LABORATORY ACTIVITY:

Step 1: Download the images from the webpage (Instructor will provide the URL at the lab)

Step 2: Read the original image into a Matrix.

```
image = cv.imread('Sample1.jpg')    # Read the image
plt.imshow(cv.cvtColor(image, cv.COLOR_BGR2RGB))
plt.title("Original image")
plt.axis('off')
plt.show()
```

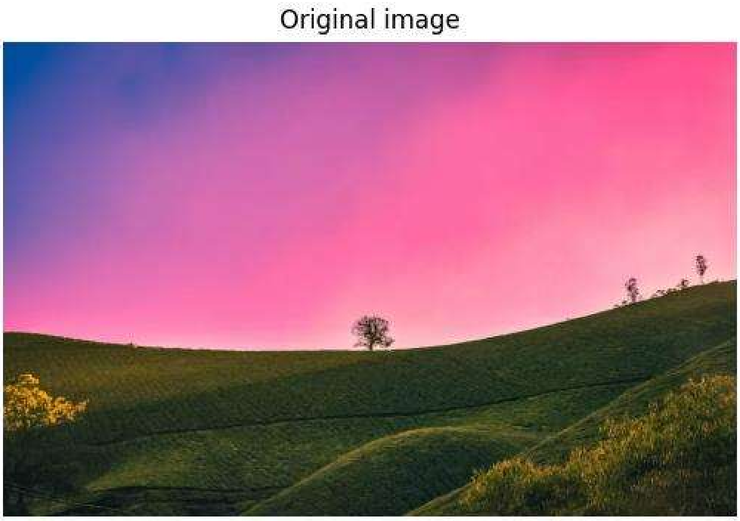

Figure 1: Original Image

Step 3:

E/18/268 => 2*60, 4*68 => (120,272)

```
# Pixel position according to my E number (E/18/268) x: 2*60 = 120 ,
y: 68*4 = 272
y, x = 272, 120
crop_size = 16
cropped_image = image[y:y+crop_size, x:x+crop_size]
cv.imwrite("Cropped_image.jpg", cropped_image)


color_cropped_image = cropped_image[:, :, 2]
# Take the red value (2) for build the huffman code
```

Figure 2: 16x16 Cropped Image

Step 4: Quantize the output at Step 3 into 8 levels (level 0-7) using uniform quantization.

```
def quantize_matrix(matrix, quantization_levels):
    # Define the quantization range
    min_range, max_range = 0, 256

    # Calculate the width of each quantization level
    level_width = (max_range - min_range) / len(quantization_levels)

    # Quantize the matrix
    quantized_matrix = np.zeros_like(matrix, dtype=np.uint8)
    for i, level in enumerate(quantization_levels):
        lower_bound = int(i * level_width)
        upper_bound = int((i + 1) * level_width)
        mask = np.logical_and(matrix >= lower_bound, matrix <
upper_bound)
        quantized_matrix[mask] = level

    return quantized_matrix


quantization_levels = [16, 48, 80, 112, 144, 176, 208, 240]  #
Quantization levels

# Quantize the rearranged cropped image
quantized_img_red = quantize_matrix(color_cropped_image,
quantization_levels)
print(quantized_img_red)
```

```
[[176 176 176 176 176 176 176 176 176 176 176 176 176 176 208 208]
 [176 176 176 176 176 176 176 176 176 176 176 176 208 208 208 208]
 [176 176 176 176 176 176 176 176 176 176 208 208 208 208 208 208]
 [176 176 176 176 176 176 208 208 176 176 208 208 208 208 208 208]
 [176 176 176 176 208 208 208 208 208 208 208 208 208 208 208 208]
 [176 176 176 176 208 208 208 208 208 208 208 208 208 208 208 208]
 [176 176 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [176 176 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]]
```

