Index

Experiment 1

Aim: Basic Image processing operations.

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

Importing Libraries:

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

- `import numpy as np` : Imports NumPy for numerical operations and array handling.
- `import matplotlib.pyplot as plt` : Imports Matplotlib for plotting graphs and images.
- `import cv2` : Imports OpenCV for image processing tasks.

Reading and displaying an image in python

```
In [6]: img_path ="cameraman.tif"
        c_img = cv2.imread(img_path)
        plt.imshow(c_img , cmap="gray")
        plt.axis("off")
```

```
Out[6]:  (np.float64(-0.5), np.float64(255.5), np.float64(255.5), np.float64(-0.5))
```

```
In [7]: c_img.shape

Out[7]: (256, 256, 3)
```

- Loads the image `"cameraman.tif"` using OpenCV.
- Displays the image in grayscale using Matplotlib.
- Hides axis ticks for cleaner display.

Convert Images to another format and compare them:

```
In [11]: from PIL import Image
         imgg = Image.open(img_path)
         imgg.save("output.jpeg" , format ='JPEG')
         imgg.save("output.png" , format = 'PNG')
```

Convert an Image to Grayscale and display:

- Opens the image using PIL and saves it in JPEG and PNG formats.
- Reads the image again with OpenCV and converts it to grayscale.
- Displays the grayscale image with a title and no axis.

```
In [13]: img = cv2.imread("cameraman.tif")
         gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
         plt.imshow(gray_img, cmap='gray')

         plt.title("Gray Img")
         plt.axis("off")
         plt.show()
```

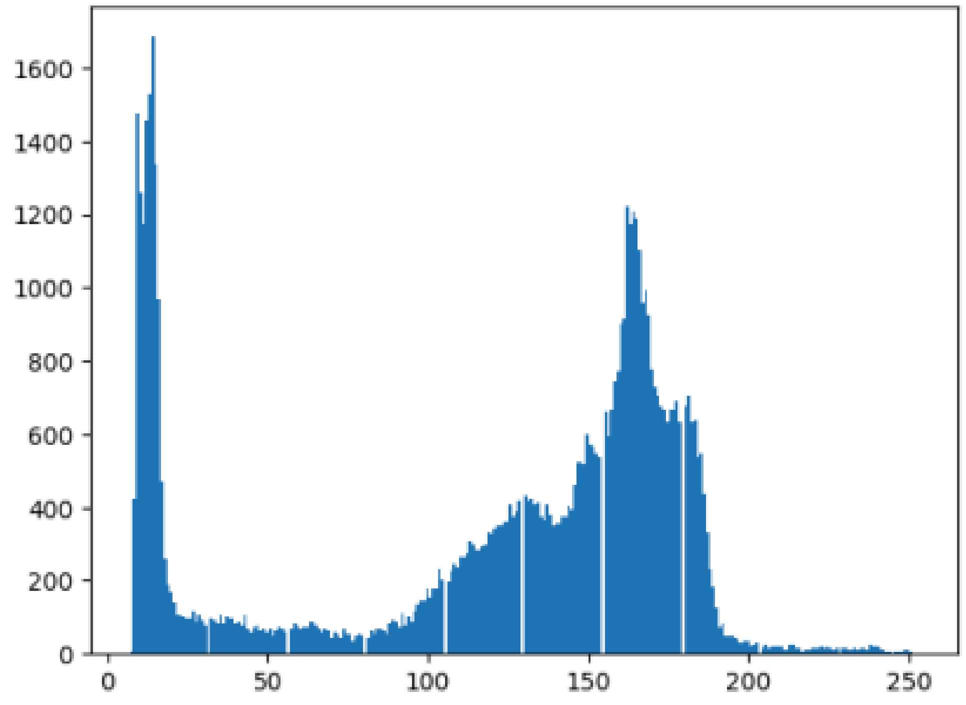
Gray Img

Perform Image enhancement operations:

Computing histogram:

- Computes the histogram of the grayscale image using pixel intensity values.
- Flattens the histogram to a 1D array for easy processing.
- Plots the histogram using Matplotlib to visualize pixel intensity distribution.

In [29]:
```
histogram_values ,bin_edges ,_ = plt.hist(gray_img.flatten() , bins =256 )
```



In [30]:
```
histogram_values = histogram_values.astype(int)
histogram_values[:50]
```

Out[30]:
```
array([   4,  423, 1477, 1259, 1175, 1456, 1529, 1685, 1338,  968,  471,
        261,  187,  169,  140,  107,  109,  104,   96,   99,  114,   86,
        108,   91,   76,    0,   99,   92,   83,  105,   85,  100,   98,
         97,   81,   86,   80,  106,   68,   59,   72,   77,   65,   70,
         61,   67,   50,   64,   74,   67])
```

Computing Probablity Density Function

- Calculates the **PDF** by dividing each histogram value by the total number of pixels.
- Represents the probability of each intensity level occurring in the image.
- Useful for understanding intensity distribution statistically.

In [31]:
```
histogram = np.zeros(256, dtype=int)
for pixel in gray_img.flatten():
    histogram[pixel] += 1

total_pixels = gray_img.size
probability_density = histogram / total_pixels

print("Intensity | Probability")
print("-" * 25)
for i in range(20):
    formatted_prob = f"{probability_density[i]:.6f}"
    print(f"{i:9d} | {formatted_prob}")
```

```
Intensity | Probability
-------------------------
        0 | 0.000000
        1 | 0.000000
        2 | 0.000000
        3 | 0.000000
        4 | 0.000000
        5 | 0.000000
        6 | 0.000000
        7 | 0.000061
        8 | 0.006454
        9 | 0.022537
       10 | 0.019211
       11 | 0.017929
       12 | 0.022217
       13 | 0.023331
       14 | 0.025711
       15 | 0.020416
       16 | 0.014771
       17 | 0.007187
       18 | 0.003983
       19 | 0.002853
```

Computing Cumulative Distribution Function

- Computes the **CDF** using cumulative sum of the PDF.
- Represents the cumulative probability up to each intensity level.
- Used to redistribute pixel intensities in histogram equalization.

In [32]:
```
cdf = np.cumsum(probability_density)
print("Intensity | CDF")
print("-" * 20)
for i in range(20):
    formatted_cdf = f"{cdf[i]:.6f}"
    print(f"{i:9d} | {formatted_cdf}")
```

```
Intensity | CDF
--------------------
        0 | 0.000000
        1 | 0.000000
        2 | 0.000000
        3 | 0.000000
        4 | 0.000000
        5 | 0.000000
        6 | 0.000000
        7 | 0.000061
        8 | 0.006516
        9 | 0.029053
       10 | 0.048264
       11 | 0.066193
       12 | 0.088409
       13 | 0.111740
       14 | 0.137451
       15 | 0.157867
       16 | 0.172638
       17 | 0.179825
       18 | 0.183807
       19 | 0.186661
```

In [33]:
```
l = 256
cdf_normalized = np.round(cdf * (l - 1)).astype(np.uint8)
print("Intensity | Normalized CDF")
print("-" * 30)
for i in range(20):
    print(f"{i:9d} | {cdf_normalized[i]:15d}")
```

```
Intensity | Normalized CDF
------------------------------
        0 |               0
        1 |               0
        2 |               0
        3 |               0
        4 |               0
        5 |               0
        6 |               0
        7 |               0
        8 |               2
        9 |               7
       10 |              12
       11 |              17
       12 |              23
       13 |              28
       14 |              35
       15 |              40
       16 |              44
       17 |              46
       18 |              47
       19 |              48
```

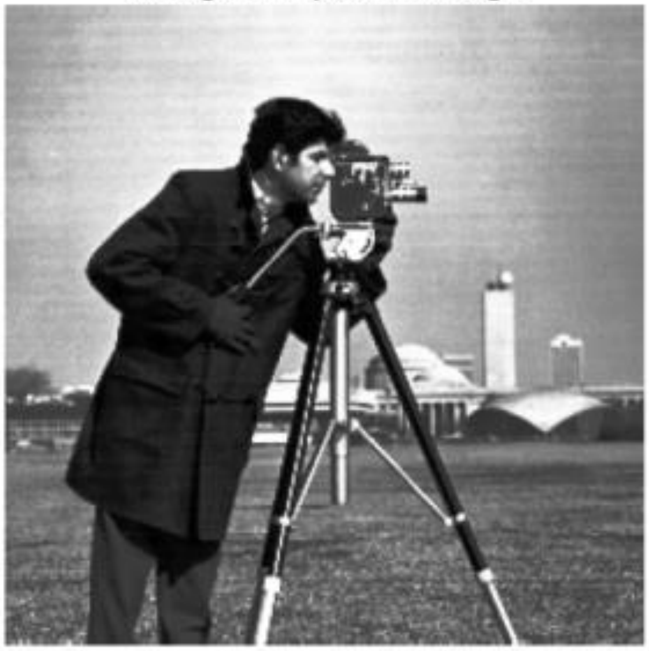
- Normalizes the CDF to the range [0, 255] by scaling.
- Converts values to `uint8` for image processing compatibility.
- Displays the first 20 normalized CDF values with their corresponding intensity levels.

Histogram Equalization

In [34]:
```
equalized_img = cdf_normalized[gray_img]
print(equalized_img)
plt.imshow(equalized_img, cmap='gray')
plt.title("Histogram Equalized Image")
plt.axis("off")
plt.show()
```

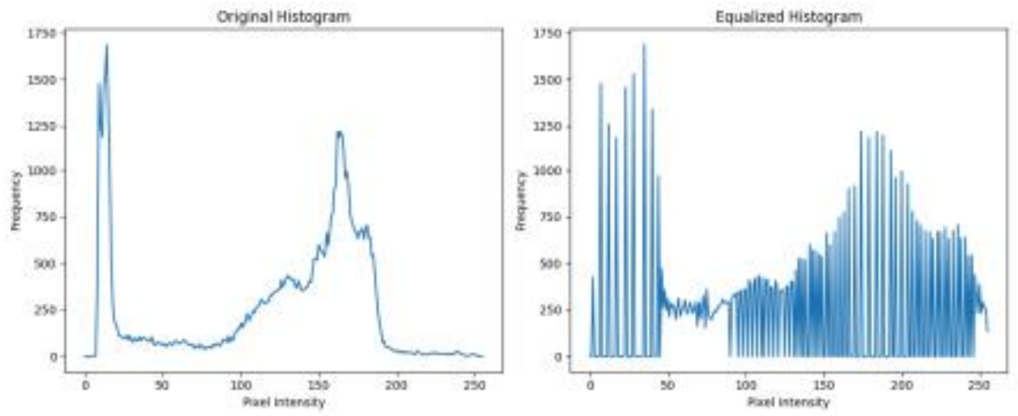
```
[[154 163 160 ... 143 145 145]
 [166 149 157 ... 149 152 147]
 [154 163 160 ... 143 145 145]
 ...
 [ 86 111  97 ... 116 119  86]
 [ 94 102 108 ... 113 108  85]
 [ 94 102 108 ... 113 108  85]]
```

Histogram Equalized Image

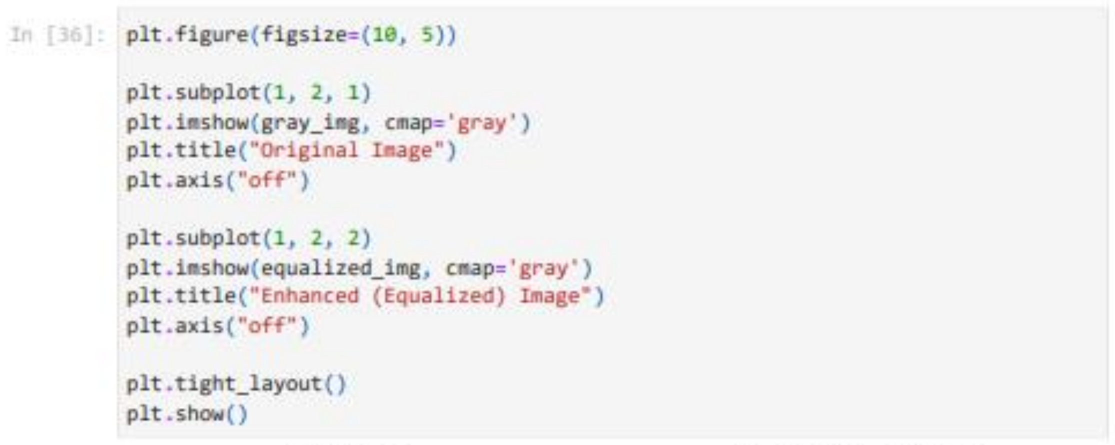


Plotting Orignal vs Equalized Histogram

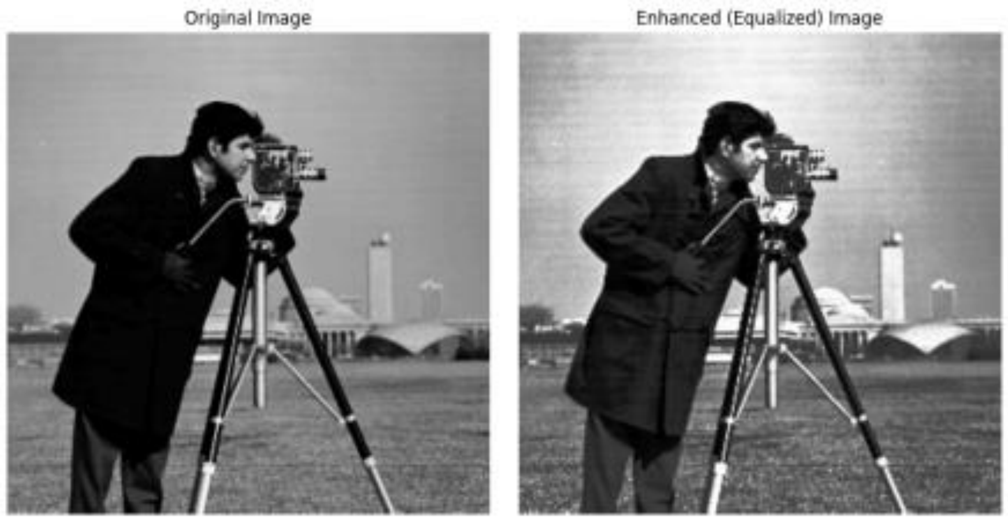
In [35]:
```
original_hist = cv2.calcHist([gray_img], [0], None, [256], [0, 256]).flatten()
equalized_hist = cv2.calcHist([equalized_img], [0], None, [256], [0, 256]).flatt

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(original_hist)
plt.title("Original Histogram")
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")

plt.subplot(1, 2, 2)
plt.plot(equalized_hist)
plt.title("Equalized Histogram")
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")

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



Plotting Orignal vs Enhanced Image



```
In [36]: plt.figure(figsize=(10, 5))

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

         plt.subplot(1, 2, 2)
         plt.imshow(equalized_img, cmap='gray')
         plt.title("Enhanced (Equalized) Image")
         plt.axis("off")

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



Conclusion:
Loaded the image using OpenCV. Converted the image to grayscale. Computed and printed the histogram. Plotted the grayscale histogram. Calculated the Probability Density Function (PDF). Computed the Cumulative Distribution Function (CDF). Normalized the CDF to the range [0, 255]. Applied histogram equalization using normalized CDF. Printed and displayed the equalized image. Computed histogram for the equalized image. Plotted original vs equalized histograms. Displayed original vs enhanced (equalized) images.

