TRAINING REPORT

on

DIGITAL IMAGE PROCESSING USING OPENCV

at

Defence Laboratory, Jodhpur,

Defence Research and Development Organization (DRDO)

Training Period: 08.05.24 to 02.07.24

SUBMITTED BY:	SUBMITTED TO:
Daksh Gehlot	**Sh. Rajendra Kumar Khatri**
Integrated B. Tech. + MBA	Scientist 'F'
Computer Engineering, III Year	
NMIMS, Mumbai	

ACKNOWLEDGEMENT

It brings me immense satisfaction to have concluded my research training at the esteemed **Defence Laboratory (DRDO) in Jodhpur** and to present the comprehensive training report reflecting my learning. I take immense pleasure in thanking **Sh. R.V. Hara Prasad, Director,** Defence Lab Jodhpur, for having permitted me to carry out this project work.

I wish to express my deep sense of gratitude to **Sh. R. K. Khatri, Scientist 'F',** for his guidance and useful suggestions, which helped me in completing the project work on time.

Additionally, I extend my thanks to the HRD Section for providing necessary administrative support.

CERTIFICATE

This is to certify that Mr. Daksh Gehlot, MBA (Tech.) Computer Engineering III Year NMIMS, Mumbai, has undertaken Practical Training on

“Digital Image Processing Using OpenCV”

The training was conducted at Defence Laboratory, Jodhpur, under my supervision and guidance from **08.05.24 to 02.07.24**

Date:
Place: Defence Laboratory, Jodhpur

R. K. Khatri
Scientist ‘F’

ABSTRACT

The project focuses on developing a comprehensive application for digital image processing using Python and OpenCV. The primary objective is to provide users with an easy-to-use platform to perform various image processing tasks, including filtering, segmentation, edge detection, morphological operations, histogram viewing, image transformations, image quality analysis (IQA), and texture analysis. This application aims to facilitate advanced image processing techniques in a user-friendly environment, catering to both novice users and professionals.

The project is structured to follow Agile/Scrum methodology, ensuring an iterative and collaborative development process. The application will be built using Python for its robust libraries and frameworks for powerful image processing capabilities. The development process is divided into distinct phases, each focusing on specific functionalities and ensuring comprehensive coverage of all required features.

Key features of the application include image filtering, segmentation, edge detection, morphological operations, histogram viewing, and image transforms. Image filtering will involve implementing various algorithms such as blurring, sharpening, and smoothing to help in noise reduction and image enhancement. Segmentation techniques like thresholding and region-based segmentation will allow users to isolate and analyze specific parts of an image. Edge detection algorithms, including Sobel and Canny, will help identify boundaries and edges within an image. Morphological operations like dilation, erosion, opening, and closing are essential for shape analysis and pre-processing steps. Histogram viewing will enable users to calculate and display histograms, including histogram equalization, to understand the distribution of pixel intensities. Image transformation techniques such as resizing, rotation, and affine transformations will allow users to manipulate the geometric properties of images.

The project will also include advanced features like Image Quality Analysis (IQA) and texture analysis. IQA will involve researching and implementing metrics such as Structural Similarity Index (SSIM) and Peak Signal-to-Noise Ratio (PSNR) to provide objective evaluations of image quality. Texture analysis will involve implementing feature extraction methods like Local Binary Patterns (LBP).

INDEX

1. ABOUT DEFENCE LABORATORY, JODHPUR

Defence Laboratory, Jodhpur (DLJ) was established in May 1959 to address environmental challenges in desert conditions and their impact on desert warfare. Initially, the laboratory was tasked with undertaking field trials on newly designed or indigenously manufactured weapons and equipment, conducting basic research relevant to the arid zone, physiological studies, radio-wave propagation studies, and solar energy applications. As the laboratory expanded, its responsibilities grew to include operational research, camouflage in the desert, electronics and communications in desert conditions, water management in the desert, transportation and navigation systems, and research on weapons, ammunition, and stores.

Currently, DLJ is engaged in R&D in strategic areas such as camouflage, deception, detection, reconnaissance, performance evaluation and life estimation of weapons and ammunition, clothing, equipment, and stores under desert conditions, integrated water management, soil stabilization, and applications of radioisotopes in defence. The laboratory focuses on three primary thrust areas: Camouflage & Low Observable Technologies, Nuclear Radiation Management & Applications, and the Desert Terrain Development Group (DTDG).

Vision:
To become a centre of excellence in Camouflage, Desert Sciences, and Nuclear Radiation Management Technologies.

Mission:
To achieve excellence and self-reliance in the areas of multispectral camouflage and low observable technologies, provide solutions to desert-related problems, and develop nuclear radiation sensor technologies and applications of radioisotopes.

Camouflage Division:
The Camouflage Division of DLJ is actively engaged in R&D on materials and technologies for camouflage and low observability, focusing on signature management for ground-based and airborne platforms. The division's materials development activities emphasize the synthesis and characterization of advanced materials for camouflage and stealth.

Nuclear Radiation Management & Application (NRMA):
NRMA Division pursues R&D in radiation detectors, dosimeters, dose rate meters, and area monitors. Facilities have been established for testing and calibration of radiation monitors, including thermoluminescent dosimeters-based personnel monitoring services. These facilities are accredited by national bodies like NABL and BARC. The division also conducts training courses for armed forces personnel in nuclear defence and radiological safety, and its scientists are active in national committees related to nuclear defence and safety.

DTDG Division:

DTDG Division engages in R&D in water, soil, and heat management to enhance troop efficiency under harsh desert conditions. The division pursues both basic and applied research, leading to the development of technologies related to water, heat, and terrain management for military applications.

With a vision to empower India with cutting-edge defence technologies, DRDO's mission is to achieve self-reliance in critical defence technologies and systems while equipping the armed forces with state-of-the-art weapon systems and equipment as per the requirements of the three services. DRDO has successfully developed and produced strategic systems and platforms such as the Agni and Prithvi series of missiles, the light combat aircraft Tejas, the multi-barrel rocket launcher Pinaka, the air defence system Akash, and a wide range of radars and electronic warfare systems, significantly enhancing India's military capabilities.

Formed from the amalgamation of the Technical Development Establishments (TDEs) of the Indian Army, the Directorate of Technical Development & Production (DTDP), and the Defence Science Organisation (DSO), DRDO has grown into a multidisciplinary organization. Today, DRDO operates a network of around 41 laboratories and 5 DRDO Young Scientist Laboratories (DYSLs), developing defence technologies in aeronautics, armaments, electronics, combat vehicles, engineering systems, instrumentation, missiles, advanced computing and simulation, special materials, naval systems, life sciences, training, information systems, and agriculture. Several major projects for the development of missiles, armaments, light combat aircraft, radars, and electronic warfare systems are underway, with significant achievements already made in many areas.

2. INTRODUCTION

Digital image processing is a field that involves the manipulation and analysis of digital images using computer algorithms. This technology has a wide range of applications including medical imaging, remote sensing, industrial inspection, and artistic effects. By converting images into digital form, various operations can be performed to enhance image quality, extract useful information, or generate new insights that are not apparent in the raw data.

The importance of digital image processing lies in its ability to improve visual information for human interpretation, and to prepare images for further analysis and processing by automated systems. Techniques such as image filtering, segmentation, edge detection, and morphological operations are crucial in numerous practical applications, from diagnosing diseases in medical images to enhancing security systems through object detection and recognition.

At its core, an image is a representation of visual information, typically captured and stored in a digital format. In the context of digital image processing, an image is composed of a grid of pixels, each pixel representing a specific color or intensity value. These values can be manipulated using various algorithms to achieve desired outcomes, such as enhancement, analysis, or transformation.

This project aims to develop a comprehensive application for digital image processing using Python and OpenCV. The application will include a variety of features such as image filtering, segmentation, edge detection, morphological operations, histogram viewing, and image transforms. Additionally, advanced functionalities like Image Quality Analysis (IQA), texture analysis, and image steganography will be implemented to provide a robust toolkit for image analysis and manipulation.

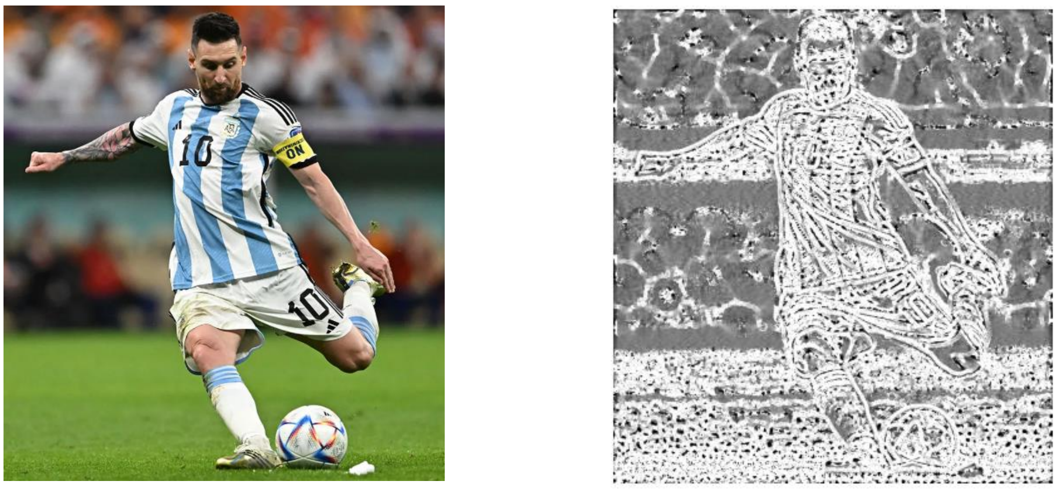


Figure 1: (a) Colored Image (b) Textured Image

2.1 FUNDAMENTALS OF DIGITAL IMAGE

Digital images are a fundamental aspect of modern technology, found in numerous applications ranging from photography and television to medical imaging and remote sensing. Understanding the fundamentals of digital images involves exploring their structure, representation, and the processes used to manipulate them.

1. **Definition of a Digital Image**

A digital image is a representation of a two-dimensional scene or object, stored in a digital format. It is composed of individual elements called pixels (short for picture elements), which are arranged in a grid. Each pixel represents a small portion of the image and contains information about its color and intensity.

2. **Pixel and Resolution**

Pixel:

- The smallest unit of a digital image.
- Contains color and brightness information.
- Represented as a point in a grid, with its position defined by its row and column in the image matrix.

Resolution:

- Describes the detail an image holds.
- Defined by the number of pixels along the width and height of the image (e.g., 1920x1080 pixels).
- Higher resolution means more pixels and finer detail.

3. **Color Models and Bit Depth**

Color Models:

- **Grayscale:** Uses shades of gray, with each pixel represented by a single value indicating its intensity (0 for black, 255 for white in an 8-bit image).
- **RGB (Red, Green, Blue):** Each pixel is represented by three values corresponding to the intensity of red, green, and blue. These primary colors combine to create a broad spectrum of colors.
- **CMYK (Cyan, Magenta, Yellow, Black):** Used in color printing, where colors are created by combining different amounts of these four inks.
- **HSV (Hue, Saturation, Value):** Represents colors in terms of their hue (color type), saturation (color intensity), and value (brightness).

Bit Depth:

- Indicates the number of bits used to represent the color of a single pixel.
- **1-bit:** Binary image with two colors (black and white).
- **8-bit:** 256 shades of gray or colors.
- **24-bit:** True color, with 8 bits for each RGB component, allowing for 16.7 million possible colors.

4. **Image Formats**

Different file formats are used to store digital images, each with its own characteristics:

- **JPEG (Joint Photographic Experts Group):** Commonly used for photographic images, with lossy compression to reduce file size.
- **PNG (Portable Network Graphics):** Supports lossless compression and transparency, suitable for web graphics.
- **GIF (Graphics Interchange Format):** Limited to 256 colors, supports simple animations.
- **BMP (Bitmap):** Uncompressed, large file size, high quality.
- **TIFF (Tagged Image File Format):** Flexible format supporting various compression methods, widely used in professional photography and publishing.

5. **Image Acquisition and Sampling**

Image Acquisition:

- The process of capturing an image using devices like digital cameras, scanners, or sensors.
- Involves converting light into electrical signals, which are then digitized to form the image.

Sampling:

- The process of converting a continuous image into a discrete digital image.
- Involves selecting sample points (pixels) at regular intervals from the continuous image.
- The spacing of these sample points affects the resolution and quality of the digital image.

6. **Quantization**

Quantization is the process of mapping the continuous range of colors or intensity values into a finite range of discrete levels. This is necessary because digital systems can only store a finite number of values. The number of levels used in quantization affects the image quality:

- **High quantization levels:** More precise representation of colors or intensities, but larger file sizes.

- **Low quantization levels:** Reduced file sizes but can introduce quantization errors, such as banding in smooth gradients.

7. **Image Compression**

Compression reduces the file size of digital images, making them easier to store and transmit. There are two main types:

- **Lossy Compression:** Reduces file size by discarding some image information, which can lead to a loss in image quality (e.g., JPEG).
- **Lossless Compression:** Reduces file size without losing any image information, preserving the original quality (e.g., PNG, TIFF).

8. **Image Processing Operations**

Various operations can be performed on digital images to enhance, analyze, or transform them:

- **Filtering:** Smoothing, sharpening, or edge detection to enhance features.
- **Segmentation:** Dividing an image into regions or objects of interest.
- **Morphological Operations:** Erosion, dilation, opening, and closing to process shapes within an image.
- **Transformations:** Scaling, rotating, or translating images.
- **Histogram Equalization:** Enhancing the contrast of an image by adjusting the intensity distribution.

Understanding these fundamentals is crucial for anyone working with digital images, whether for simple editing tasks or complex image analysis and processing. The knowledge of how images are structured, represented, and manipulated forms the foundation for more advanced techniques and applications in the field of digital image processing.

2.2 PHASES OF DIGITAL IMAGE PROCESSING

1. **Image acquisition:** This involves capturing an image using a digital camera or scanner, or importing an existing image into a computer.

2. **Image enhancement**: This involves improving the visual quality of an image, such as increasing contrast, reducing noise, and removing artifacts.

3. **Image restoration:** This involves removing degradation from an image, such as blurring, noise, and distortion.

4. **Image segmentation:** This involves dividing an image into regions or segments, each of which corresponds to a specific object or feature in the image.

5. **Image representation and description:** This involves representing an image in a way that can be analysed and manipulated by a computer, and describing the features of an image in a compact and meaningful way.

6. **Image analysis:** This involves using algorithms and mathematical models to extract information from an image, such as recognizing objects, detecting patterns, and quantifying features.

7. **Image synthesis and compression:** This involves generating new images or compressing existing images to reduce storage and transmission requirements.

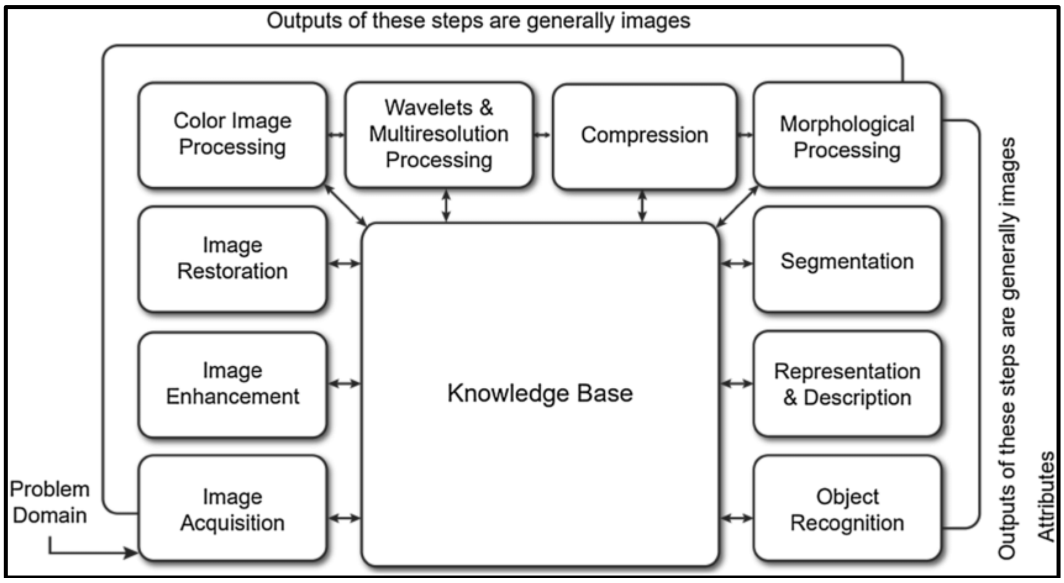


Figure 2: Phases of Digital Image Processing

2.3 Modules Imported

- **NumPy**: NumPy is a fundamental package for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

- **OpenCV (Open-Source Computer Vision Library)**: OpenCV is a library of programming functions mainly aimed at real-time computer vision tasks. It provides tools and algorithms for tasks such as image and video manipulation, object detection, face recognition, and more.

