Task 1

Lab1

Team No.7

Supervisors
Dr. Ahmed Badawi
Eng. Omar
Eng. Yara

Name	B.N	Sec
Ahmed Raafat	3	1
Aya Ahmed	14	1
Salah Mohamed	33	1
Abdelrahman Sayed	35	1

TABLE OF CONTENTS

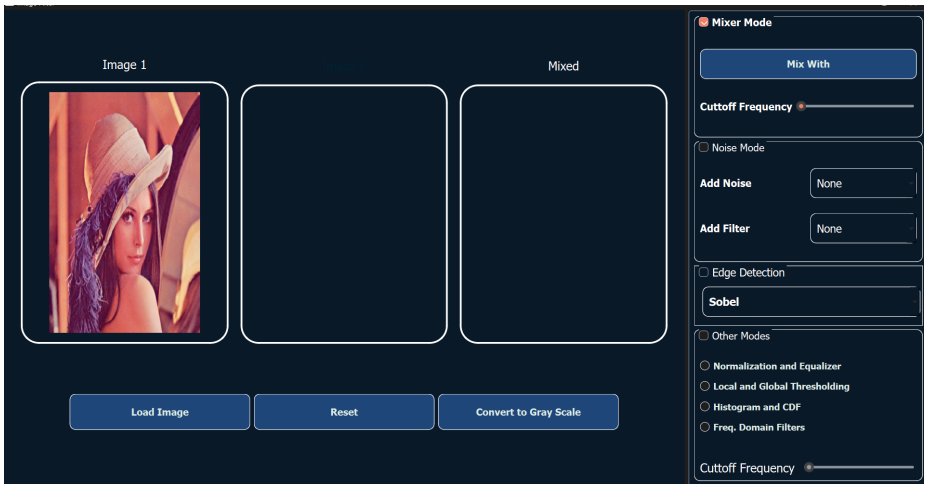
INTRODUCTION

This project implements an Image Processing Toolbox using PyQt6. It offers noise simulation, filtering, edge detection, and histogram analysis. Designed for educational and practical use, it integrates classical algorithms (Sobel, Otsu, Fourier) with a responsive GUI for real-time visualization and multi-image comparison.

UI

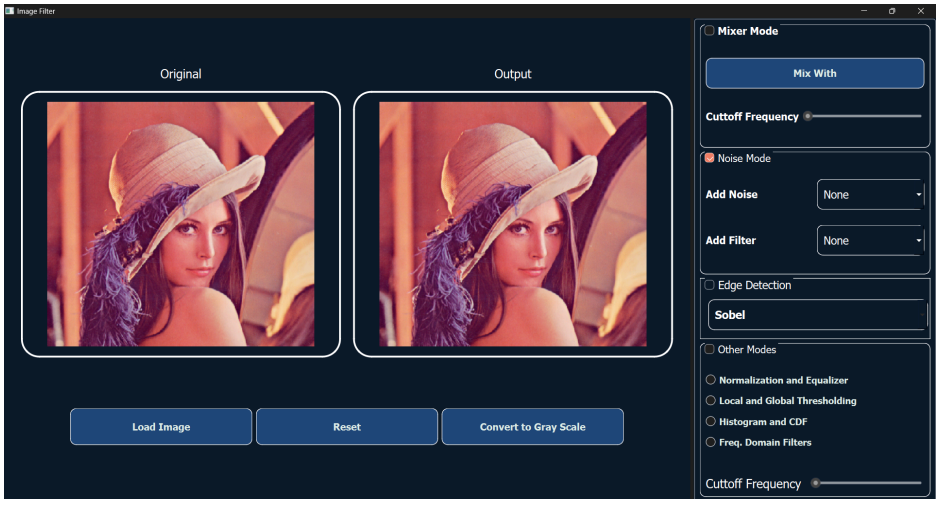
1. **Mixer Mode**:

Blend two images using frequency-domain manipulation with adjustable cutoff sliders.