Experiment 4

Aim: Image Segmentation using K-Means algorithm.

Objectives: The objective of this lab is to introduce the student to Image Segmentation using K Means algorithm.

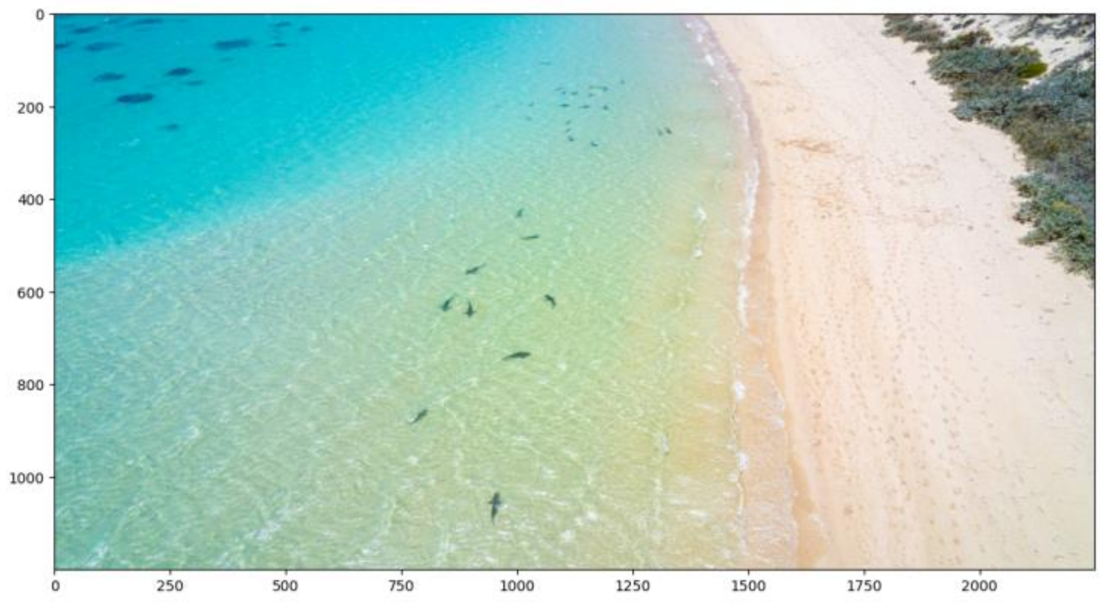
1. Applying K-means algorithm on given image
2. Try different "K" values and observe the results

Importing Libraries:

```
In [14]: import cv2
         import numpy as np
         from matplotlib import pyplot as plt
```

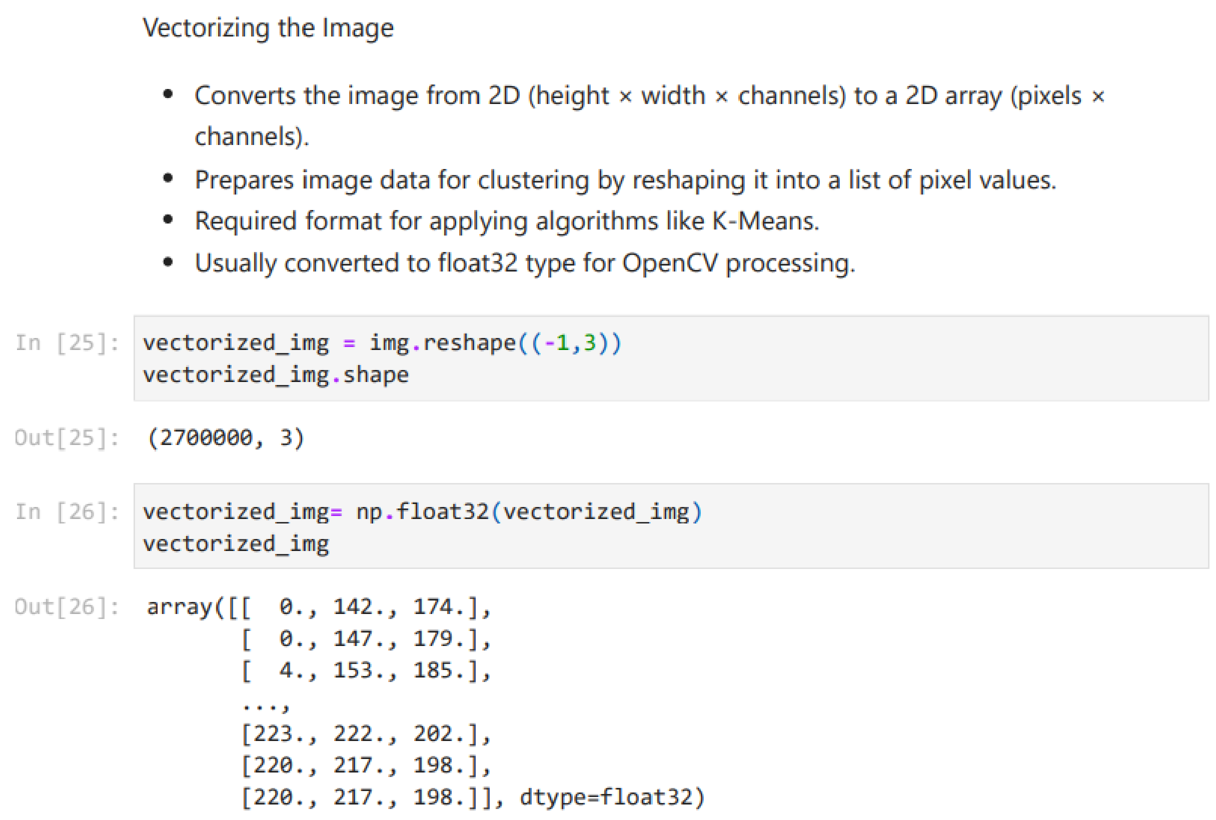
Reading and displaying an image in python

```
In [23]: img = cv2.imread("miximage.jpg")
         img=cv2.cvtColor(img ,cv2.COLOR_BGR2RGB)
         plt.figure(figsize=(13,10))
         plt.imshow(img)

Out[23]: <matplotlib.image.AxesImage at 0x28033423350>
```



```
In [24]: img.shape

Out[24]: (1200, 2250, 3)
```

Vectorizing the image:



Vectorizing the Image

- Converts the image from 2D (height × width × channels) to a 2D array (pixels × channels).
- Prepares image data for clustering by reshaping it into a list of pixel values.
- Required format for applying algorithms like K-Means.
- Usually converted to float32 type for OpenCV processing.

```
In [25]: vectorized_img = img.reshape((-1,3))
         vectorized_img.shape

Out[25]: (2700000, 3)

In [26]: vectorized_img= np.float32(vectorized_img)
         vectorized_img

Out[26]: array([[  0., 142., 174.],
                [  0., 147., 179.],
                [  4., 153., 185.],
                ...,
                [223., 222., 202.],
                [220., 217., 198.],
                [220., 217., 198.]], dtype=float32)
```

Image Segmentation:

Image Segmentation is the process of dividing a digital image into multiple segments or regions, often referred to as image objects. Each segment consists of pixels that are similar with respect to certain characteristics, such as color, intensity, or texture.

Purpose of Image Segmentation:

- Simplifies image representation.
- Makes the image more meaningful and easier to analyze.
- Helps in identifying objects and boundaries in images.

K-Means Algorithm for Image Segmentation:

K-Means Clustering is an unsupervised learning algorithm that partitions image pixels into **K** clusters based on their features (like color or intensity). Similar pixels are grouped together, which helps in segmenting the image.

Key Steps:

1. Convert the image into a 2D array (vectorizing).
2. Apply K-Means clustering to group pixels.
3. Replace each pixel with its cluster center value.

`cv2.kmeans()` Parameters:

- **criteria**: Defines the termination criteria of the algorithm.
  - `cv2.TERM_CRITERIA_EPS` : Stop if the specified accuracy (epsilon) is reached.
  - `cv2.TERM_CRITERIA_MAX_ITER` : Stop after a specified number of iterations.
  - `cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER` : Stop when either condition is met.
- **attempts**: Number of times the algorithm is executed using different initial labelings. The best output in terms of compactness is selected.
- **flags**: Method for choosing the initial centers.
  - `cv2.KMEANS_PP_CENTERS` : K-Means++ center initialization.
  - `cv2.KMEANS_RANDOM_CENTERS` : Random center initialization.

In [44]:
```
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
K = 3
attempts = 10
```

- **attempts**: Number of times the algorithm runs with different initial labelings. The best output (with minimum compactness) is chosen.
- **flags**: Specifies how initial cluster centers are chosen.
  - `cv2.KMEANS_PP_CENTERS` : Uses K-Means++ initialization method.
  - `cv2.KMEANS_RANDOM_CENTERS` : Chooses initial centers randomly.

In [45]:
```
_, label, center = cv2.kmeans(vectorized_img, K, None, criteria, attempts, cv2.K
center = np.uint8(center)
```

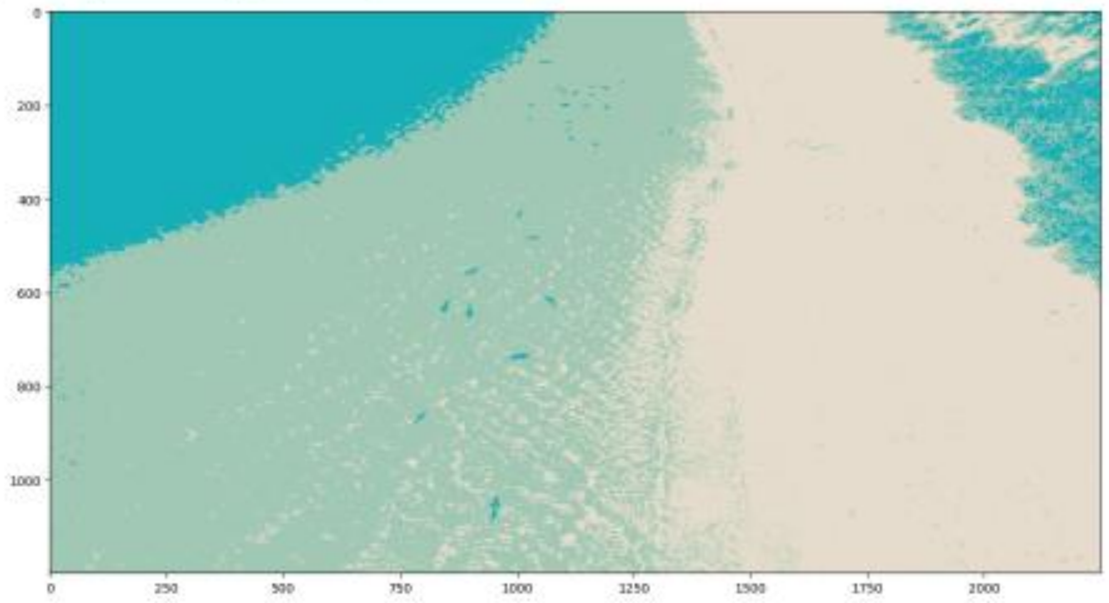
In [46]:
```
center = np.uint8(center)
center
```

Out[46]:
```
array([[233, 224, 207],
       [ 24, 175, 188],
       [160, 203, 182]], dtype=uint8)
```

