

Computer Vision

Team 6

Task 1

Name	BN
Enjy Ashraf	9
Habiba Alaa Eldin	19
Shahd Ahmed Mahmoud	31
Mostafa Ali	65

Under Supervision of:

Dr. Ahmed Badawy

Eng. Yara Wael & Eng. Omar Hesham

Table of Contents

Introduction.....	3
Algorithms Implemented in task:.....	3
1. Add Noise:	3
2. Image Filtering:.....	6
3. Edge Detection:.....	12
4. Thresholding	16
5. Histogram :	18
6. Hybrid Mode:.....	20
Comparison with Built-In Implementation	21
1. Comparson of Filtering:	21
2. Comparison of Edge Detection:	22
3. Thresholding:	23
4. Comparison of Calculating Histograms :	23
5. Comparison of Equalization :	25
Screenshots from our application:	27
1. Main Image:	27
2. Histograms:	27
3. Hybrid Image Mode:	28
References	28

Introduction

This report explores essential image processing techniques in computer vision, including image filtering, edge detection, thresholding, and histogram equalization. It presents the mathematical concepts, pseudocode implementations, and performance comparisons of various methods such as Average, Median, Gaussian filters, Sobel, Roberts, Prewitt, Canny detectors, and adaptive thresholding. The document aims to concisely understand these techniques and their role in enhancing image quality and feature extraction.

Algorithms Implemented in task:

1. Add Noise:

Noise is an unwanted variation in image intensity, often degrading image quality and making further image processing tasks more challenging. Noise can arise from various sources such as image acquisition devices, transmission errors, or environmental factors. Understanding different types of noise is essential for selecting appropriate noise removal techniques. This section discusses three common types of noise: Salt-and-Pepper Noise, Uniform Noise, and Gaussian Noise.

1.1. Salt & Pepper Noise:

Salt-and-Pepper noise appears as random black and white pixels scattered across an image, caused by sudden signal disturbances or transmission errors. It creates sharp intensity spikes, making the image look speckled.

Mathematical Formulation:

$$P(x) = \begin{cases} p & \text{for } I(x) = 0 \text{ (black pixel)} \\ q & \text{for } I(x) = 1 \text{ (white pixel)} \\ 0 & \text{otherwise} \end{cases}$$

Where p and q represent the probabilities of black and white pixels respectively, and $I(x)$ is the intensity at pixel x .

Pseudocode of implementation:

```
Function apply_salt_and_pepper_noise(salt_ratio, pepper_ratio):  
    image = Get current image from output_image_viewer  
    pixels = Total number of pixels in the image (width * height)  
    no_of_salts = pixels * salt_ratio  
    no_of_pepper = pixels * pepper_ratio  
    # Add Salt Noise (White Pixels)  
    For i in range(no_of_salts):  
        x = Random integer between 0 and image width  
        y = Random integer between 0 and image height  
        Set pixel at (x, y) to 255 (White)
```

```

# Add Pepper Noise (Black Pixels)
For i in range(no_of_pepper):
    x = Random integer between 0 and image width
    y = Random integer between 0 and image height
    Set pixel at (x, y) to 0 (Black)
Return image

```

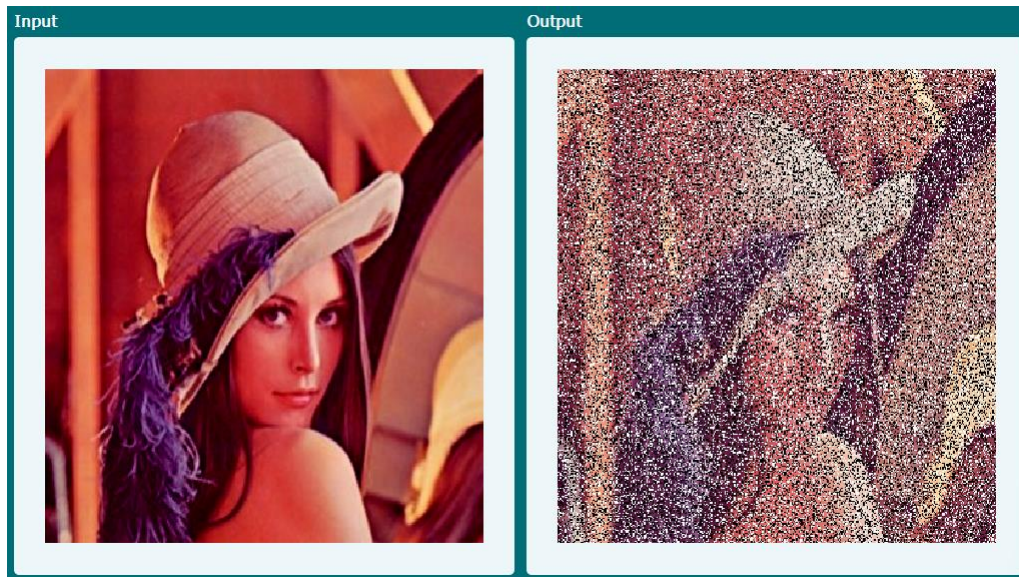


Figure 1: Output of salt & pepper noise with salt ratio 0.1 and pepper ratio 0.1

1.2. Uniform Noise:

Uniform noise is a type of noise where pixel intensity variations are randomly distributed within a fixed range. It is often caused by quantization errors or electronic device interference. The noise values are equally likely to occur, making the distribution flat across the range.

Pseudocode of implementation:

```

Function apply_uniform_noise(noise_intensity):
    image = Get current image from output_image_viewer
    # Generate random noise matrix with uniform distribution
    uniform_noise = Random numbers between -noise_intensity and +noise_intensity
                    with the same dimensions as the image
    # Add noise to the original image
    modified_image = Add image pixels with corresponding noise values
    # Ensure pixel values are within valid range [0, 255]
    Clip modified_image values to stay between 0 and 255
    Set image.modified_image = modified_image
    Return image

```

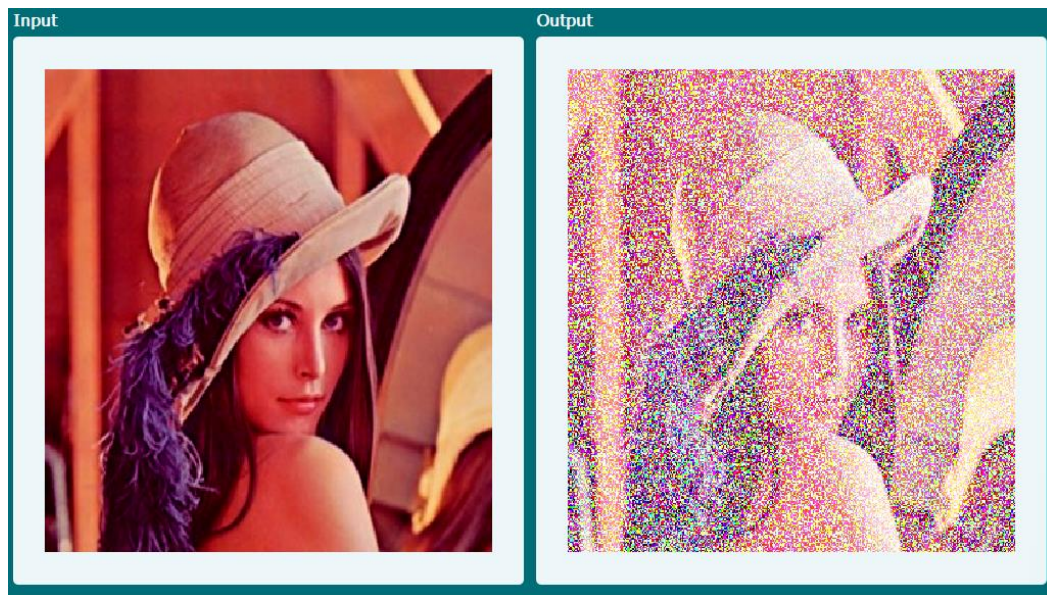


Figure 2: Output of uniform noise with noise intensity 40

1.3. Gaussian Noise:

Gaussian noise is a type of noise where pixel intensity variations follow a normal distribution, commonly caused by electronic circuit noise or low-light conditions. It creates random variations in pixel values, making the image appear grainy.

