DIGITAL IMAGE PROCESSING
ELL715
ASSIGNMENT 1

13 August 2023

Group Members

- Group Member 1: Bhavik Shankla (2020MT60873)
- Group Member 2: Ritika Soni (2020MT10838)
- Group Member 3: Sai Kiran Gunnala (2020MT60889)
- Group Member 4: Sai Niketh Varanasi (2020MT60895)
- Group Member 5: Yash Pravin Shirke (2020MT60986)

Question 1

Image denoted as $f(x, y)$ is transformed to image $g(x, y)$. $g(x, y)$ is 3 times larger along the y-axis and 2 times larger along the x-axis than $f(x, y)$. Also, $g(x, y)$ is at 6 units horizontal and 7 units vertical distance from $f(x, y)$. Write a code to do this. Show $f(x, y)$ and $g(x, y)$.

Compute $h(x, y)$, the third image, by rotating pixels of image 2, $g(x, y)$ by 75° counter clockwise. Write a code to do this. Show $f(x, y)$, $g(x, y)$ and $h(x, y)$.

Make your code generic enough so that you can do these transformations with any given values or images.

Solution:

To find $g(x, y)$ the magnified image I have used the resize() function. It takes scaling factors as input fx and fy and does linear interpolation to find get the scaled image.

Variables a, and b is taken as custom input where a represents the scaling factor along the $x - axis$ and b represent the scaling factor along the $y - axis$.

To find the translated image I have constructed a 2x3 matrix $M = [[1, 0, x], [0, 1, y]]$ multiplying this matrix with $g(x, y)$ gives us a translated image. For multiplying I used the warpAffine() function.

Variables c, and d are taken as custom input where c represents the distance to be translated along the $x - axis$ and d represents the distance to be translated along the $y - axis$.

To find the rotated image I have constructed a 2x2 matrix $M = [[cos(\theta), -sin(\theta)], [sin(\theta), cos(\theta)]]$ where θ represents the angle of rotation. To create this matrix I used an inbuilt function getRotation-Matrix2D(). The point of reference for rotation can be random and we can perform rotation from any point. So, we can take this also as input.

Variables e, f, g are taken as custom input where e represents the angle of rotation, and f, g represents the point/axis of rotation.

Python Code:

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


img = cv2.imread('baby.jpeg',1)
rows,cols,k= img.shape
img.shape


# scaling of image

#taking input of scaling factors along x-axis and y-axis
a=int(input("scaling along x-axis :"))
b=int(input("scaling along y-axis :"))

scaled_img=cv2.resize(img,(0,0),fx=a,fy=b,interpolation=cv2.INTER_LINEAR)

#used matplotlib to show results in the pdf itself
#cv.imshow() opens a window and shows the image
plt.imshow(cv2.cvtColor(scaled_img, cv2.COLOR_BGR2RGB))
# as opencv loads in BGR format by default, we want to show it in RGB.
plt.show()

scaled_img.shape


#translation of image

#taking input for translation along x-axis and y-axis
c=input("Enter shift along x-axis :")
d=input("Enter shift along y-axis :")

M = np.float32([[1,0,c],[0,1,d]])
#created a 2*3 matrix to carryout translation of image using matrix multiplication

translated_img = cv2.warpAffine(scaled_img,M,(cols,rows))

# as opencv loads in BGR format by default, we want to show it in RGB.
plt.imshow(cv2.cvtColor(translated_img, cv2.COLOR_BGR2RGB))
plt.show()


# rotation of image

#taking inputs of angle of rotation and reference coordinates for rotation
e=int(input("Enter angle of rotation :"))
f=int(input("Enter reference of rotation x-coordinate :"))
g=int(input("Enter reference of rotation y-coordinate :"))

M = cv2.getRotationMatrix2D((f,g),e,1)
# this function creates the rotation matrix [[cos(e), -sin(e)][cos(e), sine(e)]]
rotated_img = cv2.warpAffine(translated_img,M,(cols,rows))

plt.imshow(cv2.cvtColor(rotated_img, cv2.COLOR_BGR2RGB))
# as opencv loads in BGR format by default, we want to show it in RGB.
plt.show()
```

Question 2

Take an 8-bit gray scale image and perform the following operations using MATLAB,

- negative of the image, log and anti log of the image.
- Apply Gamma correction for $\gamma = 0.4$, 2.5, 10, 25, and 100.
- 2, 3, 4 power of image.
- Plot Bit-planes of image (show all the 8-plane images).
- Plot the histogram of the original image and apply Histogram equalization and plot the resulted image. Apply a transformation that highlights range $[120, 200]$ but preserves all other levels.

Solution:

Methodology

For the first part, we used direct transformations for getting the output image. For the second part, we had to follow the Power Transform law to get the gamma correction and the power of the images. For Bit Slicing the image, we used an bitget() inbuilt function for getting the specific bit of each pixel. For the last part, we saw it as selective Highlighting and was able to implement using Conditional statements.

MATLAB Code:

```
clear all;
close all;
clc;

RawImage =imread ('cameraman.tif'); % Reading Input Raw Image
[row,col]= size(RawImage);
L= 256;
% Upper Limit for the pixel value of the 8-bit Gray Scale Image

figure();

% Negative Transform of the RawImage

NegImage= uint8(zeros(row,col));
% Matrix containing the Negative Transform of the Image

for i=1:row
    for j= 1:col
        NegImage(i,j)= L- RawImage(i,j)-1;
        % Subtracting the Pixel value from the Maximum Value to get the Negative Transform
    end
end

% Log Transform of the RawImage

LogImage= uint8(zeros(row,col));
% Matrix containing the Log Transform of the Image

for i=1:row
    for j= 1:col
        LogImage(i,j)= log(double(RawImage(i,j))+1) * ((L - 1)/log(L));
        % Taking the log transform and multiplying it with the constant
    end
end

% AntiLog Transform of the RawImage
```

```
AntiLogImage= uint8(zeros(row,col));
% Matrix containing the AntiLog Transform of the Image

for i=1:row
    for j= 1:col
        AntiLogImage(i,j)= (exp(double(RawImage(i,j))) ^ (log(L) / (L-1))) - 1;
        % Taking the Antilog transform and multiplying it with the constant
    end
end

% Displaying the output

subplot(2, 2, 1); imshow(RawImage); title('\itRaw Image');
subplot(2, 2, 2); imshow(NegImage); title('\itNegative Transform');
subplot(2, 2, 3); imshow(LogImage); title('\itLog Transform');
subplot(2, 2, 4); imshow(AntiLogImage); title('\itAntiLog Transform');

figure();
%Gamma Correction using Power Law transform

Gamma= [0.4, 2.5, 10, 25, 100]; % Array with all Gamma Values
numImages= size(Gamma);

for i=1:numImages(1,2)
    GammaImage= uint8(zeros(row,col));
    C=(L-1)/((L-1)^ Gamma(1,i));
    % Calculating the constant that needs to be multiplied for the Power Law
       Transform
    for j=1:row
        for k=1:col
            GammaImage(j,k)= uint8(C*(double(RawImage(j,k))^ Gamma(1,i)));
            % Getting the pixel value after the Gamma Correction
        end
    end
    subplot(2, 3, i); imshow(GammaImage); title("\itGamma=" + Gamma(1,i));
    % Displaying the image
end

figure();
% Power of an Image

Pow = [2,3,4]; % Array with all Power Values
numImages= size(Pow);

for i=1:numImages(1,2)
    PowImage= uint8(zeros(row,col));
    C=(L-1)/((L-1)^ Pow(1,i));
    for j=1:row
        for k=1:col
            PowImage(j,k)= uint8(C*(double(RawImage(j,k))^ Pow(1,i)));
            % Calculating the Power of each pixel in the image
        end
    end
    subplot(1, 3, i); imshow(PowImage); title("\itPower=" + Pow(1,i));
end

figure();
for i=1:8
    BitPlane= bitget(RawImage,i);
    % Bitget function is used to get the ith bit of a number
    subplot(3,3,i);imshow(logical(BitPlane));title("Bit plane"+ i);
end
```

```
figure();

subplot(2,2,1);imshow(RawImage);title("Raw Image");
subplot(2,2,2);imhist(RawImage);title("Histogram of the Raw Image");
% Displaying the Histogram of the Raw Image

EqualizedImage= histeq(RawImage);
% Applying Histogram Equalization

subplot(2,2,3);imshow(EqualizedImage);
title("Equalized Image");
subplot(2,2,4);imhist(EqualizedImage);
title("Histogram of the Equalized Image");
% Displaying the Histogram of the Equalized Image

figure();

% Selective Highlighting of the RawImage

HighLightedImage= uint8(zeros(row,col));

%Initializing the range that needs to be highlighted

minVal=120;
maxVal=200;

for i=1:row
    for j= 1:col
        x= RawImage(i,j);
        if (x >= minVal) && (x <= maxVal)
        % Highlighting only those pixel values whose values lie in the range
            [120,200]
            HighLightedImage(i,j)= 255;
        else
           HighLightedImage(i,j)= x;
        end
    end
end

subplot(1, 2, 1); imshow(RawImage); title('\itRaw Image');
subplot(1, 2, 2); imshow(HighLightedImage); title('\itHighlighted Image');
```

Conclusion

Negative Transformation: In a negative transformation, each pixel value of the image is inverted, resulting in a reversal of brightness levels. Darker areas become lighter, and lighter areas become darker. This transformation can be useful for emphasizing features that were less visible in the original image or for creative purposes.

Logarithmic Transformation: Enhances details across a wide range of pixel intensities, making it useful for images with varying levels of brightness.

Anti-Logarithmic Transformation: Reverts an image back to its original appearance after applying a logarithmic transformation.

Gamma correction is used to adjust the brightness and contrast of an image to match how the human eye perceives light. A gamma-corrected image should exhibit improved contrast and more accurate representation of details in both shadows and highlights.

Bit slicing allows you to analyze and manipulate the individual bits of pixel values, which can provide insights into various aspects of an image, including features, noise, contrast, compression effects, and more. It's a versatile technique that can be used for visualization, analysis, and creative image manipulation.

Question 3

Use the test images.

- Create a function to calculate the histogram and then implement histogram equalization on the test image without using inbuilt MATLAB functions.
- Use the built-in function on the same image and compare with the histogram from step 1. Check mean squared error of both matrices.
- Apply adaptive histogram equalization (CLAHE) and compare with other mapped images.
  - Subplot the original image with the other 3 mapped images.
  - Plot the 3 histograms as well, keeping the axes same on each figure.
  - Mention the MSE from (b).

Solution:

Instead of employing a for loop to showcase histogram equalization on a set of five test images collectively, the code has been structured to individually apply the same sequence of operations on each image. This systematic approach ensures a clear presentation of the process for each image. The code begins by loading the selected image and, if necessary, converting it to gray-scale. A custom histogram equalization function is defined to calculate histograms and implement the equalization process manually. Subsequently, the custom function is executed on the current image. Similarly, the built-in histogram equalization function is employed to facilitate a comparative analysis. Mean Squared Error is computed to quantify dissimilarities between the outputs of the two methods. The resulting images before and after equalization are displayed, accompanied by their respective histograms. Additionally, the code explores the effects of Contrast Limited Adaptive Histogram Equalization (CLAHE) on the image. This structured approach provides an organized demonstration of histogram equalization's impact on image enhancement and distribution, while keeping each image's treatment separate for clarity.

MATLAB Code for Image 1:

```
% Load a test image
image_path = 'C:\Program Files\MATLAB\R2023a\toolbox\images\imdata\llama.jpg';
original_image = imread(image_path);
imshow(original_image)

original_image = rgb2gray(original_image); % Convert to grayscale if necessary

imshow(original_image)

% a. Calculate Histogram and Implement Histogram Equalization
% Function implementing histogram is written at the end of the file
equalized_image_custom = customHistogramEqualization(original_image);

% Calculate histogram of the original image
histogram_original = imhist(original_image);

% b. Use Built-in Function and Compare Histograms
equalized_image_builtin = histeq(original_image, 256);

% Calculate Mean Squared Error (MSE)
mse_custom_vs_builtin = immse(equalized_image_custom, equalized_image_builtin);

% c. Apply Adaptive Histogram Equalization (CLAHE)
clahe_image = adapthisteq(original_image, 'ClipLimit', 0.02);

% Display original and equalized images side by side
subplot(2, 2, 1); imshow(original_image); title('Original Image');
subplot(2, 2, 2); imshow(equalized_image_custom); title('Custom Histogram
    Equalization');
```

```
subplot(2, 2, 3); imshow(equalized_image_builtin); title('Built-in Histogram Equalization');
subplot(2, 2, 4); imshow(clahe_image); title('CLAHE');

% Calculate Mean Squared Error (MSE) for CLAHE
mse_clahe_vs_builtin = immse(clahe_image, equalized_image_builtin);

% Set the width of the entire figure
figure_width = 1200;  % Adjust this value as needed
figure_height = 400;  % Adjust this value as needed
figure('Position', [100, 100, figure_width, figure_height]);

% Display histograms
subplot(1, 4, 1); bar(0:255, histogram_original); title('Original Histogram');
subplot(1, 4, 2); bar(0:255, imhist(equalized_image_custom)); title('Custom Histogram');
subplot(1, 4, 3); bar(0:255, imhist(equalized_image_builtin)); title('Built-in Histogram');
subplot(1, 4, 4); bar(0:255, imhist(clahe_image)); title('CLAHE Histogram');

disp(['MSE between Custom and Built-in Histogram Equalization: ', num2str(mse_custom_vs_builtin)]);
disp(['MSE between CLAHE and Built-in Histogram Equalization: ', num2str(mse_clahe_vs_builtin)]);

function equalized_image = customHistogramEqualization(image)
    [h, w] = size(image);
    num_pixels = h * w;

    histogram = zeros(256, 1);
    for i = 1:h
        for j = 1:w
            pixel_value = image(i, j);
            histogram(pixel_value + 1) = histogram(pixel_value + 1) + 1;
        end
    end
    cumulative_histogram = cumsum(histogram) / num_pixels;
    equalized_image = uint8(255 * cumulative_histogram(double(image) + 1));
end
```

MATLAB Code for Image 2:

```
% Load a test image
image_path = "C:\Program Files\MATLAB\R2023a\toolbox\images\imdata\flamingos.jpg";
original_image = imread(image_path);
imshow(original_image)

original_image = rgb2gray(original_image); % Convert to grayscale if necessary

imshow(original_image)

% a. Calculate Histogram and Implement Histogram Equalization
% Function implementing histogram is written at the end of the file
equalized_image_custom = customHistogramEqualization(original_image);

% Calculate histogram of the original image
histogram_original = imhist(original_image);

% b. Use Built-in Function and Compare Histograms
equalized_image_builtin = histeq(original_image, 256);

% Calculate Mean Squared Error (MSE)
```

```
mse_custom_vs_builtin = immse(equalized_image_custom, equalized_image_builtin);

% c. Apply Adaptive Histogram Equalization (CLAHE)
clahe_image = adapthisteq(original_image, 'ClipLimit', 0.02);

% Display original and equalized images side by side
subplot(2, 2, 1); imshow(original_image); title('Original Image');
subplot(2, 2, 2); imshow(equalized_image_custom); title('Custom Histogram Equalization');
subplot(2, 2, 3); imshow(equalized_image_builtin); title('Built-in Histogram Equalization');
subplot(2, 2, 4); imshow(clahe_image); title('CLAHE');

% Calculate Mean Squared Error (MSE) for CLAHE
mse_clahe_vs_builtin = immse(clahe_image, equalized_image_builtin);

% Set the width of the entire figure
figure_width = 1200;  % Adjust this value as needed
figure_height = 400;  % Adjust this value as needed
figure('Position', [100, 100, figure_width, figure_height]);

% Display histograms
subplot(1, 4, 1); bar(0:255, histogram_original); title('Original Histogram');
subplot(1, 4, 2); bar(0:255, imhist(equalized_image_custom)); title('Custom Histogram');
subplot(1, 4, 3); bar(0:255, imhist(equalized_image_builtin)); title('Built-in Histogram');
subplot(1, 4, 4); bar(0:255, imhist(clahe_image)); title('CLAHE Histogram');

disp(['MSE between Custom and Built-in Histogram Equalization: ', num2str(mse_custom_vs_builtin)]);
disp(['MSE between CLAHE and Built-in Histogram Equalization: ', num2str(mse_clahe_vs_builtin)]);

function equalized_image = customHistogramEqualization(image)
    [h, w] = size(image);
    num_pixels = h * w;

    histogram = zeros(256, 1);
    for i = 1:h
        for j = 1:w
            pixel_value = image(i, j);
            histogram(pixel_value + 1) = histogram(pixel_value + 1) + 1;
        end
    end
    cumulative_histogram = cumsum(histogram) / num_pixels;
    equalized_image = uint8(255 * cumulative_histogram(double(image) + 1));
end
```

MATLAB Code for Image 3:

```
% Load a test image
image_path = "C:\Program Files\MATLAB\R2023a\toolbox\images\imdata\lighthouse.png";
original_image = imread(image_path);
imshow(original_image)

original_image = rgb2gray(original_image); % Convert to grayscale if necessary

imshow(original_image)

% a. Calculate Histogram and Implement Histogram Equalization
% Function implementing histogram is written at the end of the file
```

```
equalized_image_custom = customHistogramEqualization(original_image);

% Calculate histogram of the original image
histogram_original = imhist(original_image);

% b. Use Built-in Function and Compare Histograms
equalized_image_builtin = histeq(original_image, 256);

% Calculate Mean Squared Error (MSE)
mse_custom_vs_builtin = immse(equalized_image_custom, equalized_image_builtin);

% c. Apply Adaptive Histogram Equalization (CLAHE)
clahe_image = adapthisteq(original_image, 'ClipLimit', 0.02);

% Create a new figure with larger dimensions
figure('Position', [100, 100, 1200, 800]);

% Display original and equalized images side by side
subplot(2, 2, 1); imshow(original_image); title('Original Image');
subplot(2, 2, 2); imshow(equalized_image_custom); title('Custom Histogram Equalization');
subplot(2, 2, 3); imshow(equalized_image_builtin); title('Built-in Histogram Equalization');
subplot(2, 2, 4); imshow(clahe_image); title('CLAHE');

