|  |  |
| --- | --- |
| **EXERCISE NO:5**  **DATE:06/02/2025** | **NOISE , FILTERS AND SOBEL** |

**AIM:**

Adding various types of noise (Gaussian, Impulse, Laplacian, etc.) to an image and applying filters (Mean, Median, Sobel, etc.) for edge detection and noise reduction in a step-by-step process.

**MOON PIC**

**ALGORITHM:**

1. **Add Noise**: The algorithm starts by adding different types of noise to an image. These noise types can include Gaussian, impulse, Laplacian, multiplicative Gaussian, Poisson, random, and uniform noises. Each type of noise is added to the image by altering its pixel values in a specific way.
2. **Apply Filters**: After adding noise, the algorithm applies various filters to the noisy images to enhance edge detection and reduce noise. The filters include arithmetic mean, geometric mean, harmonic mean, median, max, min, midpoint, adaptive median, and adaptive mean filters.
3. **Edge Detection**: The algorithm employs edge detection techniques such as Sobel operators with different kernel sizes (2x2, 3x3, 5x5) and Roberts cross operators to identify and highlight edges in the images.
4. **Histogram Computation**: The algorithm calculates histograms of pixel intensity values for the filtered and edge-detected images to visualize the distribution of pixel intensities.
5. **Visualization**: Finally, the algorithm visualizes the original, noisy, filtered, and edge-detected images along with their histograms using matplotlib to provide a comprehensive understanding of the processing steps and their effects on the images.

**FUNCTION:**

1. add\_noise: Adds different types of noise (Gaussian, impulse, Laplacian, etc.) to an image.

2. apply\_filter: Applies a specified filter function to an image using a given kernel size.

3. arithmetic\_mean\_filter: Applies an arithmetic mean filter to an image for noise reduction.

4. geometric\_mean\_filter: Applies a geometric mean filter to an image for noise reduction.

5. harmonic\_mean\_filter: Applies a harmonic mean filter to an image for noise reduction.

6. median\_filter: Applies a median filter to an image to remove salt-and-pepper noise.

7. max\_filter: Applies a max filter to an image for noise reduction.

8. min\_filter: Applies a min filter to an image for noise reduction.

9. midpoint\_filter: Applies a midpoint filter to an image for noise reduction.

10. adaptive\_median\_filter: Applies an adaptive median filter to an image to remove salt-and-pepper noise.

11. adaptive\_mean\_filter: Applies an adaptive mean filter to an image for noise reduction.

12. sobel\_operator: Applies Sobel operators to an image for edge detection.

13. gradient\_image: Computes the gradient magnitude of an image using gradient filters.

14. discrete\_derivative: Computes the discrete derivative of an image.

15. gaussian\_smoothing: Applies Gaussian smoothing to an image for noise reduction.

16. compute\_histogram: Computes the histogram of pixel intensity values in an image.

17. show\_noisy\_images\_with\_histogram: Displays noisy images along with their histograms.

18. show\_filtered\_images: Displays filtered images using various filters.

19. show\_sobel\_images: Displays images processed with Sobel operators of different kernel sizes.

20. show\_edge\_detection: Displays edge detection results using various methods including Sobel, Roberts, and Canny.

**CODE:**

**import** cv2

**import** numpy **as** np

**import** random

**import** matplotlib.pyplot **as** plt

**def** add\_noise(image, noise\_type**=**"gaussian"):

**if** noise\_type **==** "gaussian":

row, col, ch **=** image**.**shape

mean **=** 0

sigma **=** 25

gauss **=** np**.**random**.**normal(mean, sigma, (row, col, ch))**.**astype('uint8')

noisy **=** cv2**.**add(image, gauss)

**return** noisy

**elif** noise\_type **==** "impulse":

amount **=** 0.02

noisy **=** np**.**copy(image)

num\_impulse **=** np**.**ceil(amount **\*** image**.**size)

coords **=** [np**.**random**.**randint(0, i **-** 1, int(num\_impulse)) **for** i **in** image**.**shape]

noisy[coords[0], coords[1], :] **=** np**.**random**.**choice([0, 255], size**=**(len(coords[0]), 1))

