Introduction to Computer Vision

ARM 354

Lab File

Artificial Intelligence & Machine Learning

(2022 - 2026)

Submitted by:
Sanjay Chaurasia
13119051622
AIML B2 (2022 - 2026)

Submitted to:
Dr. Manisha Parlewar
Assistant Professor
USAR, GGSIPU

University School of Automation & Robotics
GURU GOBIND SINGH INDRAPRASTHA UNIVERSITY
East Delhi Campus, Surajmal vihar
Delhi - 110092

Index

Sr. No	Title of the experiment	Date of Performance	Date of Correction	Sign
1	Basic image processing and image enhancement using point processing			
2	Image enhancement using spatial filtering			
3	Image noise removal using spatial filtering			
4	Image segmentation using k-means clustering			
5	Object recognition using HOG and machine learning.			
6	Camera calibration			

Computer Vision

Name = Sanjay Chaurasia

Registration No: = 13119051622

Lab - 1: Basic Image Processing **Operations**

The objective of this lab is to introduce the student to OpenCV/python, especially for image processing. In this lab, I performed several basic image processing operations using Python. The operations included loading an image, converting it into different formats, converting it to grayscale, and enhancing the image using histogram equalization. The image I used for this lab is a Tiger image.

```
import matplotlib.pyplot as plt
import matplotlib.image as img
from PIL import Image
import numpy as np
```

Reading and Displaying an Image

To start, I loaded a .tif image of a lion from my Google Drive. The image was displayed using Matplotlib without any axis labels.

Path of the Image: /content/drive/MyDrive/CV/1000059030 (300×200)

Shape of the image: Example output – (300, 200, 3)

The imshow function was used to render the image on the screen.

```
image = img.imread("/content/drive/MyDrive/CV/1000059030 (300×200)")
print("Image Dimensions:", image.shape)
plt.imshow(image)
plt.axis('off')
plt.title("Original Tiger Image")
plt.show()
```

```
Image Dimensions: (200, 300, 3)
```

Converting the Image to Different Formats

Images come in formats like .tif, .png, .jpg, .webp, etc. Each serves different purposes based on compression, quality, and compatibility. I converted my lion image to both PNG and WEBP formats and displayed all three side by side for comparison.

This step helps understand format conversion and visual differences between compression types.

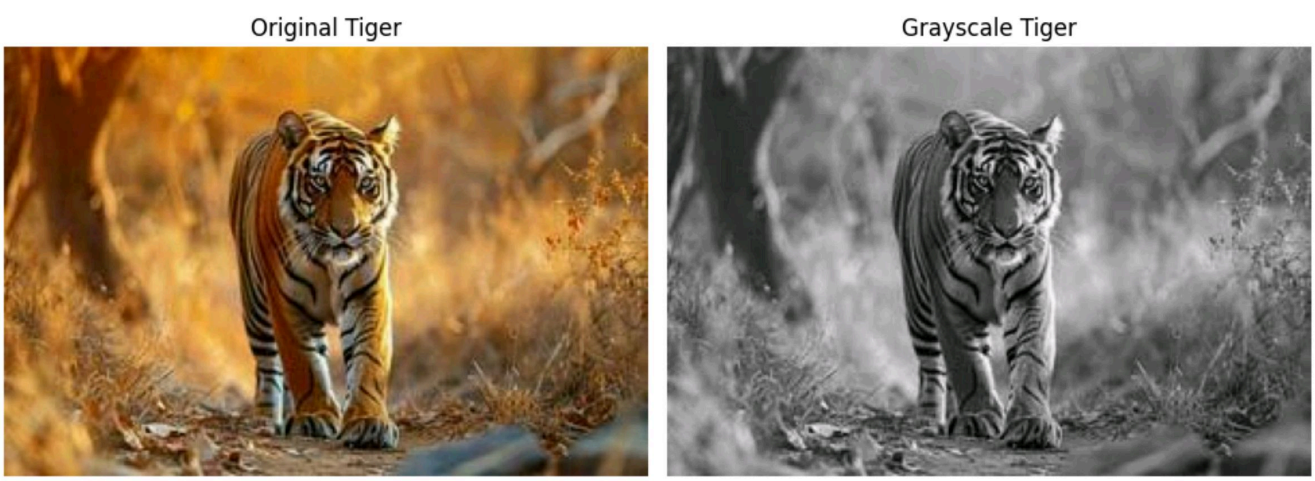
```
image_pil = Image.open("/content/drive/MyDrive/CV/1000059030 (300×200)")
image_pil.save("/content/lion.png")
image_pil.save("/content/lion.webp")
converted_png = Image.open("/content/lion.png")
converted_webp = Image.open("/content/lion.webp")
fig, axes = plt.subplots(1, 3, figsize=(12, 5))
axes[0].imshow(image_pil)
axes[0].set_title("Tiger - TIF")
axes[0].axis('off')

axes[1].imshow(converted_png)
axes[1].set_title("Tiger - PNG")
axes[1].axis('off')

axes[2].imshow(converted_webp)
axes[2].set_title("Tiger - WEBP")
axes[2].axis('off')

plt.tight_layout()
plt.show()
```

Converting an Image to Grayscale

Color images carry three channels (RGB), while grayscale images have a single channel representing brightness or intensity.

I converted the Tiger image into grayscale using the convert('L') method and visualized both the original and grayscale versions. This step is useful for tasks that do not require color, like edge detection or thresholding.

```
gray_img = image_pil.convert('L')
gray_array = np.array(gray_img)
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(image_pil)
axes[0].set_title("Original Tiger")
axes[0].axis('off')
axes[1].imshow(gray_array, cmap='gray')
axes[1].set_title("Grayscale Tiger")
axes[1].axis('off')
plt.tight_layout()
plt.show()
```



Perform Image enhancement operations

Objectives: Image enhancement improves visual quality and highlights important features in the image. This can be useful when an image has poor contrast, uneven lighting, or needs details to be highlighted.

Need for Image Enhancement: Raw images may suffer from issues like blur, low contrast, or noise. Enhancement techniques can make the image clearer and more suitable for analysis.

Types of Image Enhancement Methods:

1) Contrast Enhancement

Makes dark and bright areas more distinguishable.

Methods: Histogram Equalization, Contrast Stretching

2) Noise Reduction

Removes unwanted noise that may come from image acquisition.

Methods: Median Filter, Gaussian Smoothing

3) Sharpening

Makes edges and fine details stand out more clearly.

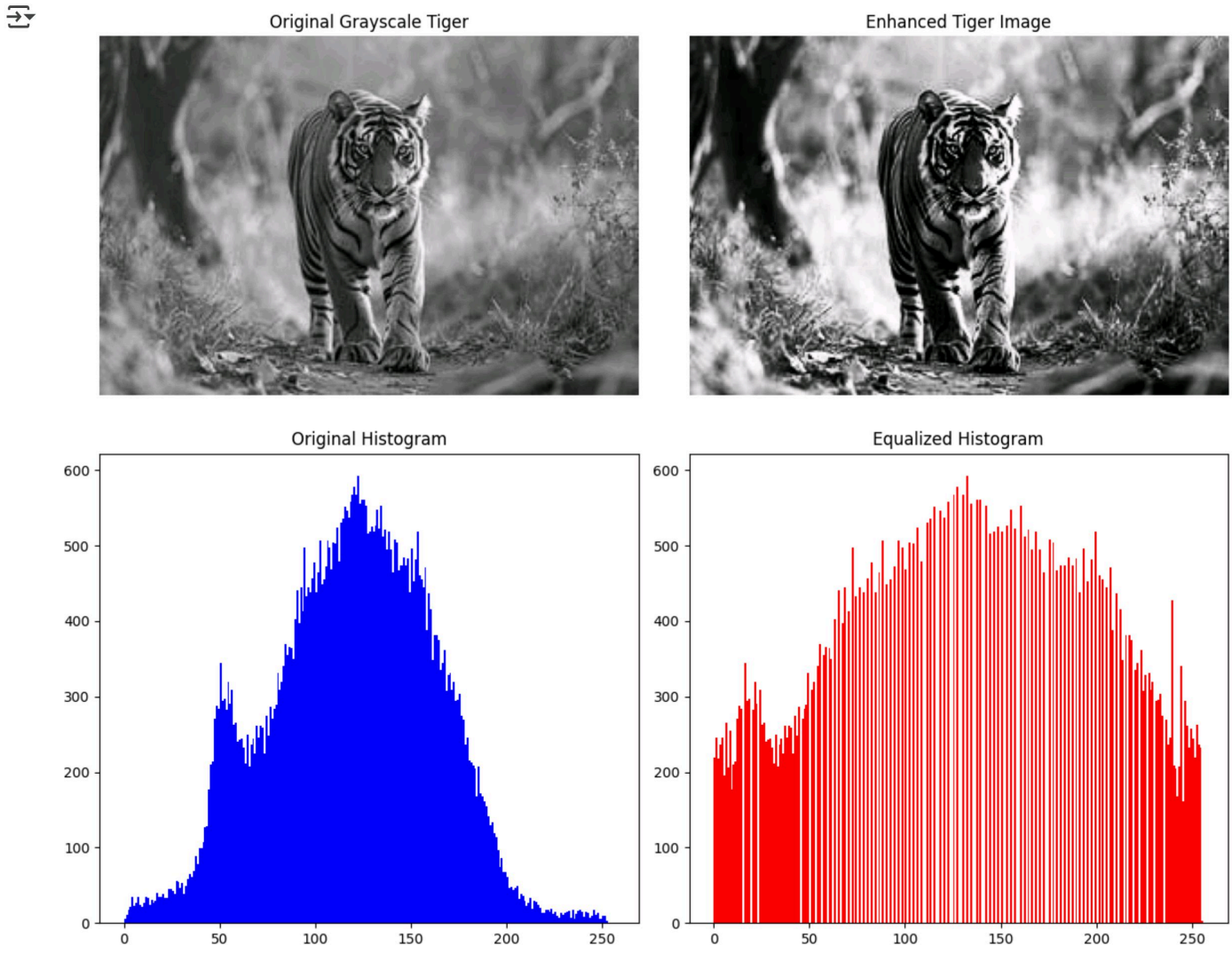
Methods: Laplacian Filtering, Unsharp Masking

Performing Image Enhancement Operations (Histogram Equalization):

Histogram equalization redistributes pixel intensities to cover a wider range and increase contrast.

```
hist, bins = np.histogram(gray_array.flatten(), 256, [0, 256])
cdf = hist.cumsum()
cdf_m = np.ma.masked_equal(cdf, 0)
cdf_m = (cdf_m - cdf_m.min()) * 255 / (cdf_m.max() - cdf_m.min())
cdf = np.ma.filled(cdf_m, 0).astype('uint8')
equalized_img = cdf[gray_array]

fig, axes = plt.subplots(2, 2, figsize=(12, 10))

axes[0, 0].imshow(gray_array, cmap='gray')
axes[0, 0].set_title('Original Grayscale Tiger')
axes[0, 0].axis('off')

axes[0, 1].imshow(equalized_img, cmap='gray')
axes[0, 1].set_title('Enhanced Tiger Image')
axes[0, 1].axis('off')

axes[1, 0].hist(gray_array.ravel(), bins=256, range=(0, 256), color='blue')
axes[1, 0].set_title('Original Histogram')
```

```
axes[1, 1].hist(equalized_img.ravel(), bins=256, range=(0, 256), color='red')
axes[1, 1].set_title('Equalized Histogram')

plt.tight_layout()
plt.show()
```



.

