Image Enhancement in Digital Image Processing

Assistant Professor Vivek Singh Sikarwar
Manipal University Jaipur

Grayscale Transformation

- Converts a color image to grayscale.

- Common techniques:
  - - Averaging method
  - - Luminosity method
  - - Weighted sum method
  - • Used in pre-processing for edge detection and segmentation.

Brightness Interpolation

- Adjusts the brightness of an image by adding/subtracting intensity values.

- Formula: I_out = I_in + β

- Used to enhance visibility in dark images.

2. Mathematical Formulation

The brightness of an image can be adjusted using th

transformations:

1. **Addition/Subtraction Method**

$$I_{out}(x, y) = I_{in}(x, y)$$

- I_{out} = Output image
- I_{in} = Input image
- β = Brightness adjustment factor

2. **Linear Interpolation Method**

$$I_{out}(x,y) = \alpha \cdot I_1(x,y) + (1-\alpha) \cdot I_2(x,$$

- I_1, I_2 = Two different brightness levels
- α = Interpolation coefficient (0 ≤ α ≤ 1)

3. **Gamma Correction (Non-Linear Transformation)**

$$I_{out}(x,y) = c \cdot I_{in}(x,y)^{\gamma}$$

- c = Normalization constant
- γ < 1 enhances dark regions; γ > 1 brightens the image

Linear Interpolation

Linear interpolation is a method to estimate an unknown value within two known values. The formula is:

$$f(x) = f(x_1) + \frac{(x - x_1)}{(x_2 - x_1)} \times (f(x_2) - f(x_1))$$

Where:

- $f(x)$ is the interpolated value,
- $(x_1, f(x_1))$ and $(x_2, f(x_2))$ are known points.

In image processing, this is extended to **bilinear interpolation**, where interpolation is performed in both **x** and **y** directions.

Bilinear Interpolation
(Extension of Linear Interpolation)

When applied to images, linear interpolation is extended to **two dimensions (bilinear interpolation)**:

1. **Interpolate in one direction** (e.g., x-axis).
2. **Interpolate the result in the other direction** (e.g., y-axis).

The bilinear interpolation formula:

$$f(x,y) = f(Q_{11}) \cdot (1-dx) \cdot (1-dy) + f(Q_{21}) \cdot dx \cdot (1-dy) + f(Q_{12}) \cdot (1-dx) \cdot dy + f(Q_{22}) \cdot dx \cdot dy$$

Where:

- $Q_{11}, Q_{12}, Q_{21}, Q_{22}$ are the pixel values at four surrounding points.
- dx and dy are distances from the known points.

Applications of Linear Interpolation in Image Enhancement

- **Image Scaling**: Enlarging or reducing an image while maintaining visual smoothness.
- **Rotation & Transformation**: Interpolating missing pixel values.
- **Contrast Enhancement**: Modifying pixel values smoothly.
- **Edge Detection Preprocessing**: Smoothing images before applying edge detection filters.

Image Scaling using Linear Interpolation

- Given a **4×4** grayscale image, scale it up to **8×8** using linear interpolation.

Each value represents grayscale intensity (0-255):

$$\begin{bmatrix} 100 & 120 & 140 & 160 \\ 110 & 130 & 150 & 170 \\ 120 & 140 & 160 & 180 \\ 130 & 150 & 170 & 190 \end{bmatrix}$$

Step 1: Calculate Scale Factor

We need to scale the image from **4×4** to **8×8**, meaning:

$$\text{Scale Factor} = \frac{\text{New Size}}{\text{Old Size}} = \frac{8}{4} = 2$$

Each pixel in the **new image** corresponds to an interpolated value based on its position.

Apply Linear Interpolation Formula

Linear interpolation formula for 1D scaling:

$$f(x) = f(x_1) + \frac{(x - x_1)}{(x_2 - x_1)} \times (f(x_2) - f(x_1))$$

For example, to find the interpolated pixel at position **(0.5, 0)**:

- $x_1 = 0$, $x_2 = 1$
- $f(x_1) = 100$, $f(x_2) = 110$
- $x = 0.5$

$$f(0.5) = 100 + \frac{(0.5 - 0)}{(1 - 0)} \times (110 - 100) = 100 + 0.5 \times 10 = 105$$

Apply Bilinear Interpolation for 2D Scaling

For a new pixel at position $(0.5, 0.5)$, we interpolate in both directions.

