

# Index

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

**Computer Vision****Name: Rohan Saini****Registration No: 04419051622****Lab - 1 : 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

## ✓ Reading and displaying an image in python

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

```
image = cv2.imread('input1.jpg')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

```
plt.figure(figsize=(6,6))
plt.imshow(image)
plt.title('Original Image')
plt.axis('off')
plt.show()
```



Original Image



## ✓ Convert Images to another format and compare them

```
import cv2
import matplotlib.pyplot as plt
from PIL import Image
import os

# Read original image (JPG)
image = cv2.cvtColor(cv2.imread('input1.jpg'), cv2.COLOR_BGR2RGB)

# Save in 3 formats with comparable quality settings
cv2.imwrite('output.jpg', cv2.cvtColor(image, cv2.COLOR_RGB2BGR), [int(cv2.IMWRITE_JPEG_QUALITY), 90]) # JPG at 90% quality
cv2.imwrite('output.png', cv2.cvtColor(image, cv2.COLOR_RGB2BGR)) # PNG (lossless)
cv2.imwrite('output.webp', cv2.cvtColor(image, cv2.COLOR_RGB2BGR), [int(cv2.IMWRITE_WEBP_QUALITY), 90]) # WEBP at 90% quality

# Get file sizes
file_sizes = {
    'JPG': f"{os.path.getsize('output.jpg') / 1024:.1f} KB",
    'PNG': f"{os.path.getsize('output.png') / 1024:.1f} KB",
```

```

'WEBP': f"{os.path.getsize('output.webp') / 1024:.1f} KB"
}

# Display all 3 formats + file sizes
fig, ax = plt.subplots(1, 3, figsize=(15, 5))
titles = ['JPG (Lossy)', 'PNG (Lossless)', 'WEBP (Modern)']
formats = ['output.jpg', 'output.png', 'output.webp']

for i, (title, img_path) in enumerate(zip(titles, formats)):
    img = cv2.cvtColor(cv2.imread(img_path), cv2.COLOR_BGR2RGB)
    ax[i].imshow(img)
    ax[i].set_title(f"{title}\nSize: {file_sizes[titles[i].split()[0]]}")
    ax[i].axis('off')

plt.tight_layout()
plt.show()

```



## ✓ Convert an Image to Grayscale and display

```

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

# Display
plt.figure(figsize=(6,6))
plt.imshow(gray_image, cmap='gray')
plt.title('Grayscale Image')
plt.axis('off')
plt.show()

```



## ✓ Perform Image enhancement operations

Here students will explain need and type of image enhancement methods

```

import numpy as np
# Enhancement examples
# a) Histogram Equalization
equalized = cv2.equalizeHist(gray_image)

```

```
# b) Sharpening
kernel = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])
sharpened = cv2.filter2D(image, -1, kernel)

# Display enhancements
fig, ax = plt.subplots(1, 3, figsize=(18,6))
ax[0].imshow(gray_image, cmap='gray')
ax[0].set_title('Original Grayscale')
ax[0].axis('off')

ax[1].imshow(equalized, cmap='gray')
ax[1].set_title('Histogram Equalized')
ax[1].axis('off')

ax[2].imshow(sharpened)
ax[2].set_title('Sharpened Color')
ax[2].axis('off')
plt.show()
```



## ✓ Conclusion

Through this lab, we learned to:

1. Read and display images using OpenCV/matplotlib
2. Convert between image formats and compare quality
3. Create grayscale versions of images
4. Apply essential enhancement techniques to improve image quality
5. Understand the importance of preprocessing in computer vision pipelines



## LAB 2 :- Image Enhancement using Spatial Domain Filtering

### Objectives:

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

1. To Understand convolution operation in images
2. To Apply spatial filters to images
3. Average Filter and its application in noise reduction

### Reading and displaying an image in python

```
import matplotlib.pyplot as plt
import matplotlib.image as img
image=img.imread('trees_gray.jpg')
```

```
plt.figure(figsize=(5,3))
plt.imshow(image)
plt.axis('off')
plt.show()
```



### Averaging/smoothing mask on image

```
import numpy as np
mask_size=3
mask=np.ones((mask_size,mask_size))/mask_size**2
print(mask)
```

□

```
[[0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]
 [0.11111111 0.11111111 0.11111111]]
```



**Averaging mask for noise removal**

```

image_height, image_width = image.shape
filter_image = np.zeros((image_height, image_width))

pad = mask_size // 2
padded_image = np.pad(image, pad, mode='constant', constant_values=0)
# Apply averaging filter
for i in range(image_height):
    for j in range(image_width):
        sum = 0
        for k in range(mask_size):
            for l in range(mask_size):
                roi = padded_image[i + k, j + l]
                sum += roi * mask[k, l]
            filter_image[i, j] = sum

plt.figure(figsize=(10, 6))

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

plt.subplot(1, 2, 2)
plt.imshow(filter_image, cmap='gray')
plt.title('Filtered Image (Averaging)')
plt.axis('off')
plt.show()

```

Original Image



Filtered Image (Averaging)







## 10 noise anlaysis on averaing filter

```
noise=10
mean=0
stddev=np.sqrt(noise/100.0)
noisy_image=np.random.normal(mean, stddev, (image_height,image_width))
noisy_image10=filter_image+noisy_image*255
noisy_image10=np.clip(noisy_image10,0,255).astype(np.uint8)
plt.imshow(noisy_image10,cmap='gray')
plt.title('Noisy Image')
plt.axis('off')
plt.show()
```

Noisy Image



```
image_height=noisy_image10.shape[0]
image_width=noisy_image10.shape[1]
filter_image=np.zeros((image_height,image_width))
pad=mask_size//2
padded_image=np.pad(noisy_image10,pad,mode='constant',constant_values=0)
for i in range(image_height):
    for j in range(image_width):
        sum=0
        for k in range(mask_size):
            for l in range(mask_size):
                roi=padded_image[i+k,j+l]
                sum+=roi*mask[k,l]
            filter_image[i,j]=sum
plt.imshow(filter_image,cmap='gray')
plt.title('Average Filter')
plt.axis('off')
plt.show()
```

Average Filter



### Conclusion

*veraging (smoothing) filters effectively reduce high-frequency noise in images. We introduced Gaussian noise and applied a  $3 \times 3$  spatial filter using convolution to smooth the image. While the filter successfully reduced noise, it also caused edge blurring due to the uniform averaging of neighboring pixel intensities. Larger kernel sizes provided stronger noise reduction but resulted in increased loss of fine details and edge sharpness.*

□



**Lab - 3 :** Noise Removal in Images

## Objective

The objective of this lab is to introduce the student types of noise and various noise removal techniques.

1. Adding noise to images
2. Removing noise from images

### ✓ Reading and displaying an image in python

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
img=cv2.imread("input2.jpg",0)
print(img.shape)
```

↗ (342, 342)

### ✓ Adding Noise to Images

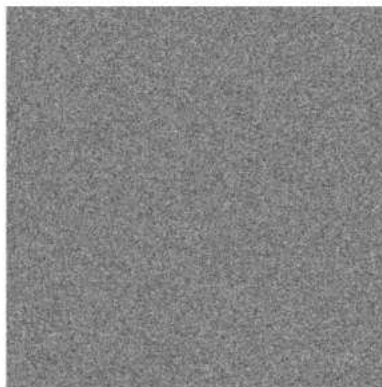
```
gauss_noise=np.zeros(img.shape,dtype=np.uint8)
cv2.randn(gauss_noise,128,20)
gauss_noise=(gauss_noise*0.5).astype(np.uint8)
gn_img=cv2.add(img,gauss_noise)
fig=plt.figure(dpi=300)
fig.add_subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis("off")
plt.title("Original")
fig.add_subplot(1,3,2)
plt.imshow(gauss_noise,cmap='gray')
plt.axis("off")
plt.title("Gaussian Noise")
fig.add_subplot(1,3,3)
plt.imshow(gn_img,cmap='gray')
plt.axis("off")
plt.title("Combined")
```

↗ Text(0.5, 1.0, 'Combined')

Original



Gaussian Noise



Combined



```
uni_noise=np.zeros(img.shape,dtype=np.uint8)
cv2.randu(uni_noise,0,255)
```

```

uni_noise=(uni_noise*0.5).astype(np.uint8)
un_img=cv2.add(img,uni_noise)
fig=plt.figure(dpi=300)
fig.add_subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis("off")
plt.title("Original")
fig.add_subplot(1,3,2)
plt.imshow(uni_noise,cmap='gray')
plt.axis("off")
plt.title("Uniform Noise")
fig.add_subplot(1,3,3)
plt.imshow(un_img,cmap='gray')
plt.axis("off")
plt.title("Combined")

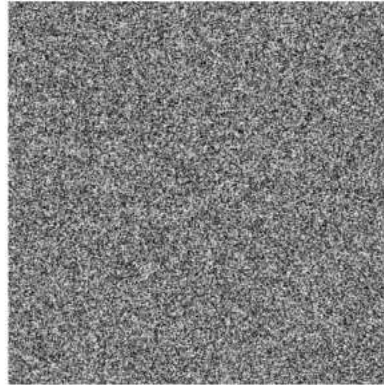
```

↗ Text(0.5, 1.0, 'Combined')

