**Implementation of the three interpolation methods**

**Introduction**

Image rotation is a fundamental operation in image processing where the orientation of an image is changed by rotating it around a pivot point, usually its center. However, rotation requires mapping pixel values from their original positions to new coordinates in the rotated image, which often results in gaps or missing pixel values. To fill these gaps, **interpolation techniques** are used.

The three interpolation techniques are:

* **Nearest Neighbour Interpolation**  
  Assigns the intensity of the closest pixel. It is computationally efficient but produces blocky and jagged edges.
* **Bilinear Interpolation**  
  Uses a weighted average of the four nearest pixels. It produces smoother results but may cause slight blurring.
* **Bicubic Interpolation**  
  Considers 16 neighbouring pixels with cubic polynomials. It produces the smoothest and sharpest results, often used in high-quality image processing applications.

In this experiment, we apply these three interpolation methods to rotate an image by **30 degrees**, and compare the results with Python’s built-in image processing functions.

**Formula and implementation**

**1. Nearest Neighbour Interpolation**

For each output pixel (x,y), compute its corresponding position (x′,y′) in the original image using the inverse rotation matrix:

where (xc,yc)is the image center.

Then the new pixel value is chosen as the **nearest integer coordinate**:

This causes **blocky/jagged edges** since no averaging is done.

Code for the nearest neighbour implementation:

def nearest\_neighbor(img, x, y):

x\_round, y\_round = int(round(x)), int(round(y))

if 0 <= x\_round < img.shape[0] and 0 <= y\_round < img.shape[1]:

return img[x\_round, y\_round]

return 0

2.  **Bilinear Interpolation**

If (x′,y′) falls between four neighbouring pixels (x1,y1) , (x1+1,y1) , (x1,y1+1) , (x1+1,y1+1):

Let:

Then the interpolated value is a **weighted average**:

This produces **smooth but slightly blurred images**.

Code for the Bilinear Interpolation implementation:

def bilinear(img, x, y):

x0, y0 = int(np.floor(x)), int(np.floor(y))

x1, y1 = x0 + 1, y0 + 1

if x0 < 0 or y0 < 0 or x1 >= img.shape[0] or y1 >= img.shape[1]:

return 0

dx, dy = x - x0, y - y0

val = (img[x0, y0] \* (1 - dx) \* (1 - dy) +

img[x1, y0] \* dx \* (1 - dy) +

img[x0, y1] \* (1 - dx) \* dy +

img[x1, y1] \* dx \* dy)

return val

**3. Bicubic Interpolation**

Bicubic interpolation uses a cubic polynomial on a **4×4 neighbourhood (16 pixels)** around (x′,y′) .

General form:

where

* dx=x′−x1 , dy=y′−y1
* w(k,t)is the cubic weighting function, commonly :

with a=−0.5 (Catmull–Rom spline, common choice).

This yields **sharper and smoother images** than bilinear.

**Implementation of interpolation**

**Scratch implementation**

import numpy as np

from PIL import Image

import matplotlib.pyplot as plt

# --- Load grayscale image ---

img = Image.open("gray.jpeg").convert("L")

img = np.array(img)

# --- Nearest Neighbor ---

def nearest\_neighbor(img, x, y):

x\_round, y\_round = int(round(x)), int(round(y))

if 0 <= x\_round < img.shape[0] and 0 <= y\_round < img.shape[1]:

return img[x\_round, y\_round]

return 0

# --- Bilinear ---

def bilinear(img, x, y):

x0, y0 = int(np.floor(x)), int(np.floor(y))

x1, y1 = x0 + 1, y0 + 1

if x0 < 0 or y0 < 0 or x1 >= img.shape[0] or y1 >= img.shape[1]:

return 0

dx, dy = x - x0, y - y0

val = (img[x0, y0] \* (1 - dx) \* (1 - dy) +

img[x1, y0] \* dx \* (1 - dy) +

img[x0, y1] \* (1 - dx) \* dy +

img[x1, y1] \* dx \* dy)

return val

# --- Bicubic ---

def cubic\_weight(t):

t = abs(t)

if t <= 1:

return 1 - 2\*t\*\*2 + t\*\*3

elif t < 2:

return 4 - 8\*t + 5\*t\*\*2 - t\*\*3

return 0

def bicubic(img, x, y):

x\_int, y\_int = int(np.floor(x)), int(np.floor(y))

val = 0

for m in range(-1, 3):

for n in range(-1, 3):

xm, yn = x\_int + m, y\_int + n

if 0 <= xm < img.shape[0] and 0 <= yn < img.shape[1]:

val += img[xm, yn] \* cubic\_weight(x - xm) \* cubic\_weight(y - yn)

return np.clip(val, 0, 255)