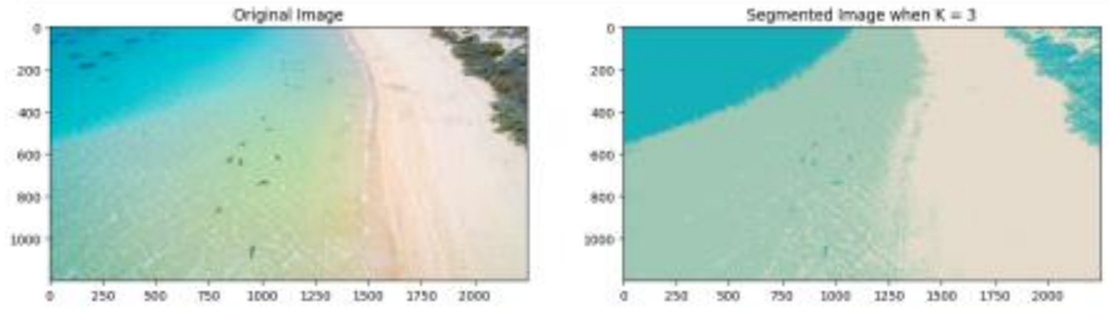
In [47]:
```
res = center[label.flatten()]
result_image = res.reshape((img.shape))
```

In [48]:
```
plt.figure(figsize=(15,10))
plt.imshow(result_image)
```

```
Out[48]: <matplotlib.image.AxesImage at 0x28039073830>
```



```
In [49]: 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()
```



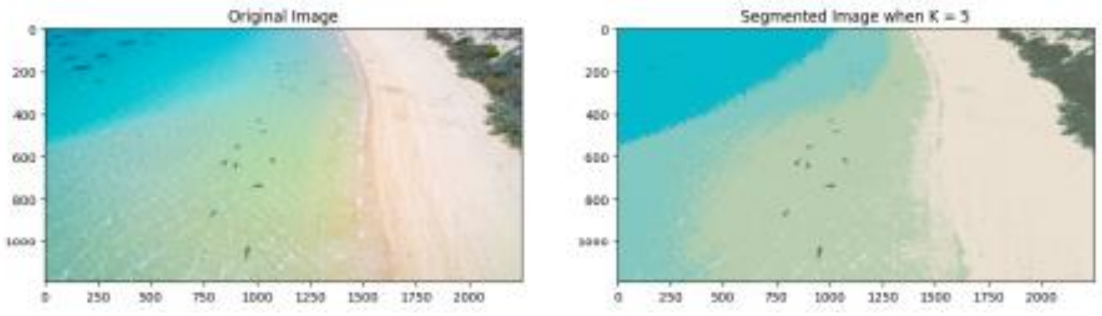
K = 5 in K-Means Clustering

Using **K = 5** means the image pixels will be grouped into 5 different clusters based on their similarity (e.g., color or intensity). Each cluster represents a segmented region in the image. This helps to simplify and highlight different parts of the image for further analysis or visualization.

```
In [50]: K = 5
         attempts = 10
         ret, label, center = cv2.kmeans(vectorized_img, K, None, criteria, attempts, cv2
         center = np.uint8(center)
```

```
In [51]: res = center[label.flatten()]
         result_image = res.reshape((img.shape))
```

```
In [52]: 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()
```



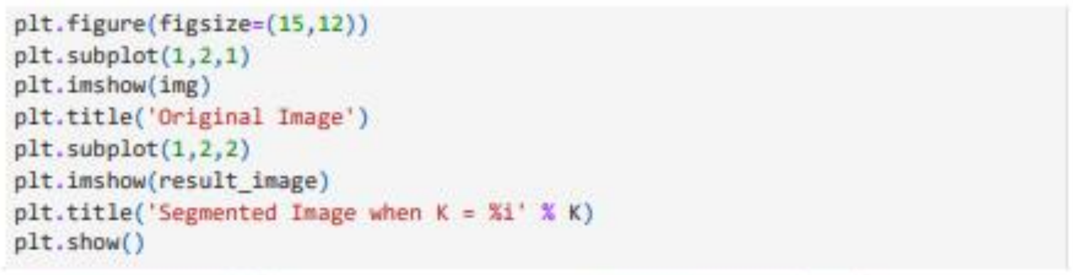
K = 20 in K-Means Clustering

Using **K = 20** divides the image into 20 distinct clusters. This allows for more detailed segmentation, capturing finer variations in color or intensity. It can reveal subtle structures in the image but may also introduce noise if K is too high for the content**.**

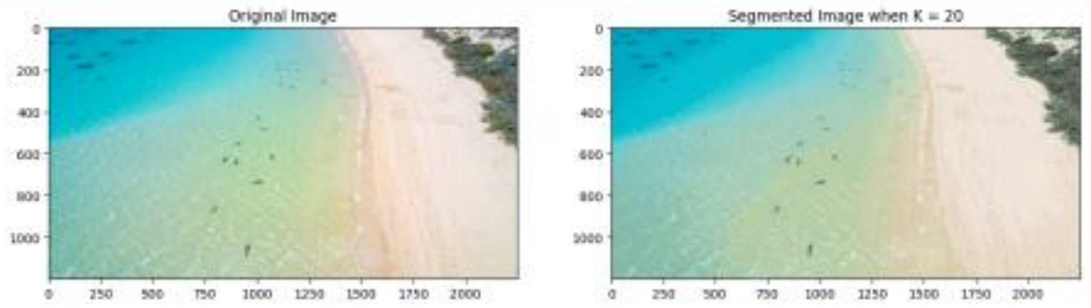
```
In [53]: K = 20
         attempts = 10
         ret, label, center = cv2.kmeans(vectorized_img, K, None, criteria, attempts, cv2
         center = np.uint8(center)
         center
```

```
Out[53]: array([[235, 221, 204],
                [184, 208, 169],
                [127, 136, 124],
                [122, 213, 217],
                [ 94, 102,  90],
                [ 95, 206, 213],
                [136, 189, 171],
                [152, 214, 209],
                [244, 232, 224],
                [  5, 193, 210],
                [139, 202, 196],
                [ 61,  66,  51],
                [  2, 128, 166],
                [206, 235, 224],
                [179, 218, 202],
                [219, 209, 175],
                [ 54, 205, 216],
                [204, 220, 192],
                [  1, 173, 196],
                [167, 208, 186]], dtype=uint8)
```

```
In [54]: 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()
```



Conclusion:

- **Image Segmentation** helps simplify complex images by dividing them into meaningful segments, making analysis easier.

- **K-Means Clustering** is an effective unsupervised method for segmenting images based on pixel similarity, such as color or intensity.

- **Choosing K** (number of clusters) plays a crucial role in segmentation quality:
  - **K = 5** offers a balance between simplicity and detail, good for broad segmentation.
  - **K = 20** provides more precise segmentation, capturing finer details but may introduce noise.

- **OpenCV's cv2.kmeans()** provides a flexible way to perform clustering, with options to specify stopping criteria, initial cluster center selection, and the number of attempts to improve results.

- K-Means clustering is a powerful tool for various image processing tasks, such as object detection, image enhancement, and pattern recognition, by simplifying the image into clusters that are easier to analyze.

Experiment 5

Aim: Harris Corner Detection in Computer Vision.

Objectives: The objective of this lab is to introduce the student to Image Segmentation using K Means algorithm.

1. Applying K-means algorithm on given image
2. Try different "K" values and observe the results

Harris Corner Detection

Concept

- Corners are points with significant intensity change in multiple directions.
- Flat regions and edges don't exhibit strong corner features.
- Shifting a window over corners results in large changes in intensity, measured by Sum of Squared Differences (SSD).

Change Function E(u,v)

- Measures intensity change after shifting a 3×3 window.
- High E(u,v) across all directions indicates a corner.

Harris Response R

- Calculated from eigenvalues of matrix M.
- R small → flat region
- R < 0 → edge
- R large → corner

Algorithm Overview

1. Convert image to grayscale.
2. Apply Gaussian blur.
3. Compute gradients using Sobel operator.
4. Calculate corner strength for each pixel.
5. Threshold and find local maxima.
6. Extract feature descriptors.

Importing Libraries:

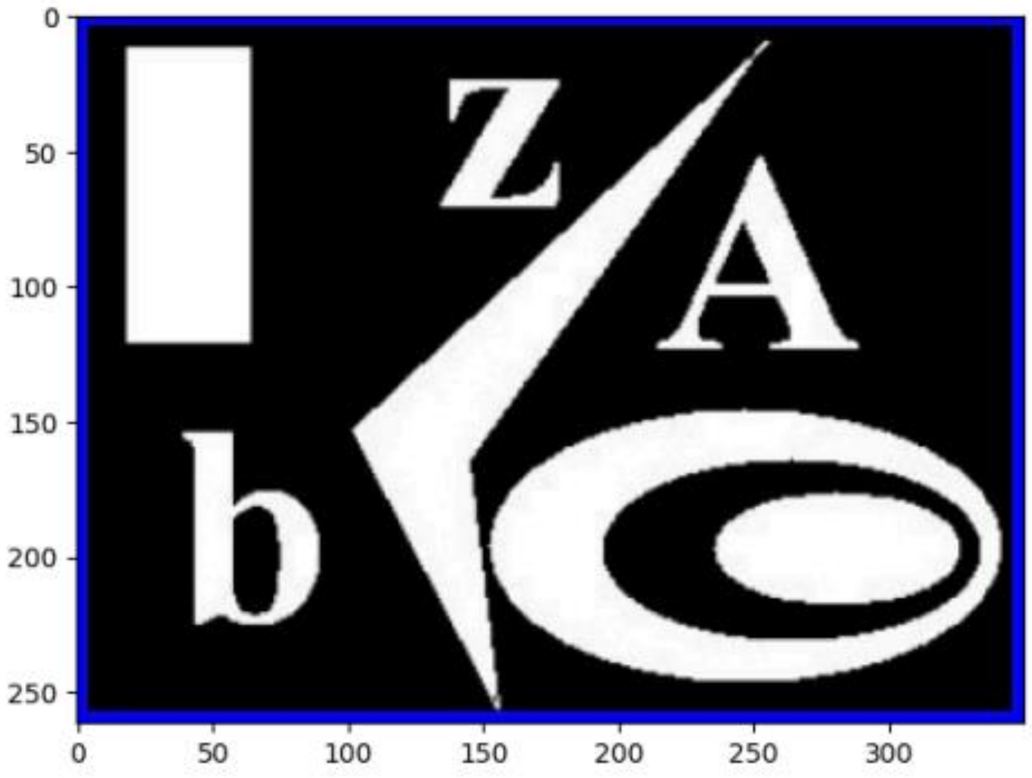
First, we need to import the required libraries for image processing and visualization.

```
In [14]: import cv2
         import numpy as np
         from matplotlib import pyplot as plt
```

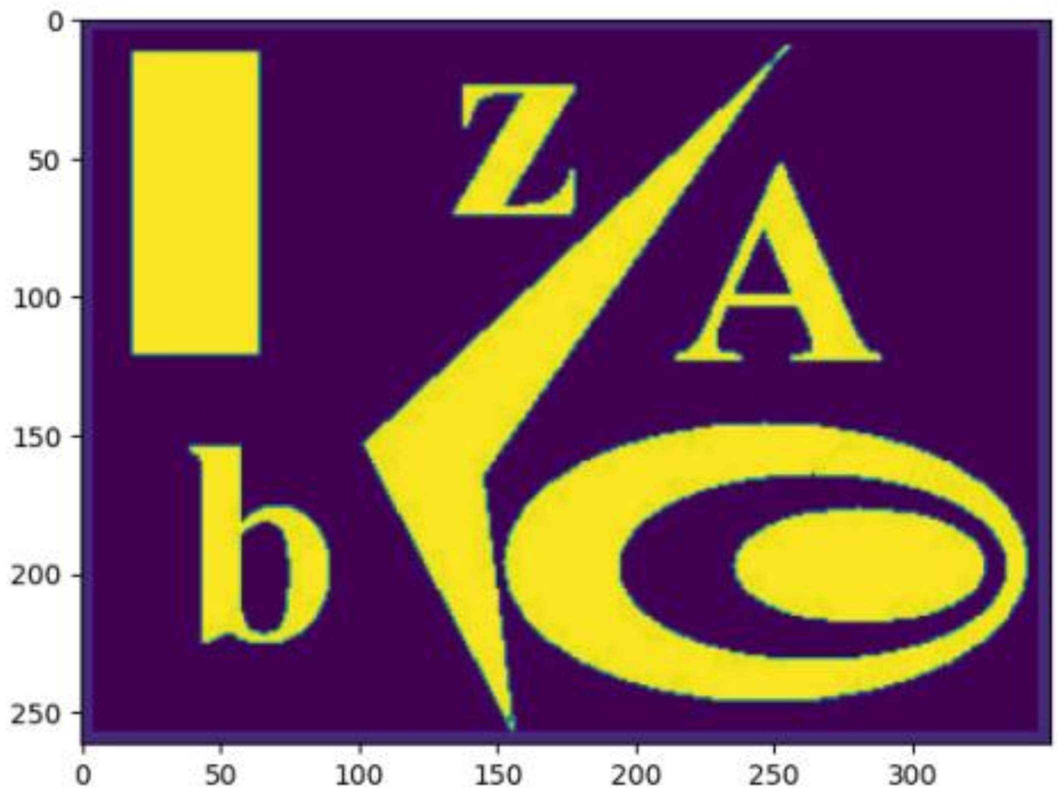
Read and Convert the Image

Read the input image in grayscale and convert it to float32 as required by the Harris detector.

```
In [63]: img = cv2.imread("lab4.jpg")
         plt.imshow(img)
         plt.show()
         print("Image shape(Color image have 3 channel)",img.shape)
         img1=cv2.cvtColor(img,cv2.COLOR_RGB2GRAY)
         img1=np.float32(img1)
         print("Image shape(Gray image have 1 channel)",img1.shape)
         plt.imshow(img1)
         plt.show()
```



```
Image shape(Color image have 3 channel) (262, 350, 3)
Image shape(Gray image have 1 channel) (262, 350)
```



```
In [61]: img.shape
```

```
Out[61]: (262, 350, 3)
```

```
In [64]: print(img1)
         sobel_y=np.array([[-1,-2,-1],[0,0,0],[1,2,1]])
         sobel_x=sobel_y.T
         print("sobel_x")
         print("-----------")
         print(sobel_x)

         print("sobel)y")
         print("-----------")
         print(sobel_y)
```

```
[[31. 29. 30. ... 28. 31. 32.]
 [29. 31. 32. ... 31. 28. 29.]
 [30. 30. 29. ... 33. 30. 32.]
 ...
 [33. 30. 30. ... 28. 30. 30.]
 [28. 30. 33. ... 35. 35. 30.]
 [31. 33. 28. ... 28. 30. 29.]]
sobel_x
-----------
[[-1  0  1]
 [-2  0  2]
 [-1  0  1]]
sobel)y
-----------
[[-1 -2 -1]
 [ 0  0  0]
 [ 1  2  1]]
```

```
In [65]: Ix=cv2.filter2D(img1,-1,sobel_x)
         Iy=cv2.filter2D(img1,-1,sobel_y)
```