## Original



## Uniform Noise



## Combined



```

imp_noise=np.zeros(img.shape,dtype=np.uint8)
cv2.randu(imp_noise,0,255)
imp_noise=cv2.threshold(imp_noise,245,255,cv2.THRESH_BINARY)[1]
in_img=cv2.add(img,imp_noise)
fig=plt.figure(dpi=300)
fig.add_subplot(1,3,1)
plt.imshow(img,cmap='gray')
plt.axis("off")
plt.title("Original")
fig.add_subplot(1,3,2)
plt.imshow(imp_noise,cmap='gray')
plt.axis("off")
plt.title("Impulse Noise")
fig.add_subplot(1,3,3)
plt.imshow(in_img,cmap='gray')
plt.axis("off")
plt.title("Combined")

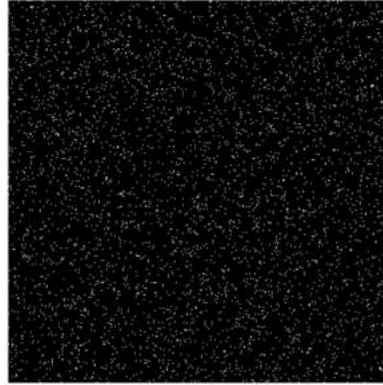
```

Text(0.5, 1.0, 'Combined')

Original



Impulse Noise



Combined



## ✓ Removal of noise from image

```
def plot_denoising(original_img, noisy_img, denoised_img, noise_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(denoised_img, cmap="gray")
    plt.axis("off")
    plt.title("After Denoising")
```

```
# Denoising and plotting for Gaussian Noise
denoised1 =cv2.fastNlMeansDenoising(gn_img, None,10, 10)
plot_denoising(img, gn_img, denoised1,"Gaussian")
```



Original



with Gaussian Noise



After Denoising



```
# Denoising and plotting for Uniform Noise
denoised2 =cv2.fastNlMeansDenoising(un_img,None,10, 10)
plot_denoising(img, un_img, denoised2,"Uniform")
```



Original



with Uniform Noise



After Denoising



```
# Denoising and plotting for Impulse Noise
denoised3 =cv2.fastNlMeansDenoising(in_img, None,10, 10)
plot_denoising(img, in_img, denoised3,"Impulse")
```





Original



with Impulse Noise



After Denoising



### Using Median Filter

```

blurred1=cv2.medianBlur(gn_img,3)
blurred2=cv2.medianBlur(un_img,3)
blurred3=cv2.medianBlur(in_img,3)
def plot_median_filter(original_img, noisy_img, filtered_img, noise_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("Median Filter")
plot_median_filter(img, gn_img, blurred1, "Gaussian")

```



Original



with Gaussian Noise



Median Filter



```
plot_median_filter(img, un_img, blurred2, "Uniform")
```



Original



with Uniform Noise



Median Filter



```
plot_median_filter(img, in_img, blurred3, "Impulse")
```





Original



with Impulse Noise



Median Filter



#### Using Gaussian filter

```

blurred21=cv2.GaussianBlur(gn_img,(3,3),0)
blurred22=cv2.GaussianBlur(un_img,(3,3),0)
blurred23=cv2.GaussianBlur(in_img,(3,3),0)
def plot_gaussian_filter(original_img, noisy_img, filtered_img,
noise_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("Gaussian Filter")
plot_gaussian_filter(img, gn_img, blurred21, "Gaussian")

```



Original

with Gaussian Noise

Gaussian Filter

Original



with Gaussian Noise



Gaussian Filter



```
[13] plot_median_filter(img, in_img, blurred2, "Uniform")
```



Original



with Uniform Noise



Median Filter



```
[13] plot_gaussian_filter(img, in_img, blurred23, "Impulse")
```



Original



with Impulse Noise



Gaussian Filter



## Conclusion

In this lab, we explored how different types of noise—**Gaussian**, **Uniform**, and **Impulse (Salt-and-Pepper)**—affect grayscale images and how various filtering techniques help in reducing these noises. Specifically, we:

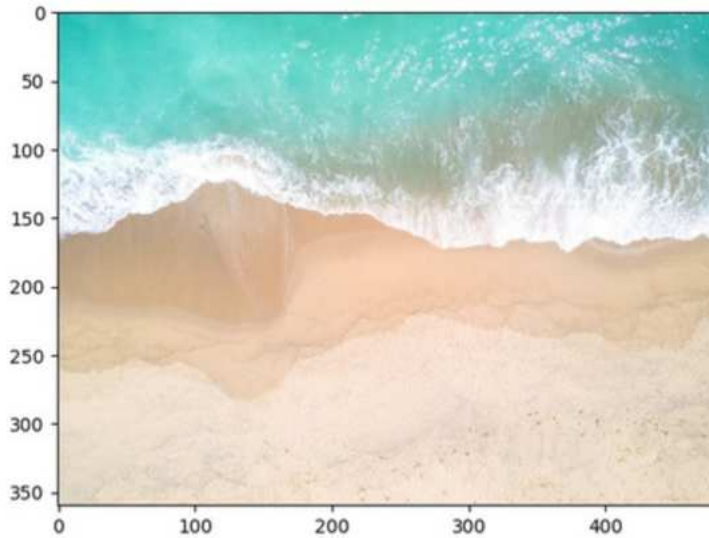
- Added synthetic noise to images using OpenCV functions.
- Applied denoising techniques such as:
  - **Non-Local Means Denoising**: Effective for Gaussian and Impulse noise.
  - **Median Filtering**: Particularly effective for Impulse noise.
  - **Gaussian Filtering**: Smoothed out Gaussian noise but may blur edges.

**Lab 4: Image Segmentation with K Means Clustering**

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

```
img = plt.imread("beach_image.jpg")
plt.imshow(img)
```

 <matplotlib.image.AxesImage at 0x7d54b883b210>




```
img = cv2.imread("beach_image.jpg")
img=cv2.cvtColor(img ,cv2.COLOR_BGR2RGB)
plt.figure(figsize=(10,8))
plt.imshow(img)
```

 <matplotlib.image.AxesImage at 0x7d54aa80cad0>



```
img.shape
```

 (360, 480, 3)

```
vectorized_img = img.reshape((-1,3))  
vectorized_img.shape
```

```
↗ (172800, 3)
```

```
vectorized_img= np.float32(vectorized_img)  
vectorized_img
```

```
↗ array([[ 33., 192., 187.],  
        [ 39., 192., 187.],  
        [ 44., 193., 187.],  
        ...,  
        [234., 219., 196.],  
        [232., 217., 196.],  
        [233., 218., 197.]], dtype=float32)
```

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

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

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

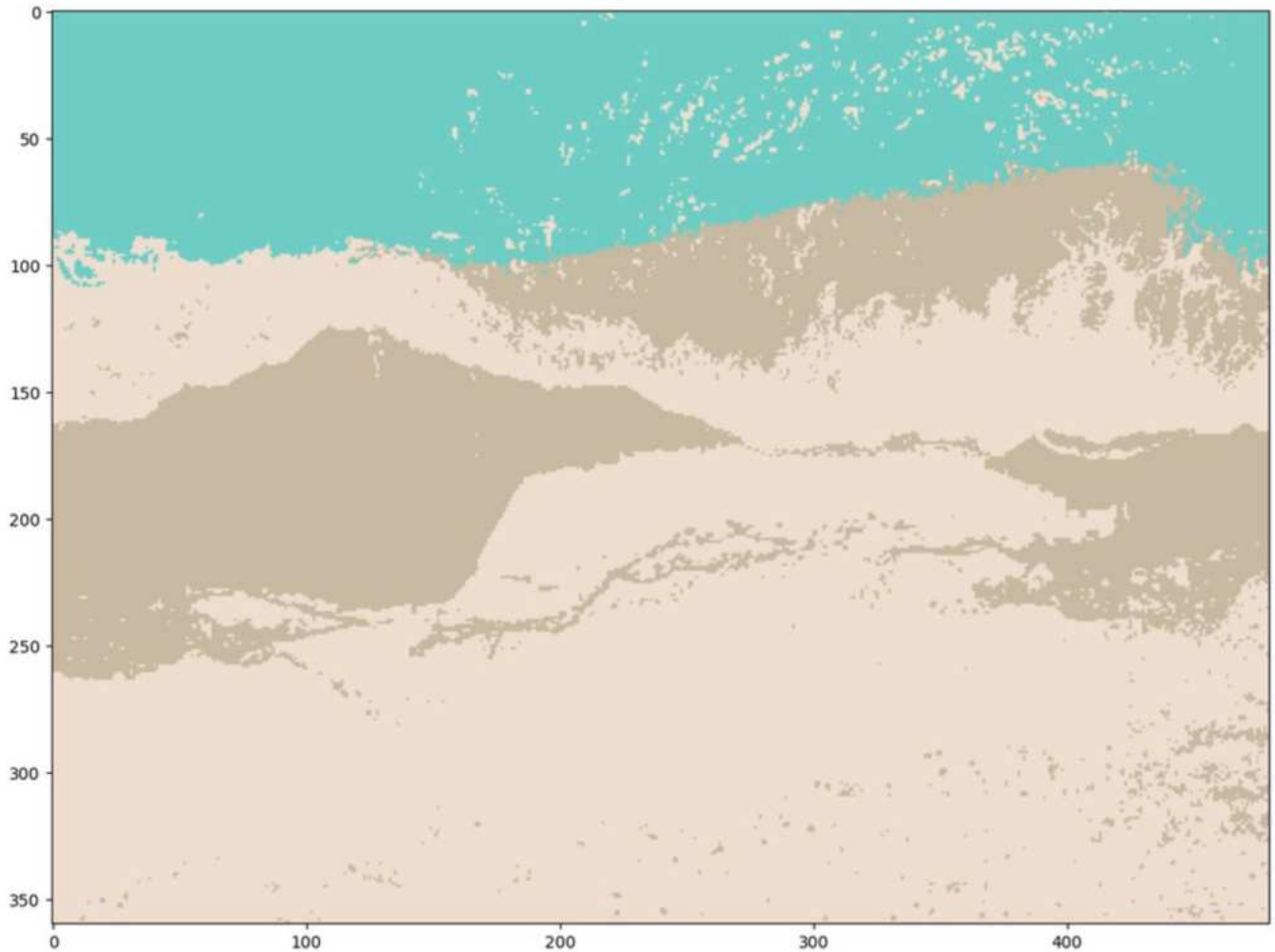
```
↗ array([[237, 222, 208],  
        [110, 206, 198],  
        [200, 185, 162]], dtype=uint8)
```

