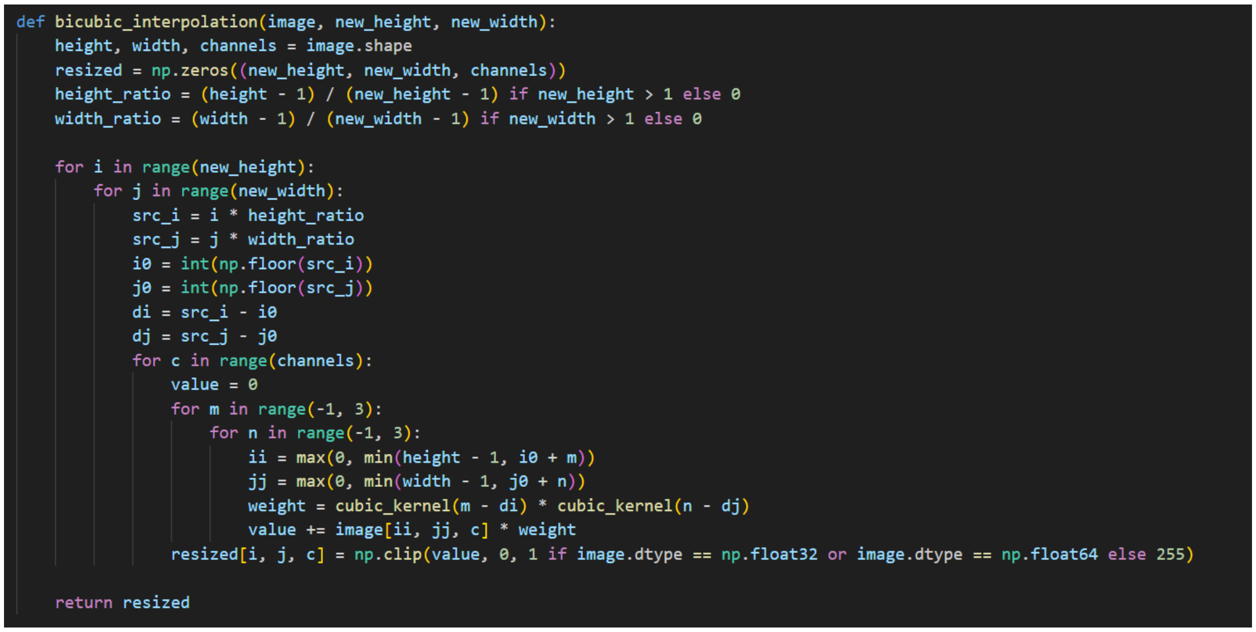
The second method, Bilinear Interpolation, improves image quality by considering the four nearest neighbors and computing a weighted average based on their distances. This results in smoother gradients and fewer visible edges. In this method, we interpolate horizontally and vertically between pixels to compute a smooth transition. The fractional distances di and dj determine how the pixels are blended.



```python
def bilinear_interpolation(image, new_height, new_width):
    height, width, channels = image.shape
    resized = np.zeros((new_height, new_width, channels))
    height_ratio = (height - 1) / (new_height - 1) if new_height > 1 else 0
    width_ratio = (width - 1) / (new_width - 1) if new_width > 1 else 0
    for i in range(new_height):
        for j in range(new_width):
            src_i = i * height_ratio
            src_j = j * width_ratio
            i0 = int(np.floor(src_i))
            i1 = min(i0 + 1, height - 1)
            j0 = int(np.floor(src_j))
            j1 = min(j0 + 1, width - 1)
            di = src_i - i0
            dj = src_j - j0
            for c in range(channels):
                top = image[i0, j0, c] * (1 - dj) + image[i0, j1, c] * dj
                bottom = image[i1, j0, c] * (1 - dj) + image[i1, j1, c] * dj
                resized[i, j, c] = top * (1 - di) + bottom * di

    return resized
```

The most advanced method used is **Bicubic Interpolation**, which considers a 4x4 grid of neighboring pixels (16 in total) and applies a cubic kernel function to compute weights. This technique yields very smooth and high-quality resized images. Here, we loop over the 4x4 surrounding pixels and compute the weighted sum using the cubic kernel. This approach provides the best visual results among the three methods.