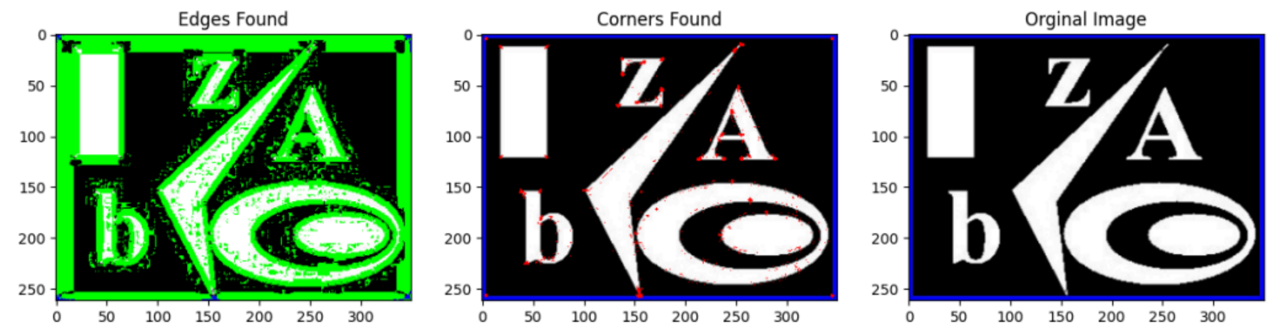
```
In [66]: Ixx=cv2.GaussianBlur(Ix*Ix,(3,3),1)
         Iyy=cv2.GaussianBlur(Iy*Iy,(3,3),1)
         Ixy=cv2.GaussianBlur(Ix*Iy,(3,3),1)
```

```
In [67]: detM=Ixx*Iyy-(Ixy)**2
         print("detM :- ",detM)
         traceM=Ixx+Iyy
         print("TraceM :- ",traceM)
```

```
detM :-  [[0.00000000e+00 1.44380844e+02 4.66365479e+02 ... 2.36957169e+02
  5.24325523e+01 1.49399700e+01]
 [1.26151009e+01 1.11913406e+02 1.17107507e+03 ... 3.39189331e+02
  5.11038208e+01 1.93026428e+01]
 [1.37646360e+01 2.38827835e+02 1.70911841e+03 ... 5.03761719e+02
  2.38540344e+02 3.41458130e+01]
 ...
 [1.21263695e+02 1.34151257e+03 3.73893604e+03 ... 1.77220762e+04
  1.26587197e+04 2.11445215e+03]
 [5.41316376e+01 8.57949677e+01 4.47593811e+02 ... 3.93589502e+03
  1.71731677e+03 7.52886658e+02]
 [1.34516201e+01 1.17623806e+01 1.00675411e+01 ... 1.86820129e+02
  1.19053474e+02 3.50532951e+01]]
TraceM :-  [[  8.770196   28.327122   48.734715  ...  41.254112   15.910633
    7.958201 ]
 [  8.348022   22.496367   75.162094  ...  46.558228   16.427422
    9.180918 ]
 [  7.9524317  39.790195  137.7602    ...  88.32341    41.235332
   16.105543 ]
 ...
 [ 24.12076    94.34854   175.8996    ... 297.18427   249.58383
  212.63113  ]
 [ 14.962745   21.887848   43.163063  ... 152.53043    91.19198
   56.27988  ]
 [ 10.962745   11.103361    9.844526  ... 159.84447    87.80141
   31.865152 ]]
```

```
In [68]: k=0.05
         harris_response=detM-k*(traceM)**2
```

```
In [69]: img_copyforedge=np.copy(img)
         img_copyforcorner=np.copy(img)
         for rowindex , response in enumerate(harris_response):
             for colindex , r in enumerate(response):
                 if r>100000:
                     img_copyforcorner[rowindex,colindex]=[255,0,0]
                 elif r<0:
                     img_copyforedge[rowindex,colindex]=[0,255,0]
         fig,ax=plt.subplots(nrows=1,ncols=3,figsize=(15,15))
         ax[0].set_title("Edges Found")
         ax[0].imshow(img_copyforedge)
         ax[1].set_title("Corners Found")
         ax[1].imshow(img_copyforcorner)
         ax[2].set_title("Orginal Image")
         ax[2].imshow(img)
         plt.show()
```

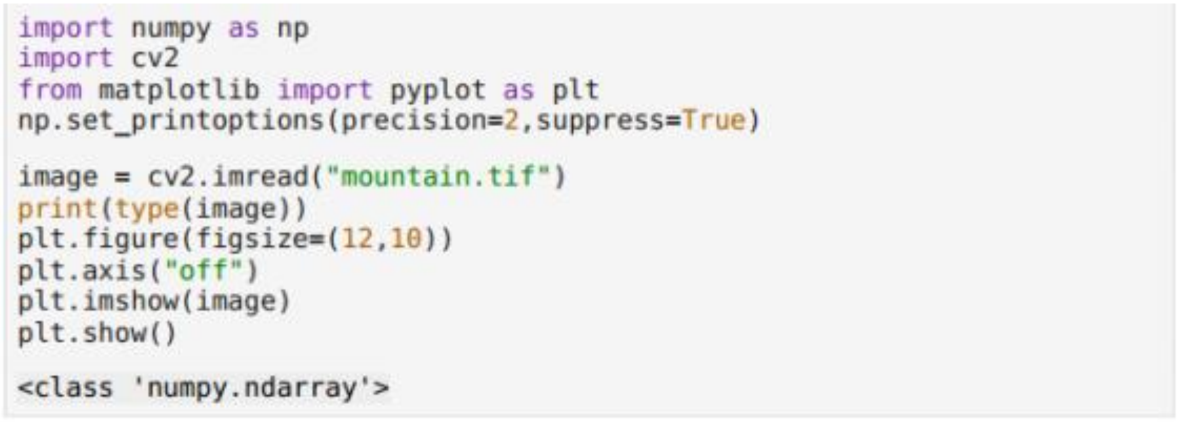


Conclusion:
Harris Corner Detection identifies points with strong intensity changes in all directions. It differentiates between flat regions, edges, and corners using gradient analysis. Corners are detected by computing a response function based on eigenvalues of a gradient matrix. It uses a threshold and local maxima check to finalize corner points. The method is efficient for feature detection in image analysis and computer vision tasks.

Experiment 6

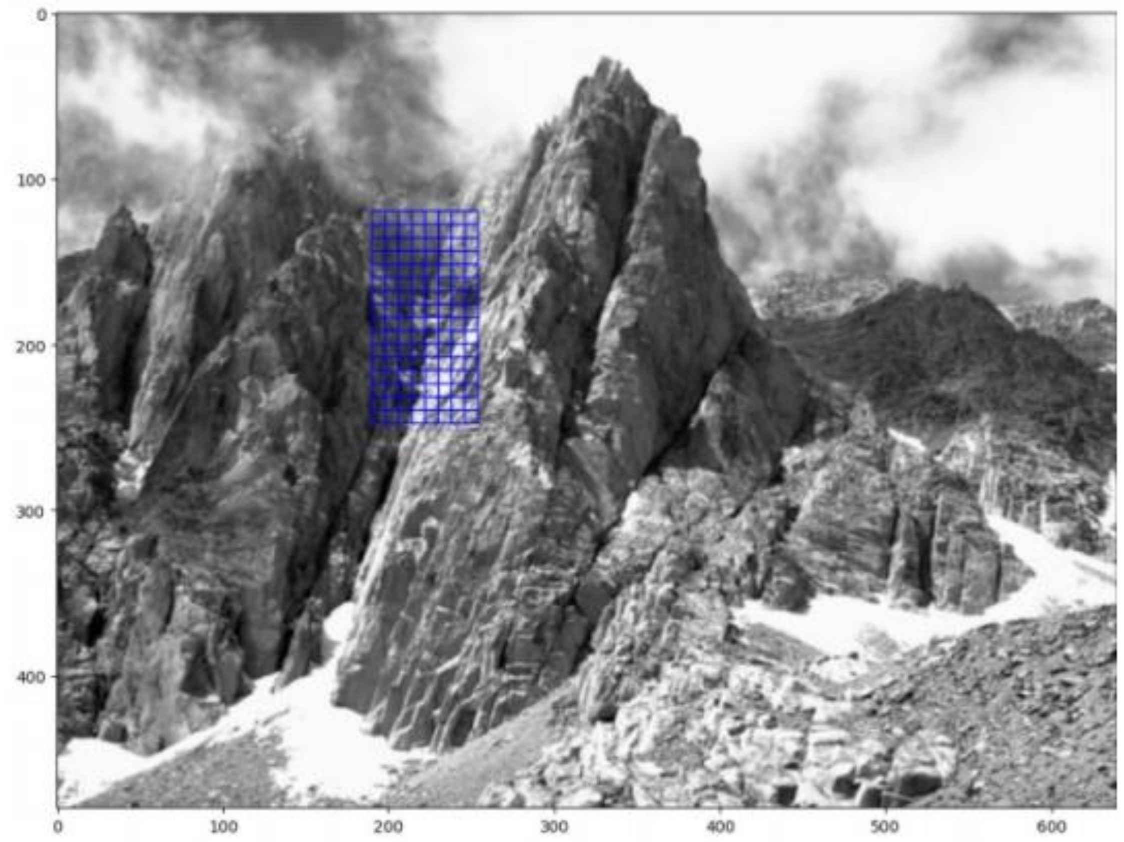
Aim: Object recognition using HOG and machine learning.

Objectives: To develop an object recognition system that utilizes Histogram of Oriented Gradients (HOG) for feature extraction and applies machine learning algorithms to accurately identify and classify objects in images.