Pseudocode of implementation:

```
Function apply_gaussian_noise(mean, standard_deviation):
    image = Get current image from output_image_viewer
    // Generate Gaussian noise matrix with normal distribution
    gaussian_noise = Random numbers with mean and standard_deviation
                        having the same dimensions as the image
    // Add noise to the original image
    modified_image = Add image pixels with corresponding noise values
    // Ensure pixel values are within valid range [0, 255]
    Clip modified_image values to stay between 0 and 255
    Set image.modified_image = modified_image
    Return image
```

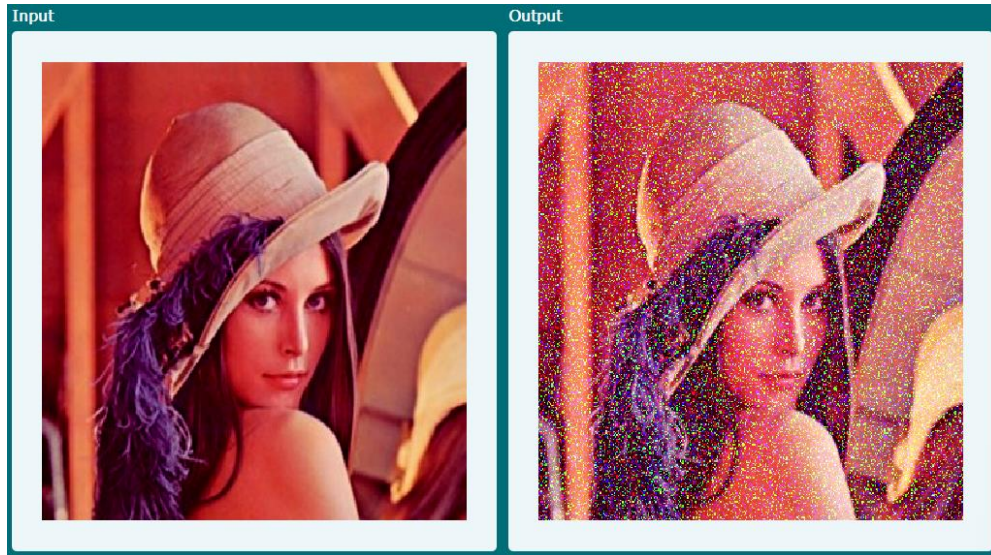



Figure 3: Output of gaussian noise with mean 10 and standard deviation 10

2. Image Filtering:

Image filtering is a crucial technique in image processing, used to enhance, smooth, or sharpen images by applying various mathematical operations. Filtering helps in noise reduction, edge detection, and feature extraction. This section discusses three primary filtering techniques: Average Filtering, Median Filtering, and Gaussian Filtering.

2.1. Average Filter:

Average filtering is a smoothing technique that reduces noise and blurs the image by replacing each pixel with the average value of its neighborhood. This technique is particularly useful for removing high-frequency noise while preserving general image structures.

Mathematical Formulation

$(i-1, j-1)$	$(i-1, j)$	$(i-1, j+1)$
$(i, j-1)$	(i, j)	$(i, j+1)$
$(i+1, j-1)$	$(i+1, j)$	$(i+1, j+1)$

A 9×9 Convolutional Kernel

Given a 9 * 9 kernel, every pixel in the image, except for those on the edges, is surrounded by **eight** pixels. The average filter recalculates each pixel's value by taking the average of these **nine** pixels (including the center one).

$$Pixel(i, j) = data[i * Width + j], \quad i, j \geq 1, i < Height - 1, \text{ and } j < Weight - 1.$$

$$newPixel(i, j) = \sum_{x=-1}^1 \sum_{y=-1}^1 originalPixel(i + x, j + y) / 9 = \sum_{x=-1}^1 \sum_{y=-1}^1 data[(i + x) * Width + (j + y)] / 9$$

Pseudocode of the implementation

```
FUNCTION apply_average_filter(modified_image, kernel_size):  
    INPUT:  
        - modified_image (grayscale image)  
        - kernel_size (size of the averaging kernel)  
    OUTPUT: filtered_image (smoothed image)  
    filtered_image = CREATE_EMPTY_IMAGE(height, width)  
    kernel = CREATE_KERNEL(kernel_size, kernel_size, value=1/kernel_size^2)  
    # Apply the filter to each pixel  
    FOR row IN range(height):  
        FOR col IN range(width):  
            region = EXTRACT_REGION(modified_image, row, col, kernel_size)  
            filtered_image[row, col] = SUM(region * kernel)  
    RETURN filtered_image
```

2.2. Median Filtering:

Median filtering is a non-linear technique that replaces each pixel's value with the median of its neighborhood. It is particularly effective for reducing salt-and-pepper noise while preserving edges.

Mathematical Formulation

Similar to the average filter, but each pixel's value is replaced with the median of the nine-grid pixels, rather than the average. The values of the nine surrounding pixels are stored in an array, and sorting methods are used to find the median, which is then assigned to the pixel (i, j).

Pseudocode of the implementation

```
FUNCTION apply_median_filter(modified_image, kernel_size):  
    INPUT:  
        - modified_image (grayscale image)  
        - kernel_size (size of the local neighborhood)  
    OUTPUT: filtered_image (denoised image)  
    filtered_image = CREATE_EMPTY_IMAGE(height, width)  
    # Iterate over each pixel  
    FOR row IN range(height):  
        FOR col IN range(width):  
            region = EXTRACT_REGION(modified_image, row, col, kernel_size)  
            filtered_image[row, col] = MEDIAN(region)  
    RETURN filtered_image
```

2.3. Gaussian Filtering:

Gaussian filtering is a linear smoothing technique that applies a Gaussian function as a weight matrix to reduce noise while preserving image details. The Gaussian kernel gives higher importance to pixels near the center of the neighborhood.

Mathematical Formulation

The Gaussian function is central to Gaussian smoothing in image processing. It is defined as a 2-D isotropic distribution and is used as a point-spread function for convolution with an image.

Gaussian Function

The 2-D Gaussian function is given by:

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

σ : Standard deviation, controlling the spread of the Gaussian.

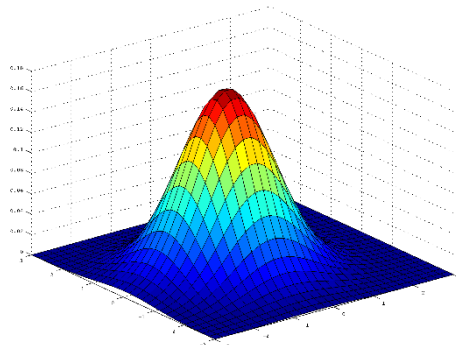


Figure 4: 2-D Gaussian distribution

The function is non-zero everywhere but effectively approaches zero beyond three standard deviations from the mean, allowing truncation for practical purposes.