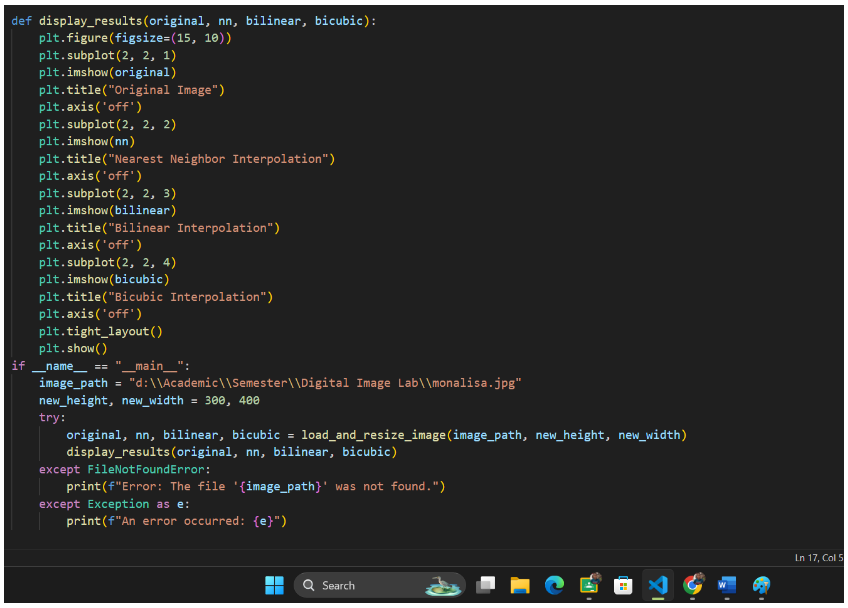


```python
def bicubic_interpolation(image, new_height, new_width):
    height, width, channels = image.shape
    resized = np.zeros((new_height, new_width, channels))
    height_ratio = (height - 1) / (new_height - 1) if new_height > 1 else 0
    width_ratio = (width - 1) / (new_width - 1) if new_width > 1 else 0

    for i in range(new_height):
        for j in range(new_width):
            src_i = i * height_ratio
            src_j = j * width_ratio
            i0 = int(np.floor(src_i))
            j0 = int(np.floor(src_j))
            di = src_i - i0
            dj = src_j - j0
            for c in range(channels):
                value = 0
                for m in range(-1, 3):
                    for n in range(-1, 3):
                        ii = max(0, min(height - 1, i0 + m))
                        jj = max(0, min(width - 1, j0 + n))
                        weight = cubic_kernel(m - di) * cubic_kernel(n - dj)
                        value += image[ii, jj, c] * weight
                resized[i, j, c] = np.clip(value, 0, 1 if image.dtype == np.float32 or image.dtype == np.float64 else 255)

    return resized
```

Then we displayed image by using matplotlib subsections.