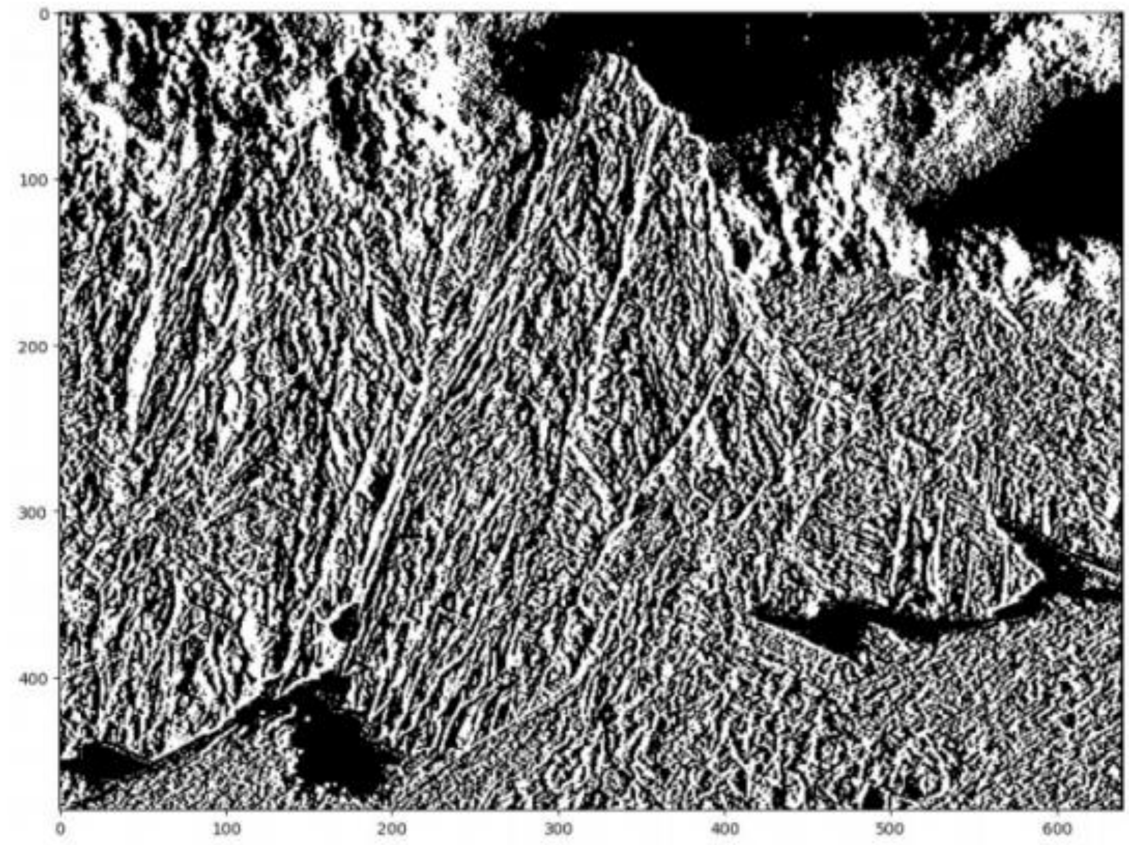


```
import numpy as np
import cv2
from matplotlib import pyplot as plt
np.set_printoptions(precision=2,suppress=True)

image = cv2.imread("mountain.tif")
print(type(image))
plt.figure(figsize=(12,10))
plt.axis("off")
plt.imshow(image)
plt.show()

<class 'numpy.ndarray'>
```

```
np_im=image.copy()
#Location=[0,0]
Location=[120,190]  # first component is vertical axis, second is
horizontal axis
PatchSize=[128,64]  # first component is vertical axis, second is
horizontal axis
numlinesY=int(PatchSize[0]/8)
numlinesX=int(PatchSize[1]/8)

#Draw frame around selected patch
plt.figure(figsize=(12,10))
cv2.rectangle(np_im, (Location[1], Location[0]),

(Location[1]+PatchSize[1],Location[0]+PatchSize[0]),
                     (0, 0, 255), 1)

#Draw region boundaries within the patch
for x in range(numlinesX):
    cv2.line(np_im,(Location[1]+8*(x+1),Location[0]),
                   (Location[1]+8*(x+1),Location[0]+PatchSize[0]),
                   (0, 0, 255), 1)
for y in range(numlinesY):
    cv2.line(np_im,(Location[1],Location[0]+8*(y+1)),
                   (Location[1]+PatchSize[1],Location[0]+8*(y+1)),
                   (0, 0, 255), 1)
plt.imshow(np_im)
plt.show()
```



```
np_im2=image.copy()
gx = cv2.Sobel(np_im2, cv2.CV_32F, 1, 0, ksize=1)
gy = cv2.Sobel(np_im2, cv2.CV_32F, 0, 1, ksize=1)

print(gx.shape)
print(gy.shape)

(480, 640, 3)
(480, 640, 3)

plt.figure(figsize=(12,10))
plt.imshow(gx,)
plt.show()

WARNING:matplotlib.image:Clipping input data to the valid range for
imshow with RGB data ([0..1] for floats or [0..255] for integers). Got
range [-204.0..239.0].
```

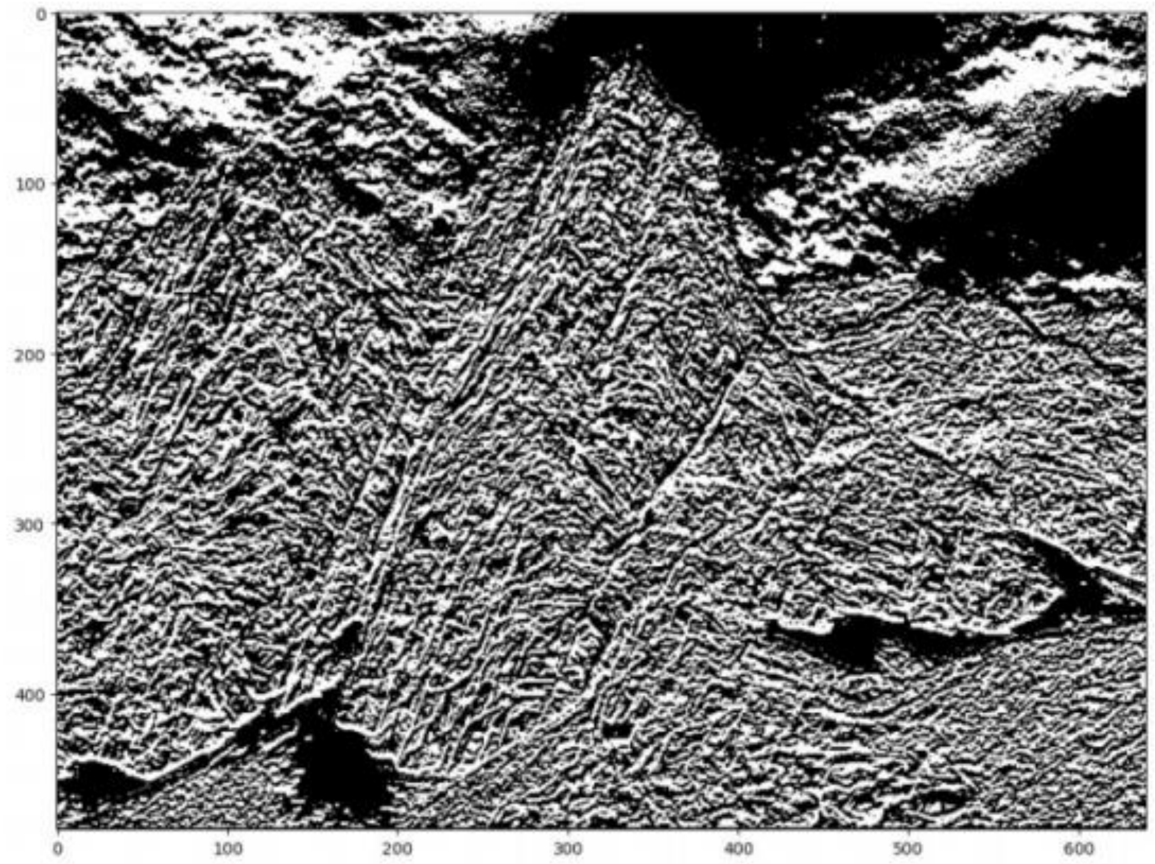


```
gx.shape
```

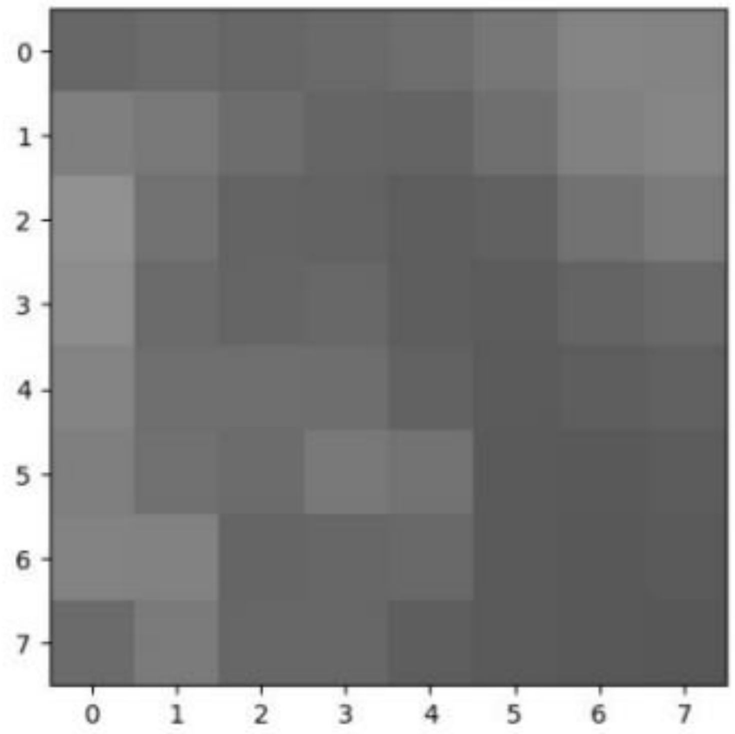
```
(480, 640, 3)
```

```
plt.figure(figsize=(12,10))
plt.imshow(gy)
plt.show()
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for
imshow with RGB data ([0..1] for floats or [0..255] for integers). Got
range [-199.0..230.0].
```



```
Chan=0
W=8

print(np_im2[Location[0]:Location[0]+W,Location[1]:Location[1]+W,Chan]
)

[[103 107 103 106 110 119 133 132]
 [127 121 109 101 100 111 129 134]
 [145 114 100  99  94  97 114 123]
 [141 107 101 104  94  92 100 105]
 [132 111 111 110  98  91  94  97]
 [127 113 108 121 115  90  89  92]
 [131 130 101 103 105  90  88  90]
 [107 123 103 103  94  90  88  87]]

plt.imshow(np_im2[Location[0]:Location[0]+W,Location[1]:Location[1]+W]
)
plt.show()
```



```
print(gx[Location[0]:Location[0]+W,Location[1]:Location[1]+W,Chan])

[[  9.   0.  -1.   7.  13.  23.  13.  13.]
 [ 23. -18. -20.  -9.  10.  29.  23.  11.]
 [ 10. -45. -15.  -6.  -2.  20.  26.   4.]
 [ -8. -40.  -3.  -7. -12.   6.  13.   9.]
 [-12. -21.  -1. -13. -19.  -4.   6.  11.]
 [-16. -19.   8.   7. -31. -26.   2.  11.]
 [ 20. -30. -27.   4. -13. -17.   0.  10.]
 [ 39.  -4. -20.  -9. -13.  -6.  -3.   8.]]

print(gy[Location[0]:Location[0]+W,Location[1]:Location[1]+W,Chan])

[[ 33.  25.   8. -14. -26. -15.  -3.   7.]
 [ 42.   7.  -3.  -7. -16. -22. -19.  -9.]
 [ 14. -14.  -8.   3.  -6. -19. -29. -29.]
 [-13.  -3.  11.  11.   4.  -6. -20. -26.]
 [-14.   6.   7.  17.  21.  -2. -11. -13.]
 [ -1.  19. -10.  -7.   7.  -1.  -6.  -7.]
 [-20.  10.  -5. -18. -21.   0.  -1.  -5.]
 [-48. -43. -15.  -6. -15.  -5.  -5.  -8.]]

mag, angle = cv2.cartToPolar(gx, gy, angleInDegrees=True)
print(mag.shape)
print(angle.shape)
```

```
(480, 640, 3)
(480, 640, 3)
```

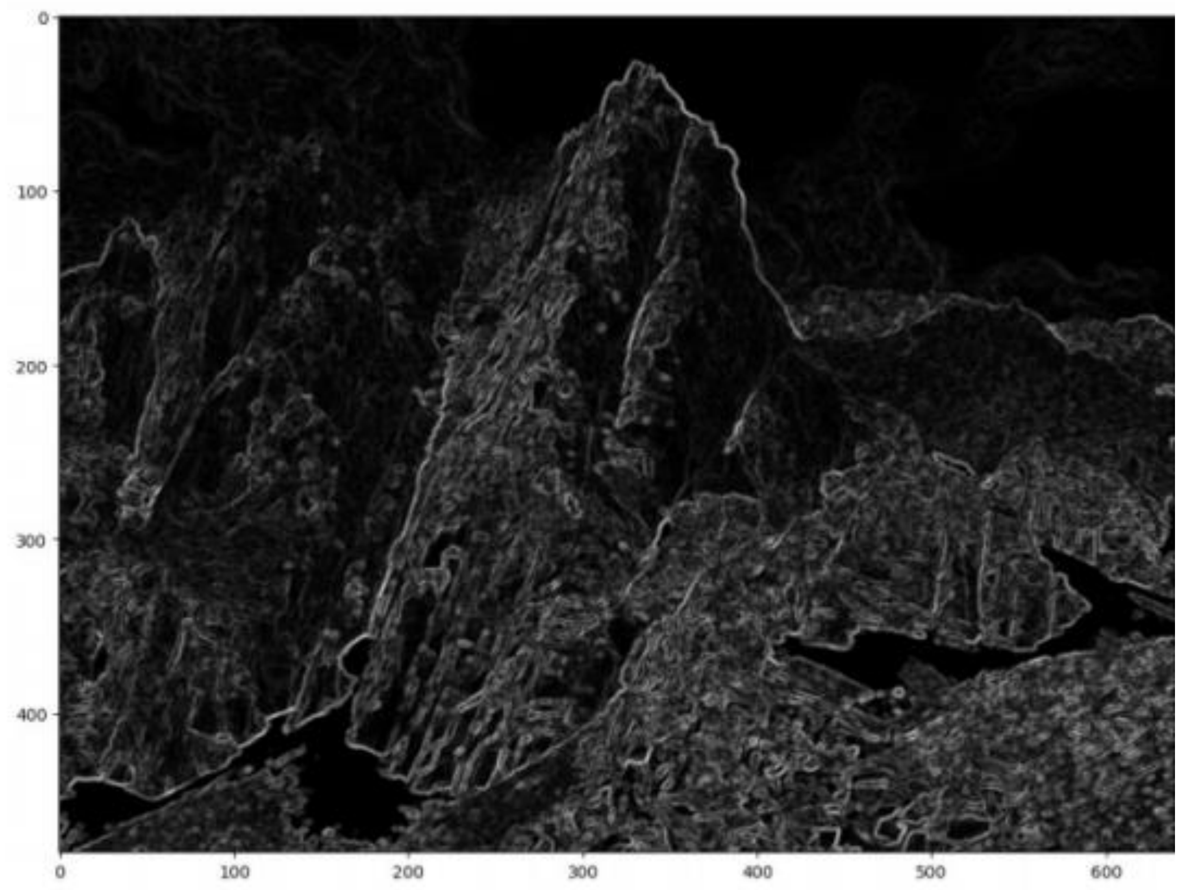
```
mag=mag.astype(int)
angle=angle.astype(int)
```

```
print(mag[Location[0]:Location[0]+W,Location[1]:Location[1]+W,Chan])
```

```
[[34 25  8 15 29 27 13 14]
 [47 19 20 11 18 36 29 14]
 [17 47 17  6  6 27 38 29]
 [15 40 11 13 12  8 23 27]
 [18 21  7 21 28  4 12 17]
 [16 26 12  9 31 26  6 13]
 [28 31 27 18 24 17  1 11]
 [61 43 25 10 19  7  5 11]]
```

```
plt.figure(figsize=(12,10))
plt.imshow(mag)
plt.show()
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for
imshow with RGB data ([0..1] for floats or [0..255] for integers). Got
range [0..261].
```



```
mag.shape
```

```
(480, 640, 3)
```

```
maxChan=np.argmax(mag,axis=2)
print(maxChan.shape)
```

```
(480, 640)
```