Discrete Kernel Approximation

Since images are discrete, the Gaussian function must be approximated using a discrete convolution kernel. The kernel is created by integrating the Gaussian over each pixel, accounting for the non-linear variation of the Gaussian across pixels. The kernel values are normalized so that their sum equals a specific value (e.g., 273 for a 5x5 kernel)

	1	4	7	4	1
	4	16	26	16	4
$\frac{1}{273}$	7	26	41	26	7
	4	16	26	16	4
	1	4	7	4	1

Figure 5: Discrete approximation to Gaussian function with $\sigma=1.0$

Once a suitable kernel has been calculated, then the Gaussian smoothing can be performed using standard convolution methods. The convolution can in fact be performed fairly quickly since the equation for the 2-D isotropic Gaussian shown above is separable into x and y components. Thus the 2-D convolution can be performed by first convolving with a 1-D Gaussian in the x direction, and then convolving with another 1-D Gaussian in the y direction.

.006	.061	.242	.383	.242	.061	.006
------	------	------	------	------	------	------

Figure 6: One of the pair of 1-D convolution

Pseudocode of the implementation

```

FUNCTION apply_average_filter(modified_image, kernel_size):
    INPUT:
        - modified_image (grayscale image)
        - kernel_size (size of the averaging kernel)
    OUTPUT: filtered_image (smoothed image)
    filtered_image = CREATE_EMPTY_IMAGE(height, width)
    kernel = CREATE_KERNEL(kernel_size, kernel_size, value=1/kernel_size^2)
    # Apply the filter to each pixel
    FOR row IN range(height):
        FOR col IN range(width):
            region = EXTRACT_REGION(modified_image, row, col, kernel_size)
            filtered_image[row, col] = SUM(region * kernel)
    RETURN filtered_image

```

2.4. Low-Pass Filtering

Low-pass filtering removes high-frequency components, preserving smooth variations in an image while reducing sharp edges and noise.

Creating Low-Pass Filter Kernel

The goal is to preserve low-frequency components (smooth variations) and remove high-frequency details (sharp edges, noise). This is achieved by creating a binary mask with a circular passband centered at the middle of the Fourier spectrum. The radius of the passband is determined by the region factor, which scales the filtering region relative to the image size. All frequencies inside this circle are set to 1 (pass through), while everything outside is set to 0 (removed).

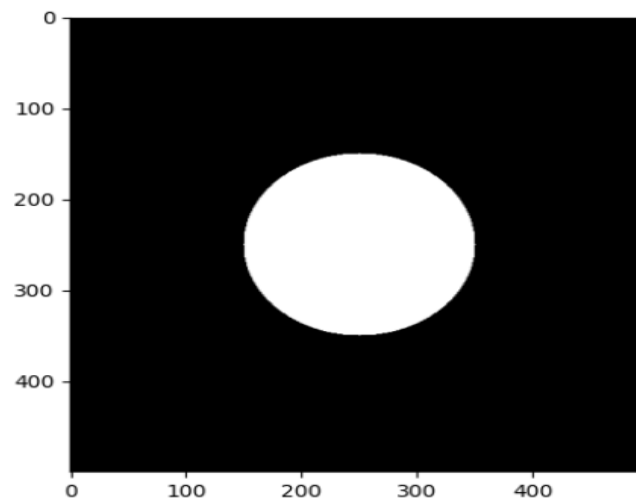


Figure 7: Low pass filter kernel with limiting factor = 0.3

Pseudocode of the implementation

```

FUNCTION apply_low_pass_filter(image, region_factor):
    INPUT:
        - image (Fourier-transformed image)
        - region_factor (controls filter size)
    OUTPUT: filtered_image (low-pass filtered image)
    mask = CREATE_FOURIER_MASK(region_factor, low_pass=True)
    filtered_fourier = image * mask
    filtered_image = INVERSE_FOURIER_TRANSFORM(filtered_fourier)
    RETURN filtered_image
  
```



Figure 8: Low-pass filtering Image and result comparison

2.5. High-pass filtering:

High-pass filtering enhances high-frequency components, preserving edges and fine details while reducing gradual intensity variations.

Creating High-Pass Filter Kernel

A high-pass filter kernel aims to preserve high-frequency components (sharp details, edges) and remove low-frequency components (smooth variations). It works as the complement of the low-pass filter. Instead of keeping low frequencies, it retains high frequencies by subtracting the low-pass mask from 1. Thus, frequencies inside the low-pass region are set to zero (removed), while frequencies outside the low-pass region are set to one (preserved).

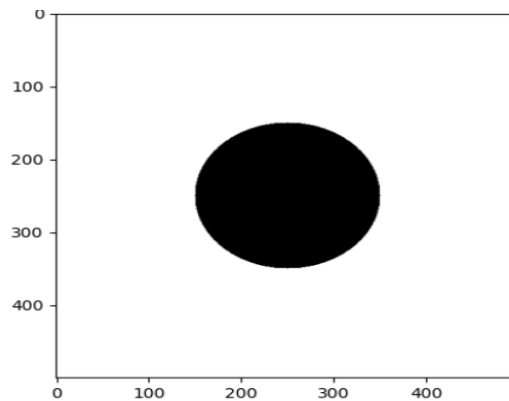


Figure 9: High pass filter kernel with limiting factor = 0.3

Pseudocode of the implementation

```
FUNCTION apply_average_filter(modified_image, kernel_size):  
    INPUT:  
        - modified_image (grayscale image)  
        - kernel_size (size of the averaging kernel)  
    OUTPUT: filtered_image (smoothed image)  
    filtered_image = CREATE_EMPTY_IMAGE(height, width)  
    kernel = CREATE_KERNEL(kernel_size, kernel_size, value=1/kernel_size^2)  
    # Apply the filter to each pixel  
    FOR row IN range(height):  
        FOR col IN range(width):  
            region = EXTRACT_REGION(modified_image, row, col, kernel_size)  
            filtered_image[row, col] = SUM(region * kernel)  
    RETURN filtered_image
```

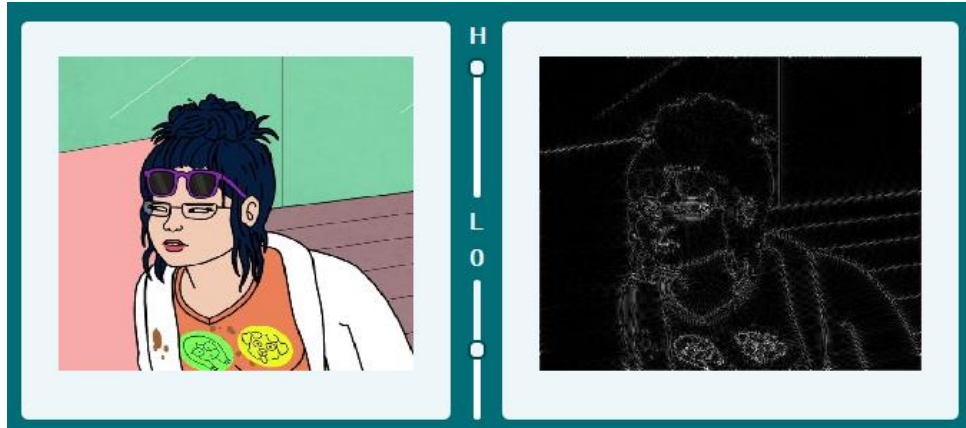


Figure 10: High-pass filtering Image and result comparison

3. Edge Detection:

Edge detection is a fundamental technique in image processing used to identify boundaries and sharp changes in intensity within an image. In this task we implemented several edge detection methods, including Sobel, Roberts, Prewitt, and Canny. Each method uses specific kernels and mathematical formulations to detect edges effectively.