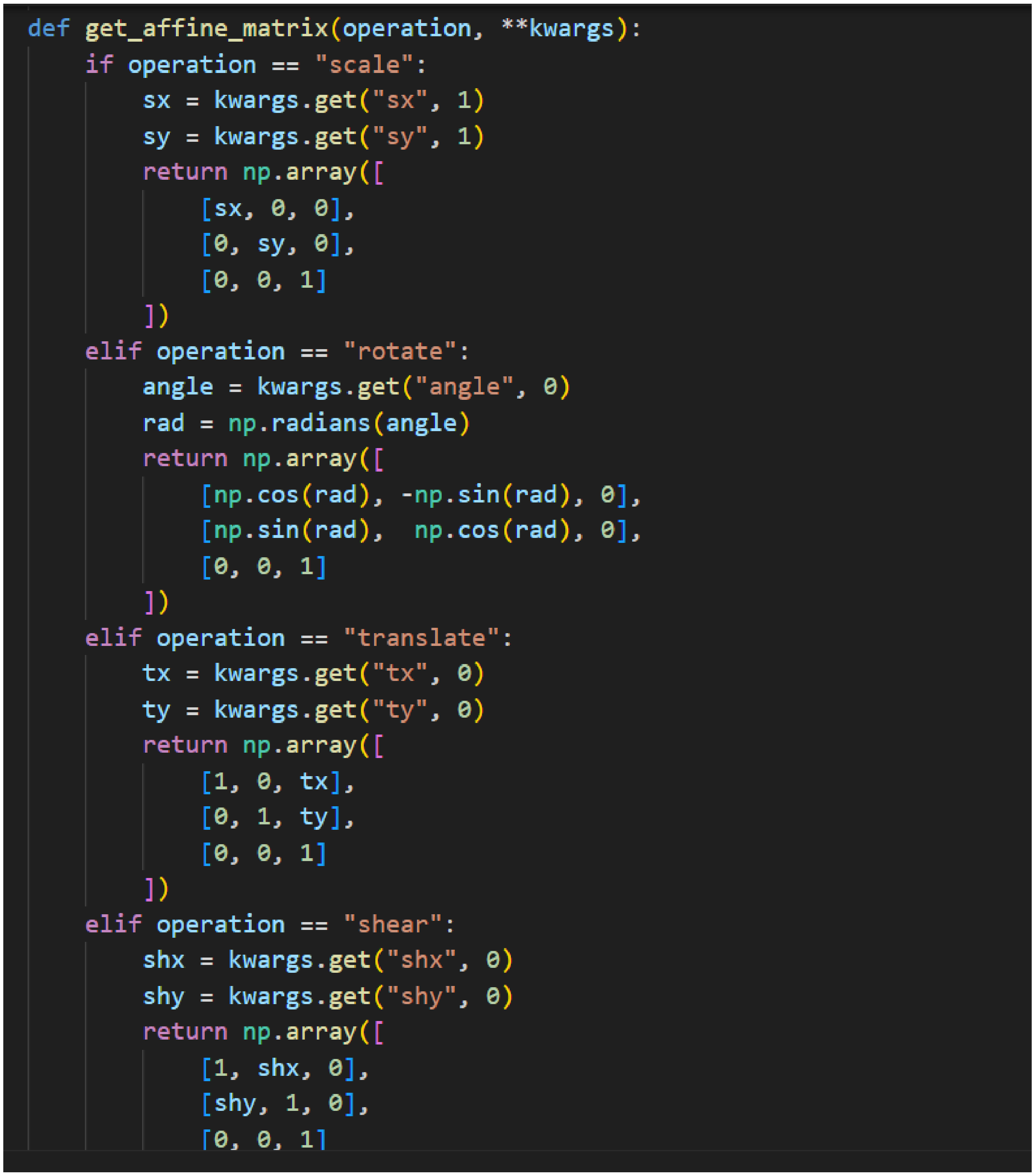


```python
def display_results(original, nn, bilinear, bicubic):
    plt.figure(figsize=(15, 10))
    plt.subplot(2, 2, 1)
    plt.imshow(original)
    plt.title("Original Image")
    plt.axis('off')
    plt.subplot(2, 2, 2)
    plt.imshow(nn)
    plt.title("Nearest Neighbor Interpolation")
    plt.axis('off')
    plt.subplot(2, 2, 3)
    plt.imshow(bilinear)
    plt.title("Bilinear Interpolation")
    plt.axis('off')
    plt.subplot(2, 2, 4)
    plt.imshow(bicubic)
    plt.title("Bicubic Interpolation")
    plt.axis('off')
    plt.tight_layout()
    plt.show()
if __name__ == "__main__":
    image_path = "d:\\Academic\\Semester\\Digital Image Lab\\monalisa.jpg"
    new_height, new_width = 300, 400
    try:
        original, nn, bilinear, bicubic = load_and_resize_image(image_path, new_height, new_width)
        display_results(original, nn, bilinear, bicubic)
    except FileNotFoundError:
        print(f"Error: The file '{image_path}' was not found.")
    except Exception as e:
        print(f"An error occurred: {e}")
```

Code-02:

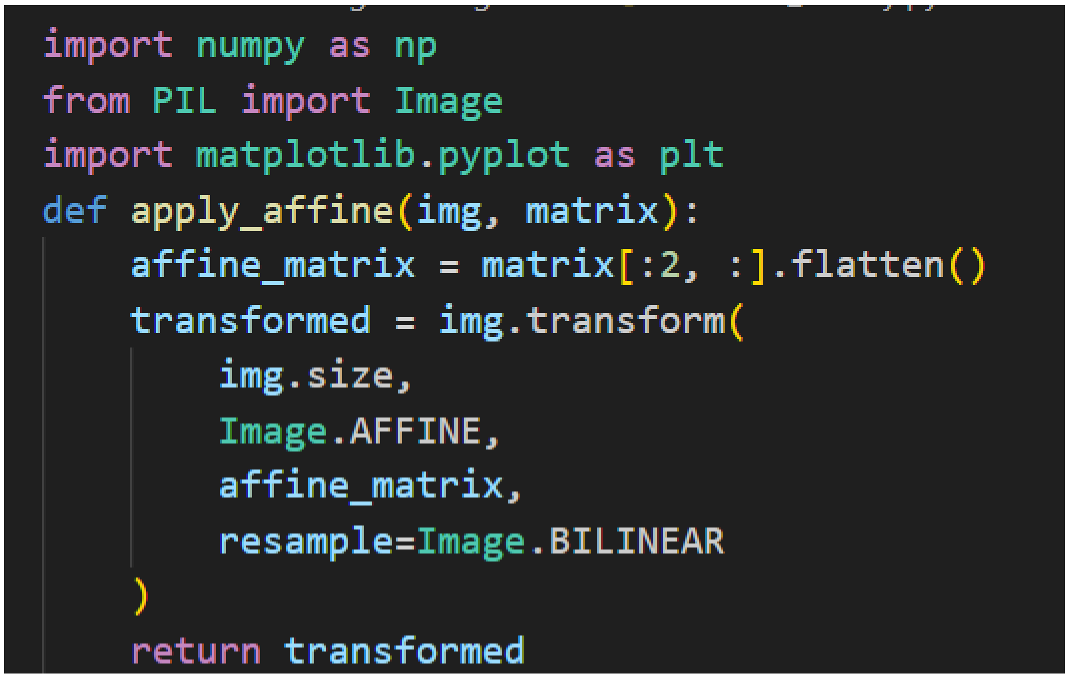
To perform geometric transformations like scaling, rotation, translation, and shearing on an image without using OpenCV, we can utilize Python libraries such as Pillow (PIL), NumPy, and Matplotlib. Affine transformation is a linear mapping method that preserves points, straight lines, and planes. It uses a 3×3 matrix to apply transformations. The top 2 rows of this matrix are used in image transformations, where each transformation can be represented as its own matrix.

We begin by importing the required libraries: numpy for mathematical operations, PIL.Image for image loading and manipulation, and matplotlib.pyplot for visualization. The function get_affine_matrix() generates a transformation matrix based on the operation type (scale, rotate, translate, or shear). For scaling, we use the matrix [[sx, 0, 0], [0, sy, 0], [0, 0, 1]], where sx and sy are scale factors along the x and y axes. For rotation, the matrix is defined as [[cos(θ), -sin(θ), 0], [sin(θ), cos(θ), 0], [0, 0, 1]], where θ is the rotation angle in radians. Translation uses the matrix [[1, 0, tx], [0, 1, ty], [0, 0, 1]], where tx and ty shift the image along the x and y directions. Shearing is performed using the matrix [[1, shx, 0], [shy, 1, 0], [0, 0, 1]], where shx and shy represent shear factors.



```python
def get_affine_matrix(operation, **kwargs):
    if operation == "scale":
        sx = kwargs.get("sx", 1)
        sy = kwargs.get("sy", 1)
        return np.array([
            [sx, 0, 0],
            [0, sy, 0],
            [0, 0, 1]
        ])
    elif operation == "rotate":
        angle = kwargs.get("angle", 0)
        rad = np.radians(angle)
        return np.array([
            [np.cos(rad), -np.sin(rad), 0],
            [np.sin(rad),  np.cos(rad), 0],
            [0, 0, 1]
        ])
    elif operation == "translate":
        tx = kwargs.get("tx", 0)
        ty = kwargs.get("ty", 0)
        return np.array([
            [1, 0, tx],
            [0, 1, ty],
            [0, 0, 1]
        ])
    elif operation == "shear":
        shx = kwargs.get("shx", 0)
        shy = kwargs.get("shy", 0)
        return np.array([
            [1, shx, 0],
            [shy, 1, 0],
            [0, 0, 1]
```