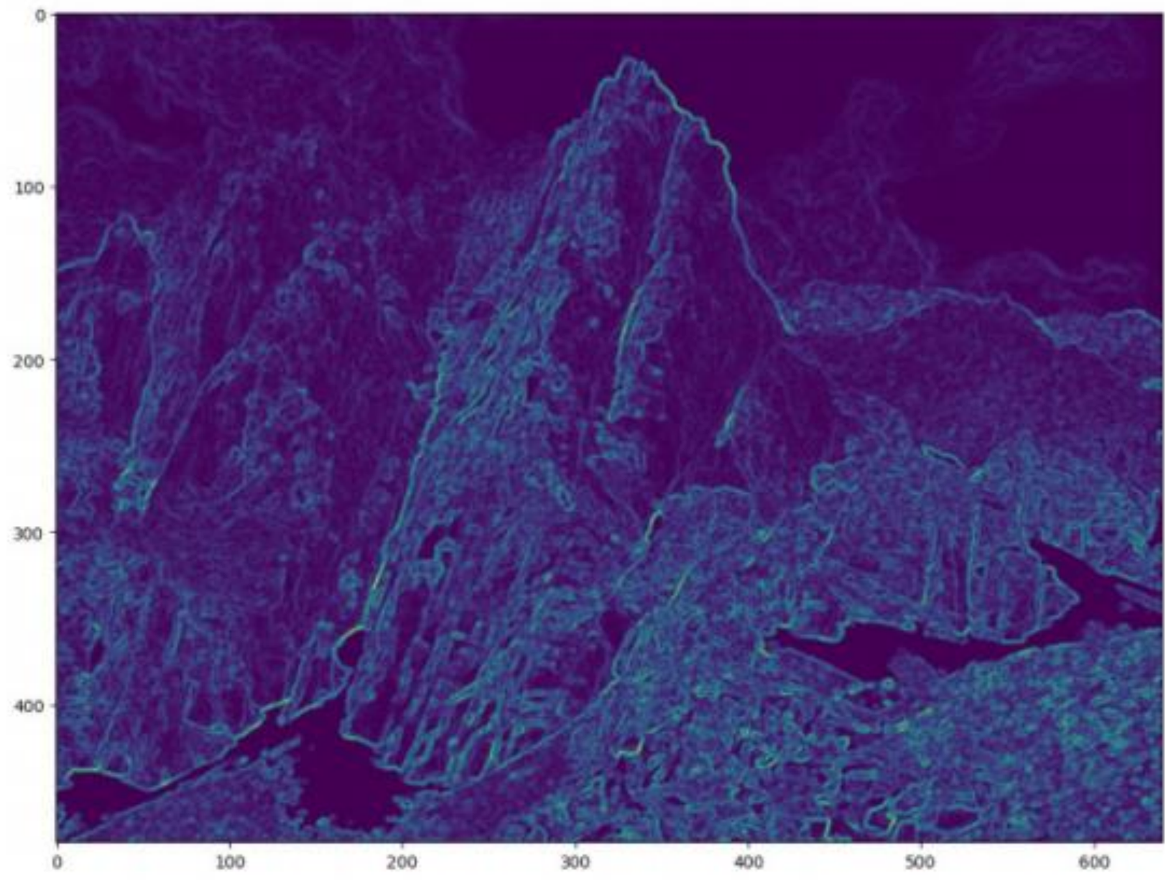
```
print(maxChan[Location[0]:Location[0]+W,Location[1]:Location[1]+W])
```

```
[[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 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]]
```

```
maxmag=np.zeros(maxChan.shape)
for r in range(maxChan.shape[0]):
    for c in range(maxChan.shape[1]):
        maxmag[r,c]=mag[r,c,maxChan[r,c]]
print(maxmag.shape)
```

```
(480, 640)
```

```
plt.figure(figsize=(12,10))
plt.imshow(maxmag)
plt.show()
```



```
maxangle=np.zeros(maxChan.shape)
for r in range(maxChan.shape[0]):
    for c in range(maxChan.shape[1]):
        maxangle[r,c]=angle[r,c,maxChan[r,c]]
print(maxangle.shape)
```

```
(480, 640)
```

```
print(maxmag[Location[0]:Location[0]+W,Location[1]:Location[1]+W])
```

```
[[34. 25.  8. 15. 29. 27. 13. 14.]
 [47. 19. 20. 11. 18. 36. 29. 14.]
 [17. 47. 17.  6.  6. 27. 38. 29.]
 [15. 40. 11. 13. 12.  8. 23. 27.]
 [18. 21.  7. 21. 28.  4. 12. 17.]
 [16. 26. 12.  9. 31. 26.  6. 13.]
 [28. 31. 27. 18. 24. 17.  1. 11.]
 [61. 43. 25. 10. 19.  7.  5. 11.]]
```

```
print(maxangle[Location[0]:Location[0]+W,Location[1]:Location[1]+W])
```

```
[[ 74.  90.  97. 296. 296. 326. 347.  28.]
 [ 61. 158. 188. 217. 302. 322. 320. 320.]
 [ 54. 197. 208. 153. 251. 316. 311. 277.]
 [238. 184. 105. 122. 161. 315. 303. 289.]
 [229. 164.  98. 127. 132. 206. 298. 310.]
 [183. 135. 308. 315. 167. 182. 288. 327.]
 [315. 161. 190. 282. 238. 180. 270. 333.]
 [309. 264. 216. 213. 229. 219. 239. 315.]]
```

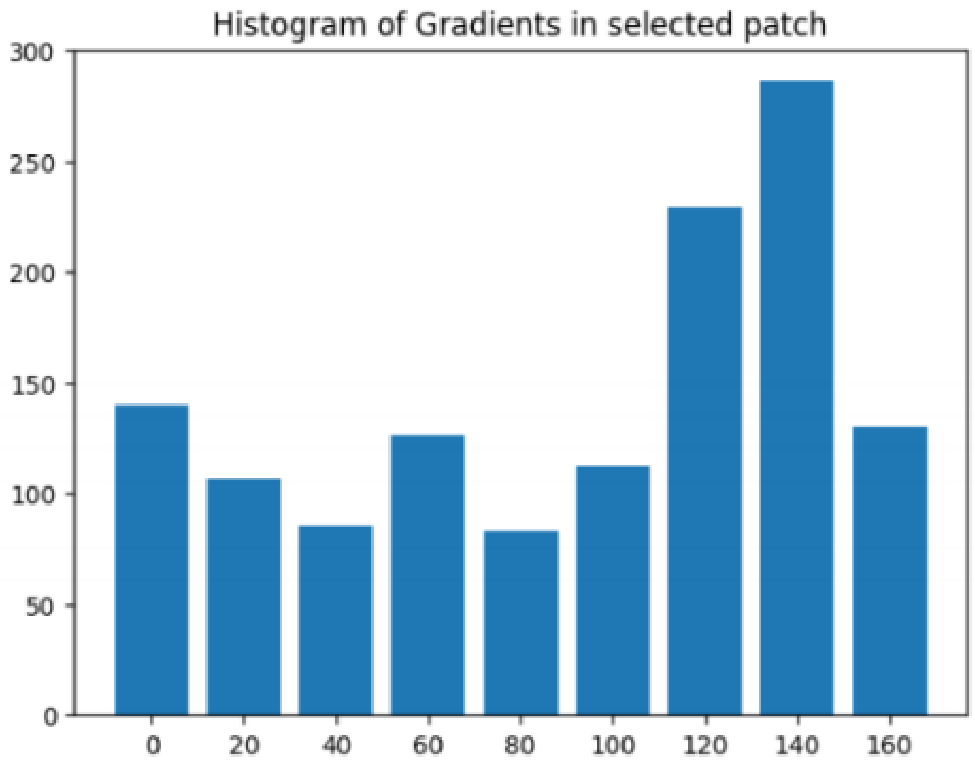
```
def anglemapper(x):
    if x >=180:
        return x-180
    else:
        return x

vfunc = np.vectorize(anglemapper)
mappedAngles=(vfunc(maxangle))
print(mappedAngles[Location[0]:Location[0]+W,Location[1]:Location[1]+W
])
```

```
[[ 74.  90.  97. 116. 116. 146. 167.  28.]
 [ 61. 158.   8.  37. 122. 142. 140. 140.]
 [ 54.  17.  28. 153.  71. 136. 131.  97.]
 [ 58.   4. 105. 122. 161. 135. 123. 109.]
 [ 49. 164.  98. 127. 132.  26. 118. 130.]
 [  3. 135. 128. 135. 167.   2. 108. 147.]
 [135. 161.  10. 102.  58.   0.  90. 153.]
 [129.  84.  36.  33.  49.  39.  59. 135.]]
```

```
def createHist(AngArray,MagArray,BS=20,BINS=9):
    hist=np.zeros(BINS)
    for r in range(AngArray.shape[0]):
        for c in range(AngArray.shape[1]):
            #print(AngArray[r,c])
            binel,rem = np.divmod(AngArray[r,c],BS)
            weightR=rem*1.0/BS
            weightL=1-weightR
            deltaR=MagArray[r,c]*weightR
            deltaL=MagArray[r,c]*weightL
            binL=int(binel)
            binR=np.mod(binL+1,BINS)
            hist[binL]+=deltaL
            hist[binR]+=deltaR
    return hist

spotAngles=mappedAngles[Location[0]:Location[0]+W,Location[1]:Location
[1]+W]
spotMag=maxmag[Location[0]:Location[0]+W,Location[1]:Location[1]+W]
spotHist=createHist(spotAngles,spotMag)
print('Gradient histogram of the selected region:')
print(spotHist)
```

```
Gradient histogram of the selected region:
[140.3  106.35  85.7  125.95  83.1  112.25 229.65 286.05 130.65]
```

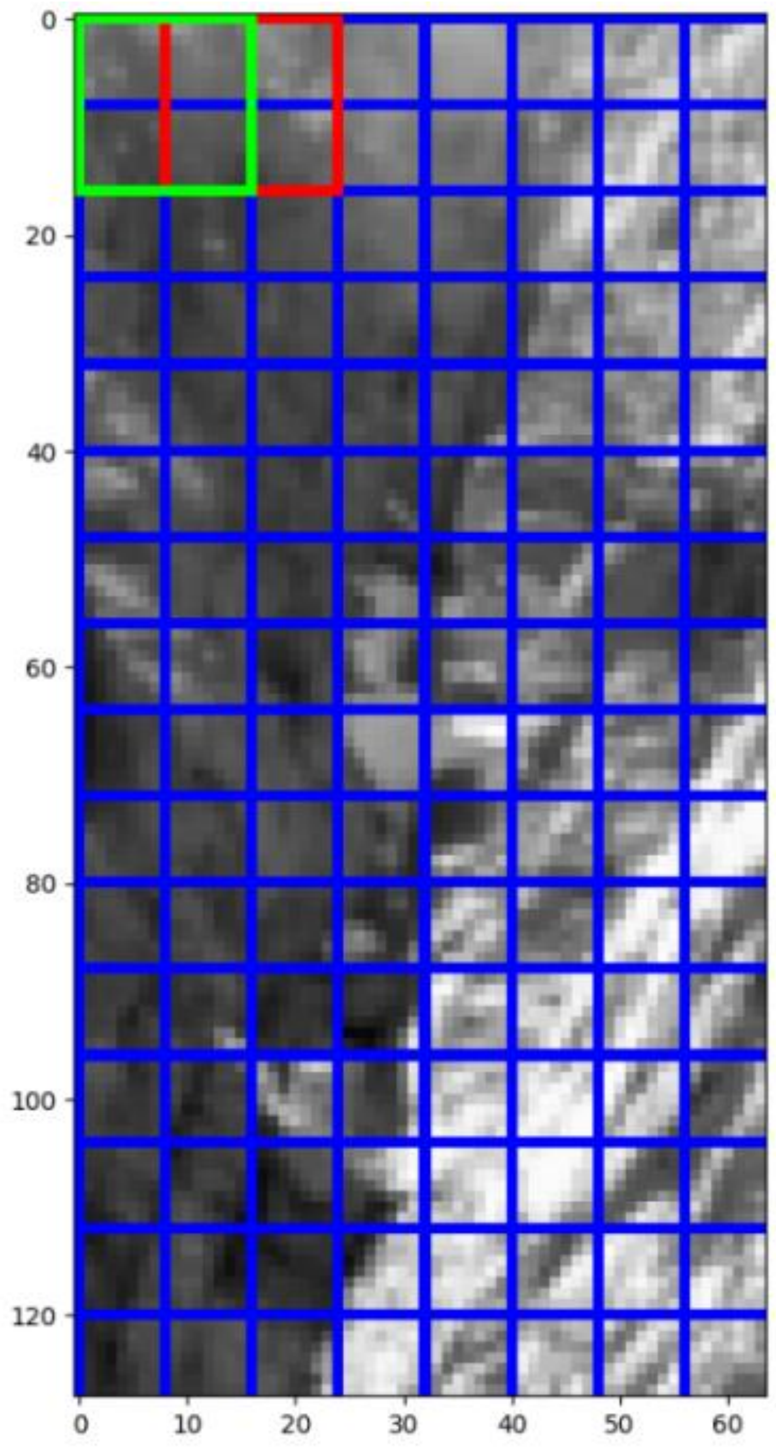
```
plt.bar(range(9),spotHist)
plt.xticks(range(9),[0,20,40,60,80,100,120,140,160])
plt.title("Histogram of Gradients in selected patch")
plt.show()
```