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

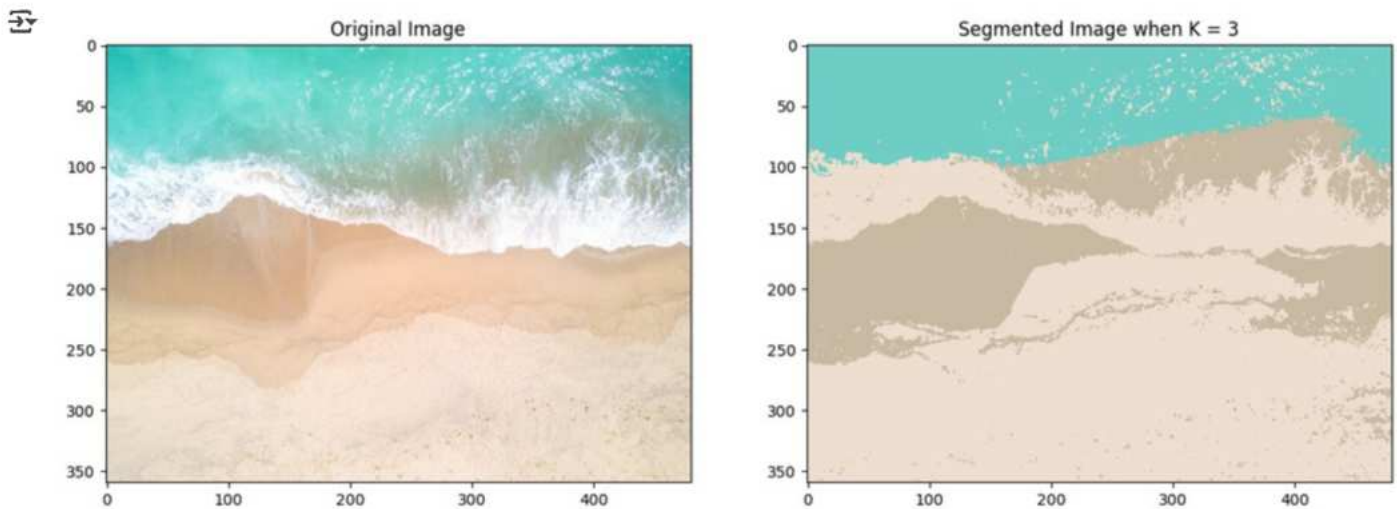
```
plt.figure(figsize=(15,10))  
plt.imshow(result_image)
```



 <matplotlib.image.AxesImage at 0x7d54aa893690>



```
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
attempts=10
ret,label,center=cv2.kmeans(vectorized_img,K,None,criteria,attempts,cv2.KMEANS_PP_CENTERS)
center = np.uint8(center)
center

```

```

array([[ 96, 207, 198], [237, 217,
198], [212, 184, 155],
[153, 195, 188],
[237, 240, 243]], dtype=uint8)

```

```

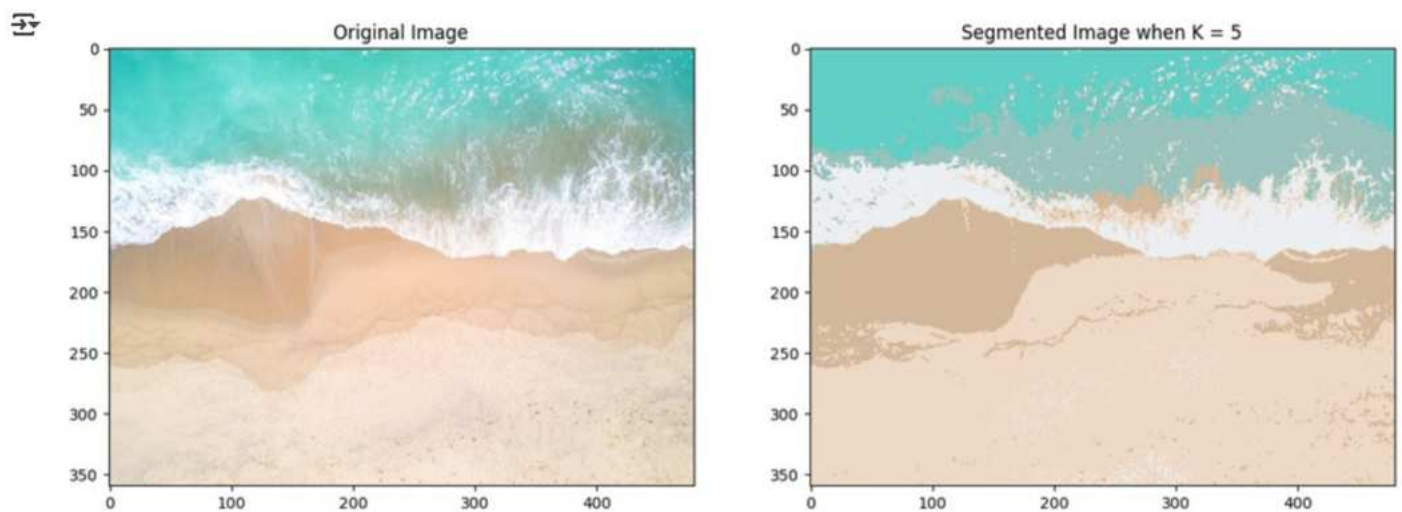
res = center[label.flatten()]
result_image = res.reshape((img.shape))

```

```

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

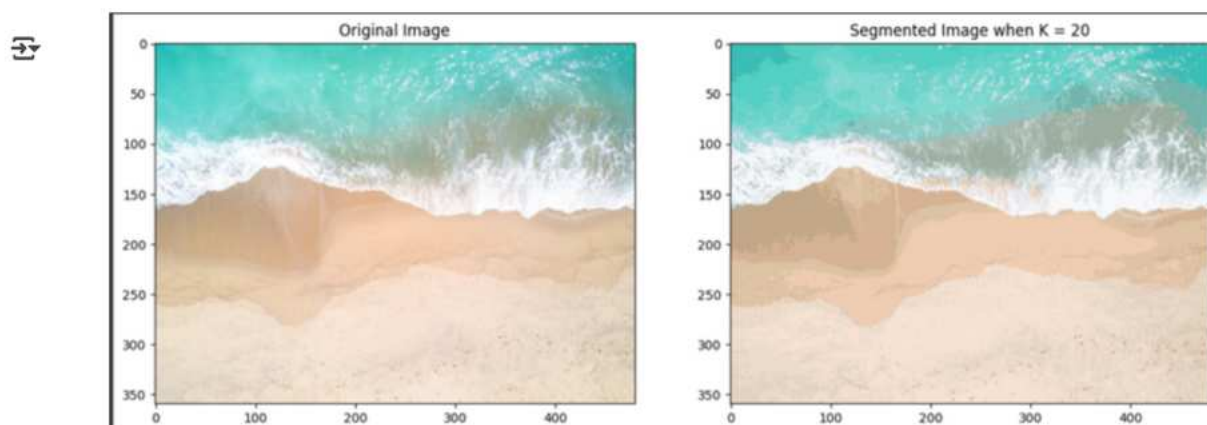
```



```

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

```



## Conclusion

In this lab, we explored **image segmentation** using **K-Means Clustering**, a popular unsupervised machine learning algorithm. We learned how to:

- Load and display images using `OpenCV` and `matplotlib`.
- Reshape image data into a suitable format for clustering.
- Apply the K-Means algorithm with different values of **K** to segment the image into distinct color clusters.
- Visually compare how increasing K affects the segmentation quality and image details.