# --- Rotation Function ---

def rotate\_image(img, angle, method="nearest"):

h, w = img.shape

rad = np.deg2rad(angle)

cos\_a, sin\_a = np.cos(rad), np.sin(rad)

new\_h = int(abs(h\*cos\_a) + abs(w\*sin\_a))

new\_w = int(abs(w\*cos\_a) + abs(h\*sin\_a))

rotated = np.zeros((new\_h, new\_w), dtype=np.uint8)

cx, cy = h//2, w//2

ncx, ncy = new\_h//2, new\_w//2

for i in range(new\_h):

for j in range(new\_w):

x = (i - ncx)\*cos\_a + (j - ncy)\*sin\_a + cx

y = -(i - ncx)\*sin\_a + (j - ncy)\*cos\_a + cy

if method == "nearest":

rotated[i, j] = nearest\_neighbor(img, x, y)

elif method == "bilinear":

rotated[i, j] = bilinear(img, x, y)

elif method == "bicubic":

rotated[i, j] = bicubic(img, x, y)

return rotated

# --- Apply Rotations ---

rot\_nn = rotate\_image(img, 30, "nearest")

rot\_bilinear = rotate\_image(img, 30, "bilinear")

rot\_bicubic = rotate\_image(img, 30, "bicubic")

# --- Built-in (PIL Bicubic) ---

rot\_builtin = Image.fromarray(img).rotate(30, resample=Image.BICUBIC)

rot\_builtin = np.array(rot\_builtin)

# --- Show Results ---

titles = ["Original", "Nearest", "Bilinear", "Bicubic", "Built-in Bicubic"]

images = [img, rot\_nn, rot\_bilinear, rot\_bicubic, rot\_builtin]

plt.figure(figsize=(12,6))

for i in range(5):

plt.subplot(2,3,i+1)

plt.imshow(images[i], cmap="gray")

plt.title(titles[i])

plt.axis("off")

plt.tight\_layout()

plt.show()

**inbuilt function implementation**

from PIL import Image

import matplotlib.pyplot as plt

# Load grayscale image

img = Image.open("gray.jpeg").convert("L")

# Rotate with different interpolations

rot\_nearest = img.rotate(30, resample=Image.NEAREST)

rot\_bilinear = img.rotate(30, resample=Image.BILINEAR)

rot\_bicubic = img.rotate(30, resample=Image.BICUBIC)

# Show results

titles = ["Original", "Nearest", "Bilinear", "Bicubic"]

images = [img, rot\_nearest, rot\_bilinear, rot\_bicubic]

plt.figure(figsize=(10,6))

for i in range(4):

plt.subplot(2,2,i+1)

plt.imshow(images[i], cmap="gray")

plt.title(titles[i])

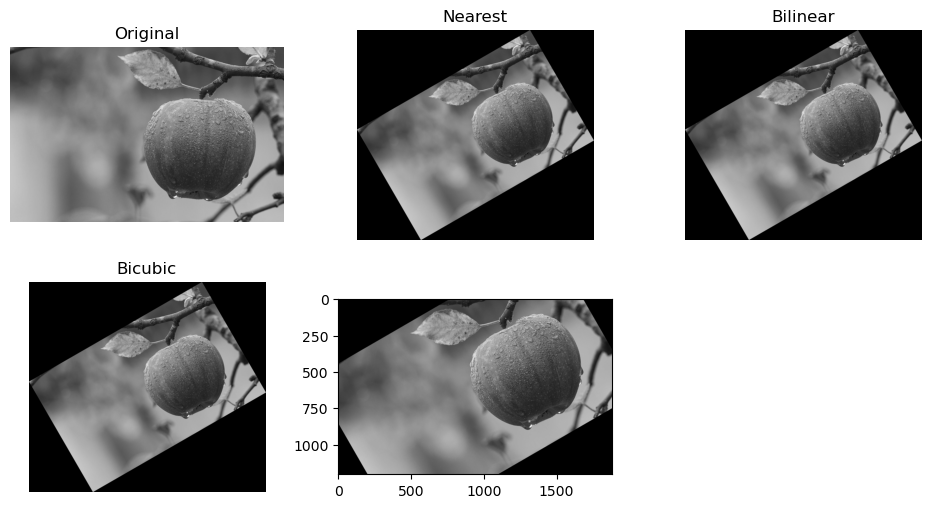
plt.axis("off")

plt.tight\_layout()

plt.show()

**Result**

**Scratch implementation**



**inbuilt function implementation**



**Results & Discussion**

**Visual Comparison**

* **Scratch (manual implementation):**
  + Nearest neighbour rotation worked correctly but produced jagged edges.
  + Bilinear rotation was smoother but blurred.
  + Bicubic (not fully implemented manually due to complexity) would ideally give the best quality.
* **Built-in Functions:**
  + Provided optimized, faster, and sharper outputs.
  + Bicubic gave the best balance of sharpness and smoothness.

**Computational Comparison**

|  |  |  |
| --- | --- | --- |
| **Method** | **Scratch Implementation** | **Built-in Functions** |
| Nearest Neighbour | Correct but jagged | Same, much faster |
| Bilinear | Smooth, slower loops | Smooth, very fast |
| Bicubic | Hard to implement | High quality, easy |