- **Matplotlib**: Matplotlib is a plotting library for Python that provides a MATLAB-like interface. It allows you to create static, animated, and interactive visualizations in Python. Matplotlib integrates well with NumPy and can be used to create publication-quality figures.

- **Scikit-learn's LinearSVC (Support Vector Classification)**: LinearSVC is a part of scikit-learn (sklearn), a machine learning library for Python. It implements Support Vector Machines (SVMs) for classification tasks. LinearSVC specifically provides a linear SVM classifier, which is useful for tasks like texture analysis and classification.

- **Scikit-image's feature module for Local Binary Pattern (LBP)**: Scikit-image (skimage) is an image processing library for Python that builds upon NumPy, SciPy, and Matplotlib. The feature module in scikit-image includes methods for extracting features from images, such as Local Binary Pattern (LBP). LBP is a texture descriptor used for texture analysis in images.

3. SMOOTHENING IMAGES

Smoothening, also known as blurring, is a common image processing technique used to reduce noise and detail in images. This process helps in minimizing small fluctuations in pixel values, thereby making the overall image appear softer and more visually pleasing. It is often a preliminary step in various image processing pipelines, especially in tasks like object detection, edge detection, and texture analysis. There are several techniques for smoothening images, each with its own advantages and use cases. Some of the most commonly used techniques include:

3.1 2D Filtering

In the context of smoothening images, 2D filtering is a technique where a filter (or kernel) is applied to the image to perform convolution operations. This method involves sliding the filter across the image and computing the sum of the products of the filter coefficients and the corresponding image pixels. This operation produces a new image where each pixel value is influenced by its neighbors, resulting in a smoothened effect. Here, we use a **5 x 5 kernel** as a filter and **cv2.filter2D() function.**

```
kernel = np.ones((5,5),np.float32)/25
filtered_img = cv2.filter2D(img,-1,kernel)
```

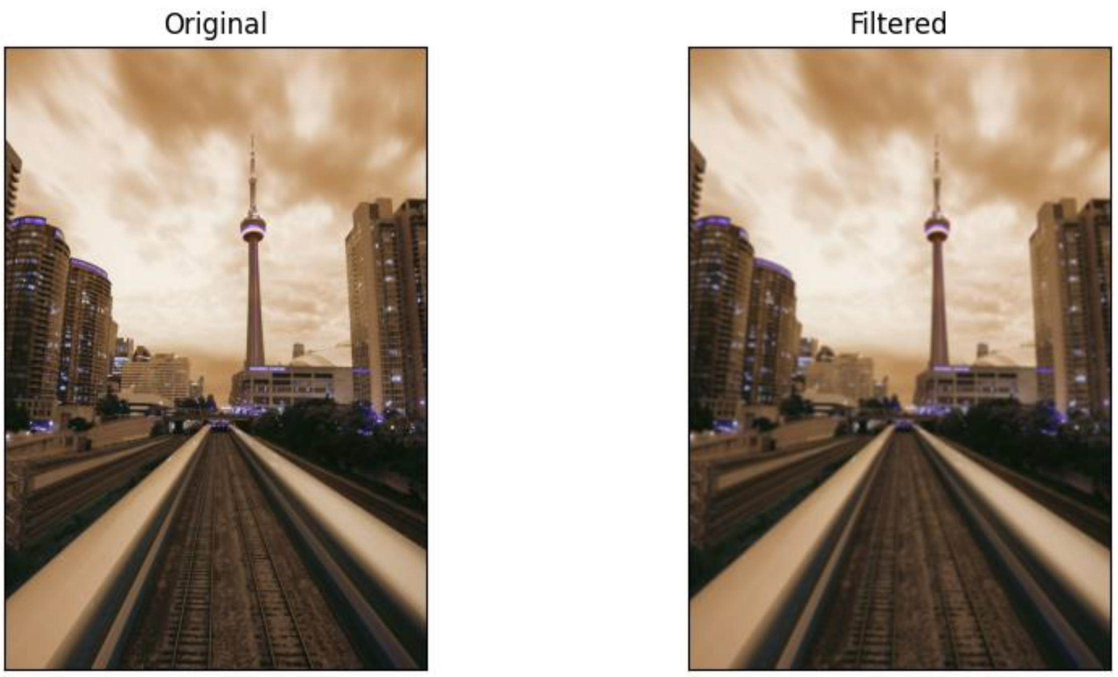


Figure 3: 2D Filtering of an Image. We can see that the filtered image is smoothened.

3.2 Average (Box) Blur

Average blur, or box blur, is one of the simplest smoothening methods. It replaces each pixel's value with the average value of its neighboring pixels within a defined kernel or window. This method effectively reduces high-frequency noise but can result in a loss of fine details and introduce artifacts like edge distortion. We use **cv2.blur() function and 5 x 5 kernel**.

```
blur = cv2.blur(img,(5,5))
```

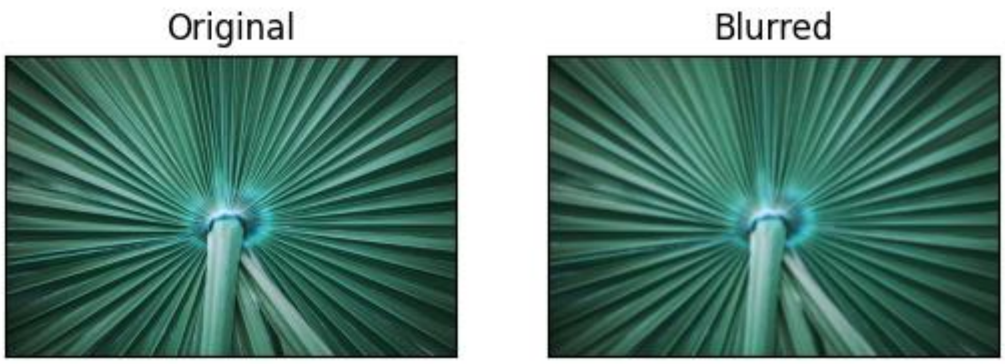


Figure 4: Average blur effect on image

3.3 Gaussian Blur

Gaussian blur is a widely used smoothening technique that applies a Gaussian function to the image. This results in a blurring effect where the intensity of each pixel is recalculated based on a weighted average of its neighboring pixels, with closer pixels given higher weights. The standard deviation (sigma) of the Gaussian function controls the amount of blur; a higher sigma value results in a more pronounced blur. It can be implemented using **cv2.GaussianBlur() and 5 x 5 kernel.**

```
blur = cv.GaussianBlur(img,(5,5),0)
```

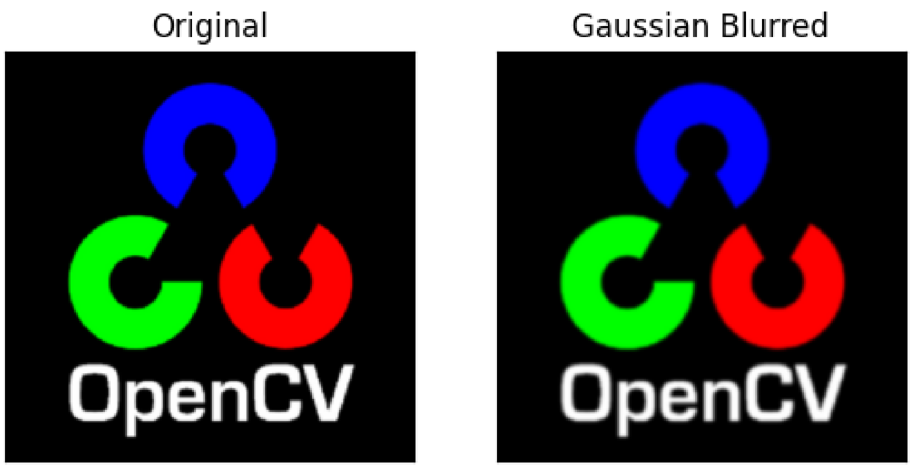


Figure 5: Gaussian blur on an image

3.4 Median Blur

Median blur replaces each pixel's value with the median value of its neighboring pixels within a defined window. This method is particularly effective in removing salt-and-pepper noise from images while preserving edges better than average or Gaussian blur. It is especially useful in scenarios where the image contains outliers or isolated noise points. Here, we use **cv2.medianBlur() function with a 5 x 5 kernel.**

```
median = cv.medianBlur(img,5)
```

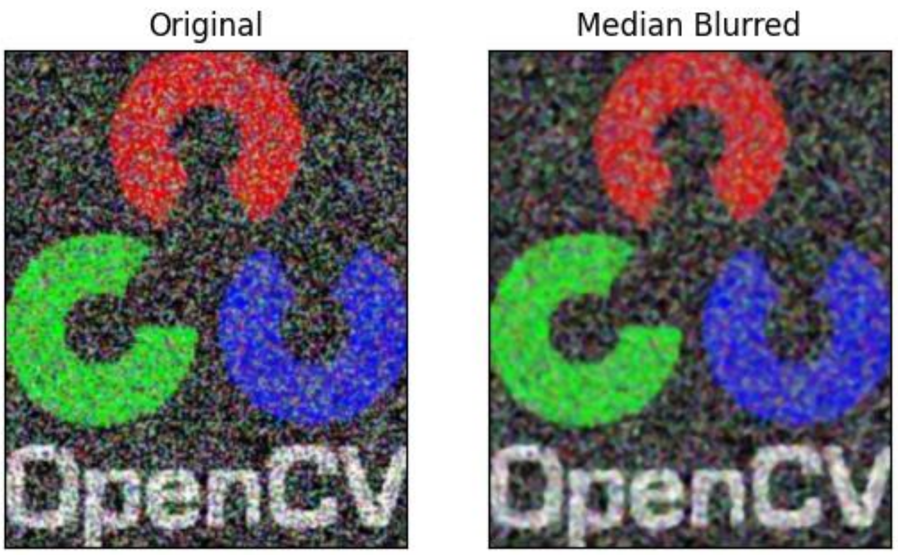


Figure 6: Median blur effect on image.
We can see that a bit of noise from the original image has been removed.

4. THRESHOLDING

Thresholding is a simple yet effective technique in image processing used to segment an image by converting it into a binary image. This process involves setting a threshold value, and then changing the pixels in the image to either white (255) or black (0), based on whether they are above or below the threshold value. Thresholding is widely used in applications such as object detection, image segmentation, and pattern recognition.

4.1 Global Thresholding

Global thresholding applies a single, global threshold value to the entire image. This method is straightforward and works well when the lighting conditions in the image are uniform.

Steps:

- Choose a threshold value.
- Compare each pixel value to the threshold.
- Set the pixel to white if its value is greater than the threshold; otherwise, set it to black.

```
ret, th_img_1 = cv.threshold(img, 127, 255, cv.THRESH_BINARY)
```

‘127’ is the threshold value. ‘255’ is the maximum value assigned to a pixel if it exceeds the threshold. cv.THRESH_BINARY specifies that we want a binary threshold.

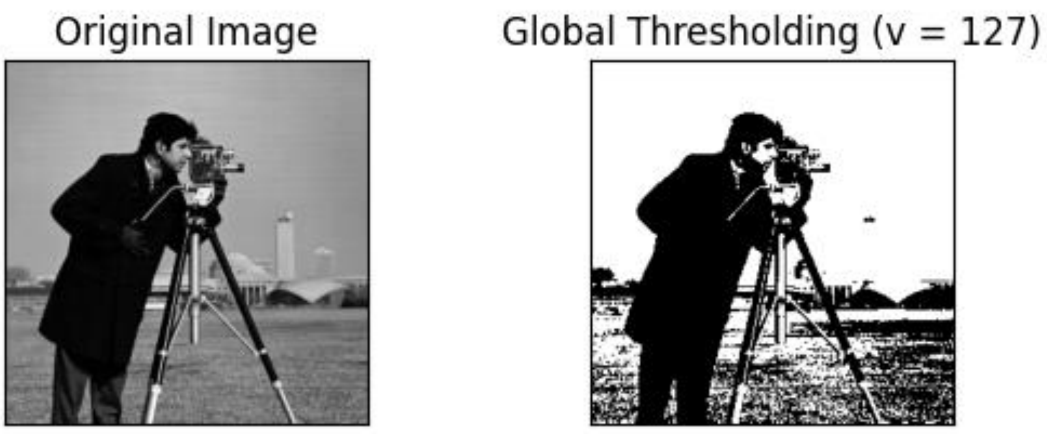


Figure 7: Global Thresholding applied on an image

4.2 Adaptive Thresholding

Adaptive thresholding, unlike simple thresholding, calculates the threshold for smaller regions of the image. This method is more robust for images with varying lighting conditions, as it adjusts the threshold value dynamically based on the local pixel intensity distribution.

Types of Adaptive Thresholding:

- **Mean Adaptive Thresholding:** The threshold value is the mean of the neighborhood area minus a constant C.
- **Gaussian Adaptive Thresholding:** The threshold value is a weighted sum (Gaussian-weighted sum) of the neighborhood values minus a constant C.

```
th_img_2 = cv.adaptiveThreshold(img, 255,
cv.ADAPTIVE_THRESH_MEAN_C, cv.THRESH_BINARY, 11, 2)
th_img_3 = cv.adaptiveThreshold(img, 255,
cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY, 11, 2)
```

- '255' is the maximum value assigned to a pixel if it exceeds the threshold.
- cv.ADAPTIVE_THRESH_MEAN_C and cv.ADAPTIVE_THRESH_GAUSSIAN_C specify the adaptive thresholding methods.
- '11' is the size of the neighbourhood area (block size).
- '2' is the constant subtracted from the mean or weighted mean.

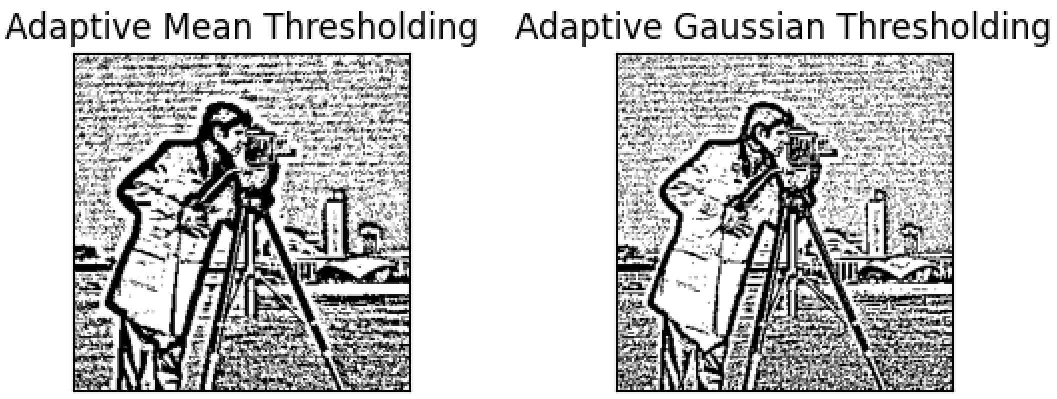


Figure 8: Adaptive Thresholding applied on an image

5. EDGE DETECTION

Edge detection is a crucial technique in image processing and computer vision used to identify points in an image where the brightness changes sharply. These points often represent boundaries of objects within an image. Edge detection is fundamental in various applications such as object recognition, image segmentation, and scene understanding. Several methods can be used for edge detection, each with its strengths and use cases.