## ✓ Lab 5 : Image interest point detection using Harrier corner detection

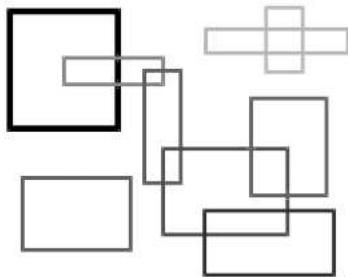
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

image_path = 'harris2.png'
img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
```

```
plt.figure(figsize=(6, 6))
plt.imshow(img, cmap='gray')
plt.title("Uploaded Image")
plt.axis('off')
plt.show()
```



Uploaded Image



```
if img is None:
    raise ValueError("Image not loaded. Check the image path or internet connection.")
```

```
gray = np.float32(img)
```

## ✓ Applying Harris corner detection

```
dst = cv2.cornerHarris(gray, blockSize=2, ksize=3, k=0.04)
```

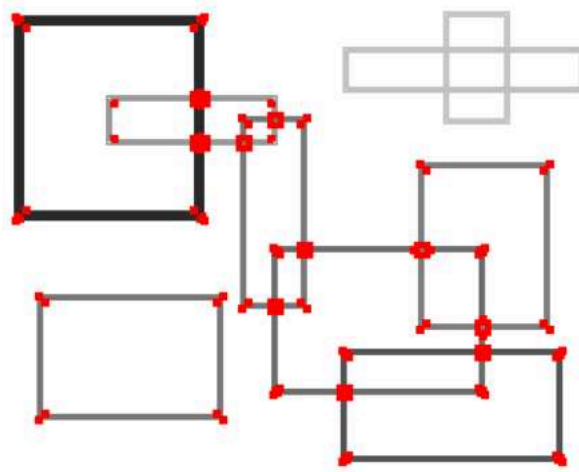
```
dst = cv2.dilate(dst, None)
```

```
img_color = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
img_color[dst > 0.01 * dst.max()] = [0, 0, 255]
```

```
plt.figure(figsize=(10, 10))
plt.imshow(cv2.cvtColor(img_color, cv2.COLOR_BGR2RGB))
plt.title('Harris Corner Detection')
plt.axis('off')
plt.show()
```



## Harris Corner Detection



## Conclusion

In this lab, we successfully implemented **Harris Corner Detection**, a fundamental technique in computer vision used to identify **interest points or corners** in an image. Here's a brief summary:

- The image was first loaded and converted to grayscale.
- The **Harris detector** was applied using OpenCV's `cv2.cornerHarris()` function.
- Corner points were enhanced and visualized by dilating them and highlighting strong responses.
- Detected corners were marked in **red** on the original image to show the points of interest.

## ✓ Lab 5: Object recognition using HoG and machine learning

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

from PIL import Image
import numpy as np
import matplotlib.pyplot as plt


image = Image.open("athletic.jpg")
image = np.array(image) # Convert to proper NumPy array

plt.figure(figsize=(12,10))
plt.axis("off")
plt.imshow(image)
plt.show()
```



```
imagePath="athletic.jpg"
image = cv2.imread(imagePath)
print(type(image))
plt.figure(figsize=(12,10))
plt.axis("off")
plt.imshow(image)
plt.show()
```



 <class 'numpy.ndarray'>

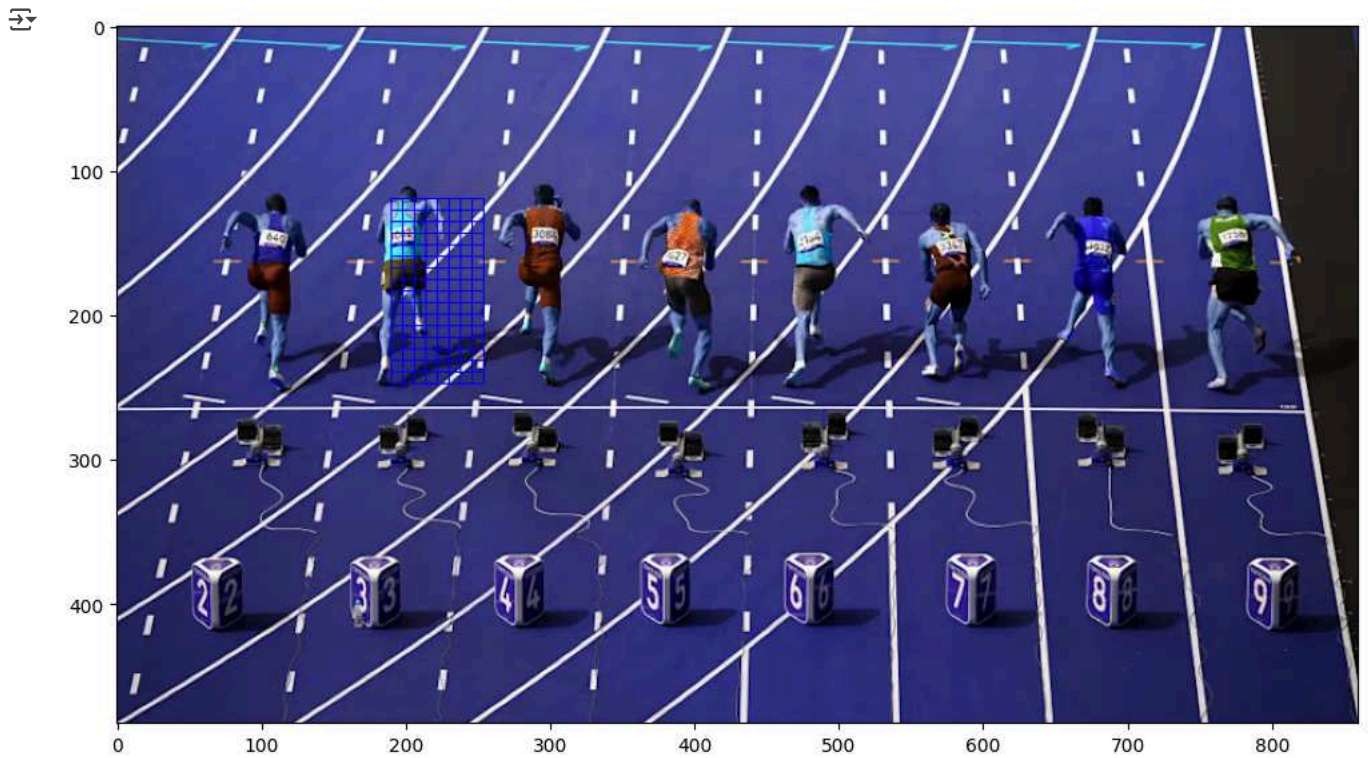


## ▼ Extract Image Patch

```
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()
```



## ▼ Calculate Gradient

Apply Sobel Filter to calculate partial derivations

For each of the 3 channels the derivation in x- and y-direction is calculated by applying the *Sobel-filter*.

```
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)
```