% Calculate Mean Squared Error (MSE) for CLAHE
mse_clahe_vs_builtin = immse(clahe_image, equalized_image_builtin);

% Set the width of the entire figure
figure_width = 1200;  % Adjust this value as needed
figure_height = 400;  % Adjust this value as needed
figure('Position', [100, 100, figure_width, figure_height]);

% Display histograms
subplot(1, 4, 1); bar(0:255, histogram_original); title('Original Histogram');
subplot(1, 4, 2); bar(0:255, imhist(equalized_image_custom)); title('Custom Histogram');
subplot(1, 4, 3); bar(0:255, imhist(equalized_image_builtin)); title('Built-in Histogram');
subplot(1, 4, 4); bar(0:255, imhist(clahe_image)); title('CLAHE Histogram');

disp(['MSE between Custom and Built-in Histogram Equalization: ', num2str(mse_custom_vs_builtin)]);
disp(['MSE between CLAHE and Built-in Histogram Equalization: ', num2str(mse_clahe_vs_builtin)]);

function equalized_image = customHistogramEqualization(image)
    [h, w] = size(image);
    num_pixels = h * w;

    histogram = zeros(256, 1);
    for i = 1:h
        for j = 1:w
            pixel_value = image(i, j);
            histogram(pixel_value + 1) = histogram(pixel_value + 1) + 1;
        end
    end
    cumulative_histogram = cumsum(histogram) / num_pixels;
    equalized_image = uint8(255 * cumulative_histogram(double(image) + 1));
end
```

MATLAB Code for Image 4:

```
% Load a test image
image_path = "C:\Program Files\MATLAB\R2023a\toolbox\images\imdata\wagon.jpg";
original_image = imread(image_path);
imshow(original_image)

original_image = rgb2gray(original_image); % Convert to grayscale if necessary

imshow(original_image)

% a. Calculate Histogram and Implement Histogram Equalization
% Function implementing histogram is written at the end of the file
equalized_image_custom = customHistogramEqualization(original_image);

% Calculate histogram of the original image
histogram_original = imhist(original_image);

% b. Use Built-in Function and Compare Histograms
equalized_image_builtin = histeq(original_image, 256);

% Calculate Mean Squared Error (MSE)
mse_custom_vs_builtin = immse(equalized_image_custom, equalized_image_builtin);

% c. Apply Adaptive Histogram Equalization (CLAHE)
clahe_image = adapthisteq(original_image, 'ClipLimit', 0.02);

% Create a new figure with larger dimensions
figure('Position', [100, 100, 1200, 800]);

% Display original and equalized images side by side
subplot(2, 2, 1); imshow(original_image); title('Original Image');
subplot(2, 2, 2); imshow(equalized_image_custom); title('Custom Histogram
    Equalization');
subplot(2, 2, 3); imshow(equalized_image_builtin); title('Built-in Histogram
    Equalization');
subplot(2, 2, 4); imshow(clahe_image); title('CLAHE');

% Calculate Mean Squared Error (MSE) for CLAHE
mse_clahe_vs_builtin = immse(clahe_image, equalized_image_builtin);

% Set the width of the entire figure
figure_width = 1200;  % Adjust this value as needed
figure_height = 400;  % Adjust this value as needed
figure('Position', [100, 100, figure_width, figure_height]);

% Display histograms
subplot(1, 4, 1); bar(0:255, histogram_original); title('Original Histogram');
subplot(1, 4, 2); bar(0:255, imhist(equalized_image_custom)); title('Custom
    Histogram');
subplot(1, 4, 3); bar(0:255, imhist(equalized_image_builtin)); title('Built-in
    Histogram');
subplot(1, 4, 4); bar(0:255, imhist(clahe_image)); title('CLAHE Histogram');

disp(['MSE between Custom and Built-in Histogram Equalization: ',
    num2str(mse_custom_vs_builtin)]);
disp(['MSE between CLAHE and Built-in Histogram Equalization: ',
    num2str(mse_clahe_vs_builtin)]);

function equalized_image = customHistogramEqualization(image)
    [h, w] = size(image);
    num_pixels = h * w;

    histogram = zeros(256, 1);
```

```
    for i = 1:h
        for j = 1:w
            pixel_value = image(i, j);
            histogram(pixel_value + 1) = histogram(pixel_value + 1) + 1;
        end
    end
    cumulative_histogram = cumsum(histogram) / num_pixels;
    equalized_image = uint8(255 * cumulative_histogram(double(image) + 1));
end
```

MATLAB Code for Image 5:

```
% Load a test image
image_path = "C:\Program
    Files\MATLAB\R2023a\toolbox\images\imdata\sherlock.jpg";
original_image = imread(image_path);
imshow(original_image)

original_image = rgb2gray(original_image); % Convert to grayscale if necessary

imshow(original_image)

% a. Calculate Histogram and Implement Histogram Equalization
% Function implementing histogram is written at the end of the file
equalized_image_custom = customHistogramEqualization(original_image);

% Calculate histogram of the original image
histogram_original = imhist(original_image);

% b. Use Built-in Function and Compare Histograms
equalized_image_builtin = histeq(original_image, 256);

% Calculate Mean Squared Error (MSE)
mse_custom_vs_builtin = immse(equalized_image_custom, equalized_image_builtin);

% c. Apply Adaptive Histogram Equalization (CLAHE)
clahe_image = adapthisteq(original_image, 'ClipLimit', 0.02);

% Display original and equalized images side by side
subplot(2, 2, 1); imshow(original_image); title('Original Image');
subplot(2, 2, 2); imshow(equalized_image_custom); title('Custom Histogram
    Equalization');
subplot(2, 2, 3); imshow(equalized_image_builtin); title('Built-in Histogram
    Equalization');
subplot(2, 2, 4); imshow(clahe_image); title('CLAHE');

% Calculate Mean Squared Error (MSE) for CLAHE
mse_clahe_vs_builtin = immse(clahe_image, equalized_image_builtin);

% Set the width of the entire figure
figure_width = 1200;  % Adjust this value as needed
figure_height = 400;  % Adjust this value as needed
figure('Position', [100, 100, figure_width, figure_height]);

% Display histograms
subplot(1, 4, 1); bar(0:255, histogram_original); title('Original Histogram');
subplot(1, 4, 2); bar(0:255, imhist(equalized_image_custom)); title('Custom
    Histogram');
subplot(1, 4, 3); bar(0:255, imhist(equalized_image_builtin)); title('Built-in
    Histogram');
subplot(1, 4, 4); bar(0:255, imhist(clahe_image)); title('CLAHE Histogram');

disp(['MSE between Custom and Built-in Histogram Equalization: ',
```

```
    num2str(mse_custom_vs_builtin)]);
disp(['MSE between CLAHE and Built-in Histogram Equalization: ',
    num2str(mse_clahe_vs_builtin)]);

function equalized_image = customHistogramEqualization(image)
    [h, w] = size(image);
    num_pixels = h * w;

    histogram = zeros(256, 1);
    for i = 1:h
        for j = 1:w
            pixel_value = image(i, j);
            histogram(pixel_value + 1) = histogram(pixel_value + 1) + 1;
        end
    end
    cumulative_histogram = cumsum(histogram) / num_pixels;
    equalized_image = uint8(255 * cumulative_histogram(double(image) + 1));
end
```

Question 4

Design image filters to get better understanding and presentation of filtering.

- Design images using identify_filter, blur_filter, large_blur_filter, sobel_filter, laplacian_filter, and high_pass_filter.
- Use these filters to make high-frequency and low-frequency images.
- Can you construct the hybrid image by combining the filtered high-frequency and low-frequency images?
- Experiment with your own image and your loved pet's image.

Solution:

Image Details - 1200 x 1200 resolution cat image

Methodology

We design several filters as follows:

- For identity filter we convolve the image with identity matrix to obtain the same image.
- For blurring the image we use Gaussian blurring with kernel size (21,21).
- For large_blur filter we use a larger kernel for achieving more smooth image, hence we use Gaussian blurring with kernel size (101,101).
- For sobel filter for edge detection I tried computing the gradients in x direction, y direction and 45° angle, we obtained the sharpest image at 45° angle due to the nature of edges in the image.
- For Laplacian Filter we use cv2.laplacian() filter.
- For high pass filter we subtract the blur image from original image.

Python Code:

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

path = r'./cat_image.jpg' # Path to Image

image = cv2.imread(path) # Reading the image
plt.imshow(image) # Displaying the Image

print("Resolution of image is" , image.shape)

# cv2.imshow(window_name, image)

identity_kernel = np.array([[0, 0, 0],
                        [0, 1, 0],
                        [0, 0, 0]])  # Creating Identity Matrix

identity_filtered_image = cv2.filter2D(image, ddepth=-1,
    kernel=identity_kernel) # Passing image through identity filter
plt.imshow(identity_filtered_image) # Displaying the Image

# Defining Kernel size for blurring
ksize = (21, 21)

# Using cv2.blur() method
```

```
blur_image = cv2.GaussianBlur(image, ksize, 0)
print(blur_image.shape)
plt.imshow(blur_image)
# Displaying the image

# Kernel size for large_blur would be higher so that image would be more smooth
ksize1 = (101, 101)

# Using cv2.blur() method
blur_image1 = cv2.GaussianBlur(image, ksize1,0)
print(blur_image1.shape)
plt.imshow(blur_image1)
# Displaying the image

# Sobel filter for edge detection
sobel_image = cv2.Sobel(image,cv2.CV_64F,1,1,ksize=21)
# Calculated gradient direction at 45* angle as it gave the most acccurate
    results as
# compared to gradient purely in x and y direction

plt.imshow(sobel_image)

laplacian_image = cv2.Laplacian(image,cv2.CV_64F,5) # Applying Laplacian
    filter with kernel size = 5
print(laplacian_image.shape)
plt.imshow(laplacian_image)

hpf = cv2.subtract(image,cv2.GaussianBlur(image, (21, 21),7)) # The high pass
    filter is created by subtracting the Gaussian Blurred image from original
    image
plt.imshow(hpf)


# Low frequency image can be obtained by applying the Gaussian Blur on the
    original filter

# Using cv2.blur() method
low_frequency_image = cv2.GaussianBlur(image, ksize1, 0)
plt.imshow(low_frequency_image)
# Displaying the image

# High frequency image can be  created in many ways, here I am subtracting the
    lareg_blur image from original image

high_frequency_image = cv2.subtract(image , cv2.GaussianBlur(image, ksize1, 0))
plt.imshow(high_frequency_image)


# Creating a hybrid image by adding the low frequency and high frequency images
plt.imshow(cv2.add(low_frequency_image,high_frequency_image))


path = r'./Human_image.enc' # Path to Image

image = cv2.imread(path) # Reading the image
plt.imshow(image) # Displaying the Image

print("Resolution of image is" , image.shape)

identity_kernel = np.array([[0, 0, 0],
                      [0, 1, 0],
                      [0, 0, 0]])  # Creating Identity Matrix

```

```
identity_filtered_image = cv2.filter2D(image, ddepth=-1, kernel=identity_kernel) # Passing image through identity filter
plt.imshow(identity_filtered_image) # Displaying the Image


# Defining Kernel size for blurring
ksize = (21, 21)

# Using cv2.blur() method
blur_image = cv2.GaussianBlur(image, ksize, 0)
print(blur_image.shape)
plt.imshow(blur_image)
# Displaying the image


# Kernel size for large_blur would be higher so that image would be more smooth
ksize1 = (101, 101)

# Using cv2.blur() method
blur_image1 = cv2.GaussianBlur(image, ksize1,0)
print(blur_image1.shape)
plt.imshow(blur_image1)
# Displaying the image

# Sobel filter for edge detection
sobel_image = cv2.Sobel(image,cv2.CV_64F,1,1,ksize=21)
# Calculated gradient direction at 45* angle as it gave the most acccurate results as
# compared to gradient purely in x and y direction

plt.imshow(sobel_image)


laplacian_image = cv2.Laplacian(image,cv2.CV_64F,5) # Applying Laplacian filter with kernel size = 5
print(laplacian_image.shape)
plt.imshow(laplacian_image)


hpf = cv2.subtract(image,cv2.GaussianBlur(image, (21, 21),7)) # The high pass filter is created by subtracting the Gaussian Blurred image from original image
plt.imshow(hpf)

# Low frequency image can be obtained by applying the Gaussian Blur on the original filter

# Using cv2.blur() method
low_frequency_image = cv2.GaussianBlur(image, ksize1, 0)
plt.imshow(low_frequency_image)
# Displaying the image


# High frequency image can be  created in many ways, here I am subtracting the lareg_blur image from original image

high_frequency_image = cv2.subtract(image , cv2.GaussianBlur(image, ksize1, 0))
plt.imshow(high_frequency_image)

# Creating a hybrid image by adding the low frequency and high frequency images
plt.imshow(cv2.add(low_frequency_image,high_frequency_image))
\begin{lstlisting}
```

Conclusion

The low frequency images can be obtained by applying the blur_filter or large_blur_filter and the high frequency filter can be obtained by applying sobel_filter, laplacian_filter or subtracting the low frequency image from the original image. The hybrid image obtained by adding the low frequency and high frequency image is slightly hazy as compared to the original image. The sharpest high frequency image was obtained by using the sobel filter.

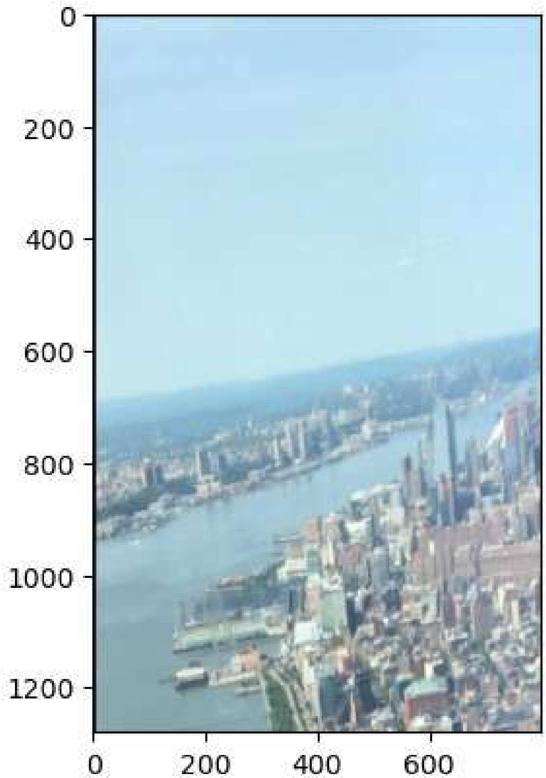
```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
import math

img = cv2.imread('baby.jpeg',1)
rows,cols,k= img.shape
img.shape
```

```
(1280, 800, 3)
```

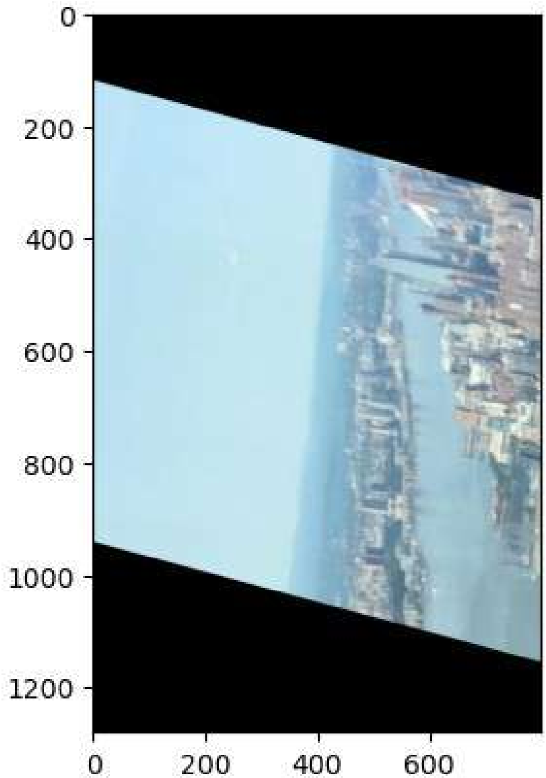
```python
# scaling of image

#taking input of scaling factors along x-axis and y-axis
a=int(input("scaling along x-axis :"))
b=int(input("scaling along y-axis :"))

scaled_img=cv2.resize(img,
(0,0),fx=a,fy=b,interpolation=cv2.INTER_LINEAR)

#used matplotlib to show results in the pdf itself
#cv.imshow() opens a window and shows the image
plt.imshow(cv2.cvtColor(scaled_img, cv2.COLOR_BGR2RGB))
# as opencv loads in BGR format by default, we want to show it in RGB.
plt.show()


scaled_img.shape
```

```
scaling along x-axis :2
scaling along y-axis :3
```


```
(3840, 1600, 3)

#translation of image

#taking input for translation along x-axis and y-axis
c=input("Enter shift along x-axis :")
d=input("Enter shift along y-axis :")

M = np.float32([[1,0,c],[0,1,d]]) #created a 2*3 matrix to carryout
translation of image using matrix multiplication

translated_img = cv2.warpAffine(scaled_img,M,(cols,rows))

# as opencv loads in BGR format by default, we want to show it in RGB.
plt.imshow(cv2.cvtColor(translated_img, cv2.COLOR_BGR2RGB))
plt.show()

Enter shift along x-axis :6
Enter shift along y-axis :7
```



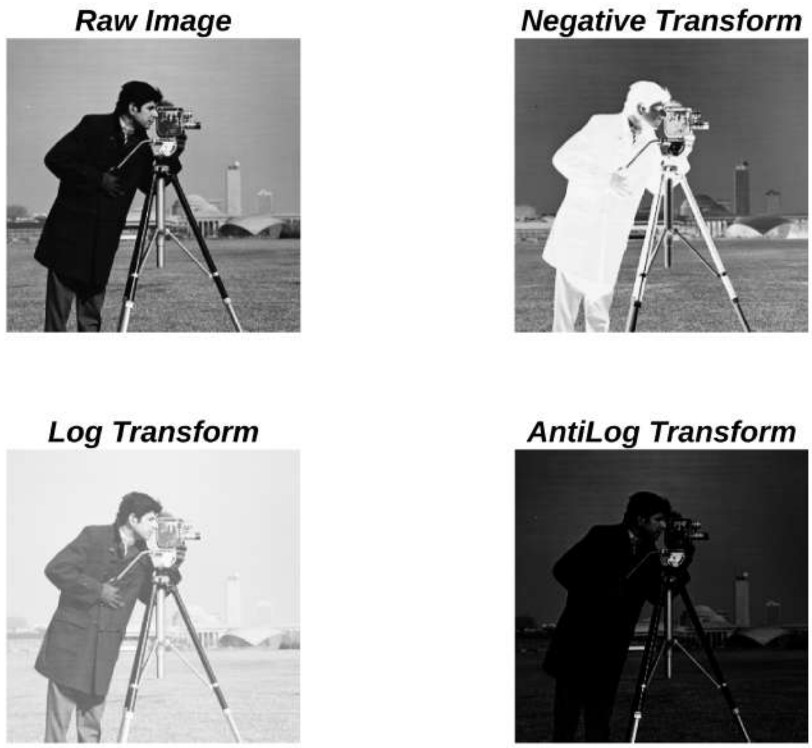

```
# rotation of image

#taking inputs of angle of rotation and reference coordinates for
rotation
e=int(input("Enter angle of rotation :"))
f=int(input("Enter reference of rotation x-coordinate :"))
g=int(input("Enter reference of rotation y-coordinate :"))

#here I performed the rotation along the midpoint of the image
M = cv2.getRotationMatrix2D((f,g),e,1) # this function creates the
rotation matrix [[cos(e), -sin(e)][cos(e), sine(e)]]
rotated_img = cv2.warpAffine(translated_img,M,(cols,rows))

plt.imshow(cv2.cvtColor(rotated_img, cv2.COLOR_BGR2RGB))
# as opencv loads in BGR format by default, we want to show it in RGB.
plt.show()

Enter angle of rotation :75
Enter reference of rotation x-coordinate :400
Enter reference of rotation y-coordinate :640
```


0
200
400
600
800
1000
1200
0
200
400
600

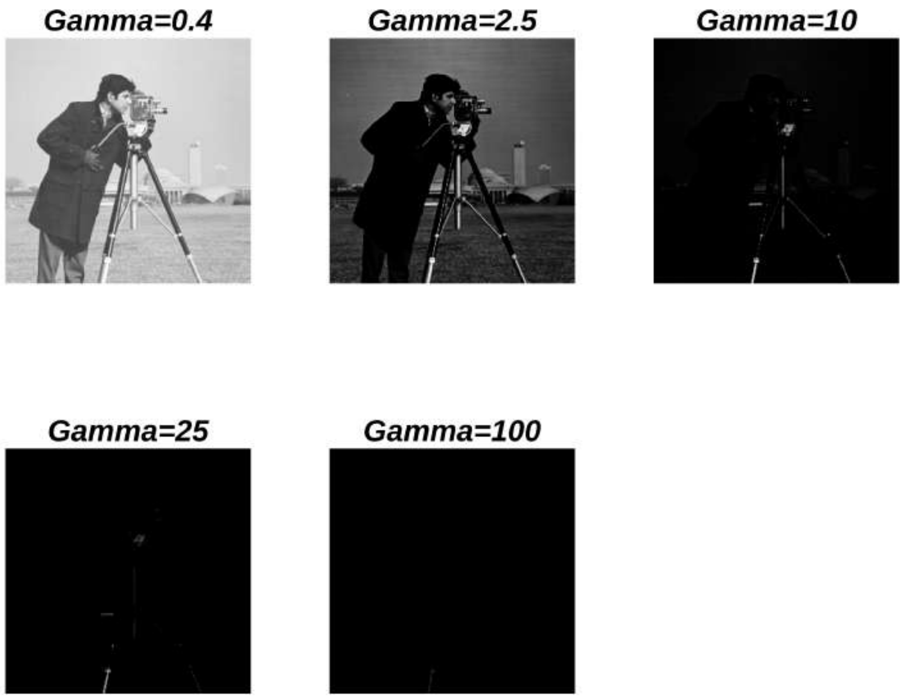
```
clear all;
close all;
clc;

RawImage =imread ('cameraman.tif'); % Reading Input Raw Image
[row,col]= size(RawImage);
L= 256;                              % Upper Limit for the pixel value of the
8-bit Gray Scale Image
```

```
figure();

% Negative Transform of the RawImage

NegImage= uint8(zeros(row,col));     % Matrix containing the Negative
Transform of the Image

for i=1:row
    for j= 1:col
        NegImage(i,j)= L- RawImage(i,j)-1;      % Subtracting the Pixel
value from the Maximum Value to get the Negative Transform
    end
end

% Log Transform of the RawImage

LogImage= uint8(zeros(row,col));     % Matrix containing the Log Transform
of the Image

for i=1:row
    for j= 1:col
        LogImage(i,j)= log(double(RawImage(i,j))+1) * ((L - 1)/log(L)); %
Taking the log transform and multiplying it with the constant
    end
end

% AntiLog Transform of the RawImage

AntiLogImage= uint8(zeros(row,col));  % Matrix containing the AntiLog
Transform of the Image

for i=1:row
    for j= 1:col
        AntiLogImage(i,j)= (exp(double(RawImage(i,j))) ^ (log(L) / (L-1))) -
1; % Taking the Antilog transform and multiplying it with the constant
    end
end

% Displaying the output
```

```
subplot(2, 2, 1); imshow(RawImage); title('\itRaw Image');
subplot(2, 2, 2); imshow(NegImage); title('\itNegative Transform');
subplot(2, 2, 3); imshow(LogImage); title('\itLog Transform');
subplot(2, 2, 4); imshow(AntiLogImage); title('\itAntiLog Transform');
```

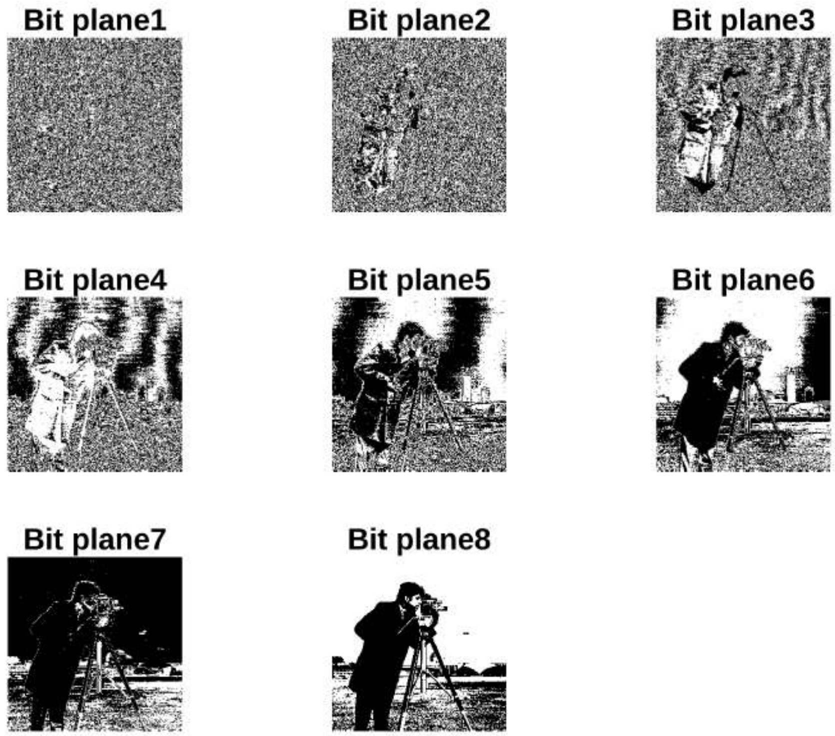



```
figure();
%Gamma Correction using Power Law transform

Gamma= [0.4, 2.5, 10, 25, 100]; % Array with all Gamma Values
numImages= size(Gamma);

for i=1:numImages(1,2)
    GammaImage= uint8(zeros(row,col));
    C=(L-1)/((L-1)^ Gamma(1,i));  % Calculating the constant that needs to
be multiplied for the Power Law Transform
    for j=1:row
        for k=1:col
            GammaImage(j,k)= uint8(C*(double(RawImage(j,k))^ Gamma(1,i))); %
Getting the pixel value after the Gamma Correction
        end
    end
    subplot(2, 3, i); imshow(GammaImage); title("\itGamma=" + Gamma(1,i)); %
Displaying the image
end
```

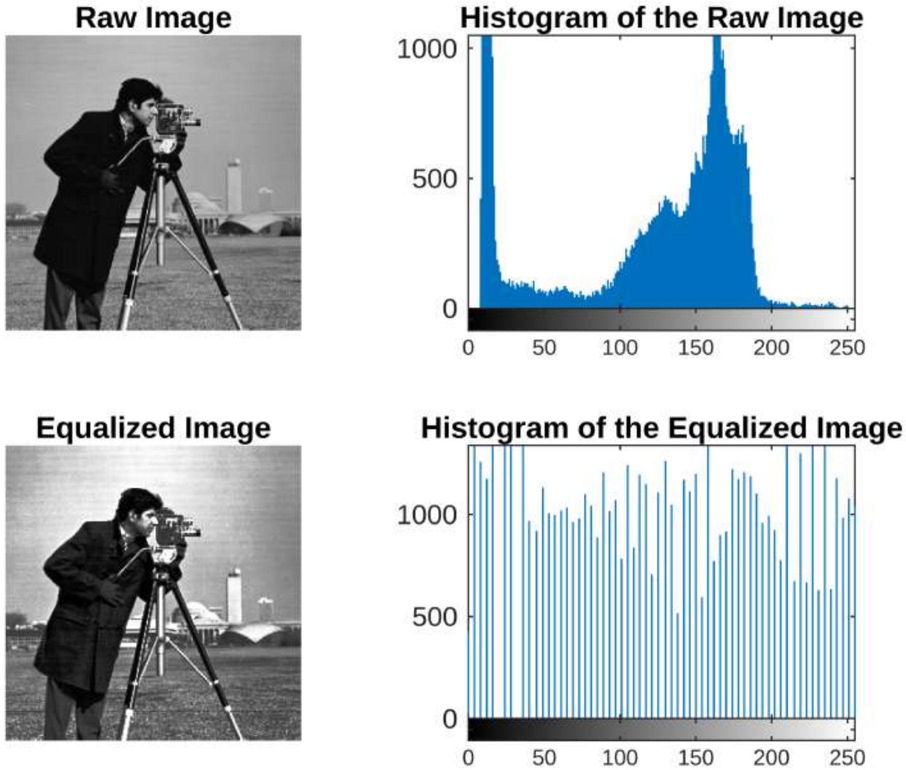



```
figure();
% Power of an Image

Pow = [2,3,4]; % Array with all Power Values
numImages= size(Pow);

for i=1:numImages(1,2)
    PowImage= uint8(zeros(row,col));
    C=(L-1)/((L-1)^ Pow(1,i));
    for j=1:row
        for k=1:col
            PowImage(j,k)= uint8(C*(double(RawImage(j,k))^ Pow(1,i)));  % Calculating the Power of each pixel in the image
        end
    end
    subplot(1, 3, i); imshow(PowImage); title("\itPower=" + Pow(1,i));
end
```


```
figure();
for i=1:8
    BitPlane= bitget(RawImage,i);     % Bitget function is used to get the
ith bit of a number
    subplot(3,3,i);imshow(logical(BitPlane));title("Bit plane"+ i);
end
```

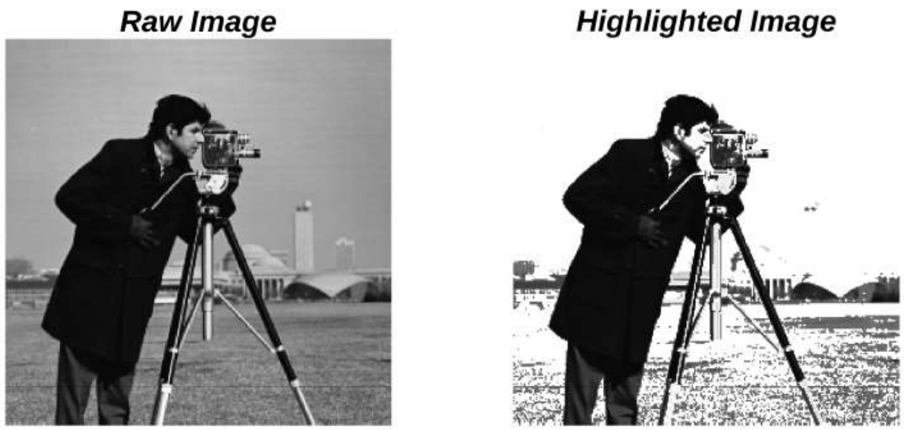



```
figure();

subplot(2,2,1);imshow(RawImage);title("Raw Image");
subplot(2,2,2);imhist(RawImage);title("Histogram of the Raw Image"); %
Displaying the Histogram of the Raw Image

EqualizedImage= histeq(RawImage);    % Applying Histogram Equalization

subplot(2,2,3);imshow(EqualizedImage);title("Equalized Image");
subplot(2,2,4);imhist(EqualizedImage);title("Histogram of the Equalized
Image"); % Displaying the Histogram of the Equalized Image
```




```
figure();

% Selective Highlighting of the RawImage

HighLightedImage= uint8(zeros(row,col));

%Initializing the range that needs to be highlighted

minVal=120;
maxVal=200;

for i=1:row
    for j= 1:col
        x= RawImage(i,j);
        if (x >= minVal) && (x <= maxVal)    % Highlighting only those pixel
values whose values lie in the range [120,200]
            HighLightedImage(i,j)= 255;
        else
           HighLightedImage(i,j)= x;
        end
    end
end
```

```
subplot(1, 2, 1); imshow(RawImage); title('\itRaw Image');
subplot(1, 2, 2); imshow(HighLightedImage); title('\itHighlighted Image');
```



```
% Load a test image
image_path = 'C:\Program Files\MATLAB\R2023a\toolbox\images\imdata\llama.jpg';
original_image = imread(image_path);
imshow(original_image)
```

```
original_image = rgb2gray(original_image); % Convert to grayscale if necessary
```

```
imshow(original_image)
```

```
% a. Calculate Histogram and Implement Histogram Equalization
% Function implementing histogram is written at the end of the file
equalized_image_custom = customHistogramEqualization(original_image);
```

```
% Calculate histogram of the original image
histogram_original = imhist(original_image);
```

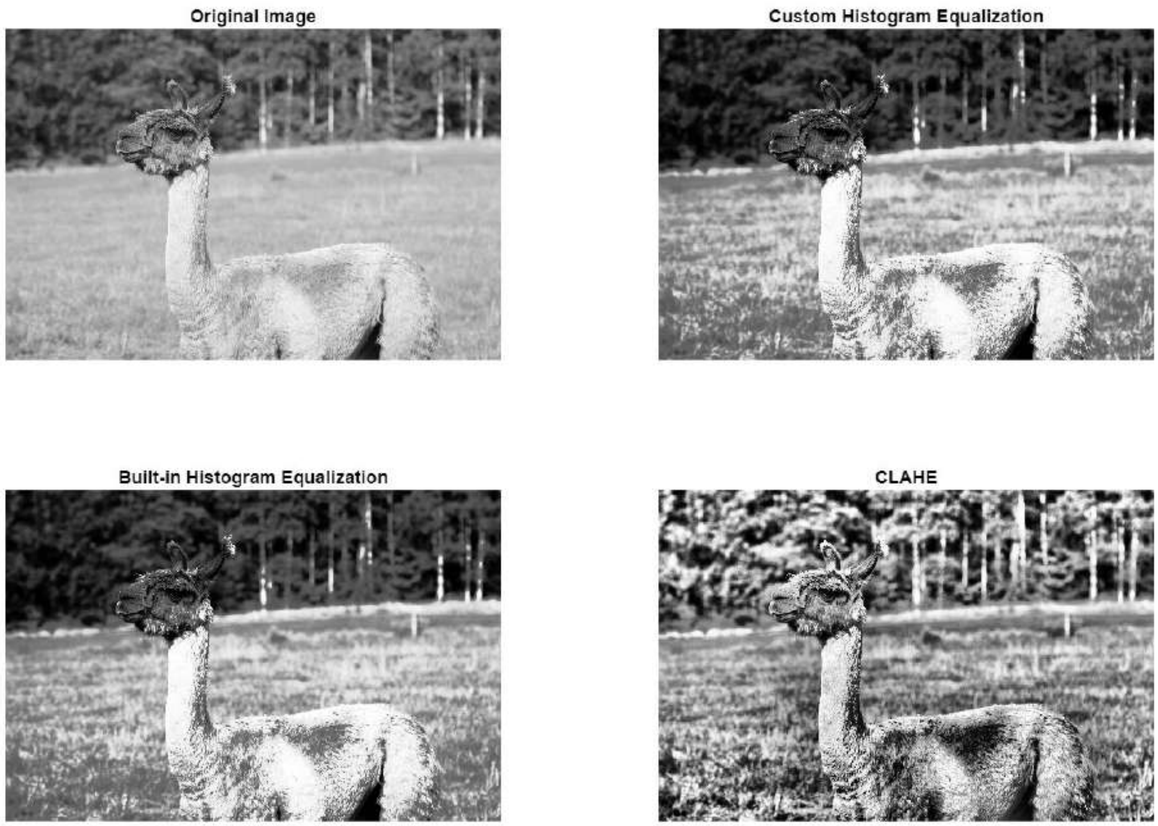
```
% b. Use Built-in Function and Compare Histograms
equalized_image_builtin = histeq(original_image, 256);
```

```
% Calculate Mean Squared Error (MSE)
mse_custom_vs_builtin = immse(equalized_image_custom, equalized_image_builtin);
```

```
% c. Apply Adaptive Histogram Equalization (CLAHE)
clahe_image = adapthisteq(original_image, 'ClipLimit', 0.02);
```

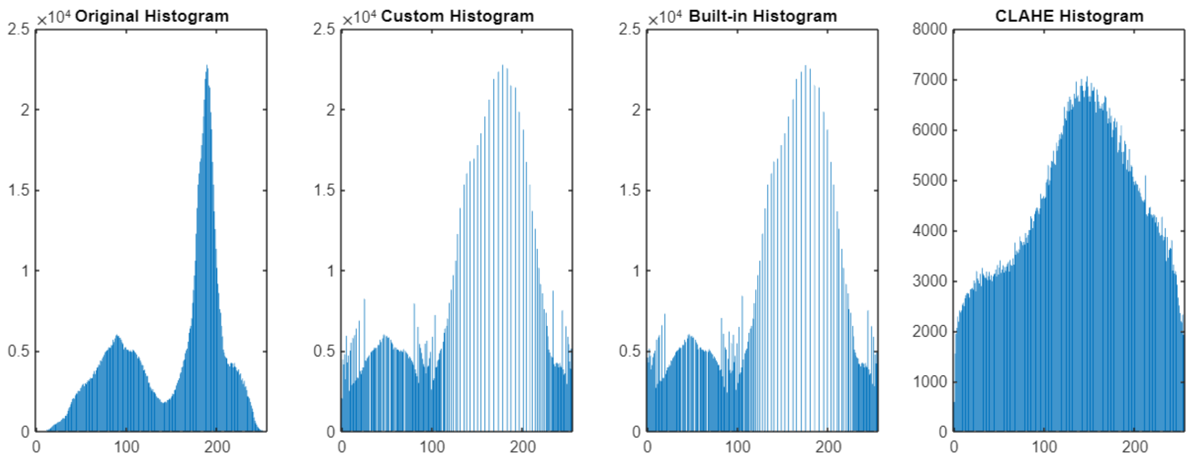
```
% Display original and equalized images side by side
subplot(2, 2, 1); imshow(original_image); title('Original Image');
```

```
subplot(2, 2, 2); imshow(equalized_image_custom); title('Custom Histogram
Equalization');
subplot(2, 2, 3); imshow(equalized_image_builtin); title('Built-in Histogram
Equalization');
subplot(2, 2, 4); imshow(clahe_image); title('CLAHE');
```




```
% Calculate Mean Squared Error (MSE) for CLAHE
mse_clahe_vs_builtin = immse(clahe_image, equalized_image_builtin);
```

```
% Set the width of the entire figure
figure_width = 1200;  % Adjust this value as needed
figure_height = 400;  % Adjust this value as needed
figure('Position', [100, 100, figure_width, figure_height]);

% Display histograms
subplot(1, 4, 1); bar(0:255, histogram_original); title('Original Histogram');
subplot(1, 4, 2); bar(0:255, imhist(equalized_image_custom)); title('Custom
Histogram');
subplot(1, 4, 3); bar(0:255, imhist(equalized_image_builtin)); title('Built-in
Histogram');
subplot(1, 4, 4); bar(0:255, imhist(clahe_image)); title('CLAHE Histogram');
```




```
disp(['MSE between Custom and Built-in Histogram Equalization: ',
num2str(mse_custom_vs_builtin)]);
```

```
MSE between Custom and Built-in Histogram Equalization: 1.9483
```

```
disp(['MSE between CLAHE and Built-in Histogram Equalization: ',
num2str(mse_clahe_vs_builtin)]);
```

```
MSE between CLAHE and Built-in Histogram Equalization: 3906.2518
```

```
function equalized_image = customHistogramEqualization(image)
    [h, w] = size(image);
    num_pixels = h * w;

    histogram = zeros(256, 1);
    for i = 1:h
        for j = 1:w
            pixel_value = image(i, j);
            histogram(pixel_value + 1) = histogram(pixel_value + 1) + 1;
        end
    end
    cumulative_histogram = cumsum(histogram) / num_pixels;
    equalized_image = uint8(255 * cumulative_histogram(double(image) + 1));
end
```

```
% Load a test image
image_path = "C:\Program Files\MATLAB\R2023a\toolbox\images\imdata\flamingos.jpg";
original_image = imread(image_path);
imshow(original_image)
```

```
original_image = rgb2gray(original_image); % Convert to grayscale if necessary
```

```
imshow(original_image)
```

```
% a. Calculate Histogram and Implement Histogram Equalization
% Function implementing histogram is written at the end of the file
equalized_image_custom = customHistogramEqualization(original_image);
```

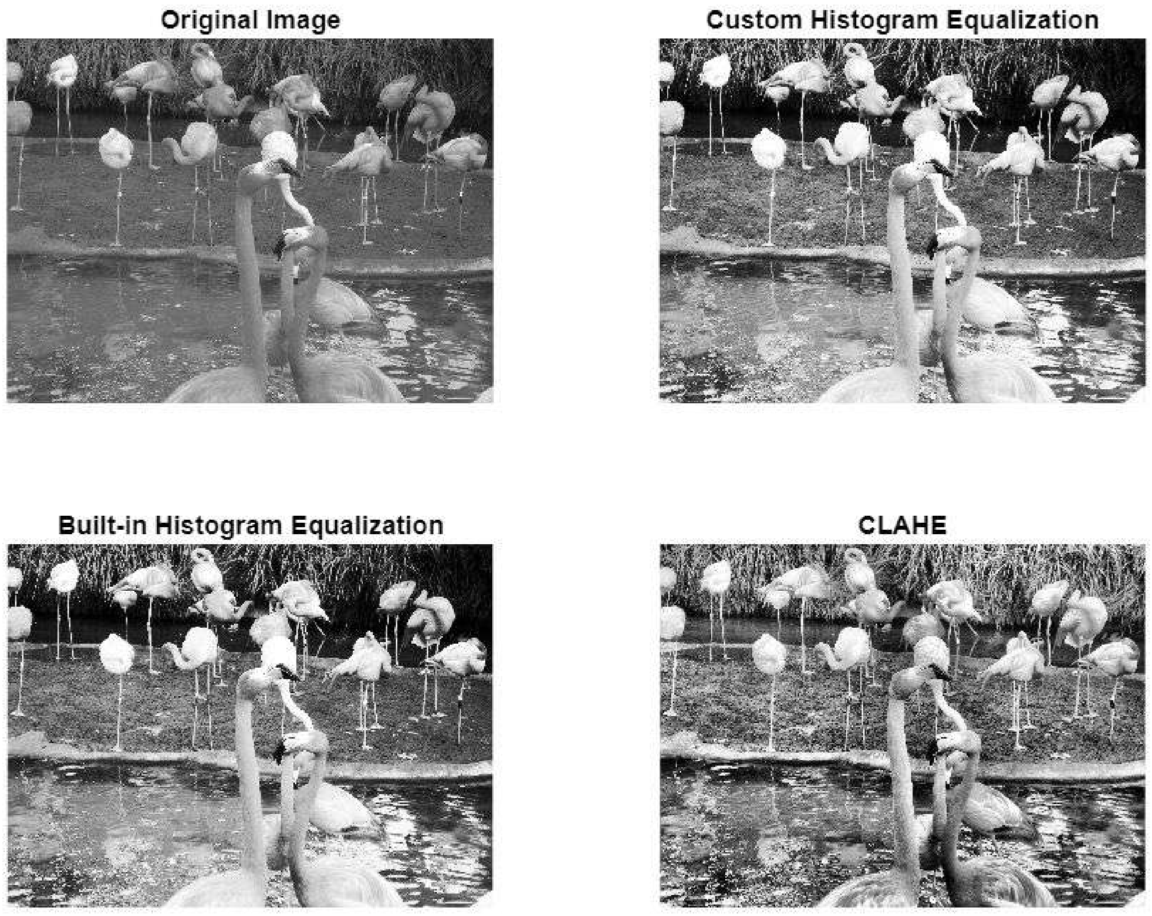
```
% Calculate histogram of the original image
histogram_original = imhist(original_image);
```

```
% b. Use Built-in Function and Compare Histograms
equalized_image_builtin = histeq(original_image, 256);
```

```
% Calculate Mean Squared Error (MSE)
mse_custom_vs_builtin = immse(equalized_image_custom, equalized_image_builtin);
```

```
% c. Apply Adaptive Histogram Equalization (CLAHE)
clahe_image = adapthisteq(original_image, 'ClipLimit', 0.02);
```

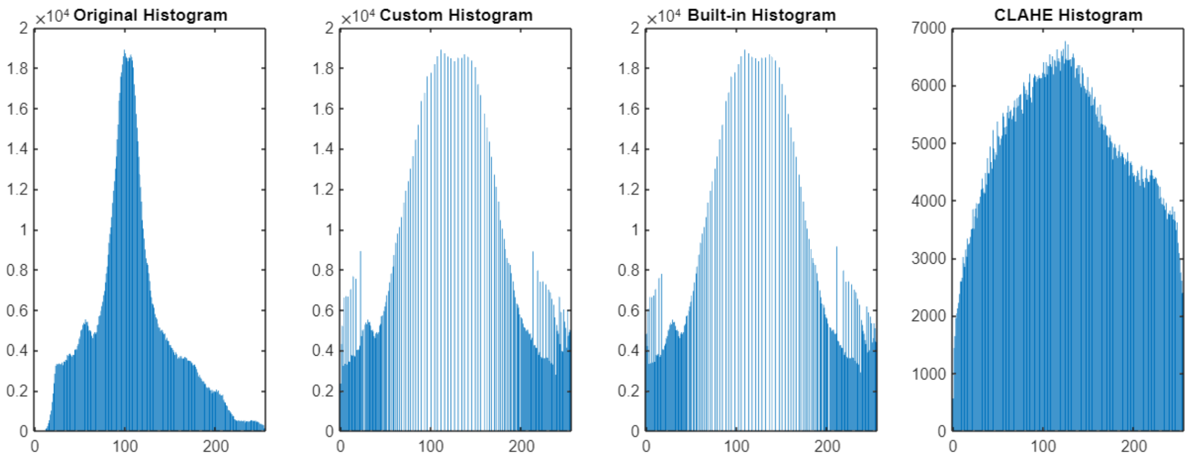
```
% Display original and equalized images side by side
subplot(2, 2, 1); imshow(original_image); title('Original Image');
subplot(2, 2, 2); imshow(equalized_image_custom); title('Custom Histogram
Equalization');
subplot(2, 2, 3); imshow(equalized_image_builtin); title('Built-in Histogram
Equalization');
subplot(2, 2, 4); imshow(clahe_image); title('CLAHE');
```




```
% Calculate Mean Squared Error (MSE) for CLAHE
mse_clahe_vs_builtin = immse(clahe_image, equalized_image_builtin);
```

```
% Set the width of the entire figure
figure_width = 1200;  % Adjust this value as needed
figure_height = 400;  % Adjust this value as needed
figure('Position', [100, 100, figure_width, figure_height]);

% Display histograms
subplot(1, 4, 1); bar(0:255, histogram_original); title('Original Histogram');
```

```
subplot(1, 4, 2); bar(0:255, imhist(equalized_image_custom)); title('Custom
Histogram');
subplot(1, 4, 3); bar(0:255, imhist(equalized_image_builtin)); title('Built-in
Histogram');
subplot(1, 4, 4); bar(0:255, imhist(clahe_image)); title('CLAHE Histogram');
```




```
disp(['MSE between Custom and Built-in Histogram Equalization: ',
num2str(mse_custom_vs_builtin)]);
```

```
MSE between Custom and Built-in Histogram Equalization: 1.6746
```

```
disp(['MSE between CLAHE and Built-in Histogram Equalization: ',
num2str(mse_clahe_vs_builtin)]);
```

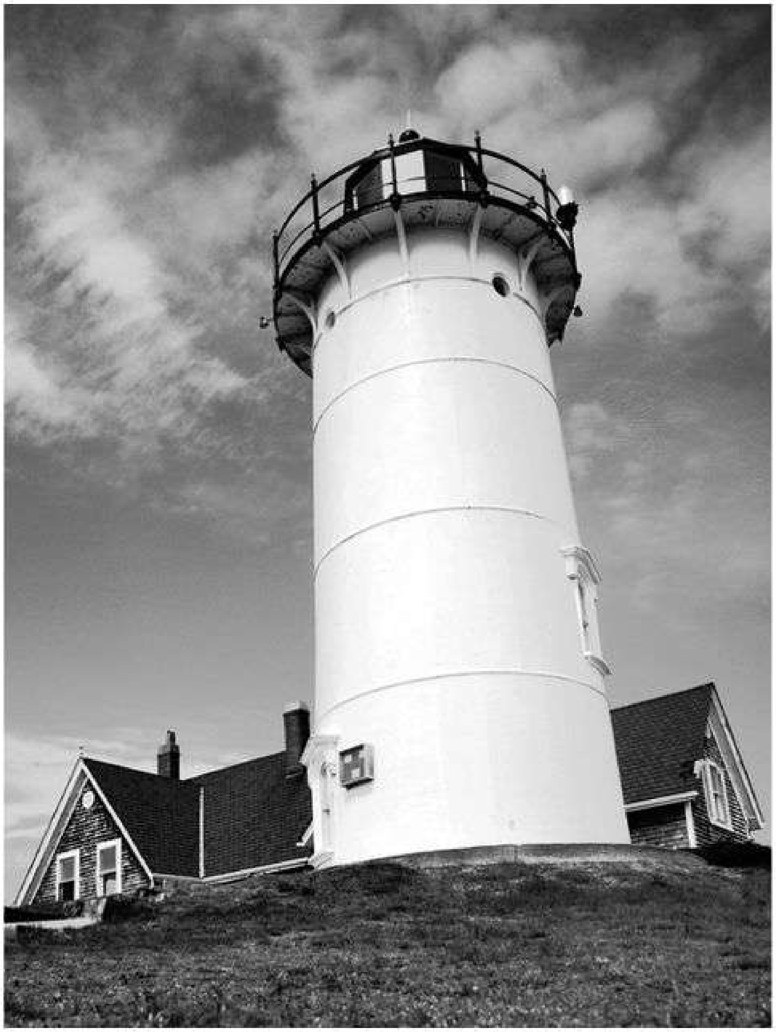
```
MSE between CLAHE and Built-in Histogram Equalization: 1921.6664
```

```
function equalized_image = customHistogramEqualization(image)
    [h, w] = size(image);
    num_pixels = h * w;

    histogram = zeros(256, 1);
    for i = 1:h
        for j = 1:w
            pixel_value = image(i, j);
            histogram(pixel_value + 1) = histogram(pixel_value + 1) + 1;
        end
    end
    cumulative_histogram = cumsum(histogram) / num_pixels;
    equalized_image = uint8(255 * cumulative_histogram(double(image) + 1));
end
```

```
% Load a test image
image_path = "C:\Program Files\MATLAB\R2023a\toolbox\images\imdata\lighthouse.png";
original_image = imread(image_path);
imshow(original_image)
```

```
original_image = rgb2gray(original_image); % Convert to grayscale if necessary
```

```
imshow(original_image)
```



```
% a. Calculate Histogram and Implement Histogram Equalization
% Function implementing histogram is written at the end of the file
equalized_image_custom = customHistogramEqualization(original_image);
```

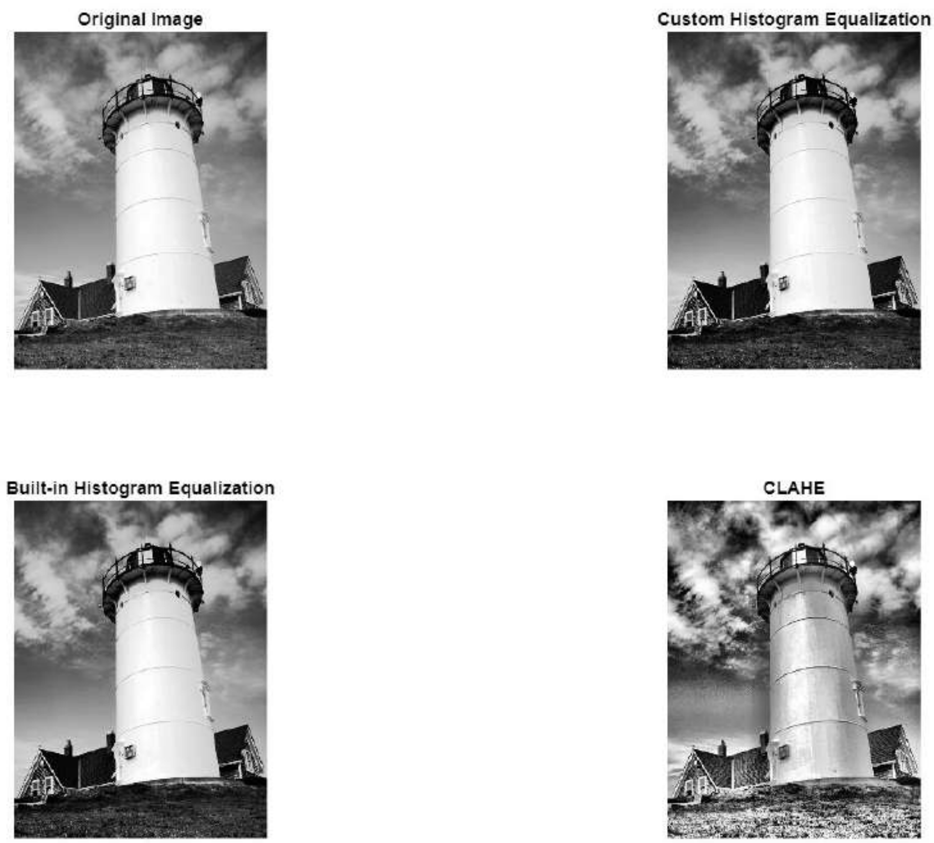
```
% Calculate histogram of the original image
histogram_original = imhist(original_image);
```

```
% b. Use Built-in Function and Compare Histograms
equalized_image_builtin = histeq(original_image, 256);
```

```
% Calculate Mean Squared Error (MSE)
mse_custom_vs_builtin = immse(equalized_image_custom, equalized_image_builtin);
```

```
% c. Apply Adaptive Histogram Equalization (CLAHE)
clahe_image = adapthisteq(original_image, 'ClipLimit', 0.02);
```

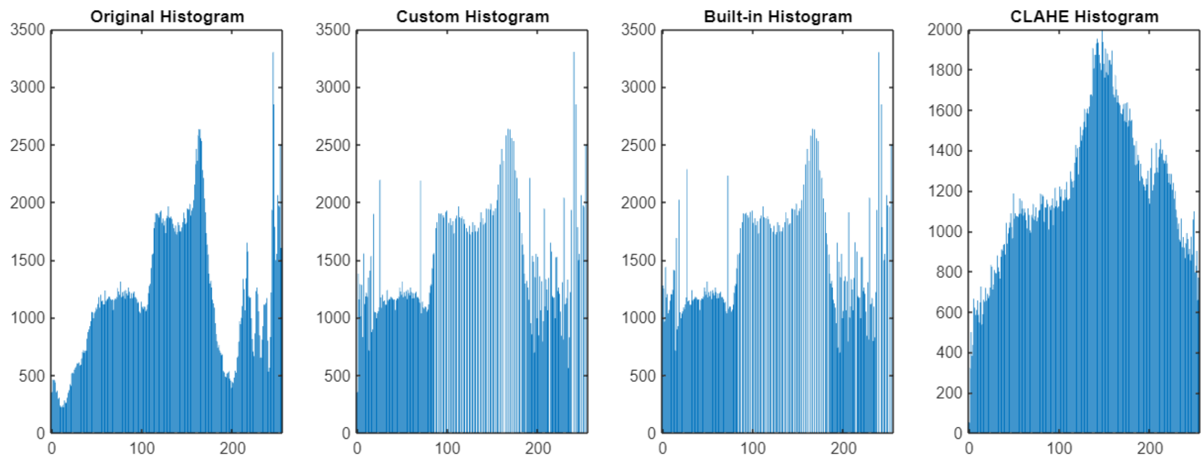
```
% Create a new figure with larger dimensions
figure('Position', [100, 100, 1200, 800]);

% Display original and equalized images side by side
subplot(2, 2, 1); imshow(original_image); title('Original Image');
subplot(2, 2, 2); imshow(equalized_image_custom); title('Custom Histogram
Equalization');
subplot(2, 2, 3); imshow(equalized_image_builtin); title('Built-in Histogram
Equalization');
subplot(2, 2, 4); imshow(clahe_image); title('CLAHE');
```




```
% Calculate Mean Squared Error (MSE) for CLAHE
mse_clahe_vs_builtin = immse(clahe_image, equalized_image_builtin);
```

```
% Set the width of the entire figure
figure_width = 1200;  % Adjust this value as needed
figure_height = 400;  % Adjust this value as needed
```

```
figure('Position', [100, 100, figure_width, figure_height]);

% Display histograms
subplot(1, 4, 1); bar(0:255, histogram_original); title('Original Histogram');
subplot(1, 4, 2); bar(0:255, imhist(equalized_image_custom)); title('Custom
Histogram');
subplot(1, 4, 3); bar(0:255, imhist(equalized_image_builtin)); title('Built-in
Histogram');
subplot(1, 4, 4); bar(0:255, imhist(clahe_image)); title('CLAHE Histogram');
```




```
disp(['MSE between Custom and Built-in Histogram Equalization: ',
num2str(mse_custom_vs_builtin)]);
```

```
MSE between Custom and Built-in Histogram Equalization: 0.62138
```

```
disp(['MSE between CLAHE and Built-in Histogram Equalization: ',
num2str(mse_clahe_vs_builtin)]);
```

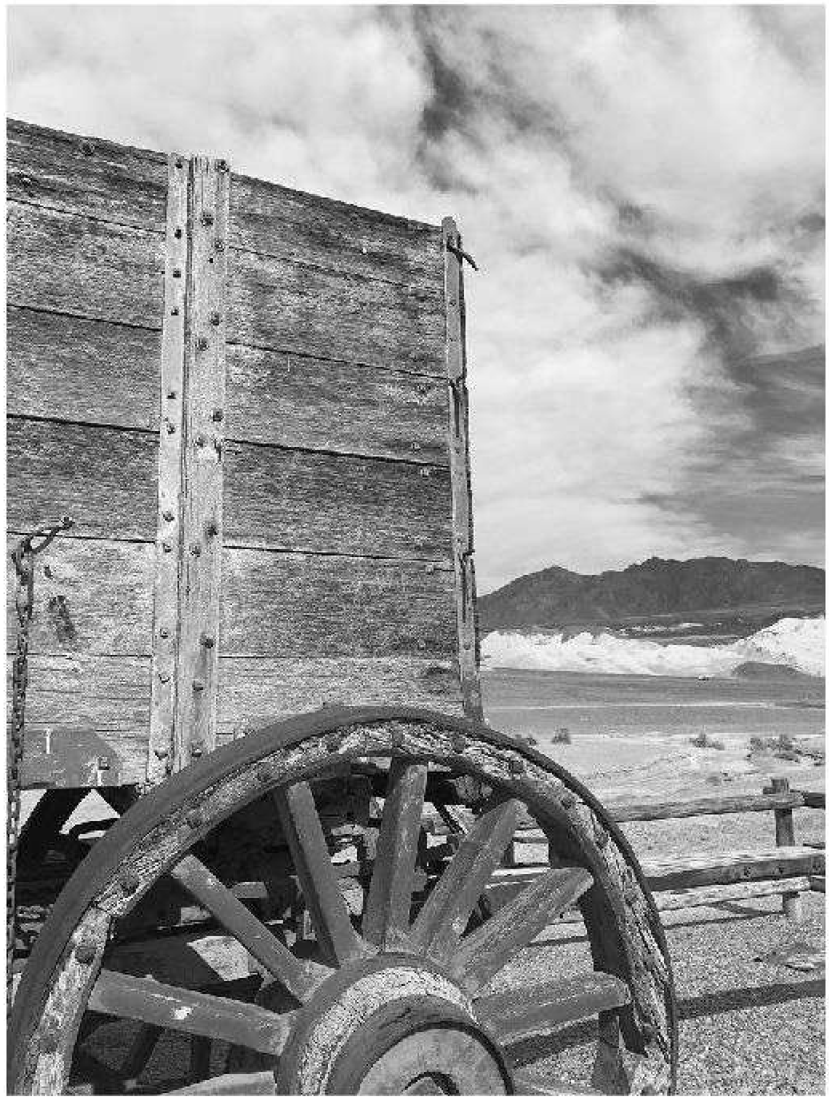
```
MSE between CLAHE and Built-in Histogram Equalization: 2452.5428
```

```
function equalized_image = customHistogramEqualization(image)
    [h, w] = size(image);
    num_pixels = h * w;

    histogram = zeros(256, 1);
    for i = 1:h
        for j = 1:w
            pixel_value = image(i, j);
            histogram(pixel_value + 1) = histogram(pixel_value + 1) + 1;
        end
    end
    cumulative_histogram = cumsum(histogram) / num_pixels;
    equalized_image = uint8(255 * cumulative_histogram(double(image) + 1));
end
```

```
% Load a test image
image_path = "C:\Program Files\MATLAB\R2023a\toolbox\images\imdata\wagon.jpg";
original_image = imread(image_path);
imshow(original_image)
```

```
original_image = rgb2gray(original_image); % Convert to grayscale if necessary
```

```
imshow(original_image)
```



```
% a. Calculate Histogram and Implement Histogram Equalization
% Function implementing histogram is written at the end of the file
equalized_image_custom = customHistogramEqualization(original_image);
```

```
% Calculate histogram of the original image
histogram_original = imhist(original_image);
```

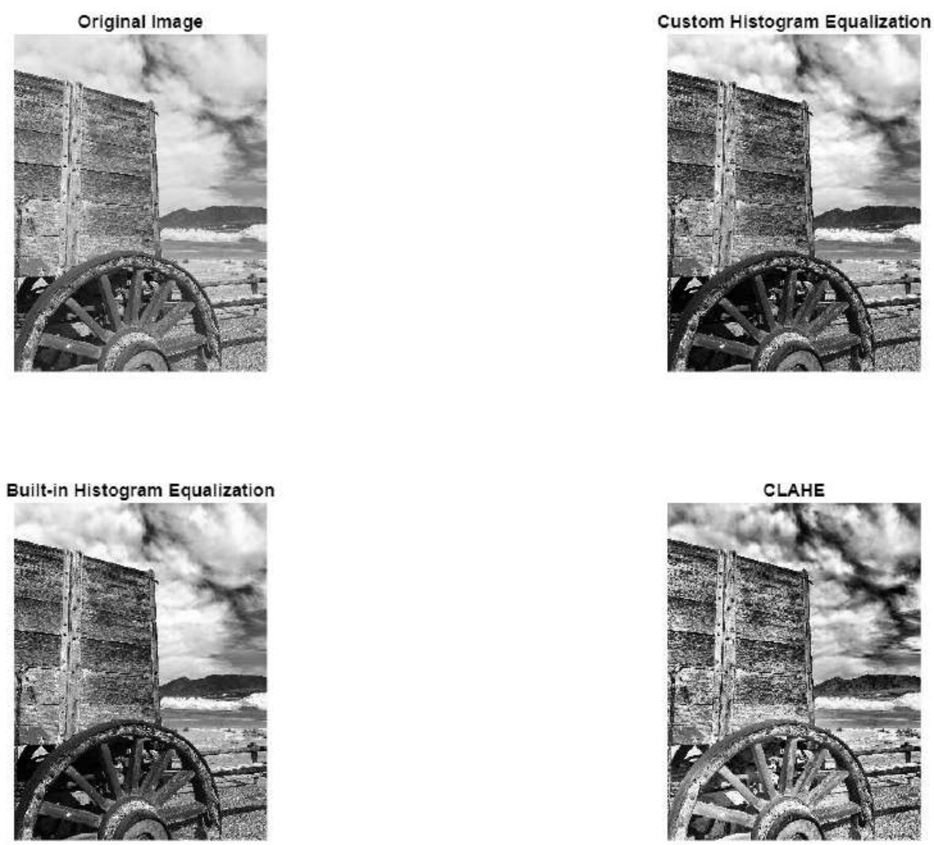
```
% b. Use Built-in Function and Compare Histograms
equalized_image_builtin = histeq(original_image, 256);
```

```
% Calculate Mean Squared Error (MSE)
```

```
mse_custom_vs_builtin = immse(equalized_image_custom, equalized_image_builtin);
```

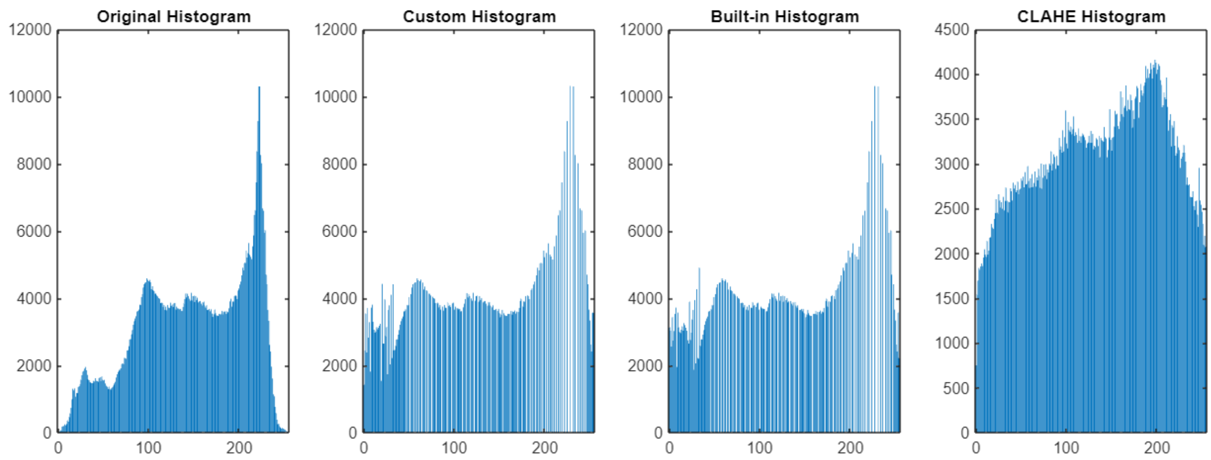
```
% c. Apply Adaptive Histogram Equalization (CLAHE)
clahe_image = adapthisteq(original_image, 'ClipLimit', 0.02);
```

```
% Create a new figure with larger dimensions
figure('Position', [100, 100, 1200, 800]);

% Display original and equalized images side by side
subplot(2, 2, 1); imshow(original_image); title('Original Image');
subplot(2, 2, 2); imshow(equalized_image_custom); title('Custom Histogram Equalization');
subplot(2, 2, 3); imshow(equalized_image_builtin); title('Built-in Histogram Equalization');
subplot(2, 2, 4); imshow(clahe_image); title('CLAHE');
```




```
% Calculate Mean Squared Error (MSE) for CLAHE
mse_clahe_vs_builtin = immse(clahe_image, equalized_image_builtin);
```

```
% Set the width of the entire figure
figure_width = 1200;  % Adjust this value as needed
figure_height = 400;  % Adjust this value as needed
figure('Position', [100, 100, figure_width, figure_height]);

% Display histograms
subplot(1, 4, 1); bar(0:255, histogram_original); title('Original Histogram');
subplot(1, 4, 2); bar(0:255, imhist(equalized_image_custom)); title('Custom
Histogram');
subplot(1, 4, 3); bar(0:255, imhist(equalized_image_builtin)); title('Built-in
Histogram');
subplot(1, 4, 4); bar(0:255, imhist(clahe_image)); title('CLAHE Histogram');
```




```
disp(['MSE between Custom and Built-in Histogram Equalization: ',
num2str(mse_custom_vs_builtin)]);
```

```
MSE between Custom and Built-in Histogram Equalization: 0.63108
```

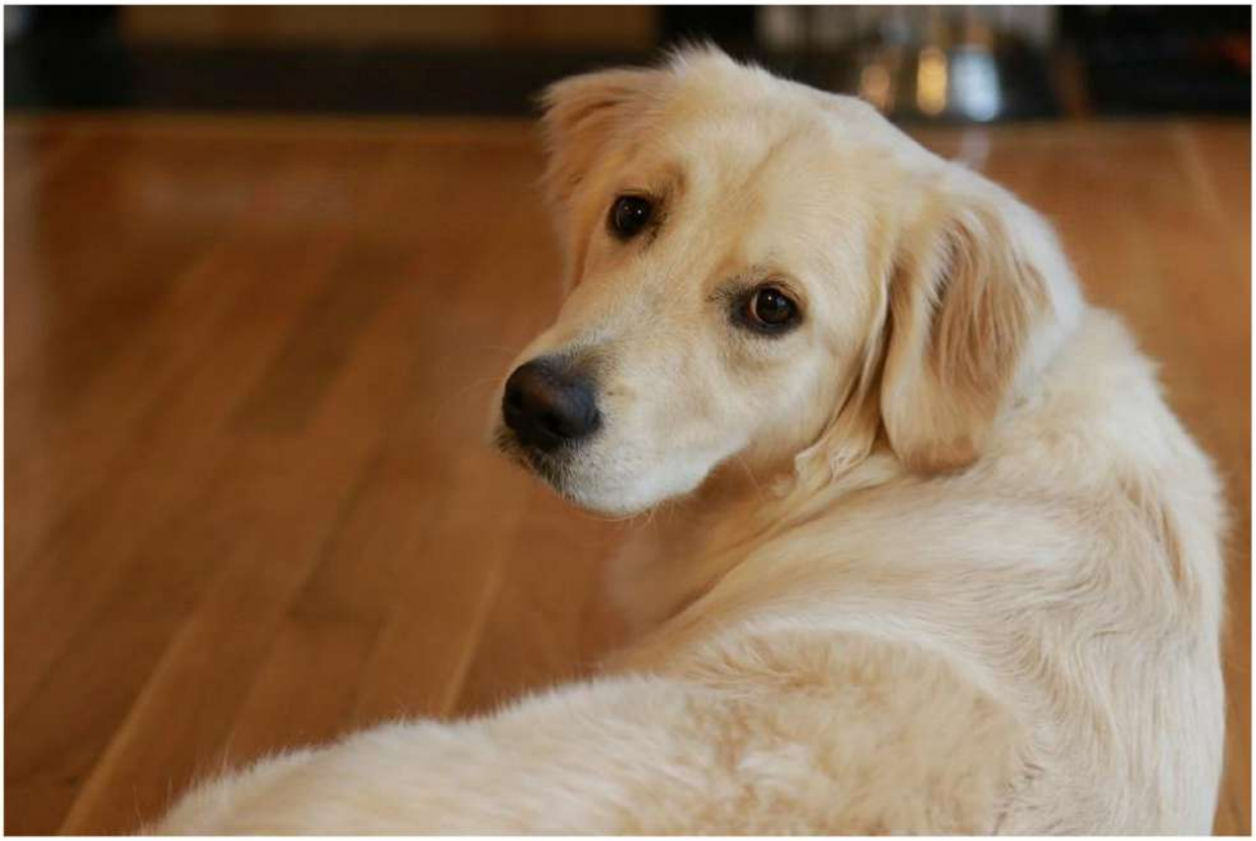
```
disp(['MSE between CLAHE and Built-in Histogram Equalization: ',
num2str(mse_clahe_vs_builtin)]);
```

```
MSE between CLAHE and Built-in Histogram Equalization: 2007.8872
```

```
function equalized_image = customHistogramEqualization(image)
    [h, w] = size(image);
    num_pixels = h * w;

    histogram = zeros(256, 1);
    for i = 1:h
        for j = 1:w
            pixel_value = image(i, j);
            histogram(pixel_value + 1) = histogram(pixel_value + 1) + 1;
        end
    end
    cumulative_histogram = cumsum(histogram) / num_pixels;
```

```
    equalized_image = uint8(255 * cumulative_histogram(double(image) + 1));
end
```

```
% Load a test image
image_path = "C:\Program Files\MATLAB\R2023a\toolbox\images\imdata\sherlock.jpg";
original_image = imread(image_path);
imshow(original_image)
```



```
original_image = rgb2gray(original_image); % Convert to grayscale if necessary
```

```
imshow(original_image)
```

```
% a. Calculate Histogram and Implement Histogram Equalization
% Function implementing histogram is written at the end of the file
equalized_image_custom = customHistogramEqualization(original_image);
```

```
% Calculate histogram of the original image
histogram_original = imhist(original_image);
```

