Faculty of Engineering

Cairo University

Computer Vision (SBE3230)

Assignment 1

“ Filtering & Edge Detection ”

Name	BN
Madonna Mosaad	49
Nariman Ahmed	71
Nancy Mahmoud	72
Yassien Tawfik	81

Under Supervision of :

Dr /Ahmed Badawy

Eng /Omar Hesham

Eng /Yara Wael

Table of Contents

Project Overview

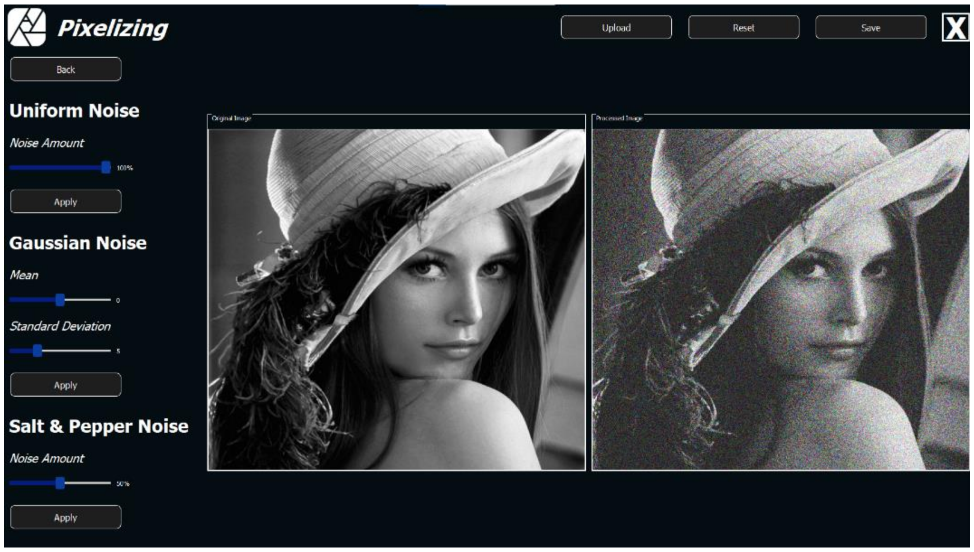
This project focuses on implementing various image processing techniques, including noise addition, filtering, edge detection, histogram analysis, and frequency domain filtering, using **Python** and **OpenCV**. To ensure modularity, scalability, and maintainability, the project has been developed using **Object-Oriented Programming (OOP)** principles. The code is organized into separate classes and files, each responsible for a specific functionality. Additionally, a user-friendly **Graphical User Interface (GUI)** has been designed using **PyQt** to allow users to interact with the application and test different image processing algorithms.

Features

1. Noise Addition

1- Uniform Noise

Functions:



Uniform Noise
Noise Amount 100%

Gaussian Noise
Mean 0
Standard Deviation 5

Salt & Pepper Noise
Noise Amount 50%

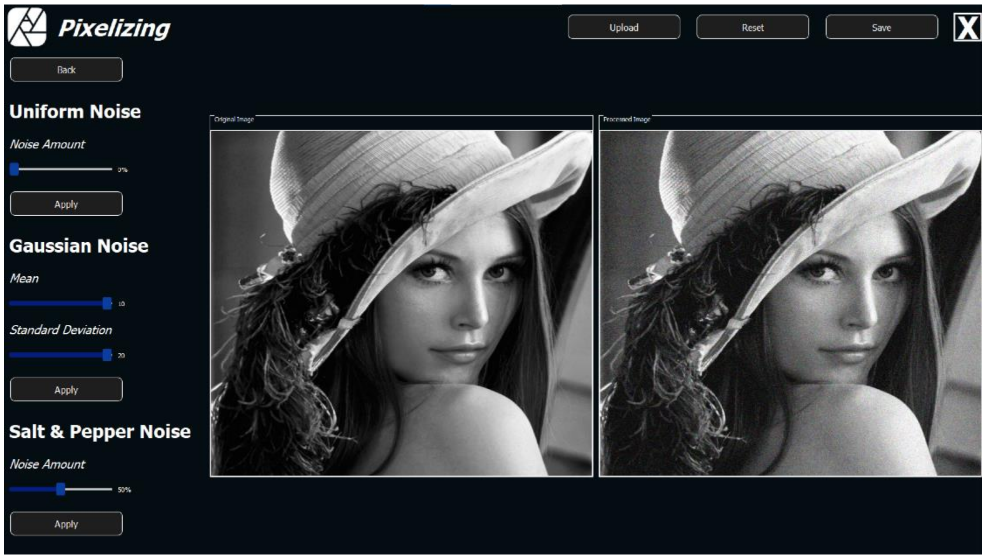
Uniform noise is a type of additive noise where the noise values are uniformly distributed across a specified range. In the “add_uniform_noise” method, the “noise_amount” parameter determines the intensity of the noise. This parameter is scaled by 100 to define the range of noise values, which ensures that the noise can be both positive and negative. The “np.random.uniform” function generates random noise values uniformly distributed between -high and high, where high is derived from the “noise_amount”. The generated noise is then added to the input image using cv2.add. Since the noise values can be negative, the image is first converted to int16 to handle the addition properly. After adding the noise, the resulting image may contain pixel values outside the valid range (0–255). To address this , “cv2.convertScaleAbs” is used to convert the image back to an 8-bit format (uint8), ensuring that all pixel values are within the valid range.

Observations:

Uniform noise is useful for simulating random variations in pixel intensities and is often used to test the robustness of image processing algorithms. However, it tends to produce a more "flat" noise pattern compared to Gaussian noise, as the values are evenly distributed across the range.

2- Gaussian Noise

Functions:



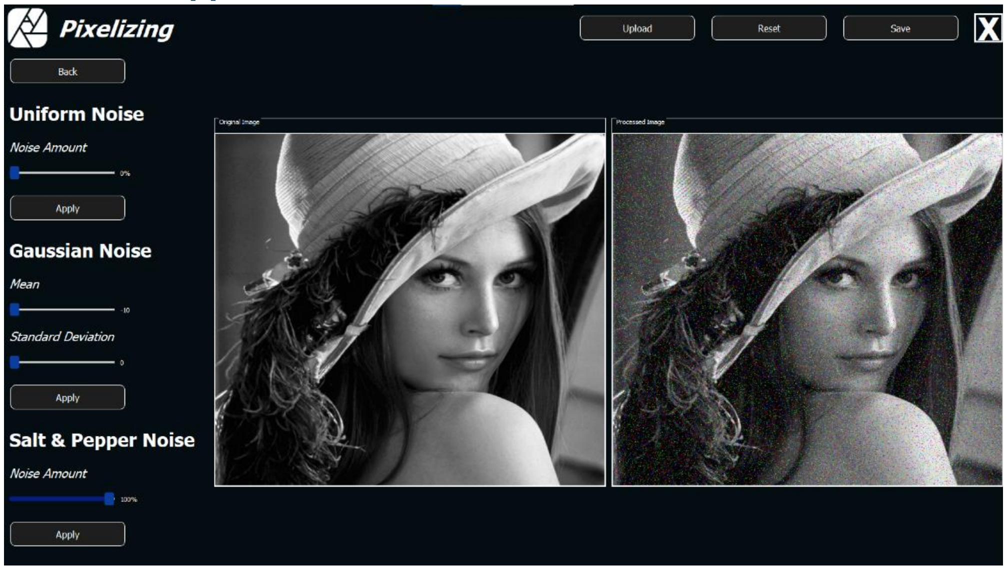
Uniform Noise
Noise Amount 0%
Gaussian Noise
Mean 10
Standard Deviation 20
Salt & Pepper Noise
Noise Amount 50%

Gaussian noise, aka normal noise, is a type of additive noise where the noise values follow a Gaussian (normal) distribution. In the "add_gaussian_noise " method, the mean and sigma parameters define the mean and standard deviation of the Gaussian distribution. The "np.random.normal" function generates random noise values based on these parameters, creating a bell-shaped distribution of noise values centered around the mean. The generated Gaussian noise is added to the input image using cv2.add. Similar to uniform noise, the image is converted to int16 to handle the addition of negative noise values. After adding the noise , cv2.convertScaleAbs is used to convert the image back to an 8-bit format (uint8), ensuring that all pixel values are within the valid range (0-255).

Observations:

The bell-shaped distribution of Gaussian noise means that most noise values are concentrated around the mean, with fewer extreme values, resulting in a smoother noise pattern compared to uniform noise.

3- Salt & Pepper Noise



Uniform Noise
Noise Amount 0%
Gaussian Noise
Mean -10
Standard Deviation 0
Salt & Pepper Noise
Noise Amount 100%

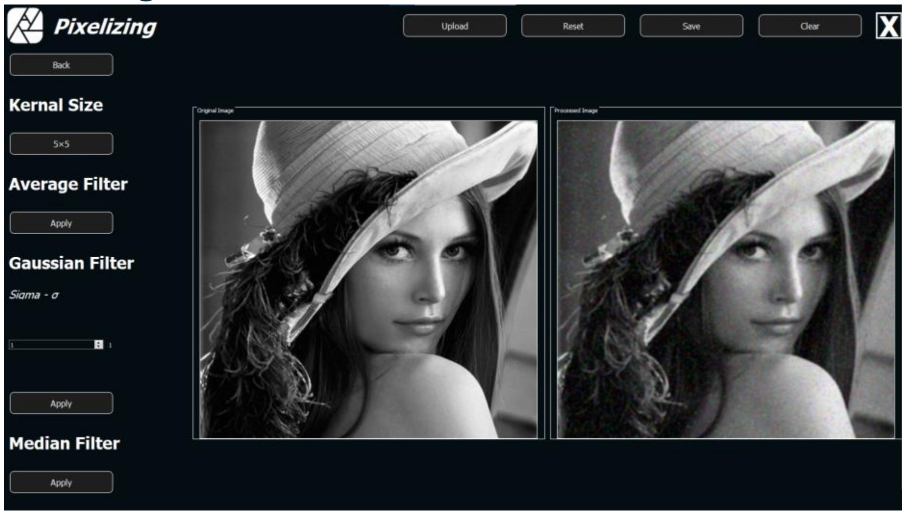
Salt and pepper noise is a type of impulsive noise where random pixels in the image are set to either maximum (salt) or minimum (pepper) intensity. In the “add_salt_and_pepper_noise“ method, the “noise_amount” parameter determines the intensity of the noise. This parameter is scaled by 0.05 to define the probability of salt and pepper noise, ensuring that only a small fraction of pixels are affected. The method first calculates the number of salt and pepper pixels based on the image dimensions and the number of channels (grayscale or RGB). For salt noise, random coordinates are generated using “np.random.randint” , and the pixel values at these coordinates are set to 255 (white). For pepper noise, the same process is followed, but the pixel values are set to 0 (black). In the case of RGB images, the noise is applied to random channels, ensuring that the noise affects all color components.

2. Filters

For all filters:

- The image is padded to handle edge pixels using reflection padding.
- The kernel is slid over the image, and the filtered value for each pixel is computed based on its neighborhood.
- The result is clipped to the valid range [0, 255] and converted to uint8.

1- Average Filter



Kernal Size 5×5

- The apply_average_filter method applies a mean filter by averaging pixel values in a neighborhood defined by the kernel size. It uses a uniform kernel where all elements are equal and normalized.

2- Gaussian Filter

- The apply_gaussian_filter method applies a Gaussian filter using a kernel generated by the gaussian_kernel method. The kernel weights are based on a Gaussian distribution, which gives more importance to central pixels and less to peripheral ones.

3- Median Filter

- The apply_median_filter method applies a median filter by replacing each pixel with the median value of its neighborhood. This is particularly effective for removing salt-and-pepper noise.

Observations

- **Noise Reduction and Blurring**: All filters reduce noise but introduce some level of blurring. Increasing the kernel size increases the blurring effect.

- **Gaussian Filter**: Increasing the sigma value improves noise reduction for Gaussian noise, as it smooths the image more effectively. The Gaussian filter produces results similar to the average filter but with better edge preservation.
- **Average Filter**: Provides general smoothing but may blur edges and fine details.
- **Median Filter**: Excels at removing salt-and-pepper noise while preserving edges better than the average and Gaussian filters.

3. Edge Detection

1. Sobel Edge Detection

The Sobel operator is a widely used edge detection technique that calculates the gradient of the image intensity. It uses two 3x3 kernels, one for detecting horizontal edges and another for vertical edges. These kernels approximate the derivatives of the image in the x and y directions. The apply_sobel method applies these kernels to the image using the convolve method. The gradients in the x and y directions are combined by computing the magnitude using the formula np.sqrt(grad_x**2 + grad_y**2). This results in an edge map that highlights edges in both directions.

2. Roberts Edge Detection

The Roberts operator is a simple and efficient edge detection method that uses two 2x2 kernels to approximate the gradient at 45-degree angles. These kernels are designed to detect edges in diagonal directions. The apply_roberts method applies these kernels to the image using the convolve method. The result is an edge map that highlights sharp edges, especially in diagonal orientations. Roberts is computationally efficient and works well for images with high contrast and sharp edges.

3. Prewitt Edge Detection

The Prewitt operator is similar to the Sobel operator but uses slightly different 3x3 kernels to approximate the gradient. These kernels are designed to detect edges in both horizontal and vertical directions. The apply_prewitt method defines the Prewitt kernels and applies them to the image using the convolve method. Prewitt is particularly effective for detecting edges in noisy images, as it provides a smoother gradient approximation compared to Sobel. The final output is converted to uint8 for display.

4. Canny Edge Detection

The Canny edge detector is a multi-stage algorithm that produces clean and well-defined edges. It begins by smoothing the image with a Gaussian filter to reduce noise. Next, it computes the gradient magnitude and direction using the Sobel operator. Non-maximum suppression is then applied to thin out the edges, followed by double thresholding to distinguish strong and weak edges. Finally, edge tracking by hysteresis is used to connect weak edges to strong edges. The apply_canny method uses OpenCV's cv2.Canny function, which implements this algorithm. Parameters such as threshold1, threshold2, and apertureSize allow for fine-tuning the edge detection process.

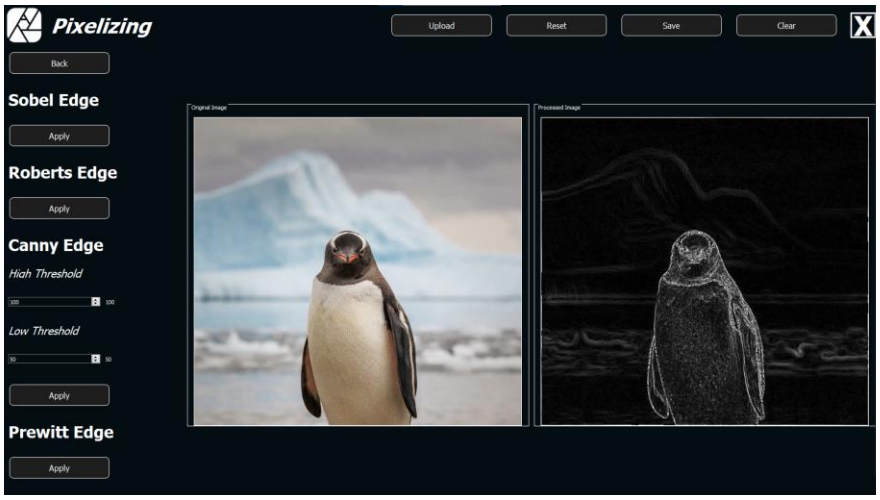
Convolve Method

The convolve method is a helper function that performs the convolution operation between an image and a given kernel. It first calculates the padding size based on the kernel dimensions and pads the image with zeros to handle edge pixels during convolution. The kernel is then slid over the image, and the convolution is computed by multiplying the kernel with the corresponding image region and summing the results. This process is repeated for every pixel in the image, producing an output array that represents the filtered image.

Observations :

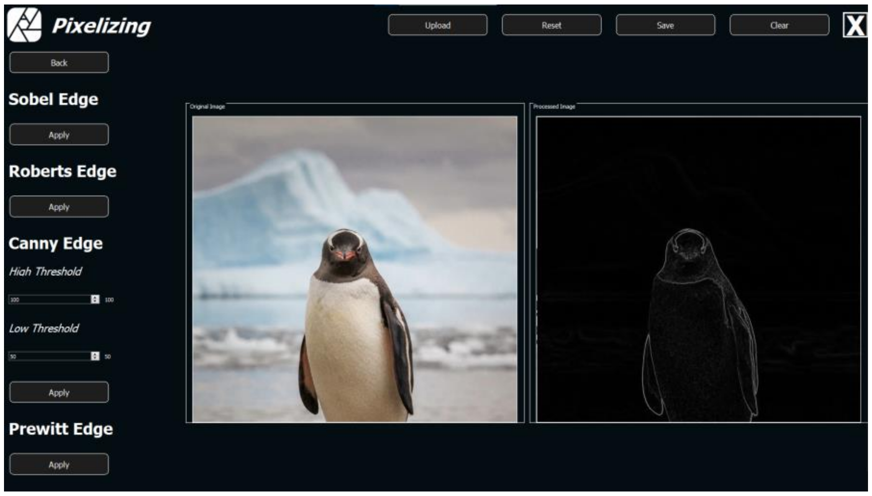
Sobel & Prewitt methods typically provide good edge detection results for general-purpose applications, while the Canny edge detector's output is highly detailed and less prone to noise, making it ideal for complex images.

1- Sobel



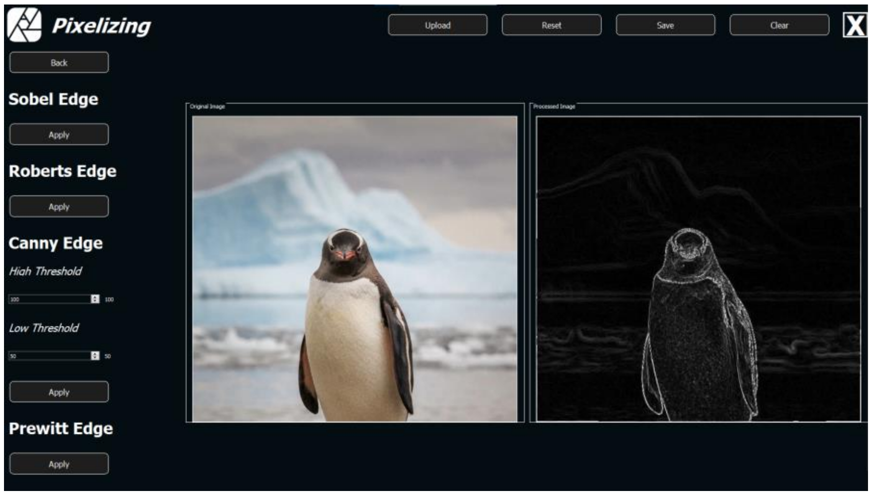
High Threshold 100
Low Threshold 50

2- Roberts



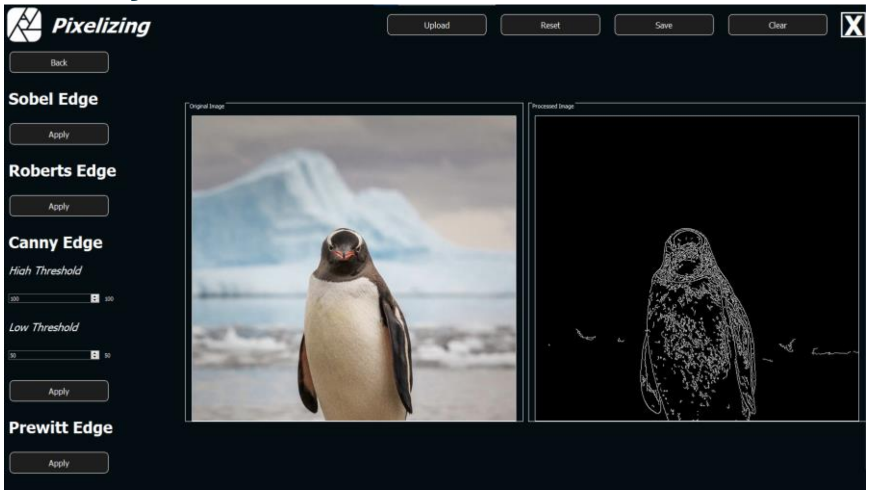
High Threshold 100
Low Threshold 50

3- Prewitt



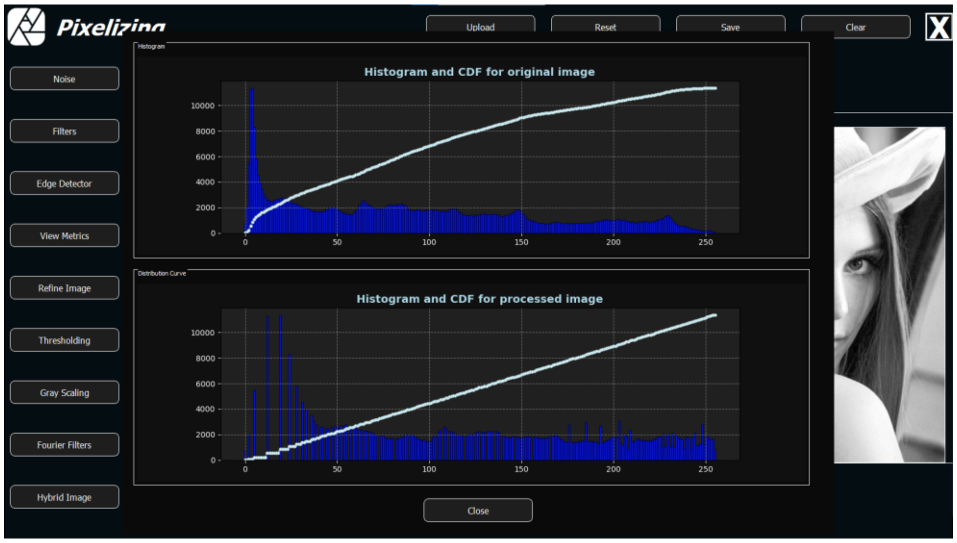
High Threshold 100
Low Threshold 50

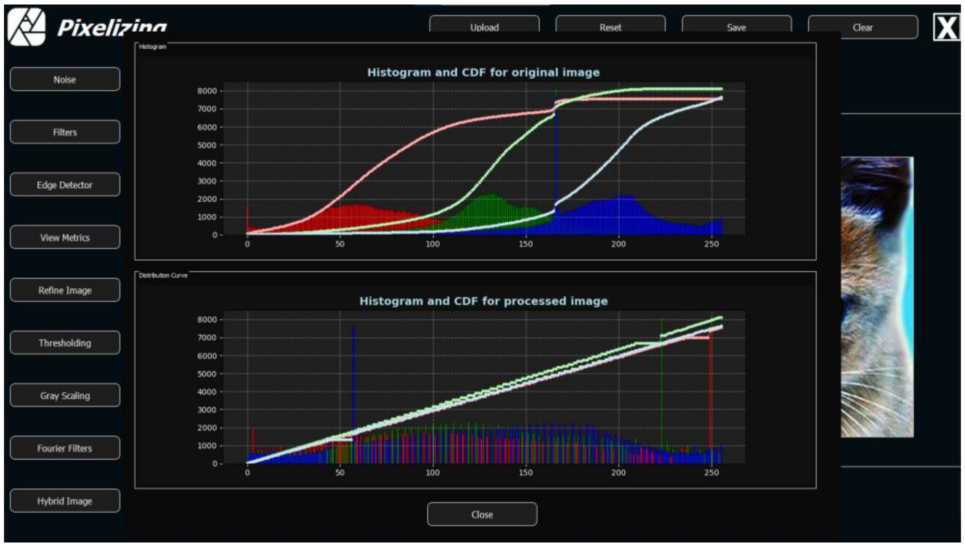
4- Canny



4. Histogram & Distribution Curve

Histogram and CDF Curve after Equalization:



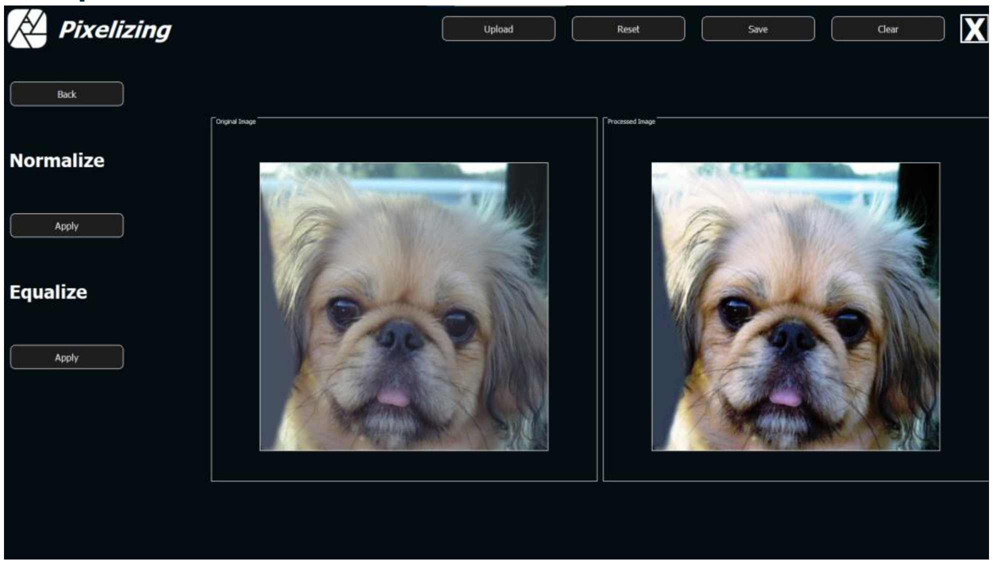


Functions:

The EqualizeHistogram class provides two methods for histogram equalization: equalizeHist for grayscale images and equalizeHistRGB for RGB images. For grayscale images, the method calculates the histogram, computes the probability density function (PDF), and derives the cumulative distribution function (CDF). The CDF is used to map the original pixel values to new values, spreading out the intensity levels to cover the full range [0, 255]. For RGB images, the process is applied independently to each channel (Red, Green, Blue) to preserve the color balance. This ensures that the equalization process enhances the contrast of the image without distorting its colors. Both methods use the histogram and CDF to redistribute pixel intensities, making the image visually clearer and more detailed.

5. Image Equalization & Image Normalization

1- Equalization



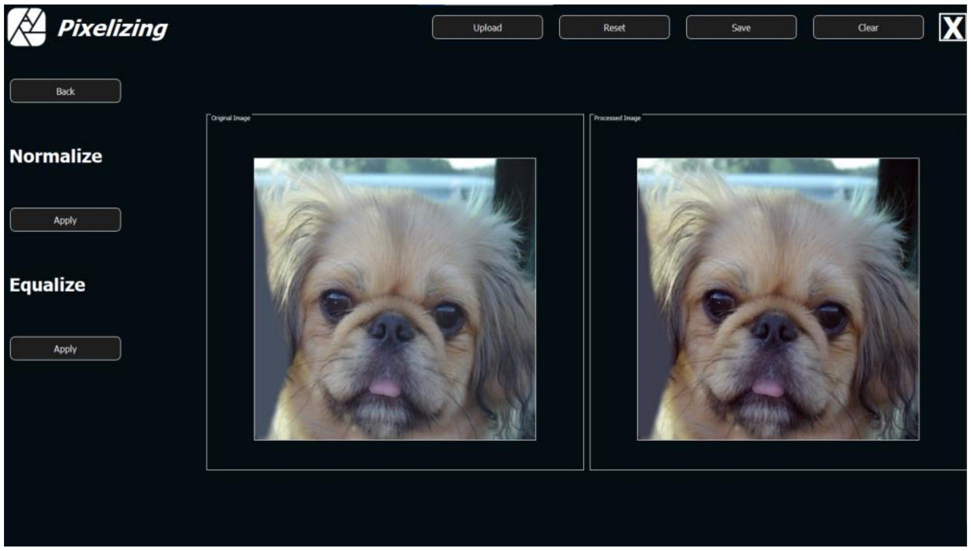
Functions :

The EqualizeHistogram class provides two methods for histogram equalization: equalizeHist for grayscale images and equalizeHistRGB for RGB images. For grayscale images, the method calculates the histogram, computes the probability density function (PDF), and then derives the cumulative distribution function (CDF). The CDF is used to map the original pixel values to new values, spreading out the intensity levels to cover the full range [0, 255]. For RGB images, the process is applied independently to each channel (Red, Green, Blue) to preserve the color balance.

Observation

After equalization, the pixel intensities in the image are redistributed to cover the entire range [0, 255], ensuring that almost all possible intensity levels are utilized. This results in an image with improved contrast and visibility of details, especially in darker or brighter regions. However, for RGB images, equalizing each channel independently can sometimes lead to slight color distortion, as the color balance may shift. Despite this, histogram equalization is highly effective for enhancing images with poor contrast or uneven lighting, as it ensures a more uniform distribution of pixel intensities.

2- Normalization



Functions

The “ImageNormalization” class provides two static methods for normalizing images: normalize_image for grayscale images and normalize_image_rgb for RGB images. Both methods normalize the pixel values of the input image to the range [0, 255]. For grayscale images, the normalization is applied directly to the single channel. For RGB images, each channel (Red, Green, Blue) is normalized independently to preserve the color balance. The normalization formula used is:

$normalized\_image = ((image - min\_val) / (max\_val - min\_val)) * 255,$

where min_val and max_val are the minimum and maximum pixel values in the image or channel. The resulting image is converted to uint8 type to ensure valid pixel values.

Observation:

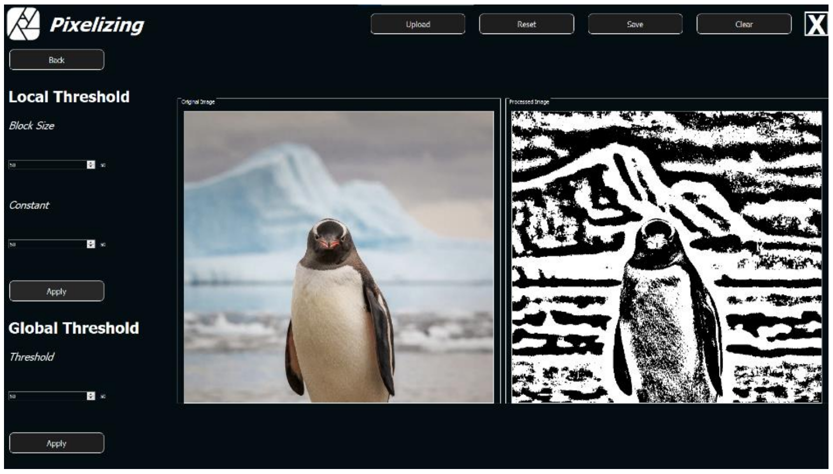
Normalization has a noticeable effect on images with pale colors or low contrast, as it stretches the pixel values to fill the entire range [0, 255]. This enhances the overall brightness and contrast of the image. However, for images that already have good brightness and clearly visible colors, normalization may have little to no effect. This is because the pixel values in such images are already well-distributed across the range, and the normalization process does not significantly alter their distribution.

6. Thresholding

1- Local Thresholding

Functions:

Local thresholding adapts the threshold value for each pixel based on the local intensity of its surrounding region. In the local_threshold method, a sliding window of a specified block_size is used to calculate the local mean intensity around each pixel. The pixel is then classified as white if its intensity is greater than the local mean; otherwise, it is classified as black. This approach is particularly effective for images with uneven lighting or varying contrast



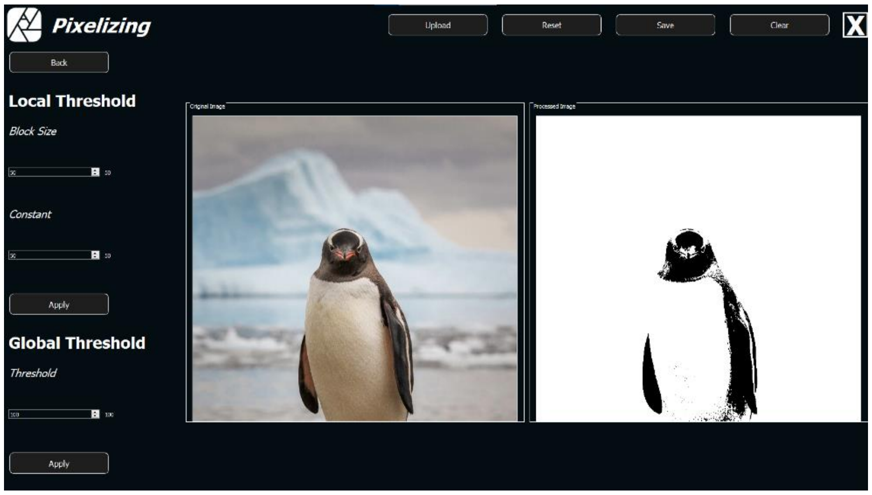
Observations:

- Local thresholding preserves more details compared to global thresholding, as it adapts to local variations in the image.
- It is particularly useful for images with uneven lighting or complex textures.
- However, it may introduce noise or artifacts in regions with low contrast.

2- Global Thresholding

Functions

Global thresholding applies a fixed threshold value to the entire image, classifying pixels as either black or white based on their intensity. In the global_threshold method, a binary image is created where pixels with values greater than the specified threshold are set to 255 (white), and those below the threshold are set to 0 (black). This method is simple and works well for images with uniform lighting or consistent contrast. However, its effectiveness diminishes in images with uneven lighting or varying intensity levels.



Observations :

- Increasing the threshold value results in more pixels being classified as black, making the image darker
- Decreasing the threshold value results in more pixels being classified as white, often leading to a loss of detail and an almost entirely white image.
- Global thresholding produces smoother results but may lose fine details, especially in images with varying lighting.

7. Color to Grayscale Transformation

Functions :

The RGBImageConverter class contains a static method rgb_to_gray that converts an RGB image to grayscale. The method first checks if the input image is in RGB format by verifying its shape. If the image is not RGB, it raises a ValueError. The conversion to grayscale is performed using the **luminosity method**, which applies specific weights to the red (R), green (G), and blue (B) channels to approximate human perception of brightness. The formula used is:

$$gray\_image = 0.2126 * R + 0.7152 * G + 0.0722 * B.$$

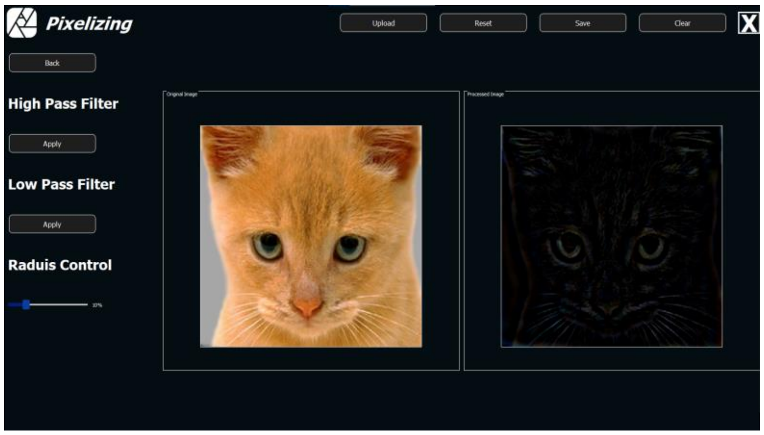
This weighted sum ensures that the **green channel** (which contributes the most to perceived brightness) has the highest weight, while the blue channel (which contributes the least) has the lowest weight. The resulting grayscale image is converted to uint8 type to ensure valid pixel values (0–255).

8. Frequency Domain Filters (Low pass & High Pass Filters)

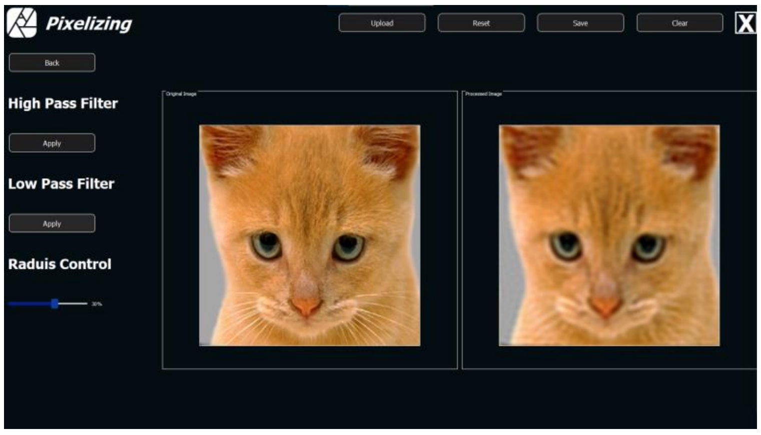
Functions :

The FourierFilters class implements frequency domain filtering using the Fourier Transform to apply **Low Pass** and **High Pass Filters** to images. The get_fft method computes the 2D Fourier Transform and shifts the zero-frequency component to the center. The apply_low_pass method uses a circular mask to allow low-frequency components (smooth regions) to pass while attenuating high-frequency components (edges and noise), whereas the apply_high_pass method inverts the mask to allow high-frequency components (edges and details) to pass while attenuating low-frequency components (smooth regions). The __apply_filter method handles both grayscale and RGB images, applying the filter to each channel independently. After filtering, the inverse Fourier Transform is computed to convert the image back to the spatial domain, and the result is clipped to the valid range [0, 255] and converted to uint8 for display. This approach is effective for smoothing images (low pass) or enhancing edges (high pass) in the frequency domain.

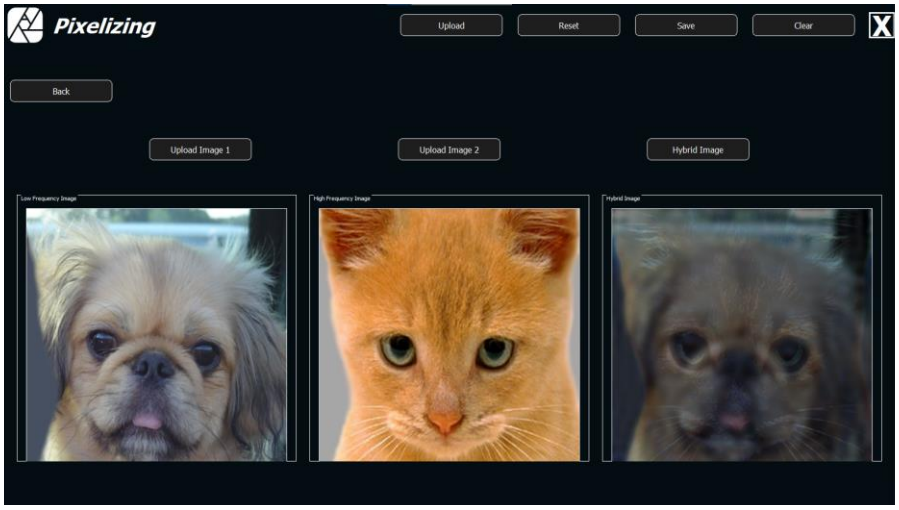
High Pass Filter :



Low Pass Filter :



9. Hybrid Images



Functions:

The HybridImageGenerator class generates a hybrid image by combining the low-frequency components of one image with the high-frequency components of another. It uses the FourierFilters class to apply **low-pass** and **high-pass filters** in the frequency domain. The generate_hybrid_image method first applies a low-pass filter to the low_freq_image to extract its smooth, low-frequency components. Then, it applies a high-pass filter to the high_freq_image to extract its detailed, high-frequency components. Finally, the two filtered images are combined using cv2.addWeighted, which blends them with equal weights (0.5 each) to create the hybrid image. The resulting hybrid image retains the smooth structure of the low-frequency image while incorporating the fine details of the high-frequency image.

Observations:

The intensity and appearance of the hybrid image depend on the radius values of the low-pass and high-pass filters. A larger radius for the low-pass filter retains more low-frequency components, making the image appear smoother, while a smaller radius for the high-pass filter emphasizes finer details. When viewing the hybrid image, the details from the high-pass filtered image are more visible when viewed up close, as high-frequency components correspond to fine textures and edges. Conversely, the smooth regions from the low-pass filtered image become more apparent when viewed from a distance, as low-frequency components dominate the overall structure.