*Heaven's Light is Our Guide*

**Rajshahi University of Engineering & Technology**

**Department of**

**Electrical & Computer Engineering**

# Lab Report 4

Study of Image Zooming and Interpolation Techniques

**Course Code:** ECE 4224

**Course Title:** Digital Image Processing Sessional


**Submitted to:**
Oishi Jyoti
Assistant Professor
Dept of ECE, RUET

**Submitted by:**
Md. Tajim An Noor
Roll: 2010025
Semester: 4th Year Even


**Submission Date:** January 27, 2026

# Contents

# Study of Image Zooming and Interpolation Techniques

## 1 Theory and Introduction

Image zooming (scaling) involves resizing a digital image matrix. Since this operation creates new spatial locations where no pixel data previously existed, **interpolation** algorithms are required to estimate the intensity values of these new pixels based on the surrounding known data [1].

This experiment compares two primary interpolation methods:

- **Nearest Neighbor Interpolation:** Assigns the value of the spatially closest original pixel to the new pixel. It preserves original data values exactly but introduces aliasing (blockiness).
- **Bilinear Interpolation:** Computes a weighted average of the four nearest neighbors using linear distance. It produces smooth gradients but acts as a low-pass filter, potentially blurring sharp edges.

## 2 Methodology

A $4 \times 4$ matrix was defined with specific intensity values to represent a high-contrast pattern. Zooming algorithms were then manually implemented with a scaling factor of $s = 4$.

### 2.1 Matrix Definition

The input matrix utilizes three distinct intensity levels: 10 (Dark), 50 (Mid-tone), and 200 (Bright).

$$I = \begin{bmatrix} 10 & 200 & 10 & 200 \\ 200 & 50 & 200 & 50 \\ 10 & 200 & 10 & 200 \\ 200 & 50 & 200 & 50 \end{bmatrix}$$

With $s = 4$, the target output dimension is $(4 \times 4) \rightarrow (16 \times 16)$.

### 2.2 Python Implementation

The implementation avoids built-in resizing functions to demonstrate the underlying logic:

1. **Nearest Neighbor:** The original matrix is iterated through, and each pixel value is replicated into a $4 \times 4$ block in the destination matrix.
2. **Bilinear:** The destination matrix is iterated through, coordinates are mapped back to the source, and the weighted sum of neighbors is calculated using the bilinear formula:

$$f(x, y) = (1 - \alpha)(1 - \beta)Q_{11} + \alpha(1 - \beta)Q_{21} + (1 - \alpha)\beta Q_{12} + \alpha\beta Q_{22}$$

```python
import numpy as np
import matplotlib.pyplot as plt
import math
import os

os.makedirs("./images/output", exist_ok=True)
# 1. Define a 4x4 Matrix "Image" Using a checkerboard-like pattern with distinct values for clarity
# 10 = Dark, 200 = Bright
img_4x4 = np.array(
    [[10, 200, 10, 200], [200, 50, 200, 50], [10, 200, 10, 200], [200, 50, 200, 50]],
    dtype=np.uint8,
)

m, n = img_4x4.shape
s = 3  # Scaling Factor

# Method 1: Nearest Neighbor (Manual Loop)
new_m = m * s
new_n = n * s
zoomed_nn = np.zeros((new_m, new_n), dtype=np.uint8)

# Loop through original 4x4 pixels
for i in range(m):
    for j in range(n):
        val = img_4x4[i, j]

        # Fill the 3x3 block in the new image
        # Row start: i*3, Col start: j*3
        for r in range(s):
            for c in range(s):
                zoomed_nn[i * s + r, j * s + c] = val

# Method 2: Bilinear Interpolation (Manual Loop)
zoomed_bl = np.zeros((new_m, new_n), dtype=np.uint8)


# Helper function to get pixel safely (clamping to edges)
def get_pixel(img, x, y):
    h, w = img.shape
    x = min(max(x, 0), h - 1)
    y = min(max(y, 0), w - 1)
    return img[x, y]


# Loop through NEW 8x8 pixels
for i in range(new_m):
    for j in range(new_n):

        # Map back to original coordinate space (i / s) scales 0..7 back to 0..3.5 range
        orig_x = i / s
        orig_y = j / s

        # Find 4 Nearest Neighbors
        x1 = int(math.floor(orig_x))
        y1 = int(math.floor(orig_y))
        x2 = x1 + 1
        y2 = y1 + 1

        # Calculate distances (0 to 1)
        alpha = orig_x - x1
        beta = orig_y - y1

        # Get values of neighbors
        Q11 = get_pixel(img_4x4, x1, y1)  # Top-Left
        Q21 = get_pixel(img_4x4, x2, y1)  # Bottom-Left
        Q12 = get_pixel(img_4x4, x1, y2)  # Top-Right
        Q22 = get_pixel(img_4x4, x2, y2)  # Bottom-Right

        # Bilinear Formula
        val = (
            (1 - alpha) * (1 - beta) * Q11
            + alpha * (1 - beta) * Q21
            + (1 - alpha) * beta * Q12
            + alpha * beta * Q22
        )

        zoomed_bl[i, j] = int(val)


# Visualization
def plot_matrix(ax, matrix, title):
    ax.imshow(matrix, cmap="viridis", vmin=0, vmax=255)
    ax.set_title(title)
    # Draw grid lines to show pixels clearly
    h, w = matrix.shape
    ax.set_xticks(np.arange(-0.5, w, 1), minor=True)
    ax.set_yticks(np.arange(-0.5, h, 1), minor=True)
    ax.grid(which="minor", color="white", linestyle="-", linewidth=1)
    ax.tick_params(which="minor", size=0)
    # Annotate values
    for i in range(h):
        for j in range(w):
            ax.text(
                j,
                i,
                str(matrix[i, j]),
                ha="center",
                va="center",
                color="white" if matrix[i, j] < 150 else "black",
                fontsize=8,
```

```python
            )


fig, axes = plt.subplots(1, 3, figsize=(15, 5))

plot_matrix(axes[0], img_4x4, "Original (4x4)")
plot_matrix(axes[1], zoomed_nn, "Nearest Neighbor (8x8)")
plot_matrix(axes[2], zoomed_bl, "Bilinear (8x8)")

plt.tight_layout()
plt.savefig("./images/output/lab4_color.png")
# plt.show()

# Print Matrices to Console for Report Verification
print("--- Original 4x4 ---")
print(img_4x4)
print("\n--- Nearest Neighbor 8x8 (Note the 2x2 blocks) ---")
print(zoomed_nn)
print("\n--- Bilinear 8x8 (Note the smoothing) ---")
print(zoomed_bl)
```