Using four neighboring points:

$$Q_{11} = 100, Q_{12} = 120, Q_{21} = 110, Q_{22} = 130$$

$$f(x, y) = Q_{11}(1 - dx)(1 - dy) + Q_{21}dx(1 - dy) + Q_{12}(1 - dx)dy + Q_{22}dxdy$$

For $dx = 0.5$, $dy = 0.5$:

$$f(0.5, 0.5) = 100(0.5)(0.5) + 110(0.5)(0.5) + 120(0.5)(0.5) + 130(0.5)(0.5)$$

$$= 25 + 27.5 + 30 + 32.5 = 115$$

So, the new pixel at **(0.5, 0.5)** is **115**.

Image Rotation using Linear Interpolation

Rotate a pixel at **(3, 3)** by **45 degrees** using interpolation.

Step 1: Compute New Coordinates

Rotation formula:

$$x' = x\cos\theta - y\sin\theta$$

$$y' = x\sin\theta + y\cos\theta$$

For $(x, y) = (3, 3)$, $\theta = 45^\circ$:

$$x' = 3\cos 45^\circ - 3\sin 45^\circ = 3(0.707) - 3(0.707) = 0$$

$$y' = 3\sin 45^\circ + 3\cos 45^\circ = 3(0.707) + 3(0.707) = 4.24$$

Step 2: Interpolate Pixel Values

Since **(0, 4.24)** is between pixels (0,4) and (0,5), interpolate using linear interpolation:

$$f(4.24) = f(4) + (4.24 - 4) \times \frac{(f(5) - f(4))}{(5 - 4)}$$

Image Transformation using Linear Interpolation

Apply a **shear transformation** on an image point $(2, 3)$.

Step 1: Shear Transformation Matrix

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & sh_x \\ sh_y & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

For $sh_x = 0.5$, $sh_y = 0$:

$$x' = 2 + (0.5 \times 3) = 3.5, \quad y' = 3$$

Step 2: Interpolation

Since $x' = 3.5$ is between pixels $(3, 3)$ and $(4, 3)$, interpolate:

$$f(3.5) = f(3) + 0.5 \times (f(4) - f(3))$$

Contrast Enhancement using Linear Interpolation

Stretch contrast of an image from range **[50, 200]** to **[0, 255]**.

Step 1: Apply Linear Stretching Formula

$$I' = \frac{I - I_{\min}}{I_{\max} - I_{\min}} \times (O_{\max} - O_{\min}) + O_{\min}$$

For $I_{\min} = 50$, $I_{\max} = 200$, and a pixel value **I = 100**:

$$I' = \frac{100 - 50}{200 - 50} \times (255 - 0) + 0$$

$$= \frac{50}{150} \times 255 = 85$$

So, the enhanced pixel is **85**.

Edge Detection Preprocessing using Linear Interpolation

Apply Gaussian blur using interpolation before edge detection.

Step 1: Define Gaussian Kernel

A 3×3 Gaussian kernel:

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Step 2: Apply Convolution with Linear Interpolation

For pixel (x, y), interpolate:

$$I'(x,y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} K(i,j) \cdot I(x+i, y+j)$$

For a pixel with values:

$$\begin{bmatrix} 100 & 120 & 140 \\ 110 & 130 & 150 \\ 120 & 140 & 160 \end{bmatrix}$$

Applying convolution gives the new pixel value.

Implementing 1D Linear Interpolation

```
import numpy as np

def linear_interpolation(x, x_points, y_points):
    """
    Perform linear interpolation for a given x using known x_points and y_points.
    """
    x1, x2 = x_points
    y1, y2 = y_points
    return y1 + ((x - x1) / (x2 - x1)) * (y2 - y1)

# Example usage
x_points = (2, 6)
y_points = (4, 12)
x = 4
interpolated_value = linear_interpolation(x, x_points, y_points)
print(f"Interpolated value at x={x}: {interpolated_value}")
```

Implementing Bilinear Interpolation for Images

```
import cv2
import numpy as np

def bilinear_interpolation(image, new_height, new_width):
    """
    Resize an image using bilinear interpolation.
    """
    height, width, channels = image.shape
    resized_image = np.zeros((new_height, new_width, channels), dtype=np.uint8)

    x_ratio = width / new_width
    y_ratio = height / new_height

    for i in range(new_height):
        for j in range(new_width):
            x = j * x_ratio
            y = i * y_ratio

            x1, y1 = int(x), int(y)
            x2, y2 = min(x1 + 1, width - 1), min(y1 + 1, height - 1)

            dx, dy = x - x1, y - y1

            for c in range(channels):
                value = (
                    image[y1, x1, c] * (1 - dx) * (1 - dy) +
                    image[y1, x2, c] * dx * (1 - dy) +
                    image[y2, x1, c] * (1 - dx) * dy +
                    image[y2, x2, c] * dx * dy
                )
                resized_image[i, j, c] = int(value)

    return resized_image

# Load an image
image = cv2.imread("input.jpg")

# Resize using bilinear interpolation
resized_image = bilinear_interpolation(image, 300, 300)

# Show images
cv2.imshow("Original Image", image)
cv2.imshow("Resized Image", resized_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

4. **Histogram Equalization**

   - Improves brightness by spreading intensity values evenly across the image histogram.

3. Types of Brightness Interpolation Methods

1. **Global Methods**
   - Apply the same brightness adjustment to the entire image.
   - Example: Simple addition, gamma correction.
2. **Local Methods**
   - Adjust brightness based on local neighborhood pixel values.
   - Example: Adaptive histogram equalization.
3. **Piecewise Linear Interpolation**
   - Maps intensity values in segments to enhance contrast.
   - Example: Contrast stretching.

3. Types of Brightness Interpolation Methods

1. **Global Methods**
    - Apply the same brightness adjustment to the entire image.
    - Example: Simple addition, gamma correction.
2. **Local Methods**
    - Adjust brightness based on local neighborhood pixel values.
    - Example: Adaptive histogram equalization.

3. **Piecewise Linear Interpolation**

- Maps intensity values in segments to enhance contrast.
- Example: Contrast stretching.

Impementation

```
import cv2
import numpy as np
# Load the image
image = cv2.imread('image.jpg')
# Simple Brightness Adjustment
beta = 50  # Brightness factor
bright_image = cv2.convertScaleAbs(image, beta=beta)
# Linear Brightness Interpolation
alpha = 0.5  # Interpolation factor (0 to 1)
image2 = np.full_like(image, 255)  # White image
interpolated_image = cv2.addWeighted(image, alpha, image2, 1 - alpha, 0)
# Display Results
cv2.imshow('Original', image)
cv2.imshow('Brightened', bright_image)
cv2.imshow('Interpolated', interpolated_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Histogram Processing using Arithmetic & Logical Operations

- Arithmetic Operations:

- Addition, Subtraction, Multiplication, Division

- Logical Operations:

- AND, OR, XOR, NOT

- Used for image enhancement and segmentation.

Smoothing Spatial Filters

- Reduces noise by averaging pixel values.
- Types:

- Mean filter

- Gaussian filter

- Median filter

- Used in pre-processing for edge detection.

Canny Edge Detection

- Multi-stage edge detection algorithm:

1. Noise Reduction (Gaussian Filter)
2. Gradient Calculation
3. Non-Maximum Suppression
4. Edge Tracking by Hysteresis

- Produces accurate and thin edges.

2. Mathematical Background

The Canny Edge Detection algorithm involves the following steps:

Step 1: Noise Reduction (Gaussian Smoothing)

Since edge detection is susceptible to noise, the image is first smoothed using a **Gaussian filter**:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where **σ** is the standard deviation, determining the level of smoothing.

Step 2: Compute Gradient Magnitude and Direction

Edges are regions with strong intensity changes. We compute the gradients using **Sobel operators:**

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

Gradient magnitude:

$$G = \sqrt{G_x^2 + G_y^2}$$

Gradient direction:

$$\theta = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

Example

Step 1: Define the Input Image

Consider a **grayscale** 3×3 image with the following intensity values:

$$I = \begin{bmatrix} 100 & 100 & 100 \\ 100 & 150 & 100 \\ 100 & 100 & 100 \end{bmatrix}$$

The central pixel (150) is the one where we will compute the gradient.

