ayush1

May 25, 2025

1 Objectives:

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

1. Reading an image in python
2. Convert Images to another format
3. Convert an Image to Grayscale
4. Perform Image enhancement operations

2 Reading and displaying an image in python

[1]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

image = mpimg.imread('H:/My Drive/U.S.A.R/6th Semester/Lab/sampleimg.tif')

plt.imshow(image, cmap='gray')
```

[1]:
```
<matplotlib.image.AxesImage at 0x24430a94980>
```

3 Convert Images to another format and compare them

[2]:
```
from PIL import Image
import os

img = Image.open('H:/My Drive/U.S.A.R/6th Semester/Lab/f14.tif')
img.save('f14_img_1.tif')
img.save('f14_img_1.jpg')
img.save('f14_img_1.png')
img.save('f14_img_1.jpeg')
img.save('f14_img_1.gif')

print("size of file in tiff",round(os.path.getsize('f14_img_1.tif')/
 ↪1024,2),"KB")
print("size of file in jpg",round(os.path.getsize('f14_img_1.jpg')/1024,2),"KB")
print("size of file in jpeg",round(os.path.getsize('f14_img_1.jpeg')/
 ↪1024,2),"KB")
print("size of file in png",round(os.path.getsize('f14_img_1.png')/1024,2),"KB")
print("size of file in gif",round(os.path.getsize('f14_img_1.gif')/1024,2),"KB")
```

```
size of file in tiff 300.17 KB
```

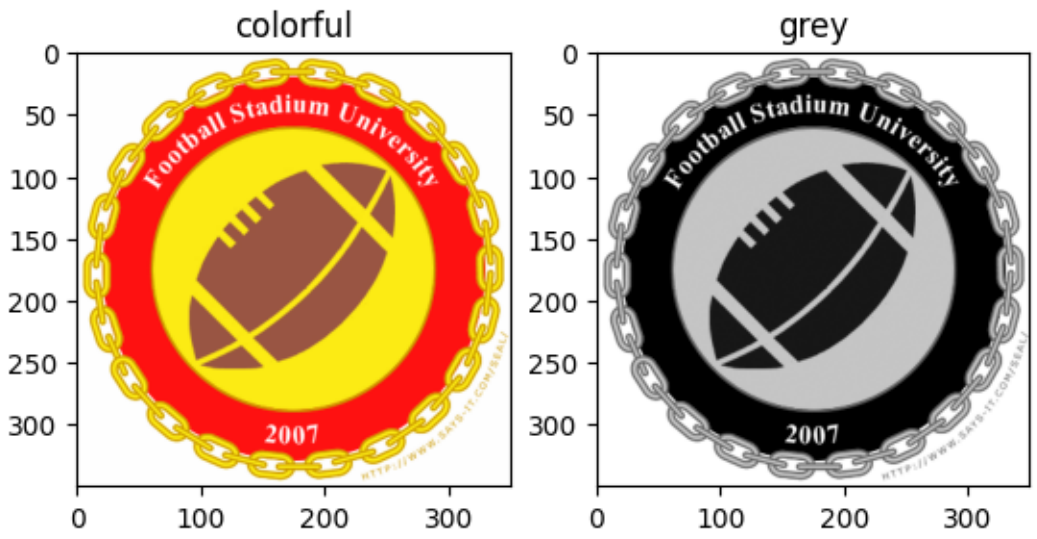
```
size of file in jpg 22.1 KB
size of file in jpeg 22.1 KB
size of file in png 144.35 KB
size of file in gif 212.61 KB
```

4 Convert an Image to Grayscale and display

[3]:
```
from PIL import Image
img = Image.open('H:/My Drive/U.S.A.R/6th Semester/Lab/football_seal.tif').
 ↪convert('L')
img.save('footbal_seal_update.tif')
```

[4]:
```
plt.subplot(1,2,1)
plt.imshow(mpimg.imread('H:/My Drive/U.S.A.R/6th Semester/Lab/football_seal.
 ↪tif'))
plt.title("colorful")
plt.subplot(1,2,2)
plt.imshow(img, cmap='gray')
plt.title("grey")
```

[4]:
```
Text(0.5, 1.0, 'grey')
```

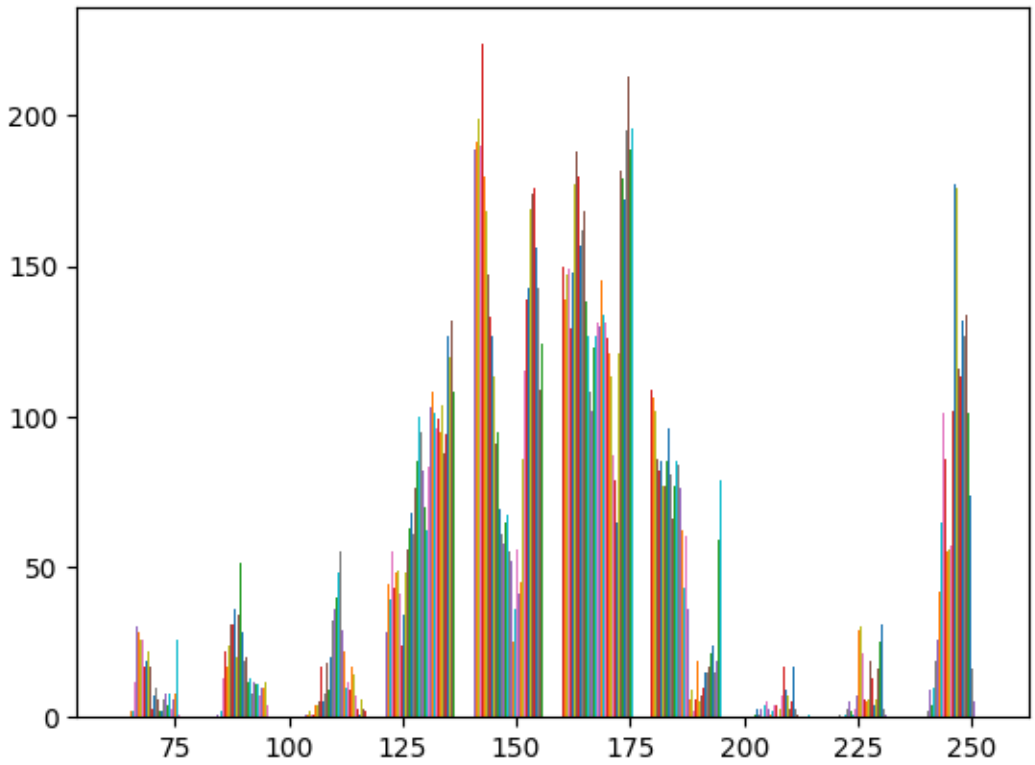


5 Perform Image enhancement operations

Here students will explain need and type of image enhancement methods

[5]:
```
plt.hist(image)
```

[5]:
```
(array([[0., 0., 0., …, 0., 0., 0.],
        [0., 0., 0., …, 0., 0., 0.],
        [0., 0., 0., …, 0., 0., 0.],
        …,
        [0., 0., 0., …, 3., 0., 0.],
        [0., 0., 0., …, 2., 0., 0.],
        [0., 0., 0., …, 1., 0., 0.]], shape=(640, 10)),
 array([ 61. ,  80.4,  99.8, 119.2, 138.6, 158. , 177.4, 196.8, 216.2,
        235.6, 255. ]),
 <a list of 640 BarContainer objects>)
```



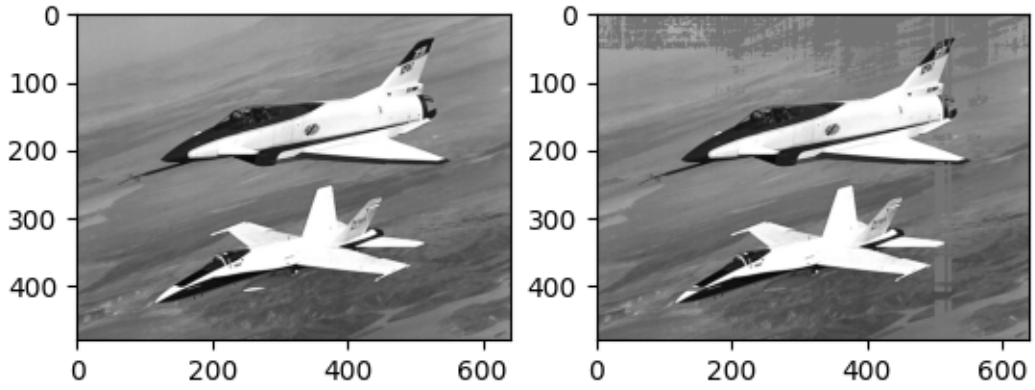
[6]:
```
plt.imshow(image, cmap='grey')
```

[6]:
```
<matplotlib.image.AxesImage at 0x24437050910>
```

[7]:
```
import numpy as np
x_new=np.copy(image)
for i in range(image.shape[0]):
    for j in range(image.shape[1]):
        if image[50,j]<=image[i,j]<=image[i,150]:
            x_new[i,j]=150
        else:
            pass
```

[8]:
```
plt.subplot(1,2,1)
plt.imshow(image,cmap='grey')

