MARCH 2024

IMAGE FILTERING & EDGE DETECTION

TEAM 14

- AHMED MOHAMED ALI
- ALI SHERIF
- MUHANNAD ABDALLAH
- OSAMA MOHAMED ALI

SUBMITTED TO

- DR. AHMED M. BADAWI
- ENG. LAILA ABBAS
- ENG. OMAR MUHAMMAD

TABLE OF CONTENTS

INTRODUCTION

In the realm of digital image processing, efficiency and user-friendliness are key. Our project aims to fulfill these requirements by delivering a comprehensive Image Processing application. Designed for users of all skill levels, our application seamlessly integrates powerful functionality with an intuitive user interface. By harnessing the capabilities of Python's OpenCV library and PyQt5 framework, users can manipulate images effortlessly and with precision.

Key Features:

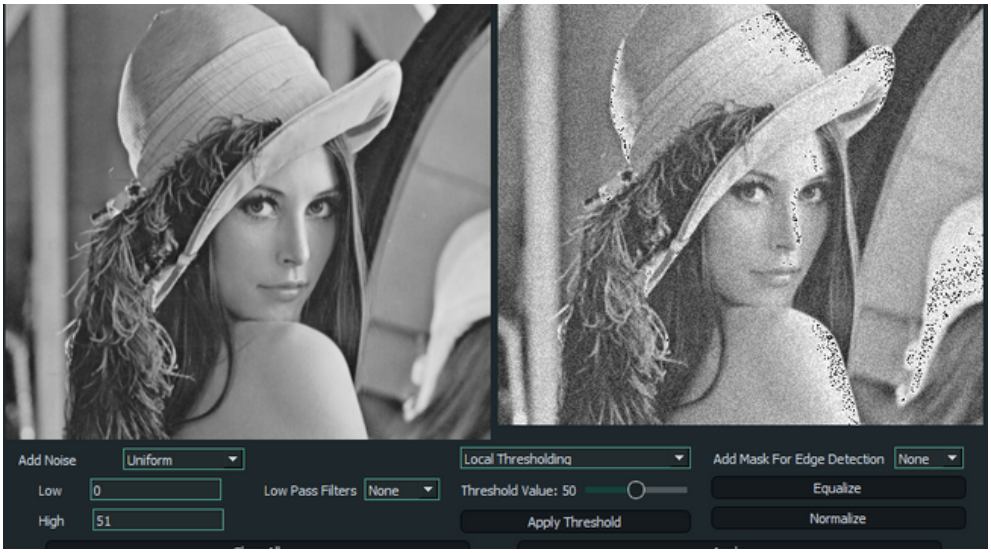
- Image Loading and Display: Load images from various formats and view them in real-time.
- Noise Addition: Simulate Gaussian, uniform, and salt and pepper noise for experimentation.
- Filtering Techniques: Apply average, Gaussian, and median filters for smoothing and edge detection.
- Masking Operations: Utilize Sobel, Roberts, Prewitt, and Canny masks for feature extraction.
- Histogram Equalization: Enhance image contrast and clarity through histogram equalization.
- Normalization and Thresholding: Adjust image contrast and segment objects with precision.
- RGB Transformation: Easily convert to RGB and see the histograms and distribution curves
- Hybrid Image Generation: Create compelling visual effects by blending images with different frequency content.

.

NOISE AND FILTERING

Uniform Noise:

- **Explanation**: Uniform noise is simulated by generating random values from a uniform distribution within a specified range and adding them to each pixel in the image. The range of values determines the amplitude of the noise, with lower and higher values representing darker and brighter noise, respectively.



- **Noise Addition Algorithm:** The add_uniform_noise function generates random values from a uniform distribution within the specified range (low to high). These random values are then added to each pixel in the image, resulting in the introduction of uniform noise.

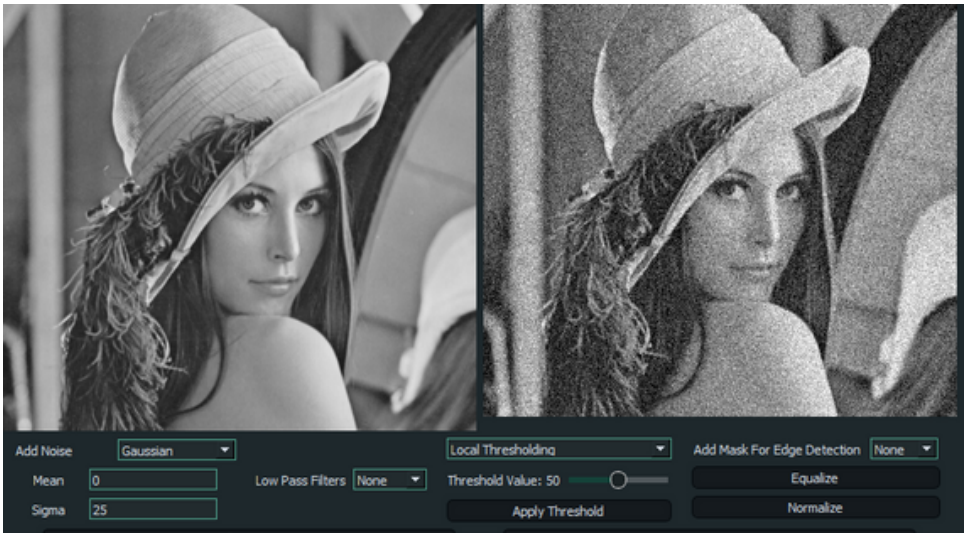
```
def add_uniform_noise(self, image, low=0, high=255 * 0.2):
    row, col = image.shape
    noise = np.zeros((row, col))
    for i in range(row):
        for j in range(col):
            noise[i, j] = np.random.uniform(low, high)  # Generate random numbe
    noisy = (image) + noise
    return noisy
```

- **Observation:** Uniform noise introduces a uniform graininess or mottling effect across the image, with the intensity of the noise determined by the range of values used. This noise can degrade image quality and affect perceptual clarity.

NOISE AND FILTERING

Gaussian Noise:

- **Explanation:** Gaussian noise is simulated by generating random values from a Gaussian distribution with a specified mean and standard deviation, and adding them to each pixel in the image. The mean represents the center of the distribution, while the standard deviation controls the spread or intensity of the noise.



- **Noise Addition Algorithm:** The add_gaussian_noise function generates random values from a Gaussian distribution with a specified mean (mu) and standard deviation (sigma). These random values are then added to each pixel in the image, resulting in the introduction of Gaussian noise.