Conclusion:

In this lab, I explored basic image processing tasks using Python. The tasks included loading, displaying, format conversion, grayscale conversion, and image enhancement using histogram equalization. I applied all operations to a lion image stored in .tif format.

Key Takeaways:

Image Loading and Display: Learned to load and display images using matplotlib.

Image Format Conversion: Used PIL to convert images into .png and .webp.

Grayscale Conversion: Converted color images to grayscale using the convert('L') method.

Histogram Equalization: Applied histogram equalization to improve contrast and highlight details.

This lab provided a foundation for understanding how images work in Python and how to process them effectively for further use in computer vision tasks.

Computer Vision

Name: Sanjay Chaurasia

Registration No: 13119051622

Lab - 2 : Image Enhancement using Spatial Domain Filtering

Objectives:

The objective of this lab is to introduce the student to OpenCV/python, especially for image processing.

1. To Understand convolution operation in images
2. To study various spatial filters
3. Edge detection using gradient and Laplacian operator
4. Laplacian of Gaussian (LOG)

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force
```

Reading and displaying an image in python

```
import matplotlib.pyplot as plt
import matplotlib.image as img
image = img.imread(r"/content/drive/MyDrive/Aadhar/img5")
plt.imshow(image)
plt.axis("off")
image_path = "/content/drive/MyDrive/Aadhar/img5"
```

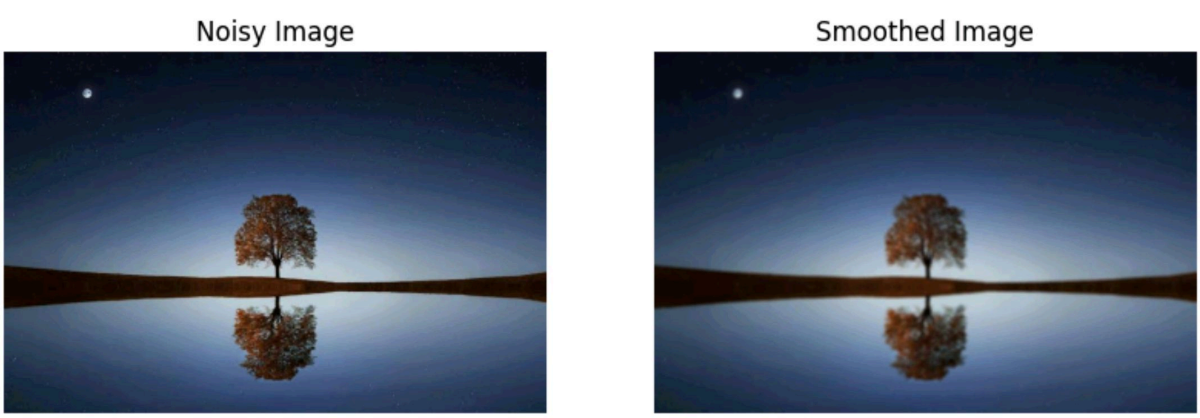
Averaging/Smoothing Mask in Image Processing

Steps for Applying Averaging/Smoothing Mask

1. Read the Image: Load the image using OpenCV.
2. Convert to RGB Format: OpenCV loads images in BGR format, so we convert it to RGB for correct visualization.
3. Convert to Grayscale: Convert the image to grayscale for processing.

4. Apply Averaging Filter: Use a kernel (e.g., 5x5) where each pixel is replaced by the average of its neighbors.
5. Display Results: Show the original and smoothed images using Matplotlib

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
#import matplotlib.image as img #This is never used so we can delete it

def apply_smoothing_mask(image_path, kernel_size=5):
    image = cv2.imread(image_path)
    if image is None:
        print("Error: Unable to load image.")
        return
    # Converting image from BGR to RGB
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    # Averaging filter kernel
    kernel = np.ones((kernel_size, kernel_size), np.float32) / (kernel_size ** 2)
    # Applying the filter using cv2.filter2D
    smoothed_image = cv2.filter2D(image, -1, kernel)
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.imshow(image)
    plt.title("Noisy Image")
    plt.axis("off")
    plt.subplot(1, 2, 2)
    plt.imshow(smoothed_image)
    plt.title("Smoothed Image")
    plt.axis("off")
    plt.show()

apply_smoothing_mask(image_path)
```

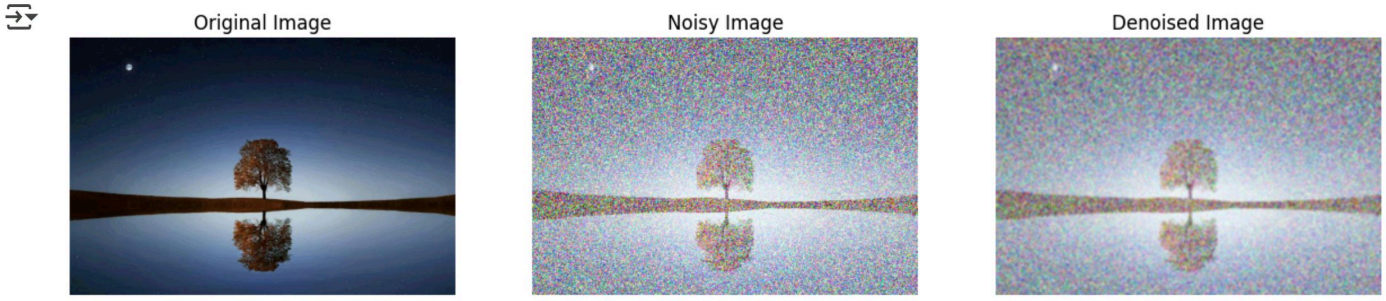


Averaging mask for noise removalAveraging mask for noise removal

Gaussian noise is randomly added to an image, following a normal distribution. This simulates real-world noise introduced by cameras or transmission errors. It is commonly used for testing noise removal algorithms.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def apply_smoothing_mask(image_path, kernel_size=5):
  image = cv2.imread(image_path)
  if image is None:
    print("Error: Unable to load image.")
    return

  image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
  # Adding Gaussian noise
  noise = np.random.normal(0, 25, image.shape).astype(np.uint8)
  noisy_image = cv2.add(image, noise)
```

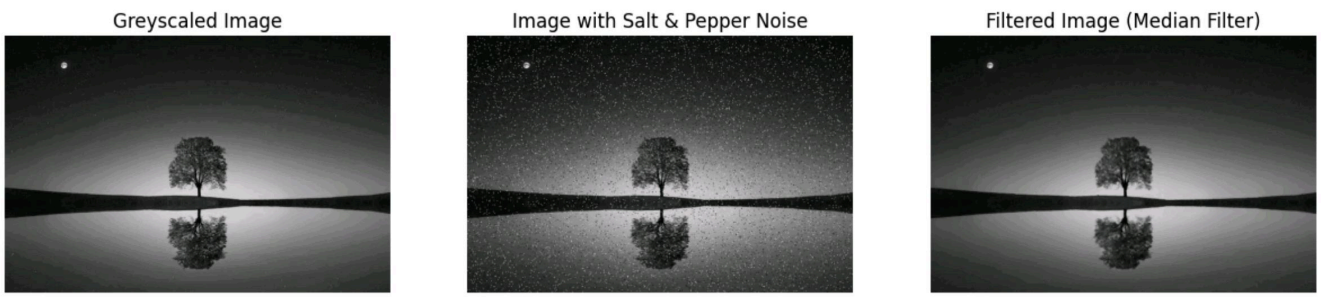
```
  kernel = np.ones((kernel_size, kernel_size), np.float32) / (kernel_size ** 2)
  smoothed_image = cv2.filter2D(noisy_image, -1, kernel)

  plt.figure(figsize=(15, 5))
  plt.subplot(1, 3, 1)
  plt.imshow(image)
  plt.title("Original Image")
  plt.axis("off")
  plt.subplot(1, 3, 2)
  plt.imshow(noisy_image)
  plt.title("Noisy Image")
  plt.axis("off")
  plt.subplot(1, 3, 3)
  plt.imshow(smoothed_image)
  plt.title("Denoised Image")
  plt.axis("off")
  plt.show()


apply_smoothing_mask(image_path)
```



Median Filter for Salt and Pepper Noise

```
def add_salt_and_pepper_noise(image, salt_prob=0.02, pepper_prob=0.02):

    noisy_image = np.copy(image)
    total_pixels = image.size
    num_salt = int(total_pixels * salt_prob)
    num_pepper = int(total_pixels * pepper_prob)

    # Add salt noise (white pixels)
    coords = [np.random.randint(0, i - 1, num_salt) for i in image.shape]
    noisy_image[tuple(coords)] = 255

    # Add pepper noise (black pixels)
    coords = [np.random.randint(0, i - 1, num_pepper) for i in image.shape]
    noisy_image[tuple(coords)] = 0

    return noisy_image

def apply_median_filter(image_path, kernel_size=3):

    image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
    if image is None:
        print("Error: Unable to load image.")
        return

    noisy_image = add_salt_and_pepper_noise(image)
    filtered_image = cv2.medianBlur(noisy_image, kernel_size)

    plt.figure(figsize=(15, 5))
    plt.subplot(1, 3, 1)
    plt.imshow(image, cmap='gray')
    plt.title("Greyscaled Image")
    plt.axis("off")
```