Figure 3: Matix of quantize the image

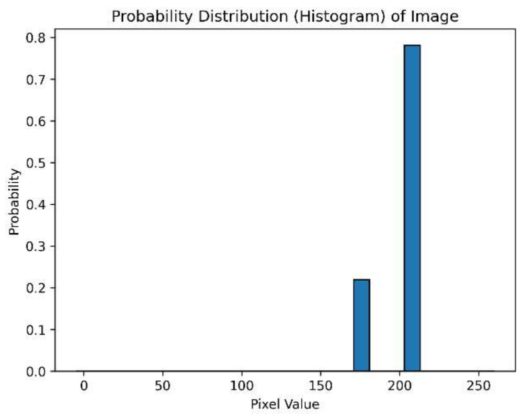

Figure 4: Histogram of quantize the image

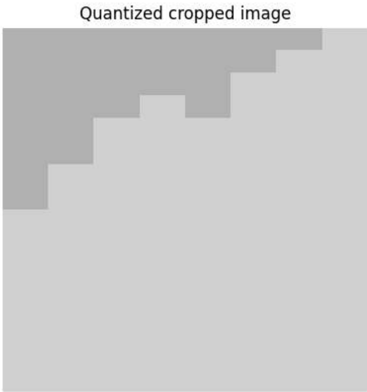

Figure 5: Red color for quantize the image

Since the cropped image has only 2 quantized values (208,176), i have assigned some pixel values in cropped image only for build the Huffman code

```
positions_and_values = {(0, 1): 20,(1, 1): 20,(0,2 ): 45,(1,2 ):
45,(3,2 ): 45,(2,2 ): 45,(0, 3): 83,(1, 3): 83,(2, 3): 83,(3, 3):
83,(0, 4): 120,(0, 5): 145,(0, 6): 245,(0, 7): 245,(0, 8): 245,(0, 9):
245,(0, 10): 245,(0, 11): 245}

for position, value in positions_and_values.items():
    color_cropped_image[position] = value
```

```
[[176 16 48 80 112 144 240 240 240 240 240 240 176 176 208 208]
 [176 16 48 80 176 176 176 176 176 176 176 176 208 208 208 208]
 [176 176 48 80 176 176 176 176 176 176 208 208 208 208 208 208]
 [176 176 48 80 176 176 208 208 176 176 208 208 208 208 208 208]
 [176 176 176 176 208 208 208 208 208 208 208 208 208 208 208 208]
 [176 176 176 176 208 208 208 208 208 208 208 208 208 208 208 208]
 [176 176 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [176 176 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]
 [208 208 208 208 208 208 208 208 208 208 208 208 208 208 208 208]]
```

Figure 6: Matix of quantize the image after after assigned values

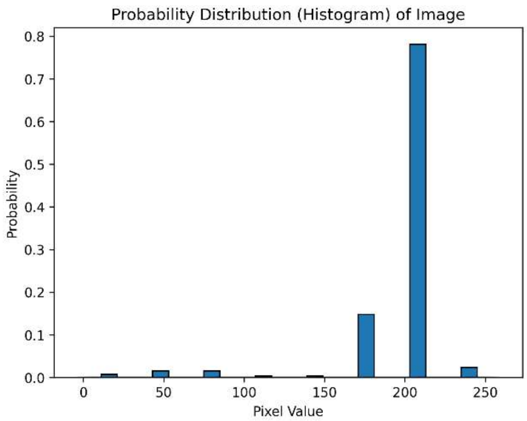

Figure 7: Matix of quantize the image assigned values

Figure 5: Red color for quantize the image after assign values

Step 5: Find the probability of each symbol distribution of the output at Step 4.

```
def calculate_probabilities(data):
    unique_values, counts = np.unique(data, return_counts=True)
    probabilities = counts / len(data)
    return zip(unique_values, probabilities)

# Flatten the rearranged quantized image array to a 1D array
flat_image = quantized_img_red.flatten()

# Calculate the histogram of the quantized image
hist, bins = np.histogram(flat_image, bins=256, range=(0, 256),
density=True)


# Calculate probabilities
probabilities = list(calculate_probabilities(flat_image))

# Sort the list based on the second element of each tuple (the
probability)
sorted_probabilities = sorted(probabilities, key=lambda x: x[1],
reverse=True)

# Display the sorted list
print("Probability of each symbol distribution:")
print(sorted_probabilities)
```

Output:

[(208, 0.78125), (176, 0.1484375), (240, 0.0234375), (48, 0.015625), (80, 0.015625), (16, 0.0078125), (112, 0.00390625), (144, 0.00390625)]

Table 1: Probabilities for each quantized levels

Quantized Level	Probability
16	0.0078125
48	0.015625
80	0.015625
112	0.00390625
144	0.00390625
176	0.1484375
208	0.78125
240	0.0234375

Step 6: Construct the Huffman coding algorithm for cropped image at Step 4.(Do not use inbuilt algorithms.)

```
def huffman_tree_build(probabilities):
    heap = [Node(value, freq) for value, freq in probabilities]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]

def huffman_codes(node, code="", mapping=None):
    if mapping is None:
        mapping = {}

    if node is not None:
        if node.value is not None:
            mapping[node.value] = code
        huffman_codes(node.left, code + "0", mapping)
        huffman_codes(node.right, code + "1", mapping)

    return mapping


# Build Huffman tree
huffman_tree = huffman_tree_build(sorted_probabilities)

# Generate Huffman codes
huffman_mapping = huffman_codes(huffman_tree)
```

```
print("Huffman Mapping:")
print(huffman_mapping)
```