plt.subplot(1,2,2)
plt.imshow(x_new,cmap='grey')
```

[8]:
```
<matplotlib.image.AxesImage at 0x24436e91a90>
```



[ ]:
```
plt.subplot(1,2,1)
plt.hist(image)
plt.subplot(1,2,2)
plt.hist(x_new)
```

6 Perform Image enhancement operations

Image enhancement is a crucial step in image processing that aims to improve the visual appearance of an image or to convert the image to a form better suited for analysis by a human or machine. The need for image enhancement arises due to various factors such as poor lighting conditions, low contrast, noise, and other degradations that can affect the quality of an image.

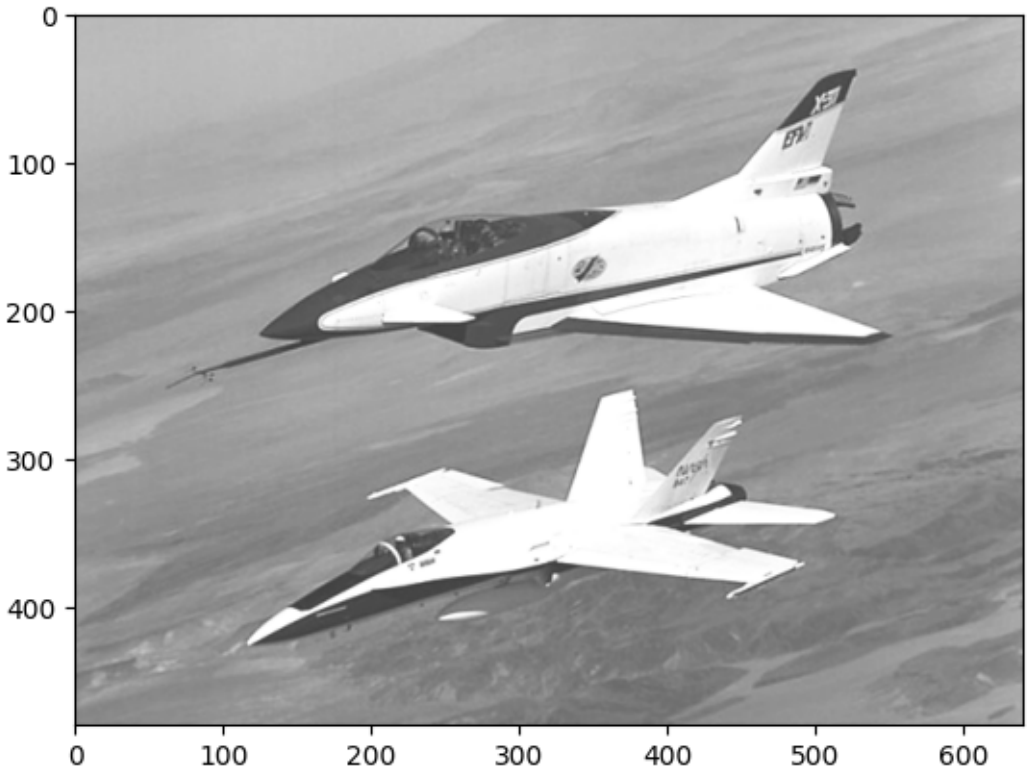
Types of image enhancement methods include:

1. **Histogram Equalization**: This method improves the contrast of an image by effectively spreading out the most frequent intensity values. It is useful for images with backgrounds and foregrounds that are both bright or both dark.
2. **Contrast Stretching**: This technique enhances the contrast by stretching the range of intensity values so that it spans the full intensity range. It is useful for images with low contrast.
3. **Smoothing**: This method reduces noise and fine details in an image. It is often used in preprocessing steps to remove high-frequency noise.
4. **Sharpening**: This technique enhances the edges and fine details in an image. It is useful for highlighting important features.
5. **Filtering**: Various filters (e.g., Gaussian, median) can be applied to an image to achieve different enhancement effects, such as noise reduction or edge enhancement.

By applying these methods, the quality of images can be significantly improved, making them more suitable for further analysis and interpretation.

[34]:
```
import cv2
import matplotlib.pyplot as plt
import numpy as np

# Load the image using cv2.imread()
testimg = cv2.imread('H:/My Drive/U.S.A.R/6th Semester/Lab/sampleimg.tif')
plt.imshow(testimg)
```

[34]:
```
<matplotlib.image.AxesImage at 0x24440586e90>
```



[35]:
```
testimg.shape
```

[35]:
```
(480, 640, 3)
```

[36]:
```
minval=256
maxval=-1
for i in range(480):
    for j in range(640):
        # Convert RGB to grayscale by taking the mean of the RGB values
        grayscale_value = int(np.mean(testimg[i][j]))
        maxval = max(maxval, grayscale_value)
        minval = min(minval, grayscale_value)
```

```
print(minval)
print(maxval)
```

```
61
255
```

[37]:
```
arr = [0]*256
```

[38]:
```
for i in range(testimg.shape[0]):
    for j in range(testimg.shape[1]):
        # Get the average intensity of the RGB channels for grayscale␣
 ↪representation
        intensity = int(np.mean(testimg[i][j]))
        arr[intensity] += 1
```

[ ]:
```
print(arr)
```

[ ]:
```
summ=0
for i in range(len(arr)):
    summ+=arr[i]
print(summ)
```

[41]:
```
probarr=[arr[i]/summ for i in range(len(arr))]
```

[ ]:
```
print(probarr)
```

[ ]:
```
summ=0
for i in range(len(probarr)):
    summ+=probarr[i]
print(summ)
```

[44]:
```
print(len(probarr))
```

```
256
```

[ ]:
```
cdf=[0]*256
for i in range(len(probarr)):
    j=0
    val=probarr[i]
    while j<i:
        val+=probarr[j]
        j+=1
    cdf[i]=val
print(cdf)
```

[ ]:
```
#Previous cells where you calculate 'probarr' should be executed before this␣
 ↪cell.
```

```
#...

cdf=[0]*256
for i in range(len(probarr)):
    j=0
    val=probarr[i]
    while j<i:
        val+=probarr[j]
        j+=1
    cdf[i]=val
print(cdf)

#Now, cdf is defined and you can use it in the next line
print(len(cdf))
```

[47]:
```
cdf=[cdf[i]*maxval for i in range(len(cdf))]
```

[ ]:
```
print(cdf)
```

[49]:
```
cdf = [round(cdf[i]) for i in range(len(cdf))]
```

[ ]:
```
print(cdf)
```

[51]:
```
cdf=[round(cdf[i]) for i in range(len(cdf))]
```

[ ]:
```
print(cdf)
```

[53]:
```
import numpy as np
```

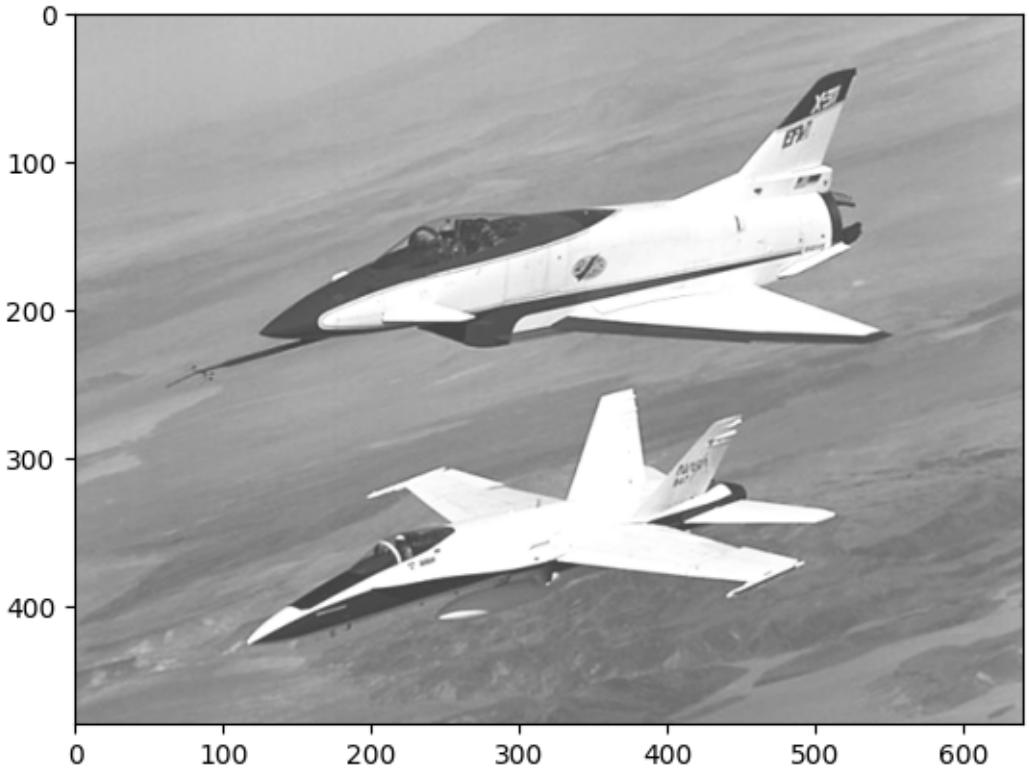
[54]:
```
enhanced_img=np.array(testimg)
enhanced_img.shape
```

[54]:
```
(480, 640, 3)
```

[55]:
```
for i in range(testimg.shape[0]):
    for j in range(testimg.shape[1]):
        grayscale_value = int(np.mean(testimg[i][j]))
        enhanced_img[i][j] = cdf[grayscale_value]
```

[56]:
```
plt.imshow(testimg,cmap='gray')
```

[56]:
```
<matplotlib.image.AxesImage at 0x24440604910>
```



[57]:
```
plt.imshow(enhanced_img,cmap='gray')
```

[57]:
```
<matplotlib.image.AxesImage at 0x24440672710>
```

7 Conclusion

In this notebook, we explored various fundamental image processing operations using Python and popular libraries such as OpenCV, PIL, and Matplotlib. We started by reading and displaying images using `matplotlib.pyplot` and `matplotlib.image`. We then converted images to different formats and compared their file sizes to understand the impact of different formats on storage.

Next, we converted an image to grayscale and displayed both the original and grayscale images using subplots for comparison. We performed image enhancement operations, including histogram equalization, to improve the visual quality of the images. We calculated the cumulative distribution function (CDF) and applied it to enhance the image contrast.

Throughout this notebook, we learned how to manipulate and enhance images, understand the importance of different image formats, and apply various image processing techniques to improve image quality. These skills are essential for any computer vision task and provide a solid foundation for more advanced image processing and analysis.

By implementing these operations, we gained hands-on experience with image processing, which is crucial for practical applications in computer vision and related fields.

ayush2

May 25, 2025

Computer Vision

Name: Ayush Kumar

Enrollment No: 10219051622

Lab - 2 : Image enhancement using spatial filtering

1 OBJECTIVE

To understand the Image enhancement using spatial filtering

[2]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

[3]:
```
img = cv2.imread(r'H:\My Drive\U.S.A.R\6th Semester\Lab\cameraman.tif', cv2.
↪IMREAD_GRAYSCALE)  # Read image in grayscale
plt.figure(figsize=(6,6))
plt.imshow(img, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.show()
```

2 ADDING NOISE TO AN IMAGE

What is Noise in an Image?

Noise in an image refers to random variations in brightness or color that are not part of the original scene. It appears as unwanted specks, graininess, or irregular pixels, and often reduces the overall quality and clarity of the image.

Why Does Noise Occur?

Noise can be introduced due to several reasons: - Low lighting conditions, which force the camera to amplify signals. - Electronic interference from sensors or hardware. - Errors during image transmission or compression. - Dust or defects on the camera sensor.

Common Types of Image Noise

Gaussian Noise - Appears as smooth, grainy variations. Pixel values follow a normal distribution. **Uniform Noise** - Pixel values are randomly spread within a fixed range. Common in synthetic testing. **Impulse Noise** - Also known as *salt-and-pepper noise*. Shows up as sudden black or white

dots.

How the function 'noise' works?

1. Create a Blank Noise Mask. Initialized it with the same shape as the image.
2. Generate the Selected Noise Type:

- Gaussian Noise: Uses a normal distribution with mean = 128 and standard deviation = 20.
- Uniform Noise: Random values uniformly distributed between 0 and 255.
- Impulse Noise: Randomly sets some pixels to white (255), simulating salt-and-pepper noise.

3. Adjust Intensity: The noise is scaled down (multiplied by 0.5) to avoid overpowering the original image.
4. Add Noise to Image by adding the noise matrix to the original image pixel-wise.
5. Visualize the Process: A 3-part subplot is created to show: The original image, The generated noise, The final noisy image.

[28]:
```python
def noise(img, noise_type):
    noise = np.zeros(img.shape, dtype=np.uint8)

