Faculty of Engineering
Cairo University

Cairo University

Computer Vision

Task1

Team 19

Name	Section	B.N
Mohamed Alaa Ali	2	19
Mohamed Ibrahim Ismail	2	9
Mohamed El-sayed Ali	2	10
Mohamed El-sayed Eid	2	11

Introduction:

Embark on an exciting journey into the world of image processing! This task will guide you through the fascinating techniques of filtering and edge detection. Unleash the power of algorithms to transform images, detect edges, and reveal hidden details. Get ready to dive deep into the realm of pixels and gradients. Let's get started!

Tasks to be Implemented:

- Additive noise
- Filtering noise
- Edge detection
- Histograms & CDF
- Image equalization & normalization
- Local & Global thresholding
- Frequency domain filters
- Hybrid images

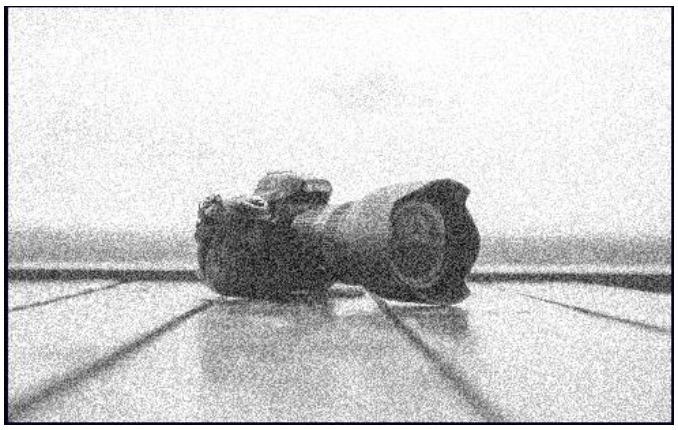
1. Additive Noise:

In the context of image processing, additive noise refers to random variations in brightness or color information across the image. This is the most common type of noise seen in images. This noise can degrade the quality of the image and obscure important details, making tasks like filtering and edge detection more challenging. Therefore, understanding and mitigating additive noise is a crucial part of image processing.

1.1. Types of Noise:

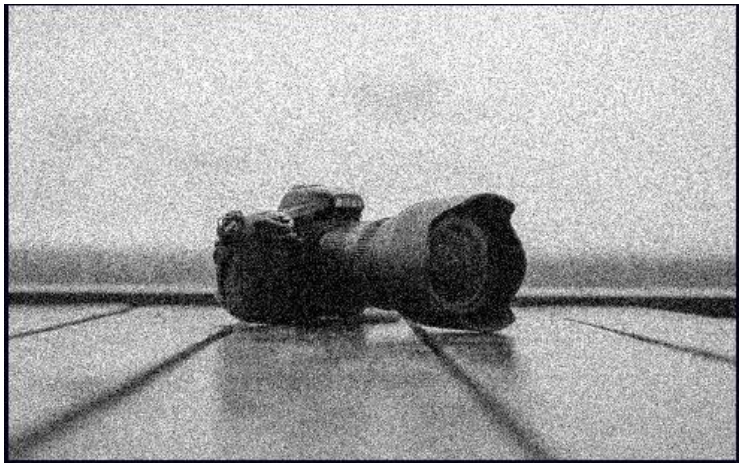
- **Uniform Noise:** a type of noise where random values are added to each pixel in the image uniformly.
- **Gaussian Noise:** a type of noise where random values sampled from a Gaussian (normal) distribution are added to each pixel in the image.
- **Salt & Pepper Noise:** a type of noise that corrupts digital images by randomly replacing some pixels with either the maximum (salt) or minimum (pepper) pixel intensity values.

1.2. Sample results:



1. Uniform Noise

2. Salt & Pepper



3. Gaussian Noise

2. Filtering Noise:

The goal of image filtering is to enhance certain features, remove noise, or extract useful information from the image. Filters can be linear or nonlinear, and they can be designed to perform a wide range of operations, such as blurring, sharpening, edge detection, noise reduction, and more.

2.1. Noise Reduction Filters:

2.1.1. Median Filter:

The median filter is a nonlinear filtering technique used to remove noise from an image while preserving edges and fine details.the median filter replaces each pixel with the median value of its neighboring pixels. It's used to remove salt & pepper noise.

Algorithm:

- Input:
  - Original image
  - Kernel size
- Input Conversion:
  - Ensure that the original image (original_img) is converted into a numpy array for efficient processing.
- Kernel Index Calculation:
  - Calculate the index for the middle element of the filter kernel. This is achieved by dividing the kernel size by 2 (integer division) to determine the number of elements to the left and right of the center.
- Initialize Filtered Image:
  - Create an empty numpy array (data_final) with the same dimensions as the original image to store the filtered data.
- Loop through Image:
  - Iterate through each pixel of the original image:
    - Loop through the rows of the data.
    - Loop through the columns of the data.
    - For each pixel, create an empty list (values) to store the pixel values within the kernel.
- Collect Kernel Values:
  - Within the kernel region around the current pixel, collect the pixel values from the original image:

        - Loop through the kernel size:
            - Calculate the row and column indices for the current position relative to the current pixel.
            - Check if the calculated indices are within the bounds of the original image:
                - If within bounds, append the corresponding pixel value to the values list.
                - If out of bounds, append 0 to the values list.
- Sort and Assign Median:
    - Sort the values list in ascending order.
    - Assign the median value from the sorted values list to the corresponding position in the final filtered data (data_final).
- Return Output:
    - Return the filtered image (filtered_img) as the result of the median filtering operation.