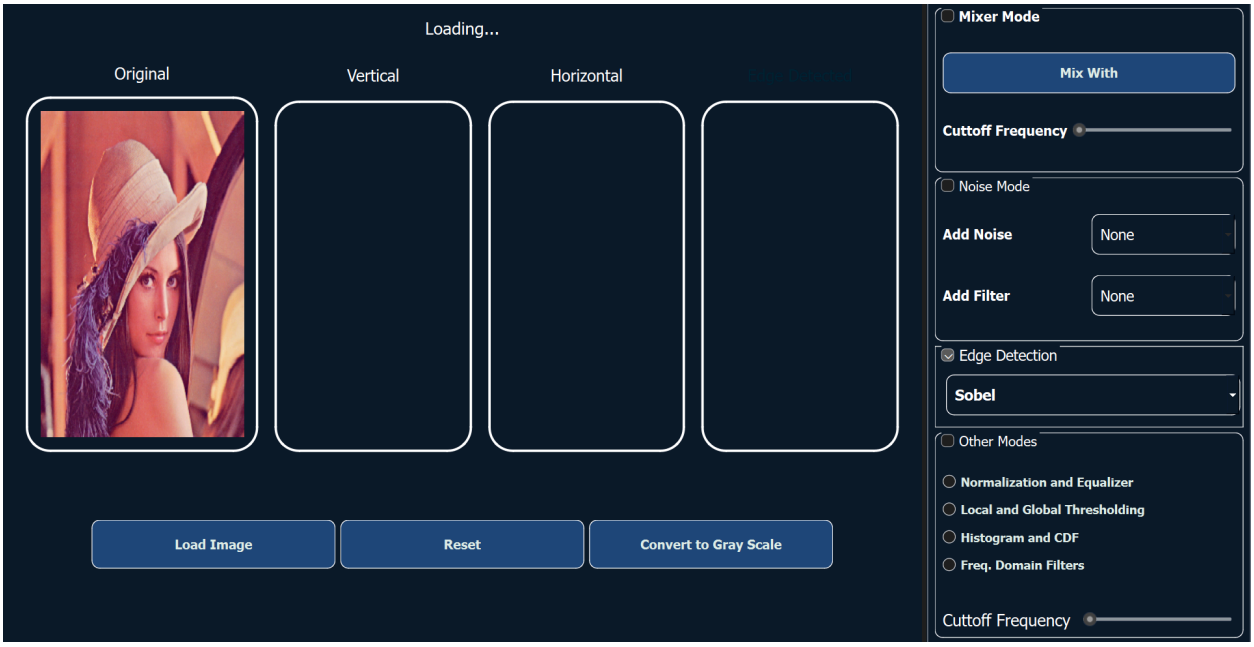
2. Noise Mode:.

Add Gaussian/Uniform/Salt-and-Pepper noise and apply filters (Median, Gaussian, Average) via real-time sliders



3. Edge Detection:

Detect edges using Sobel, Roberts, Prewitt, or Canny algorithms with a dropdown selector.



4. Other Modes:

Normalization, thresholding, histogram/CDF analysis, and frequency filtering (low/high-pass) are performed via radio buttons and sliders.

Experiments & Results

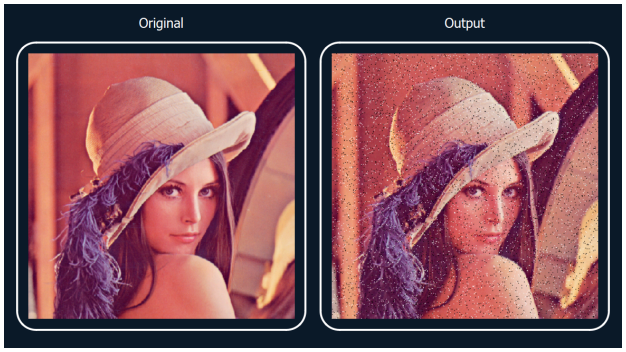
1. Adding Noise

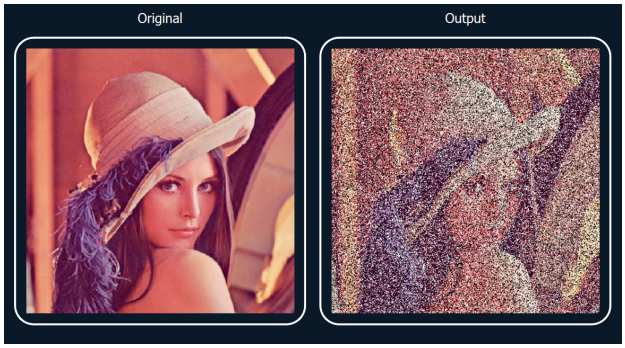
a. Salt and pepper

Salt-and-pepper noise introduces random white ("salt") and black ("pepper") pixels into an image. The algorithm:

1. Generates a random value for every pixel.
2. If the value < prob/2, the pixel becomes black (0).
3. If the value > 1 - prob/2, the pixel becomes white (255).
4. Otherwise, the pixel remains unchanged.

> Below is an example of the effect of the Prob Variable that controls how much salt and pepper noise added



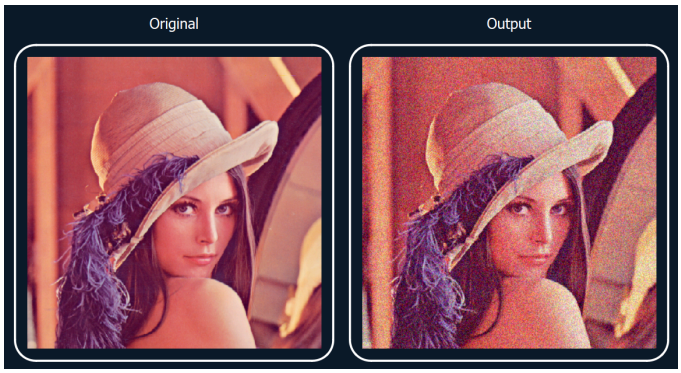


B. Uniform Noise

Uniform noise adds random values to pixels from a **uniform distribution** within a specified range (e.g., [−*intensity*, *intensity*]). The algorithm:

1. Generates a random value for every pixel from a uniform distribution.
2. Adds/subtracts this value to the pixel's intensity.
3. Clips the result to stay within [0, 255].

