VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Report

Lab 1: OpenCV - Basic

Course: Digital Image and Video Processing

Supervisors:

Prof. Dr. Ly Quoc Ngoc

MSc. Nguyen Manh Hung

MSc. Pham Thanh Tung

Student:

23127266 - Nguyen Anh Thu

November 28, 2025

Contents

List of Figures

List of Tables

Chapter I. Introduction

1. Overview

In this assignment, I implement fundamental image processing algorithms that operate on two–dimensional (2D) images. A digital image is represented as a matrix of size $H \times W$, where each element corresponds to a pixel. For grayscale images, each pixel has an intensity value in the range 0 to 255. For color images, each pixel is a triplet of values corresponding to the three channels: R (Red), G (Green), and B (Blue).

This assignment focuses on re-implementing the following algorithms: color transformation and geometric transformation algorithms

After implementing each algorithm, I compare the output produced by my NumPy-based implementation with the result generated by the corresponding OpenCV function to evaluate the accuracy of the implementation.

2. Objectives

Table 1: Main Tasks of the Assignment

Task	**Details**
Color Transformations	Linear mapping (brightness, contrast, brightness + contrast) Non-linear mapping (log, gamma) Histogram equalization Histogram specification
Geometric Transformations	Pixel coordinate transforms (bilinear, affine) Backward mapping (inverse mapping) Interpolation (nearest-neighbor, bilinear) Scaling, rotation, shear
Evaluation	Comparison with OpenCV results

3. Program structure

The program works directly with image files as input. Each image is loaded into a NumPy array representing pixel values in either grayscale or RGB format.

Input:

- image file (PNG, JPG, JPEG);

- selected transformation (color, geometric); scaling factor, or kernel size depending on the algorithm.

Output:

- processed image as a NumPy array;
- saved output image results.

Chapter II. Implementation

1. Color Transformation

1.1. Problem Statement

Given a digital image represented as a discrete function

$$f(x, y) : \Omega \subset \mathbb{Z}^2 \to [0, 255]^C,$$

where $C = 1$ for grayscale images and $C = 3$ for RGB color images, the goal of **color transformation** is to obtain a modified image $g(x, y)$ by applying an intensity-mapping function or a channel-wise transformation.

Color transformations are defined as point-wise operations of the form:

$$g(x, y) = T(f(x, y)),$$

[3]

where $T(\cdot)$ is a mapping applied independently to each pixel or to each color channel.

In this project, the transformations implemented include:

- linear intensity mappings (brightness adjustment, contrast adjustment, and combined brightness–contrast mapping),
- nonlinear mappings (logarithmic and gamma correction),
- histogram-based transformations (histogram equalization and histogram specification).

Input: an 8-bit grayscale or RGB image $f(x, y)$.

Output: a transformed image $g(x, y)$ obtained by applying the selected intensity transformation while preserving image dimensions and data range (clipping pixel values to $[0, 255]$ when necessary).

The objective is to enhance or modify the global or local intensity distribution while maintaining pixel-wise independence of the transformation.

1.2. Linear Mappings

1.2.1. Brightness Adjustment

(a) Theory

Brightness modifies pixel intensity by adding a constant value:

$$g(x, y) = f(x, y) + b,$$

[3]

where b is the brightness offset.

- $b > 0$: image becomes brighter
- $b < 0$: image becomes darker

Pixel values are clipped to stay within $[0, 255]$:

$$g(x, y) = \min(\max(g(x, y), 0), 255).$$

(b) Algorithm (Step-by-step)

- Read input image as NumPy array
- For each pixel, add brightness offset b
- Clip to $[0, 255]$
- Return the modified image

(c) Implementation Details

Function name: `brightness_adjust`

Parameters: - `image`: input grayscale/RGB image - `b`: brightness offset

Input: an image with pixel values in $[0, 255]$ *Output:* image after brightness adjustment

(d) Pseudo-code

```
Function brightness_adjust(image, b):
    For each pixel (x, y):
        g(x, y) = image(x, y) + b
    Clip g to [0, 255]
    Return g
```

1.2.2. Contrast Adjustment

(a) Theory

Contrast modifies the spread of pixel intensities by scaling:

$$g(x, y) = a \cdot f(x, y),$$

where a is the contrast coefficient.

- $a > 1$: increases contrast - $0 < a < 1$: decreases contrast

Clipping is required:

$$g(x, y) = \min(\max(g(x, y), 0), 255).$$

(b) Algorithm (Step-by-step)

- Read image - Multiply all pixel values by a - Clip to valid range - Return output

(c) Implementation Details

Function name: `contrast_adjust`

Parameters: - `image`: input image - `a`: contrast gain

Input: pixel values in $[0, 255]$ *Output:* contrast-scaled image

(d) Pseudo-code

```
Function contrast_adjust(image, a):
    For each pixel (x, y):
        g(x, y) = a * image(x, y)
```

```
    Clip g to [0, 255]
    Return g
```

1.2.3. Brightness and Contrast Adjustment

(a) Theory

This is the general linear model combining scaling and shifting:

$$g(x, y) = a \cdot f(x, y) + b.$$

[3]

This allows simultaneous adjustment of brightness and contrast.

Clipping still applies:

$$g(x, y) = \min(\max(g(x, y), 0), 255).$$

(b) Algorithm (Step-by-step)

- Read image - Apply $g = a \cdot f + b$ - Clip to [0, 255] - Convert to uint8

(c) Implementation Details

Function name: `linear_mapping`

Parameters: - `image`: input image - `a`: contrast - `b`: brightness

Input: grayscale or RGB image *Output:* transformed image

(d) Pseudo-code

```
Function linear_mapping(image, a, b):
    For each pixel (x, y):
        g(x, y) = a * image(x, y) + b
    Clip g to [0, 255]
    Return g
```

1.3. Non-linear Mappings

Non-linear intensity transformations modify pixel values using non-linear functions. Two widely used operations are logarithmic correction and gamma correction. These operations enhance specific intensity ranges more effectively than linear mappings.

1.3.1. Logarithmic Mapping

(a) Theory

The logarithmic transformation enhances dark regions of the image by applying:

$$g(x, y) = c \cdot \log(1 + f(x, y)),$$

[3]

where:

- $f(x, y)$: input pixel value - $g(x, y)$: transformed output pixel - c: scaling constant

Because the logarithmic function increases slowly for large values and quickly for small values, it stretches dark intensities and compresses bright intensities.