3.1. Sobel Edge Detection:

The Sobel operator computes the gradient of the image intensity at each pixel, highlighting regions with high spatial frequency, which correspond to edges.

Mathematical Formulation

The Sobel edge enhancement filter has the advantage of providing differentiating (which gives the edge response) and smoothing (which reduces noise) concurrently, so it uses two 3x3 kernels, G_x and G_y to approximate the horizontal and vertical derivatives of the image

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

3.2. Roberts Edge Detection:

The Roberts operator is a simple and computationally efficient edge detection method that uses 2x2 kernels to approximate the gradient.

Mathematical Formulation

The Roberts edge enhancement filter also provides differentiating and smoothing properties but focuses on detecting edges using a pair of 2x2 kernels. These kernels, G_x and G_y , approximate the gradients in diagonal directions, which helps to identify the edges in an image more effectively.

+1	0
0	-1

Gx

0	+1
-1	0

Gy

3.3. Prewitt Edge Detection:

Prewitt operator is similar to the Sobel operator and is used for detecting vertical and horizontal edges in images. However, unlike the Sobel, this operator does not place any emphasis on the pixels that are closer to the center of the mask.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

Pseudocode of the implementation

```

FUNCTION detecting_process(Gx, Gy, detector_type):
    INPUT:
        - Gx: Gradient kernel for horizontal direction
        - Gy: Gradient kernel for vertical direction
        - detector_type: Type of edge detector (e.g., "Roberts", "Sobel",
"Prewitt")
    OUTPUT:
        - Updates the output image with the edge-detected result

    # Get dimensions of the input image
    image_height, image_width =
GET_IMAGE_DIMENSIONS(self.output_image_viewer.current_image.modified_image)

    # Initialize matrices to store horizontal and vertical gradients
    G_x = CREATE_ZEROS_MATRIX(image_height, image_width, dtype=FLOAT32)
    G_y = CREATE_ZEROS_MATRIX(image_height, image_width, dtype=FLOAT32)

    # Define padding size to handle edge pixels
    pad_size = 1

    # Pad the input image to handle boundary pixels during convolution
    padded_image =
PAD_IMAGE(self.output_image_viewer.current_image.modified_image,
padding=((pad_size, pad_size), (pad_size, pad_size)),
mode='reflect')

    # Apply convolution based on the detector type
    IF detector_type == "Roberts detector":

```

```

        # Loop over each pixel in the image (excluding padding)
        FOR i FROM pad_size TO image_height + pad_size:
            FOR j FROM pad_size TO image_width + pad_size:
                # Extract the local region around the pixel
                region = EXTRACT_REGION(padded_image, i - pad_size, i + pad_size,
j - pad_size, j + pad_size)

                # Compute horizontal and vertical gradients
                G_x[i - pad_size, j - pad_size] = SUM(region * Gx)
                G_y[i - pad_size, j - pad_size] = SUM(region * Gy)

            ELSE:
                # Loop over each pixel in the image (excluding padding)
                FOR i FROM pad_size TO image_height + pad_size:
                    FOR j FROM pad_size TO image_width + pad_size:
                        # Extract the local region around the pixel
                        region = EXTRACT_REGION(padded_image, i - pad_size, i + pad_size +
1, j - pad_size, j + pad_size + 1)

                        # Compute horizontal and vertical gradients
                        G_x[i - pad_size, j - pad_size] = SUM(region * Gx)
                        G_y[i - pad_size, j - pad_size] = SUM(region * Gy)

                    # Compute the gradient magnitude
                    gradient_magnitude = SQRT(G_x ** 2 + G_y ** 2)

                    # Normalize the gradient magnitude to the range [0, 255]
                    gradient_magnitude = (gradient_magnitude / MAX(gradient_magnitude)) * 255

                    # Convert the gradient magnitude to an 8-bit unsigned integer
                    gradient_magnitude = CONVERT_TO_UINT8(gradient_magnitude)

                    # Update the output image with the edge-detected result
                    self.output_image_viewer.current_image.modified_image = gradient_magnitude

```

3.4. Canny Operator:

The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. It was developed by John F. Canny in 1986. Canny also produced a computational theory of edge detection explaining why the technique works.

Mathematical Formulation

The Canny edge detection algorithm is a multi-step process designed to detect edges in an image with high accuracy. Below is a detailed mathematical formulation of each step

3.4.1. Noise Reduction

Edge detection is highly sensitive to noise because it relies on calculating derivatives of pixel intensities. To mitigate this, a Gaussian filter is applied to smooth the image. The Gaussian kernel is defined as:

$$H_{ij} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(i - (k+1))^2 + (j - (k+1))^2}{2\sigma^2}\right); 1 \leq i, j \leq (2k+1)$$

The equation for a Gaussian filter kernel of size $(2k+1) \times (2k+1)$

where σ is the standard deviation controlling the blur intensity. The kernel is convolved with the image to reduce noise while preserving edges. For example, a 5x5 Gaussian kernel with $\sigma=1.4$ is commonly used.

3.4.2. Gradient Calculation

The gradient of the image is computed to identify regions of intensity change, which correspond to edges. This is done using Sobel operators for horizontal G_x and vertical G_y derivatives.

$$K_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, K_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}.$$

3.4.3. Non-Maximum Suppression

Non-maximum suppression is applied to thin out the edges by retaining only the local maxima in the gradient magnitude matrix. For each pixel, the algorithm checks the neighboring pixels along the gradient direction. If the current pixel is not the maximum in its neighborhood, its value is set to zero. This ensures that only the strongest edges remain, resulting in thin, well-defined edges.

3.4.4 Double Threshold

In the double thresholding process, pixels are classified into three categories: strong edges, weak edges, and non-relevant pixels. Two thresholds, T_{high} and T_{low} , are used.

- Pixels with gradient magnitude $G \geq T_{high}$ are marked as strong edges.
- Pixels with gradient magnitude $T_{low} \leq G < T_{high}$ are marked as weak edges.
- Pixels with gradient magnitude $G < T_{low}$ are discarded as non-relevant.

3.4.5 Edge Tracking by Hysteresis

In the final step, weak edges are either promoted to strong edges or discarded based on their connectivity to strong edges. If a weak edge pixel is connected to a strong edge pixel (e.g., within its 8-neighborhood), it is retained as part of the edge. Otherwise, it is discarded. This ensures that only meaningful edges are preserved, resulting in a clean and accurate edge map.

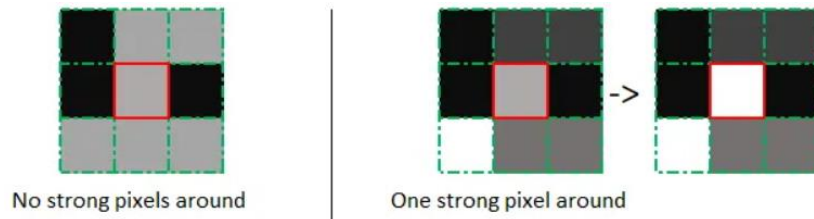


Figure 11: Canny operator Image and result comparison

4. Thresholding

Thresholding is a fundamental image processing technique used to convert an input image into a binary image. The process involves classifying each pixel in the image as either foreground or background based on a specified threshold value. This technique is widely used in applications such as object detection, segmentation, and feature extraction.