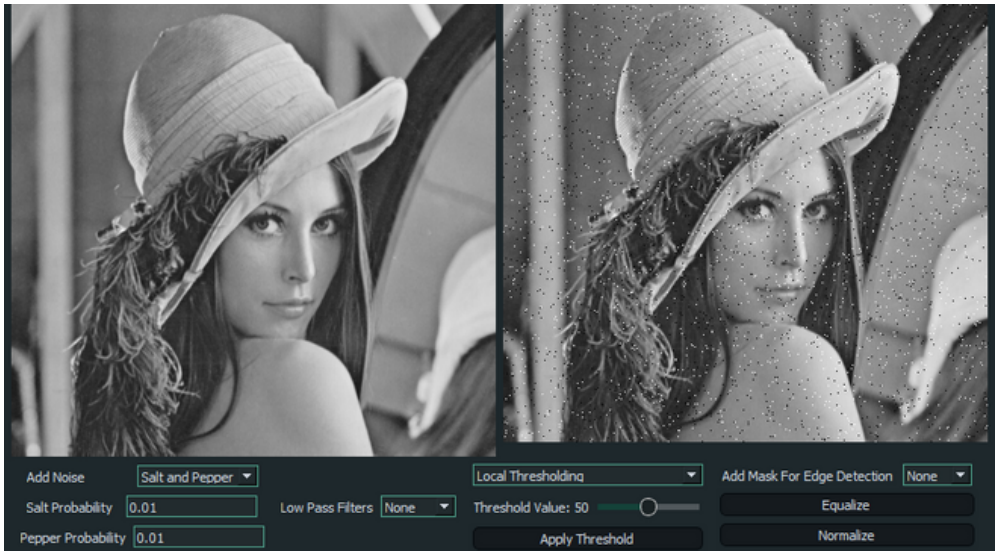
```
def add_gaussian_noise(self, image, mean=0, sigma=25):
    if len(image.shape) == 2:  # Check if the image is grayscale
        row, col = image.shape
        gauss = np.random.normal(mean, sigma, size: (row, col))
        noisy = image + gauss  # Scale the noise intensity
    else:  # Image is RGB/BGR
        row, col, ch = image.shape
        gauss = np.random.normal(mean, sigma, size: (row, col, ch))
        noisy = image + gauss  # Scale the noise intensity

    # Clip values to [0, 255] and convert to uint8
    noisy = np.clip(noisy, a_min: 0, a_max: 255).astype(np.uint8)
    return noisy
```

- **Observation:** Gaussian noise adds a more subtle and natural-looking variation to the image compared to uniform noise. It mimics real-world noise scenarios encountered in imaging systems. However, excessive Gaussian noise can still degrade image quality and affect perceptual clarity, especially if the standard deviation is high.

NOISE AND FILTERING

Salt and Pepper Noise:

- **Explanation:** Salt and pepper noise is characterized by the occurrence of sporadic bright (salt) and dark (pepper) pixels in the image. This type of noise simulates sudden and random errors or disturbances in the image capture or transmission process. The probabilities of adding salt and pepper noise control the density of these outliers, with higher probabilities leading to a greater number of affected pixels.



- **Algorithm:** The add_salt_and_pepper function randomly selects a subset of pixels in the image based on the specified probabilities (salt_prob and pepper_prob). These selected pixels are then set to either the maximum intensity value (salt) or the minimum intensity value (pepper), effectively introducing localized spikes in pixel intensities.

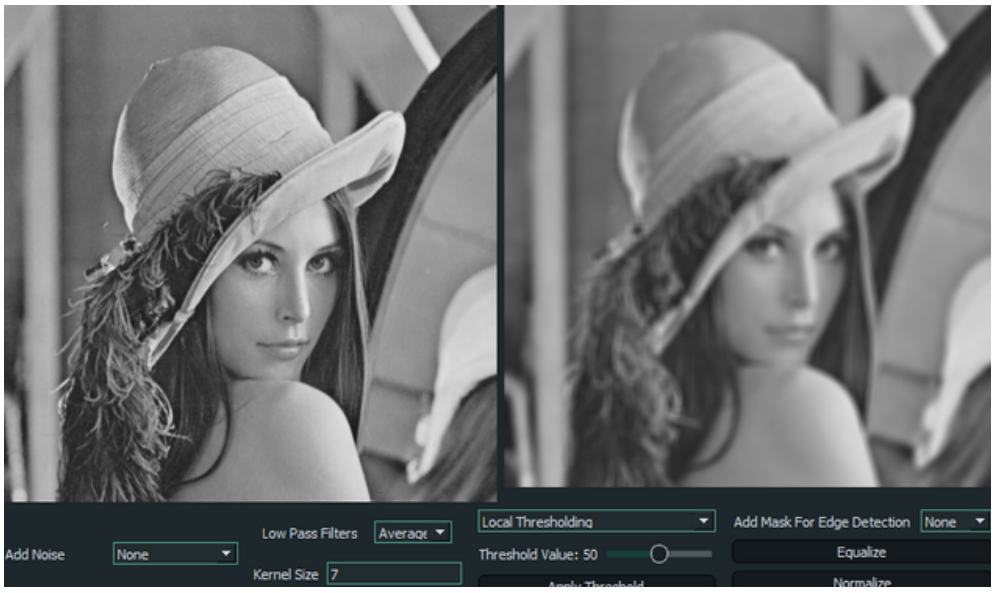
```
def add_salt_and_pepper(self, image, salt_prob=0.01, pepper_prob=0.01):
    noisy_image = np.copy(image)

    # Add salt noise
    salt_mask = np.random.rand(*image.shape) < salt_prob
    noisy_image[salt_mask] = 255

    # Add pepper noise
    pepper_mask = np.random.rand(*image.shape) < pepper_prob
    noisy_image[pepper_mask] = 0
    return noisy_image
```

NOISE AND FILTERING

Observation: Salt and pepper noise manifest as isolated bright and dark spots scattered throughout the image, disrupting the overall visual appearance. While salt and pepper noise can occur naturally in real-world imaging systems due to various factors such as sensor malfunctions or transmission errors, excessive levels of this noise can significantly degrade image quality and hinder accurate analysis or interpretation. Therefore, effective noise reduction techniques are essential for mitigating the adverse effects of salt and pepper noise on image fidelity and usability.

Average Filter

- **Explanation:** The average filter, also known as the mean filter, is a simple smoothing technique used to reduce noise and blur an image. It operates by replacing each pixel value in the image with the average value of its neighboring pixels within a defined kernel region. The size of the kernel determines the extent of smoothing applied to the image.



- **Algorithm:** The average_filter function creates a kernel of specified size filled with ones, representing equal weights for each pixel in the kernel window. This kernel is then convolved with the input image using the cv2.filter2D function, effectively averaging the pixel values within the kernel region. The borderType=cv2.BORDER_REFLECT parameter ensures that the image borders are handled properly during convolution.

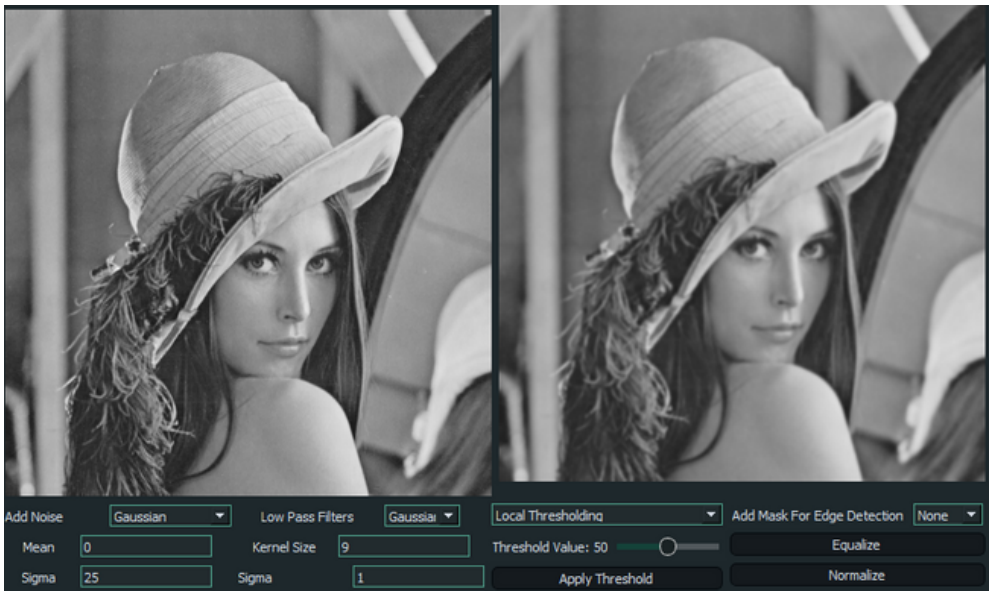
NOISE AND FILTERING

```
def average_filter(self, image, kernel_size=(3, 3)):
    # Define the kernel
    kernel = np.ones(kernel_size, dtype=np.float32) / (kernel_size[0] * kernel_size[1])

    # Perform convolution
    filtered_image = cv2.filter2D(image, -1, kernel, borderType=cv2.BORDER_REFLECT)
```