```
    plt.subplot(1, 3, 2)
    plt.imshow(noisy_image, cmap='gray')
    plt.title("Image with Salt & Pepper Noise")
    plt.axis("off")
    plt.subplot(1, 3, 3)
    plt.imshow(filtered_image, cmap='gray')
    plt.title("Filtered Image (Median Filter)")
    plt.axis("off")
    plt.show()


apply_median_filter(image_path)
```



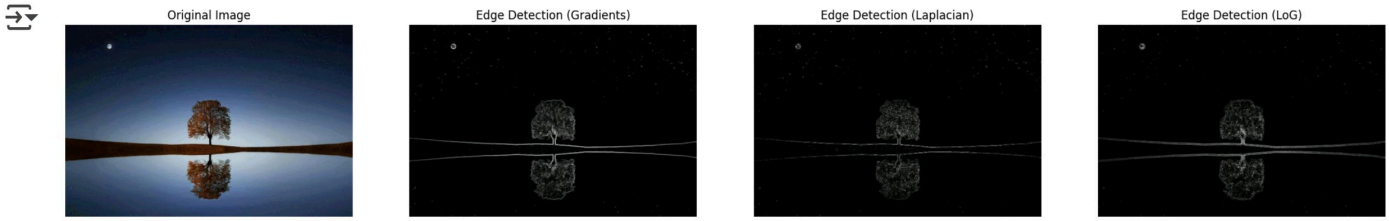
Edge detection using 1st order derivative (Image Gradients)

Laplacian edge detection is a second-order derivative method used to highlight regions of rapid intensity change (edges). It computes the Laplacian of the image and enhances edges while being sensitive to noise.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def apply_edge_detection(image_path, kernel_size=5):
    image = cv2.imread(image_path)
    if image is None:
        print("Error: Unable to load image.")
        return

    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    gray_image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

    # Sobel edge detection
    sobel_x = cv2.Sobel(gray_image, cv2.CV_64F, 1, 0, ksize=3)
    sobel_y = cv2.Sobel(gray_image, cv2.CV_64F, 0, 1, ksize=3)
    gradient_magnitude = np.sqrt(sobel_x**2 + sobel_y**2)
    gradient_magnitude = np.uint8(255 * gradient_magnitude / np.max(gradient_magnitude))

    # Laplacian edge detection
    laplacian = cv2.Laplacian(gray_image, cv2.CV_64F, ksize=3)
    laplacian = np.uint8(255 * np.abs(laplacian) / np.max(np.abs(laplacian)))

    # Laplacian of Gaussian (LoG) edge detection
    log_image = cv2.GaussianBlur(gray_image, (kernel_size, kernel_size), 0)
    log_image = cv2.Laplacian(log_image, cv2.CV_64F, ksize=3)
    log_image = np.uint8(255 * np.abs(log_image) / np.max(np.abs(log_image)))

    # Display the results
    plt.figure(figsize=(25, 5))
    plt.subplot(1, 4, 1)
    plt.imshow(image)
    plt.title("Original Image")
    plt.axis("off")
    plt.subplot(1, 4, 2)
    plt.imshow(gradient_magnitude, cmap='gray')
    plt.title("Edge Detection (Gradients)")
```

```
    plt.axis("off")
    plt.subplot(1, 4, 3)
    plt.imshow(laplacian, cmap='gray')
    plt.title("Edge Detection (Laplacian)")
    plt.axis("off")
    plt.subplot(1, 4, 4)
    plt.imshow(log_image, cmap='gray')
    plt.title("Edge Detection (LoG)")
    plt.axis("off")
    plt.show()
```

```
apply_edge_detection(image_path)
```

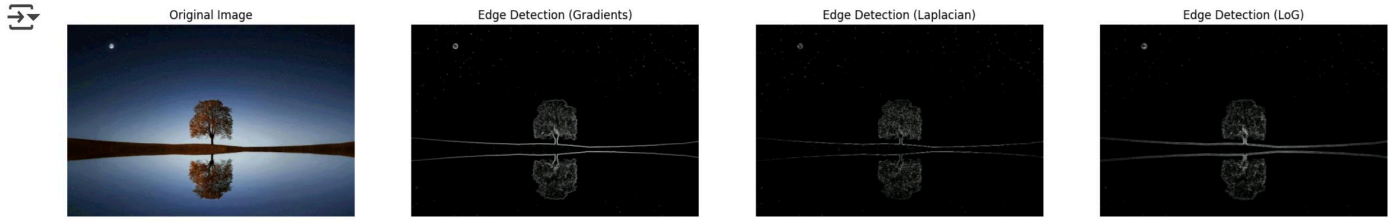


Edge detection using 2nd order deriavtive (Laplacian)

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def apply_edge_detection(image_path, kernel_size=5):

    # Read the image
    image = cv2.imread(image_path)
    if image is None:
        print("Error: Unable to load image.")
        return

    # Convert the image to RGB
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # Convert the image to grayscale
    gray_image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

    # Sobel edge detection
    sobel_x = cv2.Sobel(gray_image, cv2.CV_64F, 1, 0, ksize=3)
    sobel_y = cv2.Sobel(gray_image, cv2.CV_64F, 0, 1, ksize=3)
    gradient_magnitude = np.sqrt(sobel_x**2 + sobel_y**2)
    gradient_magnitude = np.uint8(255 * gradient_magnitude / np.max(gradient_magnitude))

    # Laplacian edge detection
    laplacian = cv2.Laplacian(gray_image, cv2.CV_64F, ksize=3)
    laplacian = np.uint8(255 * np.abs(laplacian) / np.max(np.abs(laplacian)))

    # Laplacian of Gaussian (LoG) edge detection
    log_image = cv2.GaussianBlur(gray_image, (kernel_size, kernel_size), 0)
    log_image = cv2.Laplacian(log_image, cv2.CV_64F, ksize=3)
    log_image = np.uint8(255 * np.abs(log_image) / np.max(np.abs(log_image)))

    # Display the results
    plt.figure(figsize=(25, 5))
    plt.subplot(1, 4, 1)
    plt.imshow(image)
    plt.title("Original Image")
    plt.axis("off")
    plt.subplot(1, 4, 2)
    plt.imshow(gradient_magnitude, cmap='gray')  # Changed to gradient_magnitude
    plt.title("Edge Detection (Gradients)")  # Changed title
    plt.axis("off")
```

```
    plt.subplot(1, 4, 3)
    plt.imshow(laplacian, cmap='gray')
    plt.title("Edge Detection (Laplacian)")
    plt.axis("off")
    plt.subplot(1, 4, 4)
    plt.imshow(log_image, cmap='gray')
    plt.title("Edge Detection (LoG)")
    plt.axis("off")
    plt.show()
```

```
apply_edge_detection(image_path)
```



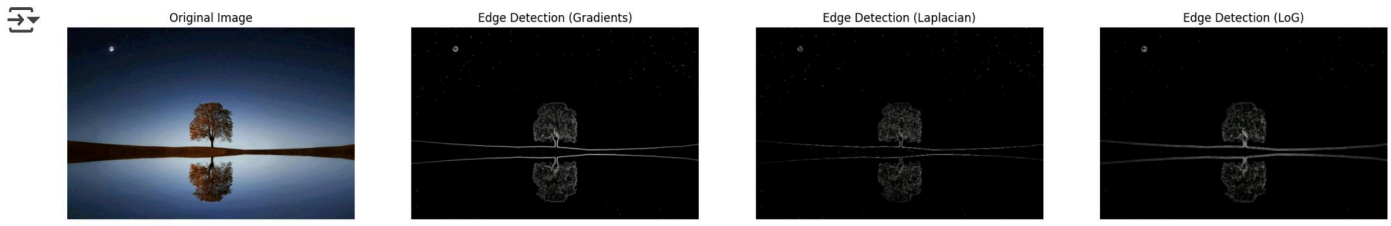
Laplacian of Gaussian (LoG)

LoG combines Gaussian smoothing and Laplacian edge detection. The Gaussian filter reduces noise before applying the Laplacian operator. This method is useful for detecting edges in noisy images while reducing false edges caused by noise.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def apply_edge_detection(image_path, kernel_size=5):
  """Applies edge detection to an image using gradients, Laplacian, and LoG.

  Args:
    image_path: The path to the image file.
    kernel_size: The size of the kernel for LoG.
  """
  image = cv2.imread(image_path)
  if image is None:
    print("Error: Unable to load image.")
    return

  image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
  gray_image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

  # Computing image gradients using Sobel operator
  sobel_x = cv2.Sobel(gray_image, cv2.CV_64F, 1, 0, ksize=3)
  sobel_y = cv2.Sobel(gray_image, cv2.CV_64F, 0, 1, ksize=3)
  gradient_magnitude = np.sqrt(sobel_x**2 + sobel_y**2)
  gradient_magnitude = np.uint8(255 * gradient_magnitude / np.max(gradient_magnitude))

  # Edge detection using Laplacian
  laplacian = cv2.Laplacian(gray_image, cv2.CV_64F, ksize=3)
  laplacian = np.uint8(255 * np.abs(laplacian) / np.max(np.abs(laplacian)))

  # Edge detection using Laplacian of Gaussian (LoG)
  log_image = cv2.GaussianBlur(gray_image, (kernel_size, kernel_size), 0)
  log_image = cv2.Laplacian(log_image, cv2.CV_64F, ksize=3)
  log_image = np.uint8(255 * np.abs(log_image) / np.max(np.abs(log_image)))

  # Display the results
  plt.figure(figsize=(25, 5))
  plt.subplot(1, 4, 1)
  plt.imshow(image)
  plt.title("Original Image")
  plt.axis("off")
```

```
plt.subplot(1, 4, 2)
plt.imshow(gradient_magnitude, cmap='gray')
plt.title("Edge Detection (Gradients)")
plt.axis("off")
plt.subplot(1, 4, 3)
plt.imshow(laplacian, cmap='gray')
plt.title("Edge Detection (Laplacian)")
plt.axis("off")
plt.subplot(1, 4, 4)
plt.imshow(log_image, cmap='gray')
plt.title("Edge Detection (LoG)")
plt.axis("off")
plt.show()
```

```
apply_edge_detection(image_path)
```



Conclusion

Spatial filters play a crucial role in image processing, helping to enhance, smooth, and detect edges in images. Smoothing filters, such as averaging and Gaussian filters, effectively reduce noise and blur images, making them suitable for preprocessing before edge detection. On the other hand, edge detection techniques like the Laplacian and Laplacian of Gaussian (LoG) enhance abrupt intensity changes, allowing for clear object boundaries. The combination of these techniques ensures better image analysis and feature extraction, which is essential for applications in computer vision and image recognition

Computer Vision

Name = Sanjay Chaurasia

Registration No: = 13119051622

Lab - 3 : Adding and removing image noise in python.