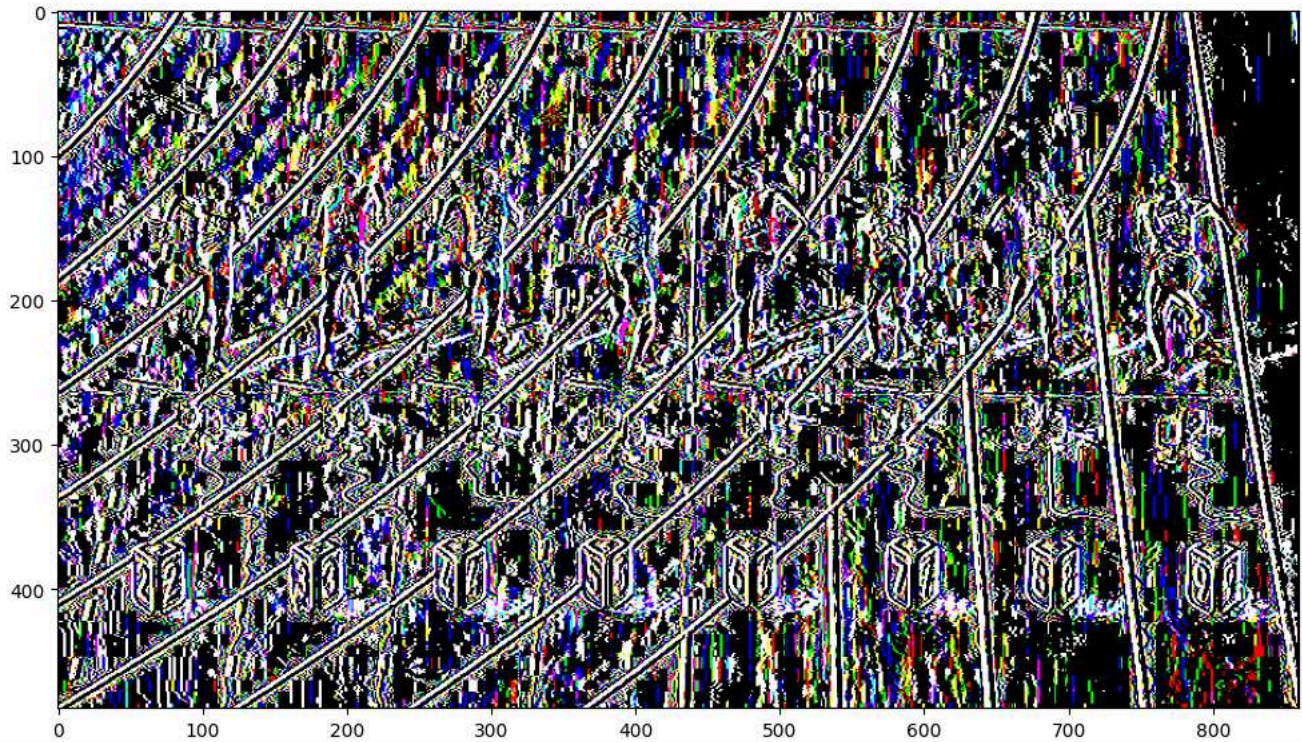
```
print(gx.shape)
print(gy.shape)
```

```
(483, 860, 3)
(483, 860, 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 integer:

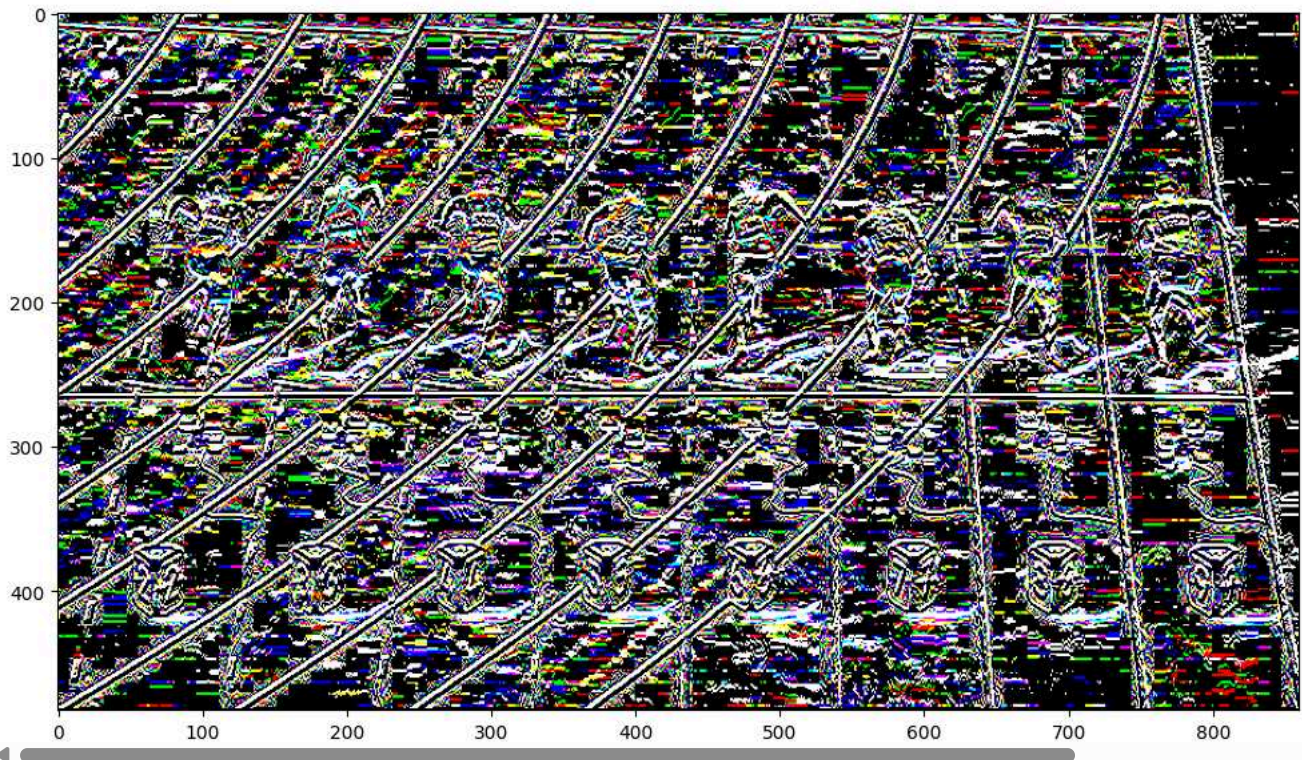


gx.shape

(483, 860, 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 integer:



### ▼ Analyse Pixels and Gradients in a $8 \times 8$ region

For the upper left  $8 \times 8$ -region of the selected patch the pixel values of channel `chan` are displayed and visualized in the following code cells. There is a vertical edge in within this region. Hence there should be high values for the gradients in direction of  $x$ .

**Note:** In this notebook an angle of 0 degrees points to the right and an angle of 90 degrees points to the top.



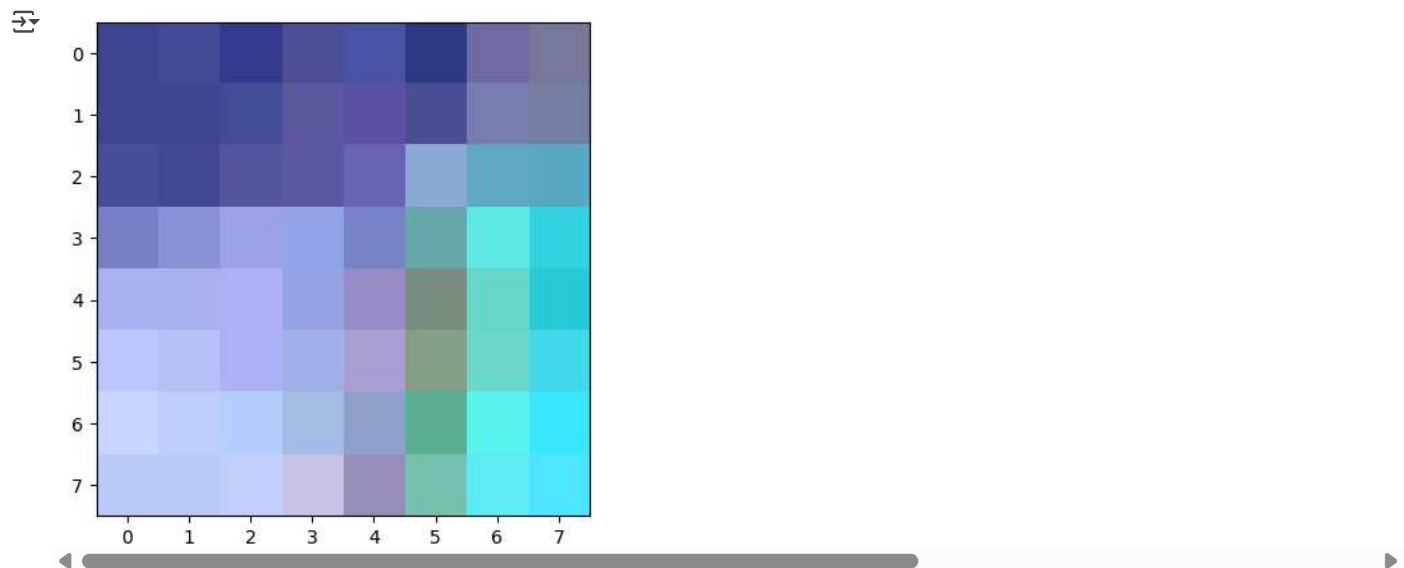
```
Chan=0 # Channel to be used for visualisation
W=8    # Region-width and -height
```

Pixel values in the selected channel at the selected region:

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

```
[[ 63  68  51  78  74  44 111 120]
 [ 63  64  66  93  91  75 120 117]
 [ 72  66  85  91 106 138  96  90]
 [120 137 156 145 120 105  94  51]
 [170 168 176 150 152 121 101  39]
 [187 181 175 161 170 134 108  66]
 [200 191 180 164 143  90  88  57]
 [187 185 194 200 152 120  93  78]]
```

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



Gradient  $g_x$  in x-direction of the selected channel:

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

```
[[ 1. -12. 10. 23. -34. 37. 76. 4.]
 [ 3.  3. 29. 25. -18. 29. 42. 13.]
 [-6. 13. 25. 21. 47. -10. -48. 17.]
 [34. 36.  8. -36. -40. -26. -54. -42.]
 [-7.  6. -18. -24. -29. -51. -82. -61.]
 [ 4. -12. -20.  -5. -27. -62. -68. -30.]
 [ 5. -20. -27. -37. -74. -55. -33.  -5.]
 [-4.  7. 15. -42. -80. -59. -42.  3.]]
```

Gradient  $g_y$  in y-direction of the selected channel:

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

```
[[ -4.  -4.  -1.  23.  23.  5.  33.  36.]
 [  9.  -2.  34.  13.  32.  94. -15. -30.]
 [57.  73.  90.  52.  29.  30. -26. -66.]
 [98. 102.  91.  59.  46. -17.  5. -51.]
 [67.  44.  19.  16.  50.  29.  14.  15.]
 [30.  23.  4.  14.  -9. -31. -13.  18.]
 [ 0.  4.  19.  39. -18. -14. -15.  12.]
 [ 9.  -3.  21.  22.  -7.  34.  -9.  12.]]
```

## ✓ Magnitude and Angle of the Gradients

The magnitude and the direction of the gradient can be calculated as follows:

$$|g| = \sqrt{g_x^2 + g_y^2}$$

$$\Theta = \arctan\left(\frac{g_y}{g_x}\right)$$

For this calculation *opencv's* `cartToPolar()` -function can be applied:

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