Observation: The average filter produces a blurring effect on the image, smoothing out high-frequency noise and sharp transitions between pixel intensities. However, it may also cause a loss of fine details and edges in the image, leading to a loss of sharpness or clarity.

Gaussian Filter:

Explanation: The Gaussian filter is a popular choice for image smoothing and noise reduction due to its ability to preserve image details while effectively reducing noise. It operates by convolving the image with a Gaussian kernel, which applies a weighted average to the neighboring pixels. The standard deviation (sigma) parameter controls the spread or intensity of the Gaussian distribution, influencing the degree of smoothing applied to the image.



Algorithm: The gaussian_filter function generates a 2D Gaussian kernel using the gaussian_kernel function with specified parameters (kernel size and sigma). This Gaussian kernel is then convolved with the input image using the cv2.filter2D function, effectively applying a weighted average to each pixel based on the Gaussian distribution.

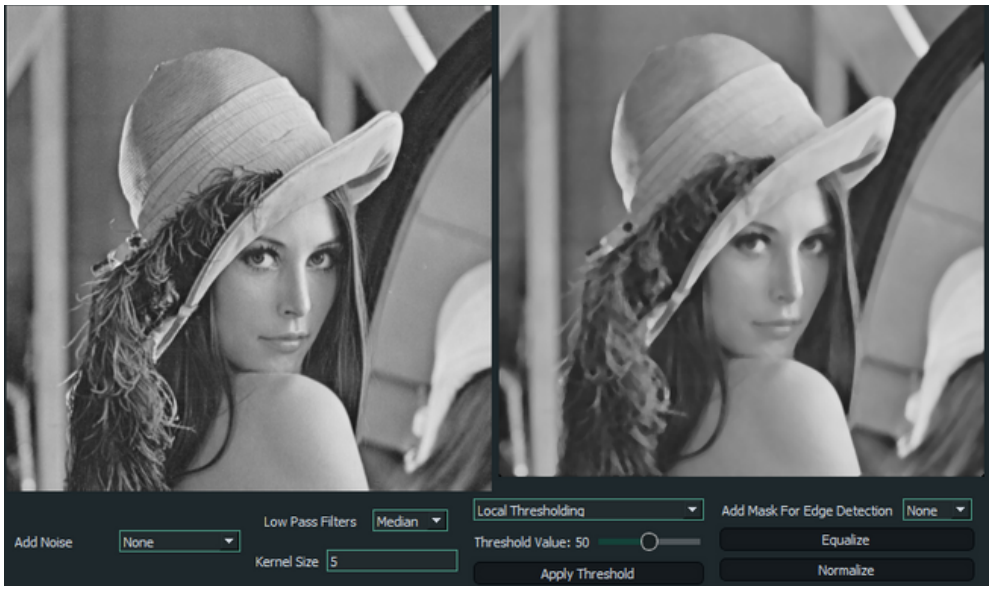
NOISE AND FILTERING

```python
def gaussian_kernel(self, kernel_size=(3, 3), sigma=1):
    # Ensure kernel size is odd
    if kernel_size[0] % 2 == 0 or kernel_size[1] % 2 == 0:
        raise ValueError("Kernel size must be odd")

    # Calculate center of the kernel
    center_x = kernel_size[0] // 2
    center_y = kernel_size[1] // 2

    # Generate grid of indices
    x = np.arange(-center_x, center_x + 1)
    y = np.arange(-center_y, center_y + 1)
    xx, yy = np.meshgrid( *xi: x, y)

    # Calculate Gaussian kernel values
    kernel = np.exp(-(xx ** 2 + yy ** 2) / (2 * sigma ** 2))
    kernel /= np.sum(kernel)
```

Observation: The Gaussian filter effectively reduces Gaussian noise while preserving image details and edges. By adjusting the standard deviation parameter, users can control the amount of smoothing applied to the image, striking a balance between noise reduction and preservation of image features.

Median Filter:

- **Explanation:** The median filter is a nonlinear filtering technique used for noise reduction in images. It replaces each pixel value with the median value of the pixel values within a defined kernel region, effectively removing outliers and impulsive noise while preserving image details and edges.



NOISE AND FILTERING

Median Filter:

- **Algorithm:** The median_filter function iterates over each pixel in the image and replaces its value with the median value of the pixel intensities within the kernel region centered around the pixel. This process effectively removes outliers or extreme values from the image, resulting in noise reduction.

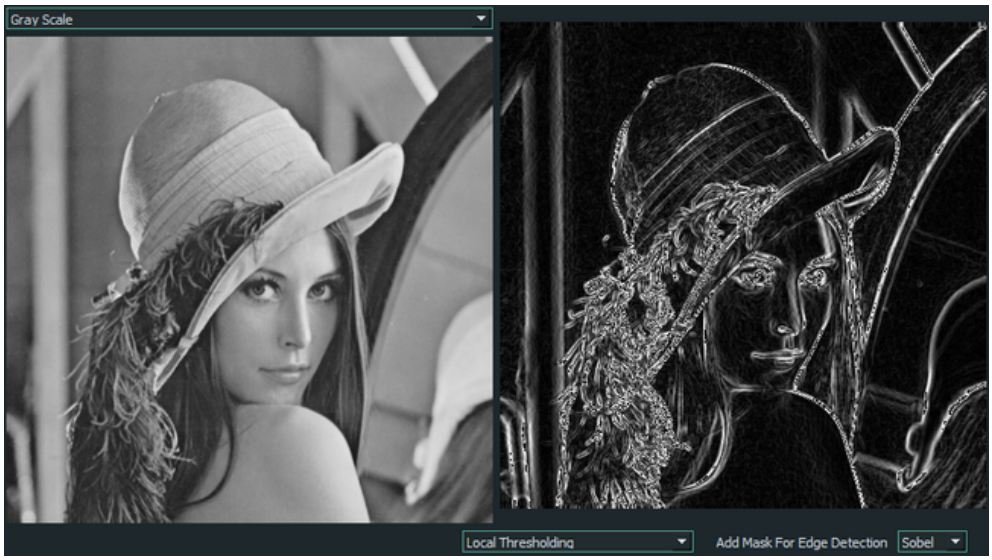
```
def median_filter(self, image, kernel_size=(3, 3)):
    # Get image dimensions
    height, width = image.shape[:2]
    # Get kernel dimensions
    kernel_height, kernel_width = kernel_size
    # Create an empty output image
    output_image = np.zeros_like(image)

    # Calculate border size for the kernel
    border_height = kernel_height // 2
    border_width = kernel_width // 2

    # Pad the image with zeros to handle border cases
    padded_image = np.pad(image, pad_width: ((border_height, border_height), (border_width, border_width)), mode='constar

    # Perform median filtering
    for i in range(border_height, height + border_height):
        for j in range(border_width, width + border_width):
            # Extract the region around the current pixel
            region = padded_image[i - border_height:i + border_height + 1, j - border_width:j + border_width + 1]

            # Compute the median value of pixel intensities in the region
            median_value = np.median(region)

            # Assign the median value to the corresponding pixel in the output image
            output_image[i - border_height, j - border_width] = median_value
    return output_image
```