    if noise_type == 'gaussian':
        cv2.randn(noise, 128, 20)  # mean, stddev
        noise = (noise * 0.5).astype(np.uint8)

    elif noise_type == 'uniform':
        cv2.randu(noise, 0, 255)
        noise = (noise * 0.5).astype(np.uint8)

    elif noise_type == 'impulse':
        cv2.randu(noise, 0, 255)
        noise = cv2.threshold(noise, 245, 255, cv2.THRESH_BINARY)[1]

    else:
        print("Choose a valid noise")

    noisy_img = cv2.add(img, noise)

    fig = plt.figure(dpi=500)
    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(noise, cmap='gray')
    plt.axis("off")
    plt.title(f"{noise_type} Noise")
```

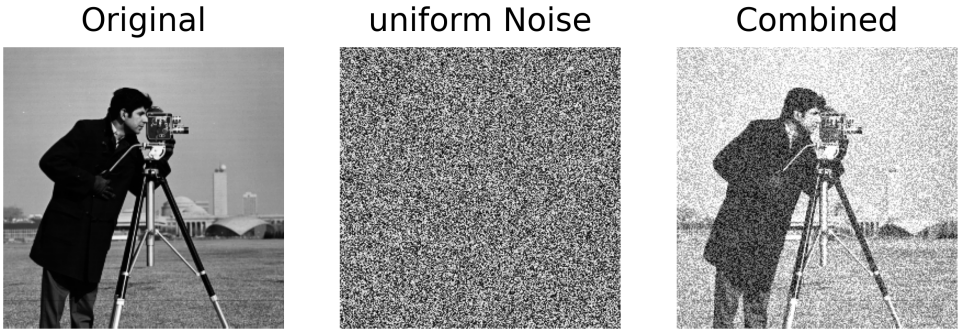
```
    fig.add_subplot(1, 3, 3)
    plt.imshow(noisy_img, cmap='gray')
    plt.axis("off")
    plt.title("Combined")

    plt.show()

    return noisy_img
```

GAUSSIAN NOISE

[29]:
```
gn_img = noise(img, 'gaussian')
```

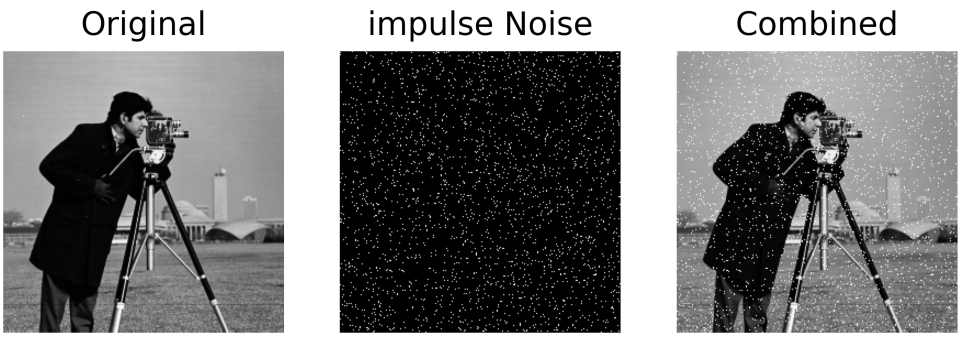
UNIFORM NOISE

[30]:
```
un_img = noise(img, 'uniform')
```



IMPULSE NOISE

[31]:
```
im_img = noise(img, 'impulse')
```



3 DENOISING AN IMAGE USING BUILT-IN FUNCTION (Non Local Means Denoising)

What is Image Denoising?

Image denoising is the process of removing unwanted noise from an image while preserving important features like edges, textures, and details.

When an image gets corrupted due to factors like poor lighting, sensor errors, or transmission issues, it may contain visual distortions called noise. These distortions degrade the image quality and can affect further analysis or interpretation.

Why Denoising is Important:

Enhances visual quality

Improves feature detection (edges, corners, textures)

Helps in medical imaging, satellite imaging, OCR, and more

Essential before tasks like image segmentation or classification.

What is Non Local Means Denoising?

Non-Local Means (NLM) Denoising is a smart technique used to remove noise from an image while keeping important details like edges and textures intact. Non-Local Means compares a pixel to other similar-looking patches all over the image

denoise function

It uses OpenCV's cv2.fastNlMeansDenoising() method, which works by:

- Comparing patches of the image
- Identifying similar regions
- Averaging them to remove noise while preserving details.

The function also displays a side-by-side comparison of The Original Image, The Noisy Image, The Denoised Output.

[32]:
```
def denoise(original_img, noisy_img, noise_type):

    denoised_img = cv2.fastNlMeansDenoising(noisy_img, None, 10, 10)

    fig = plt.figure(dpi=300)

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

    fig.add_subplot(1, 3, 2)
    plt.imshow(noisy_img, cmap='gray')
    plt.axis("off")
    plt.title(f"with {noise_type} Noise")

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

    plt.show()

    return denoised_img
```

REMOVING GAUSSIAN NOISE

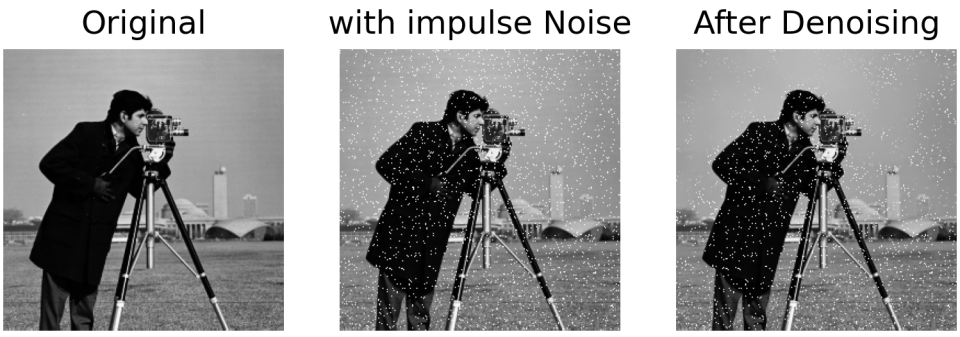
[33]:
```
denoised1 = denoise(img, gn_img, 'gaussian')
```

REMOVING UNIFORM NOISE

[34]:
```
denoised2 = denoise(img, un_img, 'uniform')
```

REMOVING IMPULSE NOISE

[35]:
```
denoised3 = denoise(img, im_img, 'impulse')
```



4 DENOISING BY APPLYING FILTER

What is Denoising by Applying a Median Filter?

A Median Filter is a simple and effective method used to remove noise from an image, especially salt-and-pepper noise (random black and white pixels).

How It Works:

- It looks at a small neighborhood (like 3×3 or 5×5) around each pixel.
- It sorts the pixel values in that neighborhood.
- It replaces the center pixel with the median value (the middle one in the sorted list)

What is Denoising by Applying a Gaussian Filter?

A Gaussian Filter is a type of image smoothing technique used to reduce noise and soften the image. It's based on the Gaussian function, which gives more weight to pixels near the center and less to those farther away.

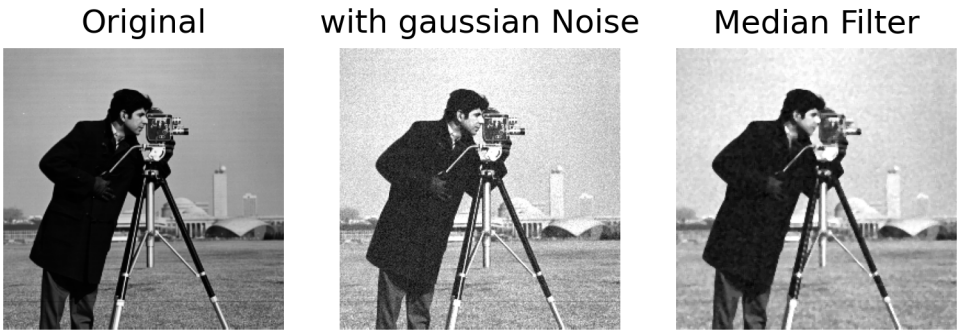
How It Works:

- It uses a bell-shaped kernel (like a blur mask) to scan the image.
- Each pixel is replaced with a weighted average of its neighbors.
- Pixels closer to the center have higher influence, while distant pixels have less.

[36]:
```python
def filter(original_img, noisy_img, noise_type, filter_type="median"):
    if filter_type == "median":
        filtered_img = cv2.medianBlur(noisy_img, 3)
        filter_title = "Median Filter"
    elif filter_type == "gaussian":
        filtered_img = cv2.GaussianBlur(noisy_img, (3, 3), 0)
        filter_title = "Gaussian Filter"
    else:
        print("Choose the proper filter type")

    fig = plt.figure(dpi=300)

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

    fig.add_subplot(1, 3, 2)
    plt.imshow(noisy_img, cmap='gray')
    plt.axis("off")
    plt.title(f"with {noise_type} Noise")

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

    plt.show()