Normalization

```
patch =
np_im[Location[0]:Location[0]+PatchSize[0],Location[1]:Location[1]+Pat
```

```
chSize[1]].copy()
SR=16
plt.figure(figsize=(14,10))
cv2.rectangle(patch, (W, 0), (W+SR,SR),(255, 0, 0), 1)
cv2.rectangle(patch, (0, 0), (SR,SR),(0, 255, 0), 1)
plt.imshow(patch)
plt.show()
```

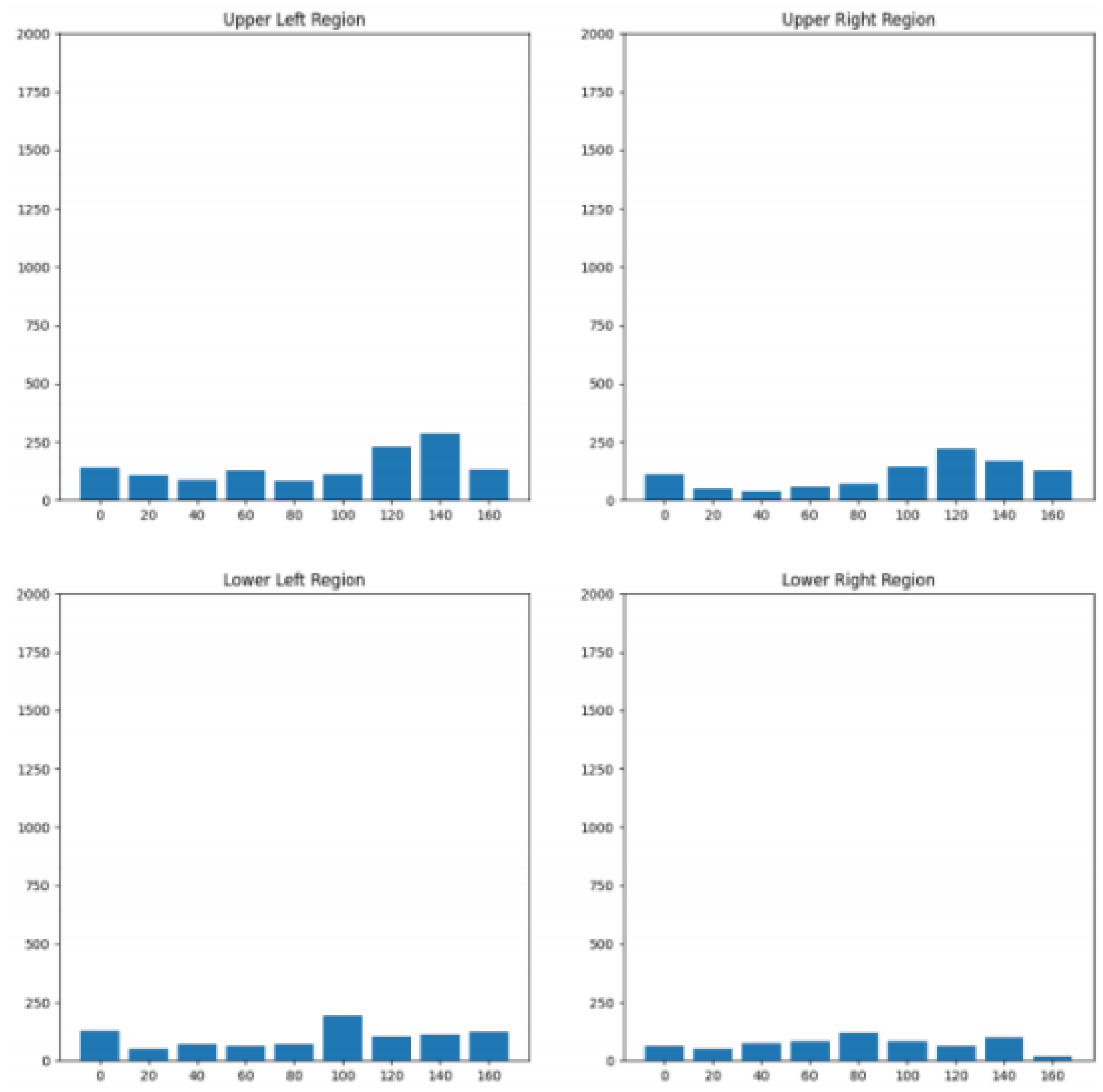


```
hist11=spotHist
print(hist11)
```

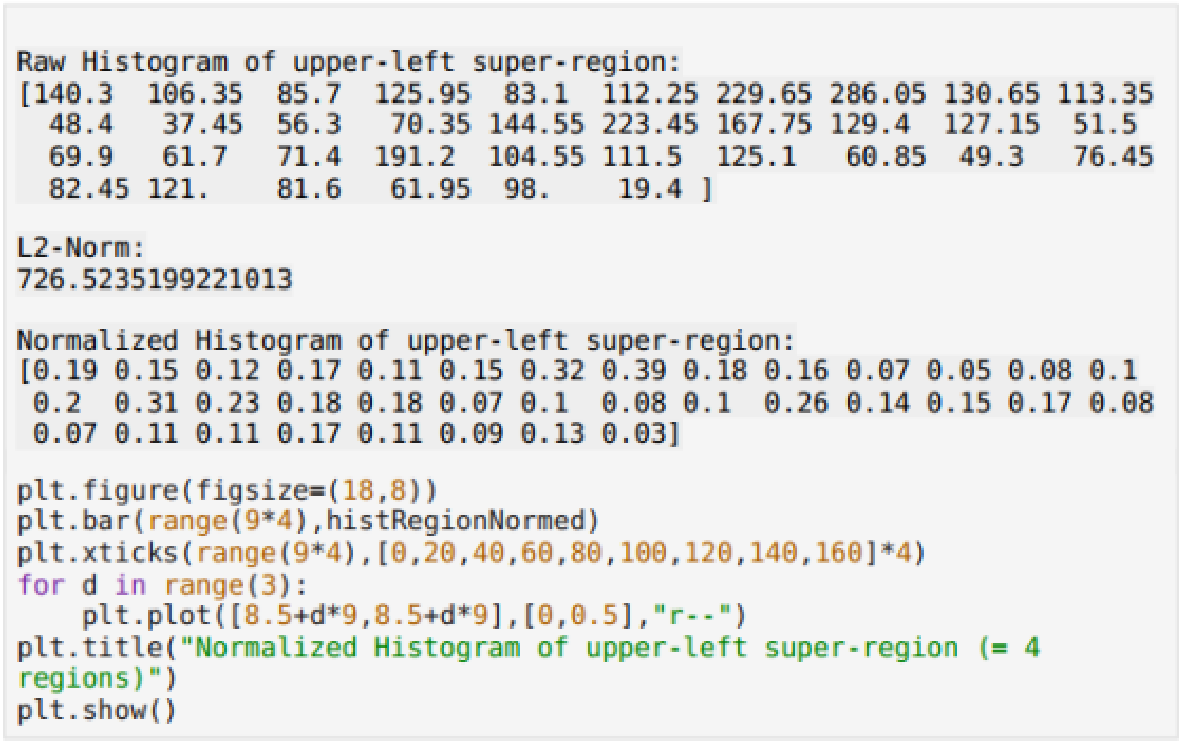
```
spotAngles=mappedAngles[Location[0]:Location[0]+W,Location[1]+W:Locati
on[1]+2*W]
spotMag=maxmag[Location[0]:Location[0]+W,Location[1]+W:Location[1]+2*W
]
hist12=createHist(spotAngles,spotMag)
print(hist12)
spotAngles=mappedAngles[Location[0]+W:Location[0]+2*W,Location[1]:Loca
tion[1]+W]
spotMag=maxmag[Location[0]+W:Location[0]+2*W,Location[1]:Location[1]+W
]
hist21=createHist(spotAngles,spotMag)
print(hist21)
spotAngles=mappedAngles[Location[0]+W:Location[0]+2*W,Location[1]+W:Lo
cation[1]+2*W]
spotMag=maxmag[Location[0]+W:Location[0]+2*W,Location[1]+W:Location[1]
+2*W]
hist22=createHist(spotAngles,spotMag)
print(hist22)
```

```
[140.3  106.35  85.7  125.95  83.1  112.25 229.65 286.05 130.65]
[113.35  48.4   37.45  56.3   70.35 144.55 223.45 167.75 129.4 ]
[127.15  51.5   69.9   61.7   71.4  191.2  104.55 111.5  125.1 ]
[ 60.85  49.3   76.45  82.45 121.    81.6   61.95  98.    19.4 ]
```

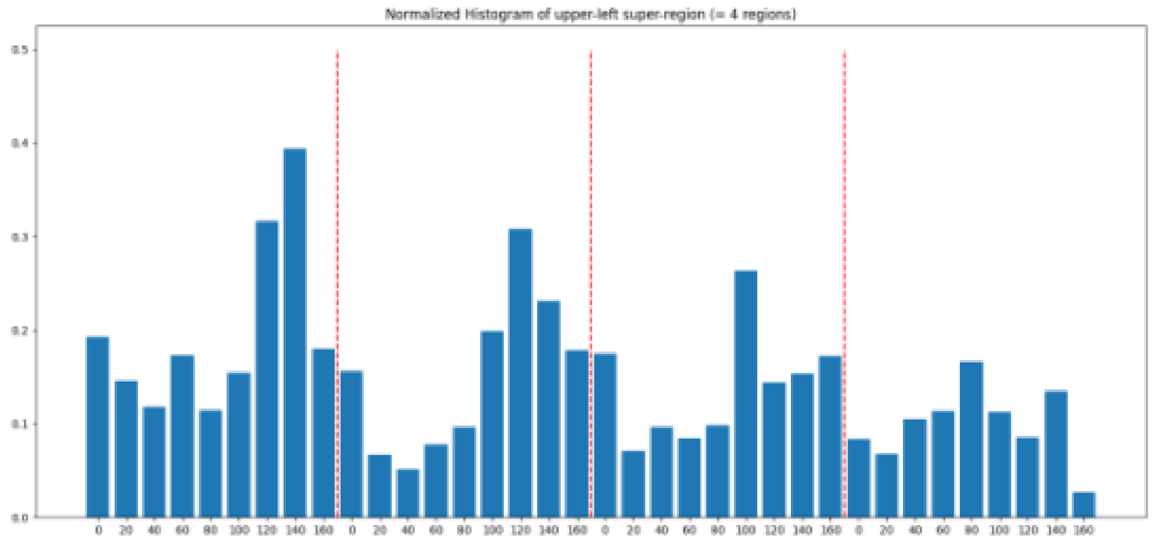
```
histList=[hist11,hist12,hist21,hist22]
titles=["Upper Left","Upper Right","Lower Left","Lower Right"]
plt.figure(figsize=(14,14))
i=1
for h in histList:
    plt.subplot(2,2,i)
    plt.title(titles[i-1]+" Region")
    plt.bar(range(9),h)
    plt.xticks(range(9),[0,20,40,60,80,100,120,140,160])
    plt.ylim((0,2000))
    i+=1
    #plt.title("Histogram of Gradients in selected patch")
plt.show()
```



```
histRegion=np.array([bin for hist in histList for bin in hist])
print("\nRaw Histogram of upper-left super-region:")
print(histRegion)

l2norm=np.sqrt(np.sum(histRegion**2))
print("\nL2-Norm:")
print(l2norm)
epsilon=1e-6 # define epsilon in order to prevent division by zero
histRegionNormed=histRegion/(l2norm+epsilon)
print("\nNormalized Histogram of upper-left super-region:")
print(histRegionNormed)
```



```
Raw Histogram of upper-left super-region:
[140.3  106.35  85.7  125.95  83.1  112.25 229.65 286.05 130.65 113.35
  48.4   37.45  56.3   70.35 144.55 223.45 167.75 129.4  127.15  51.5
  69.9   61.7   71.4  191.2  104.55 111.5  125.1   60.85  49.3   76.45
  82.45 121.    81.6   61.95  98.    19.4 ]

L2-Norm:
726.5235199221013

Normalized Histogram of upper-left super-region:
[0.19 0.15 0.12 0.17 0.11 0.15 0.32 0.39 0.18 0.16 0.07 0.05 0.08 0.1
 0.2  0.31 0.23 0.18 0.18 0.07 0.1  0.08 0.1  0.26 0.14 0.15 0.17 0.08
 0.07 0.11 0.11 0.17 0.11 0.09 0.13 0.03]

plt.figure(figsize=(18,8))
plt.bar(range(9*4),histRegionNormed)
plt.xticks(range(9*4),[0,20,40,60,80,100,120,140,160]*4)
for d in range(3):
    plt.plot([8.5+d*9,8.5+d*9],[0,0.5],"r--")
plt.title("Normalized Histogram of upper-left super-region (= 4
regions)")
plt.show()
```



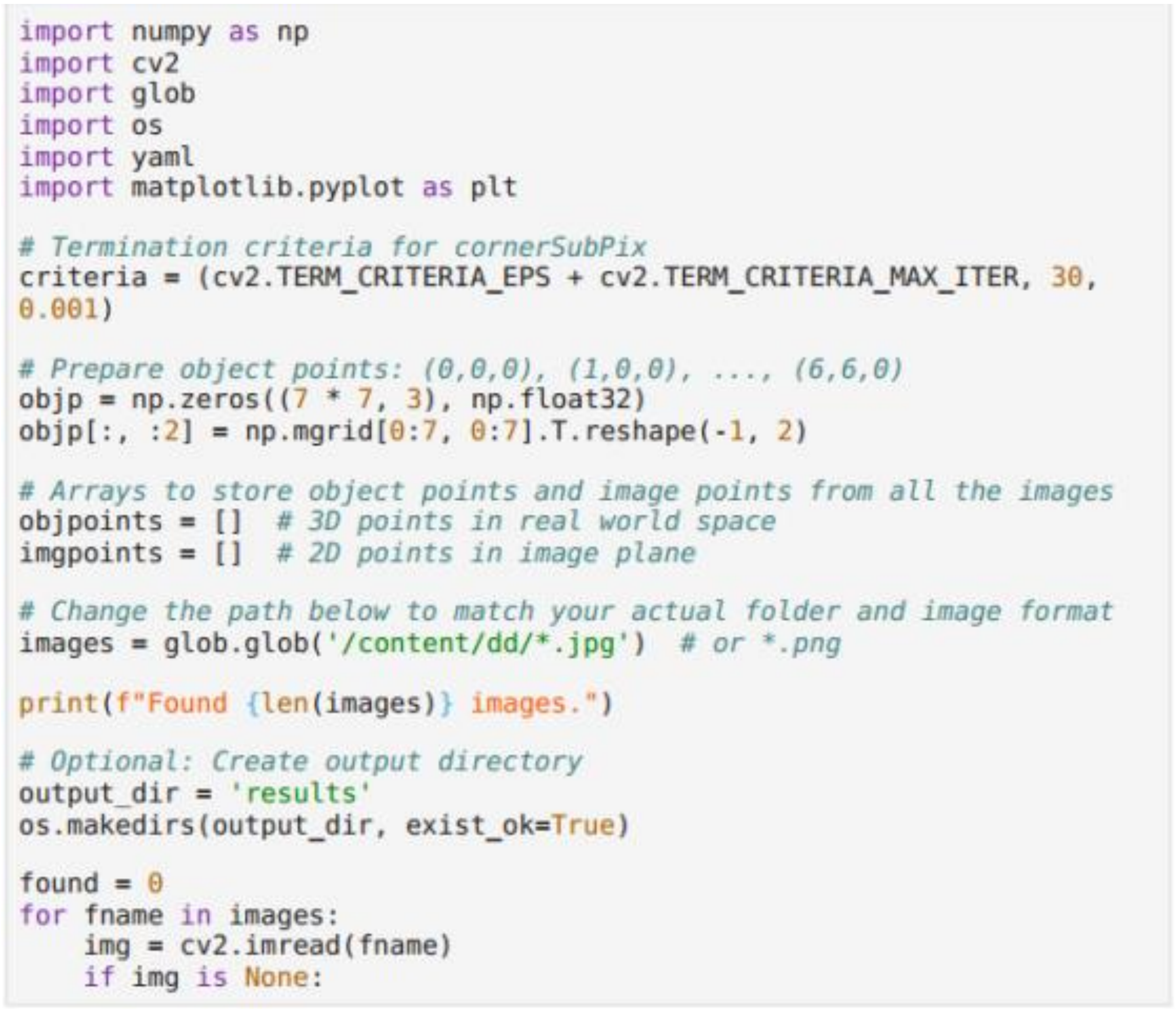
Conclusion on Histogram of Oriented Gradients (HOG):
Histogram of Oriented Gradients (HOG) is a powerful feature descriptor used primarily in object detection tasks, especially for detecting humans and other objects with well-defined edges and shapes. By computing the distribution of gradient orientations in localized regions of an image, HOG captures the structural and edge information while being invariant to geometric and photometric transformations (except object orientation). Its robustness, simplicity, and effectiveness make it a foundational technique in classical computer vision, often used in conjunction with classifiers like Support Vector Machines (SVMs) before the rise of deep learning.

Experiment 7

Aim: Camera Calibration Using OpenCV and Python

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

1. Load and process multiple images containing a 7x7 chessboard pattern.
2. Detect and refine chessboard corners in each image to extract accurate 2D image points.
3. Perform camera calibration to compute the camera matrix and distortion coefficients.
4. Save the calibration results to a YAML file for future use in correcting image distortion.



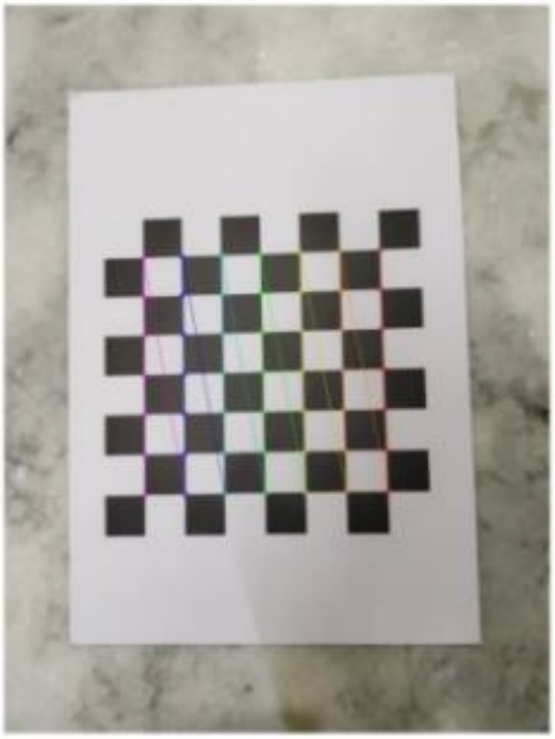
```
import numpy as np
import cv2
import glob
import os
import yaml
import matplotlib.pyplot as plt

# Termination criteria for cornerSubPix
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30,
0.001)

# Prepare object points: (0,0,0), (1,0,0), ..., (6,6,0)
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 from all the images
objpoints = []  # 3D points in real world space
imgpoints = []  # 2D points in image plane

# Change the path below to match your actual folder and image format
images = glob.glob('/content/dd/*.jpg')  # or *.png

print(f"Found {len(images)} images.")

# Optional: Create output directory
output_dir = 'results'
os.makedirs(output_dir, exist_ok=True)

found = 0
for fname in images:
    img = cv2.imread(fname)
    if img is None:
        print(f"Failed to load image: {fname}")
        continue

    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Find the chess board corners
    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)

        # Draw and show the corners using matplotlib (safe for Colab)
        img_display = cv2.drawChessboardCorners(img.copy(), (7, 7),
corners2, ret)
        plt.imshow(cv2.cvtColor(img_display, cv2.COLOR_BGR2RGB))
        plt.title(f'Chessboard Detection #{found+1}')
        plt.axis('off')
        plt.show()

        # Optional: Save images with detected corners
        out_path = os.path.join(output_dir,
f'calibresult_{found+1}.png')
        cv2.imwrite(out_path, img_display)

        found += 1
    else:
        print(f"Chessboard not found in: {fname}")

print("Number of valid images used for calibration:", found)

# Check if at least one pattern was found
if found == 0:
    print("No valid chessboard patterns were found. Exiting.")
    exit()

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

# Print calibration results
print("Camera matrix:\n", mtx)
print("Distortion coefficients:\n", dist)

# Save to YAML file
calibration_data = {
    'camera_matrix': mtx.tolist(),
    'dist_coeff': dist.tolist()
```

```
}

with open("calibration_matrix.yaml", "w") as f:
    yaml.dump(calibration_data, f)

print("Calibration data saved to calibration_matrix.yaml")
```

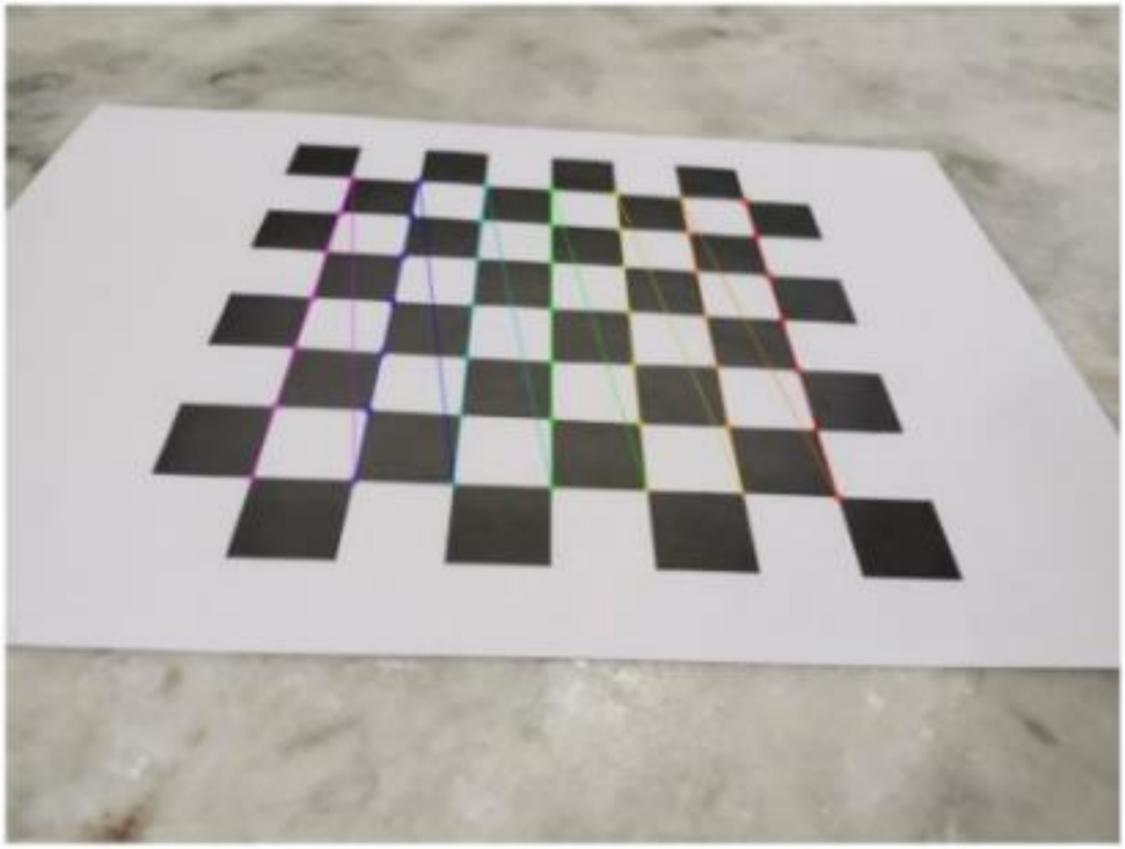
```
Found 3 images.
```

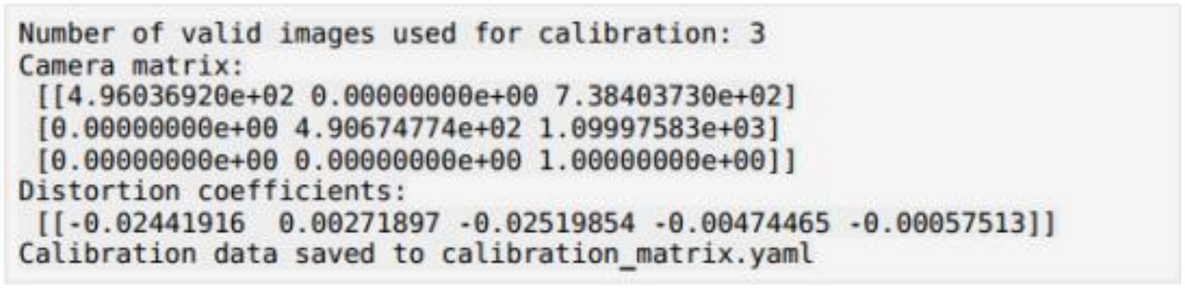
Chessboard Detection #1

Chessboard Detection #3



Chessboard Detection #2





```
Number of valid images used for calibration: 3
Camera matrix:
 [[4.96036920e+02 0.00000000e+00 7.38403730e+02]
 [0.00000000e+00 4.90674774e+02 1.09997583e+03]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
Distortion coefficients:
 [[-0.02441916  0.00271897 -0.02519854 -0.00474465 -0.00057513]]
Calibration data saved to calibration_matrix.yaml
```

Conclusion on Camera Calibration:
Camera calibration is a fundamental process in computer vision that enables accurate mapping between 3D real-world coordinates and 2D image coordinates. By estimating the intrinsic parameters (such as focal length, optical center, and skew) and extrinsic parameters (rotation and translation vectors), calibration corrects lens distortion and improves the precision of measurements and object localization in an image. This process is essential for applications like 3D reconstruction, augmented reality, robotics, and object tracking. Proper calibration ensures that computer vision systems can interpret visual data with geometric accuracy and consistency.