Methodology of Thresholding

The methodology implemented in this task involves two primary thresholding techniques: global thresholding and local (adaptive) thresholding. Each method is designed to address specific challenges in image processing, particularly in handling images with varying illumination conditions.

4.1. Global thresholding

Global thresholding is a straightforward technique where a single threshold value is applied to the entire image. Pixels with intensity values below the threshold are classified as background (set to 0), while pixels with intensity values above the threshold are classified as foreground (set to 255).

Mathematical Formulation

Given an input grayscale image and a global threshold value T , for each pixel (x,y) , the binary image $g(x,y)$ is computed as:

$$g(x,y) = \begin{cases} 1, & \text{if } f(x,y) < t, \\ 0, & \text{if } f(x,y) \geq t. \end{cases}$$

Pseudocode of the implementation

```
FUNCTION apply_global_thresholding(modified_image, global_threshold_val):
    INPUT:
        - modified_image (grayscale image)
        - global_threshold_val (threshold value)
    OUTPUT: thresholded_image (binary image)
    thresholded_image = CREATE_EMPTY_IMAGE(height, width)
    # Iterate over each pixel in the image
    FOR row IN range(height):
        FOR col IN range(width):
            # Apply the thresholding rule
            IF modified_image[row, col] < global_threshold_val:
                thresholded_image[row, col] = 0 # Background
            ELSE:
                thresholded_image[row, col] = 255 # Foreground
    RETURN thresholded_image
```

Problem with Global Thresholding

This method assumes that the image has a bimodal intensity distribution, making it suitable for images with uniform lighting conditions. However, global thresholding often fails when the image has non-uniform illumination or varying contrast. In such cases, a single threshold value cannot effectively separate the foreground from the background, leading to poor segmentation results.

4.2. Local (Adaptive) Thresholding

To address the limitations of global thresholding, local (adaptive) thresholding is used. This technique computes a unique threshold value for each pixel based on the intensity distribution of its local neighborhood. By adapting to local variations in illumination, this method is particularly effective for images with non-uniform lighting conditions.

The following comparison underscores the limitations of global thresholding in the presence of non-uniform illumination and the advantages of local thresholding for robust image segmentation.

(a) Original Image with Illumination Gradient

(b) Global Thresholding

(c) Local Thresholding

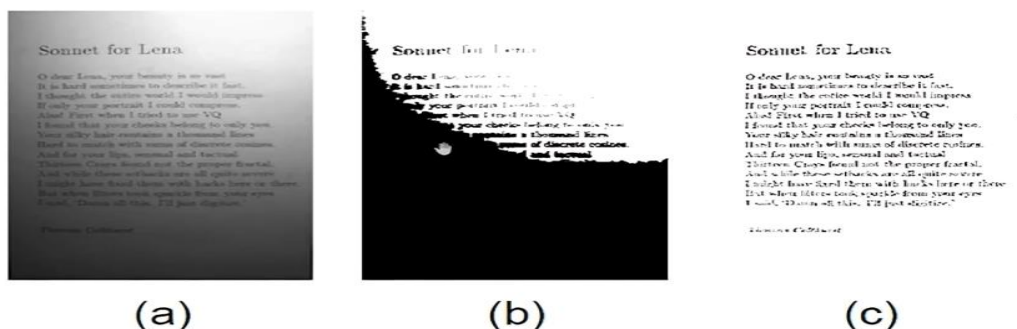


Figure 12: Comparison of Global and Local Thresholding on an Image with Illumination Gradient

Pseudocode of the implementation

```
FUNCTION apply_local_thresholding(modified_image, block_size, constant_C):  
    INPUT:  
        - modified_image (grayscale image)  
        - block_size (size of the local neighborhood)  
        - constant_C (constant subtracted from the mean)  
    OUTPUT: thresholded_image (binary image)  
    thresholded_image = CREATE_EMPTY_IMAGE(height, width)  
    # Iterate over each pixel in the image  
    FOR i IN range(height):  
        FOR j IN range(width):  
            # Define the boundaries of the local neighborhood  
            x_min = MAX(0, i - block_size // 2)  
            y_min = MAX(0, j - block_size // 2)  
  
            x_max = MIN(height - 1, i + block_size // 2)  
            y_max = MIN(width - 1, j + block_size // 2)  
            # Extract the local neighborhood  
            local_block = modified_image[x_min:x_max + 1, y_min:y_max + 1]  
            # Compute the local threshold as the mean intensity - constant  
            local_thresh = MEAN(local_block) - constant_C  
            # Apply the thresholding rule  
            IF modified_image[i, j] >= local_thresh:  
                thresholded_image[i, j] = 255 # Foreground  
            ELSE:  
                thresholded_image[i, j] = 0 # Background  
    RETURN thresholded_image
```

5. Histogram :

A histogram represents the frequency distribution of pixel intensity values in an image.

For a grayscale image with intensity levels in the range $[0, L - 1]$, the histogram $h(i)$ is defined as :

$$h(i) = \text{Number of pixels intensity } i \text{ for } i = 0, 1, 2, \dots, l - 1$$

For an RGB image, separate histograms are computed for each channel (Red, Green, Blue)

Pseudocode of the implementation

```
function calculate_histogram(image, histo_vector):  
    for each pixel in image:  
        intensity = pixel value  
        histo_vector[intensity] += 1
```

5.1. Probability Distribution Function (PDF) :

The PDF $p(i)$ represents the normalized histogram :

$$p(i) = \frac{h(i)}{N}$$

where N is the total number of pixels in the image.

5.2. Cumulative Distribution Function (CDF) :

The CDF $c(i)$ is the cumulative sum of the PDF:

$$c(i) = \sum_{j=0}^i p(j)$$

The CDF maps the original intensity values to new values in the range $[0,1]$

Psuedocode of implementation of PDF & CDF:

```
function compute_pdf_cdf(histo_vector, pdf_vector, cdf_vector, total_pixels):  
    cumulative_sum = 0  
    for i in range(0, 256):  
        pdf_vector[i] = histo_vector[i] / total_pixels  
        cumulative_sum += pdf_vector[i]  
        cdf_vector[i] = cumulative_sum
```

5.3. Histogram Equalization :

The goal of histogram equalization is to enhance contrast by redistributing the intensity values across the entire range.

The transformation function $T(i)$ is defined as:

$$T(i) = \text{round}(L - 1) \cdot c(i)$$

where L is the number of intensity levels (e.g., 256 for 8-bit images).

The new intensity value for each pixel is computed as:

$$I_{\text{new}}(x, y) = T(I_{\text{old}}(x, y))$$

This transformation ensures that the output image has a more uniform distribution of intensity values, enhancing contrast by stretching intensity values in regions with high pixel density. For instance, in dark images where intensity values are concentrated in the lower range, histogram equalization redistributes these values across the entire range, making details more visible. However, this method is not always suitable for color images, as it can distort the color balance.

Pseudocode of Implementation:

```
function equalize_image(image, cdf_vector):  
    if image is Rgb :  
        image = convert_to_grayscale(image)  
    for each pixel in image:  
        old_intensity = pixel value  
        new_intensity = round(255 * cdf_vector[old_intensity])  
        pixel value = new_intensity
```

6. Hybrid Mode:

Hybrid images is an average between two images, one high pass filtered and the other low pass filtered.