    return filtered_img
```

MEDIAN FILTER ON GAUSSIAN NOISE

[37]:
```python
blurred1 = filter(img, gn_img, 'gaussian', 'median')
```

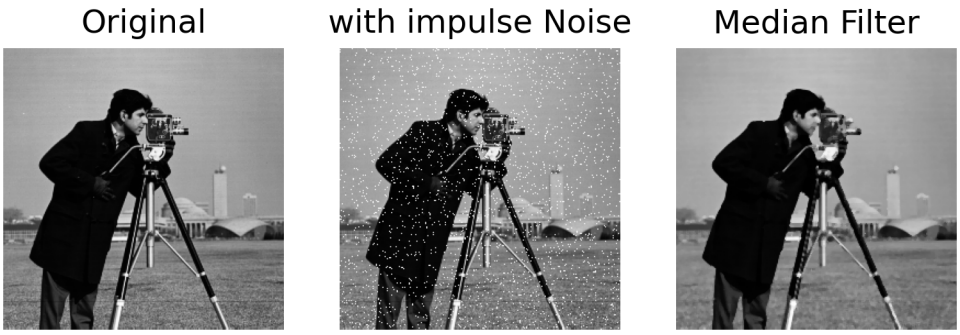


MEDIAN FILTER ON UNIFORM NOISE

[38]:
```
blurred2 = filter(img, un_img, 'uniform', 'median')
```

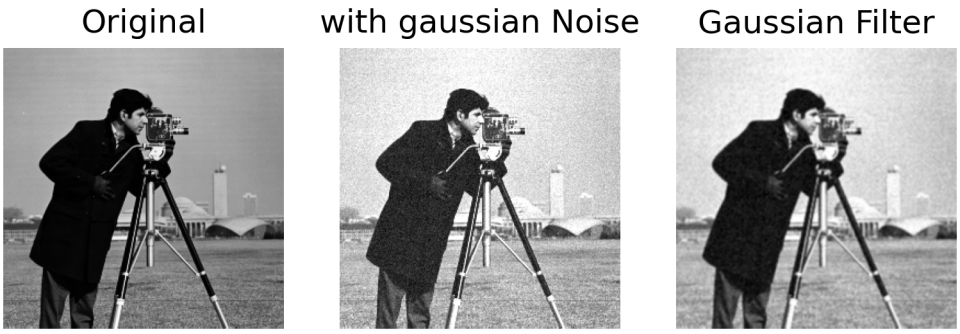
MEDIAN FILTER ON IMPULSE NOISE

[39]:
```
blurred3 = filter(img, im_img, 'impulse', 'median')
```



GAUSSIAN FILTER ON GAUSSIAN NOISE

[40]:
```
blurred21 = filter(img, gn_img, 'gaussian', 'gaussian')
```

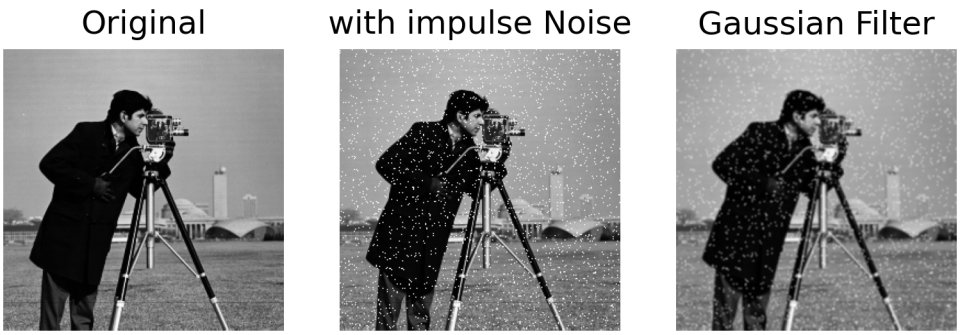


GAUSSIAN FILTER ON UNIFORM NOISE

[41]:
```
blurred22 = filter(img, un_img, 'uniform', 'gaussian')
```

GAUSSIAN FILTER ON IMPULSE NOISE

[42]:
```
blurred23 = filter(img, im_img, 'impulse', 'gaussian')
```



5 CONCLUSION -

This experiment demonstrates how different types of noise affect images and how NLM denoising and various filtering techniques can help reduce them. By comparing visual outputs, we understand the strengths and limitations of each denoising method in preserving image quality.

ayush3

May 25, 2025

Computer Vision

Name: Honey Singh

Registration No: 12419051622

Lab - 3 : Image Enhancement using Spatial Filtering

1 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)

2 Reading and displaying an image in python

[32]:
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
```

[33]:
```
img = cv2.imread(r'H:\My Drive\U.S.A.R\6th Semester\Lab\cameraman.tif', cv2.
↪IMREAD_GRAYSCALE)  # Read image in grayscale
plt.figure(figsize=(6,6))
plt.imshow(img, cmap='gray')
plt.title('Original Image')
plt.axis('off')
plt.show()
```

3 Averaging/smoothing mask on image

Averaging or smoothing is a spatial domain filtering technique used to reduce noise and smoothen an image. It involves applying a kernel (filter mask) to the image, where each pixel's value is replaced by the average of its neighboring pixel values within the kernel. This process helps in reducing sharp transitions in intensity, making the image appear smoother. However, it may also blur fine details and edges in the image. "'

[34]:
```
# Step 2: Averaging/Smoothing Mask
kernel_size = 3  # Define kernel size
kernel = np.ones((kernel_size, kernel_size)) / (kernel_size ** 2)
```

[35]:
```
# Apply filter manually
h, w = img.shape
smoothed_img = np.zeros((h, w), dtype=np.uint8)
pad = kernel_size // 2
```

```
padded_img = np.pad(img, pad, mode='constant', constant_values=0)  # Zero␣
↪padding
```

[36]:
```
for i in range(h):
    for j in range(w):
        region = padded_img[i:i+kernel_size, j:j+kernel_size]
        smoothed_img[i, j] = np.sum(region * kernel)  # Apply averaging mask
```

[37]:
```
plt.figure(figsize=(6,6))
plt.imshow(smoothed_img, cmap='gray')
plt.title('Averaged/Smoothened Image')
plt.axis('off')
plt.show()
```

4 Averaging Mask for Noise Removal

An averaging mask is a spatial filtering technique used to reduce noise in images. It works by replacing each pixel's value with the average of its neighboring pixel values within a defined kernel (e.g., 3x3, 5x5). This process smoothens the image by reducing sharp intensity variations caused by noise.

The kernel is a matrix where each element has equal weight (e.g., 1/9 for a 3x3 kernel). The mask slides over the image, and for each pixel, the weighted sum of the overlapping region is calculated and assigned to the center pixel. While effective in reducing random noise, this method can also blur fine details and edges in the image.

[38]:
```
# Step 3: Averaging Mask for Noise Removal (Same as smoothing but useful in␣
↪noisy images)
noise_removed_img = np.zeros((h, w), dtype=np.uint8)
```

[39]:
```
for i in range(h):
    for j in range(w):
        region = padded_img[i:i+kernel_size, j:j+kernel_size]
        noise_removed_img[i, j] = np.sum(region * kernel)  # Apply again
```

[40]:
```
plt.figure(figsize=(6,6))
plt.imshow(noise_removed_img, cmap='gray')
plt.title('Noise Removed Image')
plt.axis('off')
plt.show()
```

4.1 Median Filter for Salt and Pepper Noise

Median filtering is a non-linear spatial filtering technique used to reduce noise in images, particularly effective for removing salt-and-pepper noise. Salt-and-pepper noise appears as random occurrences of white and black pixels in an image.

The median filter works by replacing each pixel's value with the median value of the intensities in its neighborhood, defined by a kernel (e.g., 3x3, 5x5). The median is calculated by sorting the pixel values in the kernel and selecting the middle value. This process preserves edges better than averaging filters, as it does not introduce new intensity levels and is less sensitive to outliers.

The steps to apply a median filter are: 1. Define the kernel size (e.g., 3x3). 2. Slide the kernel over the image. 3. For each pixel, compute the median of the pixel values in the kernel and assign it to the center pixel. 4. Repeat for all pixels in the image.

[41]:
```
# Step 4: Median Filter for Salt and Pepper Noise
median_filtered_img = np.zeros((h, w), dtype=np.uint8)
```

[42]:
```
for i in range(h):
    for j in range(w):
        region = padded_img[i:i+kernel_size, j:j+kernel_size]
        median_filtered_img[i, j] = np.median(region)  # Apply median filter
```

[43]:
```
plt.figure(figsize=(6,6))
plt.imshow(median_filtered_img, cmap='gray')
plt.title('Median Filtered Image')
plt.axis('off')
plt.show()
```

5 Conclusion

In this lab, we explored various spatial domain filtering techniques for image enhancement and noise removal. The key takeaways are:

1. **Averaging/Smoothing Filter**: This technique effectively reduces noise and smoothens the

image by replacing each pixel's value with the average of its neighbors. However, it can blur fine details and edges.

2. **Averaging Mask for Noise Removal**: Similar to smoothing, this method is particularly useful for reducing random noise in images. It helps in achieving a cleaner image but may compromise edge sharpness.

3. **Median Filter**: A non-linear filtering technique that is highly effective in removing salt-and-pepper noise. Unlike averaging, it preserves edges and fine details by replacing each pixel's value with the median of its neighborhood.

These techniques demonstrate the importance of spatial domain filtering in image processing, allowing us to enhance image quality and reduce noise while balancing the trade-offs between smoothness and detail preservation.

ayush4

May 25, 2025

Computer Vision

Name: Honey Singh

Enrollment No: 12419051622

Lab - 4 : Image Segmentation using K Means

1 OBJECTIVE

To perform Image segmentation using K means Clustering

2 Reading and Displaying Image

Reading and displaying an image is the first step before applying any modifications or analysis .To read and display an image in python we first load an image into Python using OpenCV and display it using Matplotlib. We convert the image from OpenCV's default BGR color format to the standard RGB format so that colors display correctly in matplotlib or other libraries.

[2]:
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
```

[11]:
```
img = cv2.imread(r"H:\My Drive\U.S.A.R\6th Semester\Lab\OIP.jpeg")
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.figure(figsize=(6.5, 5))
plt.imshow(img)
img.shape
```

[11]:
```
(236, 379, 3)
```

3 Function for Image Segmentation

THEORITICAL CONCEPTS

What is Image Segmentation?

1. Image segmentation is the process of dividing an image into parts or regions.
2. Pixels in the same region look similar in colour, brightness, or texture.
3. Segmentation makes it easier to understand or analyze the image (e.g., find objects, backgrounds, etc).
4. Used For:
   - Object detection (finding people, cars, etc.)
   - Medical imaging (finding tumors in scans)
   - Background removal
   - Robot vision
   - Image compression

What is Vectorization?

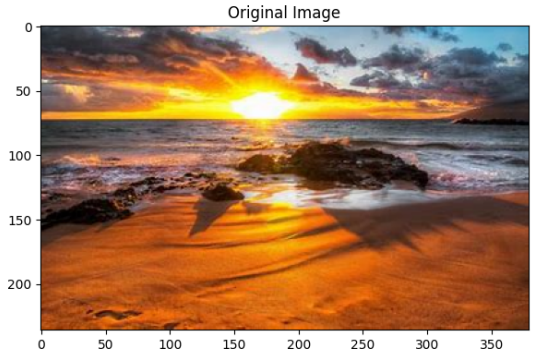
Vectorization refers to the flattening of pixels. We convert the image into a 2D array where each row is one pixel's RGB values. Shape becomes: (total_pixels, 3)

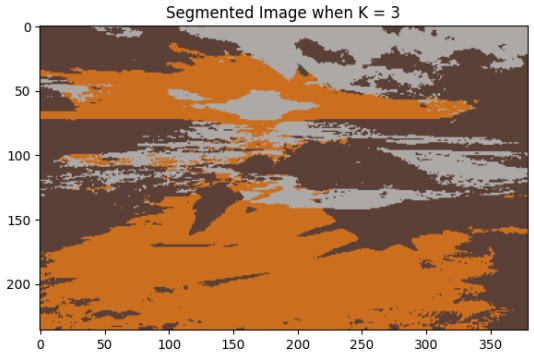
Image is a grid of pixels. Normally, an image is a 3D array: height × width × color channels (e.g 600×480×3).

We vectorize the image because algorithms like K-Means expect data as vectors (rows of features). So we reshape the image to feed it into the model.

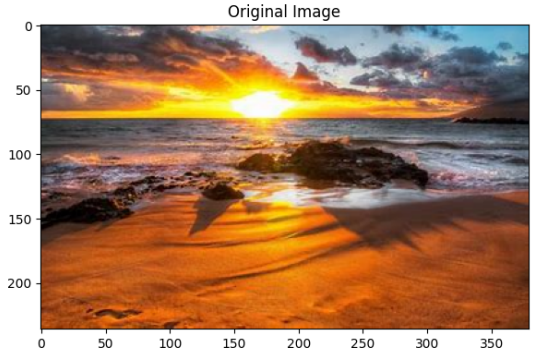
[12]:
```
def image_segmentation(img, k, attempts=10):