Objective:

To see the effect of various noises i.e Gaussian, Uniform, Impulse noise and methods to remove them

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
```

```
img=cv2.imread("/content/drive/MyDrive/Aadhar/img2",0)
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
print(img.shape)
```

```
(480, 640)
```

Adding Noise to Image

Gaussian Noise

Gaussian noise is a statistical noise having the probability density function equal to that of a normal distribution. Random Gaussian function is added to Image function to generate this noise. Since it arises in amplifiers and detectors, it is also known as 'electronic noise'. It is caused by the discrete nature of radiation of warm objects.

To create the Gaussian noise, we first create a zero image with the same dimensions of the original image. We then use a random distribution to determine the pixel values of the noise (in this case with a mean of 128 and a sigma of 20)

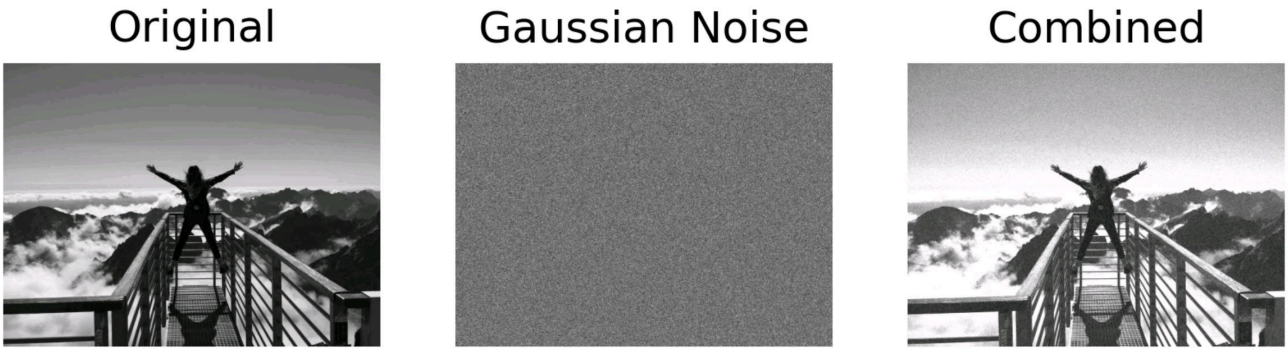
```
gauss_noise=np.zeros((480, 640),dtype=np.uint8)
cv2.randn(gauss_noise,128,20)
gauss_noise=(gauss_noise*0.5).astype(np.uint8)
```

```
gn_img=cv2.add(img,gauss_noise)
```

```
fig=plt.figure(dpi=300)

fig.add_subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis("off")
plt.title("Original")

fig.add_subplot(1,3,2)
plt.imshow(gauss_noise,cmap='gray')
plt.axis("off")
plt.title("Gaussian Noise")
```

```
fig.add_subplot(1,3,3)
plt.imshow(gn_img,cmap='gray')
plt.axis("off")
plt.title("Combined")
```

```
Text(0.5, 1.0, 'Combined')
```



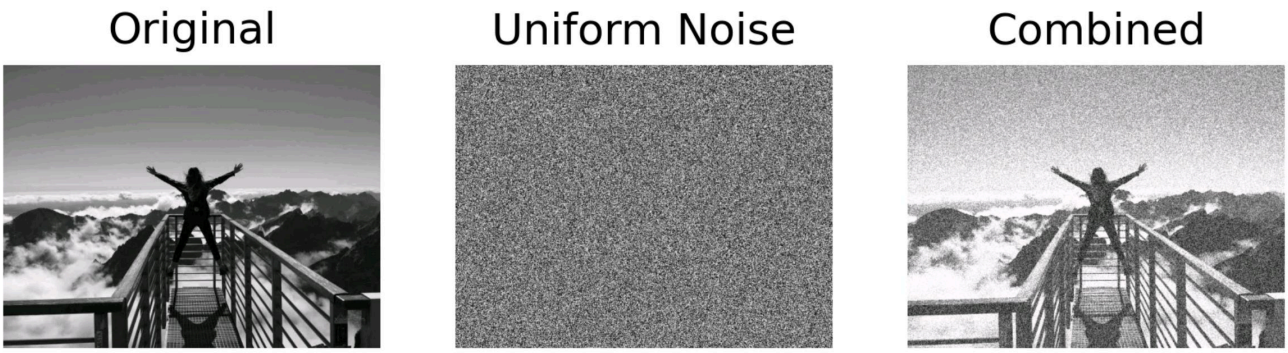
Uniform Noise

Contrary to Gaussian noise, Uniform noise is a signal dependent (unless dithering is caused or applied) that follows a uniform distribution. It is caused by the quantization of the pixels of an image to a number of discrete levels. It is generally created when analog data is converted to digital form, and is not often encountered in real-world imaging systems.

To create a Uniform noise, we create a uniform distribution whose lower and upper bounds are the minimum and maximum pixel values (0 and 255 respectively) along the dimensions of the image.

```
uni_noise=np.zeros((480, 640),dtype=np.uint8)
cv2.randu(uni_noise,0,255)
uni_noise=(uni_noise*0.5).astype(np.uint8)


un_img=cv2.add(img,uni_noise)


fig=plt.figure(dpi=300)

fig.add_subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis("off")
plt.title("Original")

fig.add_subplot(1,3,2)
plt.imshow(uni_noise,cmap='gray')
plt.axis("off")
plt.title("Uniform Noise")

fig.add_subplot(1,3,3)
plt.imshow(un_img,cmap='gray')
plt.axis("off")
plt.title("Combined")
```

```
Text(0.5, 1.0, 'Combined')
```



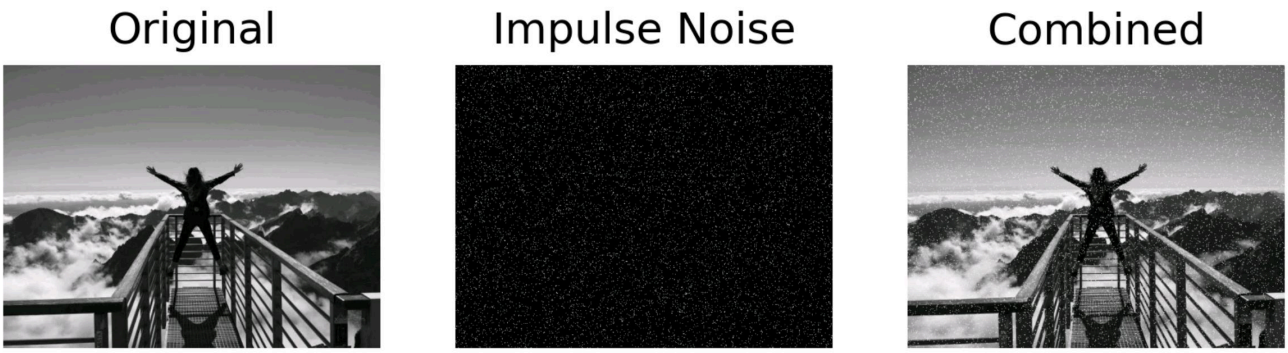
Impulse Noise

Impulse or "Salt and Pepper" noise is the sparse occurance of maximum (255) and minimum (0) pixel values in an image. This can be noticed as the presence of black pixels in bright regions and white pixels in dark regions. This type of noise is caused due to sharp and sudden disturbances in the image signal, and is mainly generated by errors in analog to digital conversion or bit transmission.

To create a Salt and Pepper noise, we first create a distribution similar to that used in Uniform noise and apply binary thresholding to create a grid of black and white pixels. The intensity of the noise can be easily altered by changing the threshold value.

```
imp_noise=np.zeros((480, 640),dtype=np.uint8)
cv2.randu(imp_noise,0,255)
imp_noise=cv2.threshold(imp_noise,245,255,cv2.THRESH_BINARY)[1]


in_img=cv2.add(img,imp_noise)


fig=plt.figure(dpi=300)

fig.add_subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis("off")
plt.title("Original")

fig.add_subplot(1,3,2)
plt.imshow(imp_noise,cmap='gray')
plt.axis("off")
plt.title("Impulse Noise")

fig.add_subplot(1,3,3)
plt.imshow(in_img,cmap='gray')
plt.axis("off")
plt.title("Combined")
```

```
Text(0.5, 1.0, 'Combined')
```



Removing Noise from Image

Using inbuilt function fastNlMeansDenoising

This function uses Non-local Means Denoising algorithm, and expects greyscale image with Gaussian white noise.

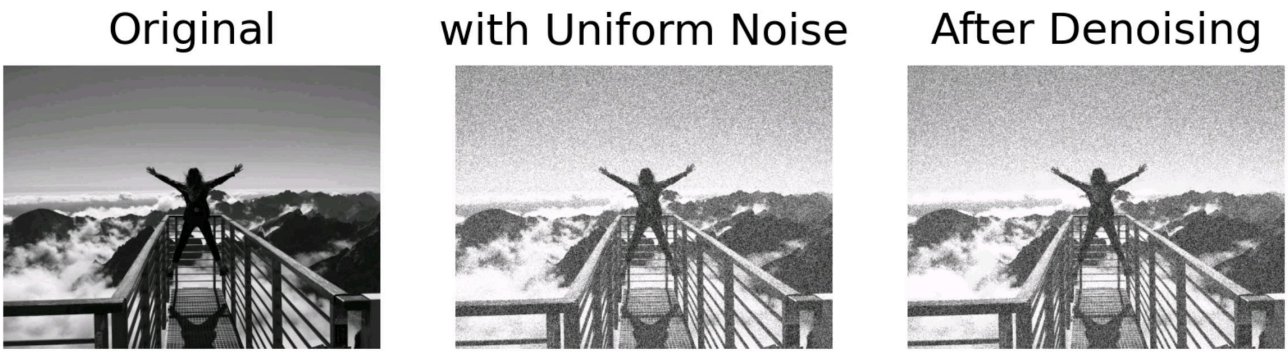
```
denoised1=cv2.fastNlMeansDenoising(gn_img,None,10,10)

fig=plt.figure(dpi=300)
fig.add_subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis("off")
plt.title("Original")

fig.add_subplot(1,3,2)
plt.imshow(gn_img,cmap='gray')
plt.axis("off")
plt.title("with Gaussian Noise")

fig.add_subplot(1,3,3)
plt.imshow(denoised1,cmap='gray')
plt.axis("off")
plt.title("After Denoising")
```

```
Text(0.5, 1.0, 'After Denoising')
```


```
denoised2=cv2.fastNlMeansDenoising(un_img,None,10,10)

fig=plt.figure(dpi=300)
fig.add_subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis("off")
plt.title("Original")

fig.add_subplot(1,3,2)
```

```
plt.imshow(un_img,cmap='gray')
plt.axis("off")
plt.title("with Uniform Noise")

fig.add_subplot(1,3,3)
plt.imshow(denoised2,cmap='gray')
plt.axis("off")
plt.title("After Denoising")
```

```
Text(0.5, 1.0, 'After Denoising')
```




```
denoised3=cv2.fastNlMeansDenoising(in_img,None,10,10)

fig=plt.figure(dpi=300)
fig.add_subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis("off")
plt.title("Original")

fig.add_subplot(1,3,2)
plt.imshow(in_img,cmap='gray')
plt.axis("off")
plt.title("with Impulse Noise")

fig.add_subplot(1,3,3)
plt.imshow(denoised3,cmap='gray')
plt.axis("off")
plt.title("After Denoising")
```

```
Text(0.5, 1.0, 'After Denoising')
```

Inference: Slightly effective against Gaussian Noise.

Using Median Blur

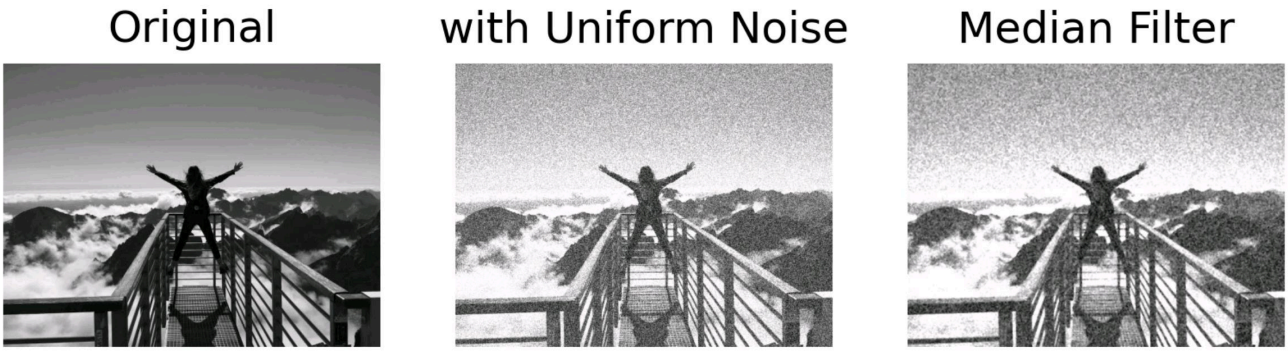
The Median Filter takes all the pixels under a kernel area and replaces the center element with the median value. It is meant to be highly effective against Impulse noise.