Pixel values are normalized before processing:

$$f_n(x, y) = \frac{f(x, y)}{255}$$

and rescaled after transformation:

$$g(x, y) = 255 \cdot \frac{g(x, y)}{\max(g)}.$$

(b) Algorithm (Step-by-step)

- Read image as NumPy array - Normalize pixel values to $[0, 1]$ - Apply logarithmic formula - Scale back to $[0, 255]$ - Clip and convert to uint8

(c) Implementation Details

Function name: `log_mapping`

Parameters: - `image`: input image - `c`: scaling constant

Input: grayscale/RGB image *Output:* log-transformed image

(d) Pseudo-code

```
Function log_mapping(image, c):
    fn = image / 255
    g  = c * log(1 + fn)
    g  = g / max(g) * 255
    Clip g to [0, 255]
```

```
    Return g
```

1.3.2. Gamma Mapping

(a) Theory

Gamma correction adjusts brightness nonlinearly using:

$$g(x, y) = 255 \cdot \left(\frac{f(x, y)}{255} \right)^{\gamma},$$

[3]

where:

- $\gamma < 1$: brightens the image - $\gamma > 1$: darkens the image

Normalization ensures the operation is applied proportionally across the range.

Gamma correction is commonly used in display systems and camera pipelines.

(b) Algorithm (Step-by-step)

- Convert image to floating-point - Normalize to $[0, 1]$ - Apply gamma exponent f^{γ} - Multiply by 255 - Clip and return

(c) Implementation Details

Function name: `gamma_mapping`

Parameters: - `image`: input image - `gamma`: gamma exponent

Input: pixel values in range $[0, 255]$ *Output:* gamma-corrected image

(d) Pseudo-code

```
Function gamma_mapping(image, gamma):
    fn = image / 255
    g  = fn ** gamma
    g  = g * 255
    Clip g to [0, 255]
    Return g
```

1.4. Histogram Equalization

1.4.1. Theory

Histogram equalization enhances image contrast by redistributing the pixel intensities so that the histogram becomes approximately uniform. The mapping uses the cumulative distribution

function (CDF):

$$T(k) = \text{round}\left(\frac{L-1}{MN}\sum_{i=0}^{k} H(i)\right),$$

where:

- $H(i)$: histogram count of intensity i - L: number of possible intensity levels (256 for 8-bit images) - $M \times N$: total number of pixels

The CDF stretches frequently occurring intensities and compresses others, increasing global contrast.

1.4.2. (b) Algorithm (Step-by-step)

- Compute histogram H of the grayscale image - Compute cumulative distribution $C(k)$ - Normalize CDF to range $[0, L-1]$ to form lookup table $T(k)$ - For each pixel value $f(x, y)$, map it to $g(x, y) = T[f(x, y)]$ - Return enhanced image

1.4.3. (c) Implementation Details

Function name: `hist_equalization`

Parameters: - `image`: input grayscale image

Input: a single-channel grayscale image with pixel values $[0, 255]$ *Output:* histogram-equalized image

1.4.4. (d) Pseudo-code

```
Function hist_equalization(image):
    Compute histogram H
    Compute CDF C from H
    For each intensity k:
        T(k) = round((L-1) * C(k) / (M*N))
    For each pixel (x, y):
        g(x, y) = T[ image(x, y) ]
    Return g
```

1.5. Histogram Specification

1.5.1. (a) Theory

Histogram specification adjusts the input image so that its histogram matches a given target histogram. It is based on matching CDF values:

$$T(i) = C_f^{-1}(C_t(i)),$$

where:

- C_f: CDF of the input image - C_t: CDF of the target histogram - $T(i)$: mapping from input intensity i to new output intensity

For each input value, we find the intensity in the target whose CDF is the closest.

1.5.2. (b) Algorithm (Step-by-step)

[3]

- Compute histogram and CDF of the input image - Compute target histogram and its CDF - Create mapping table T by matching CDF input - CDF target - For each input pixel, apply mapping:

$$g(x, y) = T[f(x, y)]$$

- Return the specified-image

1.5.3. (c) Implementation Details

Function name: `hist_specification`

Parameters: - `image`: input grayscale image - `target_hist`: array of length 256 containing target histogram

Input: grayscale image + target histogram *Output:* histogram-specified image

1.5.4. (d) Pseudo-code

```
Function hist_specification(image, target_hist):
    Cf = CDF of image
    Ct = CDF of target_hist

    For each intensity i:
```

```
        Find j such that Cf(j) is closest to Ct(i)
        T(i) = j


    For each pixel (x, y):
        g(x, y) = T[ image(x, y) ]
    Return g
```

2. Geometric Transformations

2.1. Problem Statement

Geometric transformations play a fundamental role in digital image processing by modifying the spatial arrangement of pixels while preserving the underlying intensity structure of the image. A digital image is modeled as a discrete sampling of a continuous 2D intensity function

$$f : \Omega \subset \mathbb{Z}^2 \to [0, 255]^C,$$

where Ω is the set of integer pixel coordinates and $C = 1$ for grayscale or $C = 3$ for RGB images. A geometric transformation defines a mapping between coordinates in the input image (source domain) and coordinates in the output image (target domain):

$$T : \mathbb{R}^2 \to \mathbb{R}^2, \qquad (x, y) \mapsto (x', y').$$

In practice, this mapping is represented in homogeneous coordinates:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = A_h \begin{bmatrix} x \\ y \\ 1 \end{bmatrix},$$

where $A_h \in \mathbb{R}^{3\times3}$ is an affine homogeneous matrix. Because forward mapping $(x, y) \mapsto (x', y')$ often produces empty pixels in the output grid, the implementation uses *inverse mapping*: for each output pixel (x', y') we compute its source coordinate (x, y) by applying A_h^{-1}, and then estimate the brightness value at (x', y') by interpolation from $f(x, y)$.

This subsection describes:

1. pixel coordinate transformation models (affine and bilinear),
2. inverse (backward) mapping and brightness interpolation,
3. specific applications: scaling, rotation, translation, and shearing.

2.2. Pixel Coordinate Transformations: Affine and Bilinear Models

2.2.1. Affine transform

a) Theory. According to Lecture 4, an affine transformation is a first–order polynomial mapping of the form

$$\begin{cases} x' = a_0 + a_1 x + a_2 y, \\ y' = b_0 + b_1 x + b_2 y. \end{cases}$$

In homogeneous matrix form,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_0 \\ b_1 & b_2 & b_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

This model can represent scaling, rotation, translation and shearing by choosing appropriate coefficients.

b) Algorithm.

