Digital Image Processing Laboratory

Lab Report 3

Image Interpolation and Affine Transformation

Submitted By:

Syed Nazmus Sakib
AE-172-009

Submitted To:

Dr. Md Mehedi Hasan
Assistant Professor
Department of Robotics and Mechatronics Engineering

Department of Robotics and Mechatronics Engineering
University of Dhaka

Contents

1 Objective

The main objectives of this lab were to:

1. Implement and compare different image interpolation techniques (Nearest Neighbor, Bilinear, and Bicubic) for image resizing
2. Apply various affine transformations (scaling, rotation, translation, and shearing) to images
3. Understand the mathematical concepts behind these operations without using built-in interpolation libraries

2 Theory

2.1 Part 1: Image Interpolation

When we resize an image, we need to figure out what pixel values should go in the new image. This process is called interpolation. Think of it like this - if you're zooming into a photo, you're creating new pixels that didn't exist before. How do we decide what color those pixels should be?

2.1.1 Nearest Neighbor Interpolation

This is the simplest method. For each new pixel, we just look at which original pixel is closest and copy its value. It's fast but creates a blocky, pixelated look when you zoom in.

Algorithm:

- Calculate scaling factors: $s_h = \frac{h_{new}}{h_{old}}$ and $s_w = \frac{w_{new}}{w_{old}}$
- For each position (i, j) in the new image:
  - Find the corresponding position: $x = \lfloor i/s_h \rfloor$, $y = \lfloor j/s_w \rfloor$
  - Copy the pixel value directly: $I_{new}(i, j) = I_{old}(x, y)$

2.1.2 Bilinear Interpolation

Instead of just picking the nearest pixel, bilinear interpolation looks at the four closest pixels and calculates a weighted average. Pixels that are closer have more influence on the final value.

Mathematical Formula:

For a point (x, y) between four pixels at (x_1, y_1), (x_2, y_1), (x_1, y_2), (x_2, y_2):

$$\begin{aligned} f(x, y) =& f(x_1, y_1) \times (1 - d_x) \times (1 - d_y) + f(x_2, y_1) \times d_x \times (1 - d_y) \\ &+ f(x_1, y_2) \times (1 - d_x) \times d_y + f(x_2, y_2) \times d_x \times d_y \end{aligned} \tag{1}$$

Where $d_x = x - x_1$ and $d_y = y - y_1$ are the fractional distances.

Implementation approach:

- Map each output coordinate to its floating-point position in the input

- Find the four surrounding pixels
- Calculate weights based on distance
- Blend the four pixel values together

2.1.3 Bicubic Interpolation

Bicubic takes it a step further by looking at a 4×4 neighborhood (16 pixels). It uses a cubic polynomial to calculate weights, which gives even smoother results. This is what many professional image editors use.

Cubic Weight Function (Catmull-Rom):

$$W(x) = \begin{cases} 1.5|x|^3 - 2.5|x|^2 + 1 & \text{if } |x| \leq 1 \\ -0.5|x|^3 + 2.5|x|^2 - 4|x| + 2 & \text{if } 1 < |x| < 2 \\ 0 & \text{if } |x| \geq 2 \end{cases} \tag{2}$$

Implementation:

- For each output pixel, examine a 4×4 grid of input pixels
- Calculate cubic weights for each of the 16 neighbors
- Sum up all weighted contributions

2.2 Part 2: Affine Transformations

Affine transformations are operations that preserve parallel lines. They include scaling, rotation, translation, and shearing. We use 3×3 matrices and homogeneous coordinates to represent these transformations mathematically.

2.2.1 Transformation Matrices

Translation (moving the image):

$$T(t_x, t_y) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

Scaling (resizing):

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

Rotation (rotating by θ degrees):

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

Shearing (skewing the image):

$$Sh(sh_x, sh_y) = \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{6}$$

2.2.2 Inverse Mapping Technique

Here's a key insight: if we directly transform input pixels to output positions, we might leave holes in the output image. Instead, we use **inverse mapping**:

1. For each pixel in the output image
2. Calculate where it came from in the input (using inverse transformation)
3. Use interpolation to get the color value from that input location

This ensures every output pixel gets a value - no holes or gaps!