Step 2: Apply Sobel Filters

The Sobel operators for **horizontal** (G_x) and **vertical** (G_y) gradients are:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

Step 3: Compute G_x and G_y

Compute G_x:

$$G_x = (-1)(100) + (0)(100) + (1)(100) + (-2)(100) + (0)(150) + (2)(100) + (-1)(100) + (0)(100) + (1)(100)$$

$$= (-100 + 0 + 100) + (-200 + 0 + 200) + (-100 + 0 + 100) = 0$$

Compute G_y:

$$G_y = (-1)(100) + (-2)(100) + (-1)(100) + (0)(100) + (0)(150) + (0)(100) + (1)(100) + (2)(100) + (1)(100)$$

$$= (-100 - 200 - 100) + (0 + 0 + 0) + (100 + 200 + 100) = 0$$

Thus, both $G_x = 0$ **and** $G_y = 0$ at the central pixel.

Step 4: Compute Gradient Magnitude

The gradient magnitude is given by:

$$G = \sqrt{G_x^2 + G_y^2}$$

$$= \sqrt{0^2 + 0^2} = \sqrt{0} = 0$$

Step 5: Compute Gradient Direction

The gradient direction (angle θ) is given by:

$$\theta = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

Since both G_x and G_y are zero, the gradient direction is undefined or considered to be **0**°.

Quantize the Gradient Direction

The gradient direction θ is quantized into four categories:

- **0° (horizontal)**
- **45° (diagonal right)**
- **90° (vertical)**
- **135° (diagonal left)**

For example:

- If $0^\circ \leq \theta < 22.5^\circ$ or $157.5^\circ \leq \theta < 180^\circ$, round it to **0° (horizontal)**
- If $22.5^\circ \leq \theta < 67.5^\circ$, round it to **45°**
- If $67.5^\circ \leq \theta < 112.5^\circ$, round it to **90°**
- If $112.5^\circ \leq \theta < 157.5^\circ$, round it to **135°**

3. Non-Maximum Suppression (NMS)

For each pixel (i, j), compare the gradient magnitude $G(i, j)$ with its two neighboring pixels in the gradient direction:

- If $\theta \approx 0^\circ$ (horizontal), compare $G(i, j)$ with $G(i, j-1)$ and $G(i, j+1)$.
- If $\theta \approx 45^\circ$ (diagonal right), compare $G(i, j)$ with $G(i-1, j+1)$ and $G(i+1, j-1)$.
- If $\theta \approx 90^\circ$ (vertical), compare $G(i, j)$ with $G(i-1, j)$ and $G(i+1, j)$.
- If $\theta \approx 135^\circ$ (diagonal left), compare $G(i, j)$ with $G(i-1, j-1)$ and $G(i+1, j+1)$.

Suppression Rule:
If $G(i, j)$ is not greater than both neighboring pixels, set $G(i, j) = 0$.

Numerical Example

Consider a **3×3** gradient magnitude matrix:

$$\begin{bmatrix} 10 & 20 & 10 \\ 30 & 50 & 30 \\ 10 & 20 & 10 \end{bmatrix}$$

Assume the gradient directions are:

$$\begin{bmatrix} 0^\circ & 90^\circ & 0^\circ \\ 90^\circ & 90^\circ & 90^\circ \\ 0^\circ & 90^\circ & 0^\circ \end{bmatrix}$$

Applying NMS

- The center pixel $G(1,1) = 50$ has a **90° gradient** → Compare with top ($G(0,1) = 20$) and bottom ($G(2,1) = 20$).
  - Since $50 > 20$ and $50 > 20$, keep **50**.
- The pixel $G(1,0) = 30$ has a **0° gradient** → Compare with left ($G(1,-1)$, out of bounds) and right ($G(1,1) = 50$).
  - Since $30 < 50$, **set to 0**.
- The pixel $G(1,2) = 30$ has a **0° gradient** → Compare with left ($G(1,1) = 50$) and right ($G(1,3)$, out of bounds).
  - Since $30 < 50$, **set to 0**.

After applying **Non-Maximum Suppression**, the matrix becomes:

$$\begin{bmatrix} 0 & 20 & 0 \\ 0 & 50 & 0 \\ 0 & 20 & 0 \end{bmatrix}$$

Edge Tracking by Hysteresis

Step 1: Define Thresholds

Two thresholds are set:

- **High threshold** (T_h): Pixels with gradient magnitude above this value are **strong edges**.
- **Low threshold** (T_l): Pixels with gradient magnitude below this value are **suppressed** unless they are connected to a strong edge.