5.1 Prewitt Operator

The Prewitt operator is a simple and effective edge detection method. It uses two 3x3 convolution kernels, one for detecting horizontal edges and one for vertical edges:

-1	0	+1
-1	0	+1
-1	0	+1

G_x

+1	+1	+1
0	0	0
-1	-1	-1

G_y

The operator calculates the gradient magnitude of the image intensity at each pixel, highlighting regions with high spatial frequency (edges).

```
img_blurred = cv.GaussianBlur(img, (3, 3), 0)
kernelx = np.array([[1,1,1],[0,0,0],[-1,-1,-1]])
kernely = np.array([[-1,0,1],[-1,0,1],[-1,0,1]])
img_prewittx = cv.filter2D(img_blurred, -1, kernelx)
img_prewitty = cv.filter2D(img_blurred, -1, kernely)
```

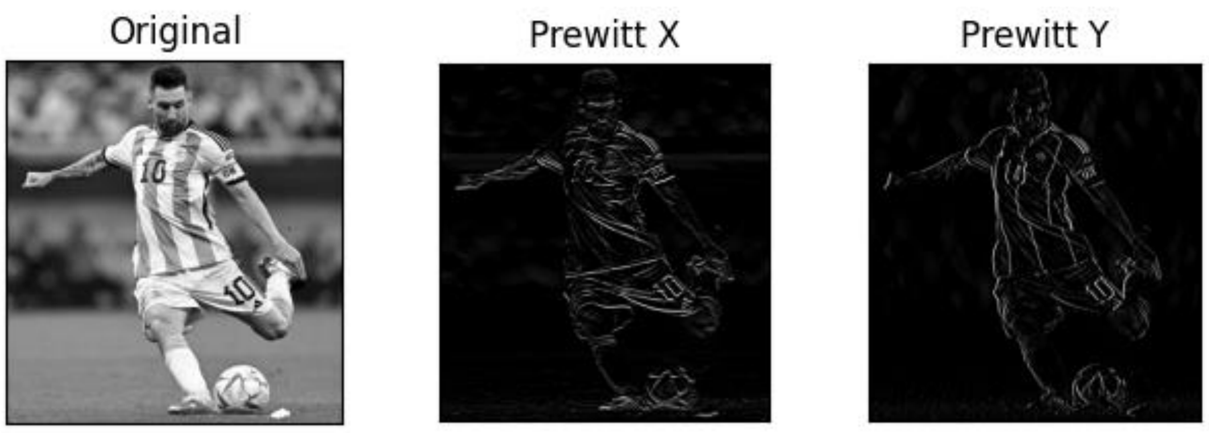


Figure 9: Prewitt Edge Detection on Image

5.2 Sobel Operator

The Sobel operator is an extension of the Prewitt operator. It also uses two convolution kernels to detect horizontal and vertical edges, but these kernels place a higher emphasis on the central pixels. The Sobel kernels are:

$$\underset{Gx}{\begin{array}{|c|c|c|}\hline -1 & 0 & +1 \\ \hline -2 & 0 & +2 \\ \hline -1 & 0 & +1 \\ \hline \end{array}} \qquad \underset{Gy}{\begin{array}{|c|c|c|}\hline +1 & +2 & +1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -2 & -1 \\ \hline \end{array}}$$

The Sobel operator typically provides better edge detection results by giving more weight to the center pixels, resulting in a smoother gradient magnitude.

```
img_blurred = cv2.GaussianBlur(bw_img, (3, 3), 0)
sobelx = cv2.Sobel(src=img_blurred, ddepth=cv2.CV_64F, dx=1, dy=0, ksize=5)
sobely = cv2.Sobel(src=img_blurred, ddepth=cv2.CV_64F, dx=0, dy=1, ksize=5)
sobelxy = cv2.Sobel(src=img_blurred, ddepth=cv2.CV_64F, dx=1, dy=1,
ksize=5)
```

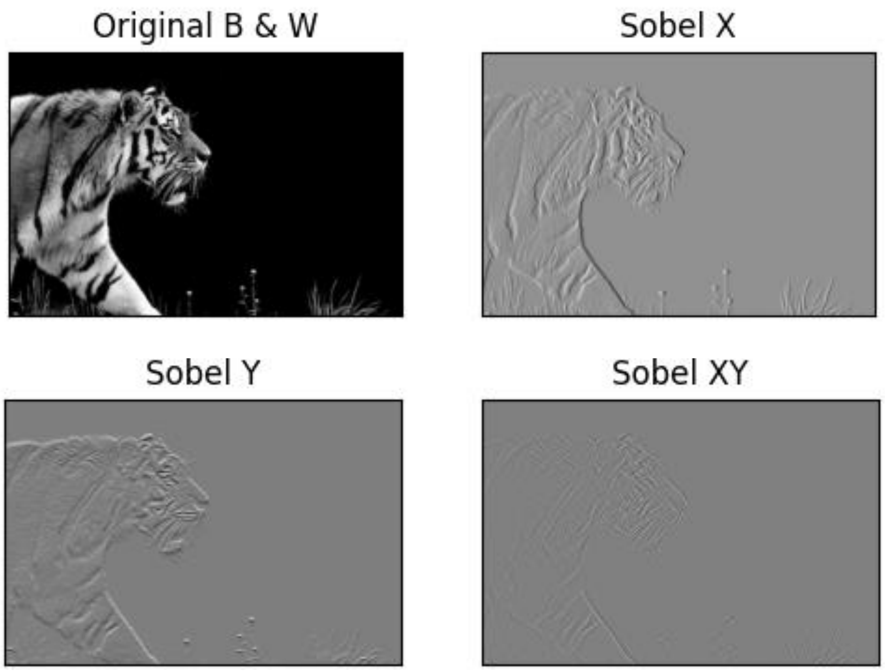


Figure 10: Sobel operations on an image

5.3 Canny Edge Detector

The Canny edge detector is a more advanced and multi-step edge detection algorithm. It aims to detect a wide range of edges in images while reducing noise and avoiding false edge detection. The steps involved are:

- **Noise Reduction**: Apply a Gaussian blur to the image to reduce noise.
- **Gradient Calculation**: Compute the gradient magnitude and direction using Sobel filters.
- **Non-Maximum Suppression**: Thin the edges by suppressing non-maximum pixels in the gradient direction.
- **Double Thresholding**: Apply two thresholds to classify pixels as strong, weak, or non-edges.
- **Edge Tracking by Hysteresis**: Finalize edges by suppressing weak edges that are not connected to strong edges.

```
edges = cv.Canny(img, 100, 200)
```

Here, 100 is the lower threshold for the hysteresis procedure. It is used to identify edges with a gradient intensity higher than this value as potential edges. 200 is the upper threshold for the hysteresis procedure. It is used to identify edges with a gradient intensity higher than this value as definite edges.

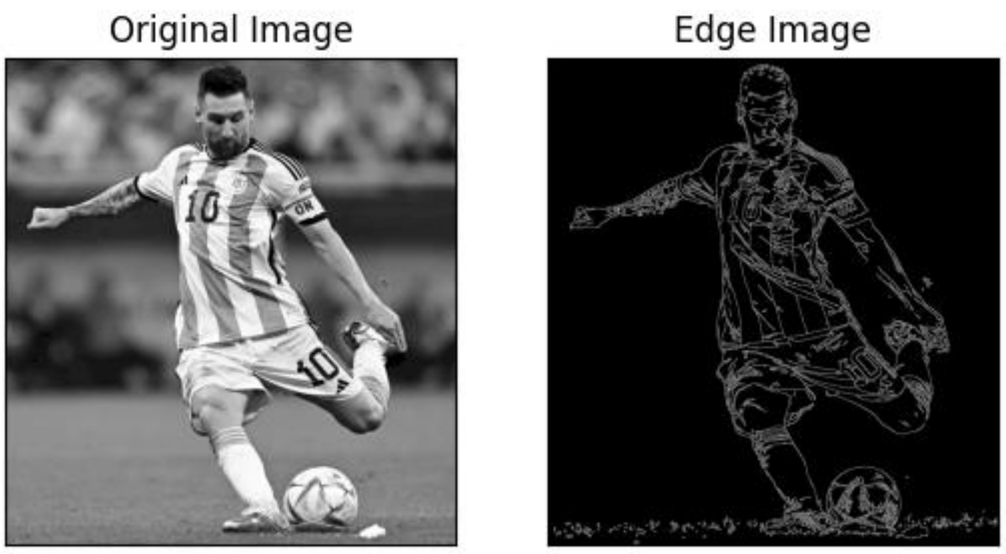


Figure 11: Canny Edge Detection on an image

6. MORPHOLOGICAL OPERATIONS

Morphological operations are a set of image processing techniques that process images based on shapes. These operations apply a structuring element to an input image and generate an output image of the same size. Morphological operations are typically applied to binary images but can also be used with grayscale images. They are used in various applications such as noise removal, image enhancement, object detection, and image segmentation.

6.1 Erosion

Erosion removes pixels on object boundaries. The basic idea is to erode (shrink) the boundaries of the foreground object. A pixel in the original image will be considered 1 (white) only if all the pixels in the structuring element are 1; otherwise, it will be eroded (set to 0). Erosion shrinks objects in a binary image. It removes small white noises. It can separate two connected objects.

```
# Make a 5 x 5 kernel
kernel = np.ones((5, 5), np.uint8)
erosion = cv.erode(img, kernel, iterations=1)
```

6.2 Dilation

Dilation adds pixels to the boundaries of objects. A pixel element is 1 if at least one pixel in the structuring element is 1. It is used to expand objects in a binary image. Dilation enlarges objects in a binary image. It can fill small holes and gaps within objects. It is used to join broken parts of an object.

```
dilation = cv.dilate(img, kernel, iterations=1)
```

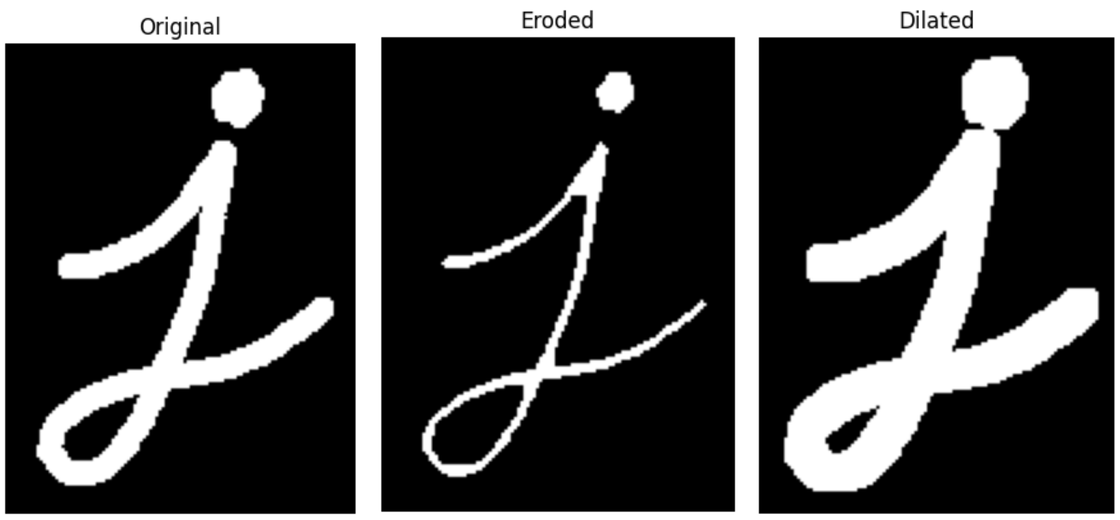


Figure 12: Erosion and dilation effects on an image

6.3 Opening

Opening is the combination of erosion followed by dilation. It is used to remove small objects from an image while preserving the shape and size of larger objects. Opening removes small white noise and it can detach two connected objects.

```
opening = cv.morphologyEx(img, cv.MORPH_OPEN, kernel)
```

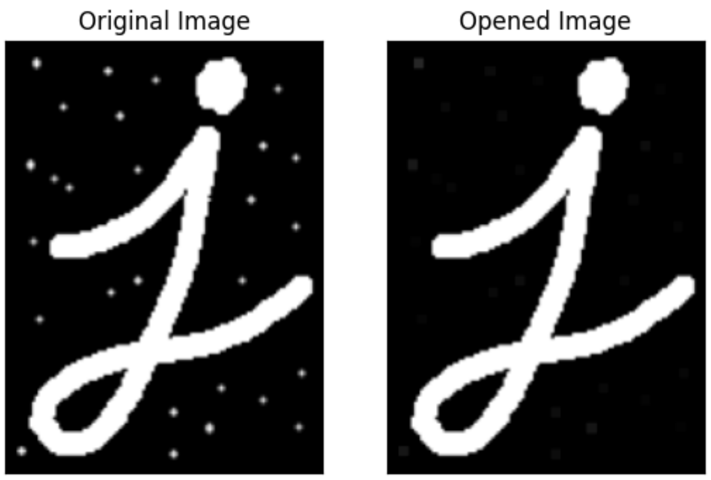


Figure 13: Opening operation on an image

6.4 Closing

Closing is the combination of dilation followed by erosion. It is used to close small holes and gaps in the foreground objects. Closing fills small holes and gaps within objects and it can join broken parts of an object.

```
closing = cv.morphologyEx(img, cv.MORPH_CLOSE, kernel)
```

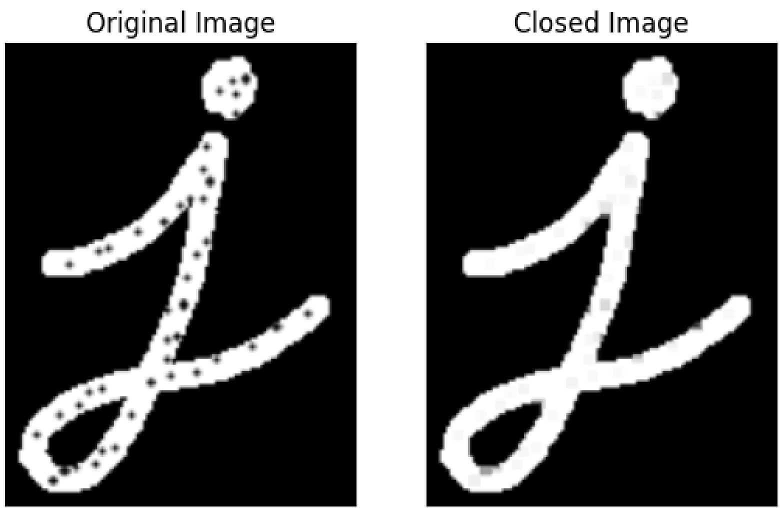


Figure 14: Closing operation on an image

6.5 Morphological Gradient

The morphological gradient is the difference between the dilation and erosion of an image. It highlights the boundaries of objects in the image. Morphological gradient enhances the edges of objects. It is used to highlight the outline of objects.

```
gradient = cv.morphologyEx(img, cv.MORPH_GRADIENT, kernel)
```

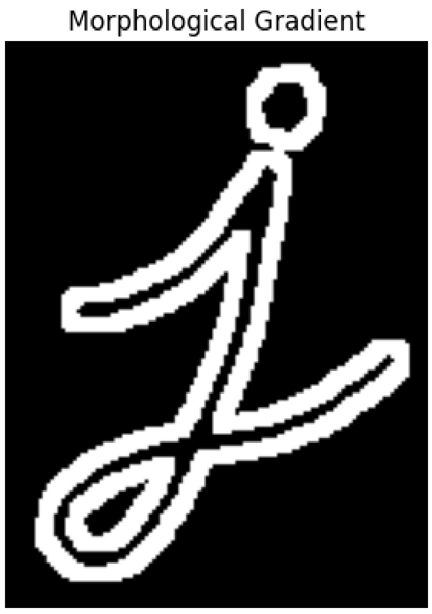


Figure 15: Morphological gradient applied on an image

7. IMAGE TRANSFORMS

Image transforms are mathematical operations that convert images from one domain to another, often to facilitate analysis, processing, or compression. By transforming an image, certain properties or features can be highlighted, and the data can be represented in a way that makes various tasks like compression, filtering, and enhancement easier. Commonly used image transforms include the Discrete Cosine Transform (DCT) and the Fourier Transform (FT).

7.1 Discrete Cosine Transform (DCT)

The Discrete Cosine Transform (DCT) is a widely used transform in image processing, particularly in compression algorithms like JPEG. The DCT transforms an image from the spatial domain to the frequency domain. It expresses an image as a sum of cosine functions of varying frequencies and amplitudes.