2.1.2. Average Filter:

The average filter, also known as the mean filter, is a type of linear smoothing filter used to reduce noise in an image. It smooths the image by replacing each pixel value with the average value of its neighboring pixels within a defined kernel.

Algorithm:

- **Input:**
    - Original image
    - Kernel size
- **Kernel Initialization:**
    - Create a square kernel of size (kernel_size x kernel_size) filled with ones.
    - Divide each element of the kernel by the total number of elements (i.e., kernel_size^2). This normalization ensures that the sum of the kernel elements equals 1, maintaining the brightness level of the image.
- **Apply Filter:**
    - Apply the computed average filter kernel to the original image using the _apply_filter function.
- **Return Output:**
    - Return the filtered image as the result of the average filtering operation.

2.1.3 Gaussian Filter:

The Gaussian filter smooths an image by convolving it with a Gaussian kernel, which assigns higher weights to central pixels and lower weights to pixels farther away. This weighted averaging results in noise reduction and blur, making the image visually smoother and more aesthetically pleasing.

Algorithm:

- **Input:**
  - Original image
  - Kernel size
  - Frequency response (optional, default value = 255)
- **Parameter Initialization:**
  - Set the standard deviation (sigma) for the Gaussian kernel to 1.
  - Compute the Gaussian kernel using the *_gaussian_kernel* function with the specified kernel size and sigma.
- **Frequency Response Adjustment:**
  - Scale the values of the Gaussian kernel by multiplying them with the given frequency response divided by 255. This ensures that the kernel values are normalized between 0 and 1.
- **Apply Filter:**
  - Apply the computed Gaussian kernel as a filter to the original image using the *_apply_filter* function.
- **Return Output:**
  - Return the filtered image as the result of the Gaussian filtering operation.

2.2 Frequency Domain Filters:

2.2.1 Low Pass Filter:

A low-pass filter smooths an image by attenuating high-frequency noise and details while preserving low-frequency components. This smoothing effect helps improve image quality and prepares the image for further processing or analysis.

Algorithm:

- **Input:**
  - Image to be filtered
  - Smoothing degree
- **Fast Fourier Transform (FFT) of the Image:**
  - Compute the FFT of the input image using fft2() function.
  - Shift the zero frequency component to the center of the spectrum using fftshift().

- **Determination of Center Frequency:**
    - Determine the center coordinates (crow, ccol) of the image.
    - Compute Crow = rows / 2 and ccol = cols / 2 where rows and cols are the dimensions of the image.
- **Cutoff Frequency and Smoothing Degree Adjustment:**
    - Define the cutoff frequency (cutoff_frequency) as a constant value.
    - Adjust the smoothing degree (smoothing_degree) based on the provided parameter.
    - Scale the smoothing degree by dividing it by 25.6 and adding 1.
- **Generation of Gaussian Filter Mask:**
    - Create an empty mask array (mask) of the same dimensions as the image.
    - Iterate over each pixel in the image:
        - Compute the Euclidean distance (distance) of the pixel from the center.
        - Calculate the Gaussian filter value using the distance, cutoff frequency, and smoothing degree.
        - Assign the computed value to the corresponding pixel position in the mask.
- **Filtering in Frequency Domain:**
    - Perform element-wise multiplication of the FFT image (fft_img) with the mask (mask) to obtain the low-pass filtered FFT image (low_pass_fft).
- **Inverse FFT (IFFT) to Obtain Filtered Image:**
    - Perform inverse FFT on the low-pass filtered FFT image using ifft2() and ifftshift() to obtain the spatial domain filtered image (low_pass).
- **Normalization and Conversion to Unsigned 8-bit Integer Format:**
    - Compute the absolute value of the filtered image to remove complex components.
    - Normalize the filtered image using a normalization function (e.g., normalize_image()).
    - Convert the normalized filtered image to the unsigned 8-bit integer format (np.uint8).

2.2.2 High Pass Filter:

a high-pass filter enhances edges and sharp transitions in an image by attenuating low-frequency components while allowing high-frequency components to pass through unchanged. This edge enhancement effect helps improve image contrast and visual clarity.

Algorithm:

- **Input:**
  - Image to be filtered
  - Edge enhancement degree
- **Fast Fourier Transform (FFT) of the Image:**
  - Compute the FFT of the input image using fft2() function.
  - Shift the zero frequency component to the center of the spectrum using fftshift().
- **Determination of Center Frequency:**
  - Determine the center coordinates (crow, ccol) of the image.
  - Compute crow = rows / 2, and ccol = cols / 2, where rows and cols are the dimensions of the image.
- **Cutoff Frequency and Edge Enhancement Degree Adjustment:**
  - Define the cutoff frequency (cutoff_frequency) as a constant value.
  - Adjust the edge enhancement degree (edge_degree) based on the provided parameter. Scale the edge degree by dividing it by 256 and adding 1.
- **Generation of High Pass Filter Mask:**
  - Create an empty mask array (mask) of the same dimensions as the image.
  - Iterate over each pixel in the image:
    - Compute the Euclidean distance (distance) of the pixel from the center.
    - Calculate the high-pass filter value using the distance, cutoff frequency, and edge enhancement degree.
    - Assign the computed value to the corresponding pixel position in the mask.
- **Filtering in Frequency Domain:**
  - Perform element-wise multiplication of the FFT image (fft_img) with the mask (mask) to obtain the high-pass filtered FFT image (high_pass_fft).
- **Inverse FFT (IFFT) to Obtain Filtered Image:**
  - Perform inverse FFT on the high-pass filtered FFT image using ifft2() and ifftshift() to obtain the spatial domain filtered image (high_pass).
- **Normalization and Conversion to Unsigned 8-bit Integer Format:**
  - Compute the absolute value of the filtered image to remove complex components.
  - Normalize the filtered image using a normalization function (e.g., normalize_image()).
  - Convert the normalized filtered image to the unsigned 8-bit integer format (np.uint8).

2.3 Sample Results

Image with salt & pepper noise

after median filter

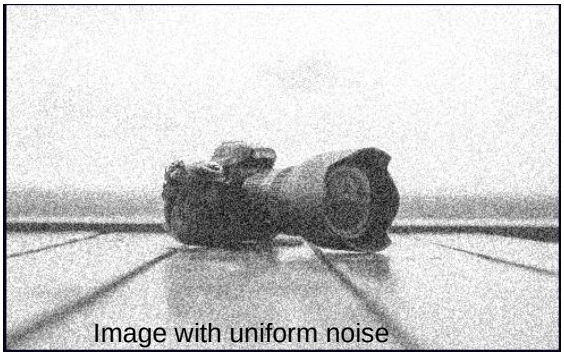
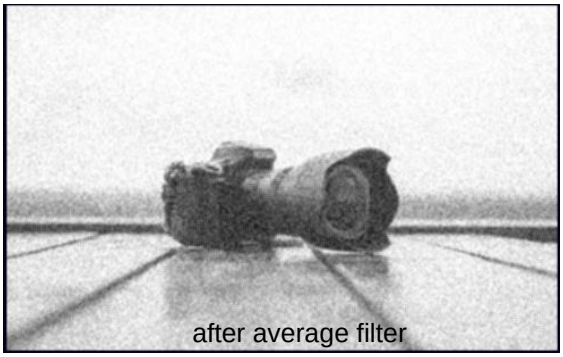


Image with uniform noise



after average filter

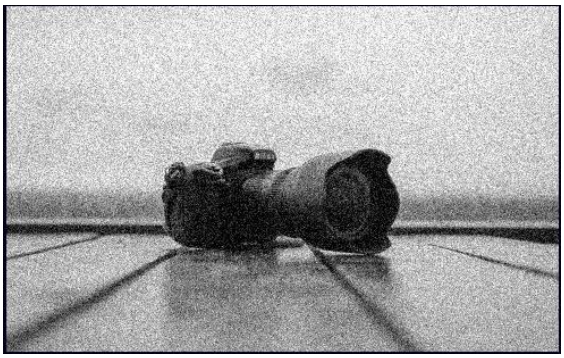
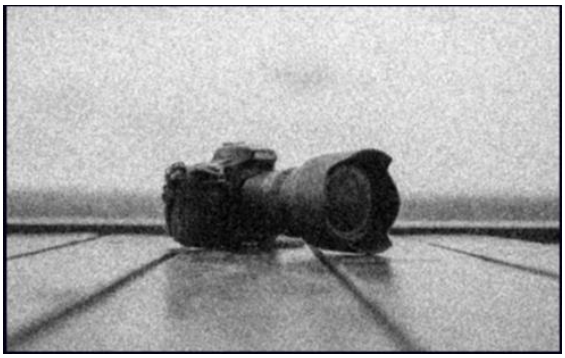


Image with Gaussian noise



After Gaussian filter

Low pass filter

High pass filter

3. Edge Detection

Edge detection is a fundamental concept in image processing that aims to identify and highlight significant changes in intensity or color within an image. Edges represent boundaries or transitions between different regions in an image, such as the boundary between objects, changes in texture, or abrupt changes in illumination.

3.1. Sobel Edge Detector:

The Sobel edge detector is a popular method for edge detection in image processing. It is based on convolving the image with Sobel kernels to compute the gradient approximation, which highlights regions of rapid intensity change.

Algorithm:

- **Input:**
  - Grayscale image
- **Kernel Definition:**
  - Define two Sobel kernels:
    - sobel_x for detecting horizontal edges and sobel_y for detecting vertical edges.
    - Each kernel is a 3x3 numpy array containing specific weightings to compute the gradient approximation in the x and y directions.
- **Convolution Operation:**
  - Convolve the grayscale image with the Sobel kernels using the filter2D function from OpenCV.
  - Apply the Sobel-x kernel (sobel_x) to the grayscale image to compute the gradient approximation in the x direction, resulting in gradient_x.
  - Apply the Sobel-y kernel (sobel_y) to the grayscale image to compute the gradient approximation in the y direction, resulting in gradient_y.
- **Compute Gradient Magnitude:**
  - Compute the gradient magnitude by combining the gradient approximations in the x and y directions.
- **Normalization:**
  - Normalize the gradient magnitude values to the range [0, 255] for display purposes.
  - Scale the gradient magnitude values by multiplying with 255.0 / max_value, where max_value is the maximum value in the gradient magnitude image.

3.2. Roberts Edge Detector:

The Roberts edge detector is a simple and computationally efficient method for edge detection in images. It's suitable for applications where computational resources are limited.

Algorithm:

- **Input:**
  - Grayscale image
- **Kernel Definition:**
  - Define two Roberts kernels: kernel_x for detecting diagonal edges from top-left to bottom-right and kernel_y for detecting diagonal edges from top-right to bottom-left.
  - Each kernel is a 2x2 numpy array containing specific weightings to compute the gradient approximation in the respective diagonal directions.
- **Convolution Operation:**
  - Convolve the grayscale image with the Roberts kernels using the filter2D function from OpenCV.
  - Apply the Roberts-x kernel (kernel_x) to the grayscale image to compute the gradient approximation in the diagonal direction from top-left to bottom-right, resulting in roberts_x.
  - Apply the Roberts-y kernel (kernel_y) to the grayscale image to compute the gradient approximation in the diagonal direction from top-right to bottom-left, resulting in roberts_y.
- **Compute Gradient Magnitude:**
  - Compute the gradient magnitude by combining the gradient approximations from both diagonal directions.
- **Normalization:**
  - Normalize the gradient magnitude values to the range [0, 255] for display purposes.
  - Convert the gradient magnitude values to unsigned 8-bit integers.

3.3. Prewitt Edge Detector:

The Prewitt edge detector is similar to the Sobel edge detector but uses slightly different kernel weightings. It is effective for detecting edges in images and is commonly used in various image processing applications, including feature extraction, object detection, and image segmentation.

Algorithm:

- **Input:**
  - Grayscale image
- **Kernel Definition:**
  - Define two Prewitt kernels: kernel_x for detecting horizontal edges and kernel_y for detecting vertical edges.
  - Each kernel is a 3x3 numpy array containing specific weightings to compute the gradient approximation in the respective directions.
- **Convolution Operation:**
  - Convolve the grayscale image with the Prewitt kernels using the filter2D function from OpenCV.
  - Apply the Prewitt-x kernel (kernel_x) to the grayscale image to compute the gradient approximation in the horizontal direction, resulting in prewitt_x.
  - Apply the Prewitt-y kernel (kernel_y) to the grayscale image to compute the gradient approximation in the vertical direction, resulting in prewitt_y.
- **Compute Gradient Magnitude:**
  - Compute the gradient magnitude by combining the gradient approximations from both directions.
- **Normalization:**
  - Normalize the gradient magnitude values to the range [0, 255] for display purposes.
  - Convert the gradient magnitude values to unsigned 8-bit integers.

3.4 Canny Edge Detector:

The Canny edge detector is widely used for its ability to detect edges accurately and robustly under various conditions. It produces thin, well-defined edges with minimal noise and false detections, making it suitable for tasks such as image segmentation, object recognition, and feature extraction.

Algorithm:

- **Input:**
  - Grayscale image
- **Gaussian Blur:**
  - Apply Gaussian blur to the grayscale image to reduce noise.
  - Use a 5x5 Gaussian kernel with a standard deviation of 0.
  - This step helps to smooth the image and reduce noise.
- **Gradient Calculation:**

    - Compute the gradients of the blurred image using the Sobel operator.
    - Calculate the gradient magnitude and direction at each pixel.
    - The gradient magnitude represents the strength of edges, while the gradient direction indicates the orientation of edges.
- **Non-maximum Suppression:**
    - Perform non-maximum suppression to thin out edges.
    - For each pixel, compare its gradient magnitude with the magnitudes of its neighbors along the gradient direction.
    - Retain the pixel's magnitude if it is the maximum among its neighbors along the gradient direction; otherwise, suppress it.
- **Hysteresis Thresholding:**
    - Apply hysteresis thresholding to distinguish between strong and weak edges.
    - Define low and high threshold values to classify edges.
    - Pixels with gradient magnitudes above the high threshold are considered strong edges, while those between the low and high thresholds are weak edges.
    - Connect weak edges to strong edges to form continuous edge contours.
    - Iterate through the image and set weak edge pixels to strong if they are adjacent to strong edge pixels.

3.5. Comparison

- Effectiveness:
    - Sobel: Effective for detecting edges in images with low to moderate noise levels.
    - Roberts: Simple and computationally efficient but less effective compared to other detectors. Suitable for basic edge detection tasks
    - Prewitt: Similar to Sobel but with slightly different kernel weightings. Effective for detecting edges in images with low noise
    - Canny: Considered one of the most effective edge detectors. It provides precise edge detection with minimal false positives and is robust to noise.
- Computational Complexity:
    - Sobel: Moderate computational complexity due to the convolution operation.
    - Roberts: Simple and computationally efficient due to the small size of the kernels.

    - Prewitt: Similar to Sobel in terms of computational complexity.
    - Canny: Relatively high computational complexity due to multiple stages involving Gaussian smoothing, gradient calculation, non-maximum suppression, and hysteresis thresholding.

- Noise Sensitivity:
    - Sobel: Sensitive to noise, especially at higher noise levels.
    - Roberts: Less sensitive to noise compared to Sobel but may still produce noisy results.
    - Prewitt: Similar to Sobel in terms of noise sensitivity.
    - Canny: Robust to noise due to Gaussian smoothing and hysteresis thresholding, making it suitable for images with moderate to high noise levels.
- Edge Thickness:
    - Sobel: May produce thicker edges compared to other detectors.
    - Roberts: Tends to produce thin edges but may miss some details.
    - Prewitt: Similar to Sobel in terms of edge thickness.
    - Canny: Produces thin, well-defined edges with minimal thickness.
- Edge Continuity:
    - Sobel: May produce fragmented edges, especially in noisy images.
    - Roberts: May produce fragmented edges and miss some continuous features.
    - Prewitt: Similar to Sobel in terms of edge continuity.
    - Canny: Produces continuous, connected edges by linking weak edges to strong edges using hysteresis thresholding.

3.6. Results

1. Sobel from Scratch

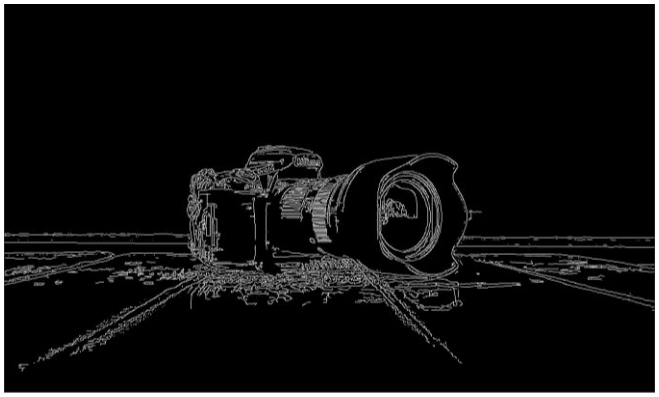
2. Sobel from OpenCv

3. Roberts from scratch

4. Prewitt from scratch

Note: there no implementation for these 2 edge detectors in OpenCv

4. Canny from scratch



5. Canny from OpenCv

4. Thresholding:

thresholding is a simple yet powerful technique used in image processing for segmentation, object detection, and enhancement, among other applications. It provides a basis for separating regions of interest in an image based on intensity levels, facilitating subsequent analysis and interpretation.

Types of thresholding:

- Local Thresholding
- Global Thresholding

4.1. Local Thresholding:

Local thresholding is a versatile technique that improves segmentation accuracy by adapting the threshold dynamically to local intensity variations in an image. It is widely used in various image processing applications, including document analysis, biomedical imaging, and scene analysis.

Algorithm:

- **Input:**
  - Grayscale image
  - Block size
- **Create Meshgrids for Block Indices:**
  - Initialize mesh grids for row and column indices with adjusted step size based on the block size.
  - Ensure that the mesh grids cover the entire image area with non-overlapping blocks.
- **Initialize Local Thresholded Image:**
  - Create an empty array local_threshold with the same dimensions as the grayscale input image.
- **Iterate Over Each Block:**
  - For each block defined by the mesh grids:
    - Extract the block from the grayscale image.
    - Calculate the local threshold for the block using a thresholding method (e.g., mean, median, or a custom method).
    - Apply binary thresholding to the block using the calculated threshold value.
    - Update the corresponding region in the local thresholded image with the thresholded block.

4.2. Global Thresholding:

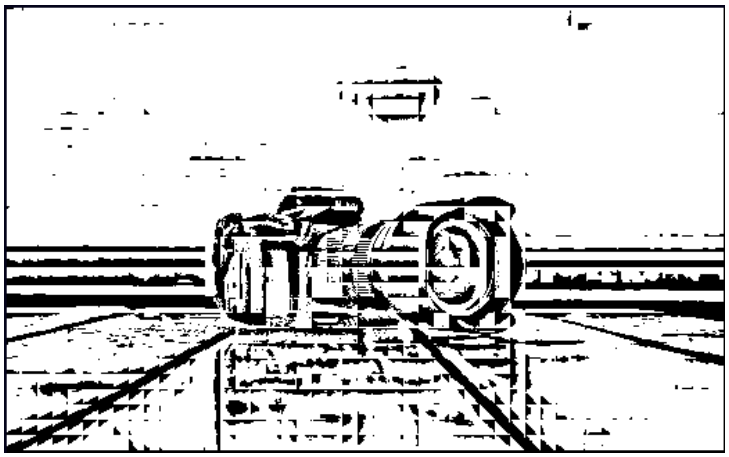
Global thresholding provides a simple and intuitive approach to segmenting images based on intensity levels, making it suitable for a wide range of applications in image analysis and computer vision. However, its effectiveness may be limited in cases where the image exhibits significant variations in illumination or contrast.

Algorithm:

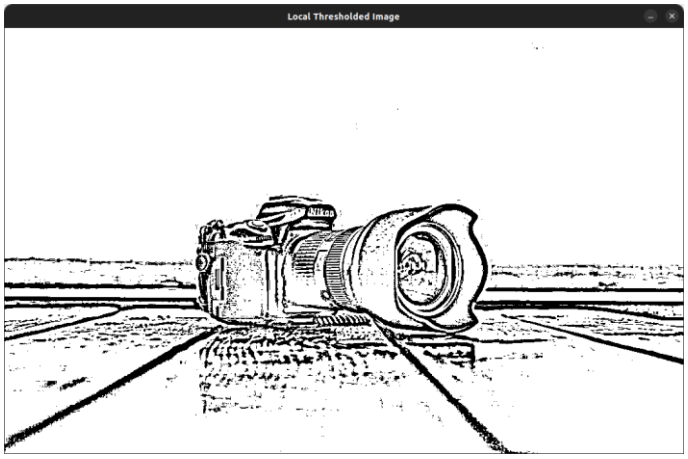
- **Input:**
  - Grayscale image
  - Threshold

- **Thresholding Process:**
    - Initialize an empty binary image binary_image with the same dimensions as the input grayscale image.
    - **For each pixel in the grayscale image:**
        - If the intensity of the pixel is greater than or equal to the threshold:
            - Set the corresponding pixel in binary_image to the maximum intensity value (e.g., 255), indicating foreground.
        - Else:
            - Set the pixel in binary_image
            - to the minimum intensity value (e.g., 0), indicating background.

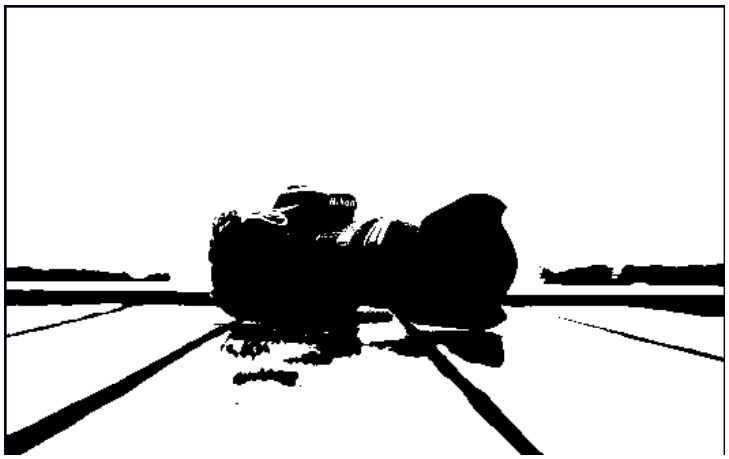
4.3. Results



1. Local thresholding from scratch



2. Local thresholding from Cv2



3. Global Thresholding from scratch

4. Global Thresholding from Cv2

5. Equalization & Normalization:

Image equalization is a technique used in image processing to adjust the contrast of an image by redistributing the intensity values of the pixels. Image normalization is a preprocessing step used to standardize the pixel values of an image to a common scale or range.

5.1. Equalization:

The objective is to ensure that all images in a dataset have similar statistical properties, which can facilitate more effective training of machine learning models.

Algorithm:

- **Input:**
  - Grayscale image
- **Histogram Calculation:**
  - Calculate the histogram of the original grayscale image using np.histogram().
  - Define the number of bins as 256 and the intensity range as [0, 256].
  - Store the histogram values in histogram
- **Cumulative Distribution Function (CDF) Calculation:**
  - Compute the cumulative sum of the histogram values to obtain the cumulative distribution function (CDF).
  - Store the CDF values in cdf
- **Normalization of CDFP:**
  - Normalize the CDF values to the range [0, 255] to ensure compatibility with the pixel intensity values.
  - Subtract the minimum value of the CDF and then scale the values by multiplying by 255 / [max(cdf)−min(cdf)].
  - Store the normalized CDF values in cdf_normalized.
- **Interpolation of CDF Values:**
  - Interpolate the normalized CDF values for each pixel intensity value in the grayscale image.
  - Flatten the grayscale image and use np.interp() to interpolate the CDF values.
  - Reshape the interpolated values to match the original image dimensions.
  - Convert the interpolated values to the data type np.uint8.
  - Store the result in equalized_image

5.2. Normalization:

The objective is to ensure that all images in a dataset have similar statistical properties, which can facilitate more effective training of machine learning models.

Algorithm:

- **Input:**
  - Grayscale image
- **Image Conversion to Floating-Point Format:**
  - Convert the grayscale image to floating-point format using astype(np.float32).
  - Store the result in image_float
- **Determination of Minimum and Maximum Pixel Values:**
  - Compute the minimum and maximum pixel values of the floating-point image.
  - Use np.min() and np.max() functions to find the minimum (min_vall) and maximum (max_vall) values, respectively.
- **Normalization of Image Values:**
  - Normalize the pixel values of the floating-point image to the range [0, 255].
  - Subtract the minimum value (min_vall) from each pixel value, divide by the range (max_val−min_val), and then scale by 255.
  - Multiply the resulting values by 255 to bring them into the range [0, 255].
  - Store the normalized image in normalized_image
- **Conversion to Unsigned 8-bit Integer Format:**
  - Convert the normalized image back to the unsigned 8-bit integer format (np.uint8) using astype(np.uint8).

5.3 Results:

1. Equalized Image

2. Normalized Image

3. Original Image

6. Histogram & CDF:

- *How to get the histogram for each channel in the image?*

  Calculate Histograms for Each Channel:
  - For each channel (R, G, B), calculate its histogram independently.
  - A histogram counts the frequency of occurrence of each intensity level (from 0 to 255) within the channel.

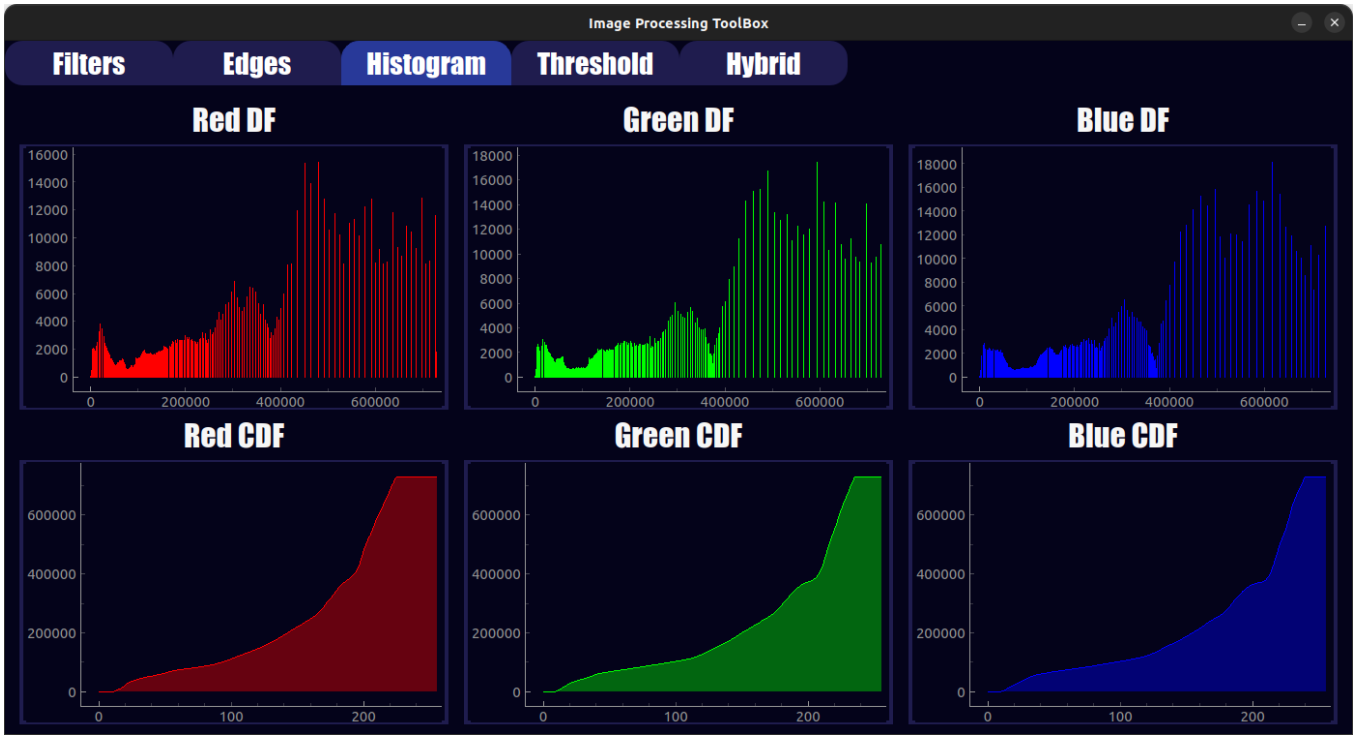
  Bin Counting:
  - Divide the intensity range (0 to 255) into bins. Typically, each bin represents one intensity level.
  - Count the number of pixels in the image that have intensity levels falling within each bin for each channel.

- *After generating the histogram, it's used to get the cumulative distribution of each channel*

  Compute Cumulative Distribution Function (CDF):
  - For each channel, calculate the cumulative sum of histogram values.
  - The cumulative sum at each intensity level represents the cumulative distribution function (CDF) for that channel.

  Normalize the CDF:
  - normalize the CDF to scale the values between 0 and 1. This step ensures that the CDF represents the distribution of pixel intensities relative to the total number of pixels in the image.

6.1 Results:



An example for histograms & CDF

7. Hybrid Images

Hybrid images are where two images with different frequency content are combined to create a single image that produces different interpretations depending on the viewing distance.

Algorithm:

- The provided code implements the generation of hybrid images using two key operations: low-pass filtering and high-pass filtering. These operations are applied to two input images, resulting in filtered versions with distinct frequency content. The filtered images are then combined to generate the hybrid image.

Low-Pass Filtering:

- Applies a low-pass filter to the input image using the Butterworth low-pass filter.
- Computes the 2D discrete Fourier Transform of the image.
- Computes the center of the image and sets the cutoff frequency.
- Creates a 2D mask for the low-pass filter based on the cutoff frequency and smoothing degree.

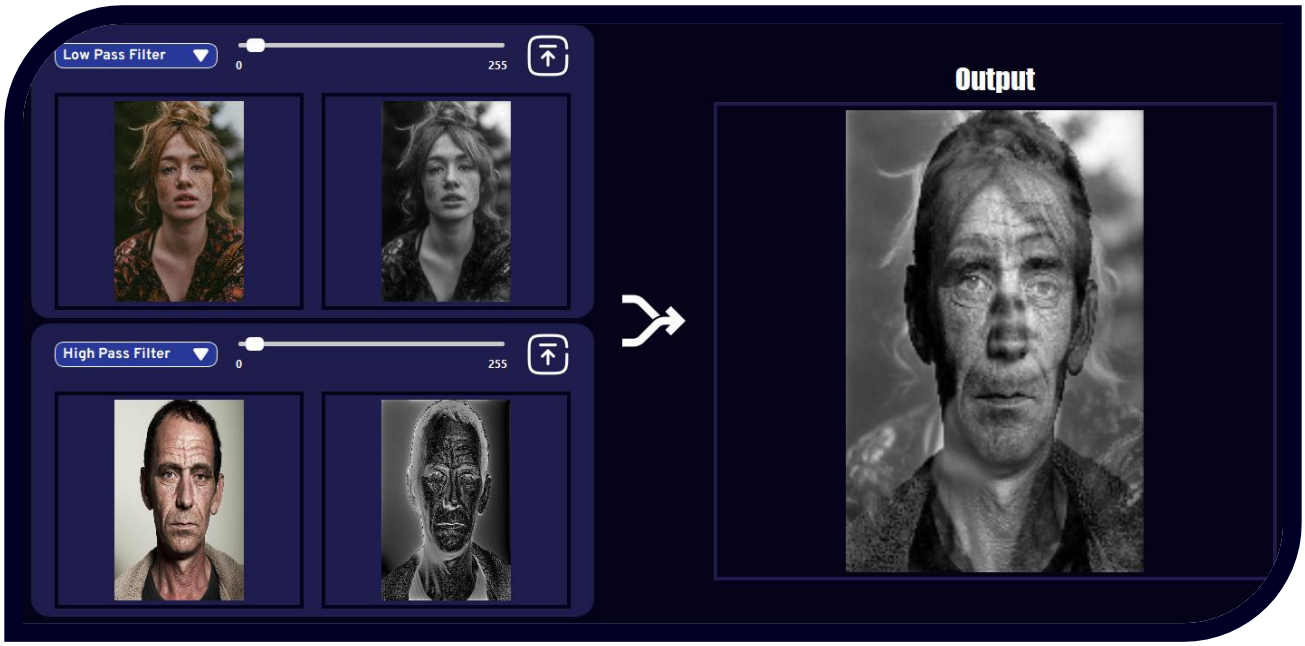
High-Pass Filtering:

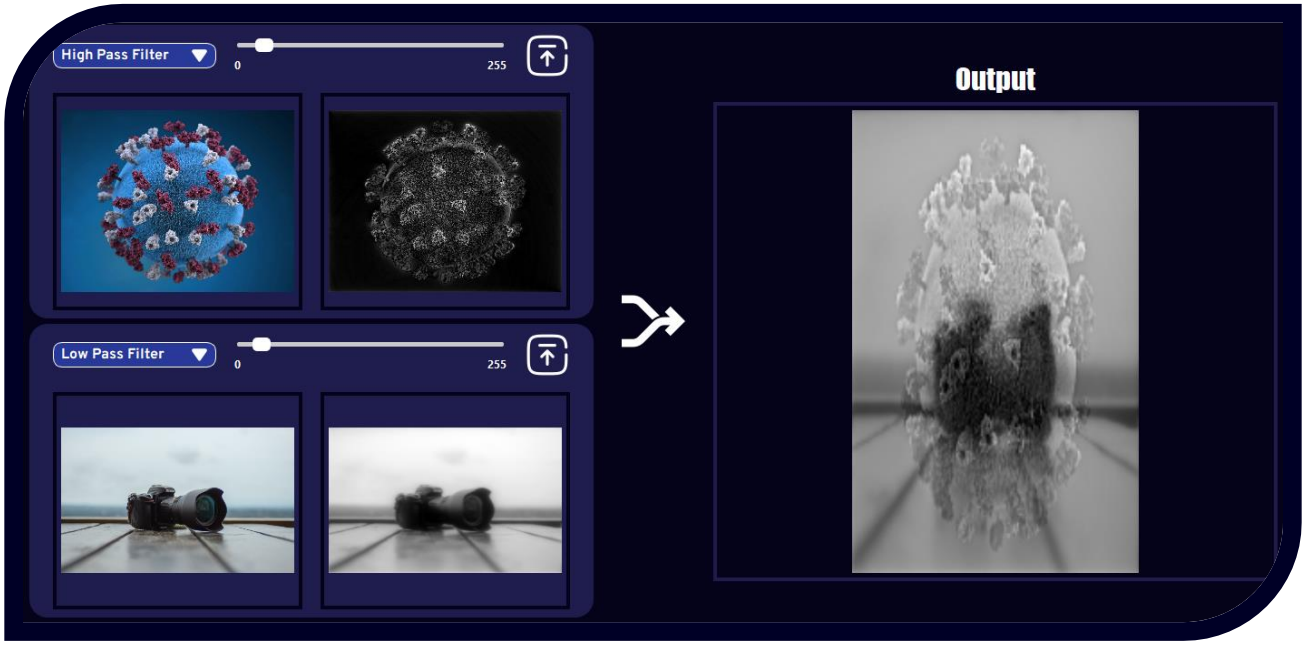
- Applies a high-pass filter to the input image using the Butterworth high-pass filter.
- Computes the 2D discrete Fourier Transform of the image.
- Computes the center of the image and sets the cutoff frequency.
- Creates a 2D mask for the high-pass filter based on the cutoff frequency and edge degree.

Generating Hybrid Image:

- Combines the low-pass and high-pass filtered images to generate the hybrid image.
- Resizes the images to ensure they have the same dimensions.
- Adds the low-pass and high-pass filtered images to obtain the hybrid image.
- Normalizes and converts the hybrid image to 8-bit integer format.

7.1 Results:


Low Pass Filter
0
255
High Pass Filter
0
255
Output


High Pass Filter
0
255
Low Pass Filter
0
255
Output