    # Reshape image to 2D array of pixels and 3 color values (RGB)
    vectorized_img = img.reshape((-1, 3))
    vectorized_img = np.float32(vectorized_img)

    # Define criteria = (type, max_iter, epsilon)
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)

    # Apply KMeans
    ret, label, center = cv2.kmeans(vectorized_img, k, None, criteria,␣
 ↪attempts, cv2.KMEANS_PP_CENTERS)

    # Convert back into uint8 and get result
    center = np.uint8(center)
    res = center[label.flatten()]
    result_image = res.reshape((img.shape))

    # Plot
    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(f'Segmented Image when K = {k}')
    plt.show()
```

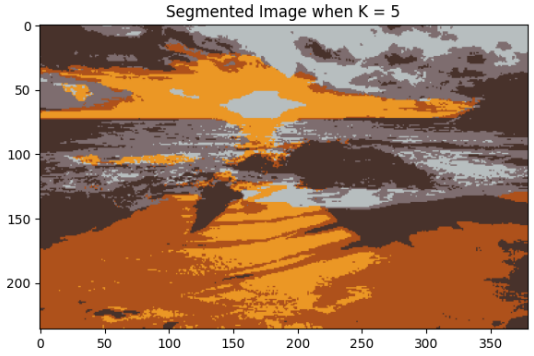
[13]:
```
# k=3
image_segmentation(img, 3)
```



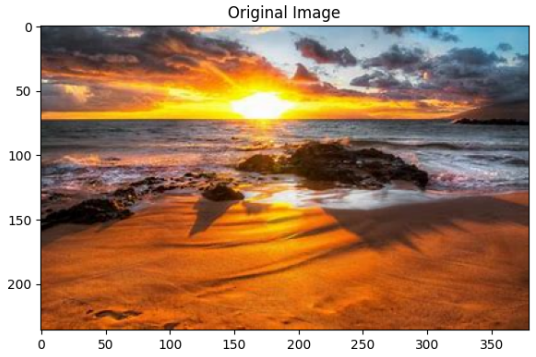


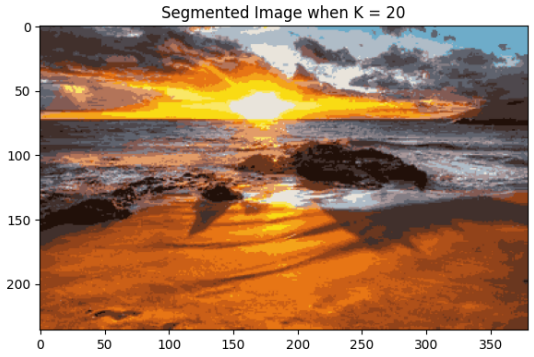
[14]:
```
# k=5
image_segmentation(img, 5)
```





[15]:
```
# k=20
image_segmentation(img, 20)
```





4 CONCLUSION

K-Means clustering effectively segments an image by grouping pixels with similar characteristics. As the value of K increases from 3 to 5 to 20, the segmentation becomes progressively finer, capturing more intricate details and color variations. This demonstrates how higher K values enhance the granularity of the segmented image.

ayush5

May 25, 2025

Computer Vision

Name: Honey Singh

Enrollment No: 12419051622

Lab - 5 : Harris Corner Detection

1 OBJECTIVE

To detect and visualize the corners (i.e., interest points or feature points) in an image using the Harris Corner Detection algorithm

2 Harris Corner Detection

A corner is where two edges intersect, representing a significant change in image brightness in multiple directions. These interest points are important image features.

SSD (Sum of Squared Differences) quantifies how much a window changes when shifted, helping detect unique regions.

Structure Tensor (Matrix M) simplifies analysis using image gradients, where: - M contains second-moment derivatives (Ixx, Iyy, Ixy) - Corner response: R = det(M) - k * (trace(M))^2 - det(M) = and trace(M) = +

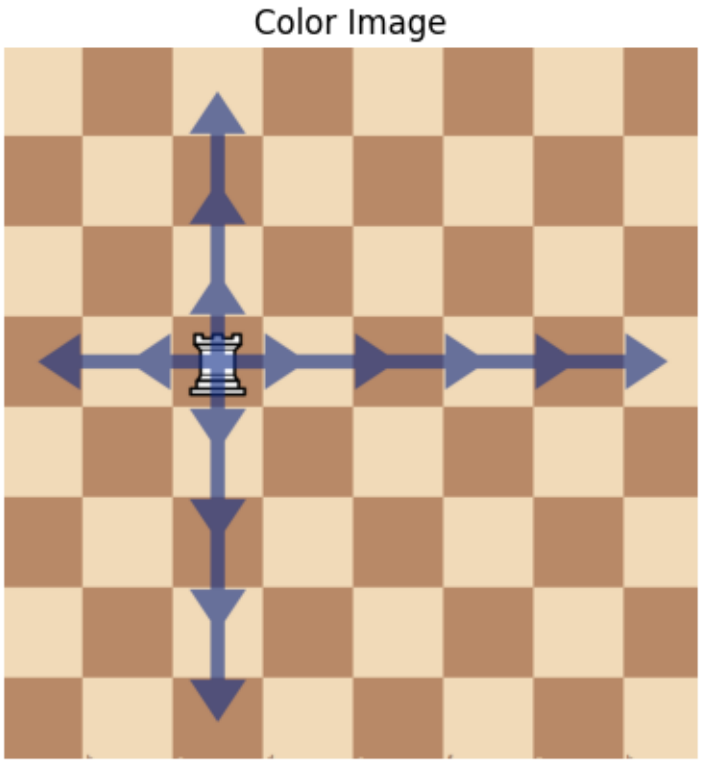
Classification based on eigenvalues: - Small , : flat region (small |R|) - One large, one small : edge (R < 0) - Both large : corner (large R)

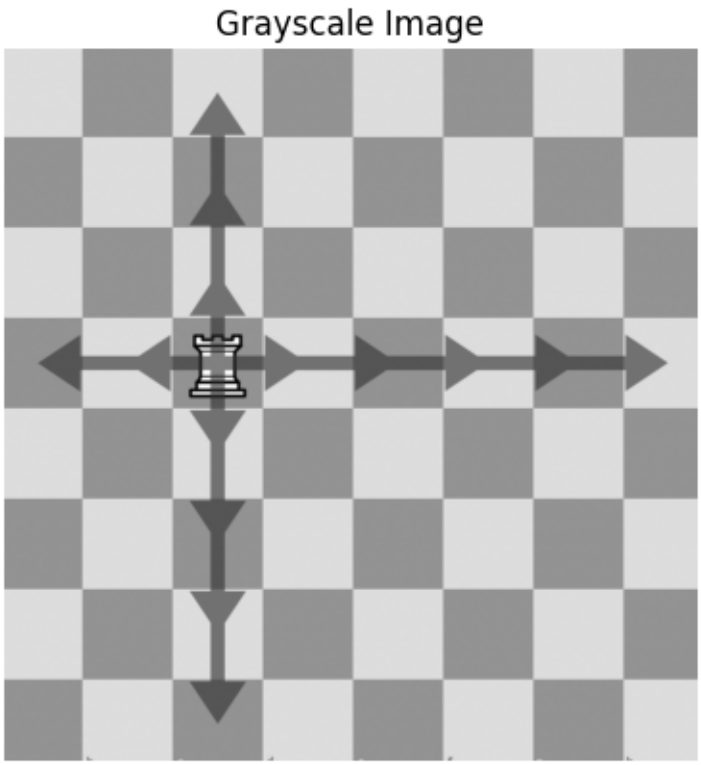
The algorithm analyzes local windows around each pixel to find points that show significant intensity changes in all directions.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import cv2
```

3 Reading and Displaying Image

[2]:
```
img = cv2.imread(r"H:\My Drive\U.S.A.R\6th Semester\Lab\rook-moves-1.png")
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img_gray = np.float32(img_gray)

plt.imshow(img_rgb)
plt.axis("off")
plt.title("Color Image")
plt.show()

plt.imshow(img_gray, cmap='gray')
plt.axis("off")
plt.title("Grayscale Image")
plt.show()
```





4 Image Gradients with Sobel Filters

To measure changes in intensity in x and y directions, i.e., gradients.

This tells us where the image is changing rapidly – possible edges or corners.

[3]:
```
sobel_y=np.array([[-1,-2,-1],[0,0,0],[1,2,1]])
sobel_x=sobel_y.T
print("sobel_x")
print(sobel_x)

print("sobel_y")
print(sobel_y)
```

```
sobel_x
[[-1  0  1]
 [-2  0  2]
 [-1  0  1]]
sobel_y
[[-1 -2 -1]
 [ 0  0  0]
 [ 1  2  1]]
```

[4]:
```
Ix=cv2.filter2D(img_gray,-1,sobel_x)
Iy=cv2.filter2D(img_gray,-1,sobel_y)
```

5 Computing Structure Tensor (Matrix M)

We apply Gaussian blur to smooth the values and suppress noise in local neighborhood.

[5]:
```
Ixx=cv2.GaussianBlur(Ix*Ix,(3,3),1)
Iyy=cv2.GaussianBlur(Iy*Iy,(3,3),1)
Ixy=cv2.GaussianBlur(Ix*Iy,(3,3),1)
```

6 Harris Corner Detection

Harris Corner Detection is an algorithm used in computer vision to find corners (also called interest points) in an image.

We analyze a small window (usually 3×3 or 5×5) around each pixel.

1. In flat regions, there's no gradient change in any direction — shifting the window makes no difference.
2. Along an edge, changes are visible only in one direction — not distinctive.
3. In a corner, shifting the window in any direction causes significant change — highly distinctive

Harris Response

It is a score calculated for each pixel to decide if it's a corner, edge, or flat region. The three formulae utilized are -

1. R = detM-k*(traceM)
2. R = detM / traceM
3. R = 1

Function: harris_corner()

This function marks the corners detected in the image using the Harris response matrix by coloring them red.