- From the problem parameters (scale factors, rotation angle, translation, shear coefficients), construct the corresponding affine matrix A_h.
- Pass A_h to the general inverse-mapping algorithm to generate the output image.

c) Pseudo-code.

```
# Build affine matrix from parameters (generic template)
Input: sx, sy, theta, tx, ty, kx, ky  # some may be unused
A_h = compose_scaling_rotation_translation_shear(sx, sy,
                                                 theta,
                                                 tx, ty,
                                                 kx, ky)
g   = apply_affine(f, A_h, interp = "bilinear")
```

```
Return g
```

2.2.2. Bilinear transform

a) Theory. The bilinear transform extends the affine model by introducing the cross term xy:

$$\begin{cases} x' = a_0 + a_1 x + a_2 y + a_3 xy, \\ y' = b_0 + b_1 x + b_2 y + b_3 xy. \end{cases}$$

This second–order model allows more complex warping than affine while still being relatively simple to evaluate.

b) Algorithm.

- Estimate the coefficients a_i, b_i from four pairs of corresponding points (e.g., the four corners of a rectangle before and after transformation).
- For each input coordinate (x, y), compute (x', y') using the two polynomials.
- In the project implementation, most operations use affine transforms; bilinear is included to illustrate the extended coordinate model.

c) Pseudo-code.

```
Input: image f, bilinear coefficients a0..a3, b0..b3

For each input pixel (x, y):
    x_prime = a0 + a1*x + a2*y + a3*x*y
    y_prime = b0 + b1*x + b2*y + b3*x*y
    # (x_prime, y_prime) can later be used in forward
   mapping demo

Return mapping field (x_prime, y_prime)
```

2.3. Backward Mapping and Brightness Interpolation

2.3.1. Backward (inverse) mapping

a) Theory. As shown in Lecture 4, the new coordinates (x', y') after transformation generally do not coincide with integer grid positions. To avoid holes in the output image, the brightness at

(x', y') is computed by inverting the planar transform:

$$(x, y) = T^{-1}(x', y'),$$

or, for affine transforms in homogeneous form,

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = A_h^{-1} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}.$$

b) Algorithm.

- Compute the inverse matrix A_h^{-1} once before scanning the output image.
- For each output pixel (x', y'), multiply by A_h^{-1} to obtain the source coordinate (x, y).
- If (x, y) lies outside the source image, assign a background value (e.g., 0).
- Otherwise, pass (x, y) to the interpolation function to estimate $g(x', y')$.

c) Pseudo-code.

```
Input: image f, affine matrix A_h, interpolation mode
   interp

A_inv = inverse(A_h)

For each output pixel (x', y'):
    (x, y, 1)^T = A_inv * (x', y', 1)^T
    If (x, y) outside bounds of f:
        g(x', y') = 0
    Else:
        g(x', y') = interpolate(f, x, y, mode = interp)

Return g
```

2.3.2. Nearest-neighbor interpolation

a) Theory. Nearest-neighbor interpolation assigns to (x', y') the brightness of the closest pixel in the input grid:

$$g(x', y') = f\Big(\text{round}(x), \text{round}(y)\Big).$$

[2] This method is very fast but can create blocky artefacts.

b) Algorithm.

- Round (x, y) to the nearest integer coordinate (l, k).
- If (l, k) is inside the image, return $f(l, k)$; otherwise return background.

c) Pseudo-code.

```
Function interpolate_nearest(f, x, y):
    l = round(x)
    k = round(y)
    If (l, k) outside bounds of f:
        return 0
    Else:
        return f[l, k]
```

2.3.3. Bilinear interpolation

a) Theory. Bilinear interpolation combines the four nearest neighbors around (x, y). Let

$$l = \lfloor x \rfloor, \quad k = \lfloor y \rfloor, \quad a = x - l, \quad b = y - k.$$

Then

$$g(x', y') = (1-a)(1-b)\, f(l, k) + a(1-b)\, f(l+1, k) + (1-a)b\, f(l, k+1) + ab\, f(l+1, k+1).$$

[2] *b) Algorithm.*

- Determine the four surrounding points (l, k), $(l+1, k)$, $(l, k+1)$, $(l+1, k+1)$.
- Compute weights $(1-a)(1-b)$, $a(1-b)$, $(1-a)b$, ab.

- Multiply each weight with the brightness of the corresponding neighbor and sum.

c) Pseudo-code.

```
Function interpolate_bilinear(f, x, y):
    l = floor(x); a = x - l
    k = floor(y); b = y - k

    If any of (l, k), (l+1, k), (l, k+1), (l+1, k+1)
       is outside bounds:
        return 0

    v00 = f[l,   k  ]
    v10 = f[l+1, k  ]
    v01 = f[l,   k+1]
    v11 = f[l+1, k+1]

    return (1-a)*(1-b)*v00 + a*(1-b)*v10 \
         + (1-a)*b*v01     + a*b*v11
```

2.4. Specific Transformations: Scaling, Rotation, Translation, Shearing

Each geometric operation is obtained by choosing a particular affine matrix A_h and reusing the same inverse–mapping and interpolation pipeline.

2.4.1. Scaling

a) Theory. Scaling with factors s_x and s_y along the x- and y-axes is represented by

$$A_h = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

If $s_x > 1$ (resp. $s_x < 1$) the image is enlarged (resp. shrunk) horizontally.

b) Algorithm.

- Choose output size $W' = \lfloor s_x W \rfloor$, $H' = \lfloor s_y H \rfloor$.

- Construct matrix A_h as above.
- Call the inverse–mapping algorithm with bilinear interpolation.

c) Pseudo-code.

```
Input: image f, sx, sy
A_h = [[sx, 0,  0],
       [0,  sy, 0],
       [0,  0,  1]]

g = apply_affine(f, A_h, interp = "bilinear")
Return g
```

2.4.2. Rotation

a) Theory. Rotation by angle θ around the origin is described by

$$A_h = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

To rotate around the image center, translations to and from the center are composed with this matrix.

b) Algorithm.

- Compute the image center (c_x, c_y).
- Create the translation matrix T_1 to move the center to the origin, the rotation matrix R, and the matrix T_2 to move back to the original center.
- Set $A_h = T_2 R T_1$ and reuse the inverse-mapping algorithm.

c) Pseudo-code.

```
Input: image f, angle    (degrees)
 _rad  = deg2rad(  )
```

```
H, W  = size(f)
cx, cy = W/2, H/2

T1 = [[1, 0, -cx],
      [0, 1, -cy],
      [0, 0,  1]]

R  = [[cos , -sin , 0],
      [sin ,  cos , 0],
      [0,     0,    1]]

T2 = [[1, 0, cx],
      [0, 1, cy],
      [0, 0, 1]]

A_h = T2 * R * T1
g   = apply_affine(f, A_h, interp = "bilinear")
Return g
```

2.4.3. Translation

a) Theory.