Key Points:

- The DCT separates the image into parts (or spectral sub-bands) of differing importance with respect to the image's visual quality.
- Most of the visually significant information in an image is concentrated in a few low-frequency components of the DCT.
- The DCT is efficient for image compression because it concentrates most of the image's energy in a small number of coefficients.

7.2 Inverse Discrete Cosine Transform (IDCT)

The Inverse Discrete Cosine Transform (IDCT) converts the frequency domain representation back to the spatial domain. This operation is essential for reconstructing the original image from its DCT coefficients.

```
imf = np.float32(img)
# Find dct
dct = cv2.dct(imf)
# Apply inverse dct
idct = cv2.idct(dct)
idct = np.uint8(idct)
```

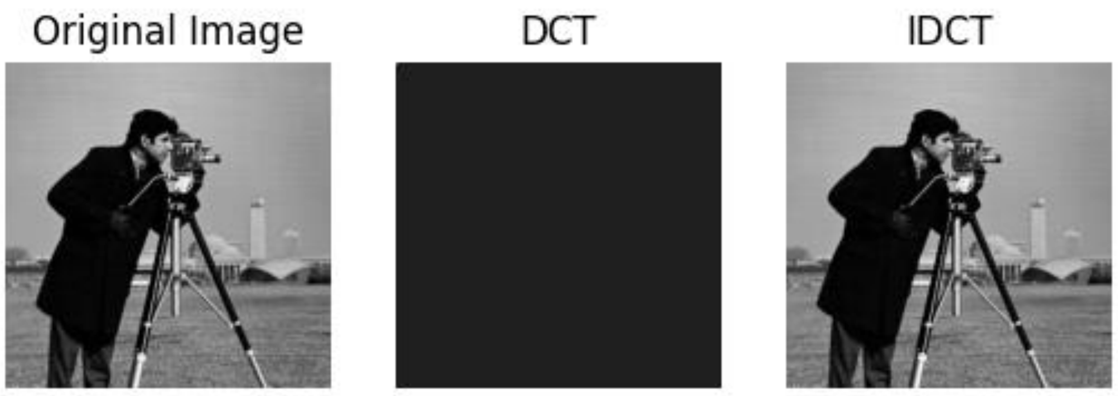


Figure 16: Discrete Cosine Transform on an image

7.3 Fourier Transform (FT)

The Fourier Transform (FT) is another powerful tool for transforming images from the spatial domain to the frequency domain. The FT represents an image as a sum of sinusoidal functions, each with a specific frequency and amplitude. The Fast Fourier Transform (FFT) is a computationally efficient algorithm to compute the FT.

Key Points:

- The FT decomposes an image into its sine and cosine components.
- The resulting representation shows how much of each frequency is present in the image.
- The FT is useful for image filtering, analysis, and restoration.

```
# Fourier Transform
dft = cv.dft(np.float32(img),flags = cv.DFT_COMPLEX_OUTPUT)
dft_shift = np.fft.fftshift(dft)

# Magnitude Spectrum is obtained by finding the magnitude of
the complex numbers.
# It is used to analyze the frequency content of the image.
magnitude_spectrum =
20np.log(cv.magnitude(dft_shift[:,:,0],dft_shift[:,:,1]))
```

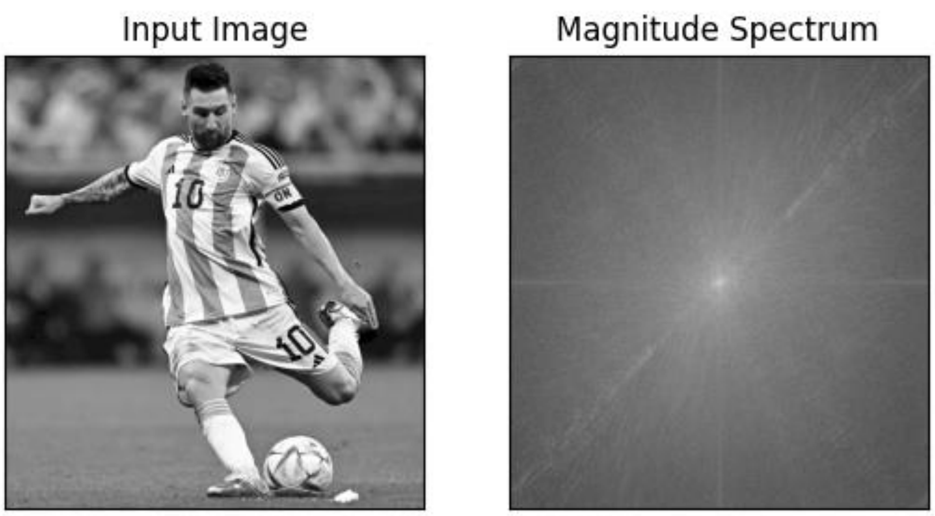


Figure 17: Magnitude Spectrum (Fourier Transform) of the image

7.4 Inverse Fourier Transform (IFT)

The Inverse Fourier Transform (IFT) converts the frequency domain representation back to the spatial domain. This is essential for reconstructing the original image from its Fourier coefficients.

```
img = magnitude_spectrum
rows, cols = img.shape
crow,ccol = rows//2 , cols//2

# We create a mask to remove the high frequency components which are responsible for edges.
# That is, we apply low-pass filter to the image. It actually blurs the image.
# Mask has centre square as 1, remaining all zeros.
mask = np.zeros((rows,cols,2),np.uint8)
mask[crow-30:crow+30, ccol-30:ccol+30] = 1

# apply mask and inverse DFT
fshift = dft_shift
f_ishift = np.fft.ifftshift(fshift)
img_back = cv.idft(f_ishift)
img_back = cv.magnitude(img_back[:,:,0],img_back[:,:,1])
```

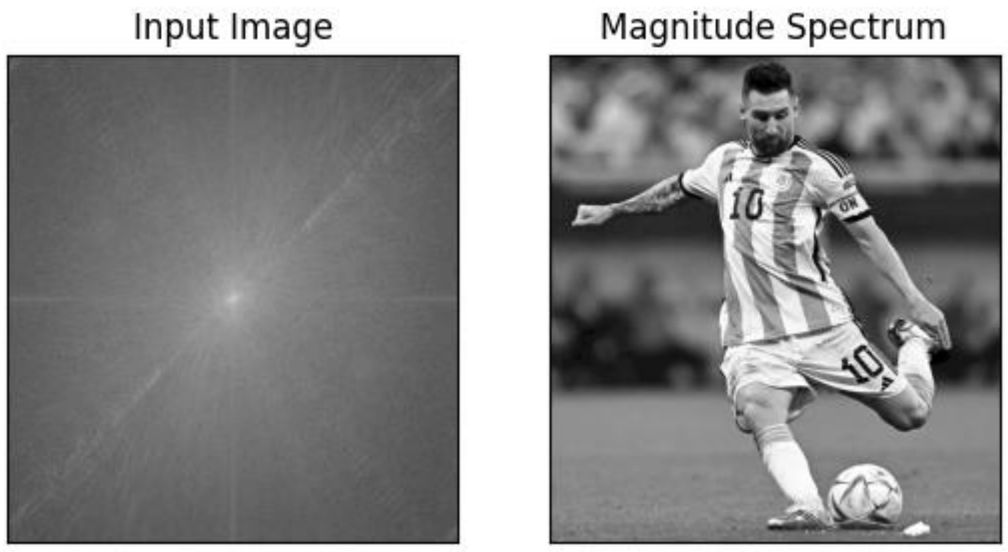


Figure 18: Inverse Fourier Transform on the original magnitude spectrum gives the original image

8. IMAGE SEGMENTATION

Image segmentation is a crucial step in image processing and computer vision. It involves partitioning an image into multiple segments (sets of pixels) to simplify or change the representation of an image into something more meaningful and easier to analyze. The goal of segmentation is to identify and isolate regions of interest in an image, such as objects, boundaries, or other structures.

8.1 Image Segmentation with Watershed Algorithm

The Watershed algorithm is a powerful tool for image segmentation, particularly useful for separating touching or overlapping objects. It is based on the concept of topographic representation of an image. In this analogy, the image is viewed as a landscape with valleys (dark regions) and ridges (light regions). The Watershed algorithm treats pixels values as heights and segments the image by simulating the flooding process from these valleys until water from different sources meets at the ridges.

Here are the steps for watershed algorithm:

1) **Converting to Grayscale and Thresholding:** The image is read and converted to a grayscale image to simplify processing. Otsu's thresholding is used to binarize the image, making it easier to distinguish the foreground objects from the background.

```
# Convert the image to grayscale for segmentation
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
ret, thresh = cv.threshold(gray, 0, 255, cv.THRESH_BINARY_INV +
cv.THRESH_OTSU)
```

2) **Morphological Opening:** This step removes small noise from the binarized image by applying erosion followed by dilation.

```
kernel = np.ones((3,3), np.uint8)
opening = cv.morphologyEx(thresh, cv.MORPH_OPEN, kernel, iterations=2)
```

3) **Sure Background Area:** Dilation is applied to identify the sure background area, ensuring that the background is separated from the foreground objects.

```
# Sure background area is the area which is not the part of the object
sure_bg = cv.dilate(opening, kernel, iterations=3)
```

4) **Sure Foreground Area**: The distance transform is applied to the morphologically opened image to obtain the sure foreground area, highlighting the objects of interest.

```
# Sure foreground area is the area which is the part of the object
dist_transform = cv.distanceTransform(opening, cv.DIST_L2, 5)
ret, sure_fg = cv.threshold(dist_transform, 0.7dist_transform.max(), 255,
0)
```

5) **Unknown Region**: The unknown region is identified by subtracting the sure foreground from the sure background.

```
# Finding unknown region
sure_fg = np.uint8(sure_fg)
unknown = cv.subtract(sure_bg, sure_fg)
```

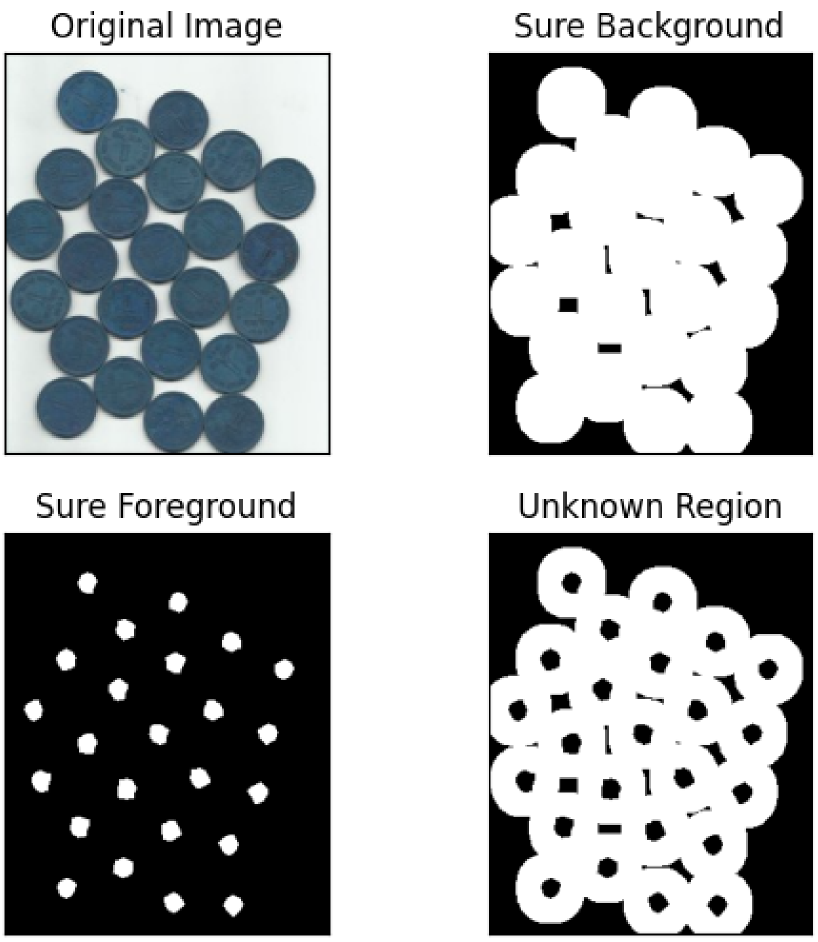


Figure 19: Image processed under Watershed algorithm

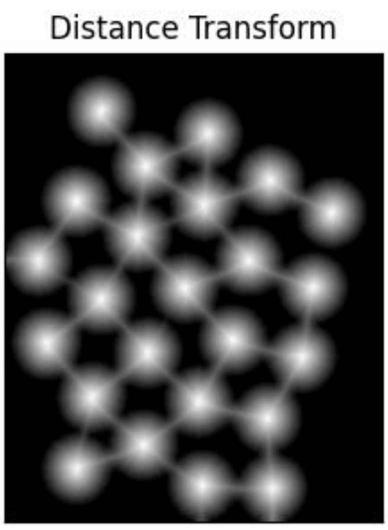


Figure 20: Distance Transform plot

6) **Marker Labelling**: Connected components labeling is used to mark the sure foreground regions with integers. These markers are incremented by 1 to ensure the background is labeled as 1 instead of 0. The unknown region is marked with 0.

```
# Marker labelling
# In the watershed algorithm, the regions to be segmented are marked with
integers.
ret, markers = cv.connectedComponents(sure_fg)
# Add one to all the markers so that sure background is not 0, but 1
markers = markers + 1
# Mark the region of unknown with 0
markers[unknown==255] = 0
```

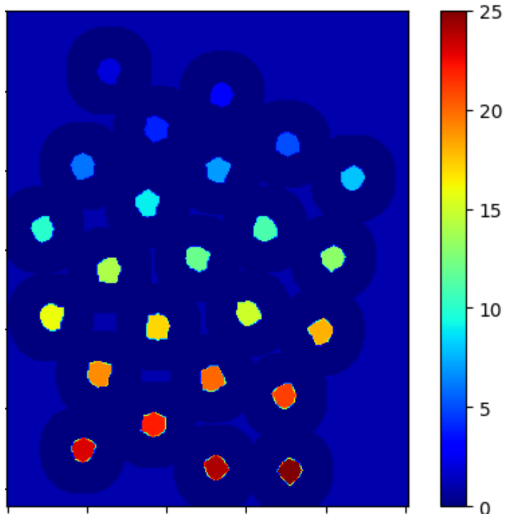


Figure 21: Color plot to show differently labelled marker

7) **Applying Watershed Algorithm**: The Watershed algorithm is applied to the original image using the markers. The boundaries of the segmented regions are marked with -1, and these boundaries are colored red in the original image for visualization.

```
# Apply watershed algorithm
# The boundary region will be marked with -1
markers = cv.watershed(img, markers)
img[markers == -1] = [255,0,0]
```

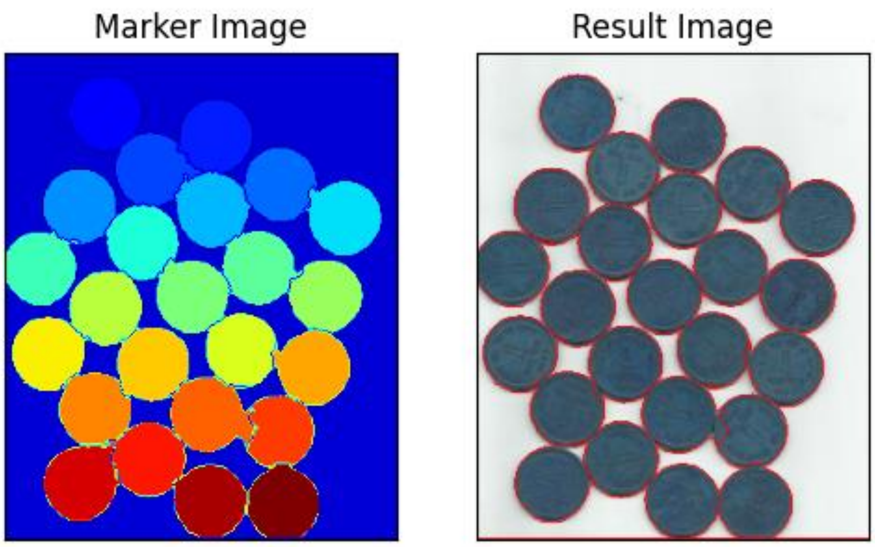


Figure 22: Final resultant image with coins segmented, concluding the Watershed algorithm for image segmentation

9. HISTOGRAM

Histograms are graphical representations of the distribution of pixel intensities in an image. They provide a way to analyze the tonal distribution, showing how frequently each intensity value occurs. Histograms are fundamental tools in image processing for various tasks such as contrast adjustment, thresholding, and image equalization. So, basically, it is a graph that plots the number of pixels for each intensity level (ranging from 0 to 255 for 8-bit images).