We can control the amount of uniform noise added by adjusting the intensity variable, the more the intensity value range, the more noise added



C. Gaussian Noise

The Gaussian noise works like the uniform noise but the random values are of normal distribution (mean =0, std = 25)

2. Noise Filter/ Removal

In our project, we implemented three main noise removal algorithms that work in the frequency domain: a Gaussian filter, an average filter, and a median filter. These algorithms are applied after adding noise (e.g., salt and pepper, uniform, or Gaussian noise) to an image. Each filter uses a different approach to smooth out the noise while trying to preserve image details.

a. Gaussian Filter

The Gaussian filter removes high-frequency noise by attenuating sharp transitions. In our implementation, we first create a Gaussian kernel directly in the frequency domain using the formula:

$$G(f_x, f_y) = \exp\left(-2\pi^2\sigma^2(f_x^2 + f_y^2)\right)$$

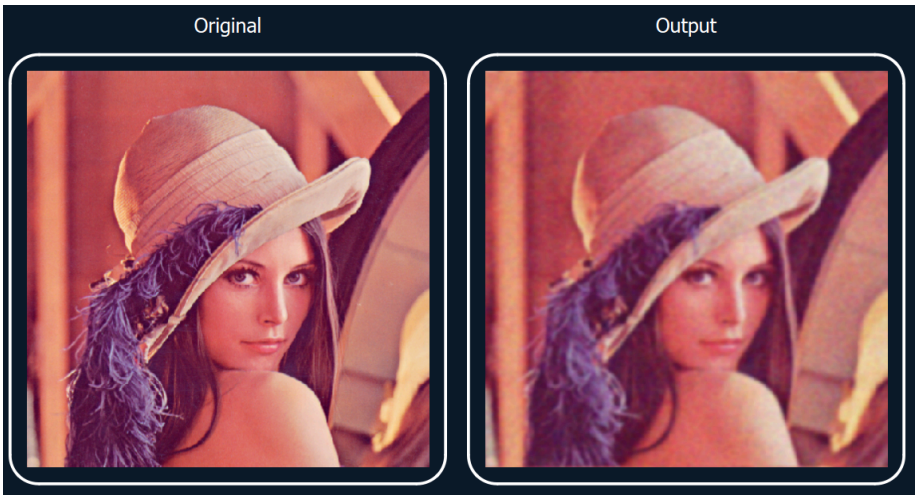
This is achieved by:

- Generating frequency bins for the image dimensions using `np.fft.fftfreq`.
- Forming a meshgrid of these frequency values.
- Computing the squared distance from the center, then applying the Gaussian function.

The filter is applied by converting each image channel to the frequency domain (via FFT), multiplying by the Gaussian kernel, and transforming back using the inverse FFT.

Key Parameter – Sigma:

- **Low Sigma:** Results in a narrow Gaussian, applying mild smoothing that preserves details while only slightly reducing noise.
- **High Sigma:** Produces a broader Gaussian, which more strongly attenuates high-frequency noise but can also blur fine details and edges.



B. Median Filter

- **How It Works:**
  The median filter is a non-linear method that replaces each pixel with the median value of its neighborhood. It is particularly effective at removing impulse noise (like salt and pepper noise) because the median is robust to outliers.
    - **Traditional Approach:** In the classic spatial-domain method, a small window (e.g., 3x3) moves across the image. For each window, the median

value of the pixels is computed and used as the new pixel value. This requires iterating over each pixel and sorting the neighborhood values—a process that can be computationally intensive.
    - **Frequency Domain Variant:** In our example, we mimic the average filter approach in the frequency domain as an alternative. Although true median filtering is non-linear and does not have a simple frequency domain equivalent, our variant provides a faster, albeit approximate, solution.
- **Computational Complexity:**
    - The **traditional median filter (That we used at first)** uses nested loops for each pixel and window, leading to high computational cost, especially on large images.
    - The **frequency domain approach (We are using now)** avoids explicit loops by performing FFTs and multiplications, which are generally faster for large-scale operations, though it may not fully capture the non-linear nature of the median operation.