```
blurred1=cv2.medianBlur(gn_img,3)
blurred2=cv2.medianBlur(un_img,3)
blurred3=cv2.medianBlur(in_img,3)


fig=plt.figure(dpi=300)
fig.add_subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis("off")
plt.title("Original")

fig.add_subplot(1,3,2)
plt.imshow(gn_img,cmap='gray')
plt.axis("off")
plt.title("with Gaussian Noise")

fig.add_subplot(1,3,3)
plt.imshow(blurred1,cmap='gray')
plt.axis("off")
plt.title("Median Filter")
```

```
Text(0.5, 1.0, 'Median Filter')
```


```
fig=plt.figure(dpi=300)
fig.add_subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis("off")
plt.title("Original")

fig.add_subplot(1,3,2)
plt.imshow(un_img,cmap='gray')
plt.axis("off")
plt.title("with Uniform Noise")

fig.add_subplot(1,3,3)
plt.imshow(blurred2,cmap='gray')
plt.axis("off")
plt.title("Median Filter")
```

```
Text(0.5, 1.0, 'Median Filter')
```




```
fig=plt.figure(dpi=300)
```

```
fig.add_subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis("off")
plt.title("Original")

fig.add_subplot(1,3,2)
plt.imshow(in_img,cmap='gray')
plt.axis("off")
plt.title("with Impulse Noise")

fig.add_subplot(1,3,3)
plt.imshow(blurred3,cmap='gray')
plt.axis("off")
plt.title("Median Filter")
```

```
Text(0.5, 1.0, 'Median Filter')
```

Inference: Slightly effective against Gaussian Noise, considerably effective against Impulse Noise.

Gaussian Blur

This method employs a Gaussian Kernel in place of a box filter and is said to be effective against Gaussian Noise.

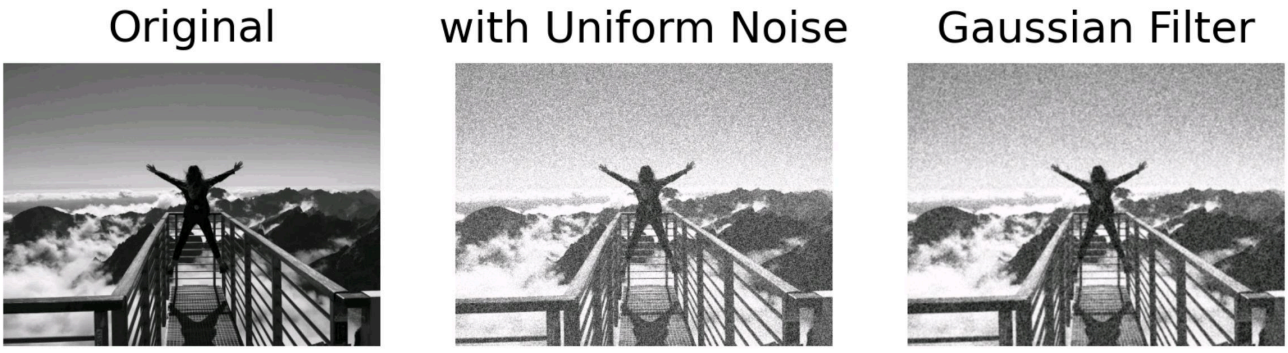
```
blurred21=cv2.GaussianBlur(gn_img,(3,3),0)
blurred22=cv2.GaussianBlur(un_img,(3,3),0)
blurred23=cv2.GaussianBlur(in_img,(3,3),0)
```

```
fig=plt.figure(dpi=300)
fig.add_subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis("off")
plt.title("Original")

fig.add_subplot(1,3,2)
plt.imshow(gn_img,cmap='gray')
plt.axis("off")
plt.title("with Gaussian Noise")

fig.add_subplot(1,3,3)
plt.imshow(blurred21,cmap='gray')
plt.axis("off")
plt.title("Gaussian Filter")
```

```
Text(0.5, 1.0, 'Gaussian Filter')
```


```
fig=plt.figure(dpi=300)
fig.add_subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis("off")
plt.title("Original")

fig.add_subplot(1,3,2)
plt.imshow(un_img,cmap='gray')
plt.axis("off")
plt.title("with Uniform Noise")

fig.add_subplot(1,3,3)
plt.imshow(blurred22,cmap='gray')
plt.axis("off")
plt.title("Gaussian Filter")
```

```
Text(0.5, 1.0, 'Gaussian Filter')
```




```
fig=plt.figure(dpi=300)
fig.add_subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis("off")
plt.title("Original")

fig.add_subplot(1,3,2)
plt.imshow(in_img,cmap='gray')
plt.axis("off")
plt.title("with Impulse Noise")

fig.add_subplot(1,3,3)
plt.imshow(blurred23,cmap='gray')
plt.axis("off")
plt.title("Gaussian Filter")
```

```
Text(0.5, 1.0, 'Gaussian Filter')
```

- Inference: Slightly Effective against Gaussian Blur.

Start coding or generate with AI.

Computer Vision

Name = Sanjay Chaurasia

Registration No: = 13119051622

Lab : Image Segmentation with K Means Clustering

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force
```

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
```

```
import os
```

```
import cv2
import matplotlib.pyplot as plt
```

1. Definition and Explanation of Image Segmentation:

Image segmentation is the classification of an image into different groups.

Image segmentation is the task of partitioning an image into multiple segments. In semantic segmentation, all pixels that are part of the same object type get assigned to the same segment. For example, in a selfdriving car's vision system, all pixels that are part of a pedestrian's image might be assigned to the "pedestrian" segment (there would be one segment containing all the pedestrians).In some applications, this may be sufficient. For example, if you want to analyze satellite images to measure how much total forest area there is in a region, color segmentation may be just fine.

Image segmentation is the process of partitioning a digital image into multiple distinct regions containing each pixel(sets of pixels, also known as superpixels) with similar attributes.

The goal of Image segmentation is to change the representation of an image into something that is more meaningful and easier to analyze.

Image segmentation is typically used to locate objects and boundaries(lines, curves, etc.) in images. More precisely, Image Segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics.

2.Where Can We Use Image Segmentation:

1. In Autonomous Vehicles: they need sensory input devices like cameras, radar, and lasers to allow the car to perceive the world around it, creating a digital map. Autonomous driving is not even possible without object detection which itself involves image classification/segmentation.
2. Healthcare Industry:if we talk about Cancer, even in today's age of technological advancements, cancer can be fatal if we don't identify it at an early stage. Detecting cancerous cell(s) as quickly as possible can potentially save millions of lives. The shape of the cancerous cells plays a vital role in determining the severity of cancer which can be identified using image classification algorithms.

```
img = plt.imread("/content/drive/MyDrive/Aadhar/img4")
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x780b8cc44050>
```

Start coding or generate with AI.

```
from mpl_toolkits.mplot3d import Axes3D
import cv2
img = plt.imread("/content/drive/MyDrive/Aadhar/img4")
#img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
r, g, b = cv2.split(img)
r = r.flatten()
g = g.flatten()
b = b.flatten()#plotting
fig = plt.figure()
ax = Axes3D(fig)
ax.scatter(r, g, b)
plt.show()
#From the plot one can easily see that the data points are forming groups –
#some places in a graph are more dense, which we can think as different colors' dominance on the image.
```

```
<Figure size 640x480 with 0 Axes>
```

3. K Means Algorithm for Image Segmentation:

K Means clustering algorithm is an unsupervised algorithm and it is used to segment the interest area from the background. It clusters, or partitions the given data into K-clusters or parts based on the K-centroids.

The algorithm is used when you have unlabeled data(i.e. data without defined categories or groups). The goal is to find certain groups based on some kind of similarity in the data with the number of groups represented by K.

```
img = cv2.imread("/content/drive/MyDrive/Aadhar/img4")
img=cv2.cvtColor(img ,cv2.COLOR_BGR2RGB)
plt.figure(figsize=(13,10))
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x780b8cb24050>
```


```
img.shape #the first is height, the second is width and the third is the color channel of the image
```

```
(438, 699, 3)
```

```
#Next, converts the HxWx3 image into a Kx3 matrix where K=HxW and each row is now a vector in the 3-D space of RGB.
vectorized_img = img.reshape((-1,3))
vectorized_img.shape
```

```
(306162, 3)
```

```
#We convert the unit8 values to float as it is a requirement of the k-means method of OpenCV.
vectorized_img= np.float32(vectorized_img)
vectorized_img
```