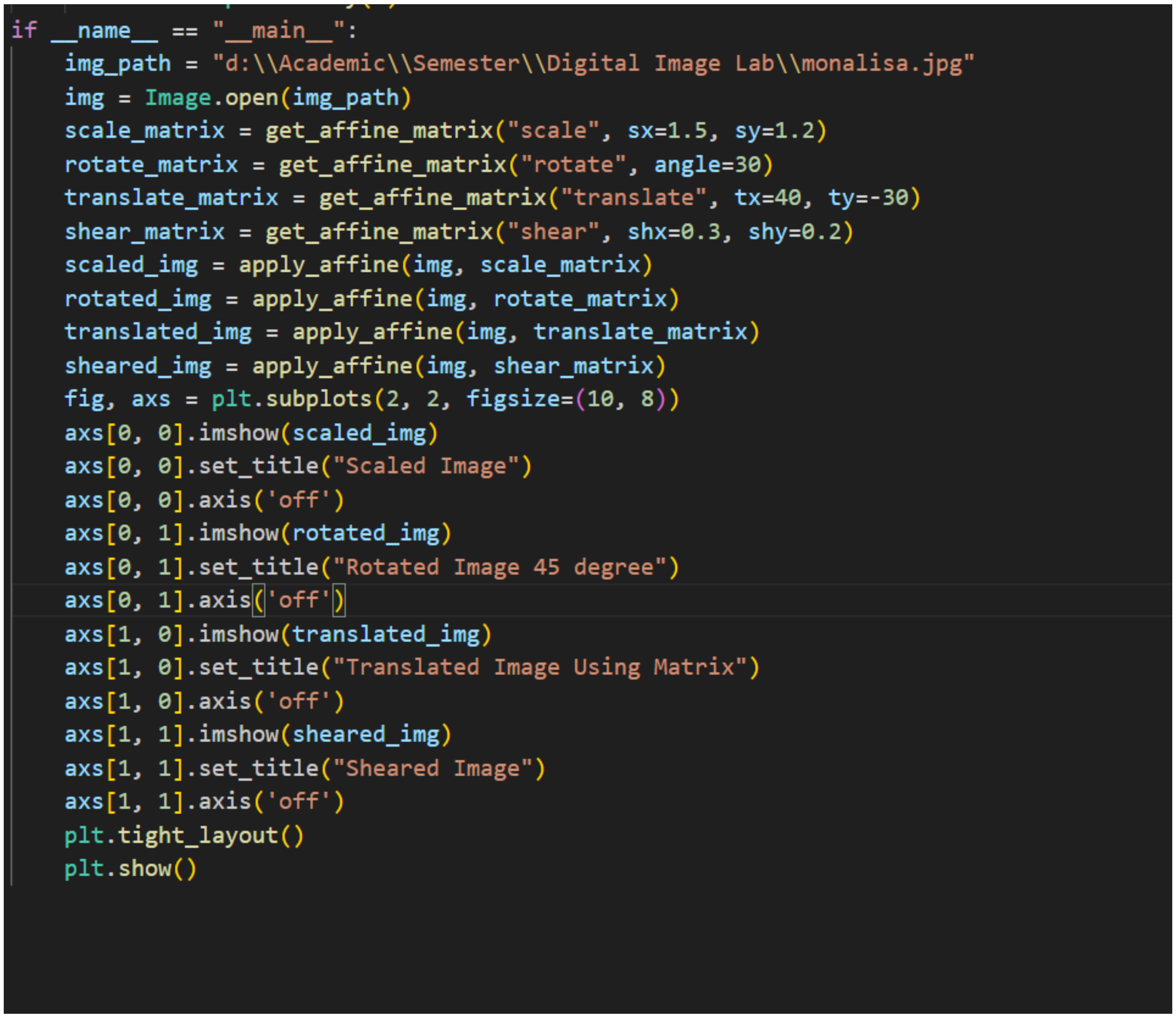
The apply_affine() function takes an image and a transformation matrix, flattens the first two rows (since PIL expects a 2×3 matrix), and applies the transformation using the img.transform() method with bilinear resampling for smoother output.



```python
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
def apply_affine(img, matrix):
    affine_matrix = matrix[:2, :].flatten()
    transformed = img.transform(
        img.size,
        Image.AFFINE,
        affine_matrix,
        resample=Image.BILINEAR
    )
    return transformed
```

In the main block, we load the original image using PIL's Image.open() and create four matrices for scaling, rotating, translating, and shearing respectively. Each matrix is passed to the apply_affine() function to get the corresponding transformed image.