Here are some examples from our application:

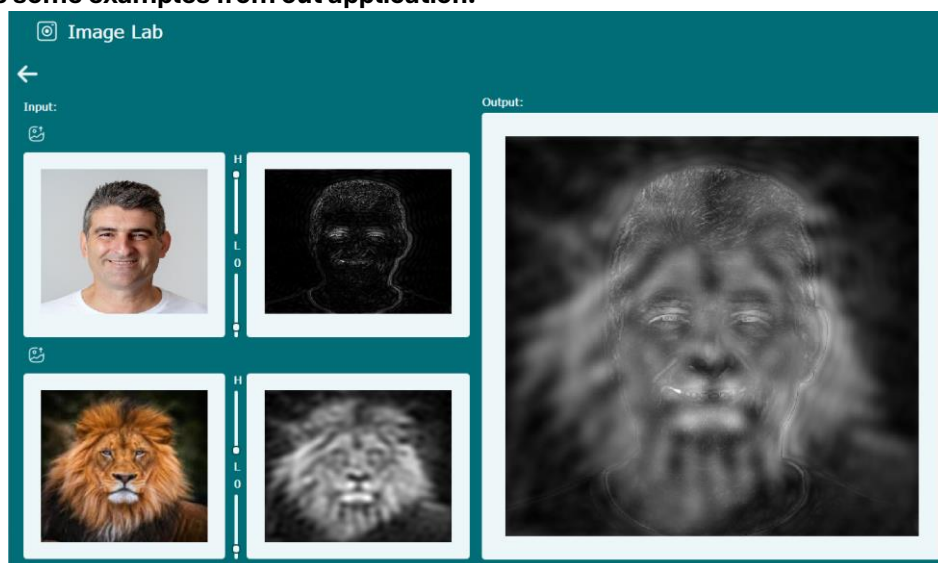
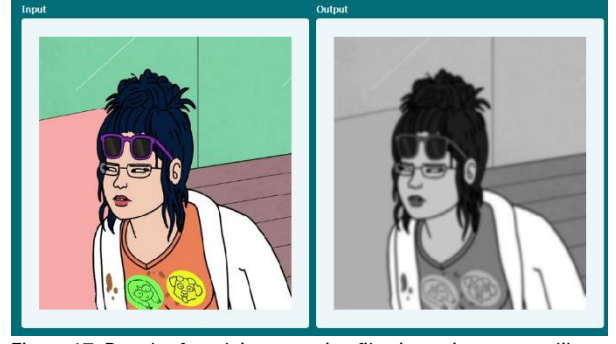



Figure 12: Hybrid image combining low and high-frequency components of two images using frequency domain filtering.

Comparison with Built-In Implementation









1. Comparson of Filtering:

	Built-in Function	Our Implementation
Average Filtering	 <p>Figure 13: Result of applying average filtering using opencv library</p>	 <p>Figure 14: Result of applying average filter using function from scratch</p>
Median Filtering	 <p>Figure 15: Result of applying median filtering using opencv library</p>	 <p>Figure 16: Result of applying median filter using function from scratch</p>
Gaussian Filtering	 <p>Figure 17: Result of applying gaussian filtering using opencv library</p>	 <p>Figure 18: Result of applying gaussian filter using function from scratch</p>

2. Comparison of Edge Detection:

	Built-in Function	Our Implementation
Sobel Edge Detection	 <p>Figure 19: Result of applying sobel edge detection using opencv library</p>	 <p>Figure 20: Result of applying sobel edge detection using function from scratch</p>
Roberts Edge Detection	 <p>Figure 21: Result of applying Roberts edge detection using opencv library</p>	 <p>Figure 22: Result of applying Roberts edge detection using function from scratch</p>
Prewitts Edge Detection	 <p>Figure 23: Result of applying Prewitts edge detection using opencv library</p>	 <p>Figure 24: Result of applying prewitts edge detection using function from scratch</p>

3. Thresholding:

	Built-in Function	Our Implementation
Global Thresholding	<div><div>Input</div><div>Output</div><div></div><div></div></div> <div>Figure 25: Result of applying global thresholding using opencv library</div>	<div><div>Input</div><div>Output</div><div></div><div></div></div> <div>Figure 26: Result of applying global thresholding using function from scratch</div>
Local Thresholding	<div><div>Input</div><div>Output</div><div></div><div></div></div> <div>Figure 27: Result of local thresholding using opencv library</div>	<div><div>Input</div><div>Output</div><div></div><div></div></div> <div>Figure 28: Result of applying local thresholding using function from scratch</div>

4. Comparison of Calculating Histograms :

Input RGB Image:



Figure 29: Input Image (RGB Image)

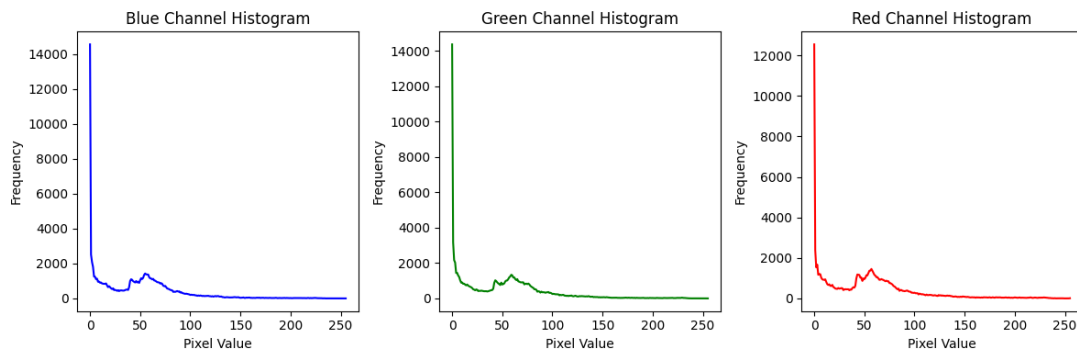


Figure 30: Color Histograms for each channel with built-in function

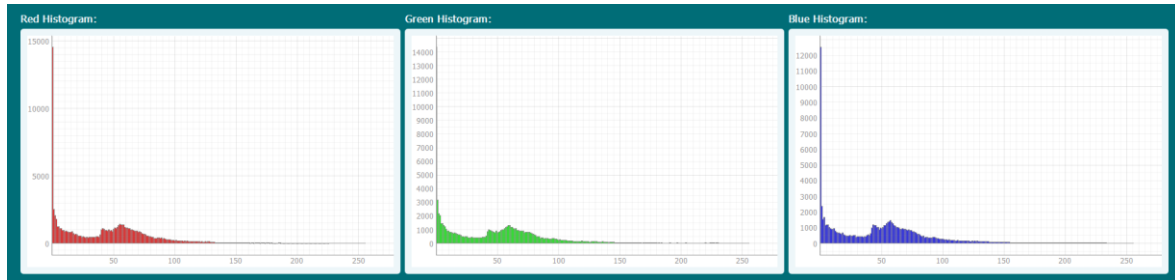


Figure 31: Color Histograms for each channel from our implementation

Input gray scale image :



Figure 32: Input Image (Gray Scale Image)