- **Observation:** The median filter is particularly effective in removing salt-and-pepper noise, as it replaces noisy pixels with more accurate median values from the surrounding area. However, it may not be as effective for reducing Gaussian noise or preserving fine image details compared to other smoothing filters.

EDGE DETECTION

1-Sobel edge detection

- Preprocessing: Normalization stabilizes pixel values, ensuring numerical stability during convolution.
- Kernel Application: Convolution with Sobel kernels captures horizontal and vertical edge information.
- Gradient Magnitude Computation: Gradient magnitude computation provides edge strength at each pixel location.
- Image Reconstruction: Reconstruction restores the image to its original range, preserving visual fidelity.



fig(1)

Figure 1 displays the result of the Sobel edge detector, where the edges in the photo are highlighted in white against a black background. This outcome is achieved by computing the magnitude of gradients obtained from both horizontal and vertical Sobel kernels. The horizontal kernel has the following values:

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

Similarly, the vertical kernel is:

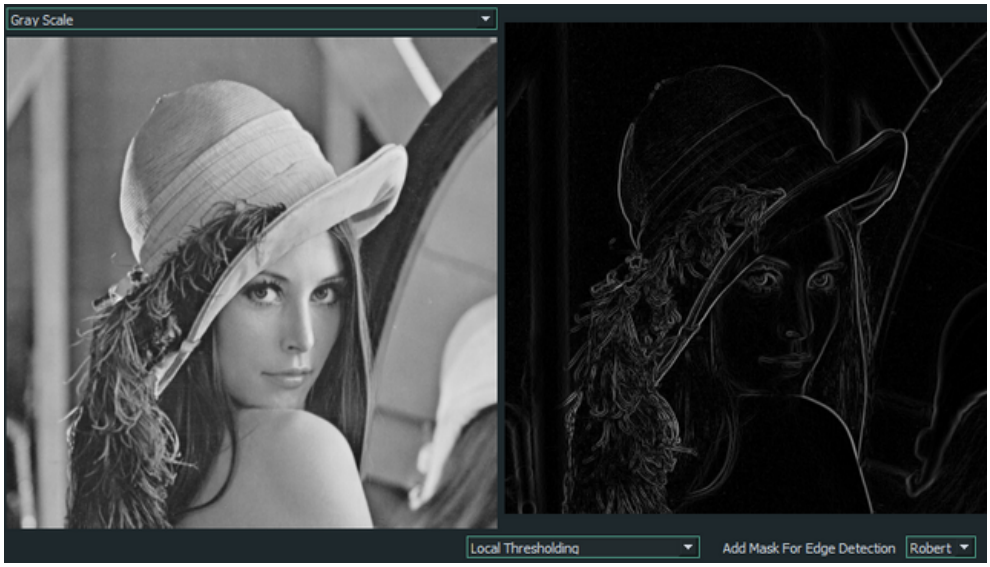
[-1, -2, -1
  0, 0, 0
 1, 2, 1]

The magnitude of gradients is calculated by taking the square root of the sum of squares of the gradient responses obtained from convolving the image with both horizontal and vertical kernels. This process effectively captures edges in both horizontal and vertical orientations, resulting in the highlighted edges seen in Figure 1.

EDGE DETECTION

2-Roberts edge detection

- Preprocessing:
- Standardize pixel values to ensure stability during convolution.
- Kernel Application:
- Apply 2x2 convolution kernels for diagonal edge detection.
- Gradient Magnitude Computation:
- Calculate gradient magnitude using both horizontal and vertical responses.
- Image Reconstruction:
- Adjust values to restore the original intensity range.



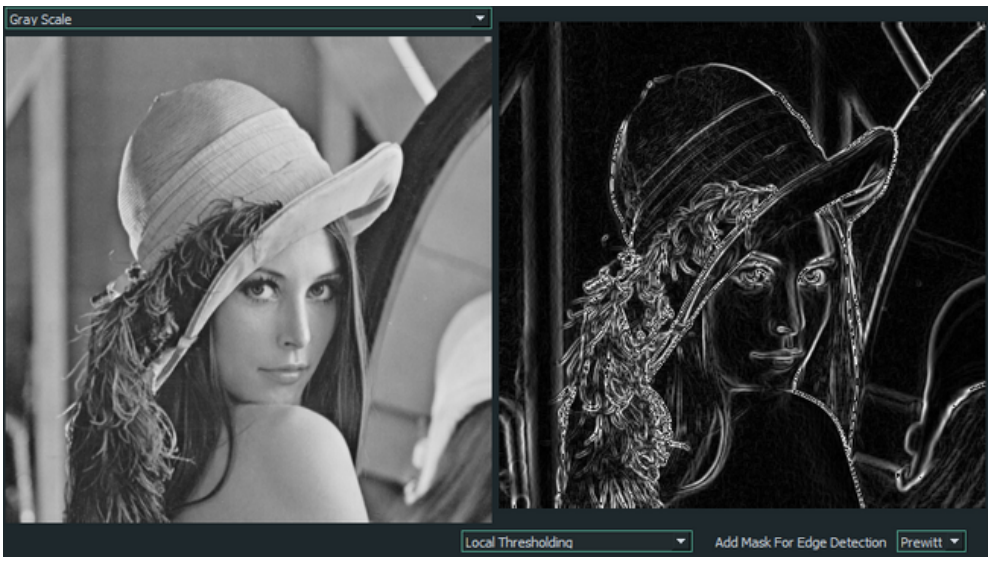
fig(2)

Figure 2 illustrates the outcome of the Roberts edge detector, showcasing edges in the image highlighted in white against a black backdrop. This result is attained by computing the gradient magnitude derived from convolving the image with two 2x2 convolution kernels: one for detecting diagonal edges from the top-left to bottom-right, and the other for detecting edges from the top-right to bottom-left. The diagonal kernels are as follows:
Diagonal (top-left to bottom-right) Kernel:
1 0
0 -1
Diagonal (top-right to bottom-left) Kernel:
0 1
-1 0
The gradient magnitude is computed by calculating the square root of the sum of squares of the gradient responses obtained from both diagonal kernels. This process effectively captures diagonal edges in the image, resulting in the highlighted edges depicted in Figure 2.

EDGE DETECTION

3-Prewitt edge detection

- Preprocessing: Normalization stabilizes pixel values, ensuring numerical stability during convolution.
- Kernel Application: Convolution with Prewitt kernels captures horizontal and vertical edge information.
- Gradient Magnitude Computation: Gradient magnitude computation provides edge strength at each pixel location.
- Image Reconstruction: Reconstruction restores the image to its original range, preserving visual fidelity.