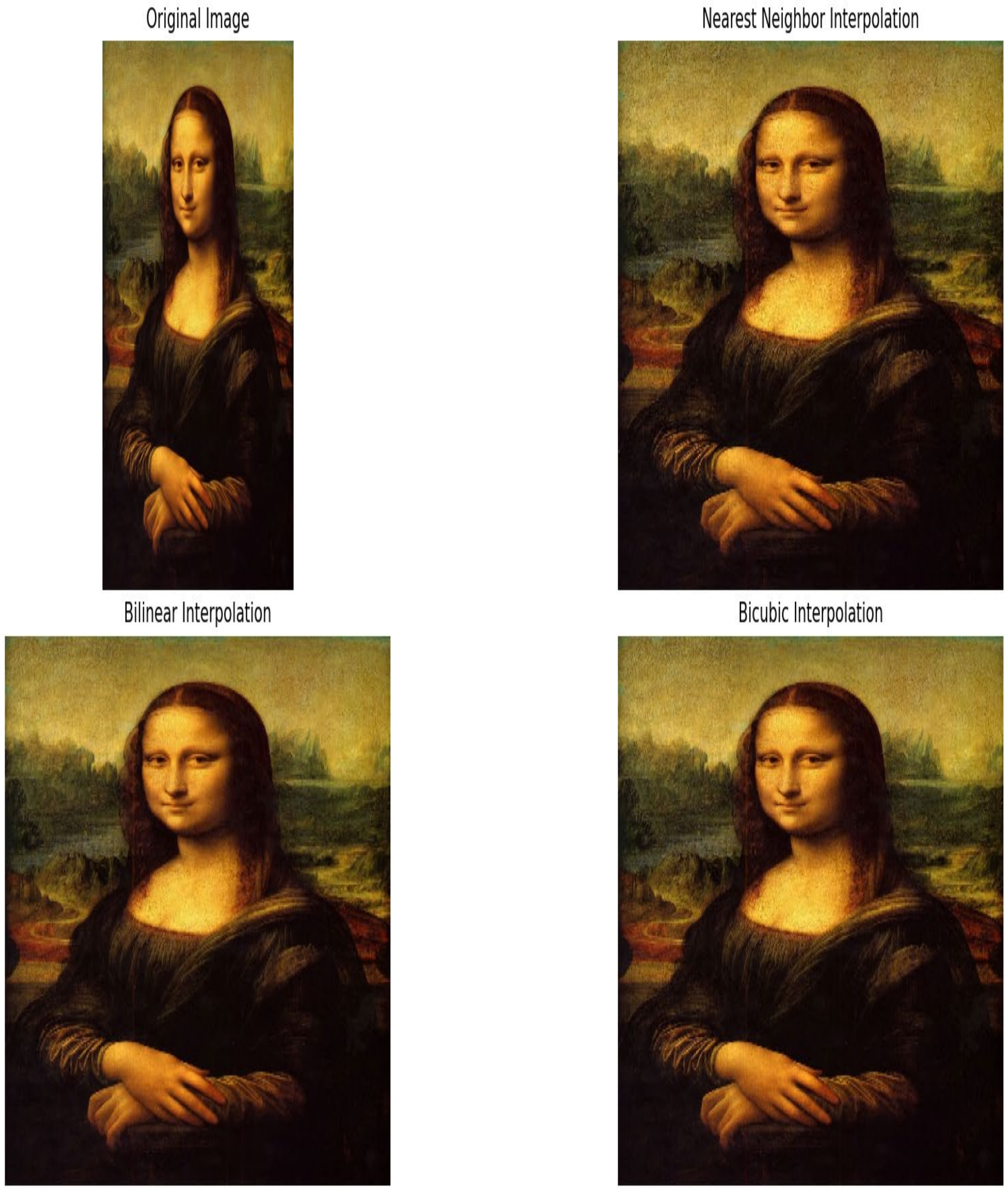
Finally, to visualize the effects, we use Matplotlib's subplots() to display all four images in a 2×2 grid. Each subplot shows one of the transformations — scaled, rotated, translated, or sheared — with titles and no axes for clarity. This demonstrates how each mathematical matrix independently alters the original image, visually highlighting the nature of affine transformations.