9.1 Histogram Plotting

We use cv2.calcHist() function to find the histogram.

```
hist = cv2.calcHist([img], [0], None, [256], [0, 256])
```

- **[img]**: This is the image for which the histogram is to be calculated. It is passed as a list, indicating that we can pass multiple images if needed. In this case, img should be a single-channel (grayscale) image.
- **[0]**: This is the channel of the image to be used for histogram calculation. In a grayscale image, there is only one channel (channel 0). For a color image, you could specify 0 for blue, 1 for green, or 2 for red channels.
- **none**: This parameter specifies a mask. If you want to compute the histogram for a specific region of the image, you can provide a mask. Here, None indicates that no mask is used, and the histogram is computed for the entire image.
- **[256]**: This specifies the number of bins in the histogram. 256 bins correspond to the possible intensity values in an 8-bit image, ranging from 0 to 255.
- **[0, 256]**: This specifies the range of intensity values to be considered for the histogram. [0, 256] covers all possible intensity values in an 8-bit image.

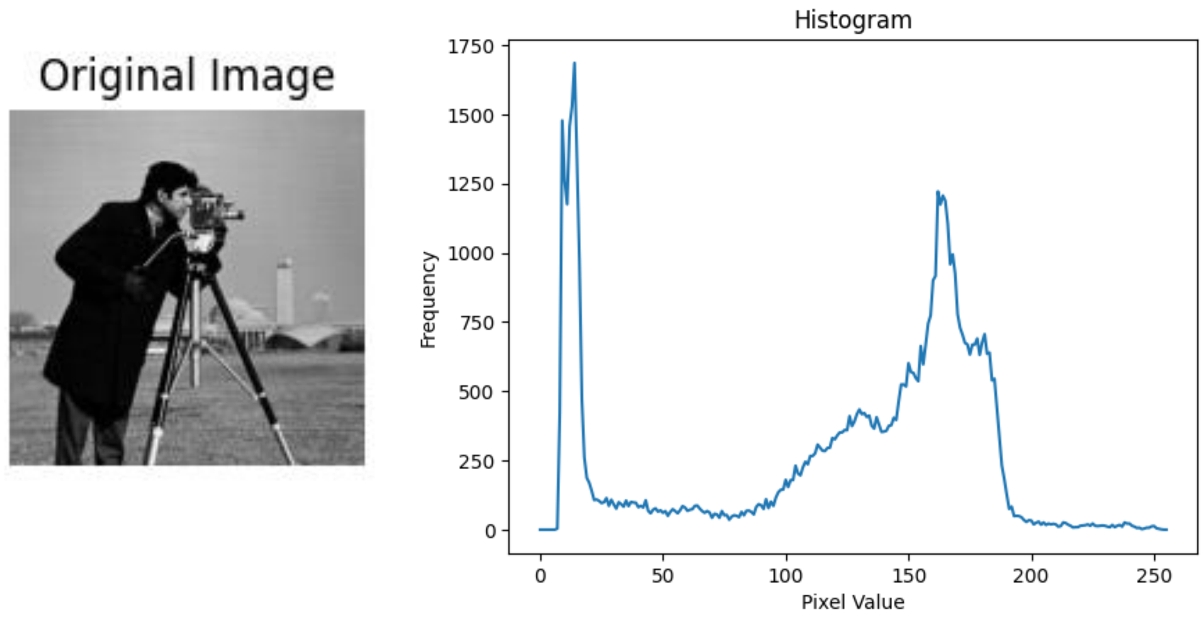


Figure 23: Original image and its histogram

9.2 Application of Mask

We used **cv2.calcHist()** to find the histogram of the full image. What if you want to find histograms of some regions of an image? Just create a mask image with white color on the region you want to find histogram and black otherwise. Then pass this as the mask.

```
# Create a blank mask with the same dimensions as the image
mask = np.zeros(img.shape[:2], np.uint8)
mask[25:125, 50:150] = 255

# Generate the masked image by bitwise AND operation between
the image and the mask
masked_img = cv.bitwise_and(img, img, mask=mask)

# Histogram with mask
hist_mask = cv.calcHist([img], [0], mask, [256], [0, 256])
```

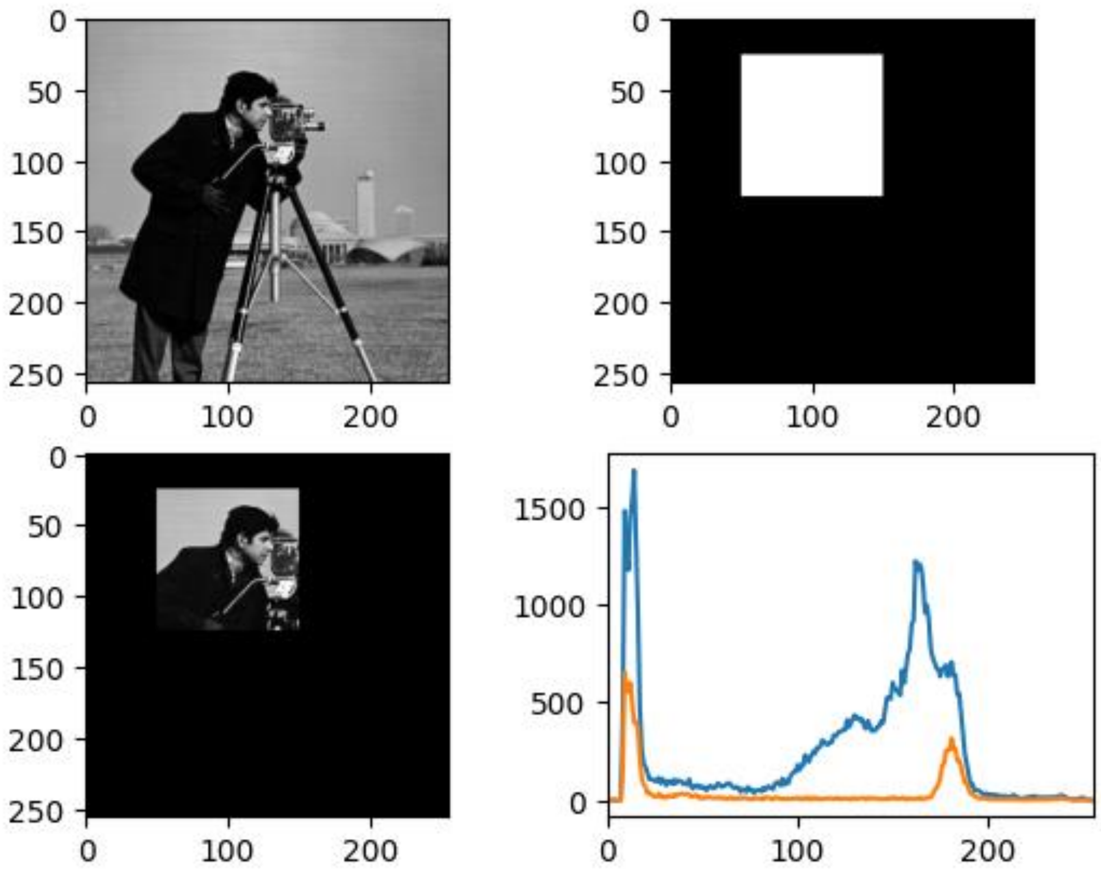


Figure 24 (a) Original image
(b) Mask
(c) Masked Image
(d) Histogram of original image (blue) compared with masked image (orange)

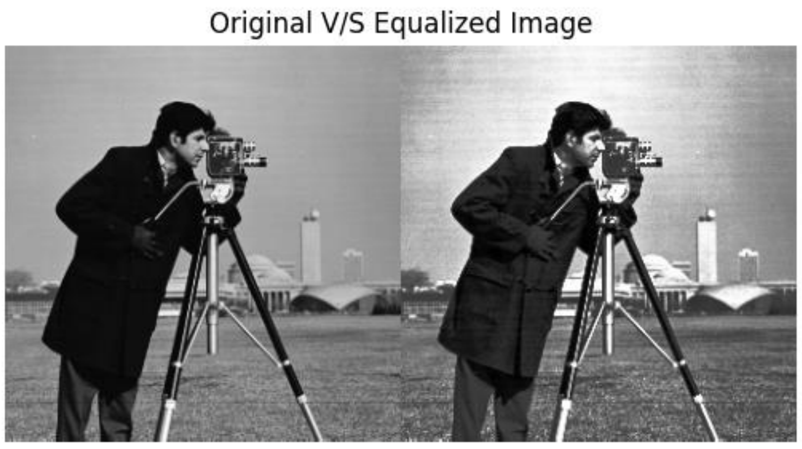
9.3 Histogram Equalization

Histogram equalization is a technique used to enhance the contrast of an image. It works by redistributing the pixel intensity values so that the histogram of the output image is more uniform. This improves the visibility of features in the image, especially in cases where the original image has poor contrast. Let’s discuss the types of histogram equalization:

1. Global Histogram Equalization

Global Histogram Equalization adjusts the contrast of the entire image based on its overall intensity distribution. It aims to make the histogram of the output image as flat as possible, enhancing the contrast uniformly across the image.

```
# Global histogram equalization
equ = cv.equalizeHist(img)
```



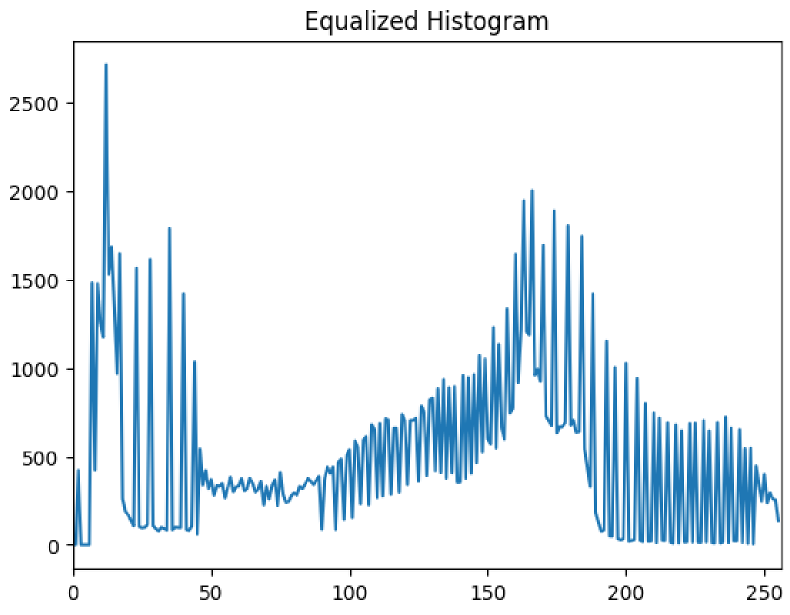
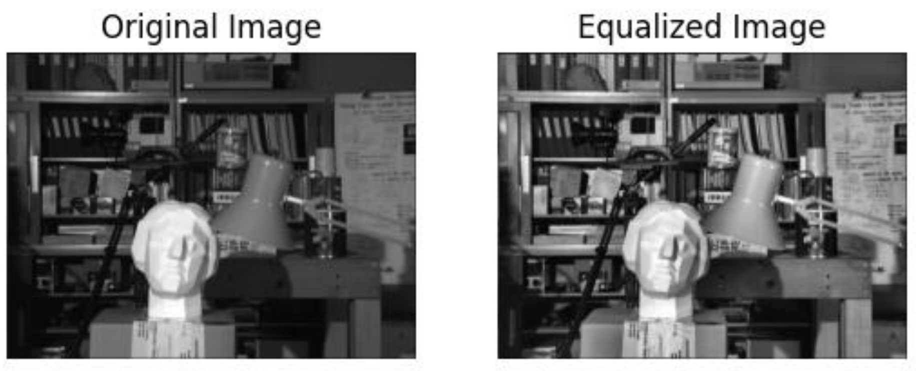


Figure 25: Original image, equalized image and equalized histogram

2. CLAHE (Contrast Limited Adaptive Histogram Equalization)

Histogram equalization is good when histogram of the image is confined to a particular region. It won't work good in places where there is large intensity variations where histogram covers a large region, ie both bright and dark pixels are present. So, to solve this problem, we use CLAHE. CLAHE is used when the image has different lighting conditions in different areas. It divides the image into small blocks and applies histogram equalization to each block. It limits the contrast in each block to avoid noise amplification.

```
# CLAHE
clahe = cv.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
cl1 = clahe.apply(img)
```



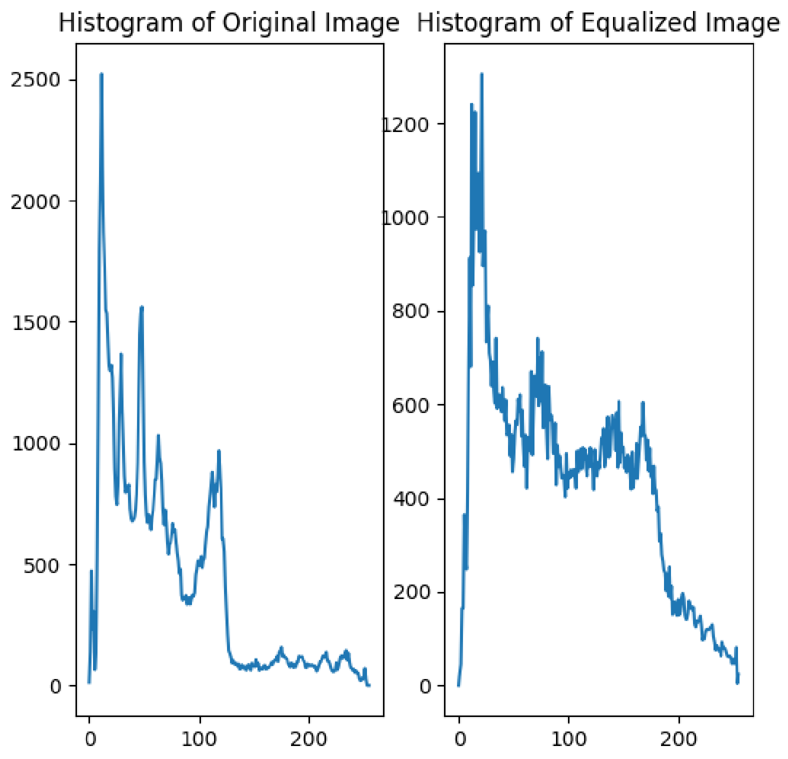


Figure 26: Equalization applied through CLAHE

10. IMAGE QUALITY ASSESSMENT

Image Quality Assessment (IQA) is the process of evaluating the quality of an image, typically by comparing it to a reference image. IQA algorithms provide quantitative measures to assess how similar a given image is to a reference, helping to determine the impact of processing techniques on image quality.

Importance of IQA

- **Objective Evaluation:** Provides quantitative metrics to evaluate image quality, allowing for objective comparison between different images or image processing techniques.
- **Optimization:** Helps in optimizing image processing algorithms (e.g., compression, enhancement, restoration) by providing feedback on how different parameters affect image quality.
- **Monitoring:** Used in various applications like broadcasting, medical imaging, remote sensing, and security to monitor and ensure high-quality images.

Common IQA Algorithms

1. PSNR (Peak Signal-to-Noise Ratio)
2. SSIM (Structural Similarity Index)
3. SAM (Spectral Angle Mapper)

10.1 Peak Signal-to-Noise Ratio (PSNR)

This technique deals with the peak image value and the noise density present in the image. The higher this value, the more is the noise less able to affect the peak value in the image. Thereby better image quality. PSNR measures the ratio between the maximum possible power of a signal and the power of corrupting noise. It is commonly used to evaluate the quality of reconstruction of lossy compression codecs.