C. Average Filter

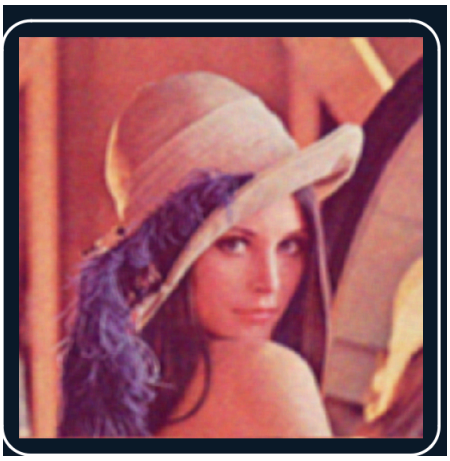
How It Works:

The average filter (or mean filter) smooths an image by replacing each pixel with the average of its surrounding pixels. Our implementation starts with a small spatial kernel filled with ones (normalized by the number of elements) to compute the mean. This kernel is padded with zeros so that its size matches the image dimensions.

- We then shift the kernel (using `np.fft.ifftshift`) and compute its FFT to obtain

its frequency domain representation.

- The image's FFT is multiplied by this frequency-domain kernel, which, thanks to the convolution theorem, is equivalent to convolving the image with an averaging kernel in the spatial domain.
- An inverse FFT converts the result back into the spatial domain.



3- Edge Detection

The implementation of Edge Detection consists of:

1- Convert image into grayscale

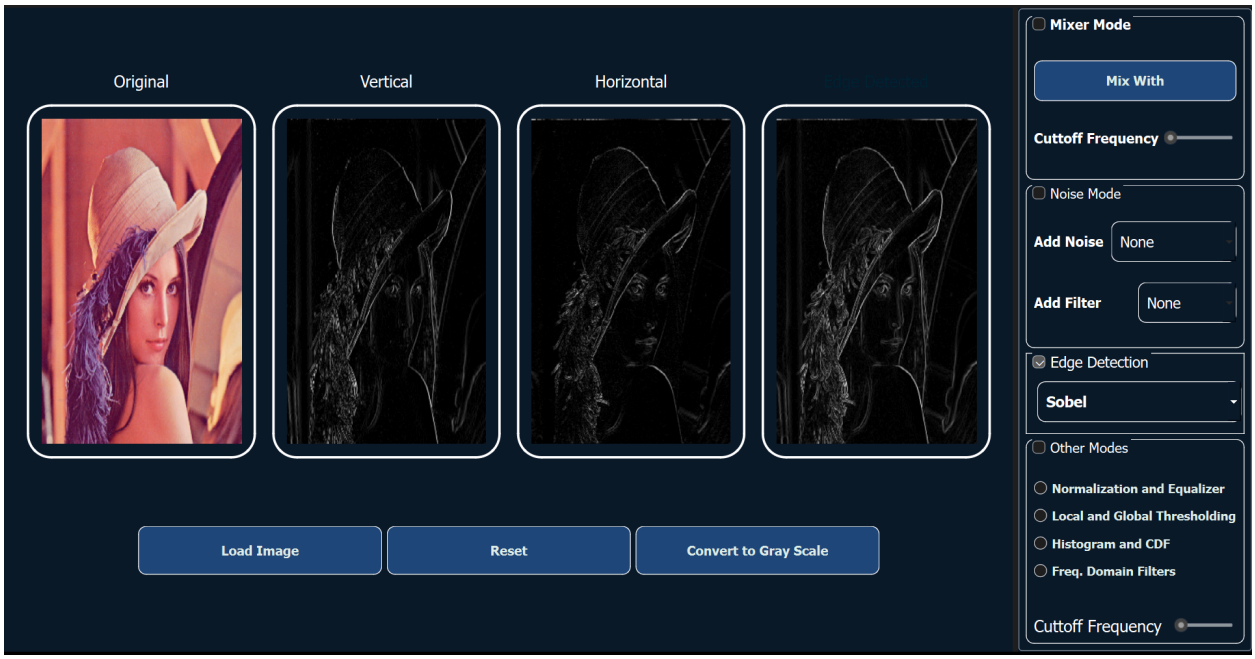
2- Transform data from the spatial domain into the frequency domain

3- Declare the mask depending on the filter type.

4- Apply Multiplication (Convolution in the Spatial domain equal to multiplication in the frequency domain