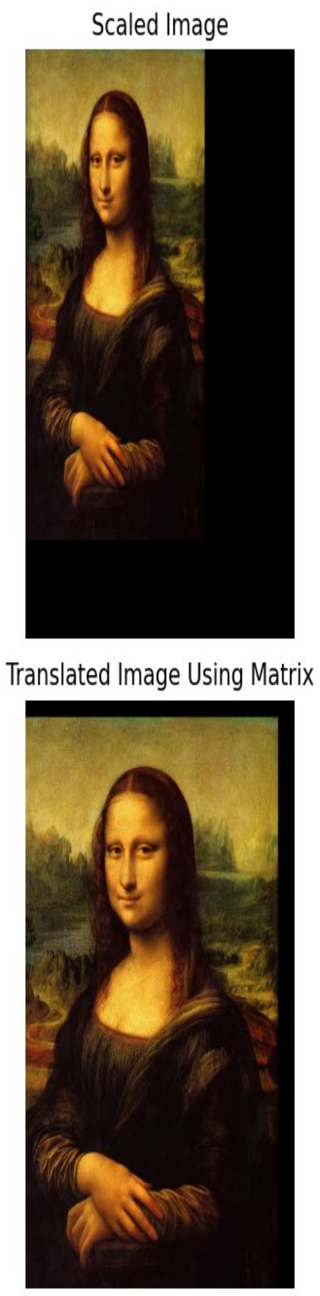


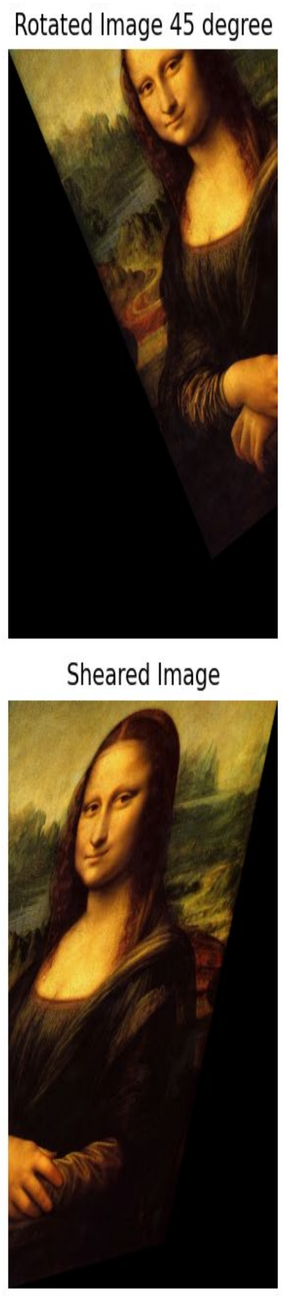
```python
if __name__ == "__main__":
    img_path = "d:\\Academic\\Semester\\Digital Image Lab\\monalisa.jpg"
    img = Image.open(img_path)
    scale_matrix = get_affine_matrix("scale", sx=1.5, sy=1.2)
    rotate_matrix = get_affine_matrix("rotate", angle=30)
    translate_matrix = get_affine_matrix("translate", tx=40, ty=-30)
    shear_matrix = get_affine_matrix("shear", shx=0.3, shy=0.2)
    scaled_img = apply_affine(img, scale_matrix)
    rotated_img = apply_affine(img, rotate_matrix)
    translated_img = apply_affine(img, translate_matrix)
    sheared_img = apply_affine(img, shear_matrix)
    fig, axs = plt.subplots(2, 2, figsize=(10, 8))
    axs[0, 0].imshow(scaled_img)
    axs[0, 0].set_title("Scaled Image")
    axs[0, 0].axis('off')
    axs[0, 1].imshow(rotated_img)
    axs[0, 1].set_title("Rotated Image 45 degree")
    axs[0, 1].axis('off')
    axs[1, 0].imshow(translated_img)
    axs[1, 0].set_title("Translated Image Using Matrix")
    axs[1, 0].axis('off')
    axs[1, 1].imshow(sheared_img)
    axs[1, 1].set_title("Sheared Image")
    axs[1, 1].axis('off')
    plt.tight_layout()
    plt.show()
```

Result:

```
1
```


Original Image
Nearest Neighbor Interpolation
Bilinear Interpolation
Bicubic Interpolation





References:

1. www.geeksforgeeks.com
2. www. W3schools.com
3. www. Icodeschool.com