# 3 Results

The code generated 16 × 16 matrices visualized using the `viridis` colormap.

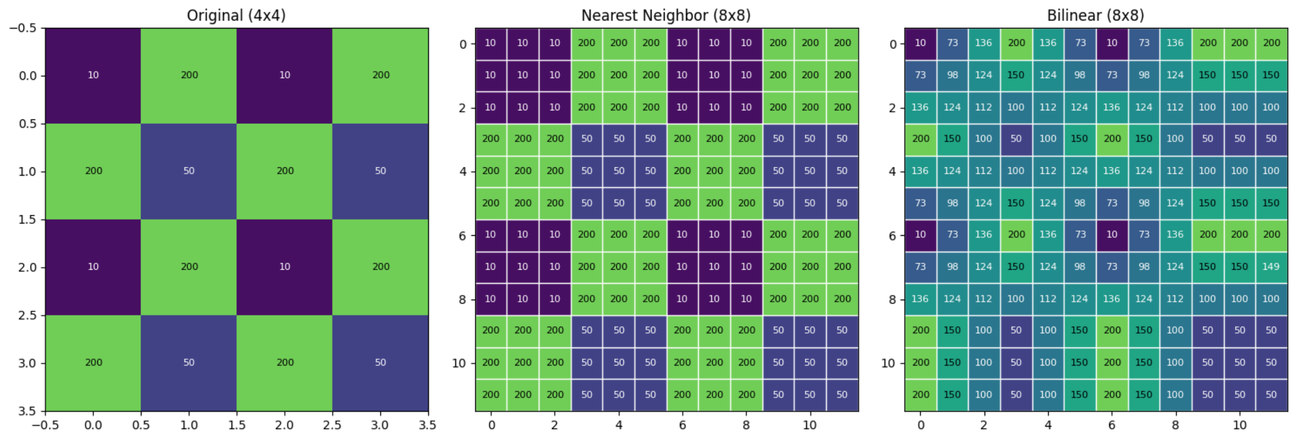


Figure 1: Zooming Analysis ($s = 4$). Left: Original 4x4. Center: Nearest Neighbor simply expands pixels into large blocks. Right: Bilinear Interpolation generates intermediate colors (gradients) between the values 10, 50, and 200.

# 4 Discussion

The comparison highlights the trade-off between sharpness and smoothness:

- **Nearest Neighbor:** The result is a direct magnification of the original grid. The output contains only the original values (10, 50, 200). It is visually jagged but preserves the exact contrast of the original data.
- **Bilinear Interpolation:** The result introduces new intensity values not present in the original image. For example, between a pixel of value 200 and a pixel of value 50, the algorithm generates a smooth transition (e.g., 163, 126, 88). This creates a visually softer image, effectively smoothing out the "checkerboard" pattern.

# 5 Conclusion

The mathematical foundations of image resampling were validated in this experiment. By implementing the algorithms manually, it was observed that Nearest Neighbor interpolation is computationally efficient and preserves discrete data types, while Bilinear interpolation provides superior visual quality for continuous-tone images by estimating intermediate intensities.

## References

[1] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed. Boston, MA, USA: Pearson, 2018.