**return** noisy

**elif** noise\_type **==** "laplacian":

laplacian\_noise **=** cv2**.**Laplacian(image, cv2**.**CV\_64F)

noisy **=** cv2**.**convertScaleAbs(image **+** laplacian\_noise)

**return** noisy

**elif** noise\_type **==** "multiplicative\_gaussian":

row, col, ch **=** image**.**shape

gauss **=** np**.**random**.**normal(0, 0.2, (row, col, ch))

noisy **=** image **\*** (1 **+** gauss)

**return** np**.**clip(noisy, 0, 255)**.**astype(np**.**uint8)

**elif** noise\_type **==** "poisson":

noisy **=** np**.**random**.**poisson(image**.**astype(np**.**uint8)**.**astype(float))**.**astype('uint8')

**return** noisy

**elif** noise\_type **==** "random":

random\_noise **=** np**.**random**.**randint(0, 256, image**.**shape, dtype**=**'uint8')

noisy **=** cv2**.**addWeighted(image, 0.5, random\_noise, 0.5, 0)

**return** noisy

**elif** noise\_type **==** "uniform":

row, col, ch **=** image**.**shape

uniform\_noise **=** np**.**random**.**uniform(**-**50, 50, (row, col, ch))**.**astype('uint8')

noisy **=** cv2**.**add(image, uniform\_noise)

**return** noisy

**else**:

**raise** ValueError("Unsupported noise type. Choose from: 'gaussian', 'impulse', 'laplacian', 'multiplicative\_gaussian', 'poisson', 'random', 'uniform'")

**def** show\_noisy\_images\_with\_histogram(image):

noise\_types **=** ["original", "gaussian", "impulse", "laplacian", "multiplicative\_gaussian", "poisson", "random", "uniform"]

fig, axes **=** plt**.**subplots(len(noise\_types), 2, figsize**=**(10, 10))

**for** i, noise **in** enumerate(noise\_types):

**if** noise **==** "original":

noisy\_img **=** image

**else**:

noisy\_img **=** add\_noise(image, noise)

axes[i, 0]**.**imshow(cv2**.**cvtColor(noisy\_img, cv2**.**COLOR\_BGR2RGB))

axes[i, 0]**.**set\_title(f"{noise**.**capitalize()} Noise")

axes[i, 0]**.**axis("off")

gray\_img **=** cv2**.**cvtColor(noisy\_img, cv2**.**COLOR\_BGR2GRAY)

hist\_values **=** cv2**.**calcHist([gray\_img], [0], **None**, [256], [0, 256])

axes[i, 1]**.**plot(hist\_values, color**=**'black')

axes[i, 1]**.**set\_xlim([0, 256])

axes[i, 1]**.**set\_title(f"Histogram of {noise**.**capitalize()} Noise")

plt**.**tight\_layout()

plt**.**show()