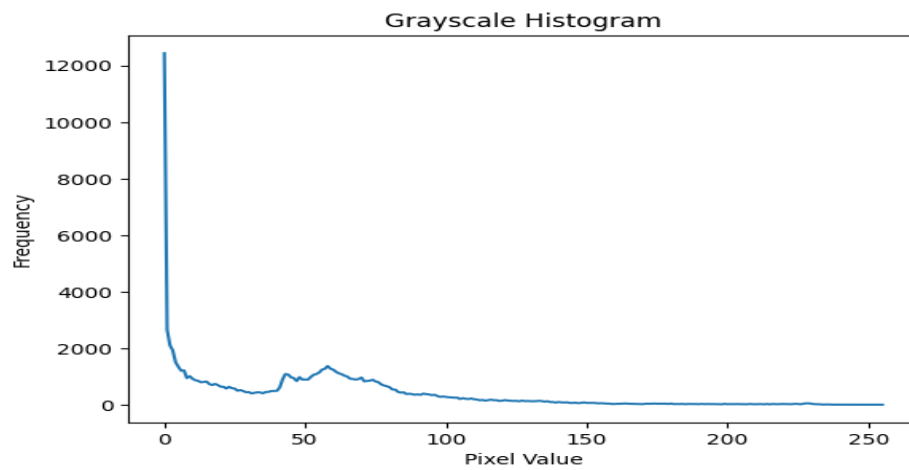


Figure 33: Histogram with built-in function

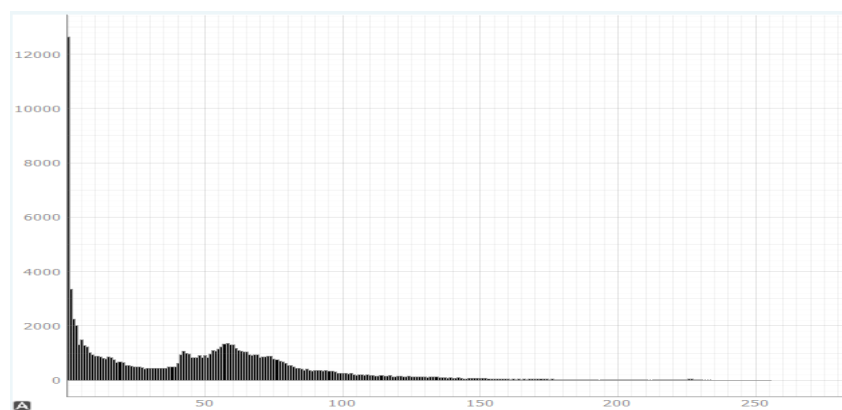


Figure 34: Histogram from our implementation

5. Comparison of Equalization :

Equalized Image with built-in function :



Figure 35: Equalized Image with built-in function

Its Histogram after Equalization:

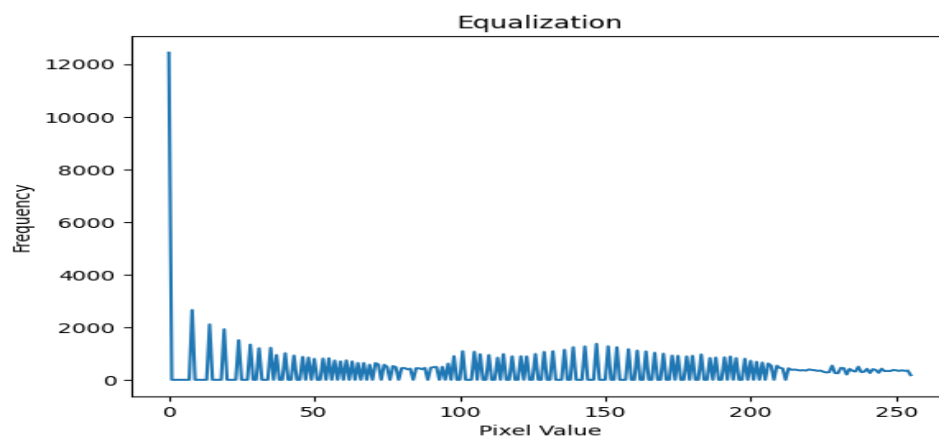


Figure 36: Histogram of equalized image with built-in function

Equalized Image with our function :



Figure 37: Equalized image with our function

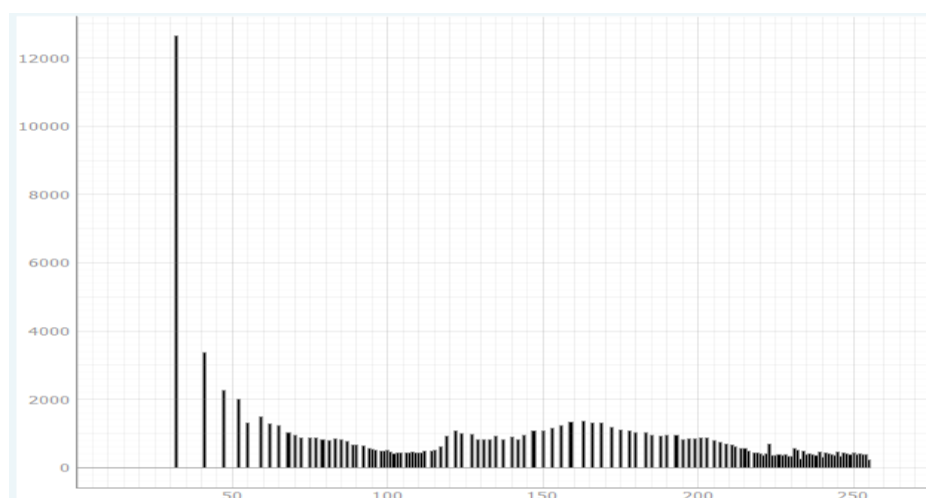


Figure 38: Histogram of equalized image with our function

Screenshots from our application:

1. Main Image:

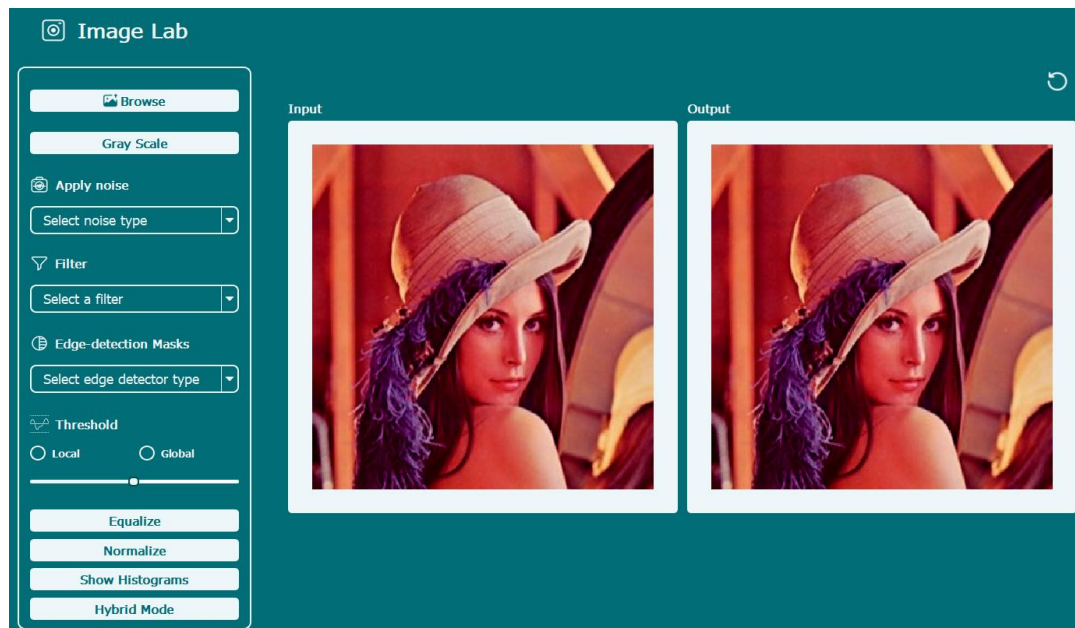


Figure 39: The main page of our application

2. Histograms:

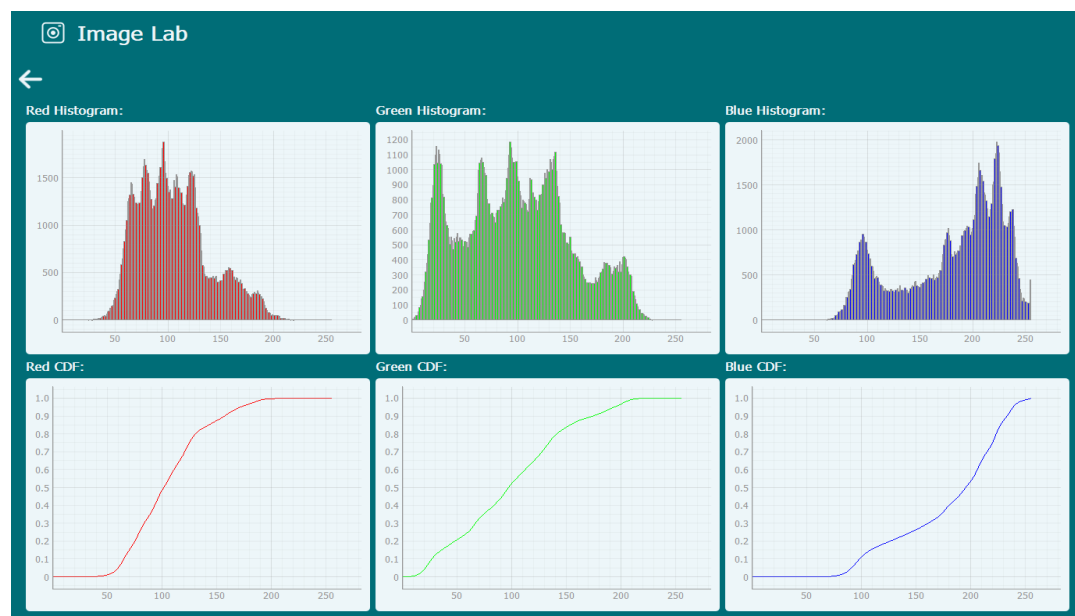


Figure 40: The page that displays histograms of RGB image

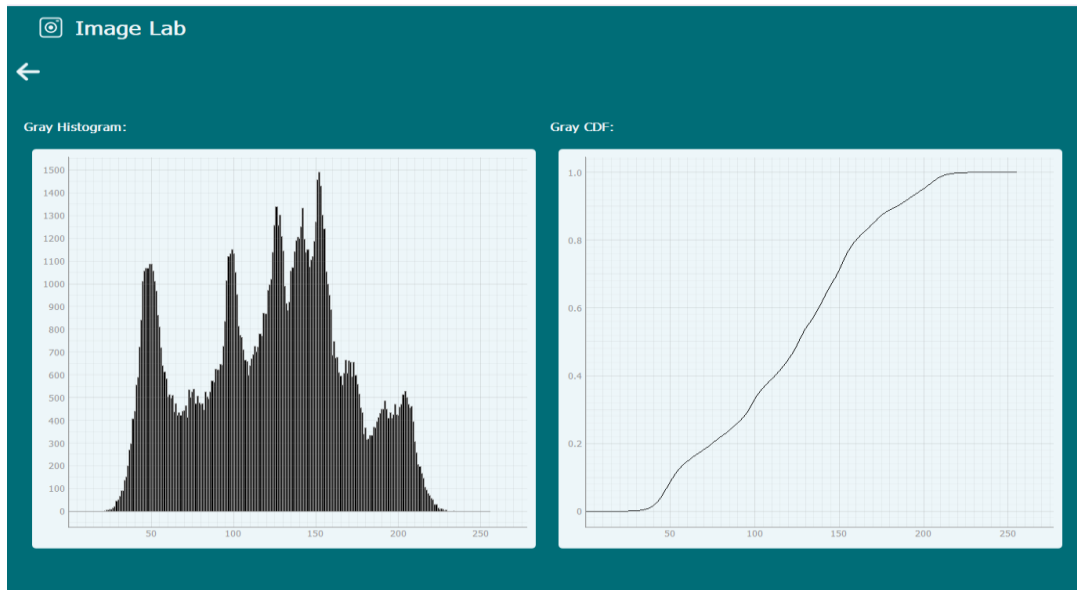


Figure 41: The page that displays the histogram gray-scaled image

3. Hybrid Image Mode:

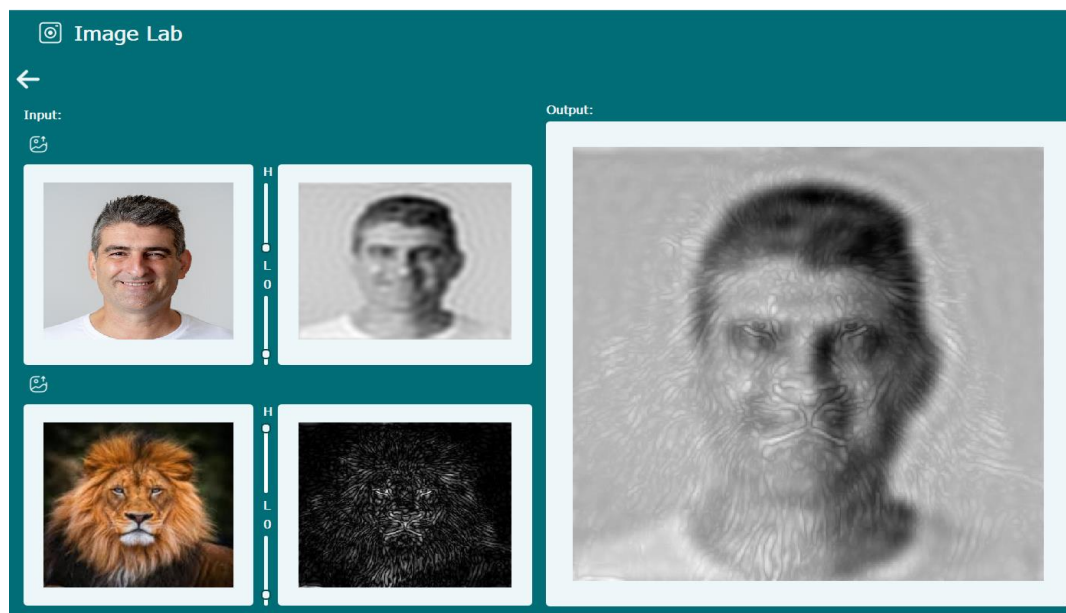


Figure 42: The page that displays the hybrid mode

Link of Github repo:

https://github.com/Mostafaali3/Image_Processing_Lab

References

- <https://www.naukri.com/code360/library/adaptive-thresholding>
- <https://medium.com/geekculture/image-thresholding-from-scratch-a66ae0fb6f09>
- <https://www.sciencedirect.com/topics/engineering/global-thresholding#:~:text=A%20global%20thresholding%20technique%20is,obtained%20from%20the%20whole%20image.>

- <https://medium.com/@wilson.linzhe/digital-image-processing-in-c-chapter-1-mean-and-median-filter-b4c4d0775e14>
- <https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm>
- <https://medium.com/@nikatsanka/comparing-edge-detection-methods-638a2919476e>
<https://medium.com/towards-data-science/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123>