Translation by (t_x, t_y) is given by

$$A_h = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}.$$

[1] *b) Algorithm.*

- Keep the original output size.
- Construct A_h from (t_x, t_y).

- Call the inverse-mapping algorithm with nearest-neighbor or bilinear interpolation.

c) Pseudo-code.

```
Input: image f, tx, ty
A_h = [[1, 0, tx],
       [0, 1, ty],
       [0, 0, 1]]

g = apply_affine(f, A_h, interp = "nearest")
Return g
```

2.4.4. Shearing

a) Theory.

Shearing with factors k_x and k_y is represented by

$$A_h = \begin{bmatrix} 1 & k_x & 0 \\ k_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

[1] *b) Algorithm.*

- Choose the shear coefficients (k_x, k_y) based on the desired slant.
- Construct the matrix A_h as above.
- Apply inverse mapping with bilinear interpolation to reduce aliasing.

c) Pseudo-code.

```
Input: image f, kx, ky
A_h = [[1,  kx, 0],
       [ky, 1,  0],
       [0,  0,  1]]

```

```
g = apply_affine(f, A_h, interp = "bilinear")
Return g
```

Chapter III. Installation and Usage

This section describes how to set up the environment, install required libraries, and execute the program in all three supported run modes. All code examples follow the structure of the LAB 01 Basic Image Processing Pipeline.

1. Environment Setup

1.1. Python Version and Required Libraries

The project requires:

- Python 3.8 or newer;
- `numpy` for linear algebra and numerical computation;
- `opencv-python` for image manipulation;

1.1.1. Creating a Virtual Environment

```
python -m venv .venv
source .venv/bin/activate      # Windows: .venv\Scripts\
   activate
pip install numpy opencv-python
```

2. How to Run the Program

The main script exposes three run modes via `-mode`:

```
python main.py --mode {test,batch,interactive}
               [--image PATH]
               [--out_dir DIR]
```

2.1. Mode 1: test

2.1.1. Purpose

This mode performs a full functional test of:

- all color transformations (linear, non-linear, histogram-based);
- geometric transformations (scale, rotate, translate, shear);
- interpolation demos and backward mapping;

2.1.2. Usage

```
python main.py
python main.py --mode test
python main.py --mode test --out_dir outputs_test
```

It is suitable for automated grading or quick validation.

2.2. Mode 2: batch

2.2.1. Purpose

Runs the full pipeline on a user-defined input image.

If `-image` is omitted, the program prompts for a path.

2.2.2. Usage

```
python main.py --mode batch --image data/camera.png
python main.py --mode batch
```

The program will:

1. load and store the original image,
2. apply color transformations,
3. run geometric experiments,
4. save all results into the specified output folder.

3. Project Structure

3.1. Project Directory Structure

```
main.py                      # Entry point (3 run modes)
color_transform.py           # Color transformation algorithms
geometric_transforms.py      # Geometric transformation algorithms
assets/
   Lenna.jpg                # Auto-downloaded if missing
outputs/
    color/
    geometric_transforms_demo/
    compare/
```

3.2. Features

No.	Function	Description
1	`brightness_adjust(img, b)`	Adds an offset b to all BGR channels while preserving their relative proportions.
2	`contrast_adjust(img, a)`	Scales intensities by factor a to expand or compress contrast.
3	`brightness_contrast_adjust(img, a, b)`	Applies contrast scaling a and brightness shift b; used in interactive mode.
4	`range_linear_mapping(gray, f1, f2, g1, g2)`	Maps grayscale interval $[f_1, f_2]$ to $[g_1, g_2]$; clamps values outside the range.
5	`log_mapping(gray, c)`	Logarithmic transform $c \cdot \log(1 + I)$ to enhance dark image regions.
6	`exp_mapping(gray, c)`	Exponential transform normalized to $[0, 255]$ to emphasize bright regions.
7	`histogram_equalization(gray)`	Computes grayscale CDF and remaps intensity values to approximate a uniform histogram.
8	`histogram_specification(src, ref)`	Adjusts the histogram of `src` to match the histogram of reference image `ref`.

Table 2: Core functions implemented for color transformations

No.	Function	Description
1	`build_scale_matrix(image, sx, sy)`	Constructs a 2×3 affine matrix and output size for scaling without cropping.
2	`build_rotation_matrix(image, angle_deg)`	Creates a rotation matrix about the image center; enlarges output to avoid corner loss.
3	`build_translation_matrix(image, tx, ty)`	Produces a translation matrix applying offsets (tx, ty).
4	`build_shear_matrix(image, kx, ky)`	Builds a shear (skew) matrix and computes expanded output size.
5	`warp_affine_nearest(image, A, out_size)`	Backward mapping using nearest-neighbor interpolation.
6	`warp_affine_bilinear(image, A, out_size)`	Backward mapping using bilinear interpolation to reduce aliasing.
7	`run_geometric_experiments(image, out_dir)`	Runs demos: affine transforms, backward mapping, scaling, rotation, translation, shear.

Table 3: Core functions implemented for geometric transformations

Chapter IV. Result and Comparison

1. Results

1.1. Color Transformation Results

This section presents the results of various color transformation algorithms applied to the input image.

1.1.1. Linear Mappings

(a) Brightness adjustment (+40)

(b) Contrast adjustment (×1.4)

Figure 1: Linear mapping transformations: brightness and contrast adjustments

Figure 2: Combined brightness and contrast adjustment

Figure 3: Range mapping: mapping intensity range [40, 200] to [0, 255]

1.1.2. Non-linear Mappings

(a) Logarithmic mapping (b) Exponential mapping

Figure 4: Non-linear intensity transformations using logarithmic and exponential functions