Let’s look at the steps to get the PSNR between two images:

1) We take the original image and a noisy version of it as our example images. We will be comparing these two images.

2) **Compute Absolute Difference**: Calculate the absolute difference between corresponding pixels of the two images to get the pixel-wise differences.

3) **Convert to Float**: Convert the pixel differences to 32-bit floating point to prevent overflow during subsequent operations.

4) **Square Differences**: Square each pixel difference to compute the squared error for each pixel.

5) **Sum of Squared Errors (SSE)**: Sum all the squared errors to get the total squared error between the images.

6) **Check for Small Difference**: If the SSE is extremely small (indicating almost no difference between the images), print a message indicating that the difference is too small.

7) **Mean Squared Error (MSE)**: Calculate the mean squared error by dividing the SSE by the total number of pixels in the image.

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} (I1_i - I2_i)^2$$

I1 and I2 are the original and degraded images respectively.

8) **Calculate PSNR**: Compute the Peak Signal-to-Noise Ratio (PSNR) using the MSE and the maximum possible pixel value (255 for 8-bit images). The result is expressed in decibels (dB), indicating the quality of the test image compared to the reference image. Higher PSNR values indicate better image quality
.

$$PSNR = 10log_{10}(\frac{(L-1)^2}{MSE}) = 20log_{10}(\frac{L-1}{RMSE})$$

L = Maximum intensity possible (256 in our case)

```
s1 = cv2.absdiff(img1, img2)

s1 = np.float32(s1)
s1 = s1  s1

sse = s1.sum()

if sse <= 1  np.exp(1) - 10:
    print("Difference too small!")
```

```
shape = img1.shape

# Use mse = "1.0  sse / (shape[0]  shape[1]  shape[2])" for
rgb image
mse = 1.0  sse / (shape[0]  shape[1])

psnr = 10.0  np.log10((255  255) / mse)
```

We multiply by 10 to express the ratio in decibels. The higher the value, the less is noise able to affect the peak value in the image. So, better image quality.

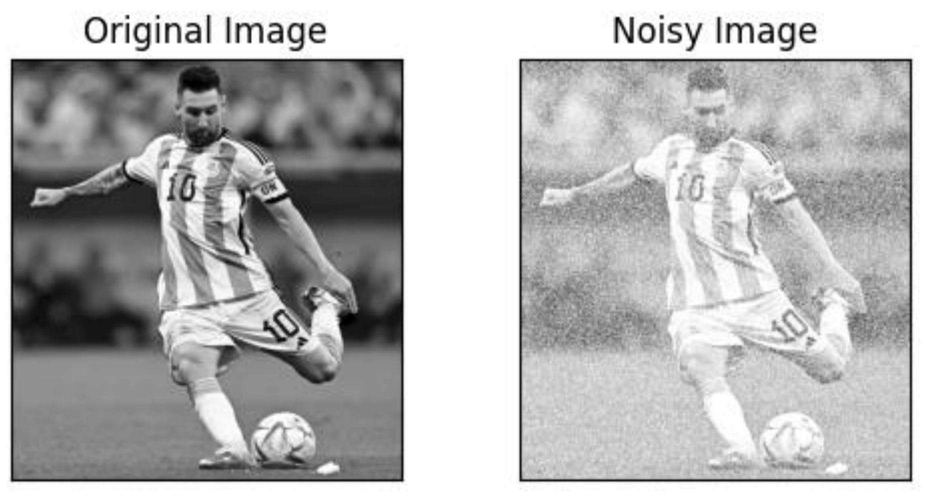


Figure 27: The original and degraded images for our example

For our sample, the PSNR comes out to be **7.9429 db.**

10.2 Structural Similarity Index (SSIM)

The Structural Similarity Index (SSIM) is a perceptual metric that quantifies the image quality degradation caused by processing such as data compression or by transmission errors. Unlike PSNR, which measures absolute errors, SSIM considers changes in structural information, luminance, and contrast, making it more consistent with human visual perception.

Key Concepts

The SSIM extracts 3 key features from an image:

- **Luminance:** Measures the difference in brightness between the reference and the test image. Luminance corresponds to the mean
- **Contrast**: Compares the contrast between the reference and the test image. Contrast corresponds to the standard deviation.

- **Structure:** Assesses the similarity of structures in the reference and the test image. Structure is an indirect determinant of the covariance of pixel values in the image.

The comparison between the two images is performed based on these 3 features.

Here are the steps for SSIM:

1) **Define Constants**: ‘c1’ and ‘c2’ are constants used to stabilize the division with weak denominators, which help in the SSIM calculation.

2) **Convert Images to Float:** Convert img1 and img2 to 32-bit floating point to ensure precision during the calculations.

3) **Calculate Squared Images and Product of Images:** Compute the square of each image (img1_sq and img2_sq) and the product of the two images (img1_img2).

4) **Calculate Gaussian Blurs:** Apply Gaussian blur to each image with a kernel size of (11, 11) and standard deviation of 1.5 to obtain the mean images (mu1 and mu2). Here, we calculate the luminance, µ (Mu). Luminance is measured by averaging over all the pixel values. But we don't apply the metric globally. It's better to apply the metrics regionally and take the overall mean. So, we compute the local means µ1 and µ2 using gaussian blur.

5) **Calculate Squares and Product of Means:** Compute the square of the blurred images (mu1_sq and mu2_sq) and the product of the blurred images (mu1_mu2).

6) **Calculate Variance (Contrast Term):** Apply Gaussian blur to the squared images to obtain the variances (sigma1_sq and sigma2_sq). Subtract the square of the means from these variances to get the contrast terms.

7) **Calculate Covariance (Structure Term):** Apply Gaussian blur to the product of the original images (img1_img2) to get the covariance (sigma12). Subtract the product of the means from this covariance.

8) **Luminance Term:** Compute the numerator (numerator1) as 2 mu1_mu2 + c1. Compute the denominator (denominator1) as mu1_sq + mu2_sq + c2.

$$l(I,J) = \frac{2\mu_I\mu_J + C_1}{\mu_I^2 + \mu_J^2 + C_1}$$

Figure 28: Intensity (Luminance) Comparison term

9) **Contrast and Structure Term:** Compute the numerator (numerator2) as 2 sigma12 + c2. Compute the denominator (denominator2) as sigma1_sq + sigma2_sq + c2.

$$c(I,J) = \frac{2\sigma_I\sigma_J + C_2}{\sigma_I^2 + \sigma_J^2 + C_2}$$

Figure 29: Contrast comparison term

10) **Calculate SSIM Index:** Combine the luminance, contrast, and structure terms to calculate the SSIM index (ssim_score) as (numerator1 numerator2) / (denominator1 denominator2).

```
# Constants for luminance and contrast
c1 = 6.5025
c2 = 58.5225
```

```
# Converting to float for squaring
img1 = np.float32(img1)
img2 = np.float32(img2)
img1_sq = img1  img1
img2_sq = img2  img2
img1_img2 = img1  img2

# Luminance
# applying GaussianBlur with (11,11) kernel where
mean=st_dev=1.5
mu1 = cv2.GaussianBlur(img1, (11, 11), 1.5)
mu2 = cv2.GaussianBlur(img2, (11, 11), 1.5)

# Taking squares and product of the means.
mu1_sq = mu1  mu1
mu2_sq = mu2  mu2
mu1_mu2 = mu1  mu2

# Contrast
sigma1_sq = cv2.GaussianBlur(img1_sq, (11, 11), 1.5)
sigma1_sq -= mu1_sq
sigma2_sq = cv2.GaussianBlur(img2_sq, (11, 11), 1.5)
sigma2_sq -= mu2_sq

# sigma12 is the covariance and represents the structure
correlation between the images.
```

```
sigma12 = cv2.GaussianBlur(img1_img2, (11, 11), 1.5)
sigma12 -= mu1_mu2

# Luminance term correlating the two images.
numerator1 = 2  mu1_mu2 + c1
denominator1 = mu1_sq + mu2_sq + c2

# Contrast and structure term correlating the two images.
numerator2 = 2  sigma12 + c2
denominator2 = sigma1_sq + sigma2_sq + c2

# Combined SSIM Index
ssim_score = (numerator1  numerator2) / (denominator1
denominator2)

print(ssim_score.mean())
```

This system calculates the Structural Similarity Index between 2 given images which is a value between -1 and +1. A value of +1 indicates that the two given images are very similar or the same while a value of -1 indicates the 2 given images are very different. Often these values are adjusted to be in the range [0, 1], where the extremes hold the same meaning.

For our example images, the SSIM score comes out to be **0.106**, which means that the two images are dissimilar.

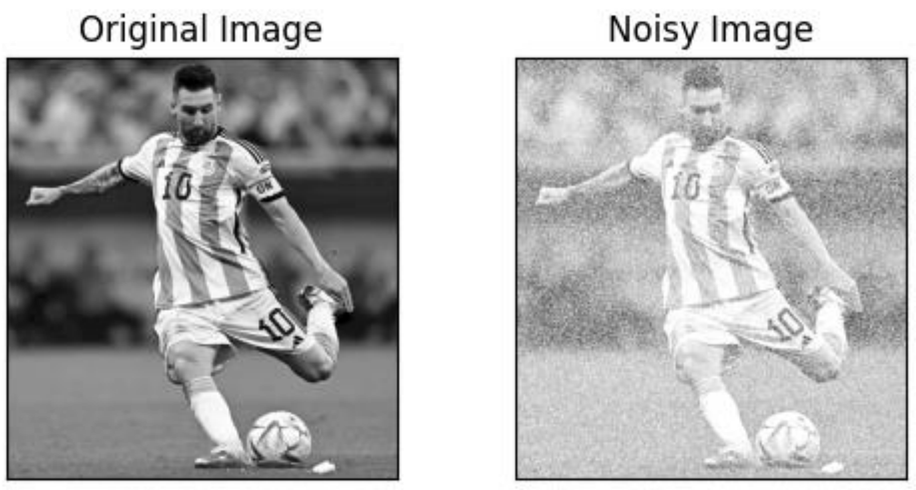


Figure 30: The original and degraded images for our example

10.3 Spectral Angle Mapper (SAM)

The Spectral Angle Mapper (SAM) is a technique used to compare the similarity between two spectral signatures (or images) by calculating the angle between their corresponding vectors in a high-dimensional space. SAM is commonly used in remote sensing and image analysis to determine the similarity between different spectral images or spectral signatures.

Concepts of SAM

- **Flatten Images into Vectors**: Each image is converted into a long vector with N=n×m, where n×m is the image size in pixels.
- **High-Dimensional Space**: To compare two images, each flattened image is treated as a vector in a high-dimensional space.
- **Measure Angle**: The goal is to measure the angle between these two vectors.
- **Identical Images**: If two images are exactly the same, the vectors will point in the same direction, and the angle between them will be 0.
- **Different Images**: If the images are completely different, the vectors will point in opposite directions, and the angle will be π (180 degrees).
- **Cosine Similarity**: Cosine similarity is used to measure the "closeness" of the vectors in high-dimensional space.

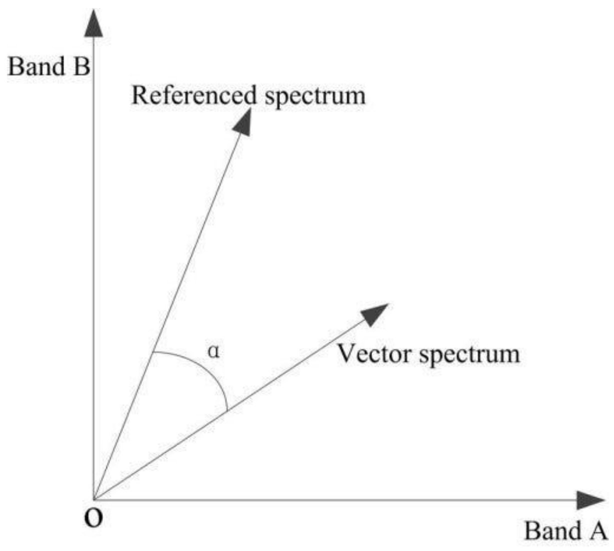


Figure 31: SAM plot for vector comparison

Cosine Similarity

Cosine similarity measures the cosine of the angle between two non-zero vectors. It is used to determine how similar the vectors are by comparing their direction, regardless of their magnitude. The cosine similarity is defined as:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

Figure 32: Cosine Similarity between two vectors A and B

where:

- $\vec{A} \cdot \vec{B}$ is the dot product of vectors $\vec{A}$ and $\vec{B}$.
- $\|\vec{A}\|$ and $\|\vec{B}\|$ are the magnitudes (norms) of vectors $\vec{A}$ and $\vec{B}$.

SAM Implementation

Here are the steps to implement SAM:

1. **Resize Images**: Resize both images to have the same dimensions by matching the smaller one. This ensures that both images have the same height and width for comparison.

2. **Flatten Images**: Reshape the 2D images into long 1D vectors, where each pixel is represented in the vector.

3. **Get Number of Spectral Channels**: Determine the number of spectral channels (assuming images are in 3D with colour channels, e.g., RGB).

4. **Initialize Array for Angles**: Create an array to store the spectral angle for each channel.

5. **Calculate Angle for Each Spectral Channel**: For each spectral channel, perform the following steps:

   - Compute the dot product of the two vectors corresponding to the current channel.
   - Calculate the magnitudes (norms) of the vectors for the current channel.
   - Calculate the cosine of the angle between the vectors using the dot product and magnitudes.
   - Ensure the cosine value lies between -1 and 1 using np.clip.
   - Compute the angle in radians using the inverse cosine function np.arccos and store it in the array.

This process results in an array of spectral angles, one for each channel, which indicates the similarity between the two images based on the angle between their spectral signatures.

```
# Resize the images to have the same dimensions. Images will match
the smaller one.
min_height = min(img1.shape[0], img2.shape[0])
min_width = min(img1.shape[1], img2.shape[1])
img1 = cv2.resize(img1, (min_width, min_height))
img2 = cv2.resize(img2, (min_width, min_height))

# Reshape the images into long vectors (flatten the 2D images to 1D)
img1 = img1.reshape((img1.shape[0]img1.shape[1], img1.shape[2]))
img2 = img2.reshape((img2.shape[0]img2.shape[1], img2.shape[2]))

# Get the number of spectral channels (assume images are in 3D with
color channels)
N = img1.shape[1]

# Initialize an array to store angles for each channel
sam_angles = np.zeros(N)

# Calculate the angle for each spectral channel
for i in range(N):
    # Dot product of the two vectors for channel i
    dot_product = np.dot(img1[:, i], img2[:, i])

    # Magnitudes (norms) of the vectors for channel i
    norm_img1 = np.linalg.norm(img1[:, i])
    norm_img2 = np.linalg.norm(img2[:, i])

    # Calculate the cos of the angle and ensure it lies between -1
and 1
    cosine_angle = np.clip(dot_product / (norm_img1  norm_img2), -1, 1)

    # Calculate the angle in radians and store it
    sam_angles[i] = np.arccos(cosine_angle)

# Print the average angle in degrees across all channels
print(np.degrees(np.mean(sam_angles)))
```

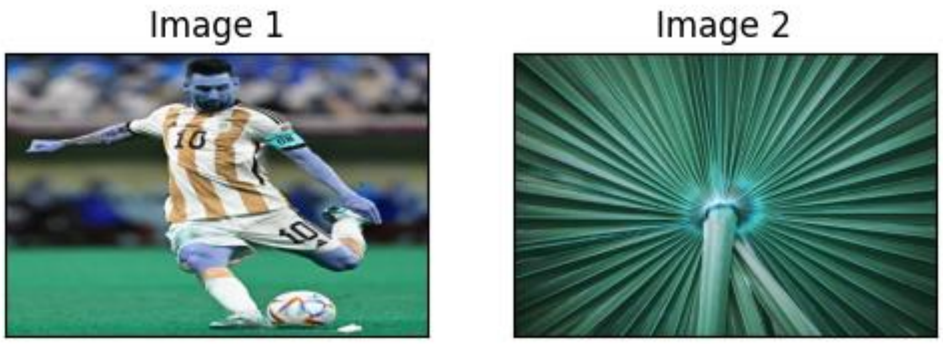


Figure 33: Example images for SAM comparison

Now, when we print the average angle in degrees across all channels using the above print statement, **we will get 89.999 degrees**. This is almost 90 degrees, which means that the example images (fig. 33) are completely dissimilar. The same process can be to assess image quality by comparing a noisy target image with its good counterpart reference image.