```
array([[ 29.,  92., 136.],
       [ 29.,  92., 136.],
       [ 29.,  92., 136.],
       ...,
       [ 93.,  93.,  55.],
       [ 94.,  95.,  53.],
       [ 94.,  95.,  51.]], dtype=float32)
```

We are going to cluster with k = 3 because if you look at the image above it has 3 colors, green-colored grass and forest, blue sea and the greenish-blue seashore.

OpenCV provides cv2.kmeans(samples, nclusters(K), criteria, attempts, flags) function for color clustering.

1. samples: It should be of np.float32 data type, and each feature should be put in a single column.
2. nclusters(K): Number of clusters required at the end
3. criteria: It is the iteration termination criteria. When this criterion is satisfied, the algorithm iteration stops. Actually, it should be a tuple of 3 parameters. They are `( type, max_iter, epsilon )`:

Type of termination criteria. It has 3 flags as below:

cv.TERM_CRITERIA_EPS — stop the algorithm iteration if specified accuracy, epsilon, is reached.
cv.TERM_CRITERIA_MAX_ITER — stop the algorithm after the specified number of iterations, max_iter.
cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER — stop the iteration when any of the above condition is met.

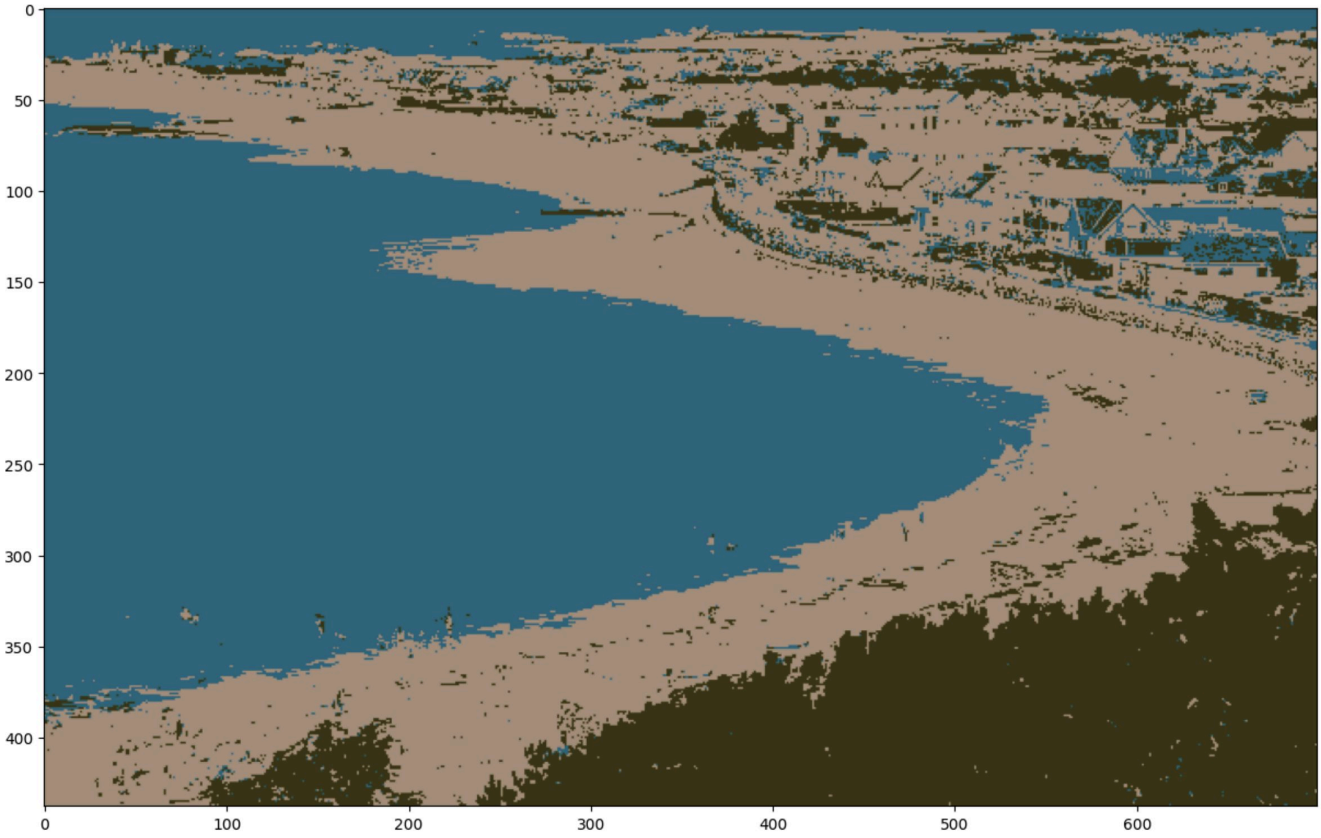
4. attempts: Flag to specify the number of times the algorithm is executed using different initial labelings. The algorithm returns the labels that yield the best compactness. This compactness is returned as output.
5. flags: This flag is used to specify how initial centers are taken. Normally two flags are used for this: cv.KMEANS_PP_CENTERS and cv.KMEANS_RANDOM_CENTERS.

```
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)


K = 3
attempts=10
ret,label,center=cv2.kmeans(vectorized_img,K,None,criteria,attempts,cv2.KMEANS_PP_CENTERS)


center = np.uint8(center)
center
```

```
array([[163, 143, 123],
       [ 51, 104, 122],
       [ 61,  56,  25]], dtype=uint8)
```

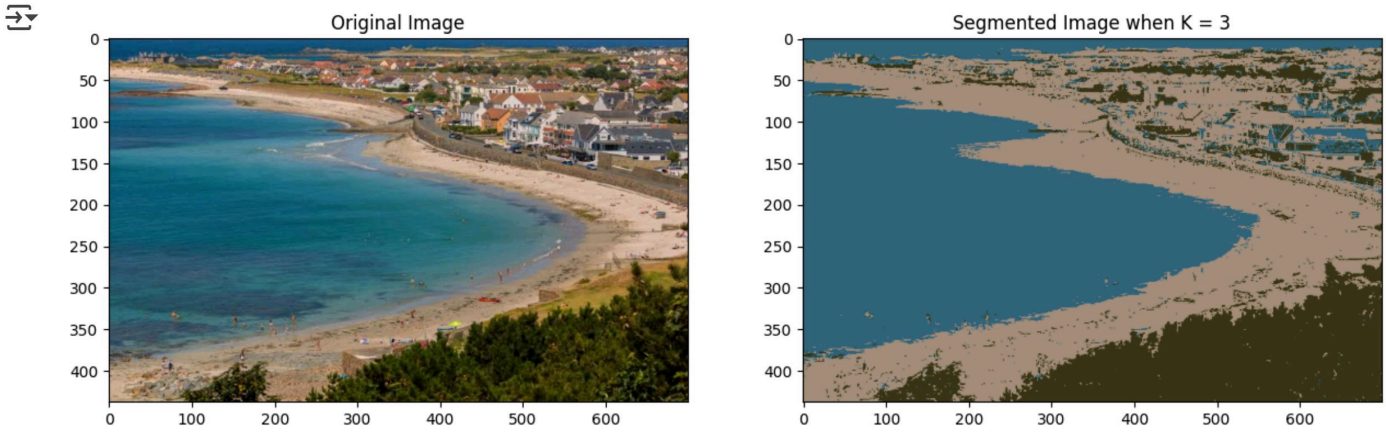
```
#Next, we have to access the labels to regenerate the clustered image
res = center[label.flatten()]
result_image = res.reshape((img.shape))


plt.figure(figsize=(15,10))
plt.imshow(result_image)
```

```
<matplotlib.image.AxesImage at 0x780b8cb084d0>
```




```
plt.figure(figsize=(15,12))
plt.subplot(1,2,1)
plt.imshow(img)
plt.title('Original Image')
```

```
plt.subplot(1,2,2)
plt.imshow(result_image)
plt.title('Segmented Image when K = %i' % K)
plt.show()
```



Let's see what happens when we change the value of K=5:

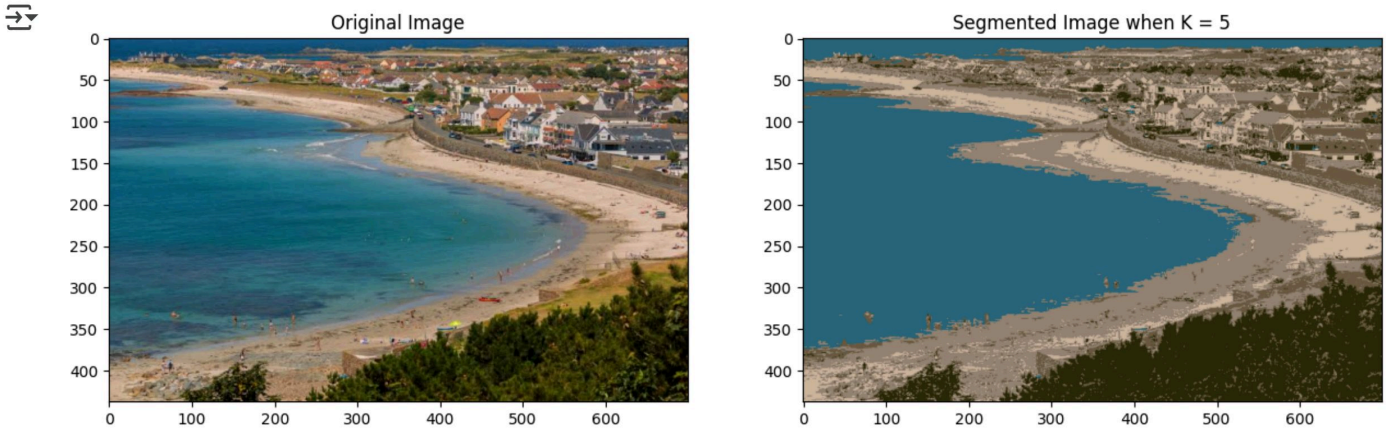
```
K = 5
attempts=10
ret,label,center=cv2.kmeans(vectorized_img,K,None,criteria,attempts,cv2.KMEANS_PP_CENTERS)
```

```
center = np.uint8(center)
center
```

```
array([[ 47,  44,  14],
       [ 44, 104, 124],
       [147, 133, 117],
       [204, 180, 159],
       [108,  94,  70]], dtype=uint8)
```

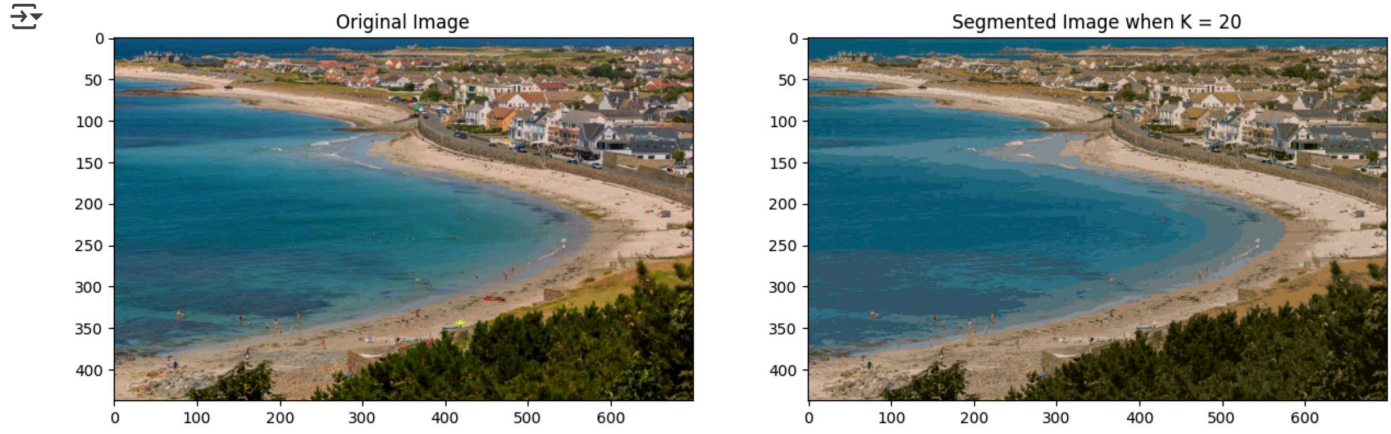
```
res = center[label.flatten()]
result_image = res.reshape((img.shape))
```

```
plt.figure(figsize=(15,12))
plt.subplot(1,2,1)
plt.imshow(img)
plt.title('Original Image')
plt.subplot(1,2,2)
plt.imshow(result_image)
plt.title('Segmented Image when K = %i' % K)
plt.show()
```



As you can see with an increase in the value of K, the image becomes clearer because the K-means algorithm can classify more classes/cluster of colors.