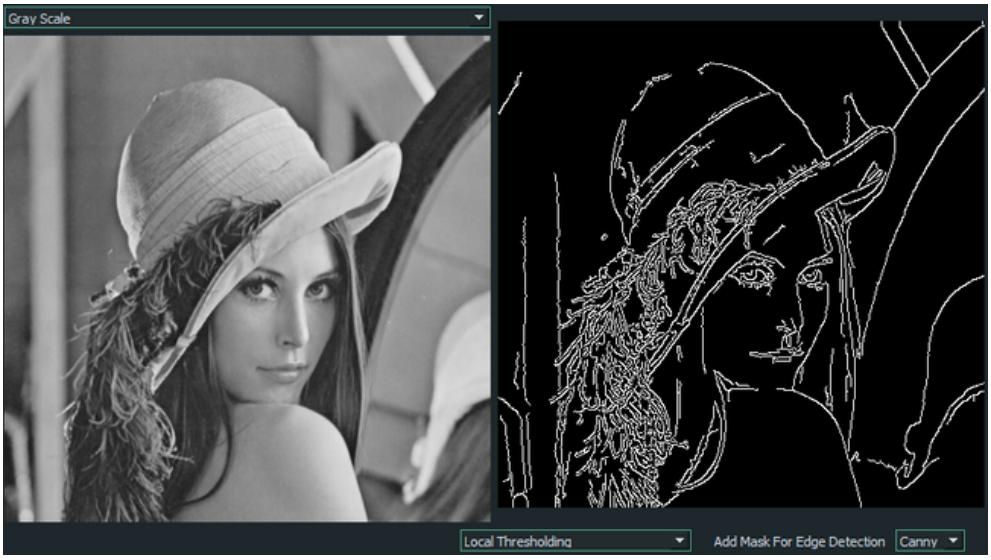
fig(3)

Figure 3 displays the result of the Prewitt edge detector, where the edges in the photo are highlighted in white against a black background. This outcome is achieved by computing the magnitude of gradients obtained from both horizontal and vertical Prewitt kernels. The horizontal kernel has the following values:
[-1, 0, 1
-1, 0, 1
-1, 0, 1]
Similarly, the vertical kernel is:
[-1, -1, -1
 0, 0, 0
 1, 1, 1]
The magnitude of gradients is calculated by taking the square root of the sum of squares of the gradient responses obtained from convolving the image with both horizontal and vertical kernels. This process effectively captures edges in both horizontal and vertical orientations, resulting in the highlighted edges seen in Figure 3. But it is less sensitive to the edges unlike Sobel edge detector

EDGE DETECTION

4-Canny edge detection

The built-in Canny edge detection function with a low threshold of 60 and a high threshold of 200 automatically identifies edges in an image. It smooths the image to reduce noise, calculates gradients, performs non-maximum suppression, and applies double thresholding to categorize pixels into strong, weak, and non-edge pixels. Finally, it tracks edges by hysteresis, ensuring accurate edge detection. This approach yields clean and reliable edge detection results, essential for various image processing applications.



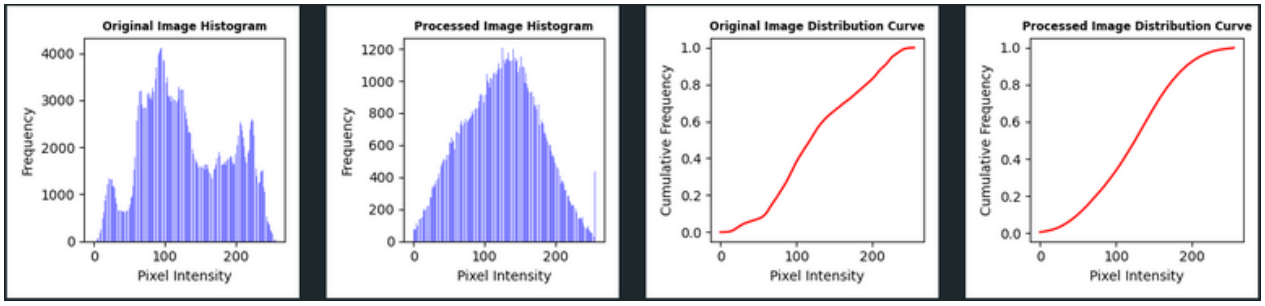
fig(4)

In Figure 4, the Canny edge detection algorithm, set with thresholds of 60 (low) and 200 (high), primarily highlights sharp, straight edges. This sensitivity to gradients within the specified range helps filter out noise and detect prominent edges, resulting in the detection of distinct, straight features in the image.

HISTOGRAM & DISTRIBUTION CURVES

Histograms and CDF

- **Explanation:** In image processing, a histogram represents the frequency distribution of pixel intensities in an image, providing insights into the image's contrast, brightness, and intensity distribution. Each pixel's intensity, for grayscale images, ranges from 0 (black) to 255 (white), and the histogram plots the frequency of each of these intensity levels. Building on the histogram, the Cumulative Distribution Function (CDF) offers a cumulative probability measure of pixel intensities, effectively showing the proportion of pixels with intensity values less than or equal to any given value. The transformation from histogram to CDF involves the cumulative summation of the normalized histogram values, ranging from 0 to 1. Together, these tools play a crucial role in image analysis, allowing for sophisticated manipulation and enhancement techniques by providing a detailed understanding of an image's intensity landscape.



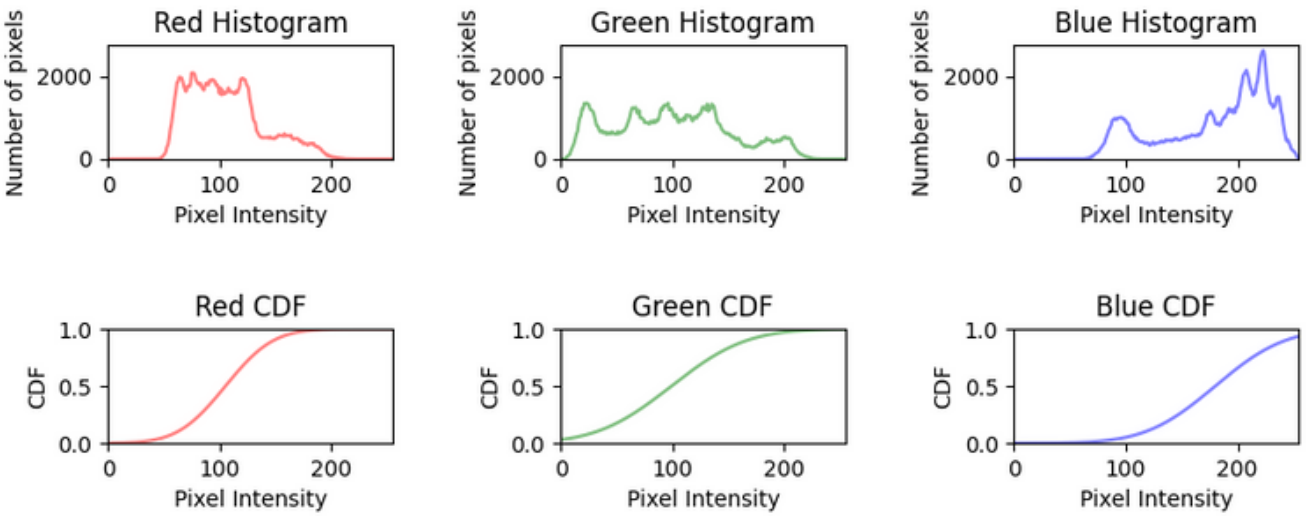
- **Approach:** The plot_gray_histogram function computes and plots the grayscale histogram of an input image on a specified label, while the plot_gray_distribution_curve function computes and plots the grayscale cumulative distribution curve (CDF) . Both functions first compute the histogram using the compute_gray_histogram method, then plot either the histogram as a bar graph or the CDF as a curve. Finally, the plots are converted to Pixmaps and displayed on the label after layout adjustments.
- **Observation:** From histograms and CDF curves of original and processed images provide crucial insights into image characteristics and alterations. Histograms reveal changes in intensity distribution, contrast, and brightness between the original and processed images. Peaks and valleys in histograms aid segmentation, while sudden spikes may indicate visual artifacts. CDF curves highlight shifts in cumulative probability distributions, indicating changes in image dynamics such as stretching or compression of intensity ranges. Evaluating the overall shape and smoothness of histograms and CDF curves offers qualitative measures of image quality and the effectiveness of normalization techniques.