Example thresholds:

$$T_h = 40, \quad T_l = 15$$

Step 2: Gradient Magnitude Matrix

Consider a **5×5** gradient magnitude matrix G:

$$\begin{bmatrix} 10 & 20 & 25 & 20 & 10 \\ 15 & 50 & 80 & 50 & 15 \\ 20 & 90 & 120 & 90 & 20 \\ 15 & 50 & 80 & 50 & 15 \\ 10 & 20 & 25 & 20 & 10 \end{bmatrix}$$

Step 3: Classify Pixels

Each pixel is classified based on the thresholds:

- **Strong Edge (S)**: $G(i, j) \geq T_h$
- **Weak Edge (W)**: $T_l \leq G(i, j) < T_h$
- **Non-Edge (0)**: $G(i, j) < T_l$

Applying $T_h = 40$ and $T_l = 15$:

$$\begin{bmatrix} 0 & W & W & W & 0 \\ W & S & S & S & W \\ W & S & S & S & W \\ W & S & S & S & W \\ 0 & W & W & W & 0 \end{bmatrix}$$

Step 4: Edge Tracing

Now, weak edges (W) are kept **only if they are connected to a strong edge (S)**.

Step 4.1: Find Connected Weak Pixels

Start from strong edges and trace connected weak edges using **8-connectivity** (adjacent neighbors).

1. **Keep all strong edges.**
2. **For each strong edge, check its 8 neighbors:**
    - If a weak edge is connected, promote it to a strong edge.
    - Repeat recursively for newly promoted strong edges.

Step 4.2: Update the Matrix

After tracing:

$$\begin{bmatrix} 0 & 0 & W & 0 & 0 \\ 0 & S & S & S & 0 \\ W & S & S & S & W \\ 0 & S & S & S & 0 \\ 0 & 0 & W & 0 & 0 \end{bmatrix}$$

- **Isolated weak edges (not connected to strong edges) are removed.**
- **Connected weak edges are converted into strong edges.**

Final Output

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & S & S & S & 0 \\ 0 & S & S & S & 0 \\ 0 & S & S & S & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