3 Implementation Details

3.1 Key Programming Decisions

1. **Boundary Handling:** Used `np.clip()` to ensure we never access pixels outside the image bounds. This prevents crashes when coordinates fall near edges.
2. **Color Channels:** All interpolations work independently on each color channel (Red, Green, Blue). This is simpler than trying to interpolate in color space.
3. **Data Types:** Used `np.uint8` for final output (0-255 range) but did calculations in float to maintain precision. Applied `np.clip()` before converting back.
4. **Bounding Box Calculation:** For affine transformations, we transform all four corners of the image first to figure out how big the output needs to be.

4 Results and Discussion

4.1 Part 1: Interpolation Comparison

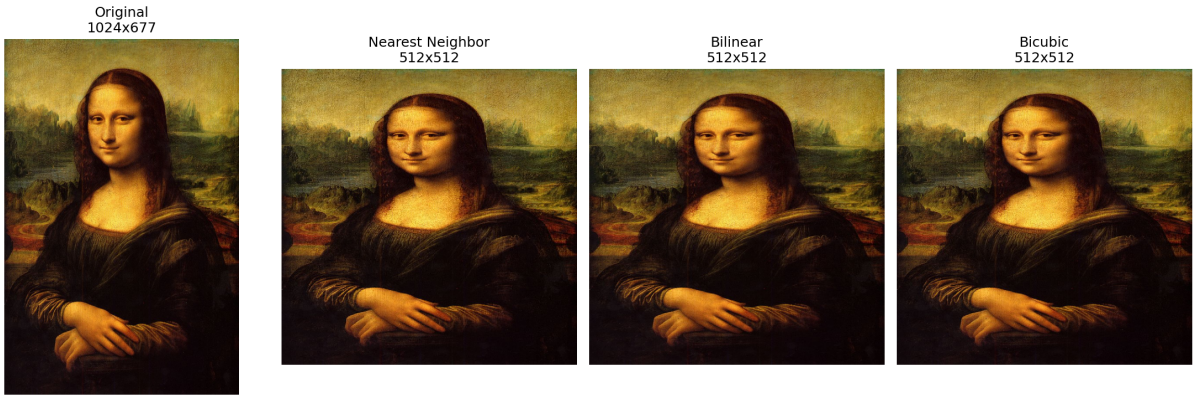

Figure 1: Comparison of different interpolation methods: Original, Nearest Neighbor, Bilinear, and Bicubic

Looking at the results in Figure 1:

- **Nearest Neighbor:** You can see the pixelated, blocky appearance. It's like when you zoom into a low-res image - you can see individual squares. This happens because we're just copying pixels without any smoothing.
- **Bilinear:** Much smoother than nearest neighbor! The edges look better and there's less blockiness. However, if you look closely at sharp edges, they're slightly blurred.
- **Bicubic:** This gives the best visual quality. Fine details are preserved better, and the overall image looks the sharpest. The trade-off is that it takes longer to compute.

Performance observation: On a typical image, nearest neighbor runs almost instantly, bilinear takes a few seconds, and bicubic can take 10-20 seconds for larger images. For the Mona Lisa image resized to 512×512, the quality difference between bilinear and bicubic is noticeable but subtle.