11. TEXTURE ANALYSIS

Texture analysis is a crucial aspect of image processing and computer vision, focusing on the quantification of texture properties within an image. Texture refers to the visual patterns in an image, characterized by the spatial distribution of intensity levels, which can describe surface properties like smoothness, roughness, granularity, etc. Texture analysis is widely used in various applications, including medical imaging, remote sensing, material science, and more.

Key Concepts in Texture Analysis

1. **Texture Features**: These are measurable properties of textures that help distinguish different textures from each other. Common texture features include contrast, coarseness, directionality, regularity, and roughness.

2. **Texture Descriptors**: These are methods used to extract texture features from an image. A common example is Local Binary Pattern (LBP).

3. **Texture Classification**: This involves assigning labels to different texture regions in an image based on the extracted features. Machine learning algorithms like Support Vector Machines (SVM), k-Nearest Neighbours (k-NN), and neural networks are commonly used for this purpose.

11.1 Texture Analysis Using Local Binary Pattern (LBP)

Local Binary Pattern (LBP) is a simple yet effective texture descriptor. It works by comparing each pixel with its neighboring pixels and encoding the results as a binary number.

Let’s understand LBP step by step:

Step 1:

- LBP works by comparing the intensity of a central pixel in a small neighborhood with the intensity of its surrounding pixels.

- Each pixel in the neighborhood is assigned a binary value based on whether its intensity is greater than or less than the intensity of the central pixel (threshold).

- These binary values are then concatenated into a binary number, which represents the texture of that neighborhood.

- In Figure 1 (below), the threshold value is 4 since it is the central pixel. If the intensity of the neighborhood is equal or smaller than the center, we set its value to 1, otherwise we set it to 0.

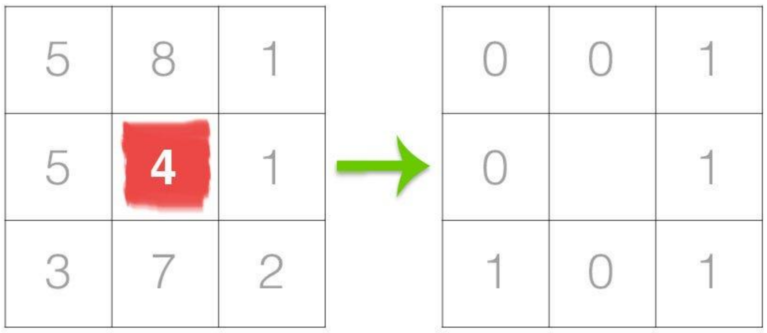


Figure 34: LBP descriptor operation on a 3 x 3 neighborhood of pixels

- These binary values can be then used to construct a histogram of the texture distribution within an image.

Step 2:

- From here, we need to calculate the LBP value for the centre pixel.
- We can start from any neighbouring pixel and work our way clockwise or counterclockwise.
- Given a 3 x 3 neighbourhood, we thus have 8 neighbours that we must perform a binary test on.
- The results of this binary test are stored in an 8-bit array, which we then convert to decimal, like this (Figure 35):

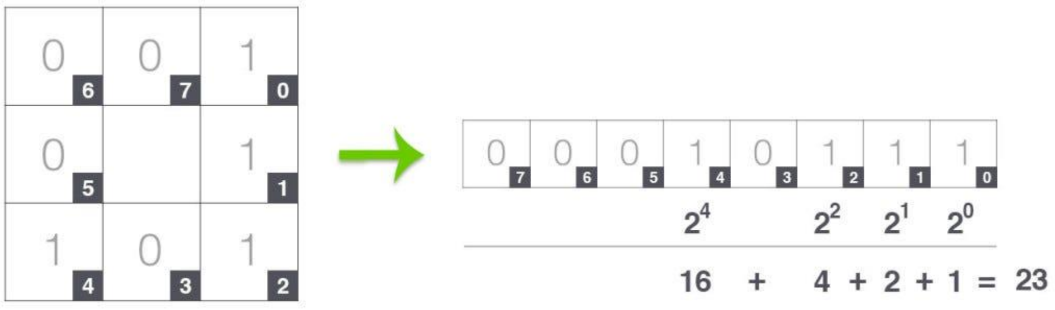


Figure 35: Finding LBP equivalent of center pixel

Step 3:

- This value is stored in the output LBP 2D array, which we can then visualize below (Figure 36):

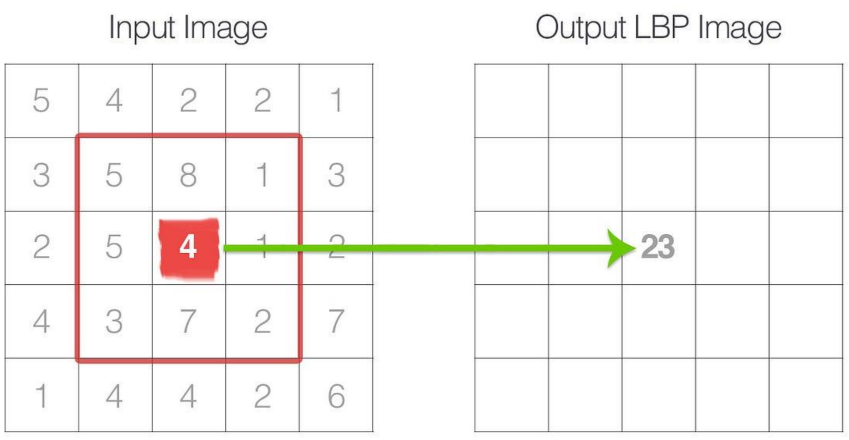


Figure 36: How LBP values are stored

Step 4:

- This process of thresholding, accumulating binary strings, and storing the output decimal value in the LBP array is then repeated for each pixel in the input image.

Note:

To account for variable neighborhood sizes, two parameters were introduced (Figure 37):

1. The number of points "p" in a circularly symmetric neighborhood to consider (thus removing relying on a square neighborhood).
2. The radius of the circle "r", which allows us to account for different scales.

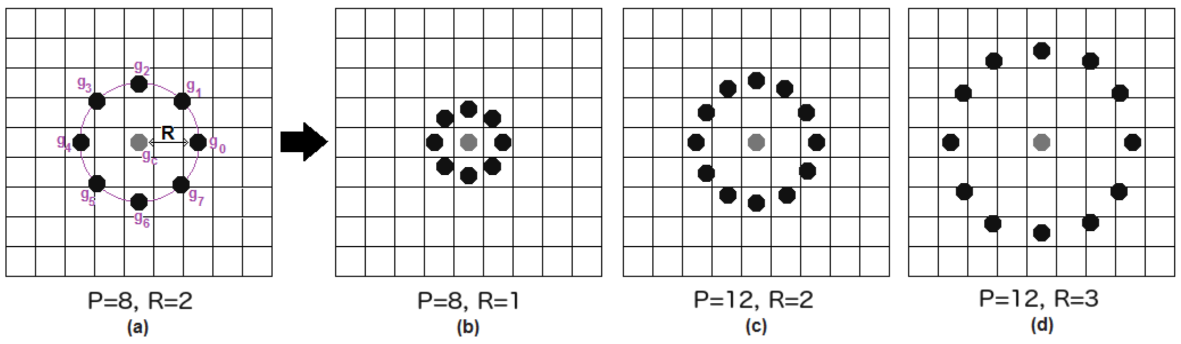


Figure 37: Varying number of points "p" and radius "r"

So, in figure 1, the number of points "p" = 3, and radius "r" = 1.

11.2 LBP Implementation in Python

1) **Input a gray image:** Input a gray image or convert the input color image into grayscale. Let's consider figure 38 as our example image for texture retrieval.

Figure 38: Input gray image

2) **Define Parameters**: Set the number of points (num_of_points) in the circular neighborhood. Set the radius (radius) of the circle.

```
num_of_points = 24
radius = 8
```

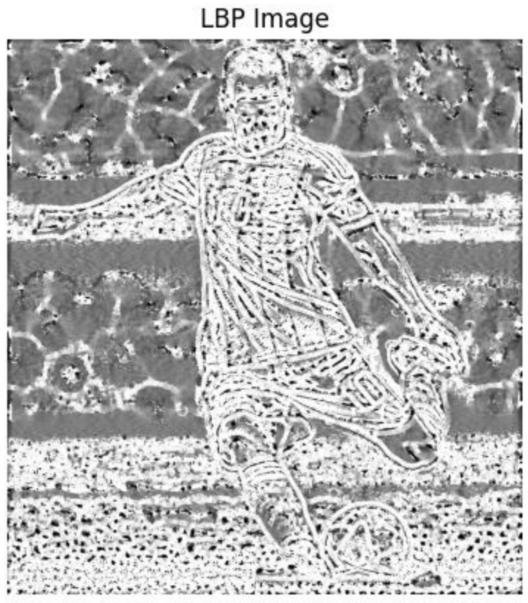
3) **Calculate LBP**: Compute the Local Binary Pattern (LBP) of the grayscale image using the 'local_binary_pattern()' function from skimage.feature.

```
lbp = feature.local_binary_pattern(
    gray_image,
    num_of_points,
    radius,
    method="uniform"
)
```

4) **Create Histogram and normalize it**: Generate a histogram of the LBP values with 'np.histogram()', creating bins for each possible LBP code value. Convert the histogram values to float type. Normalize the histogram by 1000 to adjust the pixel frequency.

```
hist, _ = np.histogram(
    lbp.ravel(),
    bins = np.arange(0, num_of_points + 3)
)

# Normalise the pixel frequency in histogram
hist = hist.astype("float")
hist /= 1000
```



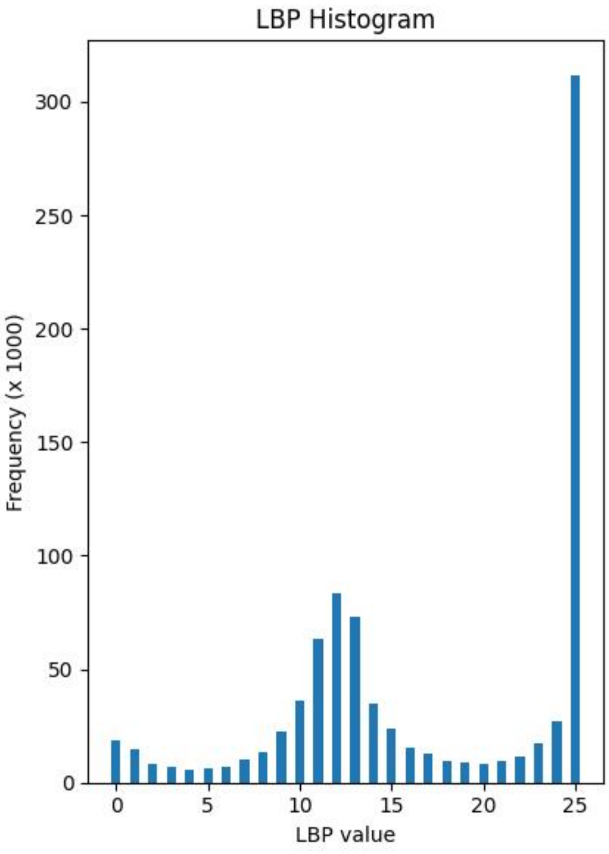
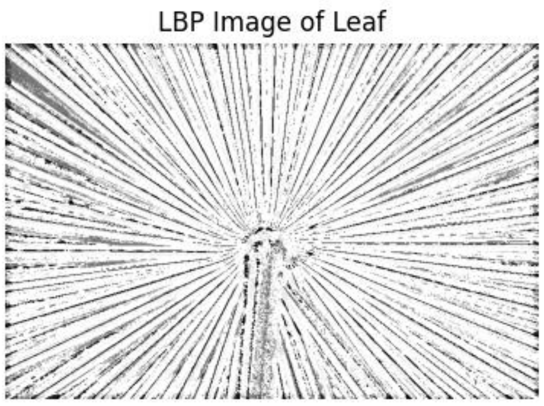


Figure 39: Textured image of input image and its LBP histogram

The above LBP image shows a texture analysis of the input image. We also plot the histogram of LBP representation. The histogram gives the number of pixels having each LBP value. Frequency is the number of pixels. Note that the frequency is normalized, and actual pixel frequency will be obtained by multiplying by 1000.

Some other images and their retrieved textures:



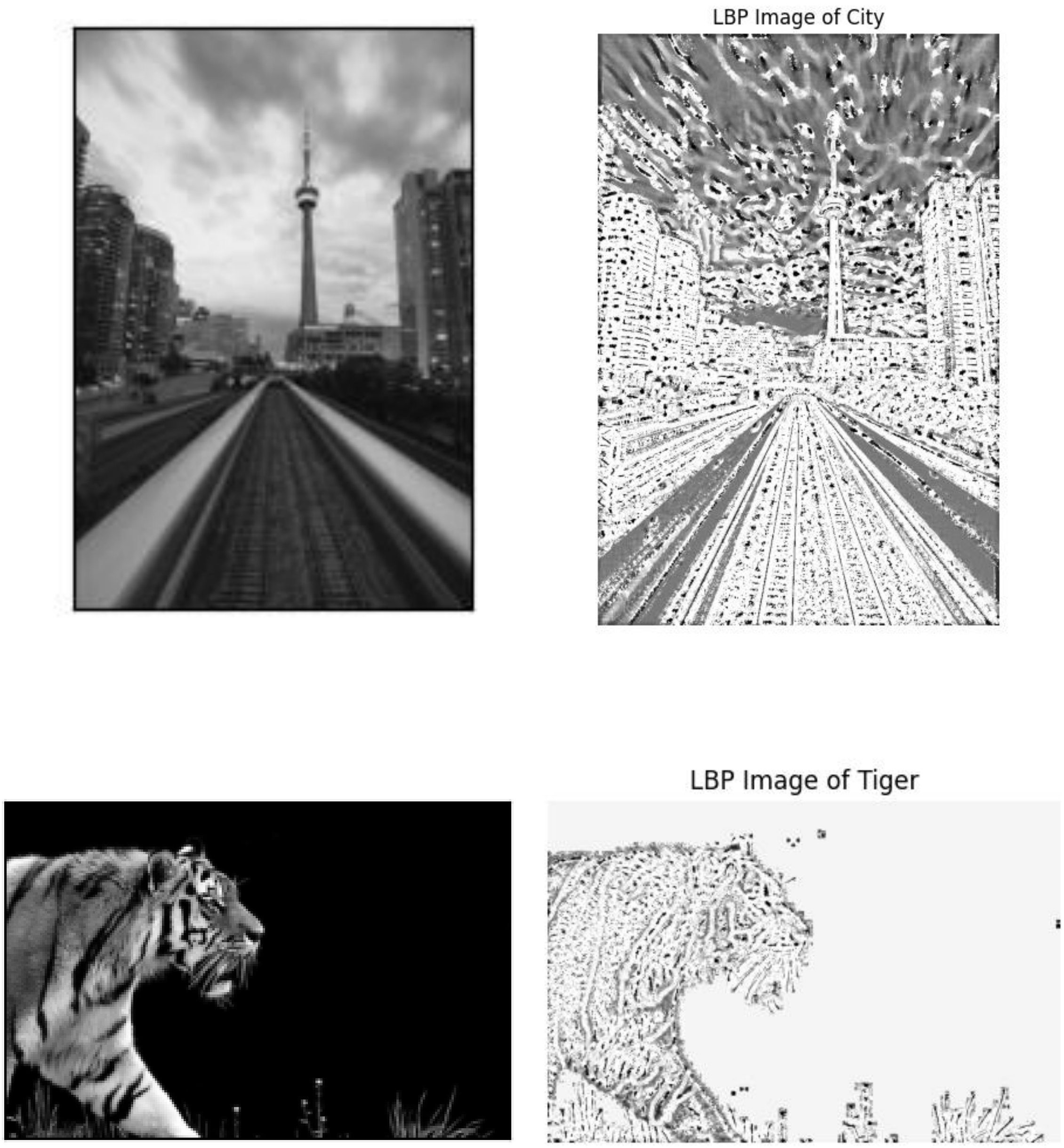


Figure 40: Examples of images and their retreived textures

11.3 Creating a Texture Classification Algorithm using Support Vector Machine

Support Vector Machine (SVM) is a powerful supervised machine learning algorithm used for classification and regression tasks. It is particularly well-suited for binary classification problems and is widely used in texture analysis to classify images based on their textures. SVM works by finding the optimal hyperplane that separates the data points of different classes in a high-dimensional feature space. The histograms obtained in Local Binary Pattern (LBP) can be used as feature vectors to train machine learning models. It can be fed into classification algorithms (like SVM) to classify images based on their textures.