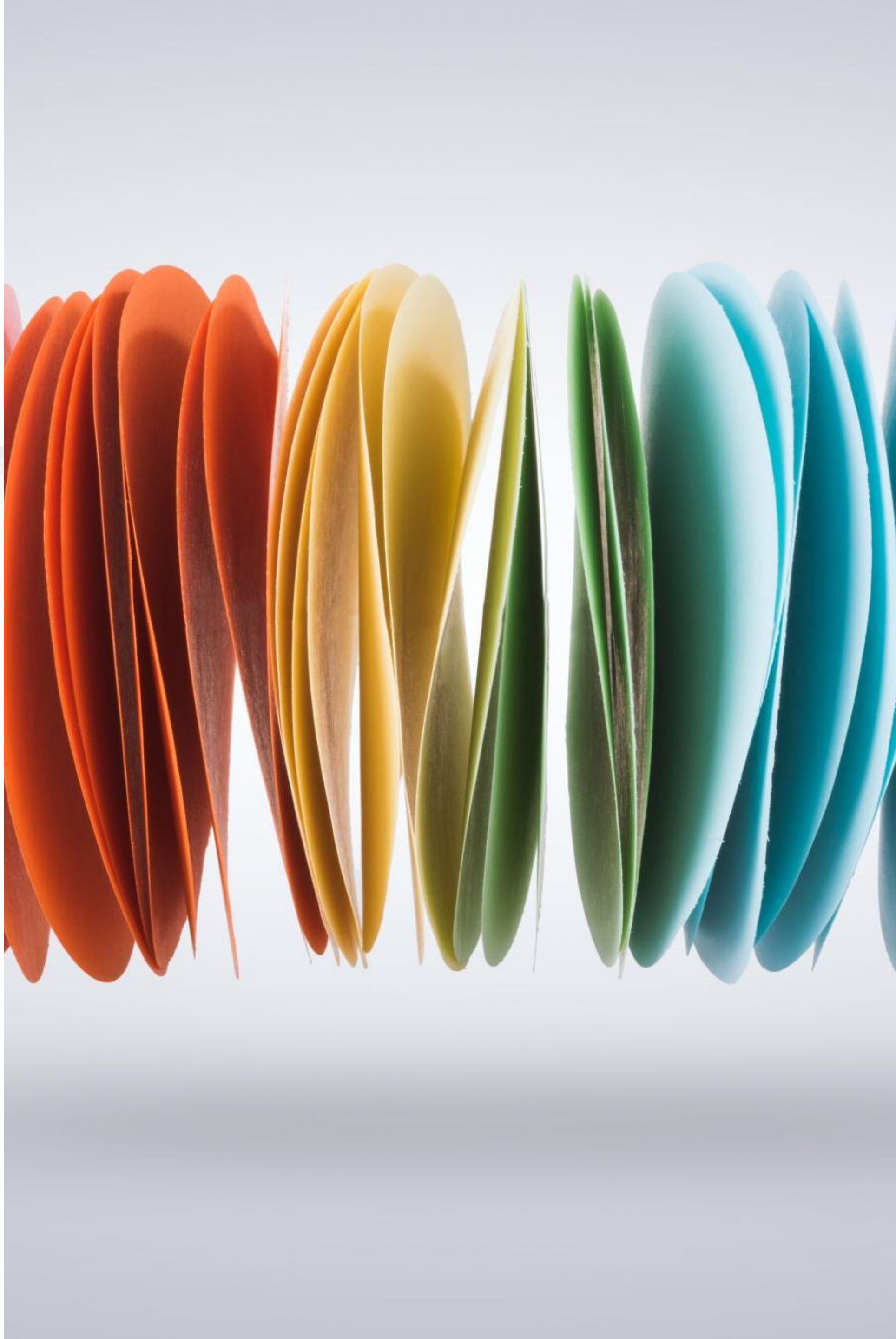
Final edges contain only strong edges and their connected weak edges.

```python
import numpy as np
import cv2

def non_maximum_suppression(gradient_magnitude, gradient_direction):
    """Applies Non-Maximum Suppression (NMS) to thin edges."""
    rows, cols = gradient_magnitude.shape
    suppressed = np.zeros((rows, cols), dtype=np.float32)

    # Quantize gradient directions to nearest 0, 45, 90, 135 degrees
    angle = gradient_direction * (180.0 / np.pi)  # Convert radians to degrees
    angle[angle < 0] += 180  # Ensure all angles are positive

    for i in range(1, rows - 1):
        for j in range(1, cols - 1):
            q, r = 255, 255  # Initialize neighbors

            # Angle 0 degrees (horizontal edge)
            if (0 <= angle[i, j] < 22.5) or (157.5 <= angle[i, j] <= 180):
                q = gradient_magnitude[i, j + 1]
                r = gradient_magnitude[i, j - 1]
            # Angle 45 degrees (diagonal right)
            elif 22.5 <= angle[i, j] < 67.5:
                q = gradient_magnitude[i + 1, j - 1]
                r = gradient_magnitude[i - 1, j + 1]
            # Angle 90 degrees (vertical edge)
            elif 67.5 <= angle[i, j] < 112.5:
                q = gradient_magnitude[i + 1, j]
                r = gradient_magnitude[i - 1, j]
            # Angle 135 degrees (diagonal left)
            elif 112.5 <= angle[i, j] < 157.5:
                q = gradient_magnitude[i - 1, j - 1]
                r = gradient_magnitude[i + 1, j + 1]

            # Suppress non-max values
            if (gradient_magnitude[i, j] >= q) and (gradient_magnitude[i, j] >= r):
                suppressed[i, j] = gradient_magnitude[i, j]
            else:
                suppressed[i, j] = 0

    return suppressed

# Example usage with Sobel edge detection
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)
Gx = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
Gy = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)

gradient_magnitude = np.sqrt(Gx**2 + Gy**2)
gradient_direction = np.arctan2(Gy, Gx)  # In radians

nms_result = non_maximum_suppression(gradient_magnitude, gradient_direction)
cv2.imshow('Non-Maximum Suppression', nms_result)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Implementation

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
# Load the image in grayscale
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)
# Apply Gaussian Blur to reduce noise
blurred = cv2.GaussianBlur(image, (5,5), 1.4)
# Apply Canny Edge Detection
edges = cv2.Canny(blurred, 50, 150)  # (image, T_low, T_high)
# Display results
plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.title("Original Image")
plt.imshow(image, cmap='gray')
plt.subplot(1,2,2)
plt.title("Canny Edge Detection")
plt.imshow(edges, cmap='gray')
plt.show()
```

Sharpening Spatial Filters: A Complete Tutorial



- Sharpening filters are used in image processing to enhance the edges and fine details in an image. These filters work by emphasizing **high-frequency components**, which correspond to rapid intensity changes (edges).