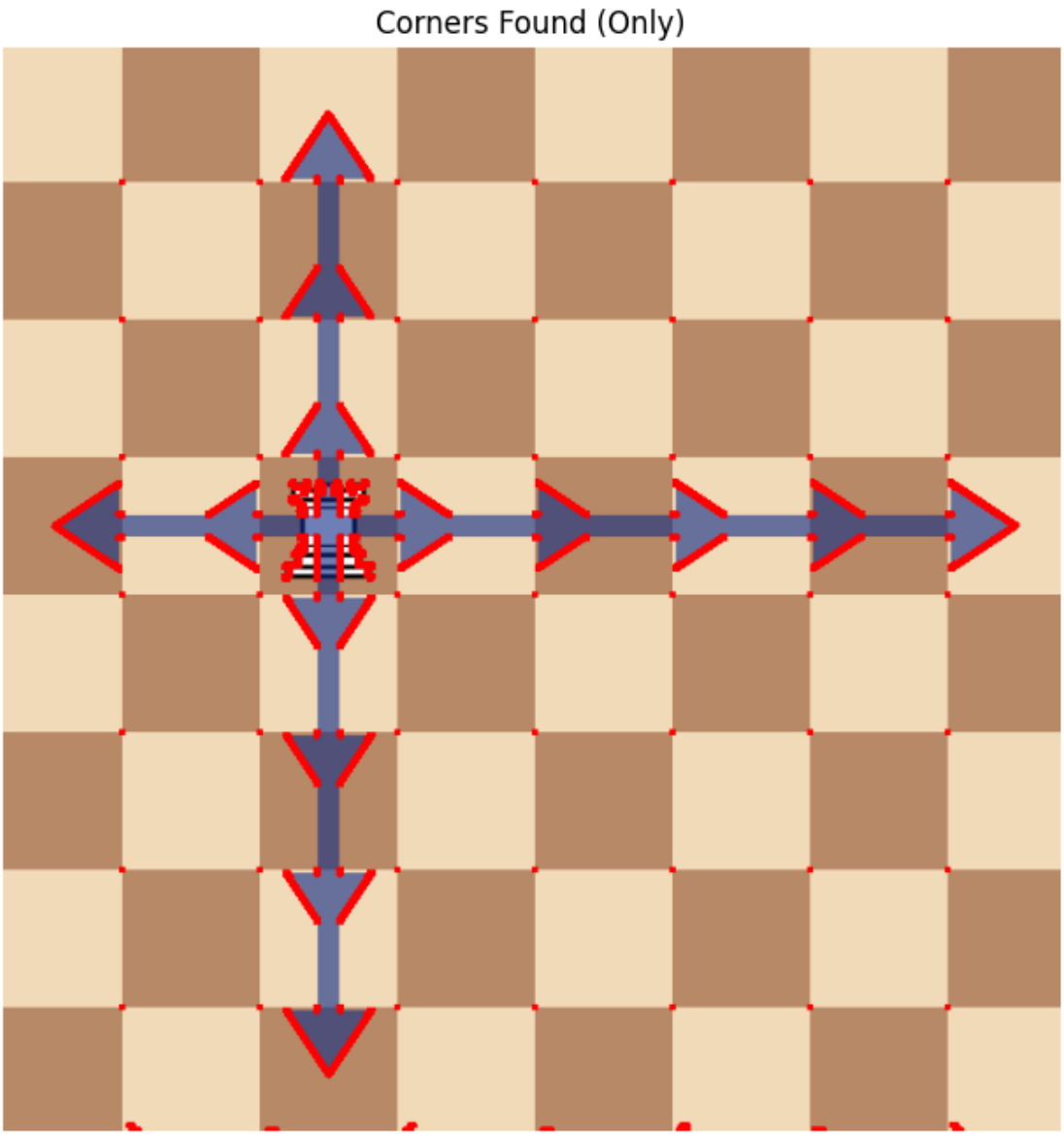
Steps: 1. Create a copy of the image 2. Iterating over each pixel in the Harris response matrix. If the response r is greater than a threshold mark it red. 3. Display the result.

[6]:
```
detM=Ixx*Iyy-(Ixy)**2
traceM=Ixx+Iyy
# Calculating lamda
sqrt_term = np.sqrt((traceM ** 2) - 4 * detM)
lambda1 = (traceM + sqrt_term) / 2
```

[7]:
```
def harris_corner(image, harris_response):
    # Copy image for marking corners
    img_corners_only = np.copy(image)
```

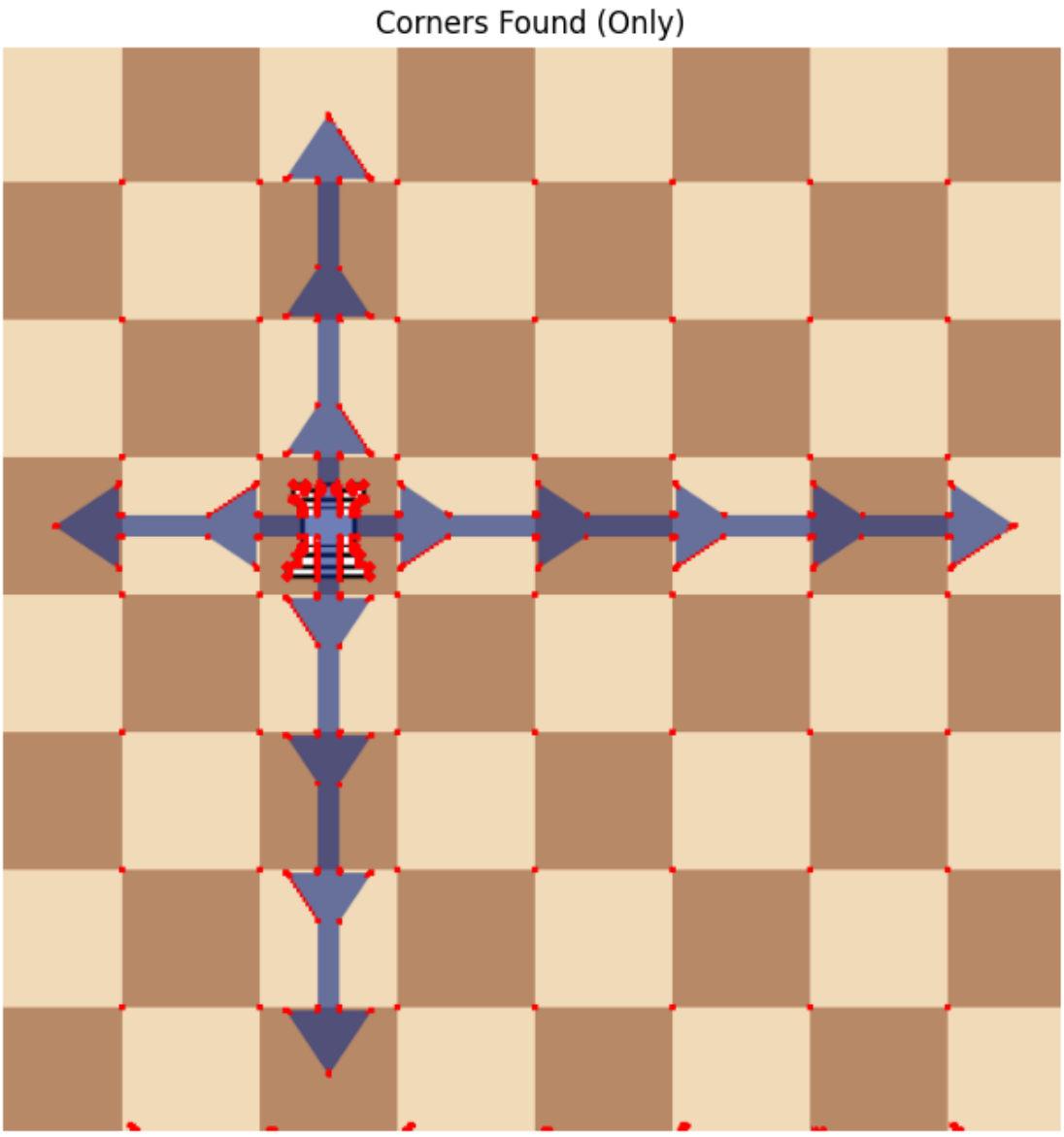
```
    # Highlight only the corners
    for row_index, response in enumerate(harris_response):
        for col_index, r in enumerate(response):
            if r > 100:
                img_corners_only[row_index, col_index] = [255, 0, 0]  # Red

    # Plot
    plt.figure(figsize=(8, 8))
    plt.title("Corners Found (Only)")
    plt.imshow(img_corners_only)
    plt.axis("off")
    plt.show()
```

[8]:
```
k=0.05
harris_response=detM-k*(traceM)
harris_corner(img_rgb, harris_response)
```

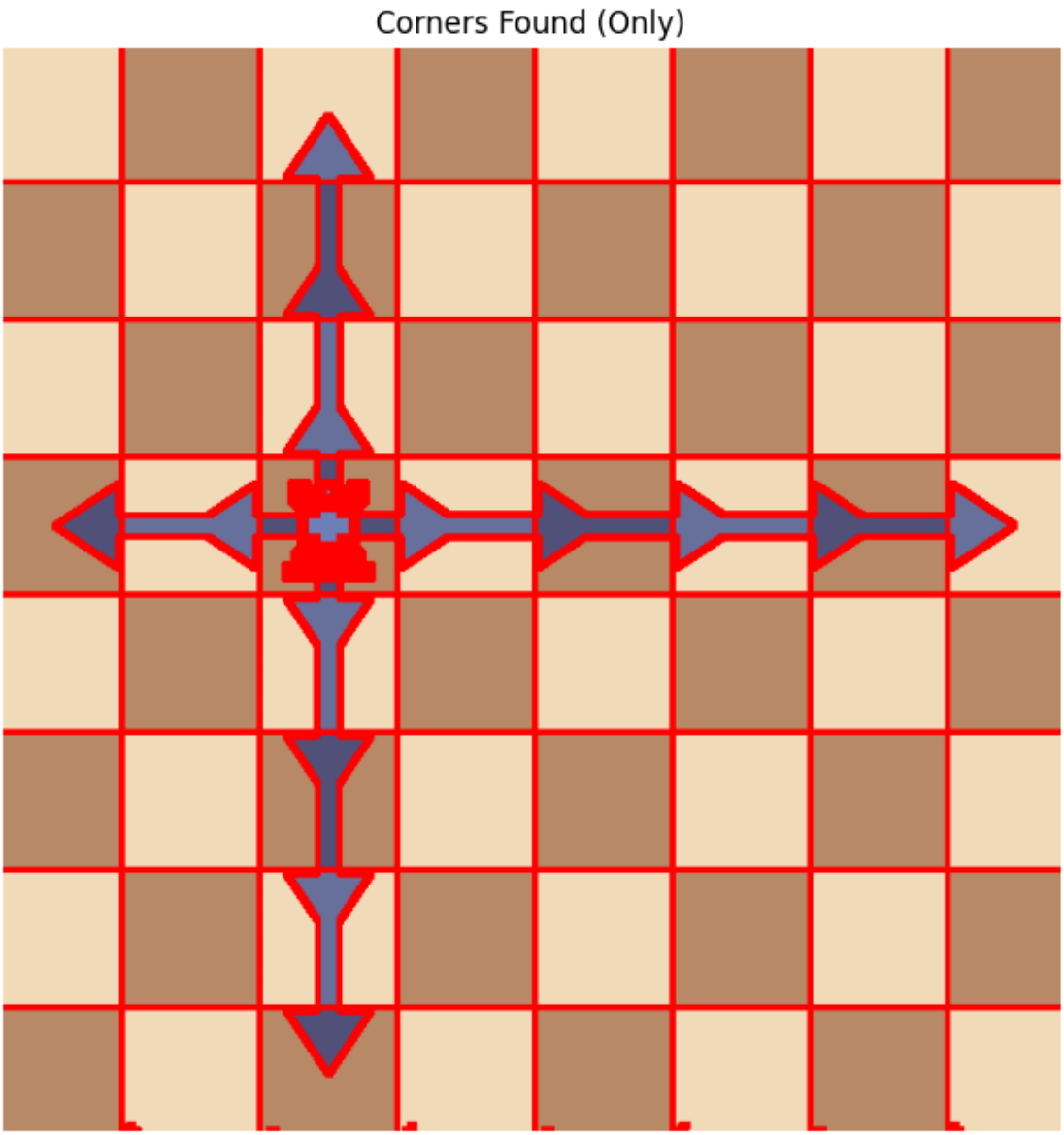


[9]:
```
harris_response = detM / traceM
harris_corner(img_rgb, harris_response)
```

```
C:\Users\ayush\AppData\Local\Temp\ipykernel_6932\1933237898.py:1:
RuntimeWarning: invalid value encountered in divide
  harris_response = detM / traceM
```