```
(483, 860, 3)
(483, 860, 3)
```

For ease of visualization, magnitude and angle are represented as integers:

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

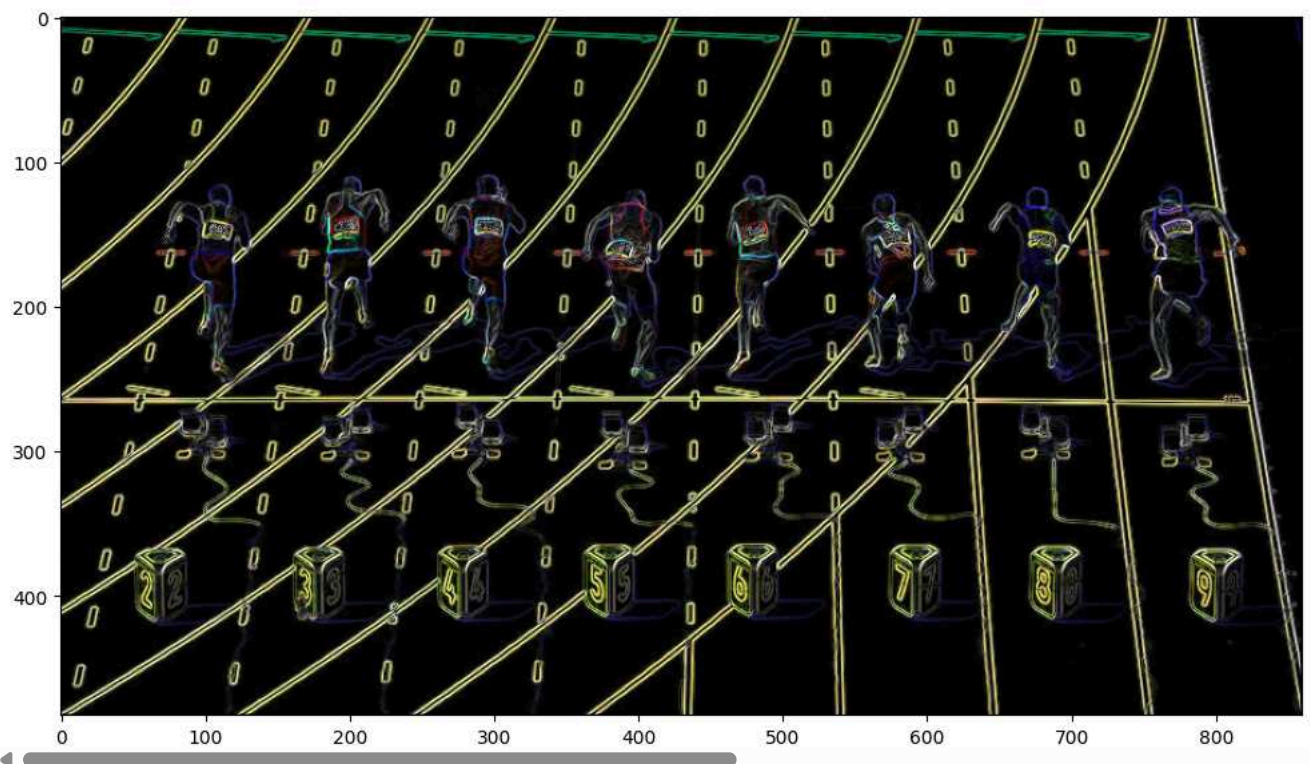
Gradient-magnitude of the selected channel and region:

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

```
[[ 4 12 10 32 41 37 82 36]
 [ 9  3 44 28 36 98 44 32]
 [ 57 74 93 56 55 31 54 68]
 [103 108 91 69 60 31 54 66]
 [ 67 44 26 28 57 58 83 62]
 [ 30 25 20 14 28 69 69 34]
 [ 5 20 33 53 76 56 36 13]
 [ 9  7 25 47 80 68 42 12]]
```

```
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 integer:



```
mag.shape
```

```
(483, 860, 3)
```

## ✓ Apply max-operator over channels

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

```
(483, 860)
```

As shown below, at the upper-left pixel in the selected patch channel 0 has the strongest gradient. One pixel below, the highest gradient-magnitude is in channel 1.

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

```
[[0 0 0 0 0 0 0 0]
 [1 2 0 0 0 1 1 1]
 [1 1 0 1 1 1 1 1]
 [1 1 2 1 0 1 1 0]
 [1 1 0 2 2 1 2 0]
 [1 1 1 2 2 0 2 2]
 [1 0 1 0 2 1 2 2]
 [0 2 0 1 0 1 2 2]]
```

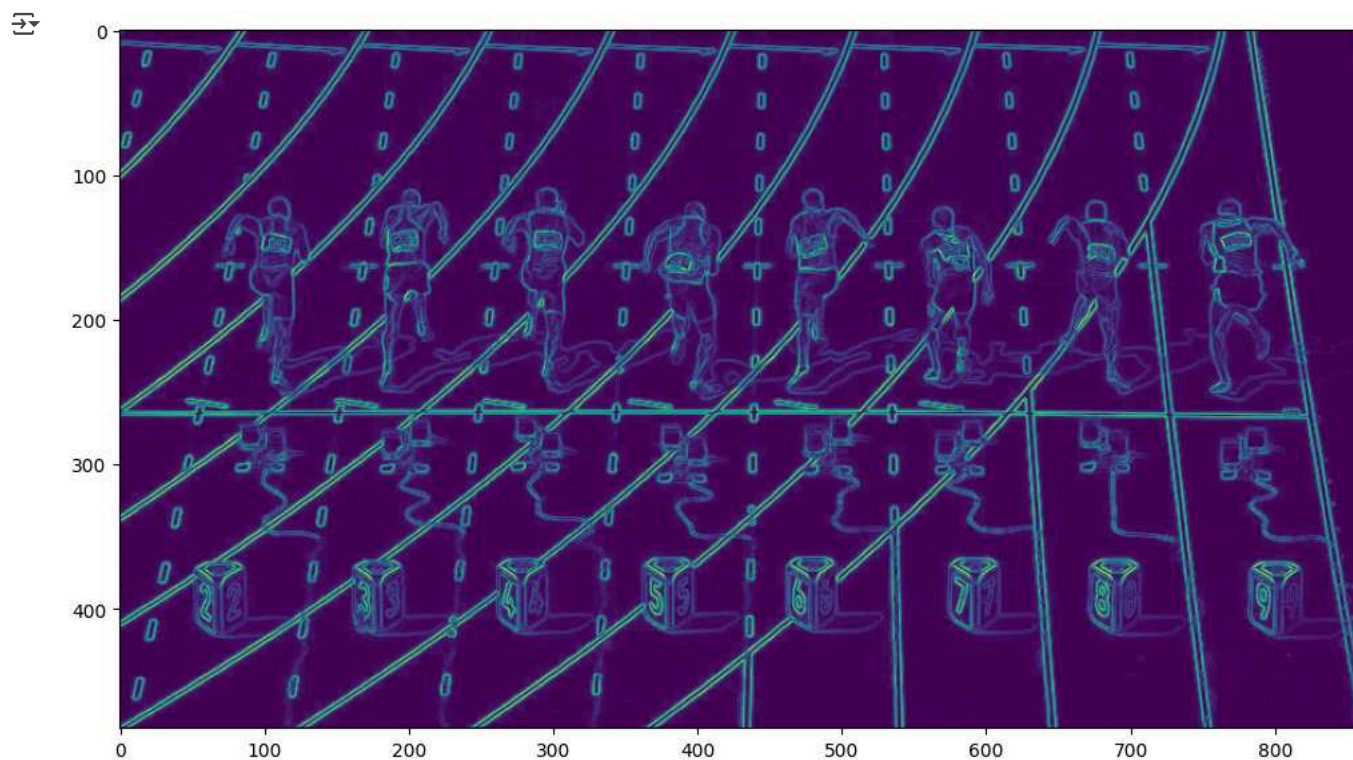
Calculate a 2-dimensional array, which contains only the **gradient-magnitude** of the locally *strongest* channel (strongest=channel with highest gradient).

```
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)
```

```
(483, 860)
```

Visualisation of the maximum Gradient:

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



Calculate a 2-dimensional array, which contains only the **gradient-angles** of the locally *strongest* channel (strongest=channel with highest gradient).

```
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)
```

```
(483, 860)
```

Strongest gradient-magnitudes in the selected region:



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

```
↩ [[ 4. 12. 10. 32. 41. 37. 82. 36.]
 [ 10. 9. 44. 28. 36. 121. 80. 53.]
 [ 58. 76. 93. 78. 97. 113. 109. 86.]
 [107. 111. 93. 83. 60. 106. 63. 66.]
 [ 71. 47. 26. 47. 102. 74. 91. 62.]
 [ 35. 34. 35. 37. 97. 69. 107. 50.]
 [ 7. 20. 34. 53. 87. 90. 116. 21.]
 [ 9. 14. 25. 65. 80. 102. 78. 16.]]
```

Strongest gradient-angles in the selected region:

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

```
↩ [[284. 198. 354. 44. 145. 7. 23. 83.]
 [ 73. 319. 49. 27. 119. 68. 53. 65.]
 [ 94. 84. 74. 79. 31. 51. 90. 86.]
 [ 69. 71. 79. 112. 131. 344. 44. 230.]
 [ 94. 94. 133. 177. 173. 354. 343. 166.]
 [ 81. 127. 120. 178. 178. 206. 20. 49.]
 [ 23. 168. 118. 133. 194. 21. 21. 58.]
 [113. 285. 54. 172. 184. 24. 354. 270.]]
```