1.1.3. Histogram-based Transformations

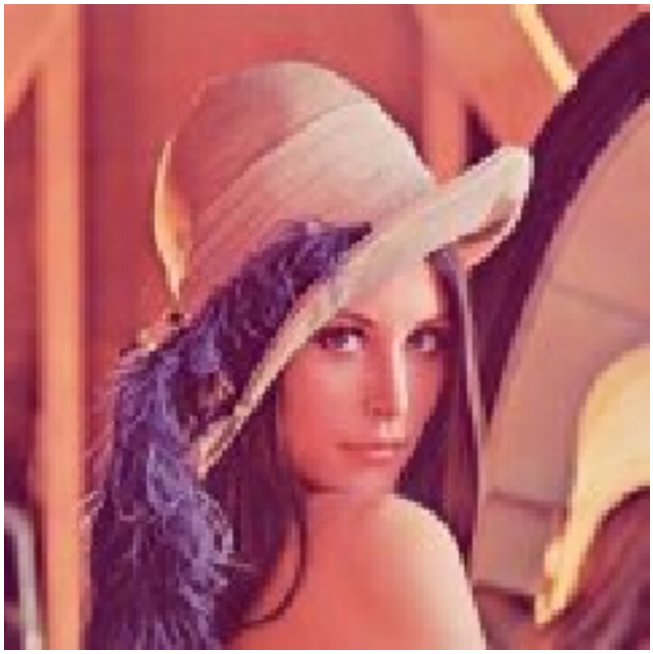
(a) Histogram equalization (b) Histogram specification

Figure 5: Histogram-based transformations for contrast enhancement and intensity redistribution

1.2. Geometric Transformation Results

This section demonstrates various geometric transformation techniques and their visual effects.

1.2.1. Interpolation Techniques



(a) Nearest-neighbor interpolation

(b) Bilinear interpolation

Figure 6: Comparison of interpolation techniques. Nearest-neighbor interpolation (6a) exhibits pixelated edges, while bilinear interpolation (6b) produces smoother transitions.

1.2.2. Affine Transform

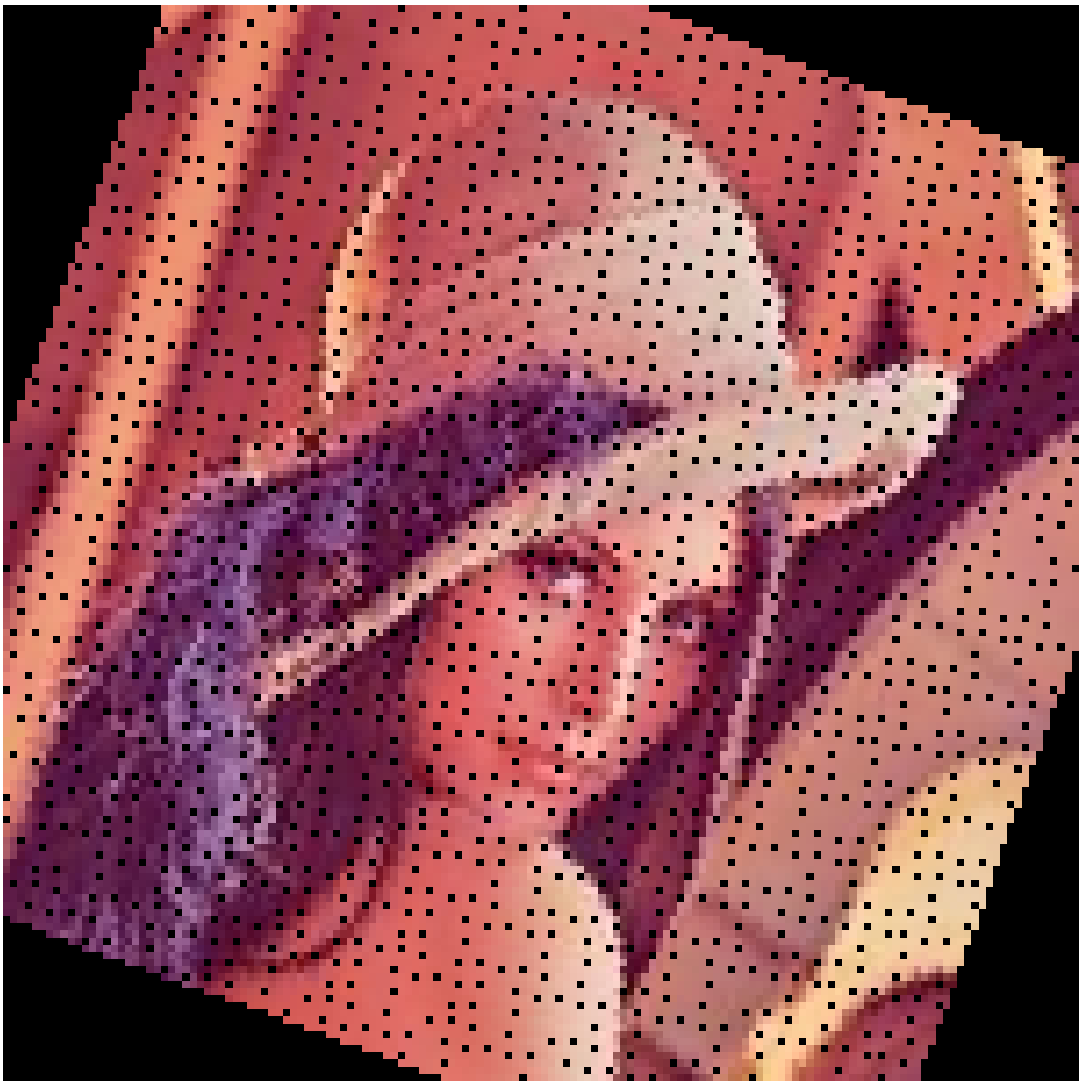


Figure 7: Forward mapping using affine transformation. This approach transforms source coordinates to destination coordinates directly.