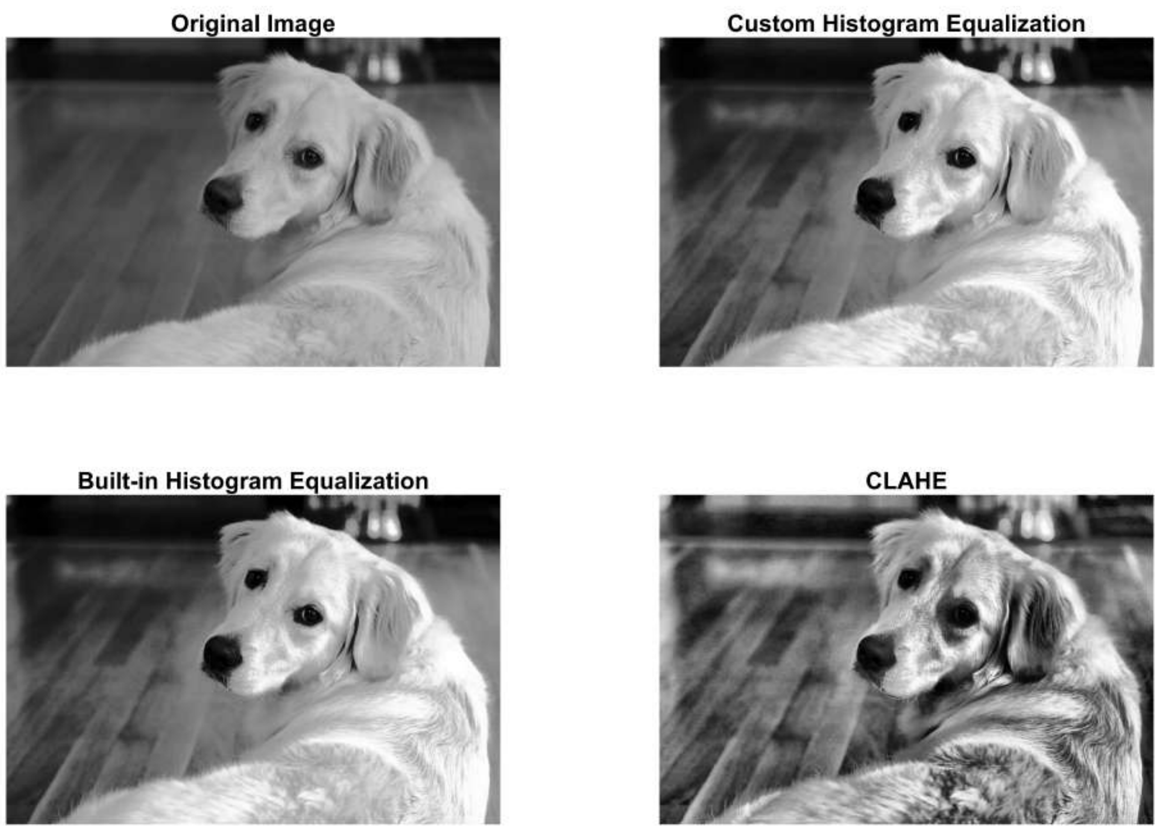
```
% b. Use Built-in Function and Compare Histograms
equalized_image_builtin = histeq(original_image, 256);
```

```
% Calculate Mean Squared Error (MSE)
mse_custom_vs_builtin = immse(equalized_image_custom, equalized_image_builtin);
```

```
% c. Apply Adaptive Histogram Equalization (CLAHE)
clahe_image = adapthisteq(original_image, 'ClipLimit', 0.02);
```

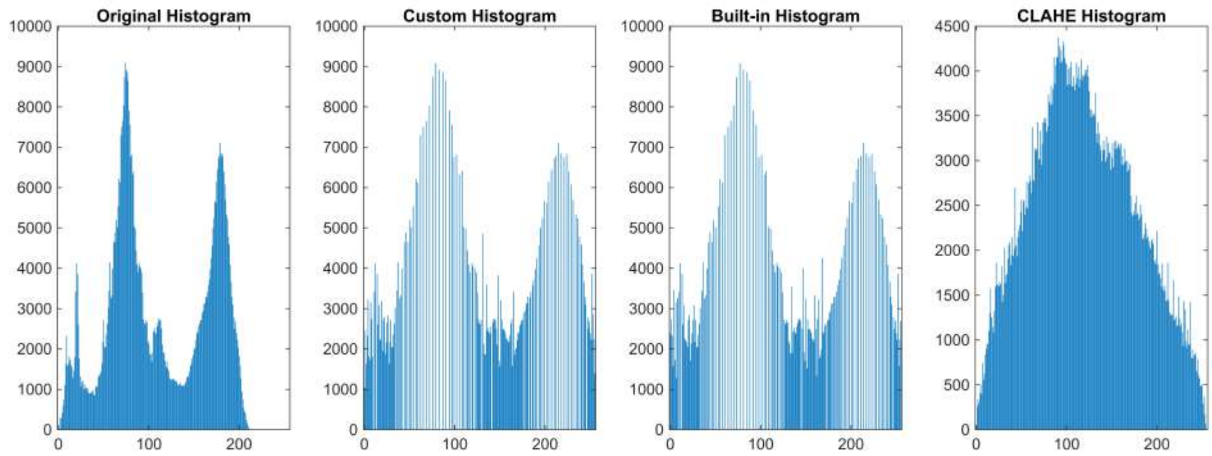
```
% Display original and equalized images side by side
subplot(2, 2, 1); imshow(original_image); title('Original Image');
```

```
subplot(2, 2, 2); imshow(equalized_image_custom); title('Custom Histogram
Equalization');
subplot(2, 2, 3); imshow(equalized_image_builtin); title('Built-in Histogram
Equalization');
subplot(2, 2, 4); imshow(clahe_image); title('CLAHE');
```




```
% Calculate Mean Squared Error (MSE) for CLAHE
mse_clahe_vs_builtin = immse(clahe_image, equalized_image_builtin);
```

```
% Set the width of the entire figure
figure_width = 1200;  % Adjust this value as needed
figure_height = 400;  % Adjust this value as needed
figure('Position', [100, 100, figure_width, figure_height]);

% Display histograms
subplot(1, 4, 1); bar(0:255, histogram_original); title('Original Histogram');
subplot(1, 4, 2); bar(0:255, imhist(equalized_image_custom)); title('Custom
Histogram');
subplot(1, 4, 3); bar(0:255, imhist(equalized_image_builtin)); title('Built-in
Histogram');
subplot(1, 4, 4); bar(0:255, imhist(clahe_image)); title('CLAHE Histogram');
```




```
disp(['MSE between Custom and Built-in Histogram Equalization: ',
num2str(mse_custom_vs_builtin)]);
```

```
MSE between Custom and Built-in Histogram Equalization: 1.4404
```

```
disp(['MSE between CLAHE and Built-in Histogram Equalization: ',
num2str(mse_clahe_vs_builtin)]);
```

```
MSE between CLAHE and Built-in Histogram Equalization: 2572.6691
```

```
function equalized_image = customHistogramEqualization(image)
    [h, w] = size(image);
    num_pixels = h * w;

    histogram = zeros(256, 1);
    for i = 1:h
        for j = 1:w
            pixel_value = image(i, j);
            histogram(pixel_value + 1) = histogram(pixel_value + 1) + 1;
        end
    end
    cumulative_histogram = cumsum(histogram) / num_pixels;
    equalized_image = uint8(255 * cumulative_histogram(double(image) + 1));
end
```

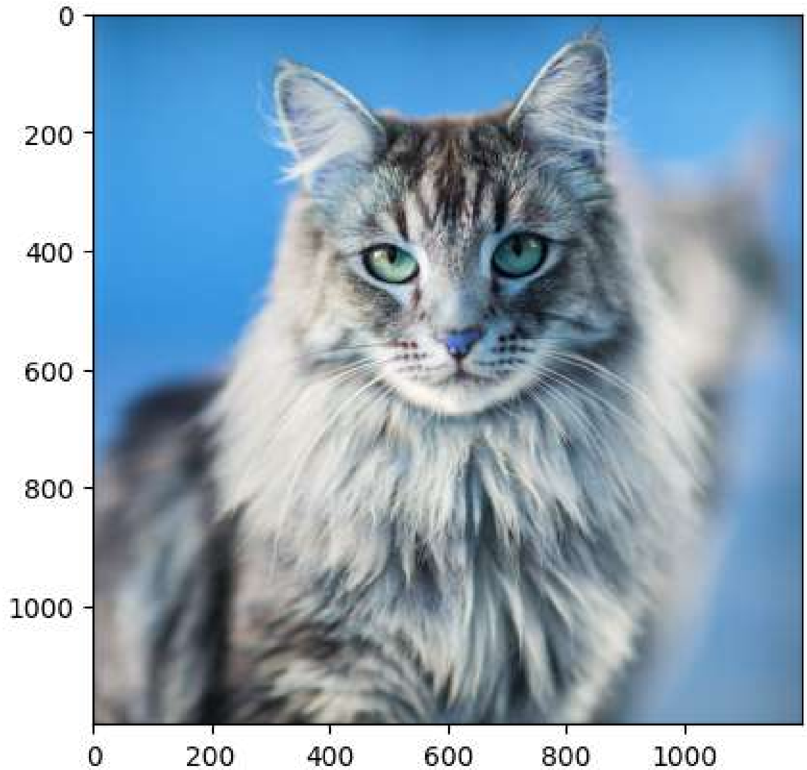
q4

August 13, 2023

[4]:
```
import cv2
import matplotlib.pyplot as plt
import matplotlib.image as img
import numpy as np
```

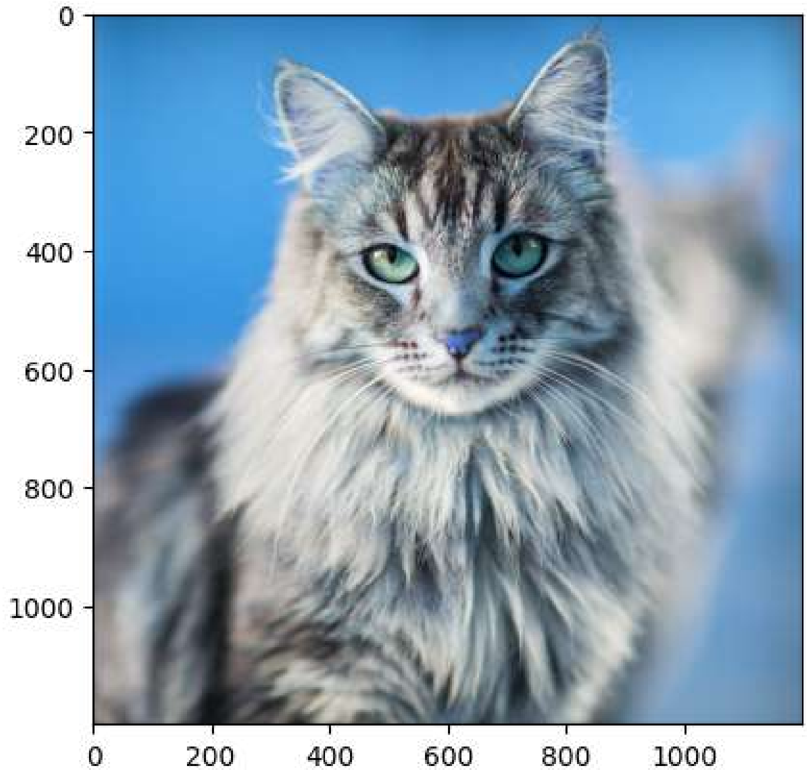
[2]:
```
path = r'./cat_image.jpg' # Path to Image


image = cv2.imread(path) # Reading the image
plt.imshow(image) # Displaying the Image

print("Resolution of image is" , image.shape)

# cv2.imshow(window_name, image)
```

[2]: (1200, 1200, 3)



[5]:
```
identity_kernel = np.array([[0, 0, 0],
                            [0, 1, 0],
                            [0, 0, 0]])  # Creating Identity Matrix

identity_filtered_image = cv2.filter2D(image, ddepth=-1,␣
 ↪kernel=identity_kernel) # Passing image through identity filter
plt.imshow(identity_filtered_image) # Displaying the Image
```

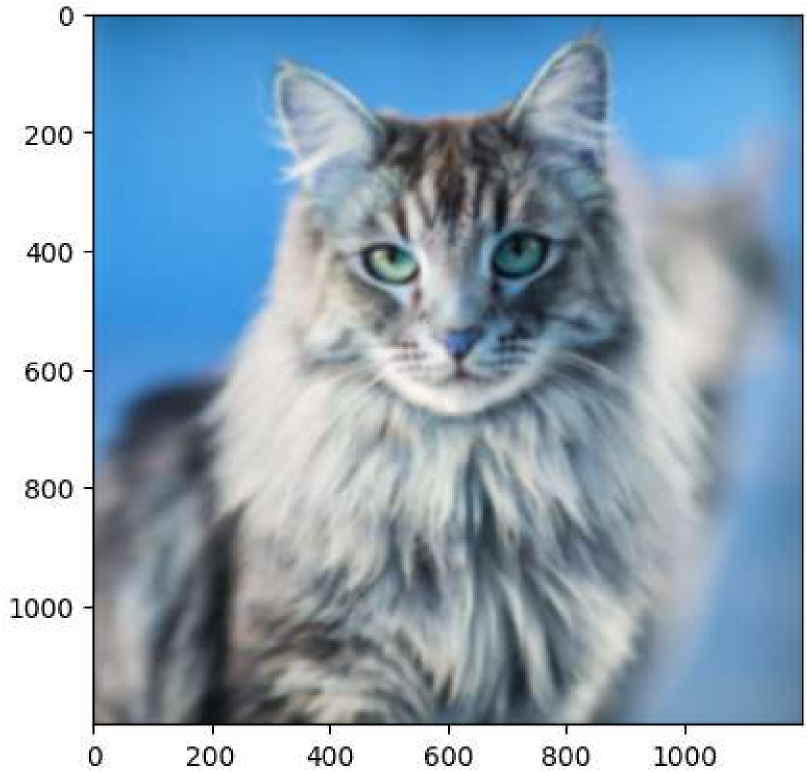
[5]: <matplotlib.image.AxesImage at 0x7f8248d2c160>



[27]:
```
# Defining Kernel size for blurring
ksize = (21, 21)

# Using cv2.blur() method
blur_image = cv2.GaussianBlur(image, ksize, 0)
print(blur_image.shape)
plt.imshow(blur_image)
# Displaying the image
```

```
(1200, 1200, 3)
```

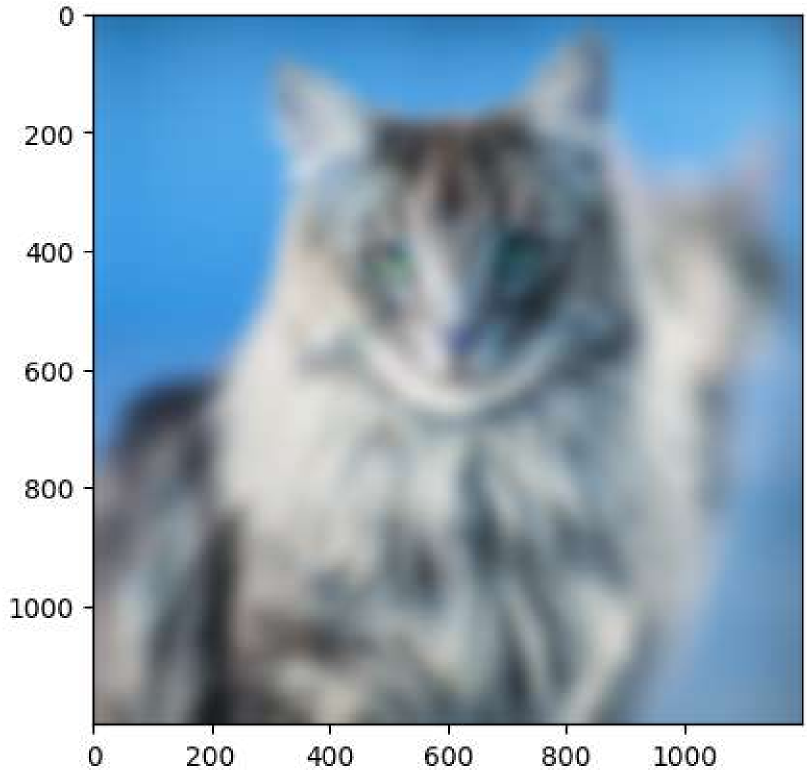
[27]: <matplotlib.image.AxesImage at 0x7f8245cc7730>



[21]:
```
# Kernel size for large_blur would be higher so that image would be more smooth
ksize1 = (101, 101)

# Using cv2.blur() method
blur_image1 = cv2.GaussianBlur(image, ksize1,0)
print(blur_image1.shape)
plt.imshow(blur_image1)
# Displaying the image
```

```
(1200, 1200, 3)
```

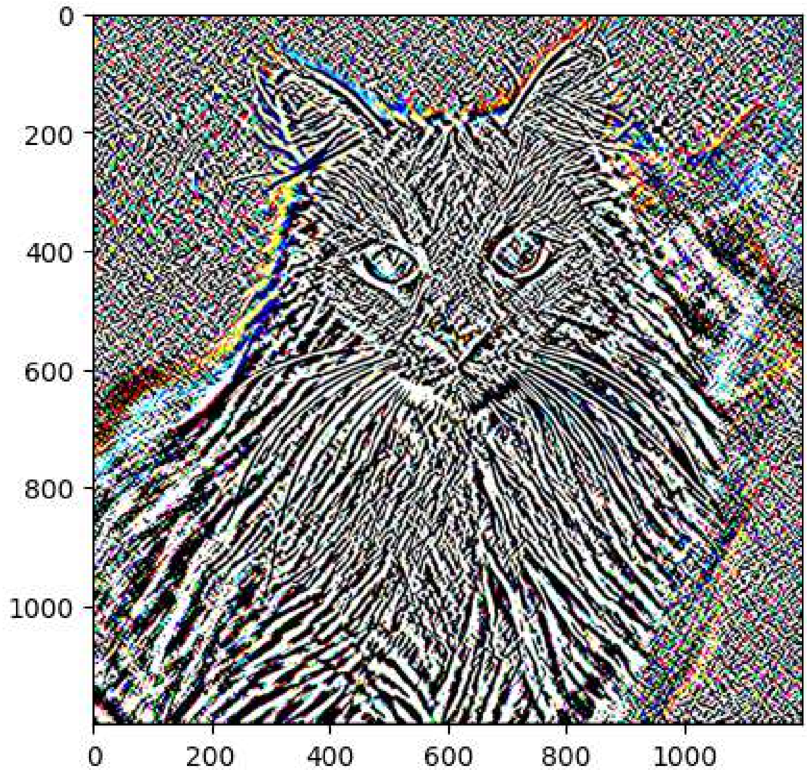
[21]:
```
<matplotlib.image.AxesImage at 0x7f8245dde830>
```