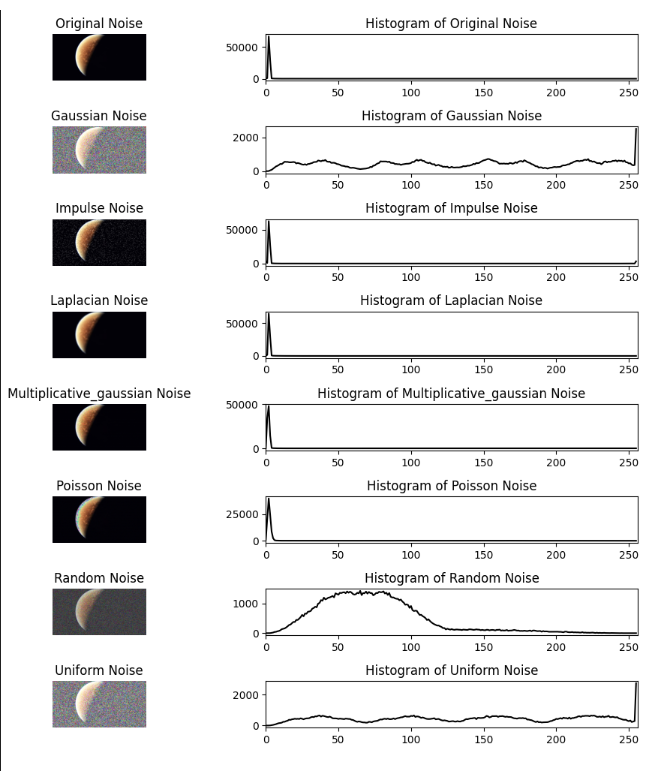
*# Example Usage*

**if** \_\_name\_\_ **==** "\_\_main\_\_":

image **=** cv2**.**imread("Downloads/naruto .jpeg")

show\_noisy\_images\_with\_histogram(image)

**OUTPUT:**



**CODE:**

import cv2

import numpy as np

import matplotlib.pyplot as plt

def apply\_filter(image, kernel\_size, function):

padded\_image = np.pad(image, kernel\_size // 2, mode='edge')

filtered\_image = np.zeros\_like(image, dtype=np.float32)

for i in range(image.shape[0]):

for j in range(image.shape[1]):

region = padded\_image[i:i+kernel\_size, j:j+kernel\_size].flatten()

filtered\_image[i, j] = function(region)

return filtered\_image.astype(np.uint8)

def arithmetic\_mean\_filter(image, kernel\_size=3):

return apply\_filter(image, kernel\_size, lambda x: np.mean(x))

def geometric\_mean\_filter(image, kernel\_size=3):

return apply\_filter(image, kernel\_size, lambda x: np.exp(np.mean(np.log(np.maximum(x, 1e-5)))))

def harmonic\_mean\_filter(image, kernel\_size=3):

return apply\_filter(image, kernel\_size, lambda x: len(x) / np.sum(1.0 / np.maximum(x, 1e-5)))

def median\_filter(image, kernel\_size=3):

return apply\_filter(image, kernel\_size, lambda x: np.median(x))

def max\_filter(image, kernel\_size=3):

return apply\_filter(image, kernel\_size, lambda x: np.max(x))

def min\_filter(image, kernel\_size=3):

return apply\_filter(image, kernel\_size, lambda x: np.min(x))

def midpoint\_filter(image, kernel\_size=3):

return apply\_filter(image, kernel\_size, lambda x: int(np.max(x) + np.min(x)) / 2.0)

def adaptive\_median\_filter(image, kernel\_size=3):

def adaptive\_median(values):

sorted\_vals = np.sort(values)

median = np.median(sorted\_vals)

min\_val, max\_val = sorted\_vals[0], sorted\_vals[-1]

if min\_val < median < max\_val:

return median

else:

return values[len(values) // 2]

return apply\_filter(image, kernel\_size, adaptive\_median)

def adaptive\_mean\_filter(image, kernel\_size=3):

return apply\_filter(image, kernel\_size, lambda x: np.mean(x))

def show\_filtered\_images(image):

filters = {

"Original": image,

"Arithmetic Mean": arithmetic\_mean\_filter(image),

"Geometric Mean": geometric\_mean\_filter(image),

"Harmonic Mean": harmonic\_mean\_filter(image),

"Median": median\_filter(image),

"Max": max\_filter(image),

"Min": min\_filter(image),

"Midpoint": midpoint\_filter(image),

"Adaptive Median": adaptive\_median\_filter(image),

"Adaptive Mean": adaptive\_mean\_filter(image)

}

fig, axes = plt.subplots(2, 5, figsize=(15, 6))

for ax, (title, filtered\_img) in zip(axes.flat, filters.items()):

ax.imshow(filtered\_img, cmap='gray')

ax.set\_title(title)

ax.axis("off")

plt.tight\_layout()

plt.show()