```
K = 20
attempts=10
ret,label,center=cv2.kmeans(vectorized_img,K,None,criteria,attempts,cv2.KMEANS_PP_CENTERS)
center = np.uint8(center)
res = center[label.flatten()]
result_image = res.reshape((img.shape))
plt.figure(figsize=(15,12))
plt.subplot(1,2,1)
plt.imshow(img)
plt.title('Original Image')
plt.subplot(1,2,2)
plt.imshow(result_image)
plt.title('Segmented Image when K = %i' % K)
plt.show()
```



Computer Vision

Name = Sanjay Chaurasia

Registration No: = 13119051622

Lab : Histogram of Oriented Gradients.

Objective:

The objective is to understand the underlying principles of HOG and how it encodes the structural information of an image by analyzing the distribution of gradient orientations within local regions.

Histogram of Oriented Gradients:

Histogram of Oriented Gradients (HoG) is a global feature representation, in the sense that one feature description is calculated for the entire image or an image-patch. The descriptor is a vector which contains many histograms. Each histogram belongs to a local area within the image and counts the frequency of gradient-directions in this local area. In images gradients in x- and y-direction, can easily be calculated by filters like Prewitt, Sobel, or the first-order derivative of a Gaussian. The magnitude of a gradient is large at edges and the orientation of the gradient-vector indicates the orientation of edges. Hence HoG-features encode the structure of objects in an image and can be applied for all detection and recognition tasks, for which structure-information is assumed to be crucial. HoG is particularly well suited for human detection and tracking. For this task it has initially be investigated in Oriented Gradients for Human Detection. Dalal and Triggs; Histograms of In this notebook the calculation of a HoG-descriptor of an image-patch is demonstrated step-by-step. In follow-up notebooks the application of HoG-descriptors for the detection of pedestrians in images and for tracking of pedestrians in videos are demonstrated

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force
```

1 STEP BY STEP - HOG Implementation

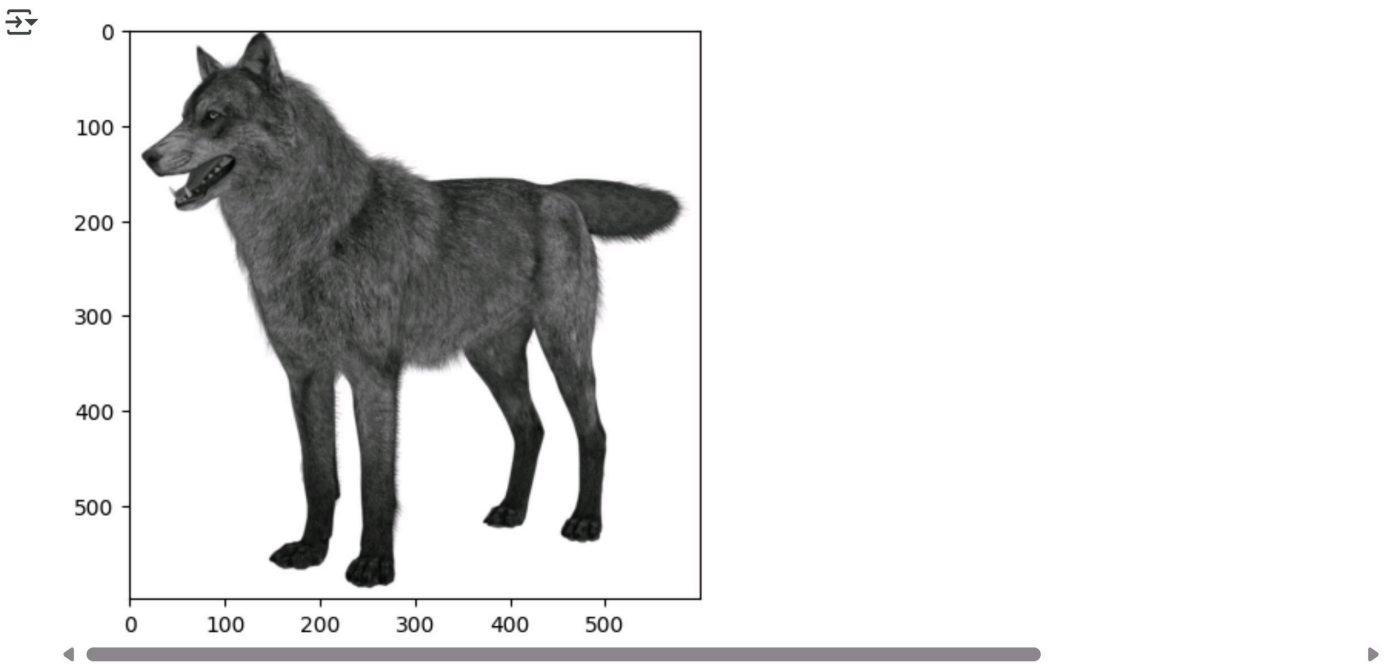
1.1 Import Libraries

```
#importing required libraries
import numpy as np
import skimage.color
import skimage.io
from skimage.feature import hog
import matplotlib.pyplot as plt
import math
import cv2
```

1.2 Import Image

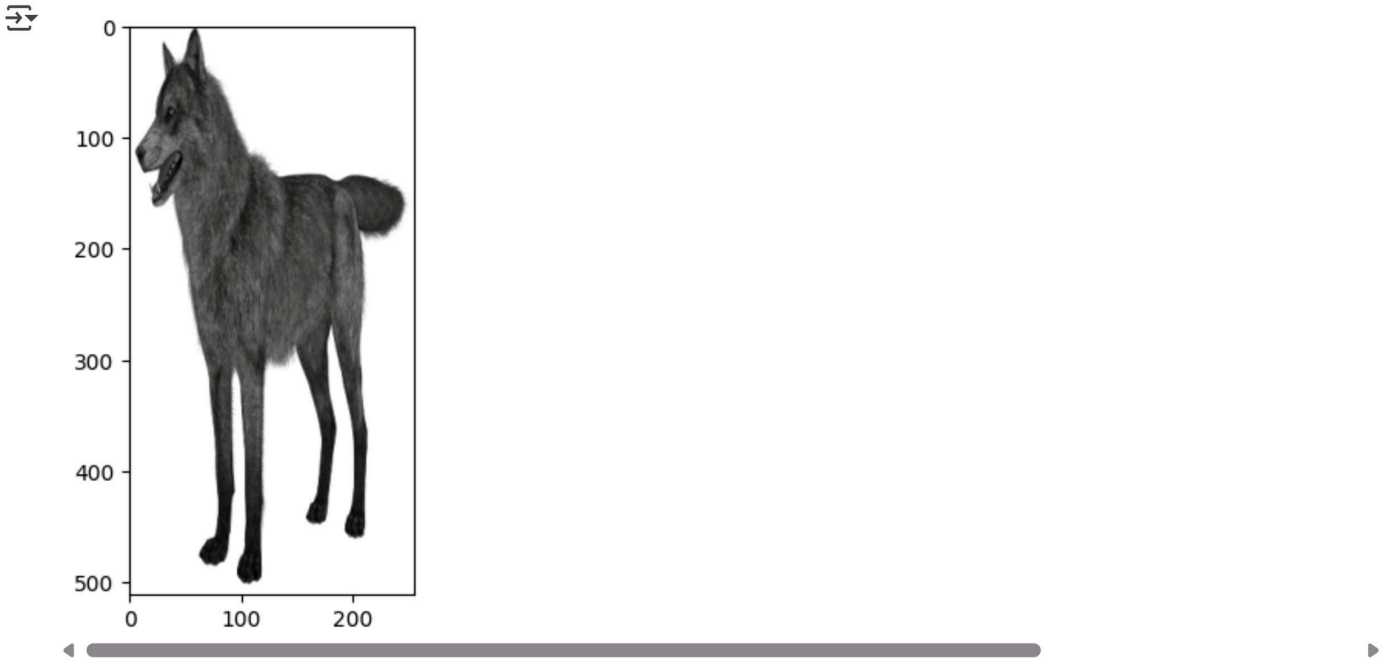
```
# load the image
grayscale_image = skimage.io.imread(fname = '/content/drive/MyDrive/Aadhar/img10.png', as_gray = True)

# display the image
```

```
x = plt.imshow(grayscale_image, cmap = "gray")
plt.show()
```



Resizing image

```
resized_img = cv2.resize(grayscale_image, (4*64,4*128))
plt.imshow(resized_img, cmap = "gray")
plt.show()
```



1.3 Creating and visualizing HOG features

```
fed, hog_img = hog(resized_img, orientations=9, pixels_per_cell=(8, 8),
                   cells_per_block=(2, 2), visualize=True)
plt.axis("off")
plt.imshow(hog_img, cmap="gray")
plt.show()
print(hog_img.shape)
print('pixel intensity',hog_img)
```

```
(512, 256)
pixel intensity [[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

1.4 Calculating gradient and angle of the image

```
Gx = np.array([[-1.0, 0.0, 1.0]])
Gy = np.array([[-1.0], [0.0], [1.0]])
[rows, columns] = np.shape(resized_img)
hog_mag_image = np.zeros(shape=(rows, columns))
hog_angle_image = np.zeros(shape=(rows, columns))

gx = cv2.filter2D(resized_img, -1, Gx)
gy = cv2.filter2D(resized_img, -1, Gy)

g_mag = np.sqrt(np.square(gx) + np.square(gy))
hog_image_angle = np.arctan(gy/gx)

print('Magnitude of Gradient')
print(g_mag)

print('\n Direction of Gradient')
print(hog_image_angle)
```

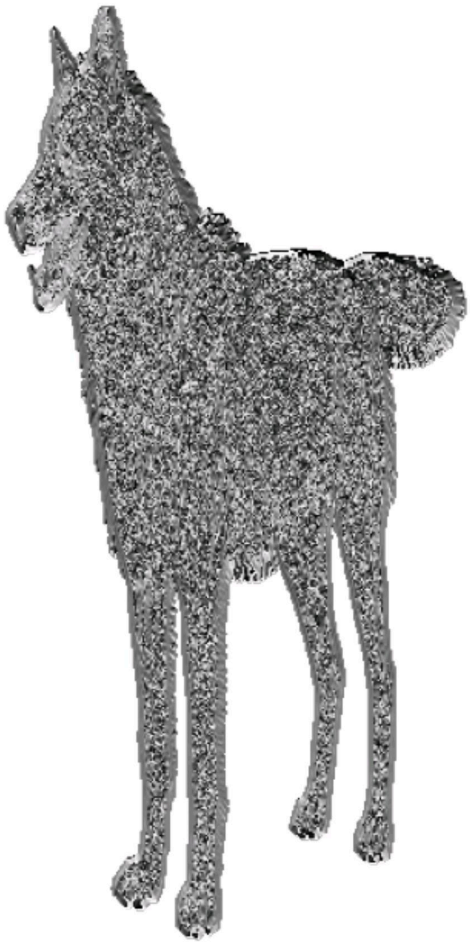
```
Magnitude of Gradient
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]

 Direction of Gradient
[[nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]
 ...
 [nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]
 [nan nan nan ... nan nan nan]]
<ipython-input-37-0f9ca16853e8>:11: RuntimeWarning: divide by zero encountered in divide
  hog_image_angle = np.arctan(gy/gx)
<ipython-input-37-0f9ca16853e8>:11: RuntimeWarning: invalid value encountered in divide
  hog_image_angle = np.arctan(gy/gx)
```

Visualizing Gradient Magnitude