Huffman Mapping:

{80: '0000', 16: '00010', 112: '000110', 144: '000111', 48: '0010', 240: '0011', 176: '01', 208: '1'}

Table 2: Probabilities and Huffman codebook for each quantized levels

Quantized Level	Probability	Huffman codebook
16	0.0078125	00010
48	0.015625	0010
80	0.015625	0000
112	0.00390625	000110
144	0.00390625	000111
176	0.1484375	01
208	0.78125	1
240	0.0234375	0011

Step 7: Compress both cropped and original images using the algorithm and the codebook generated at step 6. You may round any intensity values outside the codebook, to the nearest intensity value in the codebook, where necessary.

Compression of cropped image

```
def huffman_encode(data, huffman_mapping):
    encoded_data = "".join(huffman_mapping[value] for value in
data.flatten())
    return encoded_data


'''
Encode the cropped image for Red, Green, Blue color channels
'''
# Take the cropped images in 3 colors
blue_cropped_image = cropped_image[:, :, 0]  # 0 for blue
green_cropped_image = cropped_image[:, :, 1]  # 1 for green
red_cropped_image = cropped_image[:, :, 2]  # 2 for red

# Quantize each cropped images into quantized levels
blue_quantized_img = quantize_matrix(blue_cropped_image,
quantization_levels)
green_quantized_img = quantize_matrix(green_cropped_image,
quantization_levels)
red_quantized_img = quantize_matrix(red_cropped_image,
quantization_levels)
```

```python
# Encode each color channels of cropped images using Huffman codes
red_encoded_data = huffman_encode(red_quantized_img, huffman_mapping)
green_encoded_data = huffman_encode(green_quantized_img,
huffman_mapping)
blue_encoded_data = huffman_encode(blue_quantized_img,
huffman_mapping)
```

Ex: Encode text for Red color cropped image

01000100010000000011000011100110011001100110011001101011101000100010000001010101010101011111010100100000010101010101111111010100100000010111010111111010101011111111111110101010101111111111111010111111111111111010111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111

Compression of full image

```python
'''
Encode the full image into Red, Green, Blue color channels
'''
# Take the separated color channels of full image
blue_image = image[:, :, 0]   # 0 for blue
green_image = image[:, :, 1]   # 1 for green
red_image = image[:, :, 2]   # 2 for red

# Quantize each full images
blue_full_img = quantize_matrix(blue_image, quantization_levels)
green_full_img = quantize_matrix(green_image, quantization_levels)
red_full_img = quantize_matrix(red_image, quantization_levels)

# Encode each color channels of full image using Huffman codes
red_encoded_full_image = huffman_encode(red_full_img, huffman_mapping)
green_encoded_full_image = huffman_encode(green_full_img,
huffman_mapping)
blue_encoded_full_image = huffman_encode(blue_full_img,
huffman_mapping)
```

Step 8: Save the compressed image into a text file.

```python
# Compressed encoded three color channels as text files
with open("Red compressed data.txt", "w") as text_file:   # Red
    text_file.write(red_encoded_full_image)

with open("Green compressed data.txt", "w") as text_file:   # Green
    text_file.write(green_encoded_full_image)
```

```
with open("Blue compressed data.txt", "w") as text_file:  # Blue
    text_file.write(blue_encoded_full_image)
```

Step 9: Compress the original image using Huffman encoding function in the Matlab tool box and save it into another text file.

```
import huffman

frequency_dict = defaultdict(int)
for value in flat_image:
    frequency_dict[value] += 1

# Generate Huffman codes
codebook = huffman.codebook(frequency_dict.items())
print(codebook)

#for full image

  blue_image = image[:, :, 0]  # 0 for blue
  green_image = image[:, :, 1]  # 1 for green
  red_image = image[:, :, 2]  # 2 for red

  blue_full_img = quantize_matrix(blue_image, quantization_levels)
  green_full_img = quantize_matrix(green_image, quantization_levels)
  red_full_img = quantize_matrix(red_image, quantization_levels)

  # Encode each color channel using Huffman codes
  red_encoded_full_image = huffman_encode(red_full_img, codebook)
  green_encoded_full_image = huffman_encode(green_full_img, codebook)
  blue_encoded_full_image = huffman_encode(blue_full_img,codebook)

with open("Red compressed data inbuilt huffman.txt", "w") as
text_file:
    text_file.write(red_encoded_full_image)

with open("Green compressed data inbuilt huffman.txt", "w") as
text_file:
    text_file.write(green_encoded_full_image)

with open("Blue compressed data inbuilt huffman.txt", "w") as
text_file:
    text_file.write(blue_encoded_full_image)
```

{16: '000', 80: '001', 48: '01', 208: '100', 176: '1010', 144: '10110', 112: '10111', 240: '11'}

Table 2: Huffman codebook generated from the inbuilt Huffman library in python

Quantized Level	Huffman codebook
16	000
48	01
80	001
112	10111
144	10110
176	1010
208	100
240	11

Step 10: Decompress the outputs at Step 8 and 9, by reading in the text files.