RGB HISTOGRAMS

RGB Histograms & CDF

- **RGB Histograms:** RGB histograms are constructed by separating an image into its red, green, and blue color channels and computing the frequency of pixel intensities within each channel. This method provides valuable insights into the color composition of an image, allowing for the analysis of color balance, the presence of dominant colors, and potential color casts. By plotting the histograms, with pixel intensity on the x-axis and frequency of occurrence on the y-axis, the distribution of colors can be visualized and interpreted.

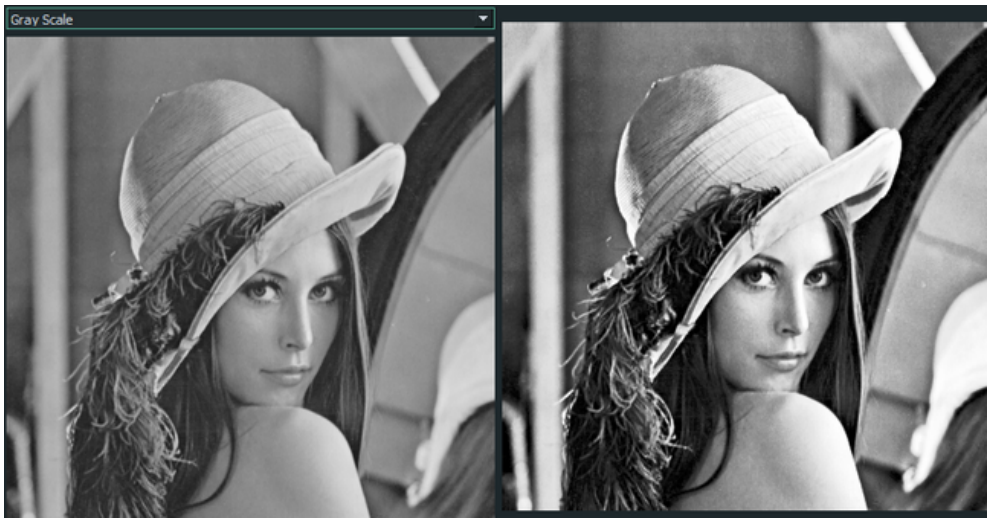
- **The RGB Cumulative Distribution Function (CDF):** is derived from the RGB histograms of an image, where each color channel (red, green, and blue) is individually processed to compute its respective histogram and CDF. The CDF represents the cumulative distribution of pixel intensities across each color channel, indicating the proportion of pixels with intensity values equal to or less than a given value. By plotting the RGB CDF, with pixel intensity on the x-axis and cumulative frequency on the y-axis, the overall distribution of pixel intensities in each color channel can be visualized. This information is valuable for understanding the tonal distribution and dynamic range of colors present in an image, aiding in color correction, contrast adjustment, and image enhancement tasks.

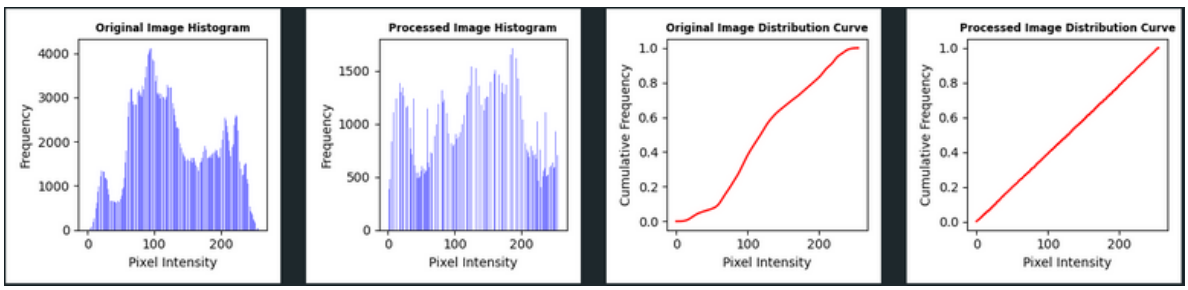


EQUALIZATION & NORMALIZATION

Equalization

- **Explanation:** Equalization is a technique used to enhance the contrast and dynamic range of an image by redistributing the pixel intensities. It works by mapping the cumulative distribution function (CDF) of the image histogram to a new histogram with a uniform distribution of pixel intensities. This process effectively stretches the intensity range of the image, enhancing the visibility of details and improving overall image quality.