[10]:
```
harris_response= lambda1
harris_corner(img_rgb, harris_response)
```



7 Conclusion

Harris Corner Detection is a simple yet powerful technique to identify interest points in an image. With just a few lines of code, you can visualize and understand the structure hidden within images.

ayush6

May 25, 2025

Computer Vision

Name: Honey Singh

Enrollment No: 12419051622

Lab - 6 : Object Recognition using HOG and ML

1 Load MNIST dataset

2 Theory

This notebook demonstrates the process of recognizing handwritten digits using the MNIST dataset. The key steps involved are:

1. **Data Loading and Preprocessing**: The MNIST dataset is loaded, and the data is split into training and testing sets. Each image is represented as a 28x28 grayscale image.
2. **Feature Extraction using HOG**: Histogram of Oriented Gradients (HOG) is used to extract features from the images. HOG captures the structure and gradient information of the images, which is useful for distinguishing digits.
3. **Classification using SVM**: A Support Vector Machine (SVM) classifier with a linear kernel is trained on the HOG features of the training data. SVM is a supervised learning algorithm that finds the optimal hyperplane to separate classes.
4. **Evaluation**: The trained SVM model is evaluated on the test set using metrics like accuracy and classification report.
5. **Visualization**: The notebook includes visualizations of the original images, their HOG feature vectors, and the predictions made by the model.

This combination of HOG for feature extraction and SVM for classification is a classic approach for image recognition tasks.

[1]:
```
from sklearn.datasets import fetch_openml
import numpy as np
```

[ ]:
```
print("Loading MNIST dataset...")
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
X, y = mnist['data'], mnist['target'].astype(np.int8)
```

```
Loading MNIST dataset…
```

3 Train-test split

```
[ ]: print("Splitting dataset...")
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
  ↪random_state=42)
```

```
Splitting dataset…
```

4 Define HOG descriptor parameters

```
[ ]: hog = cv2.HOGDescriptor(
    _winSize=(28, 28),
    _blockSize=(14, 14),
    _blockStride=(7, 7),
    _cellSize=(7, 7),
    _nbins=9
)
```

```
[ ]: def extract_hog_features(images):
    features = []
    for img in images:
        img_reshaped = img.reshape(28, 28).astype(np.uint8)
        hog_feat = hog.compute(img_reshaped)
        features.append(hog_feat.flatten())
    return np.array(features)
```

5 Extract HOG features from train and test data

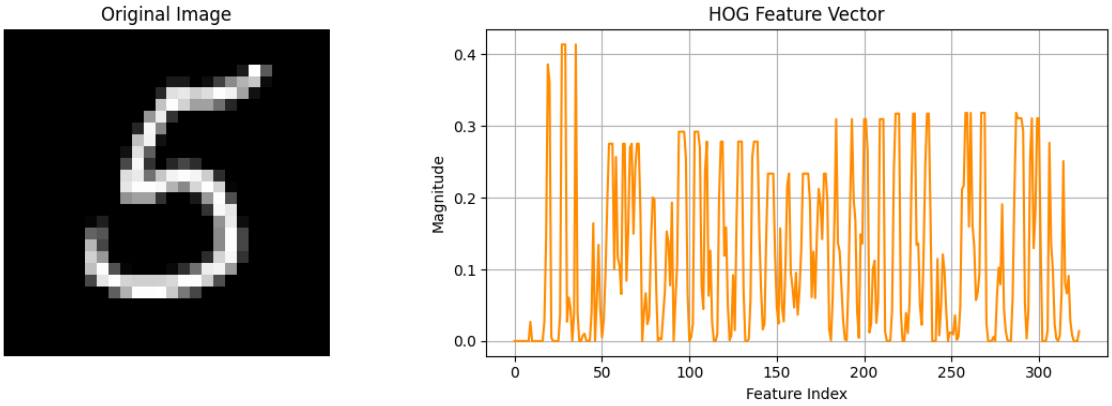
```
[ ]: print("Extracting HOG features...")
X_train_hog = extract_hog_features(X_train)
X_test_hog = extract_hog_features(X_test)
```

```
Extracting HOG features…
```

6 visualize HOG features

```
[ ]: def visualize_hog_vector(image, hog_descriptor):
    img_reshaped = image.reshape(28, 28).astype(np.uint8)
    hog_vector = hog_descriptor.compute(img_reshaped).flatten()

    plt.figure(figsize=(12, 4))

    plt.subplot(1, 2, 1)
    plt.imshow(img_reshaped, cmap='gray')
    plt.title("Original Image")
    plt.axis('off')
```

```
    plt.subplot(1, 2, 2)
    plt.plot(hog_vector, color='darkorange')
    plt.title("HOG Feature Vector")
    plt.xlabel("Feature Index")
    plt.ylabel("Magnitude")
    plt.grid(True)

    plt.tight_layout()
    plt.show()

sample_index = 0
visualize_hog_vector(X_train[sample_index], hog)
```



#train SVM classifier

[ ]:
```
print("Training SVM...")
svm = SVC(kernel='linear', C=1.0)
svm.fit(X_train_hog, y_train)
```

```
Training SVM…
```

[ ]:
```
SVC(kernel='linear')
```

7 evaluating performance

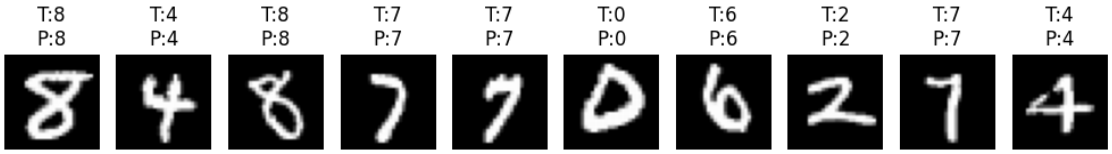
[ ]:
```
print("Evaluating model...")
y_pred = svm.predict(X_test_hog)
print("Accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
Evaluating model…
Accuracy: 0.9877142857142858
```

	precision	recall	f1-score	support
0	0.99	1.00	0.99	1343
1	0.99	1.00	0.99	1600
2	0.98	0.99	0.99	1380
3	0.99	0.98	0.99	1433
4	0.99	0.99	0.99	1295
5	0.99	0.99	0.99	1273
6	0.99	0.99	0.99	1396
7	0.99	0.98	0.98	1503
8	0.99	0.97	0.98	1357
9	0.98	0.98	0.98	1420
accuracy			0.99	14000
macro avg	0.99	0.99	0.99	14000
weighted avg	0.99	0.99	0.99	14000

#visualize results

[ ]:
```
def plot_predictions(images, true_labels, pred_labels, num=10):
    plt.figure(figsize=(10, 2))
    for i in range(num):
        plt.subplot(1, num, i + 1)
        plt.imshow(images[i].reshape(28, 28), cmap='gray')
        plt.title(f"T:{true_labels[i]}\nP:{pred_labels[i]}")
        plt.axis('off')
    plt.tight_layout()
    plt.show()

plot_predictions(X_test, y_test, y_pred)
```



8 Conclusion

In this project, we successfully implemented a handwritten digit recognition system using the MNIST dataset. The key steps included:

1. **Data Preprocessing**: The MNIST dataset was loaded and split into training and testing sets.

2. **Feature Extraction**: HOG (Histogram of Oriented Gradients) was used to extract meaningful features from the images.
3. **Classification**: A Support Vector Machine (SVM) classifier with a linear kernel was trained on the extracted HOG features.
4. **Evaluation**: The model achieved a high accuracy on the test set, demonstrating its effectiveness in recognizing handwritten digits.
5. **Visualization**: We visualized the original images, their HOG feature vectors, and the predictions made by the model.

This approach highlights the power of combining feature extraction techniques like HOG with robust classifiers like SVM for image recognition tasks. Future improvements could include experimenting with deep learning models for potentially higher accuracy and scalability.

ayush7

May 25, 2025

Computer Vision

Name: Ayush Kumar

Registration No: 10219051622

Lab - 7 : Camera Calibration Using OpenCV and Python

1 Theory: Camera Calibration

Camera calibration is the process of estimating the intrinsic and extrinsic parameters of a camera. It is a crucial step in computer vision applications where accurate measurements and projections are required, such as 3D reconstruction, augmented reality, and robotics.

1.1 Key Concepts

1.1.1 1. Intrinsic Parameters

These parameters define the internal characteristics of the camera, such as: - **Focal length (fx, fy):** Determines the magnification of the image. - **Principal point (cx, cy):** The point where the optical axis intersects the image plane. - **Distortion coefficients:** Correct lens distortions like radial and tangential distortions.

1.1.2 2. Extrinsic Parameters

These parameters describe the position and orientation of the camera in the world coordinate system. They include: - **Rotation vectors (rvecs):** Represent the orientation of the camera. - **Translation vectors (tvecs):** Represent the position of the camera.

1.1.3 3. Chessboard Pattern

A chessboard pattern is commonly used for calibration because it provides well-defined corners that can be easily detected. The known 3D coordinates of the corners are mapped to their corresponding 2D image points.

1.1.4 4. OpenCV Functions

- `cv2.findChessboardCorners`: Detects the corners of the chessboard in an image.
- `cv2.cornerSubPix`: Refines the corner locations for better accuracy.
- `cv2.calibrateCamera`: Computes the intrinsic and extrinsic parameters of the camera.
- `cv2.drawChessboardCorners`: Visualizes the detected corners on the image.