```
plt.figure(figsize=(15, 8))
plt.imshow(g_mag, cmap="gray")
plt.axis("off")
plt.show()
```

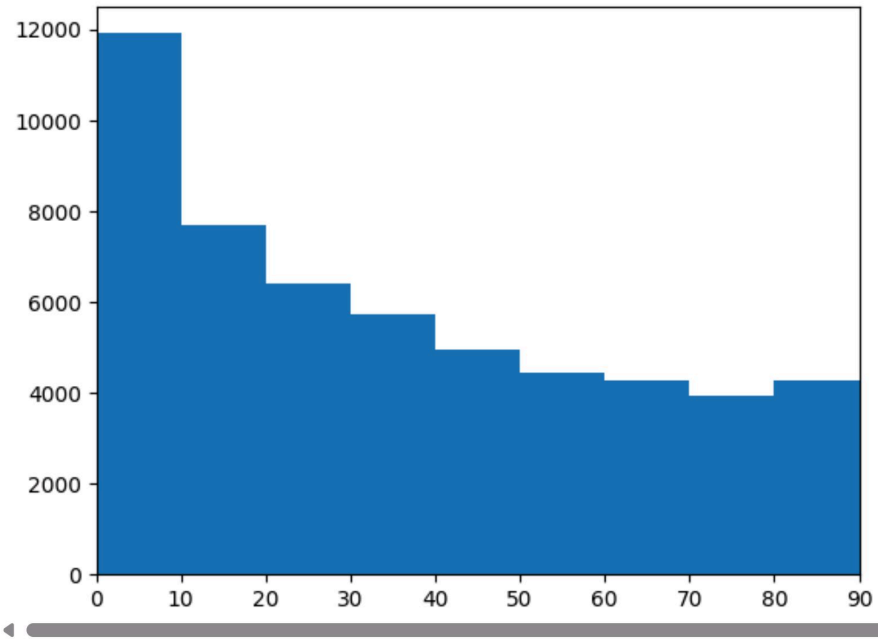
Visualizing Gradient Angle

```
plt.figure(figsize=(15, 8))
plt.imshow(hog_image_angle, cmap="gray")
plt.axis("off")
plt.show()
```



1.5 Visualizing Histogram

```
hog_angle_abs = np.abs(hog_image_angle)
hog_deg_image = np.degrees(hog_angle_abs)
x = hog_deg_image.flatten()

plt.hist(x, bins=9)
plt.xlim([0,90])
plt.show()
```



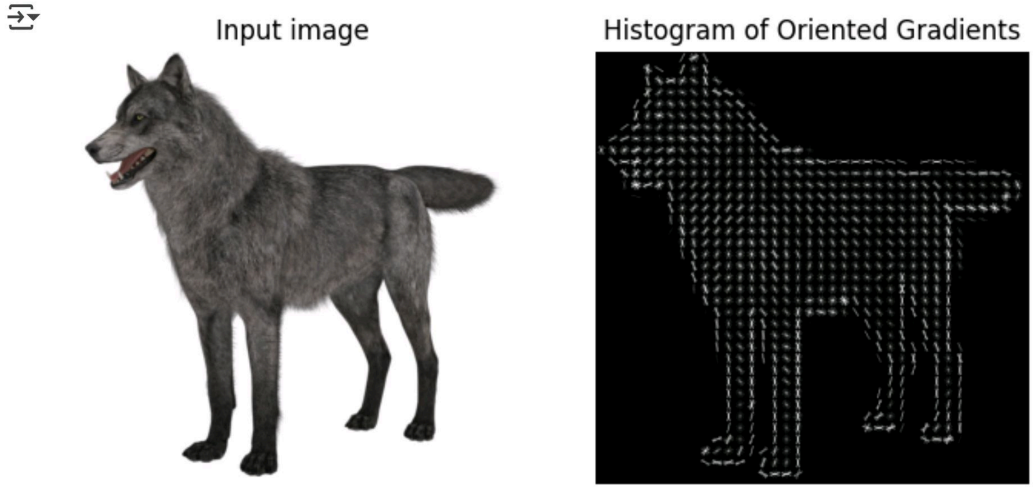
2 Summary/Algorithm Overview

The Histogram of Oriented Gradient (HOG) feature descriptor is popular for object detection.

Theoretically, we compute a HOG by:

- (optional) global image normalisation
- computing the gradient image in x and y
- computing gradient histograms
- normalising across blocks
- flattening into a feature vector

```
from skimage.io import imread
from skimage.feature import hog
from skimage import exposure
import matplotlib.pyplot as plt

image = imread('/content/drive/MyDrive/Aadhar/img10.png')

fd, hog_image = hog(
    image,
    orientations=8,
    pixels_per_cell=(16, 16),
    cells_per_block=(1, 1),
    visualize=True,
    channel_axis=-1
)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4), sharex=True, sharey=True)

ax1.axis('off')
ax1.imshow(image)
ax1.set_title('Input image')

hog_image_rescaled = exposure.rescale_intensity(hog_image, in_range=(0, 10))

ax2.axis('off')
ax2.imshow(hog_image_rescaled, cmap=plt.cm.gray)
ax2.set_title('Histogram of Oriented Gradients')

plt.show()
```



Start coding or generate with AI.

Start coding or generate with AI.

Computer Vision

Name = Sanjay Chaurasia

Registration No: = 13119051622

Lab 6: Camera Calibration

Camera calibration

Camera calibration is the process of estimating the internal characteristics (intrinsic parameters) and distortions of a camera. It is essential for computer vision tasks where accurate measurements or undistorted images are required, such as 3D reconstruction, robot navigation, and augmented reality.

The main goal of camera calibration is to determine:

Intrinsic parameters – These describe the camera's internal characteristics such as:

Focal length,

Optical center (principal point),

Skew (usually zero for modern cameras).

Extrinsic parameters – These define the camera's position and orientation in the world coordinate system.

Lens distortion coefficients – Real-world lenses cause distortion, especially radial and tangential distortions. Calibration helps correct these effects.

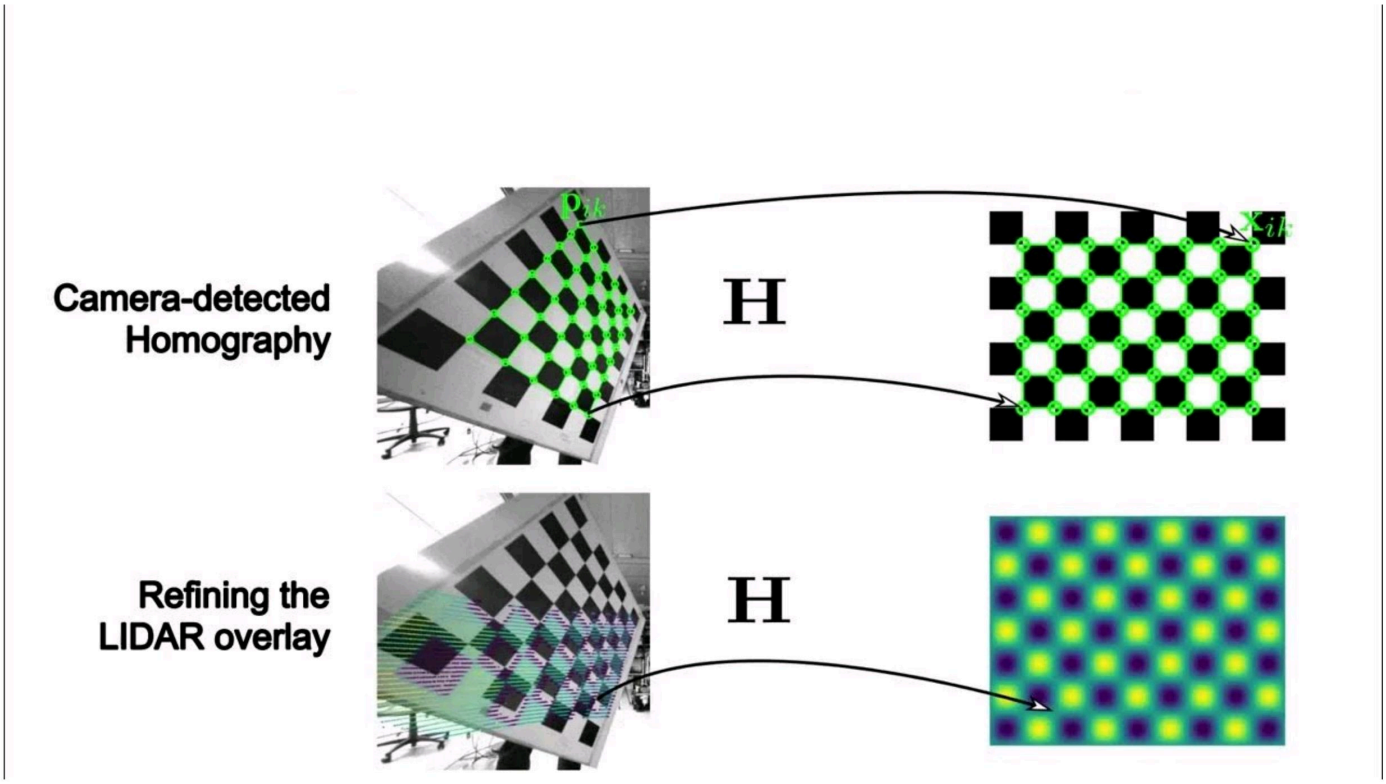
🧪 Calibration Process Steps

Capture multiple images of a known pattern (commonly a chessboard).

Detect corner points in the image (using cv2.findChessboardCorners()).

Map image points to real-world 3D coordinates.

Use these correspondences to compute calibration parameters using cv2.calibrateCamera().



```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
!pip install opencv-python
```

```
Requirement already satisfied: opencv-python in /usr/local/lib/python3.11/dist-packages (4.11.0.86)
Requirement already satisfied: numpy>=1.21.2 in /usr/local/lib/python3.11/dist-packages (from opencv-python) (2.0
```

```
import numpy as np
import cv2
from google.colab.patches import cv2_imshow

# Set correct pattern size for internal corners
pattern_size = (7, 7)

# Prepare object points
objp = np.zeros((pattern_size[0] * pattern_size[1], 3), np.float32)
objp[:, :2] = np.mgrid[0:pattern_size[0], 0:pattern_size[1]].T.reshape(-1, 2)

# Read the image
img = cv2.imread('/content/drive/MyDrive/Aadhar/img13.png')

if img is None:
    print("Failed to load image.")
else:
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Find the chessboard corners
    ret, corners = cv2.findChessboardCorners(gray, pattern_size, None)

    if ret:
        cv2.drawChessboardCorners(img, pattern_size, corners, ret)
        cv2_imshow(img)
        print("Chessboard corners detected.")
    else:
        print("Chessboard corners NOT found.")
```