From step 8 ,

```
def huffman_decode(encoded_data, huffman_tree, original_shape):
    decoded_data = []
    current_node = huffman_tree

    for bit in encoded_data:
        if bit == '0':
            current_node = current_node.left
        else:
            current_node = current_node.right

        if current_node.value is not None:
            decoded_data.append(current_node.value)
            current_node = huffman_tree

    return np.array(decoded_data).reshape(original_shape)


'''
Decompressing the text files of constructed huffman code
'''
# Read the saved text files , encoded by constructed huffman code
content_red = read_compressed_data("Red compressed data.txt")
content_green = read_compressed_data("Green compressed data.txt")
content_blue = read_compressed_data("Blue compressed data.txt")

# Decode the data of the text file
red_decoded_data_txt = huffman_decode(content_red, huffman_tree,
red_full_img.shape)
green_decoded_data_txt = huffman_decode(content_green, huffman_tree,
green_full_img.shape)
```

```
blue_decoded_data_txt = huffman_decode(content_blue, huffman_tree,
blue_full_img.shape)

# Combine decoded data of all three channels and construct the colored
full image
decoded_image_from_txt = np.stack([blue_decoded_data_txt,
green_decoded_data_txt, red_decoded_data_txt], axis=-1)

plt.imshow(cv.cvtColor(decoded_image_from_txt, cv.COLOR_BGR2RGB))
plt.title("Decompressed image")
plt.axis('off')
plt.show()
```

Figure 6: Decompressed image encoded from the created Huffman code

From step 9,

```
'''
Decompressing the text files of build in huffman codebook in python
'''
# Read the saved text files , encoded by build in function in python
content_red_inbuilt = read_compressed_data("Red compressed data
inbuilt huffman.txt")
content_green_inbuilt = read_compressed_data("Green compressed data
inbuilt huffman.txt")
content_blue_inbuilt = read_compressed_data("Blue compressed data
inbuilt huffman.txt")

# Decode the data of the text files (build in huffman codebook)
red_decoded_txt_inbuilt = huffman_decode(content_red_inbuilt,
```

```
huffman_tree, red_full_img.shape)
green_decoded_txt_inbuilt = huffman_decode(content_green_inbuilt,
huffman_tree, green_full_img.shape)
blue_decoded_txt_inbuilt = huffman_decode(content_blue_inbuilt,
huffman_tree, blue_full_img.shape)

# Combine decoded data of all three channels and construct the colored
full image(buil in huffman codebook)
decoded_image_from_txt_inbuilt = np.stack([blue_decoded_txt_inbuilt,
green_decoded_txt_inbuilt, red_decoded_txt_inbuilt], axis=-1)


plt.imshow(cv.cvtColor(decoded_image_from_txt_inbuilt,
cv.COLOR_BGR2RGB))
plt.title("Decompressed image using inbuilt Huffman code in python")
plt.axis('off')
plt.show()
```

Figure 7: Decompressed image encoded from the inbuilt Huffman code

Step 11: Calculate the entropy of the Source

```
def calculate_entropy(probability_distribution):
    probabilities = np.array([probability for _, probability in
probability_distribution])
    entropy = -np.sum(probabilities * np.log2(probabilities))
    return entropy
# find entropy
entropy_rearranged_cropped = calculate_entropy(sorted_probabilities)
print(f"Entropy of the source: {entropy_rearranged_cropped}")
```

Output:

Entropy of the source: 1.118350552851062

Step 12: Evaluate the PSNR of

i. The original images
ii. The decompressed images

```
def calculate_psnr(original_image, compressed_image, max_pixel_value=255):
    mse = np.mean((original_image - compressed_image) ** 2)
    if mse == 0:
        return float('inf')  # PSNR is infinity when MSE is zero
    else:
        psnr_value = 10 * np.log10((max_pixel_value ** 2) / mse)
        return psnr_value
```

Original image:

```
# find PSNR original image
psnr = calculate_psnr(image, image)
print(f"PSNR original image: {psnr} dB")
```