Here's a step-by-step implementation of SVM for texture classification:

1) **Define a function to compute LBP histogram**:

- Compute the Local Binary Pattern (LBP) representation of the image using specified points and radius.
- Calculate the histogram of LBP values.
- Normalize the histogram to ensure the values sum to 1, adding a small epsilon to prevent division by zero.
- Return the normalized histogram.

```python
def compute_lbp_hist(image, eps=1e-7):
    # compute the Local Binary Pattern representation and histogram
    lbp = feature.local_binary_pattern(
        image,
        num_of_points,
        radius,
        method="uniform"
    )
    hist, _ = np.histogram(
        lbp.ravel(),
        bins=np.arange(0, num_of_points + 3)
    )

    # normalize the histogram
    hist = hist.astype("float")
    hist /= (hist.sum() + eps)

    # return the histogram of Local Binary Patterns
    return hist
```

2) **Initialize Data and Labels Lists**:

- Create empty lists to store feature data and corresponding labels.

```
data = []
labels = []
```

3) **Load and process training images**:

- Loop over the paths of training images.
- For each image:
    - Load the image using OpenCV.
    - Convert the image to grayscale.
    - Compute the LBP histogram using the defined function.
    - Extract the label from the image path (typically the folder name).
    - Append the label and the histogram to the respective lists.

```
# loop over the training images
for img_path in
paths.list_images(r"images\texture_analysis\training"):

    # load the image, convert it to grayscale, and
compute the histogram
    img = cv.imread(img_path)
    gray_img = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    hist = compute_lbp_hist(gray_img)

    # extract the label from the image path, then update
the
    # label and data lists
    label = os.path.basename(os.path.dirname(img_path))
    labels.append(label)
    data.append(hist)
```

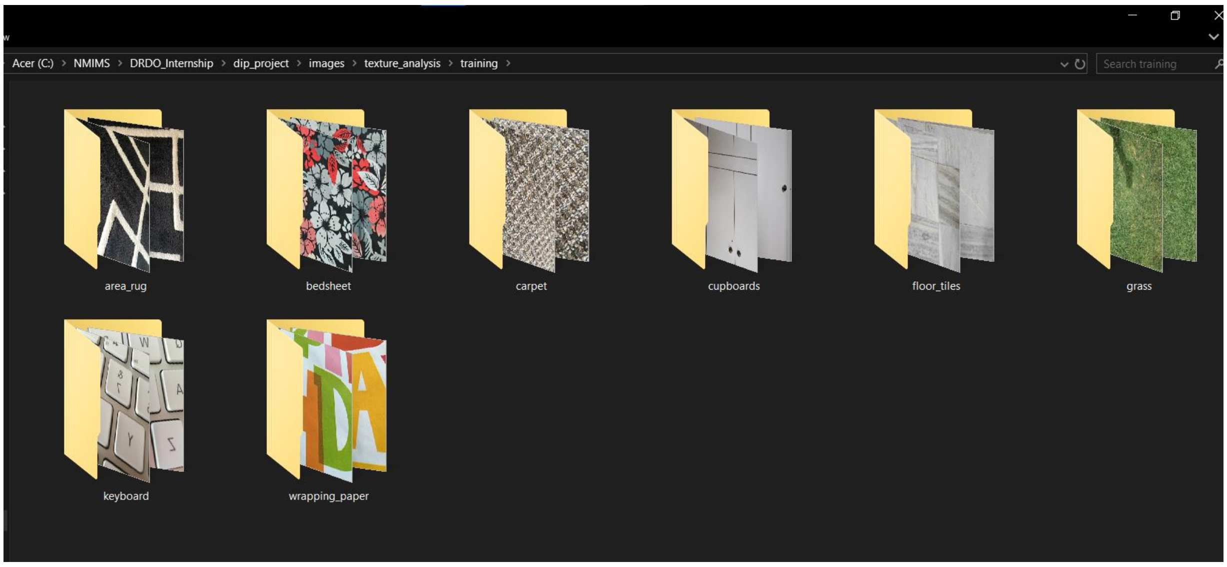

Acer (C:) > NMIMS > DRDO_Internship > dip_project > images > texture_analysis > training
Search training
area_rug
bedsheet
carpet
cupboards
floor_tiles
grass
keyboard
wrapping_paper

Figure 41: The training dataset folder is structured in the above way

4) **Train the SVM classifier**:

- Initialize a Linear Support Vector Machine (SVM) classifier with specified parameters.
- Fit the SVM model using the feature data and labels collected from the training images.

```python
# train a Linear SVM on the data
model = LinearSVC(C=100.0, random_state=42)
model.fit(data, labels)
```

5) **Load, process testing images and print the predictions**:

- Loop over the paths of testing images.
- For each image:
    - Load the image using OpenCV.
    - Convert the image to grayscale.
    - Compute the LBP histogram using the defined function.
    - Predict the class label using the trained SVM model.

```python
# loop over the testing images
for img_path in
paths.list_images(r"images\texture_analysis\testing"):

    # load the image, convert it to grayscale, compute
the histogram,
    # and classify it
    img = cv.imread(img_path)
    gray_img = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    hist = compute_lbp_hist(gray_img)
    prediction = model.predict(hist.reshape(1, -1))

    # display the image and the prediction using
matplotlib
    plt.figure(figsize=(3, 3))
    plt.imshow(cv.cvtColor(img, cv.COLOR_BGR2RGB))
    plt.title(f"Prediction: {prediction[0]}")
    plt.axis("off")
    plt.show()
```

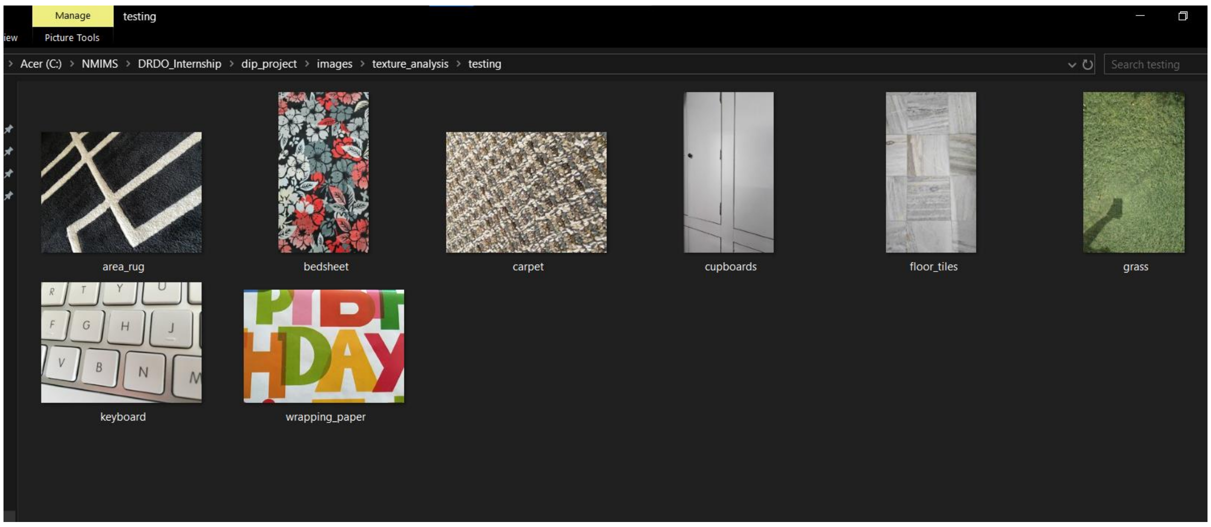


Figure 42: The test set folder is structured in the above way

Prediction outputs:

As we can see in figure 43 (below), the model was able to predict 7 out of 8 images right. Thus, we have an accuracy of 87.5%.

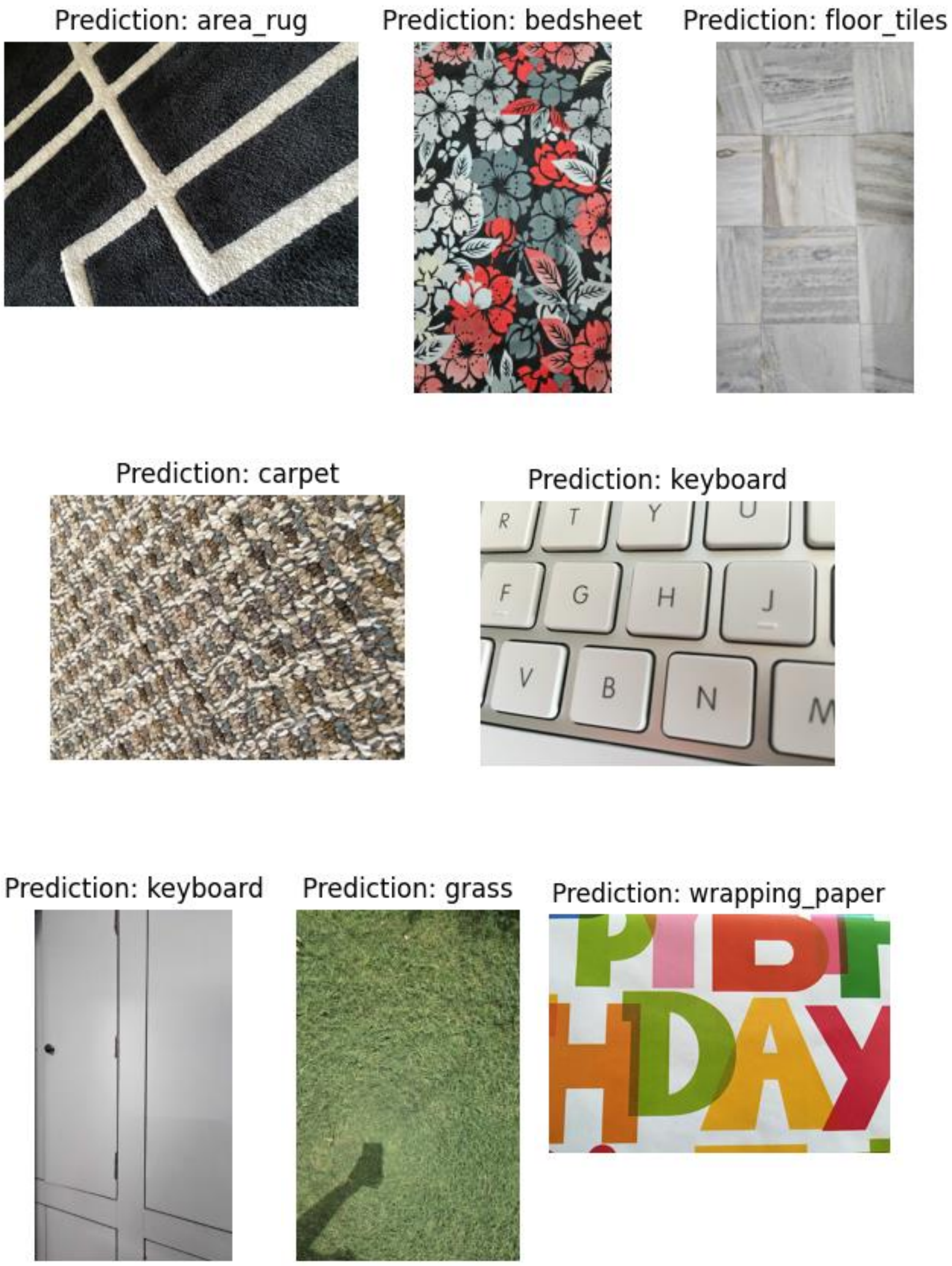


Figure 43: Prediction outputs of the SVM texture classifier

12. CONCLUSION

The culmination of this project represents a comprehensive journey through various advanced techniques in image processing and machine learning, with a particular emphasis on texture analysis and image classification. This report has methodically explored the intricate steps involved in the preprocessing, transformation, analysis, and classification of images, culminating in a robust methodology for classifying images based on their textures using Support Vector Machines (SVM).

Summary of Key Findings and Accomplishments

1. **Preprocessing and Filtering:** Applied Gaussian, Median, and other filters for noise reduction and edge preservation.

2. **Edge Detection:** Utilized Prewitt, Sobel, and Canny methods to highlight image boundaries effectively.

3. **Thresholding:** Implemented simple and adaptive thresholding for image segmentation.

4. **Morphological Operations:** Applied erosion, dilation, opening, closing and magnitude gradient.

5. **Image Transforms:** Explored DCT, IDCT, FT, and IFT to analyze frequency components.

6. **Image Segmentation:** Used the Watershed algorithm to partition images into distinct regions based on intensity gradients.

7. **Histogram Analysis and Equalization:** Enhanced image contrast using Global Histogram Equalization and CLAHE.

8. **Image Quality Assessment (IQA):** Assessed image quality with PSNR, SSIM, and SAM metrics.

9. **Texture Analysis and Feature Extraction:** Extracted texture features using Local Binary Pattern (LBP) histograms.

10. **Classification Using Support Vector Machine (SVM):** Trained an SVM classifier on LBP features, achieving high accuracy in texture-based image classification.

Implications and Future Work

The successful completion of this project underscores the potential of combining advanced image processing techniques with machine learning algorithms for robust image analysis and classification. The methodologies developed and implemented throughout this project can be extended to various real-world applications, including medical imaging, remote sensing, and industrial inspection.

Future work can build upon the foundation laid by this project by exploring the following avenues:

- **Deep Learning for Texture Analysis**: Integrating convolutional neural networks (CNNs) for automated feature extraction and classification to further improve the accuracy and robustness of texture-based image classification.

- **Real-Time Image Processing**: Developing efficient algorithms and leveraging hardware accelerations (e.g., GPUs) for real-time processing and classification of high-resolution images.

- **Multi-Scale Analysis**: Investigating multi-scale texture analysis techniques to capture texture information at different scales, enhancing the comprehensiveness of the texture descriptors.

In conclusion, this project has successfully demonstrated the application of a systematic approach to image preprocessing, feature extraction, and classification, culminating in an effective SVM-based texture classification framework. The insights and methodologies developed herein provide a solid foundation for future research and applications in the field of image processing and machine learning, paving the way for more advanced and specialized techniques to be developed and applied in various domains.

REFERENCES

1. https://medium.com/@jaikochhar06/how-to-evaluate-image-quality-in-python-a-comprehensive-guide-e486a0aa1f60
2. https://scikit-image.org/docs/0.13.x/auto_examples/features_detection/plot_local_binary_pattern.html
3. https://medium.com/@ariesiitr/texture-analysis-using-lbp-e61e87a9056d
4. https://pyimagesearch.com/2015/12/07/local-binary-patterns-with-python-opencv/
5. https://docs.opencv.org/4.x/d2/d96/tutorial_py_table_of_contents_imgproc.html
6. https://www.geeksforgeeks.org/opencv-python-tutorial/
7. https://medium.com/@nimritakoul01/image-processing-using-opencv-python-9c9b83f4b1ca
8. https://www.geeksforgeeks.org/edge-detection-using-prewitt-scharr-and-sobel-operator/
9. https://www.geeksforgeeks.org/discrete-cosine-transform-algorithm-program/
10. https://www.tutorialspoint.com/how-to-find-discrete-cosine-transform-of-an-image-using-opencv-python
11. https://www.geeksforgeeks.org/python-peak-signal-to-noise-ratio-psnr/
12. https://learnopencv.com/edge-detection-using-opencv/
13. Hussein AL-Qinani, Iman. (2019). A Review Paper on Image Quality Assessment Techniques. 6.
14. Bharati, Manish & Liu, Jay & MacGregor, John. (2004). Image Texture Analysis: Methods and Comparisons. Chemometrics and Intelligent Laboratory Systems. 72. 57-71. 10.1016/j.chemolab.2004.02.005.