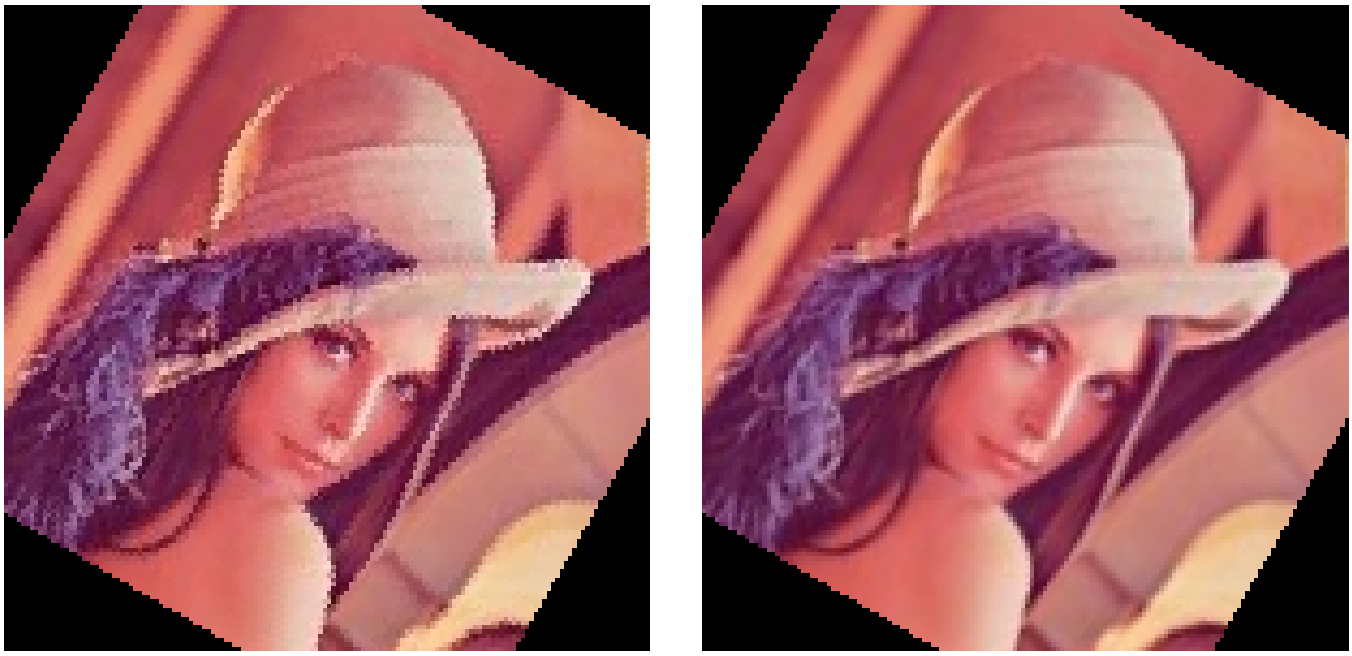
1.2.3. Backward Mapping

(a) Backward mapping with nearest-neighbor interpolation

(b) Backward mapping with bilinear interpolation

Figure 8: Backward mapping results using different interpolation methods. This technique ensures complete pixel coverage by mapping from destination to source coordinates.

1.2.4. Combined Transformations



(a) Rotation with nearest-neighbor (b) Rotation with bilinear interpolation

Figure 9: Image rotation transformation results with different interpolation methods

Rotation

(a) Scaling with nearest-neighbor (b) Scaling with bilinear interpolation

Figure 10: Image scaling (resizing) transformation results with different interpolation methods

Scaling

(a) Shear with nearest-neighbor (b) Shear with bilinear interpolation

Figure 11: Image shear transformation results with different interpolation methods

Shear

2. Algorithm Comparison

2.1. Color Transformation Comparison

2.1.1. Image Quality Comparison

Figure 12: Histogram equalization comparison between custom implementation and OpenCV.

Both methods produce nearly identical results, with minimal differences in pixel intensity distribution. This confirms the correctness of the custom implementation.

2.1.2. Performance

Table 4: Color Transform (log/exp/specification)

Op	Impl	Time (ms)	Peak mem (KB)
brightness	manual	0.111	287.2
brightness	opencv	0.013	22.1
contrast	manual	0.067	287.2
contrast	opencv	0.009	22.1
bright+contrast	manual	0.068	287.2
bright+contrast	opencv	0.009	22.1
range_map	manual	0.249	398.2
range_map	opencv+LUT	0.018	22.1
hist_equalize	manual	0.153	202.7
hist_equalize	opencv	0.022	22.1

3. Geometric Transformation Comparison

3.1. Image Quality Comparison

3.1.1. Interpolation Methods Comparison

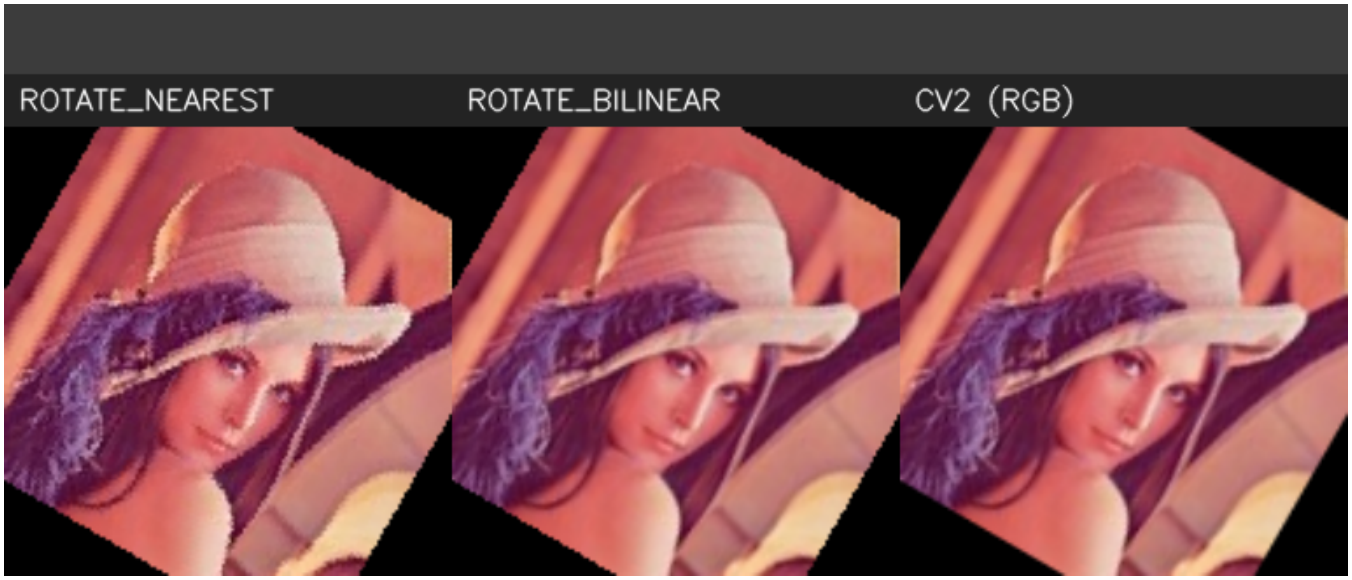


Figure 13: Rotation results using different interpolation methods.

Nearest-neighbor produces noticeable aliasing, bilinear interpolation gives smoother edges, and OpenCV achieves the highest visual quality.