Types of Sharpening Filters

- Sharpening filters are typically implemented using **spatial filtering** with convolution masks (kernels). The main types include:

1. **Laplacian Filter** (Second derivative-based)
2. **Unsharp Masking** (Smoothing + Subtraction)
3. **High-Boost Filtering** (Generalized unsharp masking)

3. Laplacian Filter (Second Derivative-Based)

The **Laplacian operator** detects edges by computing the second derivative of an image:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Laplacian Kernel Examples

The Laplacian operator is implemented using convolution masks such as:

4-neighbor kernel (cross pattern)

$$L_4 = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Unsharp Masking (First Derivative-Based)

This technique sharpens an image by subtracting a **blurred version** from the original.

Formula

$$\text{Sharpened Image} = \text{Original} + k(\text{Original} - \text{Blurred})$$

where:

- k is a scaling factor (typically between 1 and 2).

Steps to Apply Unsharp Masking

1. Blur the original image using **Gaussian filtering**.
2. Subtract the blurred image from the original image.
3. Add the **difference** back to the original image, scaled by k.

High-Boost Filtering (Generalized Unsharp Masking)

High-boost filtering is an extension of unsharp masking where we multiply the original image by a factor A before subtracting the blurred version:

$$\text{High-Boost Image} = A \cdot \text{Original} - \text{Blurred}$$

For standard **unsharp masking**, $A = 1.5$. If $A > 1.5$, more sharpening occurs.

8-neighbor kernel (diagonal + cross pattern)

$$L_8 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Steps to Apply Laplacian Filtering

1. Convert the image to grayscale.
2. Apply the **Laplacian kernel** using convolution.
3. Enhance edges by subtracting the Laplacian from the original image.

Corner Detection using Python

- Detects corners in an image based on intensity variations.
- Example methods:

- Harris Corner Detection

- Shi-Tomasi Corner Detection

- Used in feature extraction and image matching.

2. Mathematical Background

Step 1: Compute Image Gradients

We first compute the intensity gradients using the Sobel opera

$$I_x = \frac{\partial I}{\partial x}, \quad I_y = \frac{\partial I}{\partial y}$$

Step 2: Compute Structure Tensor (Second Moment Matrix)

A small window is considered around each pixel, and the struct

computed as:

$$M = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}$$

Step 3: Compute Corner Response Function

Using the determinant and trace of M, the **Harris response** is calculated as:

$$R = \det(M) - k \cdot (\text{trace}(M))^2$$

where:

- $\det(M) = (\sum I_x^2)(\sum I_y^2) - (\sum I_x I_y)^2$
- $\text{trace}(M) = \sum I_x^2 + \sum I_y^2$
- k **is an empirically determined constant (usually 0.04 - 0.06).**

If R **is large**, it indicates a corner.

Harris Corner Detection using OpenCV

- import cv2
- import numpy as np

- # Load image in grayscale
- image = cv2.imread('image.jpg')
- gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

- # Apply Harris Corner Detection
- gray = np.float32(gray)
- harris_corners = cv2.cornerHarris(gray, blockSize=2, ksize=3, k=0.04)

- # Dilate to mark the corners
- harris_corners = cv2.dilate(harris_corners, None)

- # Mark corners in red
- image[harris_corners > 0.01 * harris_corners.max()] = [0, 0, 255]

- # Display result
- cv2.imshow('Harris Corner Detection', image)
- cv2.waitKey(0)
- cv2.destroyAllWindows()

Shi-Tomasi (Good Features to Track) Method

- Shi-Tomasi improves Harris detection by using **eigenvalues of MMM** instead of the determinant.

Shi-Tomasi

- # Load image in grayscale
- gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

- # Detect corners using Shi-Tomasi method
- corners = cv2.goodFeaturesToTrack(gray, maxCorners=50, qualityLevel=0.01, minDistance=10)

- # Convert corners to integer values
- corners = np.int0(corners)

- # Draw detected corners
- for i in corners:
-     x, y = i.ravel()
-     cv2.circle(image, (x, y), 3, (0, 255, 0), -1)

- # Display result
- cv2.imshow('Shi-Tomasi Corner Detection', image)
- cv2.waitKey(0)
- cv2.destroyAllWindows()