[10]:
```
# Sobel filter for edge detection
sobel_image = cv2.Sobel(image,cv2.CV_64F,1,1,ksize=21)
# Calculated gradient direction at 45* angle as it gave the most acccurate␣
↪results as
# compared to gradient purely in x and y direction

plt.imshow(sobel_image)
```

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

[10]: <matplotlib.image.AxesImage at 0x7f8246950280>

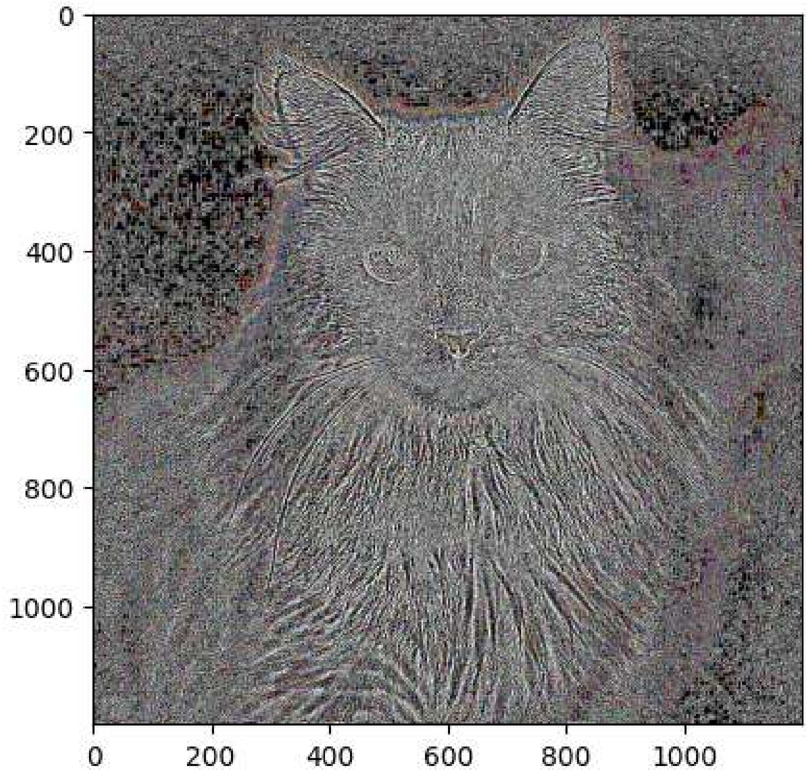


[13]:
```
laplacian_image = cv2.Laplacian(image,cv2.CV_64F,5) # Applying Laplacian filter
    ↪with kernel size = 5
print(laplacian_image.shape)
plt.imshow(laplacian_image)
```

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

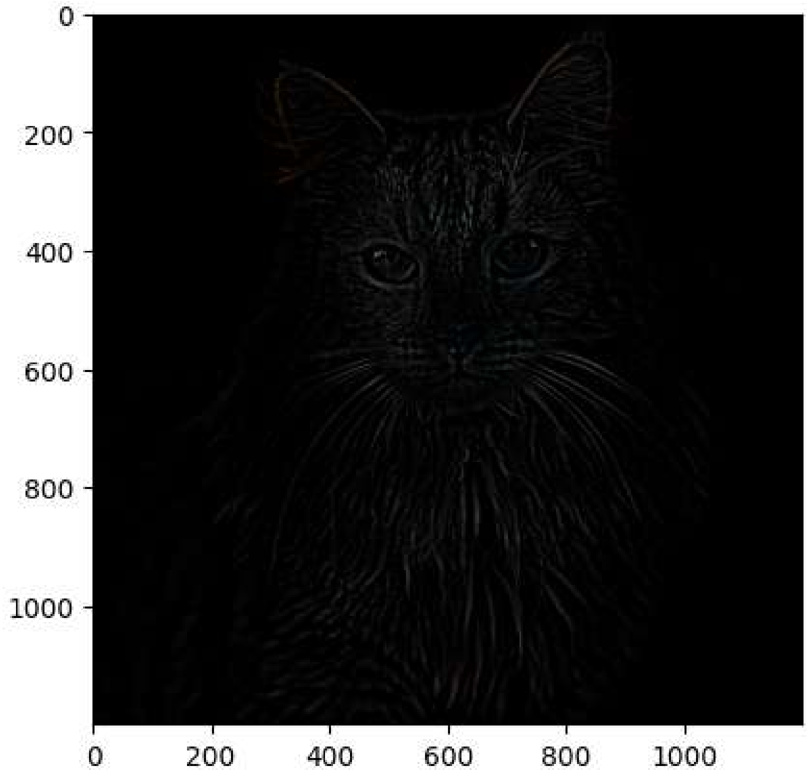
```
(1200, 1200, 3)
```

[13]: <matplotlib.image.AxesImage at 0x7f8246a5f580>



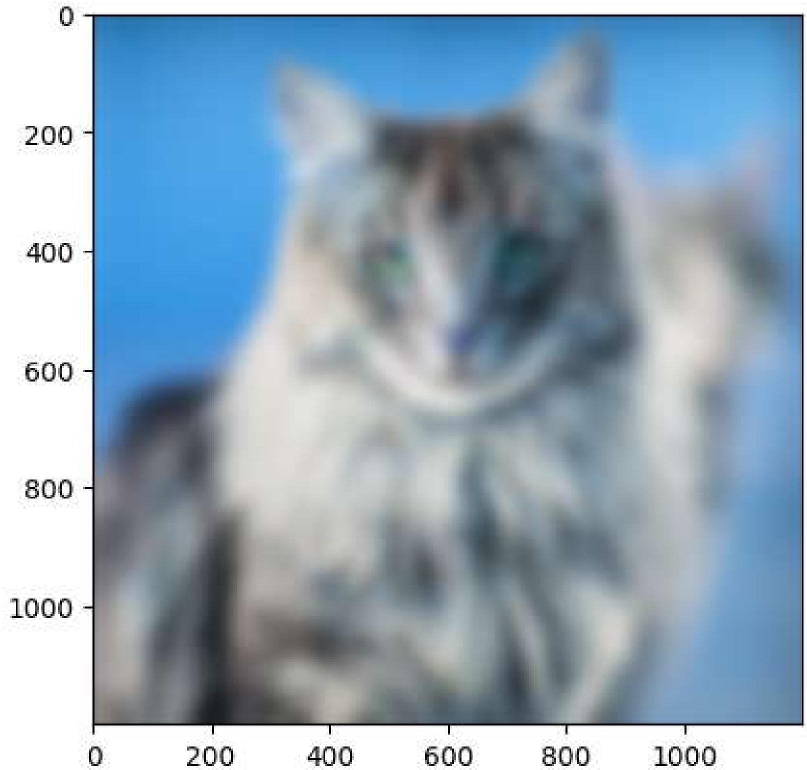

```
[17]: hpf = cv2.subtract(image,cv2.GaussianBlur(image, (21, 21),7)) # The high pass␣
      ↪filter is created by subtracting the Gaussian Blurred image from original␣
      ↪image
      plt.imshow(hpf)
```

```
[17]: <matplotlib.image.AxesImage at 0x7f8245f5be20>
```



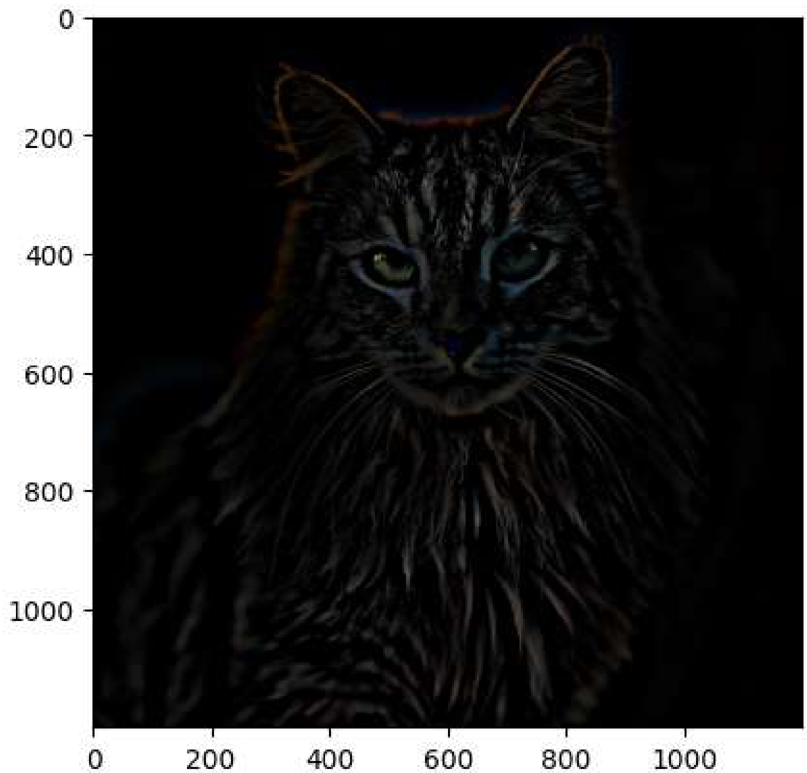
[37]:
```
# Low frequency image can be obtained by applying the Gaussian Blur on the␣
↪original filter



# Using cv2.blur() method
low_frequency_image = cv2.GaussianBlur(image, ksize1, 0)
plt.imshow(low_frequency_image)
# Displaying the image
```

[37]: <matplotlib.image.AxesImage at 0x7f824594ca90>



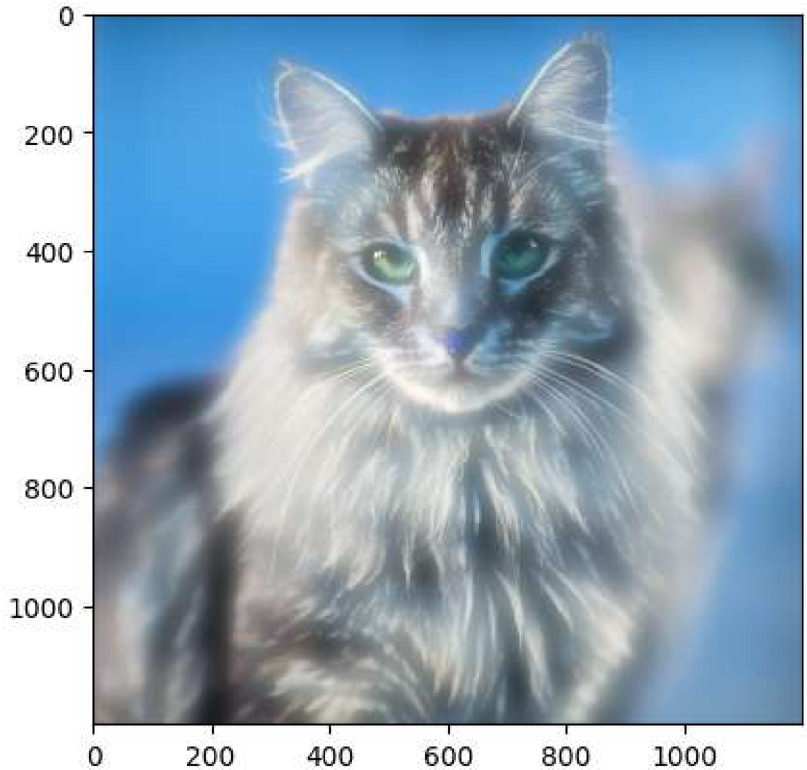
[32]:
```
# High frequency image can be  created in many ways, here I am subtracting the␣
↪lareg_blur image from original image

high_frequency_image = cv2.subtract(image , cv2.GaussianBlur(image, ksize1, 0))
plt.imshow(high_frequency_image)
```

[32]: <matplotlib.image.AxesImage at 0x7f8245af47f0>



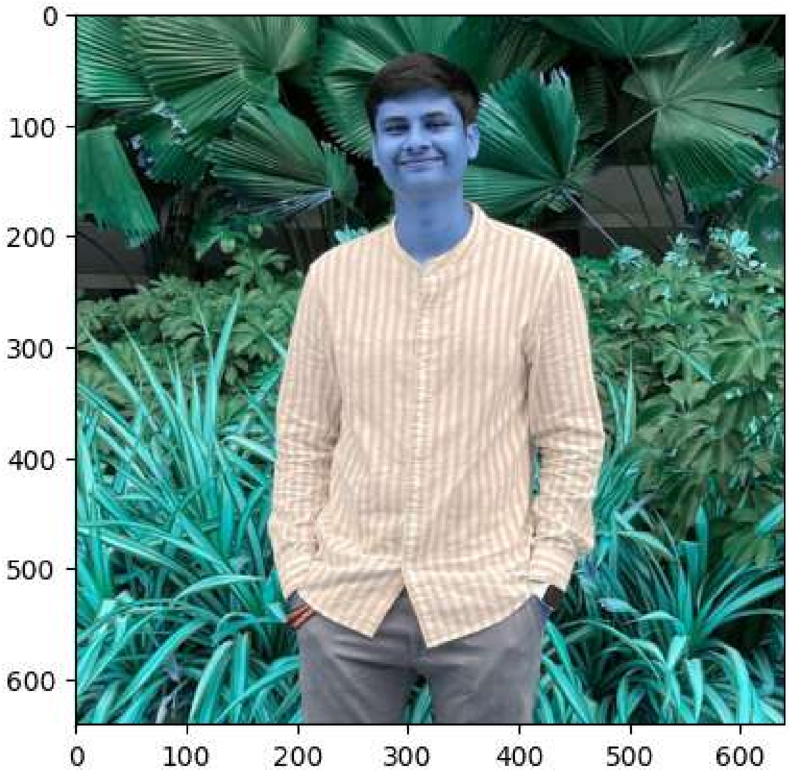
[38]:
```
# Creating a hybrid image by adding the low frequency and high frequency images
plt.imshow(cv2.add(low_frequency_image,high_frequency_image))
```

[38]: <matplotlib.image.AxesImage at 0x7f82457bd150>



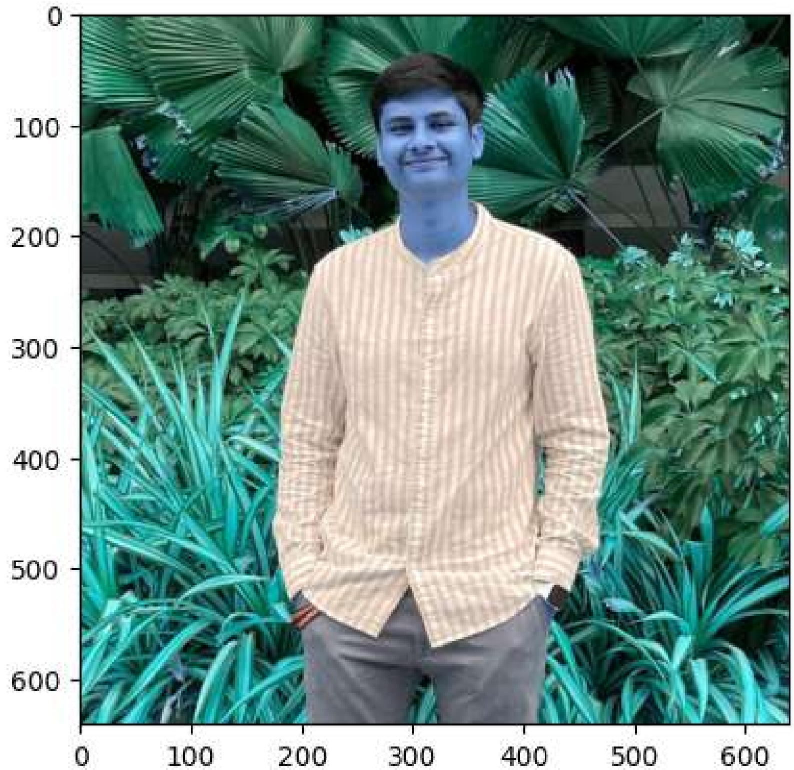
[40]:
```
path = r'./Human_image.enc' # Path to Image


image = cv2.imread(path) # Reading the image
plt.imshow(image) # Displaying the Image

print("Resolution of image is" , image.shape)
```

```
Resolution of image is (641, 640, 3)
```




```
[41]: identity_kernel = np.array([[0, 0, 0],
                                  [0, 1, 0],
                                  [0, 0, 0]])  # Creating Identity Matrix

      identity_filtered_image = cv2.filter2D(image, ddepth=-1,␣
       ↪kernel=identity_kernel) # Passing image through identity filter
      plt.imshow(identity_filtered_image) # Displaying the Image
```