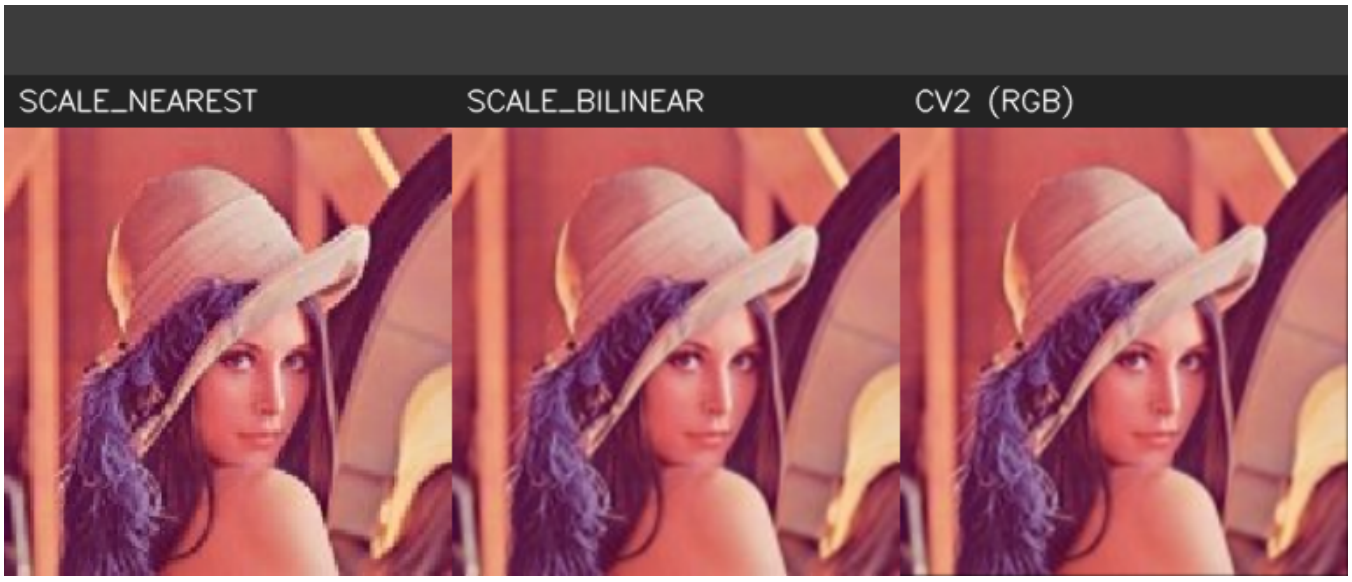


Figure 14: Scaling results using different interpolation methods.

Nearest-neighbor introduces pixelation artifacts, bilinear improves smoothness, and OpenCV provides the best quality overall.

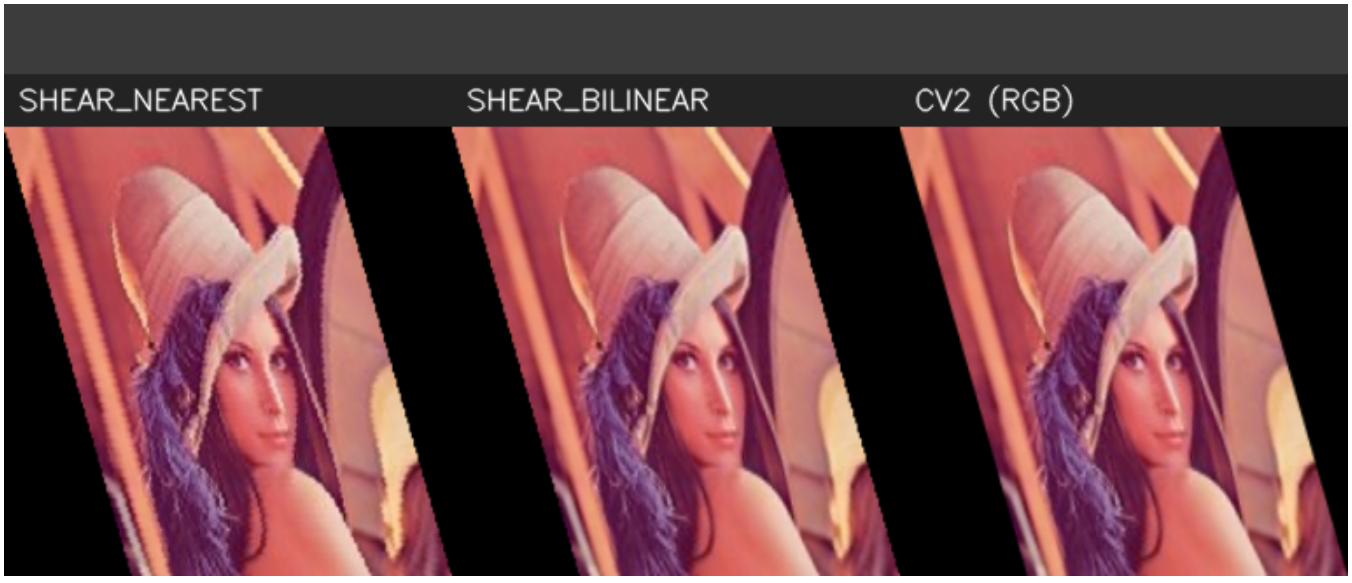


Figure 15: Shear transformation results with different interpolation methods.

Nearest-neighbor shows stair-step artifacts, bilinear interpolation yields smoother gradients, and OpenCV produces the most refined output.