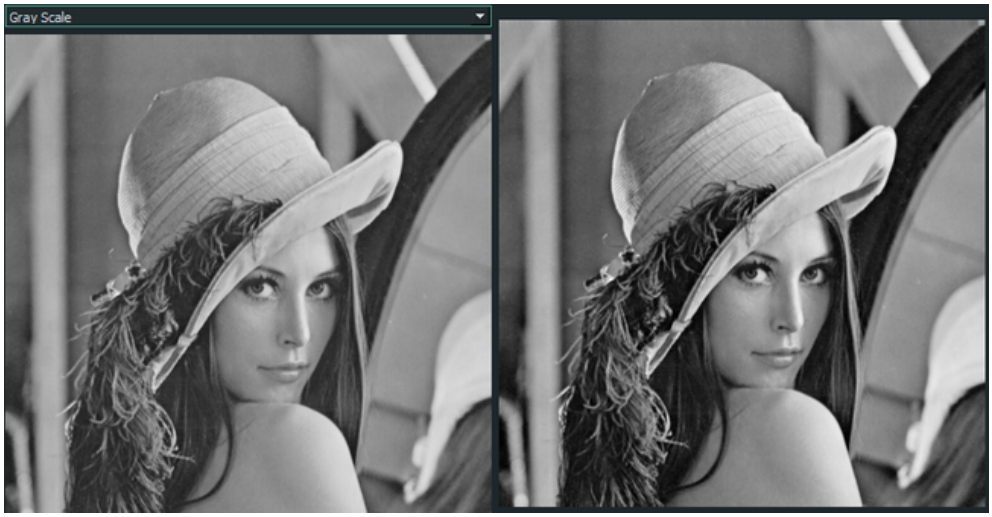


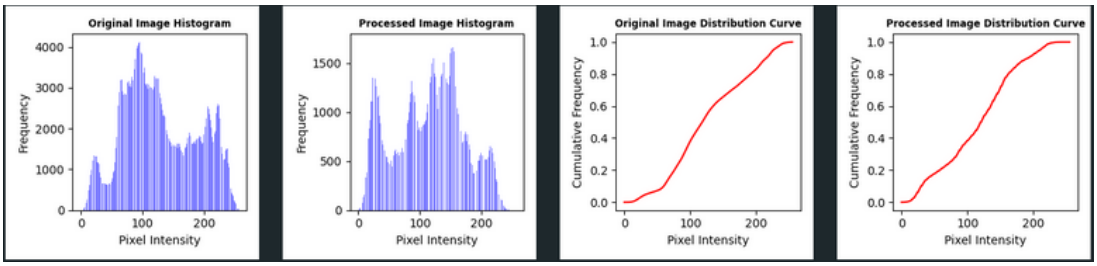
- **Approach:** The image_equalization function computes the histogram of the input image and then calculates the cumulative distribution function (CDF) of the histogram. It then normalizes the CDF to the range [0, 255] and maps each pixel intensity in the original image to its corresponding value in the equalized histogram, resulting in an equalized image.
- **Observation:** Equalization is very effective in enhancing the contrast of images with low dynamic range or uneven distribution of pixel intensities. However, it may lead to over-amplification of noise in regions with low contrast, resulting in a noisy appearance in some cases.

EQUALIZATION & NORMALIZATION

Normalization

- **Explanation:** Normalization is a preprocessing step used to scale the pixel intensities of an image to a specific range, typically [0, 1] or [0, 255]. It ensures that the pixel values fall within a consistent range, making it easier to compare and process images with varying intensity distributions.





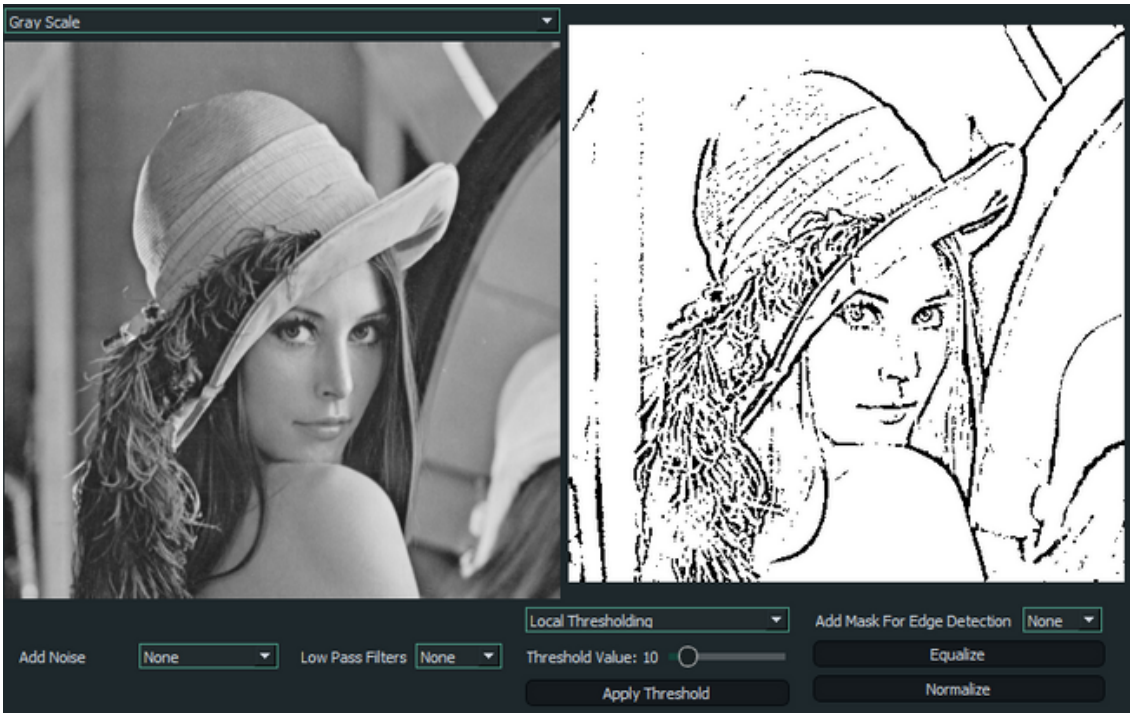
- **Approach:** The normalization process involves dividing each pixel intensity in the image by the maximum intensity value (255 for an 8-bit grayscale image) to scale the intensities to the range [0, 1]. This ensures that the pixel values are represented as floating-point numbers between 0 and 1, facilitating subsequent image processing operations.

- **Observation:** Normalization is a fundamental preprocessing step in image processing, commonly used to ensure consistency in pixel intensity ranges across different images. It enables effective comparison and analysis of images and helps mitigate issues related to varying illumination conditions or sensor characteristics.

LOCAL & GLOBAL THRESHOLDING

Local Thresholding

- **Explanation:** Local thresholding adapts the binarization process to local intensity characteristics by computing a threshold for each pixel based on its neighborhood. The threshold is determined by the mean intensity of a local window centered at the pixel. If the pixel intensity exceeds the threshold, it is set to white (255); otherwise, it is set to black (0).



- **Approach:** Iterate over each pixel:

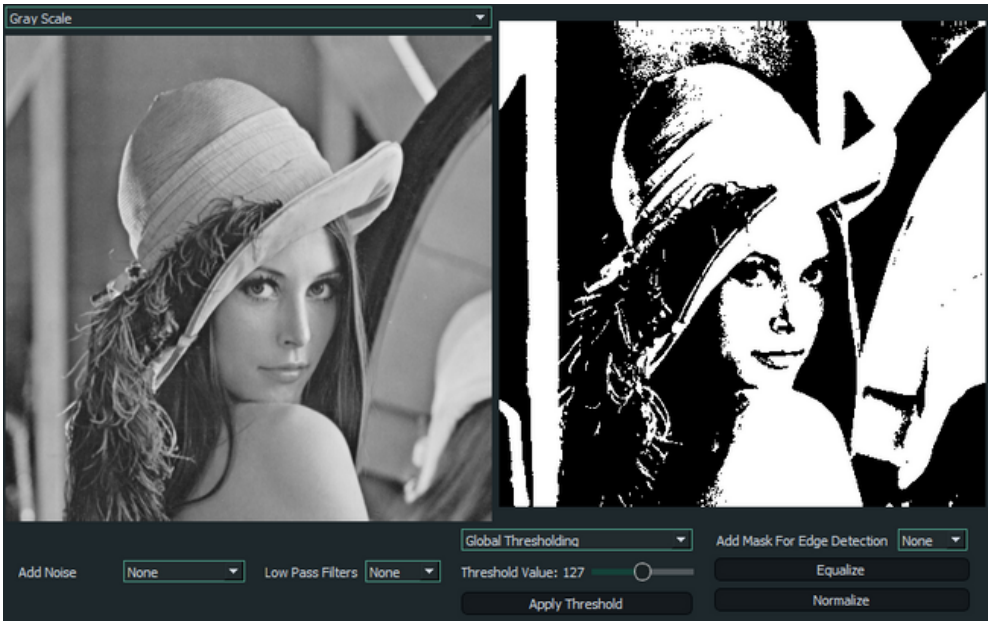
1. Extract a local window around the pixel.
2. Calculate the mean intensity of the window.
3. Apply a threshold to the pixel based on the mean intensity plus a margin.
4. Set the pixel value to 255 if it exceeds the threshold; otherwise, set it to 0.