## ✓ Create Histogram of single $8 \times 8$ -region

For each  $8 \times 8$ -region of the selected patch a 9-bin histogram is calculated. The bin-centers are

$$0^\circ, 20^\circ, 40^\circ, 60^\circ, 80^\circ, 100^\circ, 120^\circ, 140^\circ, 160^\circ.$$

Gradient angles  $\Theta$  with a value of  $\Theta > 180^\circ$  are mapped to their opposite direction  $\Theta' = \Theta - 180^\circ$ .

```
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])
```

```
↩ [[104. 18. 174. 44. 145. 7. 23. 83.]
 [ 73. 139. 49. 27. 119. 68. 53. 65.]
 [ 94. 84. 74. 79. 31. 51. 90. 86.]
 [ 69. 71. 79. 112. 131. 164. 44. 50.]
 [ 94. 94. 133. 177. 173. 174. 163. 166.]
 [ 81. 127. 120. 178. 178. 26. 20. 49.]
 [ 23. 168. 118. 133. 14. 21. 21. 58.]
 [113. 105. 54. 172. 4. 24. 174. 90.]]
```

After this mapping into the range  $[0, 180]$ , each of the 64 gradient-angles in the region contributes to one or two bins of the histogram as follows:

- If the gradient-angle  $\Theta_{ij}$  is equal to one of the 9 bin-centers, then position  $i, j$  adds the value  $|g_{ij}|$  to the bin with center  $\Theta_{ij}$ , where  $|g_{ij}|$  is the gradient-magnitude at position  $i, j$ .
- If the gradient-angle  $\Theta_{ij}$  lies between two bin-centers, where  $\Theta_L$  is the next lower and  $\Theta_R$  is the next higher bin-center, then position  $i, j$  adds
  - the value  $|g_{ij}| \cdot r$  to the bin with center  $\Theta_R$ , and
  - the value  $|g_{ij}| \cdot (1 - r)$  to the bin with center  $\Theta_L$ ,

where

$$r = \frac{\Theta_{ij} - \Theta_L}{20}.$$

Due to the circular nature of angles, the next higher bin center  $\Theta_R$  for all angles  $160^\circ < \Theta_{ij} < 180^\circ$  is  $0^\circ$ .

**Example:** Assume that at a given pixel the gradient-magnitude is  $|g| = 40$  and the gradient-angle is  $325^\circ$ . The gradient-angle is mapped to  $\Theta = 325 - 180 = 145$ . The next lower bin-center is  $\Theta_L = 140$ , the next higher bin-center is  $\Theta_R = 160$ . This gradient contributes a value of  $0.25 \cdot 40 = 10$  to the bin centered at  $\Theta_R = 160$  and a value of  $0.75 \cdot 40 = 30$  to the bin centered at  $\Theta_L = 140$ .

The function `createHist()`, as defined in the following code cell, calculates for the gradient-angles `AngArray` and the gradient magnitudes `MagArray` of a given array the corresponding histogram.

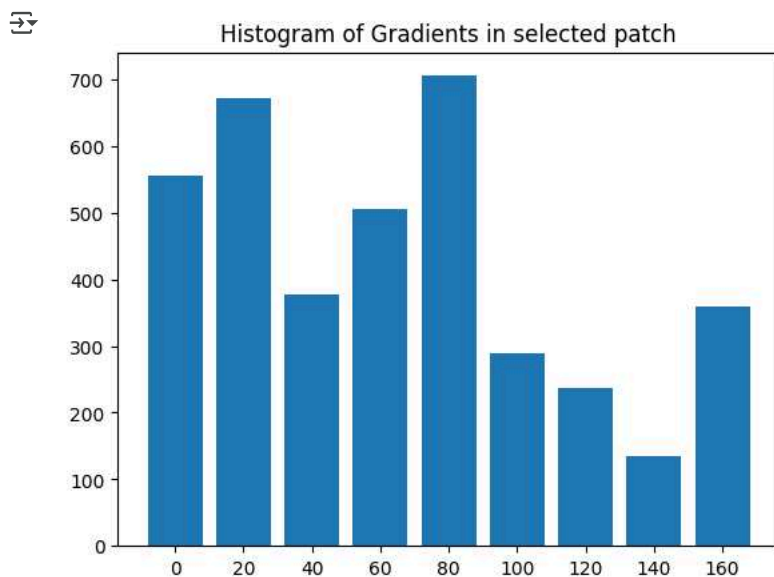
```
def createHist(AngArray,MagArray,BS=20,BINS=9):
    hist=np.zeros(BINS)
    for r in range(AngArray.shape[0]):
        for c in range(AngArray.shape[1]):
            #print(AngArray[r,c])
            binel,rem = np.divmod(AngArray[r,c],BS)
            weightR=rem*1.0/BS
            weightL=1-weightR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

Next, we apply this function `createHist()` for calculating the histogram of the upper left region in the selected patch:

```
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)
```

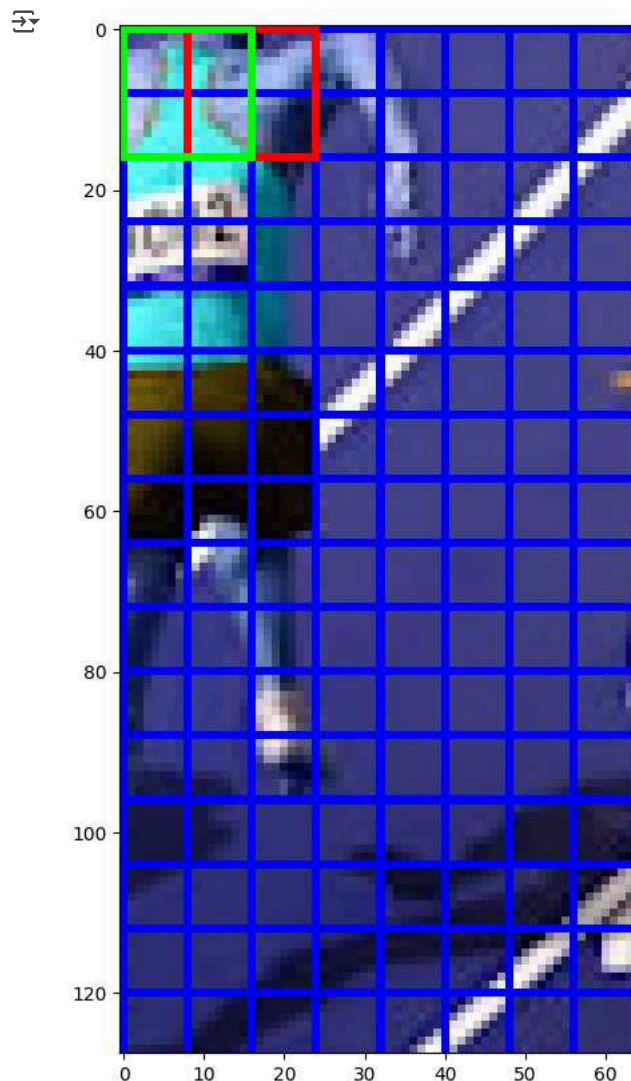
```
↗ Gradient histogram of the selected region:
[556.05 670.75 377.05 505.95 705.05 289.1 236.95 135.55 358.55]
```