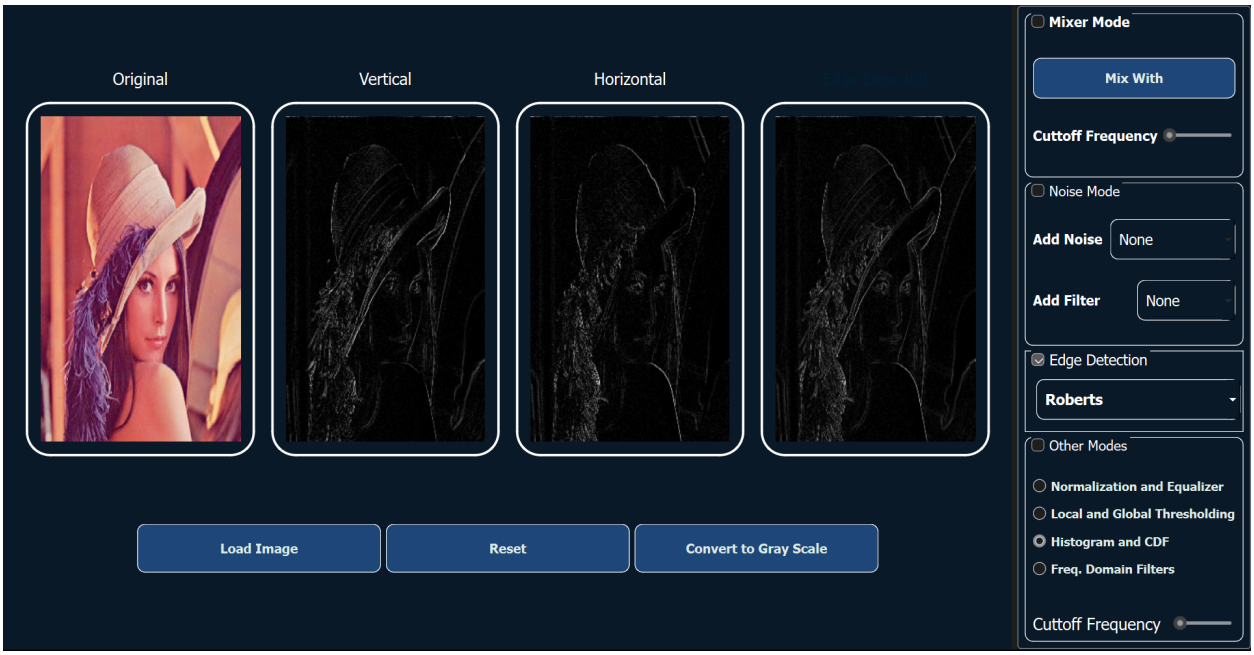
5- Inverse Fourier transform and display data as QImage

1- Sobel Algorithm

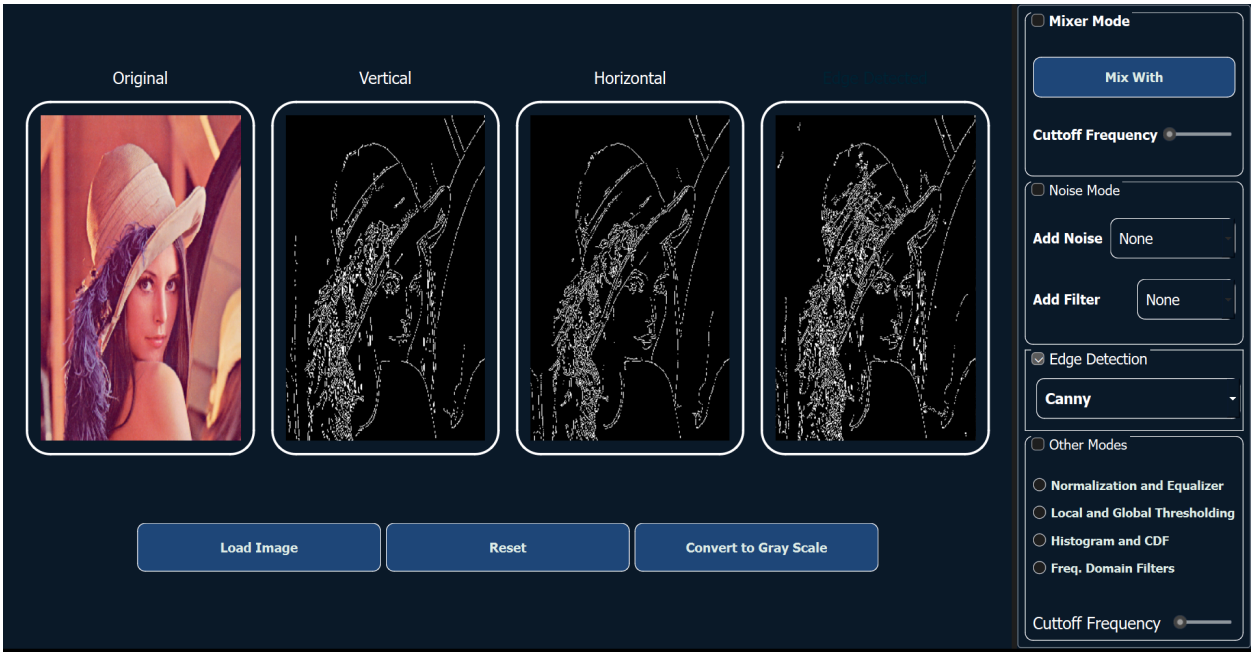


2- Perwitt Algorithm

3- Robert Algorithm



4- Canny Algorithm



4- Other Modes

A- Normalization and Equalization

The Implementation of this Mode consists of:

1- Convert the image into Grayscale

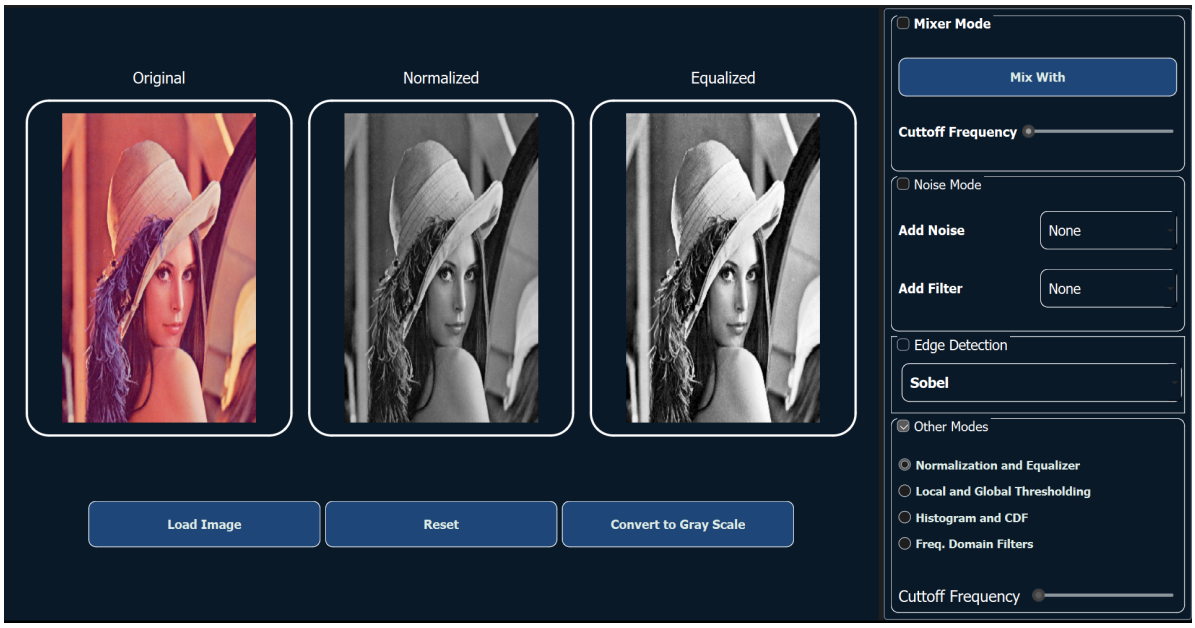
2- Calculate the histogram (add one for each intensity value that exists in the image's pixels

3- Mapping pixel's value using normalized CDF (Cumulative Distribution Function)

4- Normalize the image data depending on :

$$z = \frac{x - \min(x)}{\max(x) - \min(x)}$$

5- return both (normalized and equalized) image



B- Histogram and CDF

This mode's implementation consists of :

1- Calculating Histogram by adding one (1) for the value of intensity stored in the pixel to get the frequency of each intensity value