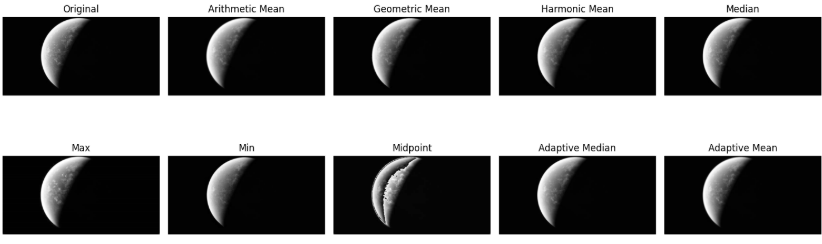
*# Example Usage*

if \_\_name\_\_ == "\_\_main\_\_":

image = cv2.imread("Downloads/naruto .jpeg", cv2.IMREAD\_GRAYSCALE)

show\_filtered\_images(image)

**OUTPUT:**

****

**CODE:**

import cv2

import numpy as np

import matplotlib.pyplot as plt

def apply\_filter(image, kernel):

k\_size = kernel.shape[0] *# Get kernel size*

pad = k\_size // 2

padded\_image = np.pad(image.astype(np.float32), pad, mode='edge')

filtered\_image = np.zeros\_like(image, dtype=np.float32)

for i in range(image.shape[0]):

for j in range(image.shape[1]):

region = padded\_image[i:i+k\_size, j:j+k\_size]

filtered\_image[i, j] = np.sum(region \* kernel)

return np.clip(filtered\_image, 0, 255).astype(np.uint8)

def sobel\_operator(image, kernel\_size=3):

if kernel\_size == 2:

Gx = np.array([[1, 0], [0, -1]], dtype=np.float32)

Gy = np.array([[0, 1], [-1, 0]], dtype=np.float32)

elif kernel\_size == 3:

Gx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype=np.float32)

Gy = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]], dtype=np.float32)

elif kernel\_size == 5:

Gx = np.array([[-2, -1, 0, 1, 2],

[-2, -1, 0, 1, 2],

[-4, -2, 0, 2, 4],

[-2, -1, 0, 1, 2],

[-2, -1, 0, 1, 2]], dtype=np.float32)

Gy = np.array([[-2, -2, -4, -2, -2],

[-1, -1, -2, -1, -1],

[0, 0, 0, 0, 0],

[1, 1, 2, 1, 1],

[2, 2, 4, 2, 2]], dtype=np.float32)

else:

raise ValueError("Invalid kernel size. Choose 2, 3, or 5.")

sobel\_x = apply\_filter(image, Gx)

sobel\_y = apply\_filter(image, Gy)

sobel\_combined = np.sqrt(sobel\_x.astype(np.float32)\*\*2 + sobel\_y.astype(np.float32)\*\*2)

return np.clip(sobel\_combined, 0, 255).astype(np.uint8)

def show\_sobel\_images(image):

kernel\_sizes = [2, 3, 5]

fig, axes = plt.subplots(1, len(kernel\_sizes) + 1, figsize=(15, 5))

axes[0].imshow(image, cmap='gray')

axes[0].set\_title("Original")

axes[0].axis("off")

for ax, size in zip(axes[1:], kernel\_sizes):

sobel\_img = sobel\_operator(image, size)

ax.imshow(sobel\_img, cmap='gray')

ax.set\_title(f"Sobel {size}x{size}")

ax.axis("off")

plt.tight\_layout()

plt.show()

*# Example Usage*

if \_\_name\_\_ == "\_\_main\_\_":

image = cv2.imread("Downloads/naruto .jpeg", cv2.IMREAD\_GRAYSCALE)

if image is None:

raise ValueError("Error loading image. Check file path.")

image = image.astype(np.float32) *# Ensure image is float before processing*

show\_sobel\_images(image)

**OUTPUT:**

****

**RESULTS :**

WE SUCCESSFULLY IMPLEMENTED AND EXCUTED OF THE NOISE , FILTERS , SOBEL: