الاسم : عمرو موسي علي عليوة        الفرقة: الثالثة

الاسم : يوسف مصطفي مسعد محمد    الفرقة: الثالثة

# 1. Introduction

The Pattern Recognition and Image Processing Application is a Python-based desktop tool designed to perform a variety of image processing tasks through a user-friendly graphical user interface (GUI). Built using libraries such as OpenCV, Tkinter, NumPy, Matplotlib, and PIL, the application provides functionalities for manipulating images, applying filters, detecting edges, performing transformations, and conducting morphological operations. This report provides a comprehensive overview of the project, including its objectives, architecture, features, technical implementation, and potential applications.

# 2. Project Objectives

The primary objectives of the project are:

- To create an intuitive GUI for users to load, process, and save images without requiring extensive programming knowledge.
- To implement a wide range of image processing techniques, including noise addition, brightness/contrast adjustment, filtering, edge detection, and morphological operations.
- To demonstrate the application of computer vision concepts such as pattern recognition and image transformation using OpenCV.
- To provide a modular and extensible framework for future enhancements in image processing tasks.

# 3. Project Architecture

The application is structured as a single Python script (`22.py`) that defines a class `ImageProcessingApp` encapsulating all functionality. The architecture can be broken down into the following components:

- **GUI Framework**: Tkinter is used to create the interface, including buttons, sliders, labels, and image display panels.
- **Image Processing Engine**: OpenCV handles core image processing tasks, such as loading images, applying filters, and performing transformations.
- **Supporting Libraries**:
  - NumPy for numerical operations and array manipulations.
  - Matplotlib for generating histograms.
  - PIL (Python Imaging Library) for converting images to a format compatible with Tkinter's display.
- **Main Class**: `ImageProcessingApp` contains methods for initializing the GUI, handling user interactions, and executing image processing operations.

## 3.1. File Structure

- **Filename**: `22.py`
- **Dependencies**:
  - `cv2` (OpenCV)
  - `numpy`
  - `matplotlib.pyplot`
  - `tkinter`
  - `PIL` (Image, ImageTk)
  - `random`

## 4. Key Functionalities

The application supports a comprehensive set of image processing operations, organized into intuitive control panels within the GUI. Below is a detailed breakdown of each functionality.

### 4.1. Image Loading and Display

- **Description**: Users can load an image from their file system using a file dialog (`filedialog.askopenfilename`).
- **Implementation**: The `open_image` method reads the image using `cv2.imread` and stores it as `self.original_image`. The `display_images` method converts images from BGR to RGB (for display compatibility), resizes them if necessary (max size of 500 pixels), and displays them in two Tkinter labels: one for the original and one for the processed image.
- **Use Case**: Supports common image formats (e.g., JPEG, PNG) for processing.

**Code Snippet**:

```
def open_image(self):
    self.image_path = filedialog.askopenfilename()
    if self.image_path:
        self.original_image = cv2.imread(self.image_path)
        self.processed_image = self.original_image.copy()
        self.display_images()
```

### 4.2. Color Mode Toggle

- **Description**: Users can switch between full-color (RGB) and grayscale modes.
- **Implementation**: The `toggle_color` method uses `cv2.cvtColor` to convert the image to grayscale (`COLOR_BGR2GRAY`) and back to BGR for display consistency (`COLOR_GRAY2BGR`). The `is_gray` flag tracks the mode.
- **Use Case**: Grayscale conversion simplifies processing for edge detection and morphological operations.

**Code Snippet**:

```
def toggle_color(self):
    if self.original_image is not None:
        if self.color_var.get() == "gray":
            self.processed_image = cv2.cvtColor(self.original_image,
cv2.COLOR_BGR2GRAY)
```

```
            self.processed_image = cv2.cvtColor(self.processed_image,
cv2.COLOR_GRAY2BGR)
            self.is_gray = True
        else:
            self.processed_image = self.original_image.copy()
            self.is_gray = False
        self.display_images()
```

## 4.3. Noise Addition

- **Description**: Users can add Salt & Pepper, Gaussian, or Poisson noise to simulate real-world image degradation.
- **Implementation**: The `apply_noise` method applies:
    - **Salt & Pepper**: Randomly sets 4% of pixels to 255 (white) or 0 (black).
    - **Gaussian**: Adds noise from a normal distribution with mean 0 and variance 0.1.
    - **Poisson**: Applies noise based on pixel intensity, scaled by unique pixel values.
- **Use Case**: Tests filter robustness or simulates sensor noise in applications like medical imaging.

**Code Snippet** (Gaussian Noise):

```
elif noise_type == "Gaussian":
    row, col, ch = img.shape
    mean = 0
    var = 0.1
    sigma = var**0.5
    gauss = np.random.normal(mean, sigma, (row, col, ch))
    gauss = gauss.reshape(row, col, ch)
    noisy = img + gauss * 50
    self.processed_image = np.clip(noisy, 0, 255).astype(np.uint8)
```

## 4.4. Brightness and Contrast Adjustment

- **Description**: Sliders allow users to adjust brightness (-100 to 100) and contrast (0.1 to 3.0).
- **Implementation**: The `update_brightness_contrast` method uses `cv2.convertScaleAbs` with parameters `alpha` (contrast) and `beta` (brightness) to transform pixel intensities: `output = alpha * input + beta`.
- **Use Case**: Enhances visibility in low-light images or improves contrast for feature detection.

**Code Snippet**:

```
def update_brightness_contrast(self):
    if self.original_image is None:
        return
    img = self.original_image.copy()
    if self.is_gray:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        img = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
    self.processed_image = cv2.convertScaleAbs(img,
alpha=self.contrast_value, beta=self.brightness_value)
    self.display_images()
```

## 4.5. Histogram Analysis

- **Description**: Users can display a histogram or apply histogram equalization.
- **Implementation**:
    - **Show Histogram**: The `show_histogram` method uses `cv2.calcHist` to compute and plot histograms for color (R, G, B channels) or grayscale images using Matplotlib.
    - **Histogram Equalization**: The `histogram_equalization` method equalizes the Y channel in YCrCb color space for color images or directly applies `cv2.equalizeHist` for grayscale images.
- **Use Case**: Histogram analysis reveals pixel intensity distribution, while equalization enhances contrast in low-contrast images.

**Code Snippet** (Histogram Equalization):

```
def histogram_equalization(self):
    if self.processed_image is None:
        return
    img = self.processed_image.copy()
    if len(img.shape) == 3:  # Color image
        ycrcb = cv2.cvtColor(img, cv2.COLOR_BGR2YCrCb)
        ycrcb[:,:,0] = cv2.equalizeHist(ycrcb[:,:,0])
        self.processed_image = cv2.cvtColor(ycrcb,
cv2.COLOR_YCrCb2BGR)
    else:  # Grayscale image
        self.processed_image = cv2.equalizeHist(img)
    self.display_images()
```

## 4.6. Local Transformations (Filters)

- **Description**: Four filters are available: Low Pass, High Pass, Median, and Averaging.
- **Implementation** (in `apply_filter`):
    - **Low Pass**: Uses a 5x5 averaging kernel (`np.ones((5,5), np.float32)/25`) with `cv2.filter2D` to blur the image.
    - **High Pass**: Applies a 3x3 kernel (`[[-1,-1,-1], [-1,9,-1], [-1,-1,-1]]`) to enhance edges.
    - **Median**: Uses `cv2.medianBlur` with a 5x5 kernel to remove impulsive noise.
    - **Averaging**: Applies `cv2.blur` with a 5x5 kernel for smoothing.
- **Use Case**: Filters preprocess images for noise reduction or feature enhancement in applications like object detection.

**Code Snippet** (Median Filter):

```
elif filter_type == "median":
    filtered = cv2.medianBlur(img, 5)
```

## 4.7. Edge Detection

- **Description**: Eight edge detection methods are supported: Laplacian, Gaussian, Sobel (Vertical/Horizontal), Prewitt (Vertical/Horizontal), Canny, and Zero Cross.
- **Implementation** (in `edge_detection`):
    - **Laplacian**: Uses `cv2.Laplacian` to detect edges via second derivatives.
    - **Canny**: Applies `cv2.Canny` with thresholds (100, 200) for robust edge detection.
    - **Sobel/Prewitt**: Use directional kernels for vertical or horizontal edge detection.
    - **Zero Cross**: Implements a custom zero-crossing algorithm based on Laplacian.
- **Use Case**: Edge detection is critical for shape analysis in robotics or medical imaging.

**Code Snippet** (Canny):

```
elif method == "canny":
    edges = cv2.Canny(img, 100, 200)
```

## 4.8. Global Transformations

- **Description**: Supports Hough Line and Circle detection.
- **Implementation**:
    - **Hough Line**: Uses `cv2.HoughLines` to detect lines in Canny edge-detected images, drawing them in red.
    - **Hough Circle**: Uses `cv2.HoughCircles` to detect circles, drawing them in green with red centers.
- **Use Case**: Detects geometric shapes in applications like autonomous driving or industrial inspection.

**Code Snippet** (Hough Circle):

```
def hough_circle_transform(self):
    img = self.processed_image.copy()
    if len(img.shape) == 3 and not self.is_gray:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img = cv2.medianBlur(img, 5)
    circles = cv2.HoughCircles(img, cv2.HOUGH_GRADIENT, 1, 20,
                             param1=50, param2=30, minRadius=0,
maxRadius=0)
    if circles is not None:
        result = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR) if
len(img.shape) == 2 else img.copy()
        circles = np.uint16(np.around(circles))
        for i in circles[0,:]:
            cv2.circle(result, (i[0],i[1]), i[2], (0,255,0), 2)
            cv2.circle(result, (i[0],i[1]), 2, (0,0,255), 3)
        self.processed_image = result
        self.display_images()
```

### 4.9. Morphological Operations

- **Description**: Supports Dilation, Erosion, Opening, Closing, Thinning, Thickening, and Skeletonization with customizable kernel types (rectangle, cross, ellipse, diamond) and sizes (3 to 15).
- **Implementation** (in `morphological_operation`):
    - Uses `cv2.getStructuringElement` for standard kernels or a custom diamond kernel.
    - Implements Zhang-Suen thinning and iterative skeletonization algorithms.
- **Use Case**: Morphological operations are used in binary image processing, such as text recognition or shape analysis.

**Code Snippet** (Skeletonization):

```
elif operation == "skeleton":
    skeleton = np.zeros(img.shape, np.uint8)
    element = cv2.getStructuringElement(cv2.MORPH_CROSS, (3,3))
    done = False
    while not done:
        eroded = cv2.erode(img, element)
        temp = cv2.dilate(eroded, element)
        temp = cv2.subtract(img, temp)
        skeleton = cv2.bitwise_or(skeleton, temp)
        img = eroded.copy()
        zeros = cv2.countNonZero(img)
        done = (zeros == 0)
    result = skeleton
```

### 4.10. Image Saving

- **Description**: Users can save the processed image as JPEG or PNG.
- **Implementation**: The `save_image` method uses `cv2.imwrite` with a file dialog (`filedialog.asksaveasfilename`).
- **Use Case**: Preserves processed results for further use or documentation.

## 5. Technical Details

- **GUI Design**: The interface uses a scrollable control panel (Tkinter `Canvas` with `Scrollbar`) for operations and two `Label` widgets for image display. The layout is divided into left (image display) and right (controls) frames.
- **Performance Optimization**: Images are resized to a maximum of 500 pixels to ensure smooth display on the GUI. OpenCV operations like `filter2D` and `medianBlur` are optimized for performance.
- **Error Handling**: Basic checks prevent operations on unloaded images, and file dialogs ensure valid file paths.
- **Extensibility**: The modular class structure allows easy addition of new filters or operations by extending the `create_widgets` method.

## 6. Practical Applications

The application is versatile and can be applied in various domains:

- **Computer Vision**: Preprocessing images for object detection, facial recognition, or autonomous navigation.
- **Medical Imaging**: Enhancing X-rays or MRI scans using filters and histogram equalization.
- **Industrial Automation**: Detecting defects or shapes in manufacturing using edge detection and Hough transforms.
- **Document Processing**: Applying morphological operations for text extraction or OCR preprocessing.
- **Education**: Teaching image processing concepts in academic settings.

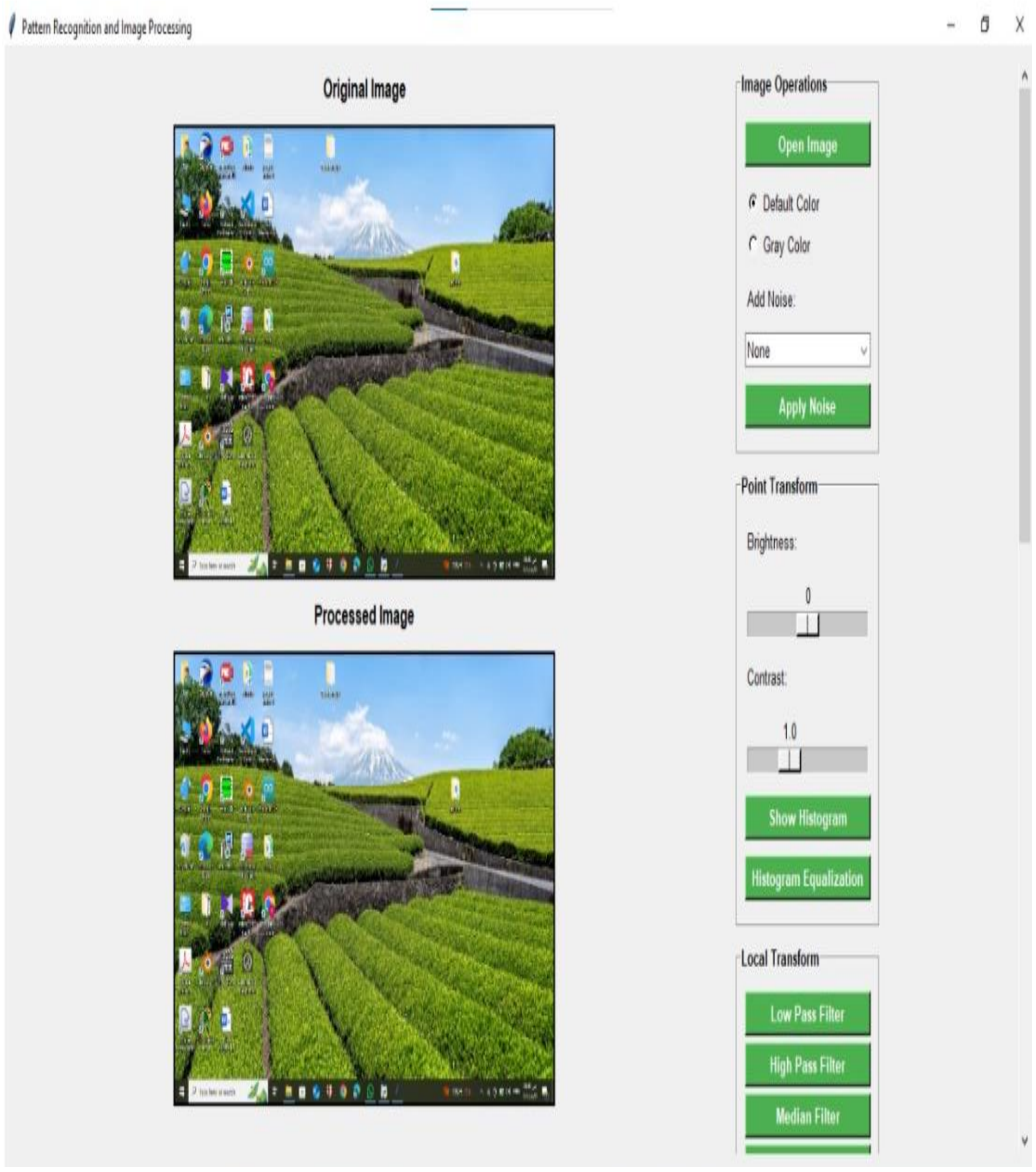
## 7. Limitations

- **Fixed Filter Parameters**: Kernel sizes for filters are fixed (e.g., 5x5), limiting flexibility compared to morphological operations.
- **No Real-Time Preview**: Adjustments like brightness require button clicks, lacking live feedback.
- **Basic Error Handling**: Limited validation for edge cases (e.g., unsupported image formats).
- **No Undo Functionality**: Users cannot revert changes, requiring reloading the original image.

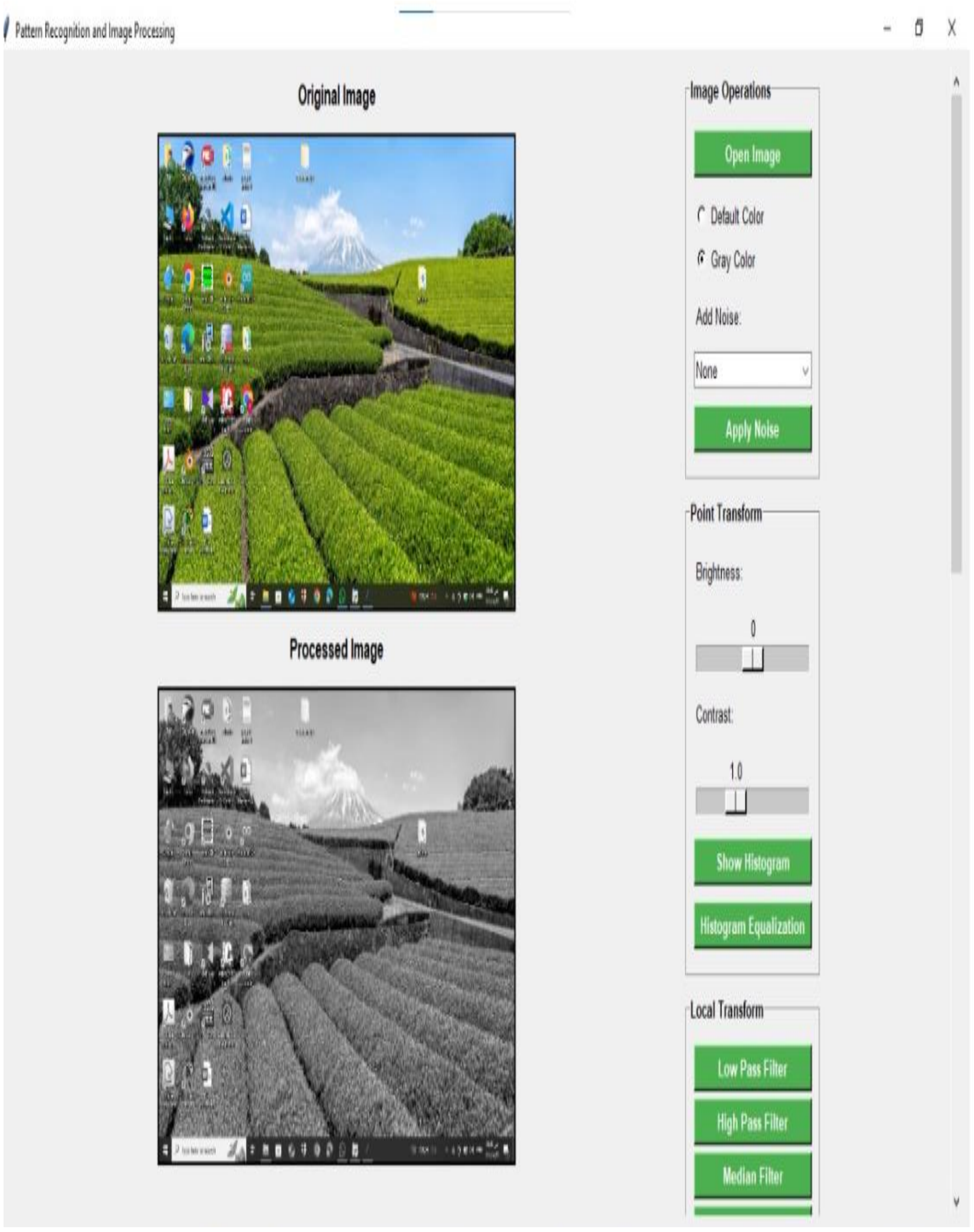
## 8. Conclusion

The Pattern Recognition and Image Processing Application is a robust tool that combines an intuitive GUI with powerful image processing capabilities. Leveraging OpenCV and Tkinter, it provides a comprehensive suite of operations for manipulating images, from basic adjustments to advanced morphological transformations. Its modular design and wide range of functionalities make it suitable for educational, research, and practical applications in computer vision. With suggested improvements, the application could become even more versatile, catering to advanced users and real-time processing needs.

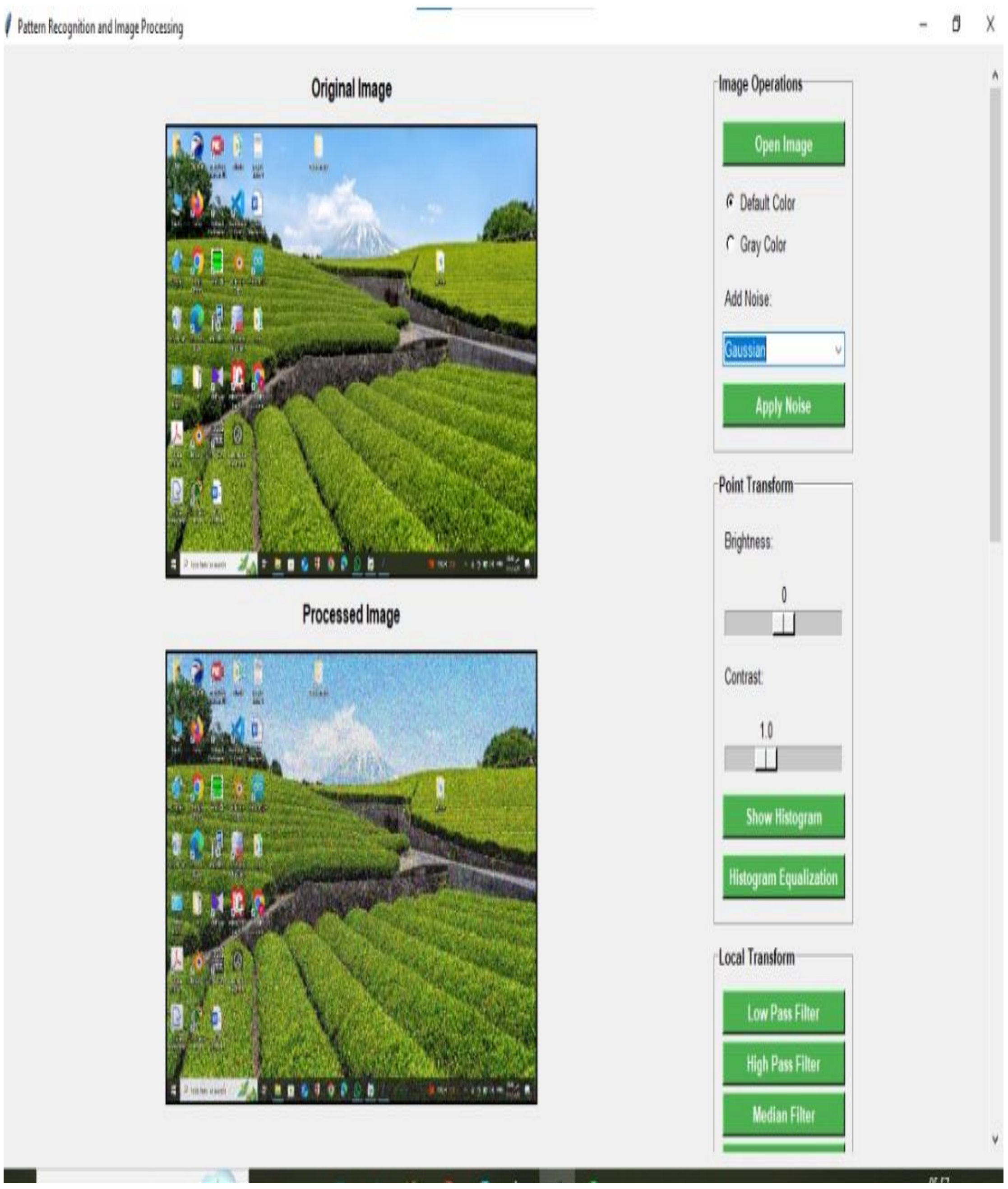
## The original image



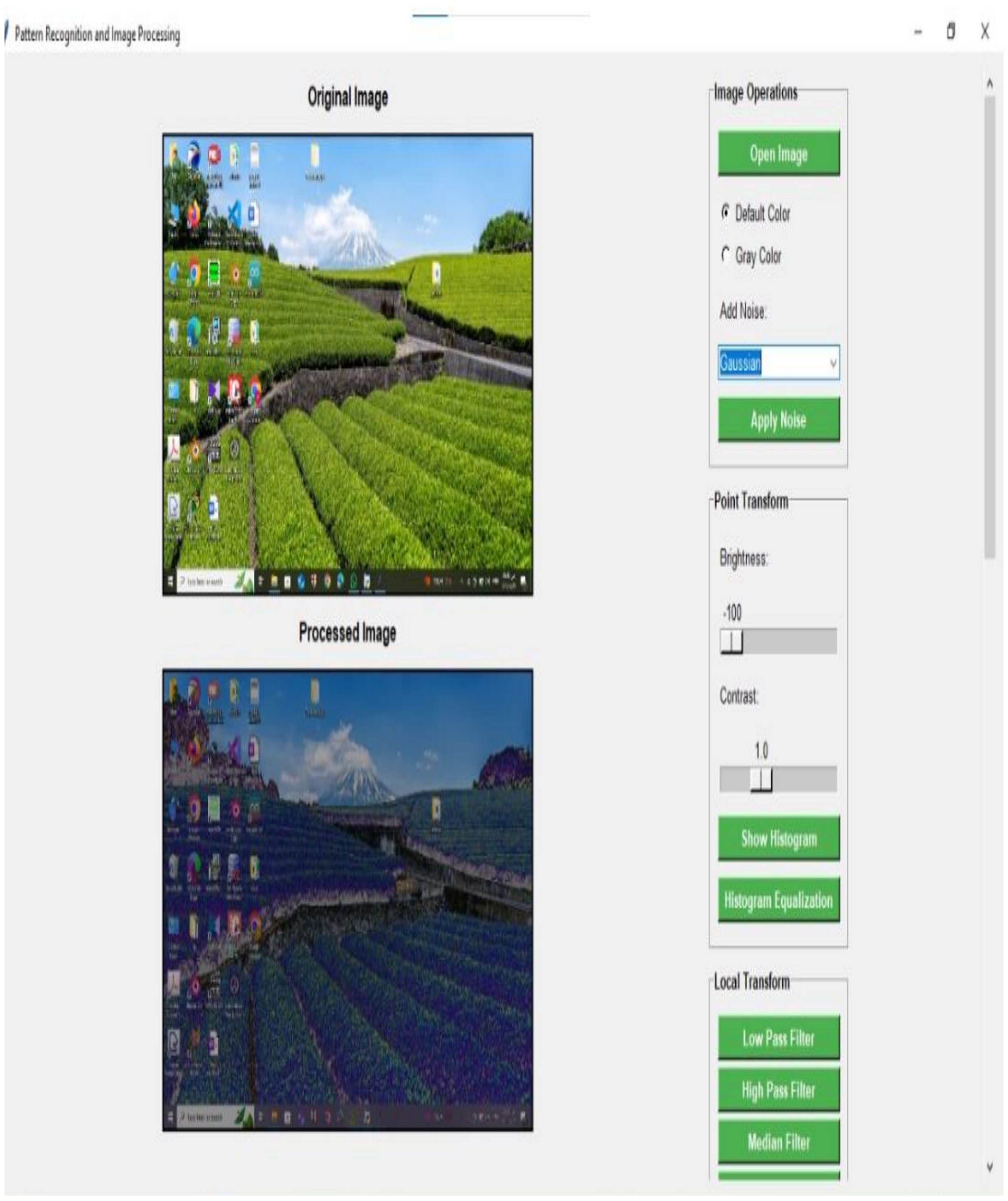
## Gray


Pattern Recognition and Image Processing
Original Image
Processed Image
Image Operations
Open Image
Default Color
Gray Color
Add Noise:
None
Apply Noise
Point Transform
Brightness:
0
Contrast:
1.0
Show Histogram
Histogram Equalization
Local Transform
Low Pass Filter
High Pass Filter
Median Filter

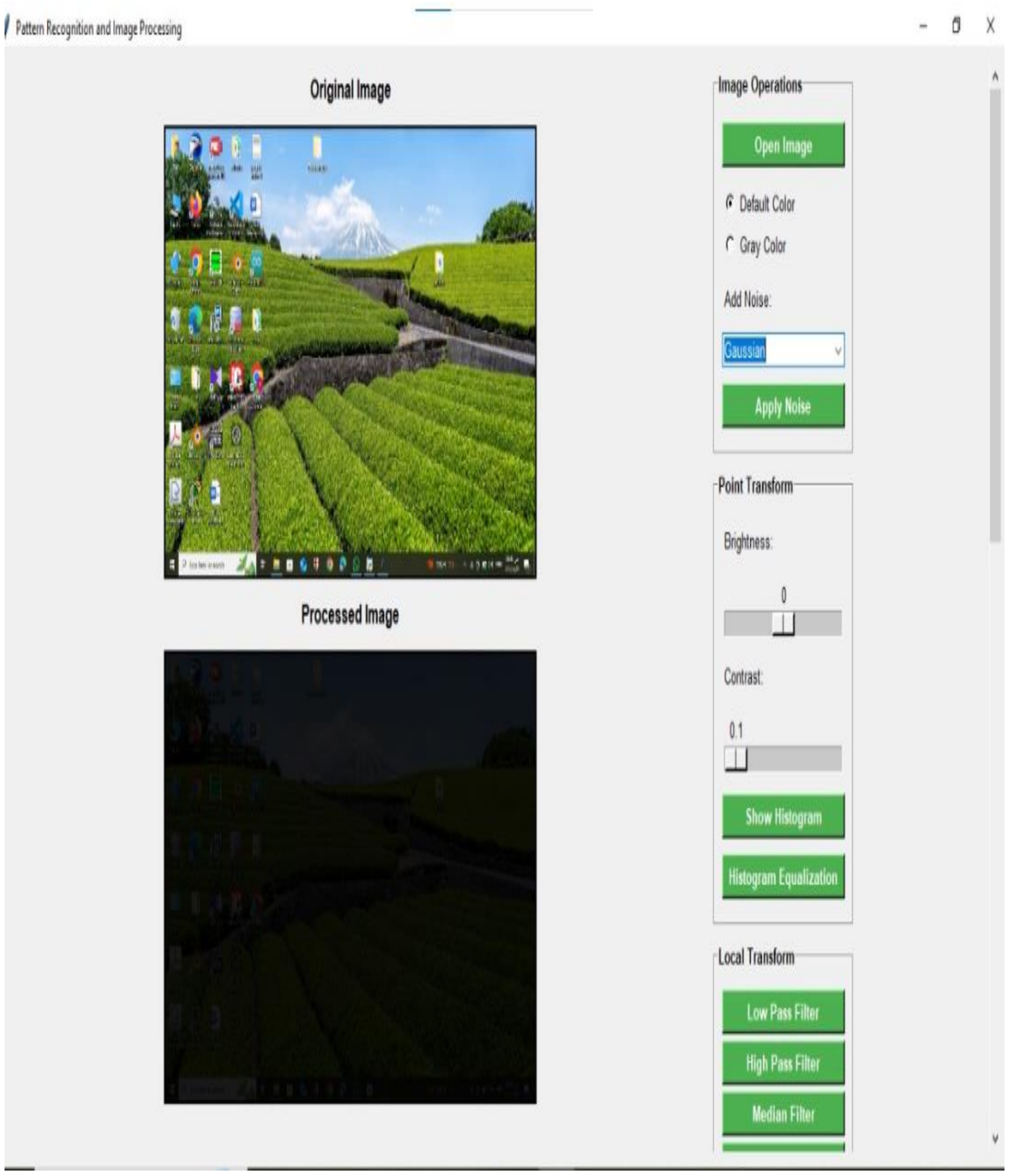
Gray



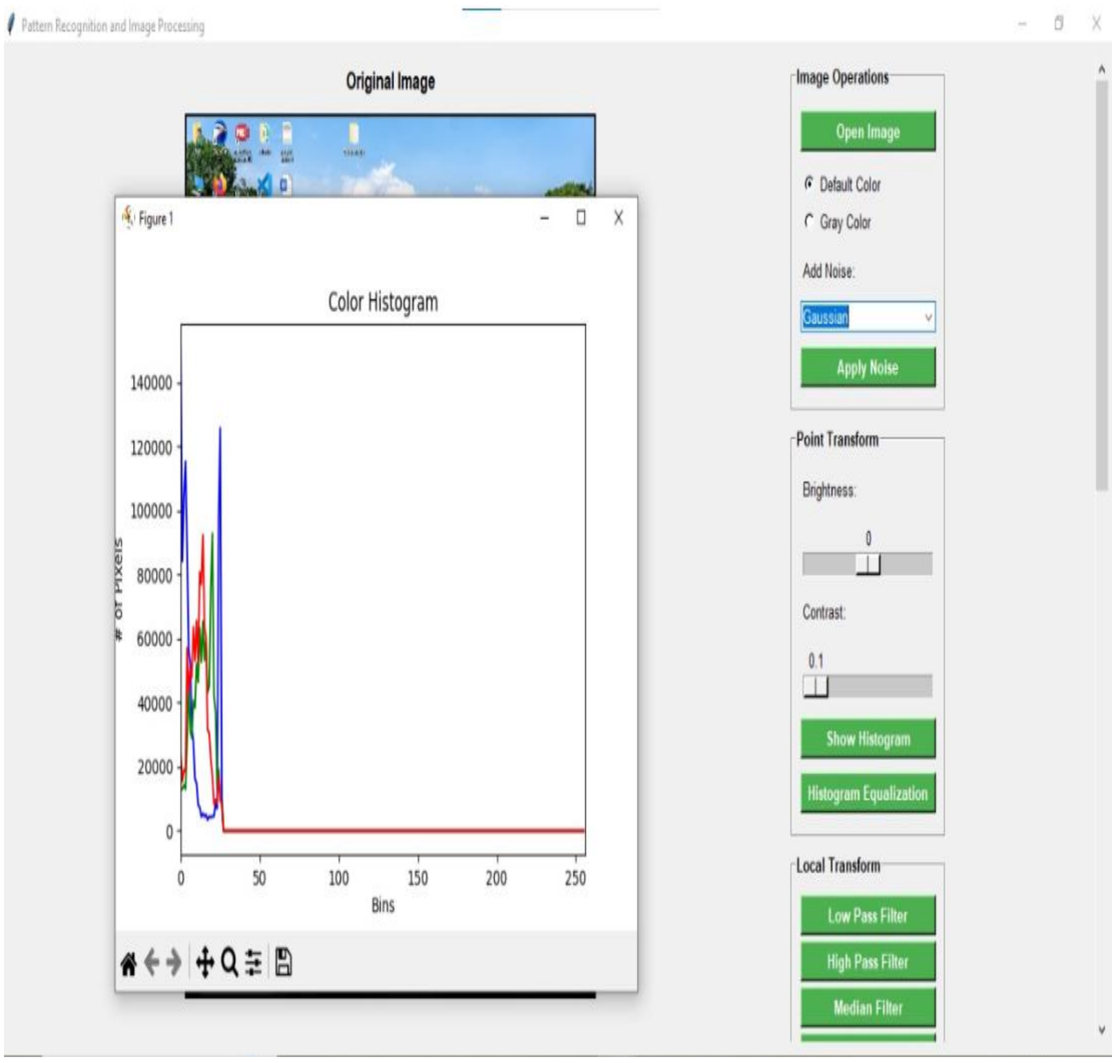
**Add Noise Gaussian**



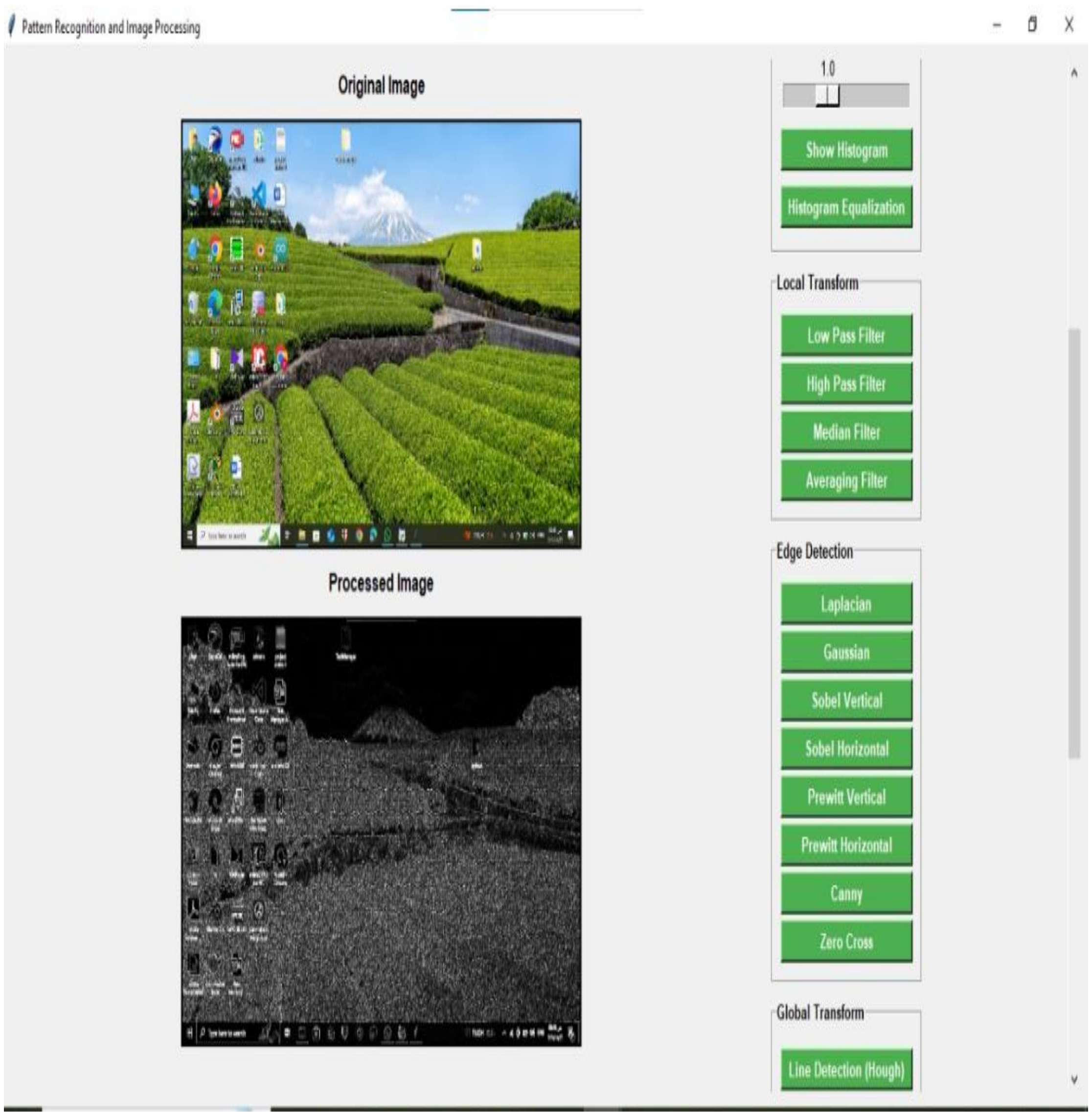
**Brightness = -100**


Pattern Recognition and Image Processing
Original Image
Processed Image
Image Operations
Open Image
Default Color
Gray Color
Add Noise:
Gaussian
Apply Noise
Point Transform
Brightness:
0
Contrast:
0.1
Show Histogram
Histogram Equalization
Local Transform
Low Pass Filter
High Pass Filter
Median Filter

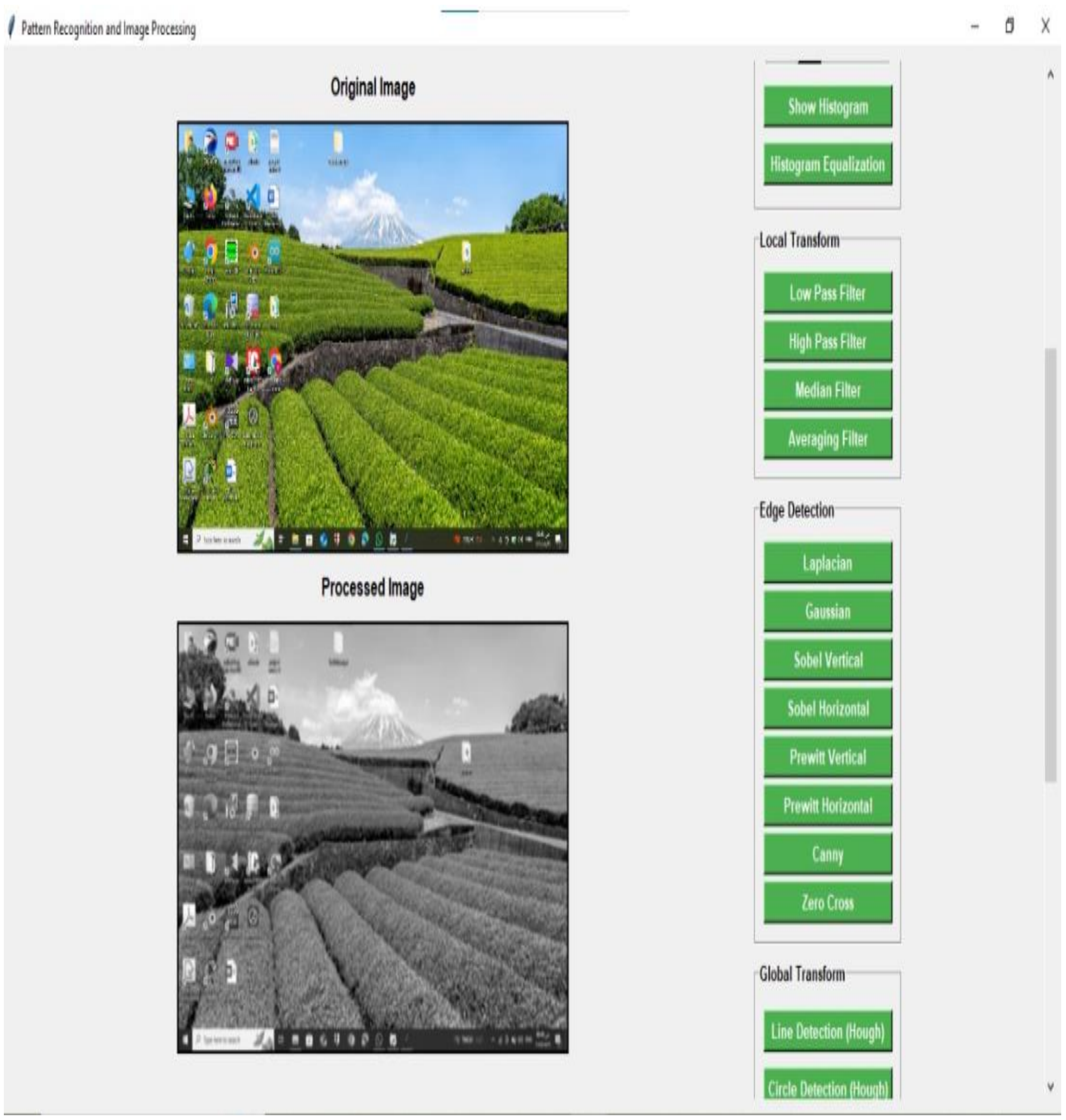
Contrast = 0.1


Pattern Recognition and Image Processing
Original Image
Figure 1
Color Histogram
# of Pixels
Bins
Image Operations
Open Image
Default Color
Gray Color
Add Noise:
Gaussian
Apply Noise
Point Transform
Brightness:
0
Contrast:
0.1
Show Histogram
Histogram Equalization
Local Transform
Low Pass Filter
High Pass Filter
Median Filter

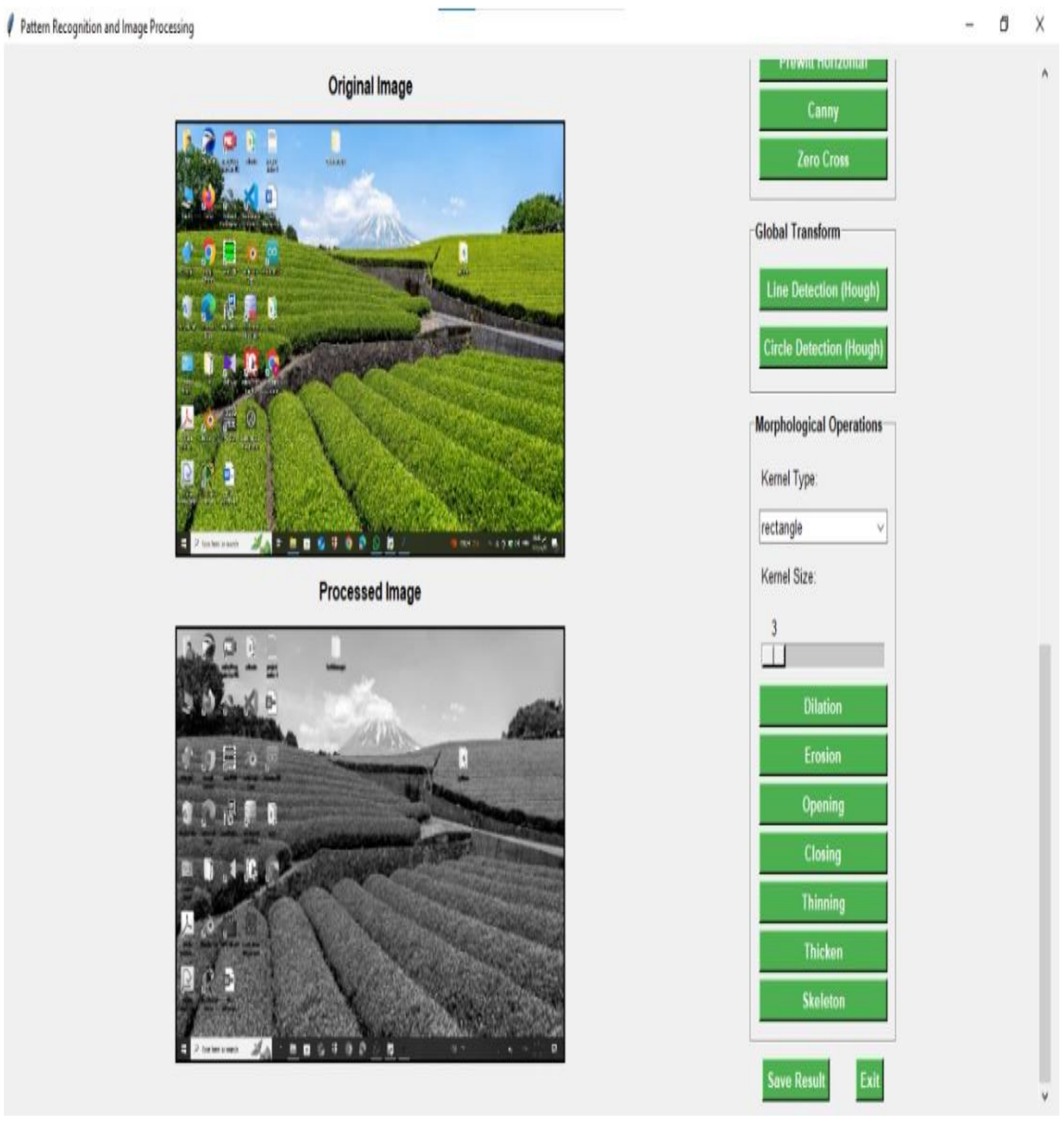
**Histogram**



**Edge Detection: Laplication**



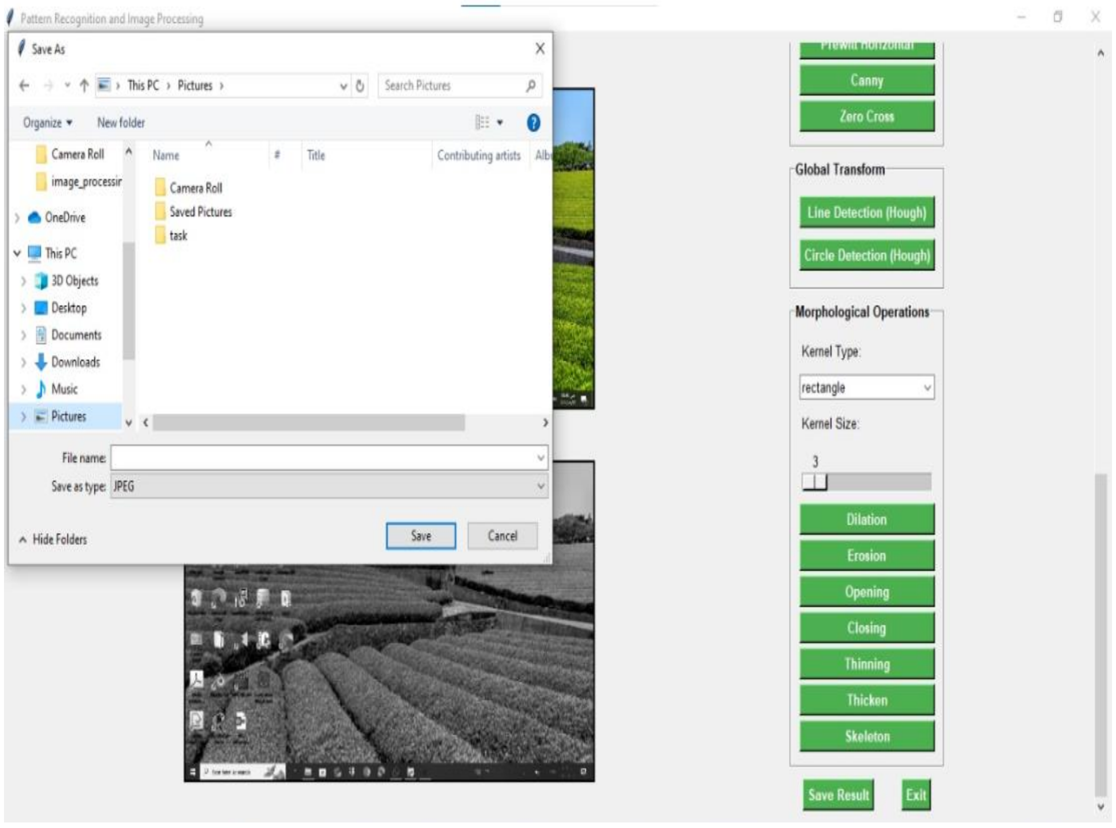
**Local Transform: Low Pass Filter**



Kernel Type: rectangle

Kernel Size: 3

**Morphological Operation : Openning**



**To save picture after changes**