3.1.2. Forward vs Backward Mapping Comparison

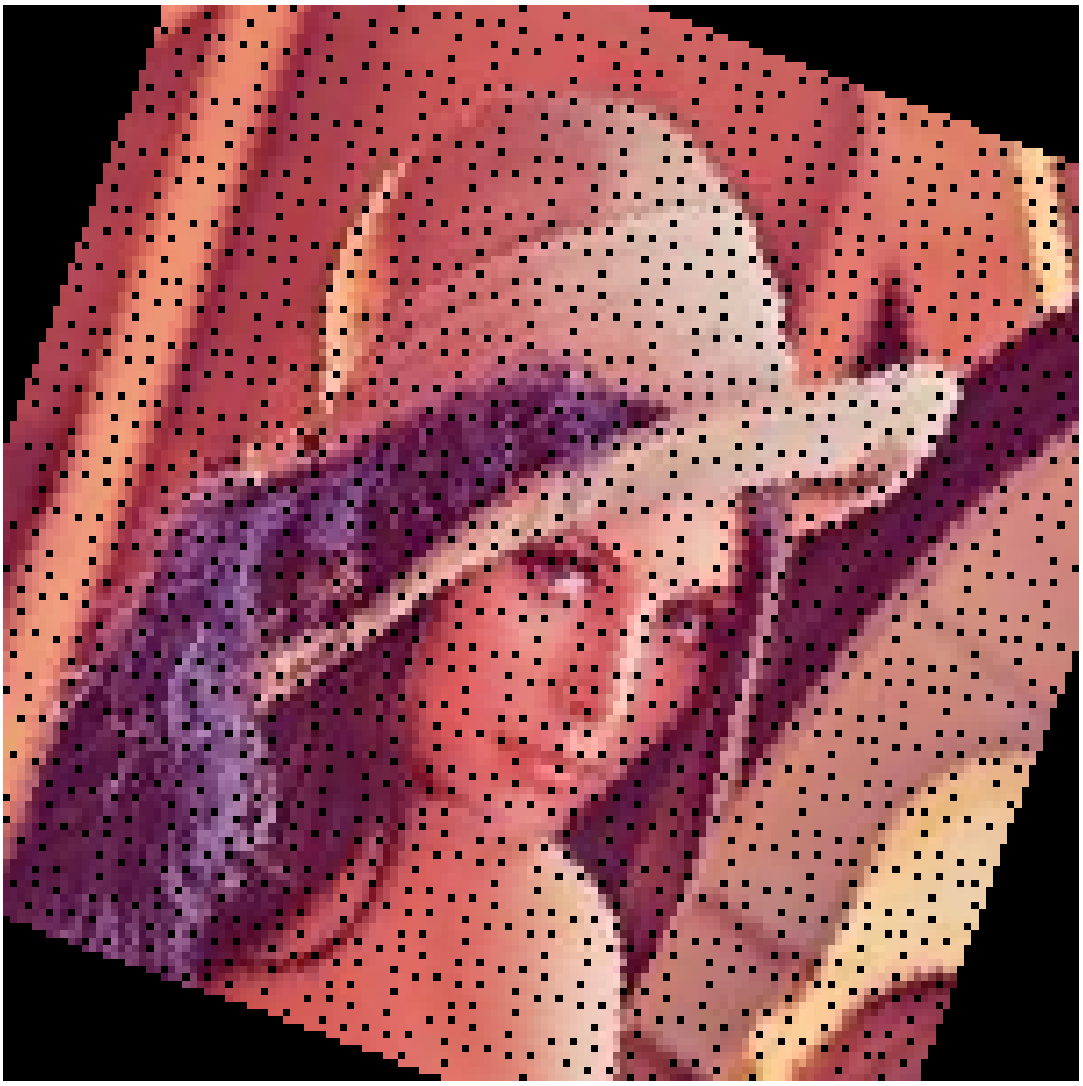


Figure 16: Forward mapping using affine transform.

Forward mapping may produce holes or overlaps in the output image because source pixels do not map uniformly to destination coordinates.

(a) Backward mapping (nearest-neighbor)

(b) Backward mapping (bilinear)

Figure 17: Backward mapping results with different interpolation methods.

Backward mapping with inverse transform ensures that every output pixel is assigned a source value, eliminating holes. Nearest-neighbor produces jagged edges, while bilinear interpolation gives smoother and more visually consistent results.

3.1.3. Performance

Table 5: Geometric Transforms (interp + affine/backward/scale/rotate/shear)

Op	Impl	Time (ms)	Peak mem (KB)
scale_1.6	manual_nn	1821.697	1520.7
scale_1.6	manual_bilinear	10989.450	1520.8
scale_1.6	opencv_nn	0.152	168.9
scale_1.6	opencv_linear	0.242	168.9
rotate_30	manual_nn	980.091	594.3
rotate_30	manual_bilinear	2671.371	594.4
rotate_30	opencv_nn	0.088	66.0
rotate_30	opencv_linear	0.196	66.0
shear_kx0.4	manual_nn	955.072	831.6
shear_kx0.4	manual_bilinear	3260.896	831.7
shear_kx0.4	opencv_nn	0.111	92.4
shear_kx0.4	opencv_linear	0.275	92.4

3.1.4. Discussion

The comparison reveals several key observations:

- **Color Transformations:** Custom implementation and OpenCV produce nearly identical histogram equalization outputs, confirming correct algorithm behavior.
- **Interpolation Quality:** Nearest-neighbor is fast but causes aliasing; bilinear improves smoothness; and OpenCV (often using bicubic or optimized bilinear) yields the best quality.
- **Forward vs Backward Mapping:** Forward mapping may leave holes or overlaps due to uneven pixel mapping. Backward mapping avoids this by computing the correct source position for each output pixel.
- **Visual Quality:** Backward mapping combined with bilinear interpolation provides a strong balance between computational efficiency and output smoothness.

Chapter V. Conclusion

1. Summary of Achievements

The overall completion of the lab is shown in Table:

Section	Completion	Notes
Color Transformations	100%	Linear, log, exp, histogram equalization, specification implemented
Geometric Transformations	100%	Affine, interpolation, scaling, rotation, shear implemented
Total Lab	**100%**	**All required tasks completed and tested**

Table 6: Overall completion of Lab 01.

Overall, I have completed all required tasks related to color and geometric transformations in this lab, from theoretical formulation, to algorithm design, to practical implementation and evaluation.

2. Future Improvements

Although the current implementations are functional, my approach to both color and geometric transformations remains quite basic. In the future, I would like to improve several aspects

of the system. For color transformations, I can explore more advanced tone-mapping and local contrast enhancement methods, extend the framework to support different color spaces (such as HSV or Lab), and make the parameter selection more adaptive to the input image content instead of using fixed global parameters.

For geometric transformations, there is still much room for improvement in terms of robustness and efficiency. Possible directions include implementing higher-order interpolation methods, improving the handling of image boundaries and invalid mappings, and generalizing the framework to support arbitrary sequences of transformations in a more modular way. In addition, I could perform a more systematic quantitative evaluation on a larger set of test images, and investigate the use of vectorization or GPU acceleration to further reduce computation time.

Despite these limitations, the project has provided me with a solid foundational understanding of how color and geometric transformations operate at the pixel level, and has given me a working codebase that I can continue to refine and extend in future courses or projects.

References

[1] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Pearson, 4 edition, 2018.

[2] Ly Quoc Ngoc. Lecture 4: Geometric transformations. Lecture Slides, Digital Image and Video Processing, 2024. Accessed from course materials.

[3] Ly Quoc Ngoc. Lecture 3: Color transformations. Lecture Slides, Digital Image and Video Processing, 2025. Accessed from course materials.