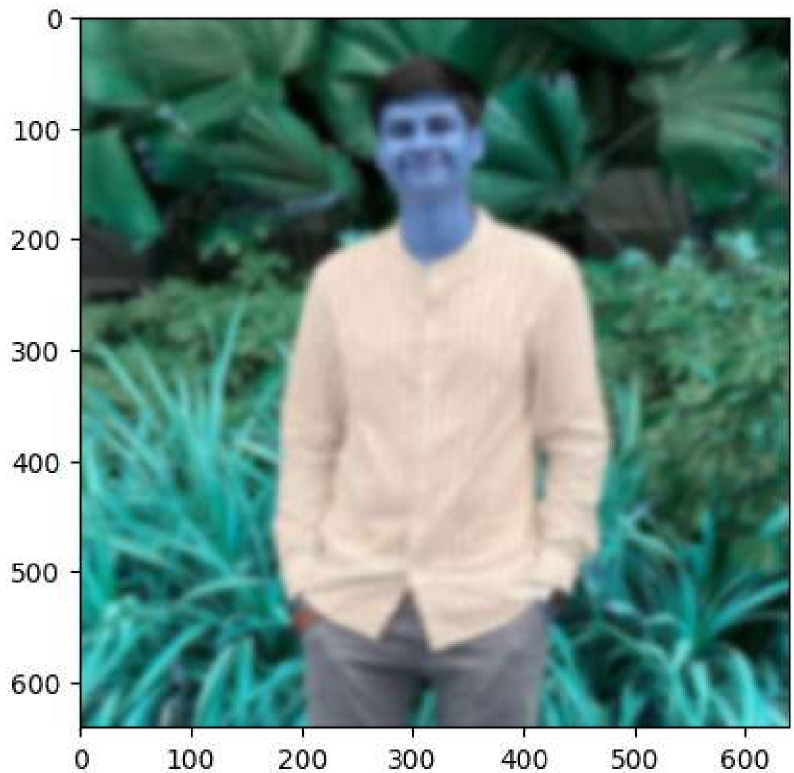
```
[41]: <matplotlib.image.AxesImage at 0x7f824b95df90>
```



```
[42]: # Defining Kernel size for blurring
      ksize = (21, 21)

      # Using cv2.blur() method
      blur_image = cv2.GaussianBlur(image, ksize, 0)
      print(blur_image.shape)
      plt.imshow(blur_image)
      # Displaying the image
```

```
(641, 640, 3)
```

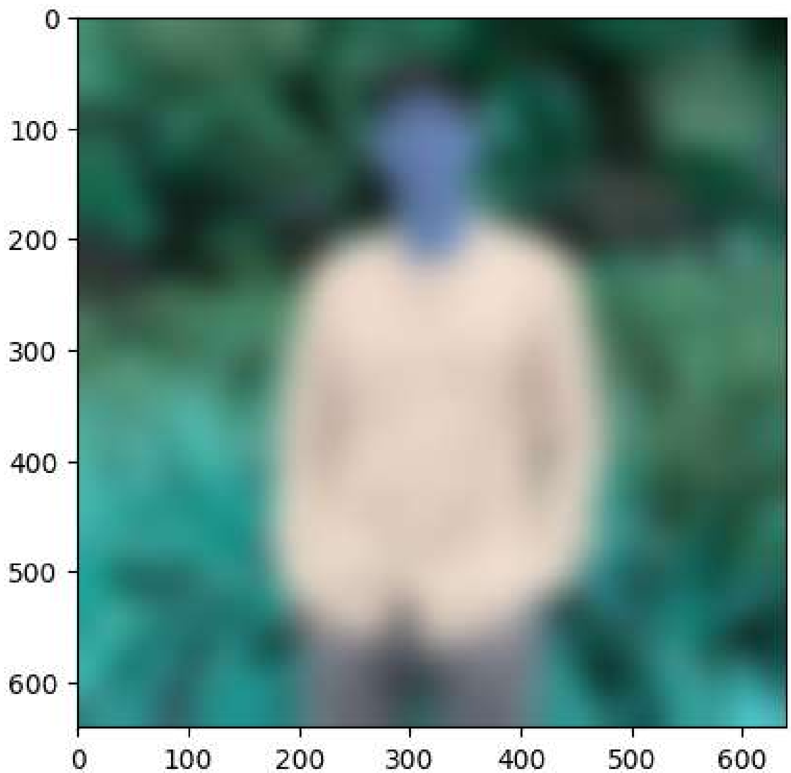
```
[42]: <matplotlib.image.AxesImage at 0x7f824b7cebc0>
```



```
[44]: # Kernel size for large_blur would be higher so that image would be more smooth
ksize1 = (101, 101)

# Using cv2.blur() method
blur_image1 = cv2.GaussianBlur(image, ksize1,0)
print(blur_image1.shape)
plt.imshow(blur_image1)
# Displaying the image
```

```
(641, 640, 3)
```

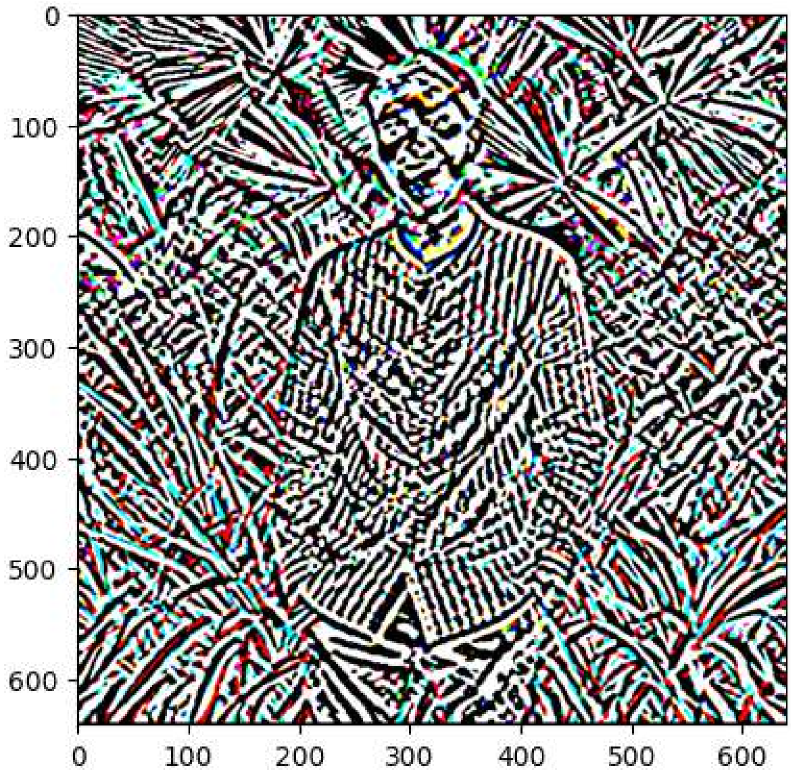
```
[44]: <matplotlib.image.AxesImage at 0x7f824b6ca3e0>
```



[45]:
```
# Sobel filter for edge detection
sobel_image = cv2.Sobel(image,cv2.CV_64F,1,1,ksize=21)
# Calculated gradient direction at 45* angle as it gave the most acccurate
 ↪results as
# compared to gradient purely in x and y direction

plt.imshow(sobel_image)
```

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

[45]: `<matplotlib.image.AxesImage at 0x7f824b745e10>`

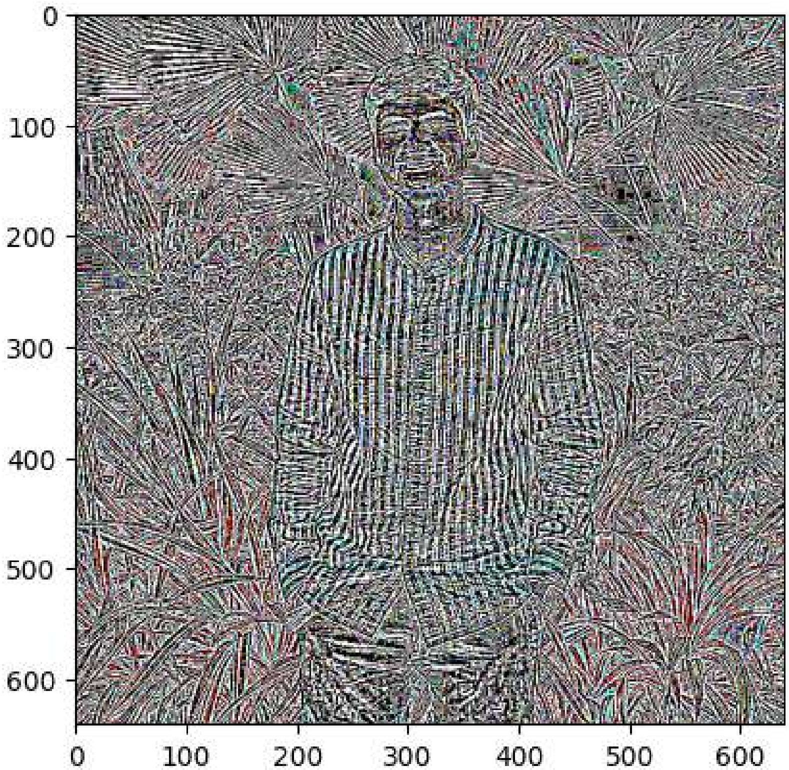


[46]:
```
laplacian_image = cv2.Laplacian(image,cv2.CV_64F,5) # Applying Laplacian filter␣
↪with kernel size = 5
print(laplacian_image.shape)
plt.imshow(laplacian_image)
```

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

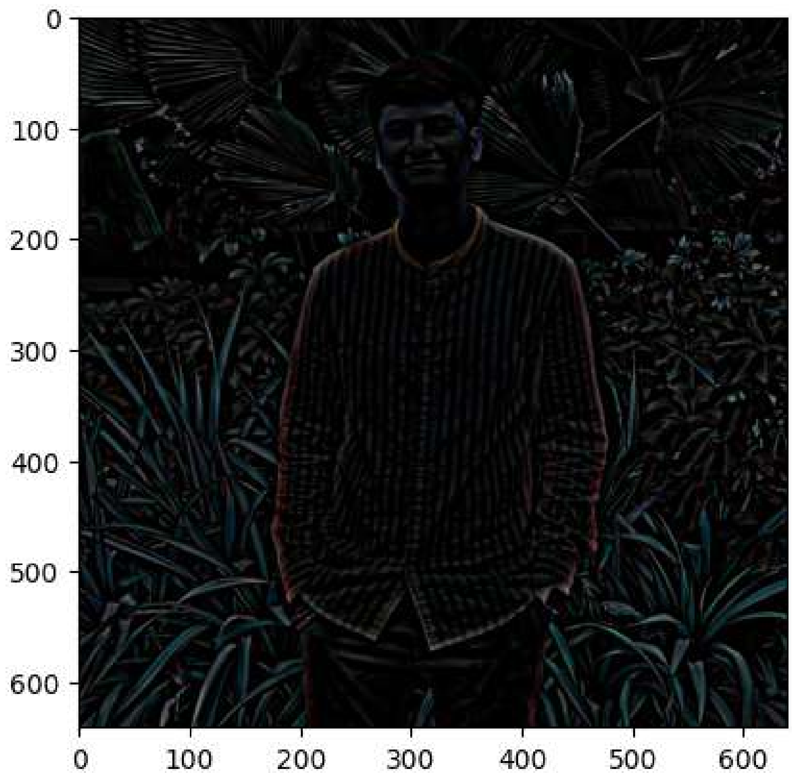
```
(641, 640, 3)
```

[46]:
```
<matplotlib.image.AxesImage at 0x7f824b7bd690>
```



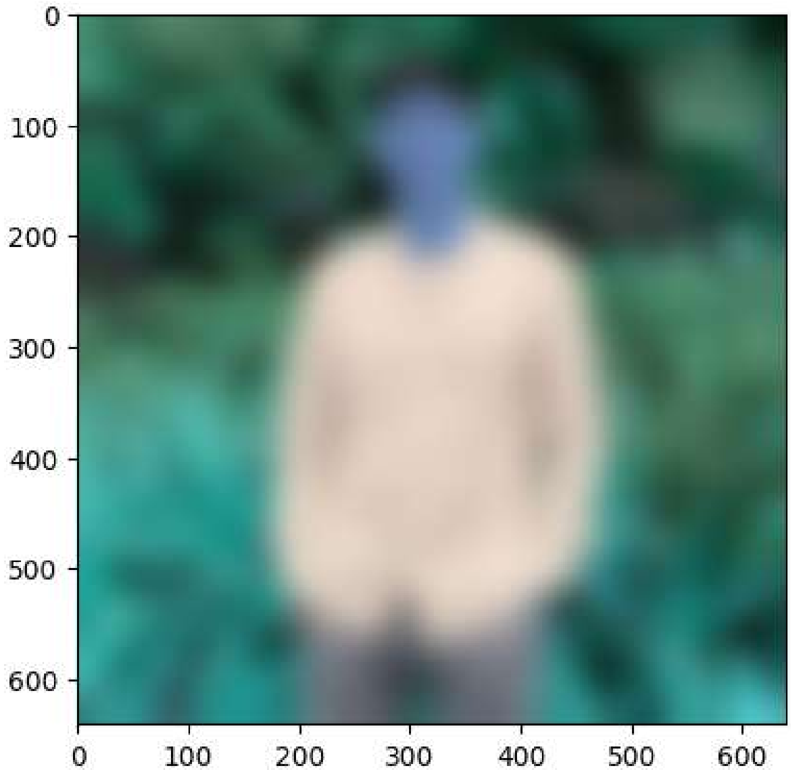
[48]:
```
hpf = cv2.subtract(image,cv2.GaussianBlur(image, (21, 21),7)) # The high pass
 ↪filter is created by subtracting the Gaussian Blurred image from original
 ↪image
plt.imshow(hpf)
```

[48]: <matplotlib.image.AxesImage at 0x7f8248a8c250>



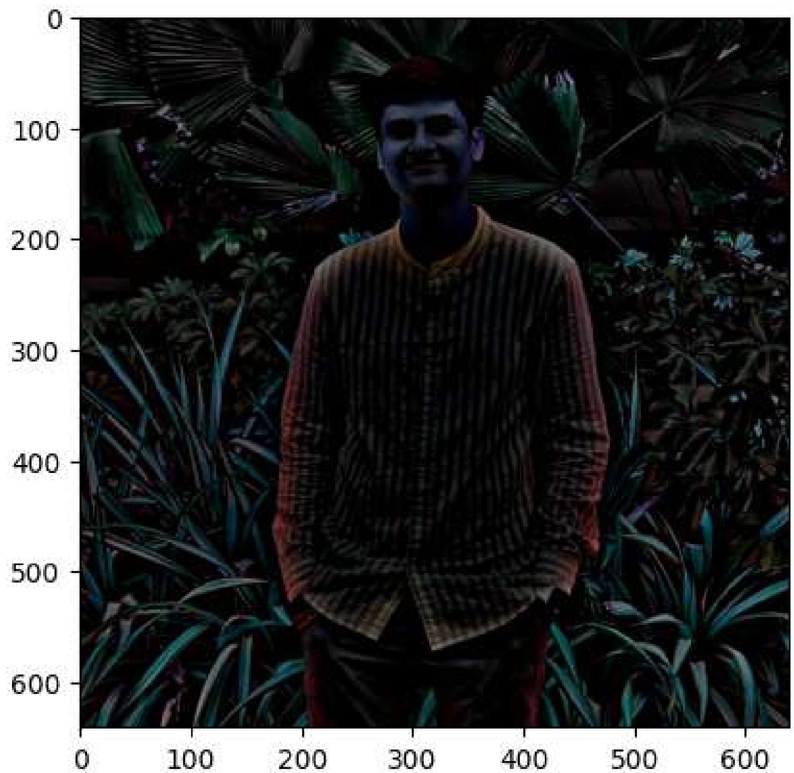

```
[51]: # Low frequency image can be obtained by applying the Gaussian Blur on the␣
      ↪original filter



      # Using cv2.blur() method
      low_frequency_image = cv2.GaussianBlur(image, ksize1, 0)
      plt.imshow(low_frequency_image)
      # Displaying the image
```

```
[51]: <matplotlib.image.AxesImage at 0x7f824b915900>
```



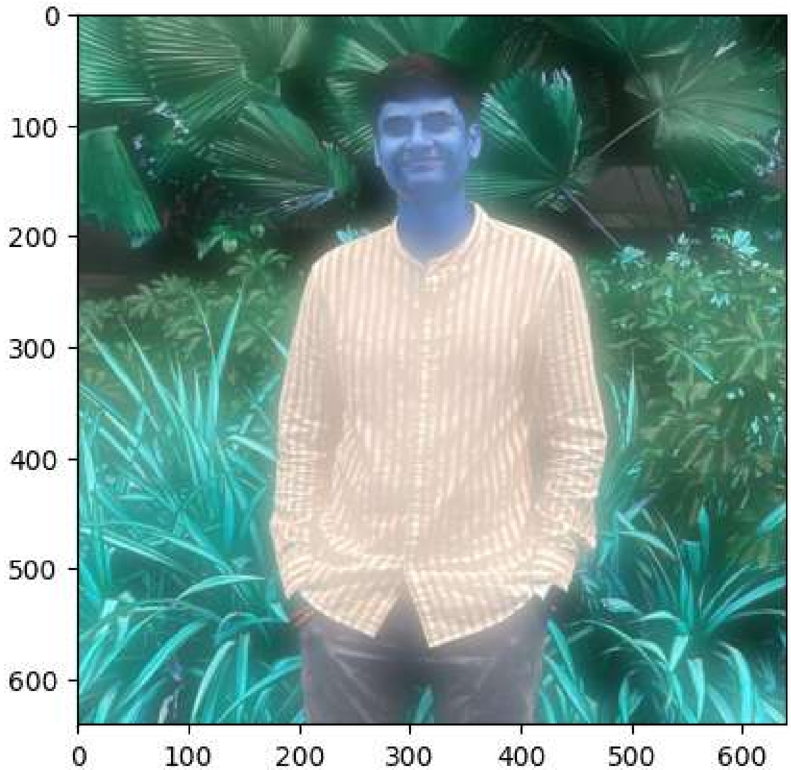
[53]:
```
# High frequency image can be  created in many ways, here I am subtracting the␣
 ↪lareg_blur image from original image

high_frequency_image = cv2.subtract(image , cv2.GaussianBlur(image, ksize1, 0))
plt.imshow(high_frequency_image)
```

[53]: <matplotlib.image.AxesImage at 0x7f8245c37a90>



[54]:
```
# Creating a hybrid image by adding the low frequency and high frequency images
plt.imshow(cv2.add(low_frequency_image,high_frequency_image))
```

[54]: <matplotlib.image.AxesImage at 0x7f8248a4a410>


0
100
200
300
400
500
600
0
100
200
300
400
500
600