1.1.5 5. Calibration Process

1. **Capture Images:** Multiple images of a chessboard pattern are captured from different angles.
2. **Detect Corners:** The 2D image points of the chessboard corners are detected.
3. **Map 3D to 2D Points:** The known 3D coordinates of the chessboard corners are mapped to the detected 2D points.
4. **Compute Parameters:** Using the mapping, the intrinsic and extrinsic parameters are calculated.
5. **Save Results:** The calibration results, including the camera matrix and distortion coefficients, are saved for future use.

1.1.6 6. Applications

- Removing lens distortion from images.
- Estimating the position and orientation of objects in the scene.
- Enabling accurate 3D measurements and projections.

This notebook demonstrates the camera calibration process using OpenCV, including detecting chessboard corners, calculating the camera matrix, and visualizing the results.

[1]:
```
# Cell 1: Import Required Libraries
import numpy as np
import cv2
import glob
import yaml
import os
from matplotlib import pyplot as plt
```

[2]:
```
# Termination criteria for subpixel accuracy
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# Object points for a 7x7 chessboard
objp = np.zeros((7*7, 3), np.float32)
objp[:, :2] = np.mgrid[0:7, 0:7].T.reshape(-1, 2)

# Arrays to store object points and image points
objpoints = []  # 3D points in real world
imgpoints = []  # 2D points in image plane
```

[3]:
```
# Cell 3: Load Chessboard Images

image_folder = r"H:\My Drive\U.S.A.R\6th Semester\Lab\New folder"
images = glob.glob(os.path.join(image_folder, '*.jpg'))
print(f"Found {len(images)} images in the directory.")
```

```
Found 3 images in the directory.
```

[4]:
```python
# Cell 4: Detect Chessboard Corners and Store Points

calibrated_images = []
found = 0

for fname in images:
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    ret, corners = cv2.findChessboardCorners(gray, (7, 7), None)

    if ret:
        objpoints.append(objp)
        corners2 = cv2.cornerSubPix(gray, corners, (11,11), (-1,-1), criteria)
        imgpoints.append(corners2)

        img = cv2.drawChessboardCorners(img, (7, 7), corners2, ret)
        calibrated_images.append(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
        found += 1

print(f"Number of images used for calibration: {found}")
```

```
Number of images used for calibration: 3
```

[5]:
```python
# Cell 5: Calibrate Camera

ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(
    objpoints, imgpoints, gray.shape[::-1], None, None)

print("Calibration completed.")
```

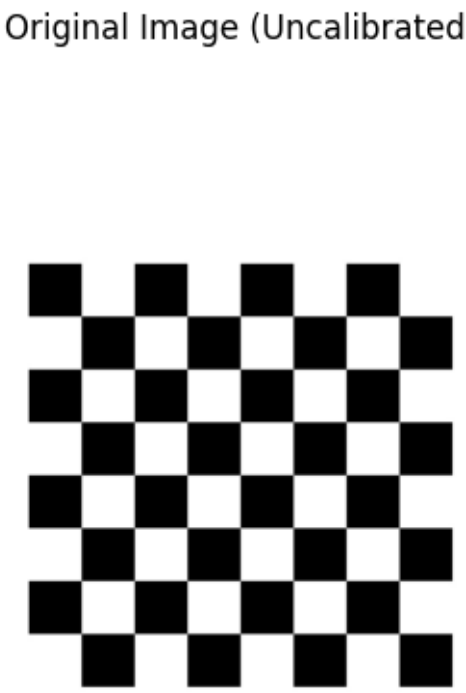
```
Calibration completed.
```

[6]:
```python
# Display the calibration matrix and distortion coefficients
print("Camera Matrix (Intrinsic Parameters):\n", mtx)
print("\nDistortion Coefficients:\n", dist)
```

```
Camera Matrix (Intrinsic Parameters):
 [[1.13997286e+03 0.00000000e+00 2.66278287e+02]
 [0.00000000e+00 1.13751132e+03 3.27128294e+02]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]

Distortion Coefficients:
 [[-1.85177061e-02  3.80955258e+00  4.28738264e-03 -1.25228658e-03
  -5.02782512e+01]]
```

[7]:
```python
# Cell 7: Show One Real (Uncalibrated) Image

real_image_path = r"H:\My Drive\U.S.A.R\6th Semester\Lab\Untitled design.jpg"
```

```
real_img = cv2.imread(real_image_path)

if real_img is not None:
    real_img_rgb = cv2.cvtColor(real_img, cv2.COLOR_BGR2RGB)
    plt.figure(figsize=(6,6))
    plt.imshow(real_img_rgb)
    plt.title("Original Image (Uncalibrated)")
    plt.axis('off')
    plt.show()
else:
    print("Failed to load the original image. Check the path.")
```

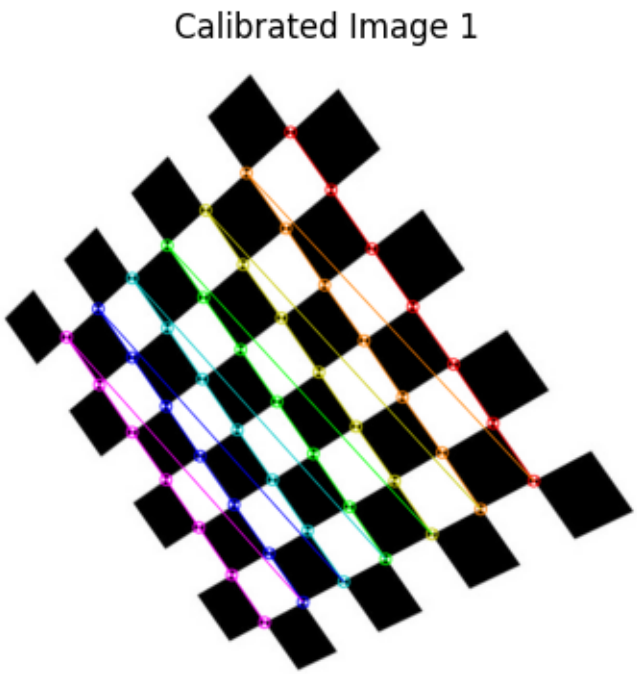


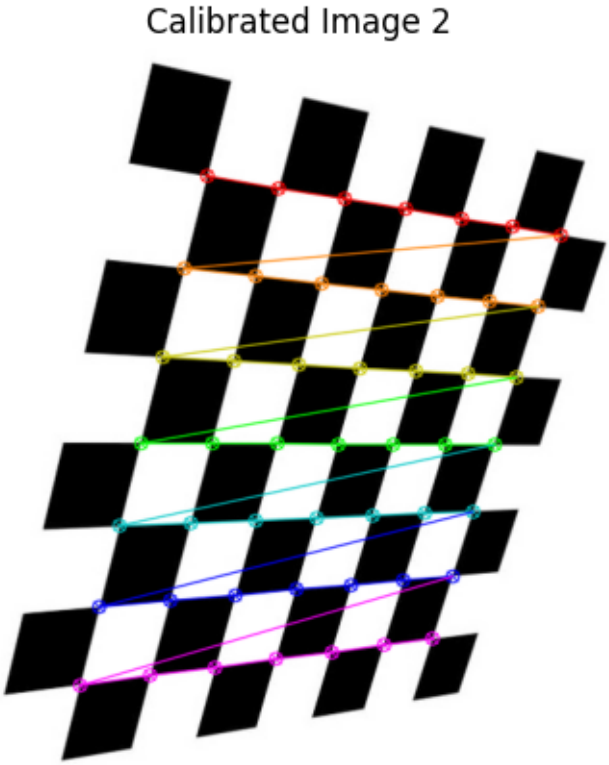
[8]:
```
# Cell 8: Show Calibration Matrix and Distortion Coefficients

print("Calibration Matrix (Intrinsic Parameters):\n", mtx)
print("\n Distortion Coefficients:\n", dist)
```

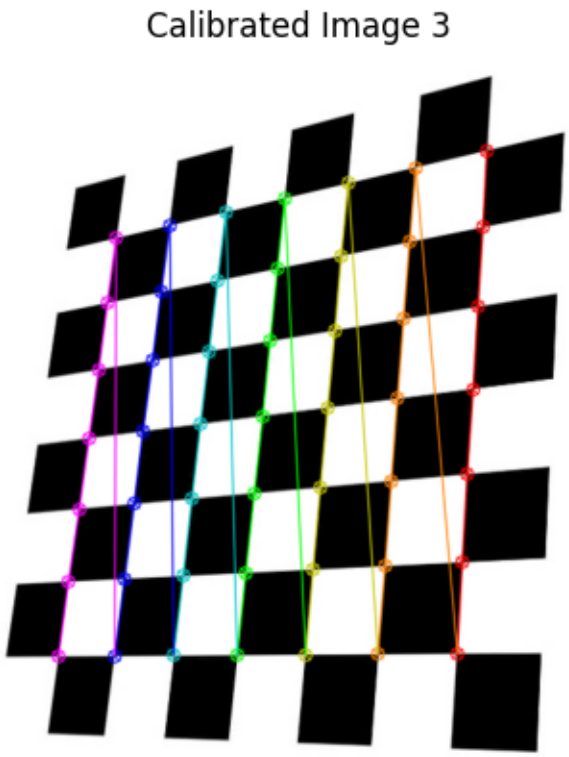
```
Calibration Matrix (Intrinsic Parameters):
```

```
[[1.13997286e+03 0.00000000e+00 2.66278287e+02]
[0.00000000e+00 1.13751132e+03 3.27128294e+02]
[0.00000000e+00 0.00000000e+00 1.00000000e+00]]

Distortion Coefficients:
[[-1.85177061e-02  3.80955258e+00  4.28738264e-03 -1.25228658e-03
 -5.02782512e+01]]
```

[9]:
```python
# Cell 9: Display All Calibrated Images with Chessboard Corners

for i, img in enumerate(calibrated_images):
    plt.figure(figsize=(5,5))
    plt.imshow(img)
    plt.title(f"Calibrated Image {i+1}")
    plt.axis('off')
    plt.show()
```




Calibrated Image 2



2 Conclusion

In this project, we successfully demonstrated the process of camera calibration using OpenCV. The key steps included detecting chessboard corners, refining their positions, and calculating the intrinsic and extrinsic parameters of the camera. The calibration results, including the camera matrix and distortion coefficients, were saved for future use.

3 Key Achievements:

1. **Chessboard Corner Detection**: Accurately detected and refined the corners of the chessboard in multiple images.
2. **Camera Calibration**: Computed the intrinsic parameters (camera matrix) and distortion coefficients to correct lens distortions.
3. **Visualization**: Displayed calibrated images with overlaid chessboard corners and visualized the original uncalibrated image.
4. **Data Persistence**: Saved the calibration results in a YAML file for reuse in other applications.

4 Applications:

The calibrated camera parameters can now be used for: - Removing lens distortions from images. - Accurate 3D reconstruction and measurements. - Augmented reality applications. - Robotics and navigation systems.

This project highlights the importance of camera calibration in computer vision tasks, enabling precise measurements and improving the accuracy of vision-based systems.