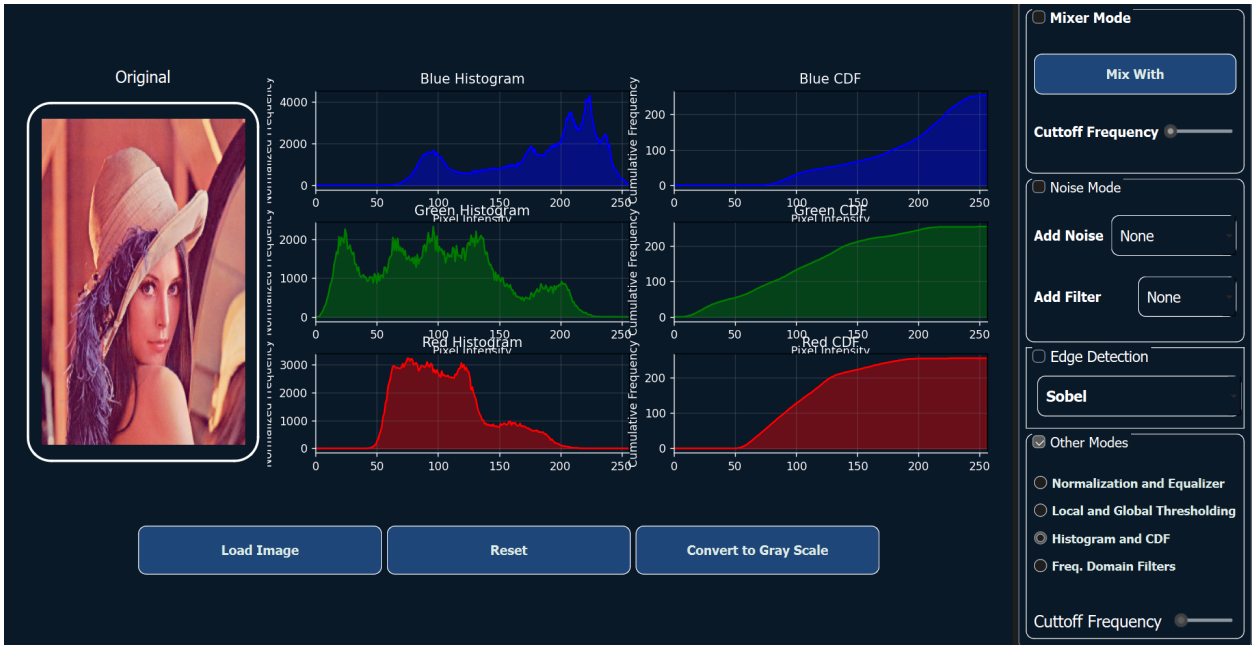
$$n = \sum_{i=1}^{k} m_i .$$

2- Calculating CDF (Cumulative Distribution Function) by summoning the values of the histogram and then normalizing it to make the values between (0 - 255)

$$\mathrm{CDF}(i) = \sum_{j=0}^{i} h(j) \qquad \mathrm{CDF}_{\mathrm{normalized}} = \frac{\mathrm{CDF} - \mathrm{CDF}_{\min}}{\mathrm{CDF}_{\max} - \mathrm{CDF}_{\min}} \times 255$$

3- Apply it to all three channels (Red, green, and blue)

4- Take this data and draw using matploitlib



C- Local and Global thresholding

This mode's implementation consists of :

1- Convert the image into Grayscale

2-Apply Local (Adaptive Mean) Thresholding, which calculates a threshold for small regions of the image dynamically.

$$T(x,y) = \frac{1}{S} \sum_{i,j \in S} f(i,j) - C$$

Where:

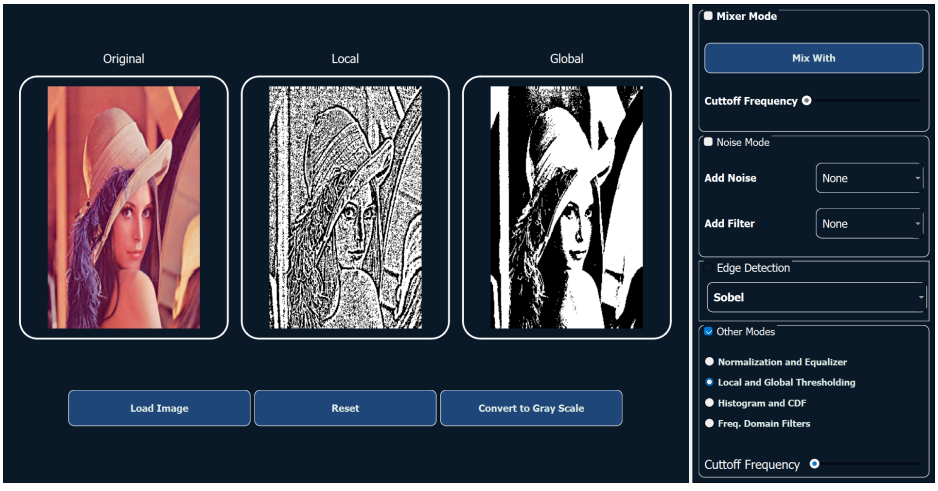
- S is the local window size
- C is a constant subtracted to fine-tune the threshold

3- Apply Global (Otsu's) Thresholding, which automatically determines an optimal threshold based on the image histogram.

$$g(x,y) = \begin{cases} 255, & \text{if } f(x,y) > T \\ 0, & \text{otherwise} \end{cases}$$

4- Normalize the thresholded images to the range **0-255** using the normalization formula.