```
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]+PatchSize[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()
```



In the following code-cells the normalized 36-Bin histogram for the green super-region is calculated.

First, the four 9-Bin histograms for the regions within the green super-region are obtained:

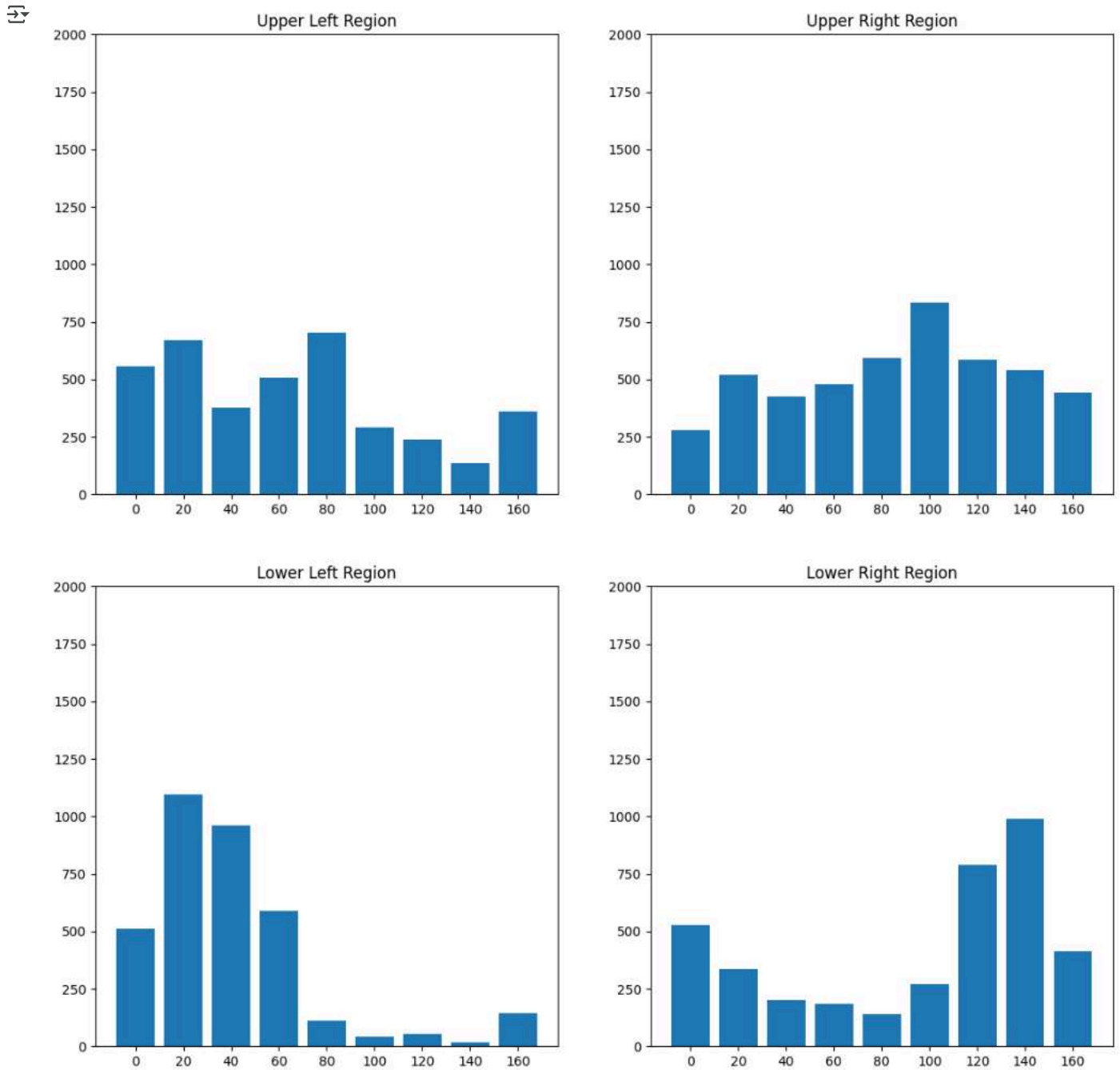
```
hist11=spotHist
print(hist11)
spotAngles=mappedAngles[Location[0]:Location[0]+W,Location[1]+W:Location[1]+2*W]
spotMag=maxmag[Location[0]:Location[0]+W,Location[1]+W:Location[1]+2*W]
hist12=createHist(spotAngles,spotMag)
print(hist12)
spotAngles=mappedAngles[Location[0]+W:Location[0]+2*W,Location[1]:Location[1]+W]
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:Location[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)
```

```
[556.05 670.75 377.05 505.95 705.05 289.1 236.95 135.55 358.55]
[279.8 520.5 427.15 479.1 592.15 835.2 584.2 541.95 440.95]
[ 512.9 1093.6 960.6 590.25 110.85 42. 55.8 17.5 143.5 ]
[527.55 334.9 202.75 183.25 140.65 272.55 788.8 988.95 411.6 ]
```

The four non-normalized histograms are visualized in the code-cell below:

```
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()
```



Next, L2-normalization is performed over a vector, which is the concatenation of all 4 histograms in the region:

```
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:

```
[ 556.05  670.75  377.05  505.95  705.05  289.1   236.95  135.55  358.55
 279.8   520.5   427.15  479.1   592.15  835.2   584.2   541.95  440.95
 512.9  1093.6   960.6   590.25  110.85   42.     55.8    17.5   143.5
 527.55  334.9   202.75  183.25  140.65  272.55  788.8   988.95  411.6 ]
```

L2-Norm:

```
3110.7783334721876
```

Normalized Histogram of upper-left super-region:

```
[0.18 0.22 0.12 0.16 0.23 0.09 0.08 0.04 0.12 0.09 0.17 0.14 0.15 0.19
```

**Conclusion:**

In this lab, we explored the implementation and evaluation of feature detection and matching using the SIFT algorithm. We learned how to detect keypoints, extract descriptors, and apply feature matching techniques between images. Additionally, we gained practical experience with homography estimation to align images based on matched features. This lab reinforced the importance of robust feature detection and matching in various computer vision applications such as image stitching and object recognition.



## Lab 6: 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])

# 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()
}

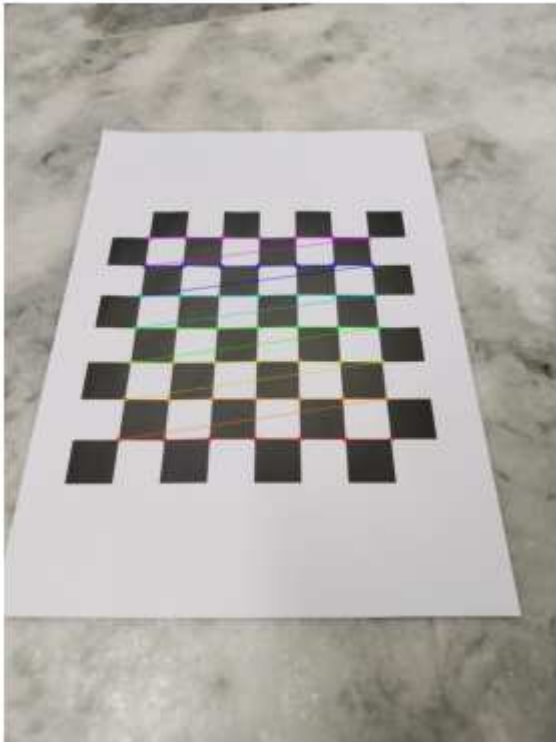
with open("calibration_matrix.yaml", "w") as f:
    yaml.dump(calibration_data, f)

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

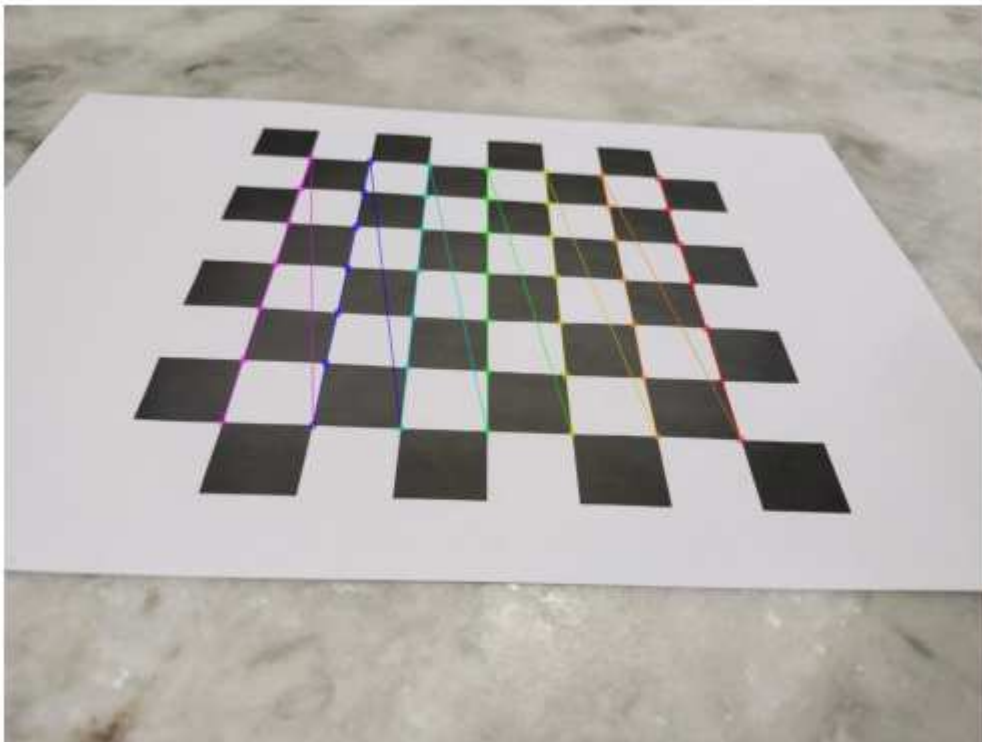
```



Chessboard Detection #1

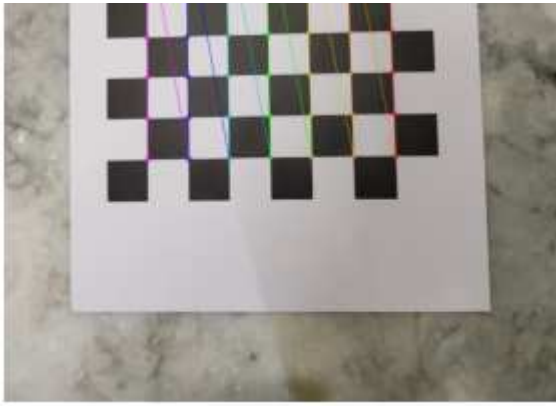


Chessboard Detection #2



Chessboard Detection #3





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

In this lab, we successfully performed **camera calibration** using images of a 7×7 chessboard pattern. The key steps included:

- **Detecting and refining corners** in multiple calibration images.
- **Collecting object points and image points** from valid detections.
- **Calibrating the camera** to compute the camera matrix and distortion coefficients.
- **Saving the calibration results** to a YAML file for future use in undistorting images.

The calibration yielded a valid **camera matrix** and **distortion coefficients**, confirming that the camera's internal parameters and lens distortion were accurately estimated using OpenCV.