Output: PSNR original image: inf dB

Decompressed image:

```
# find PSNR decompressed image
psnr_decompressed = calculate_psnr(image, decoded_image_from_txt)
print(f"PSNR decompressed image: {psnr_decompressed} dB")
```

Output: PSNR decompressed image: 28.88336034955035 dB

Outputs of codes:

For created Huffman codebook

```
C:\Users\Rashmi\AppData\Local\Programs\Python\Python311\python.exe "D:\SEM 7\EE596 - Video & image\My codes\build_huffman.py"
Probability of each symbol distribution:
[(208, 0.78125), (176, 0.1484375), (240, 0.0234375), (48, 0.015625), (80, 0.015625), (16, 0.0078125), (112, 0.00390625), (144, 0.00390625)]
Entropy of the source: 1.118350552851062
Huffman Mapping:
{80: '0000', 16: '00010', 112: '000110', 144: '000111', 48: '0010', 240: '0011', 176: '01', 208: '1'}
PSNR original image: inf dB
PSNR decompressed image: 28.88336034955035 dB
Entropy of the original image: 6.991851635352326
Entropy of the cropped image: 3.640650634143472
Entropy of the decompress image: 2.5747276696997448
Average length of the cropped image: 1.3828125 bits per symbol
Compression Ratio of cropped image: 5.785310734463277
Average length of the full image: 3.8341558641975313 bits per symbol
Compression Ratio of full image: 2.0865088127225517

Process finished with exit code 0
```

For in built Huffman codebook

```
C:\Users\Rashmi\AppData\Local\Programs\Python\Python311\python.exe "D:\SEM 7\EE596 - Video & image\My codes\huffman_inbuilt.py"
Huffman codebook
{16: '000', 48: '01', 80: '001', 112: '10111', 144: '10110', 176: '1010', 208: '100', 240: '11'}
Encoded text files saved
Average length of the cropped image Huffman inbuilt: 3.125 bits per symbol
Compression Ratio of cropped image Huffman inbuilt: 2.56
Average length of the full image Huffman inbuilt: 2.629859567901234 bits per symbol
Compression Ratio of full image Huffman inbuilt: 3.0419875257386537

Process finished with exit code 0
```

DISCUSSION

1. Calculate the entropy of,
   i. The original image
   ii. The cropped image
   iii. The decompressed images

1. Original image

```
#Calculate entropy for full image
red_original_flat_image = image[:, :, 2].flatten()  # flat the red
quantized image
probability_red_image =
list(calculate_probabilities(red_original_flat_image))
entropy_full_image = calculate_entropy(probability_red_image)
print(f"Entropy of the original image: {entropy_full_image}")
```

Entropy of the original image: 6.991851635352326

2. Cropped image

```
#calculate entropy of cropped image
red_cropped_flat_image = color_cropped_image.flatten()  # flat the red
quantized image
probability_red_cropped_image =
list(calculate_probabilities(red_cropped_flat_image))
entropy_cropped_image =
calculate_entropy(probability_red_cropped_image)
print(f"Entropy of the cropped image: {entropy_cropped_image}")
```

Entropy of the cropped image: 3.640650634143472

3. Decompressed image

```
# Calculate entropy for decompress image
red_decompress_flat_image = decoded_image_from_txt[:, :, 2].flatten()
# flat the red quantized image
probability_red_decompress_image =
list(calculate_probabilities(red_decompress_flat_image))
entropy_decompress_image =
calculate_entropy(probability_red_decompress_image)
print(f"Entropy of the decompress image: {entropy_decompress_image}")
```

Entropy of the decompress image: 2.5747276696997448

2. Calculate the average length of the cropped image.

```
# Calculate the average length of the cropped image
average_length_cropped = sum(prob * len(huffman_mapping[symbol]) for
symbol, prob in sorted_probabilities)
print(f"Average length of the cropped image: {average_length_cropped}
bits per symbol")
```

Average length of the cropped image: 1.3828125 bits per symbol

3.Compare the performance of your algorithm and inbuilt algorithm of Matlab by comparing the compression ratios, for cropped and original images.

- Compression ratio = Bits before compression / Bits after compression

```
def calculate_compression_ratio(average_length):
        # Calculate compression ratio
        compression_ratio = 8 / average_length

        return compression_ratio

# Calculate compression ratio of cropped image
compression_ratio_cropped =
calculate_compression_ratio(average_length_cropped)
print("Compression Ratio of cropped image:", compression_ratio_cropped)
```

<u>Prepared algorithm</u>

Compression Ratio of cropped image: 5.785310734463277

Compression Ratio of full image: 2.0865088127225517