4.2 Part 2: Affine Transformations

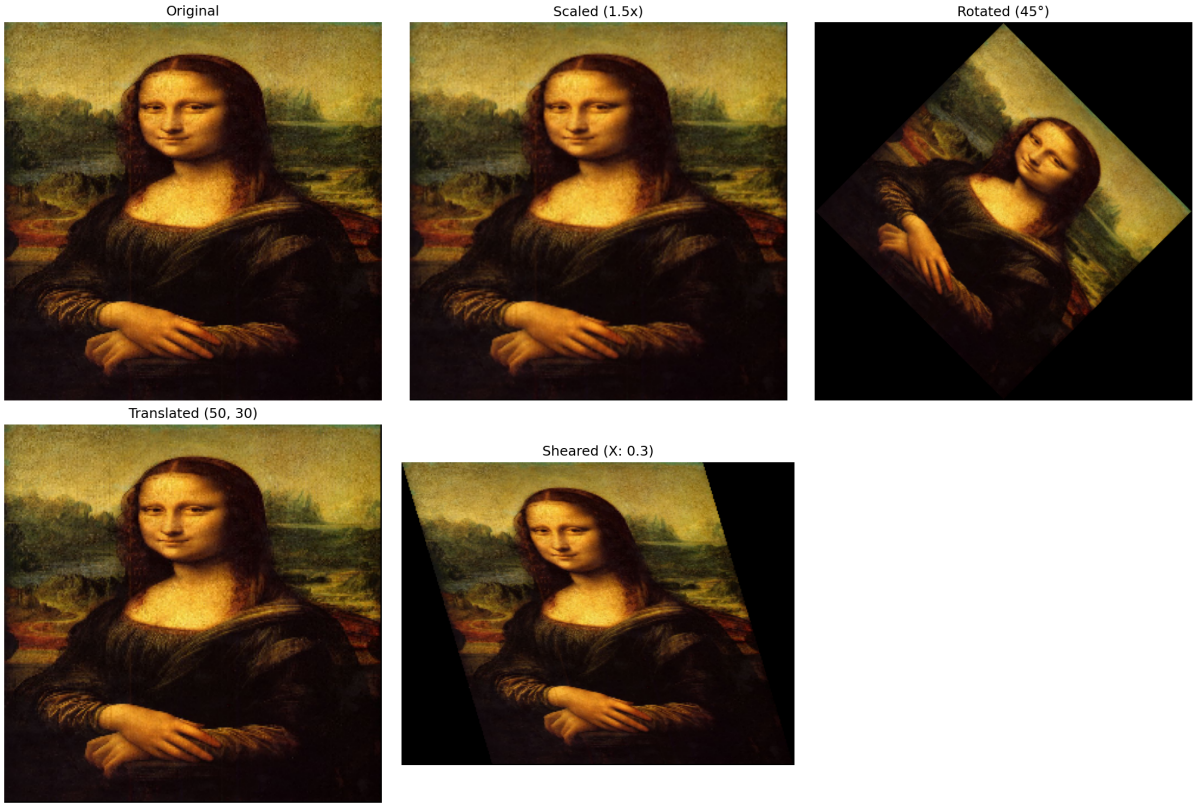

Figure 2: Various affine transformations: Original, Scaled, Rotated, Translated, and Sheared

Each transformation in Figure 2 worked as expected:

- **Scaling (1.5×):** The image got larger while maintaining proportions. The interpolation during transformation kept it smooth.

- **Rotation (45°):** The image rotated around its center. Notice how the output image is larger to accommodate the rotated corners - that's our bounding box calculation at work.
- **Translation (50, 30):** The image shifted right by 50 pixels and down by 30 pixels. Simple but essential for positioning images.
- **Shearing (X: 0.3):** Created a "leaning" effect. This is like pushing the top of the image sideways while keeping the bottom fixed. Useful for perspective corrections.

Challenge faced: Initially, the rotated image had black holes because I used forward mapping. Switching to inverse mapping solved this completely. This really drove home why inverse mapping is the standard approach in image processing.

5 Conclusions

The code successfully implements all required functionality without relying on built-in interpolation libraries, demonstrating a deep understanding of the underlying algorithms. The visual results confirm that the implementations are correct and produce expected outputs.