- **Observation:** Local thresholding is effective for images with varying illumination or intensity gradients, producing accurate binarization results for different regions. It addresses issues of uniformity in global thresholding methods, enhancing performance in applications like document processing.

LOCAL & GLOBAL THRESHOLDING

Global Thresholding

- **Explanation:** Global thresholding applies a single threshold value to all pixels in the image, determining whether each pixel is set to white (255) or black (0) based on its intensity compared to the threshold. This method simplifies binarization but may not handle variations in illumination or intensity gradients.



- **Algorithm:** Iterate over each pixel:

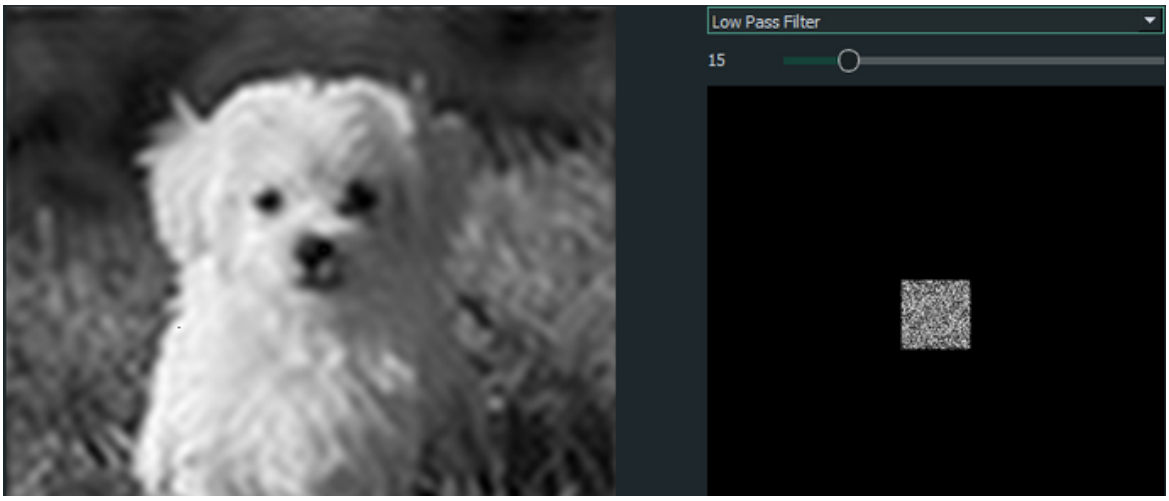
1. Compare the pixel intensity to a predefined threshold.
2. Set the pixel value to 255 if it exceeds the threshold; otherwise, set it to 0.

- **Observation:** Global thresholding simplifies images into binary representations based on a fixed threshold value. While effective for images with relatively uniform lighting conditions, it may struggle with scenes featuring varying illumination levels or intensity gradients. In such scenarios, local thresholding methods, which adjust thresholds based on local pixel neighborhoods, offer improved segmentation accuracy.

FREQUENCY DOMAIN FILTERS

Low Pass Filter

- **Explanation**: The low-pass filter attenuates high-frequency components in an image while allowing low-frequency components to pass through. This results in a smoothing effect on the image.



- **Approach**:

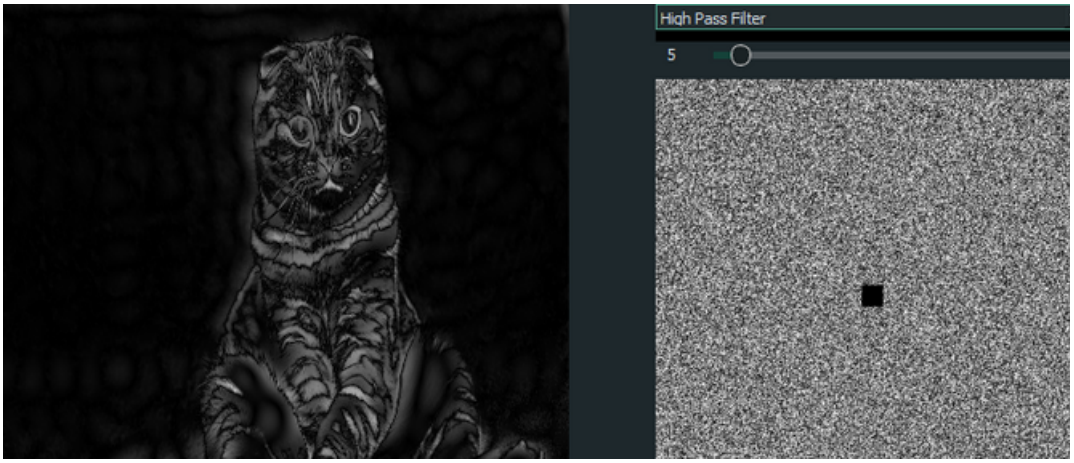
1. Convert the input image to grayscale.
2. Apply Fourier transform to the grayscale image.
3. Shift the zero-frequency component to the center.
4. Create a mask for the low-pass filter, where the central portion (low frequencies) is preserved, and the outer portion (high frequencies) is attenuated.
5. Apply the mask to the Fourier transform.
6. Perform inverse Fourier transform to obtain the filtered image.

- **Observation**: Applying a low-pass filter results in a smoother image with reduced high-frequency noise. It effectively blurs the image and reduces details.

FREQUENCY DOMAIN FILTERS

High Pass Filter

Explanation: The high-pass filter attenuates low-frequency components in an image while allowing high-frequency components to pass through. This enhances the edges and fine details in the image.



Approach:

1. Convert the input image to grayscale.
2. Apply Fourier transform to the grayscale image.
3. Shift the zero-frequency component to the center.
4. Create a mask for the high-pass filter, where the central portion (low frequencies) is attenuated, and the outer portion (high frequencies) is preserved.
5. Apply the mask to the Fourier transform.
6. Perform inverse Fourier transform to obtain the filtered image.

Observation: Applying a high-pass filter enhances the edges and details in the image, making them more prominent. It can result in increased noise and artifacts in regions with low-frequency content.

MIXING HYBRID IMAGES

Explanation:
Mixing images involves applying Fourier transforms to each image, applying a high-pass filter to one and a low-pass filter to the other, combining the filtered images, and then performing an inverse Fourier transform to obtain the result. Finally, display the resulting image.

Appraoch:

1. Apply Fourier transform to each input image.
2. Apply a high-pass filter to one Fourier-transformed image.
3. Apply a low-pass filter to the other Fourier-transformed image.
4. Combine the filtered images.
5. Perform inverse Fourier transform on the combined image.
6. Display the resulting enhanced image.

Observation: Mixing images creates a composite image that gives the impression of two photos merged into one. The dominant color tones are derived from the low-pass filtered image, while the edges and finer details are preserved from the high-pass filtered image.