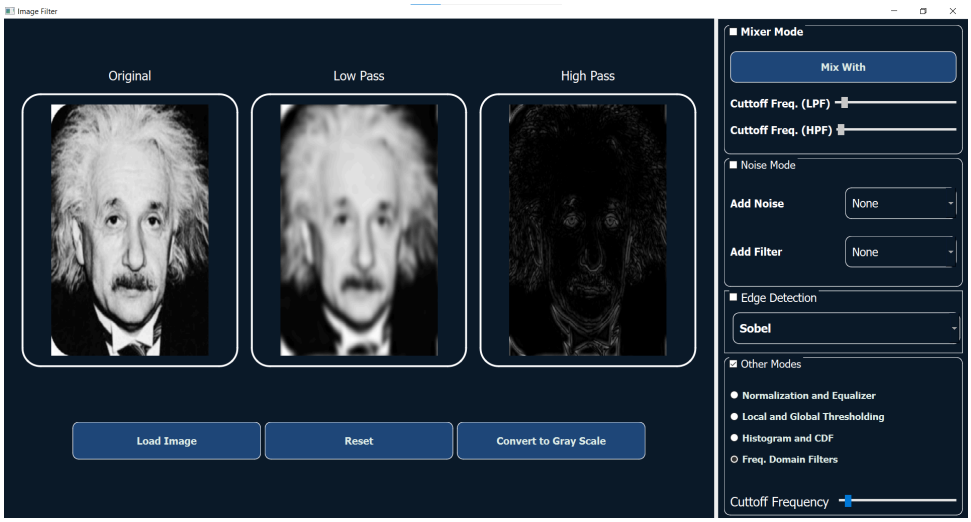
5- Return both thresholded images (local and global thresholding).



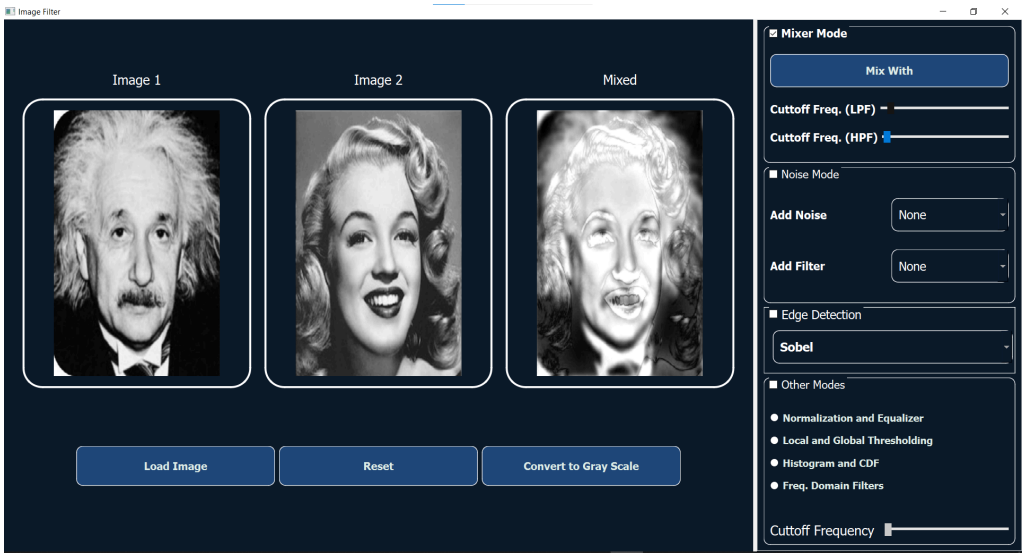
d- frequency domain filters

1. **Frequency Mask Creation:**
    - The `_create_frequency_mask` function generates a circular mask in the frequency domain.
2. **Filters Application:**
    - The `apply_hpf` and `apply_lpf` functions apply the filter to the image.
    - It iterates through each channel of the image and applies the filter.
    - The `_apply_filter_to_channel` function performs the actual filtering in the frequency domain.
3. **Filter Application Details (`_apply_filter_to_channel`):**
    - It computes the 2D Fourier transform using `fftpack.fft2` and shifts the zero-frequency component to the center using `fftpack.fftshift`.
    - The frequency mask (LPF or HPF) is applied to the shifted Fourier transform.
    - The inverse Fourier transform is computed to get the filtered image.
4. **Frequency Mask Formula:**
    - The frequency mask is created based on the distance from the center of the frequency spectrum:

$$\text{center\_dist} = \sqrt{(x - c_{\text{col}})^2 + (y - c_{\text{row}})^2}$$



4- Mixer modes (Hybrid Images)



1. Image Resizing and Padding:
    - The **_resize_to_match** function resizes and pads the two input images to have the same dimensions.
    - It finds the maximum height and width between the two images.
    - The images are resized using PIL (Pillow) to the maximum dimensions and then padded with zeros to ensure they have the same size.

2. Frequency Domain Filtering:
    - The first image (img1) is passed through a Low-Pass Filter (LPF) using the **apply_lpf** function with a specified cutoff frequency (cutoff_lpf).
    - The second image (img2) is passed through a High-Pass Filter (HPF) using the apply_hpf function with a specified cutoff frequency (cutoff_hpf).

3. Image Combination:
    - The filtered images (low-passed and high-passed) are added together.
    - The resulting image is clipped to the range [0, 255] and converted to an 8-bit unsigned integer format.

4. Formula:
    - The hybrid image is created using the following formula:

$$\text{Hybrid Image} = \text{LPF}(\text{Image}_1, \text{cutoff}_{\text{lpf}}) + \text{HPF}(\text{Image}_2, \text{cutoff}_{\text{hpf}})$$