Assignment 3 (60/25 points)

Histogram Equalization *(20/10)*

a) Read and display an image of your choice. (To help you with the next problem, look for an image that has interesting objects of different intensities.)
b) Calculate and display the histogram of this image.
c) Enhance the contrast of the intensity image using histogram equalization and display both the uniform histogram and the newly enhanced intensity image.
d) Explain why the two histograms (of the original image and of the enhanced image) are different.

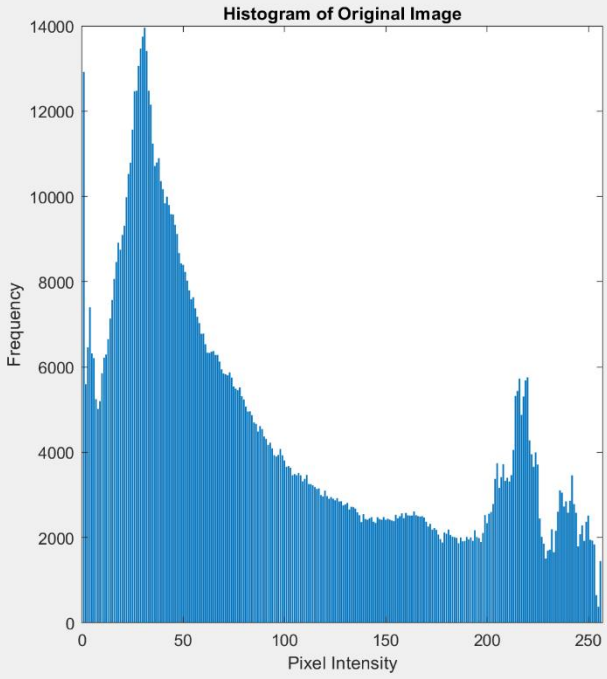
CODE:

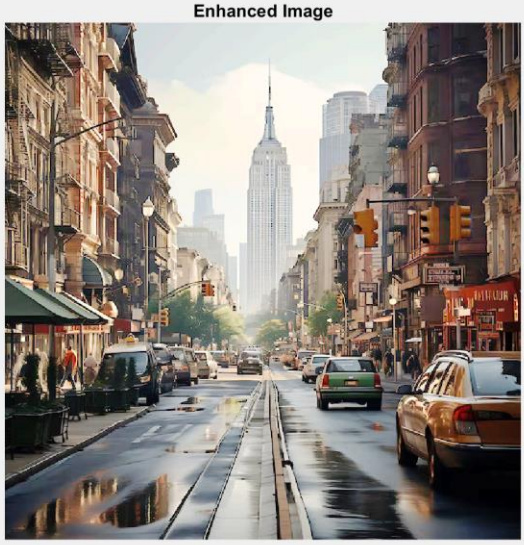
```
% a) Read and display an image of your choice.
nyc_image = imread('nyc_view2.jpg');
figure;
subplot(1, 2, 1);
imshow(nyc_image);
title('Original Image');

% b) Calculate and display the histogram of this image.
hist_nyc = imhist(nyc_image);
subplot(1, 2, 2);
bar(hist_nyc);
title('Histogram of Original Image');
xlabel('Pixel Intensity');
ylabel('Frequency');

% c) Enhance the contrast of the intensity image using histogram equalization
% and display both the uniform histogram and the newly enhanced intensity image.
equalized_image = histeq(nyc_image);
equalized_hist = imhist(equalized_image);
figure;
subplot(1,2,1);
imshow(equalized_image);
title('Enhanced Image');

subplot(1,2,2)
bar(equalized_hist);
title('Histogram of Enhanced Image');
xlabel('Pixel Intensity');
ylabel('Frequency');
```

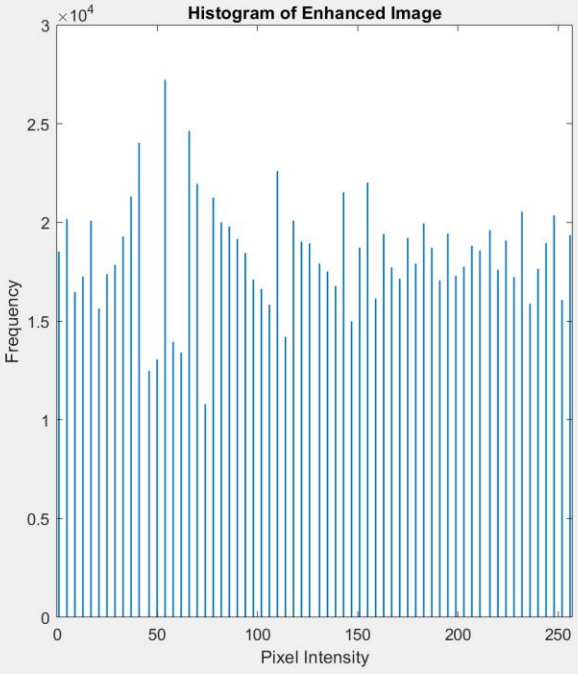
O/P:

The dissimilarity between the histograms of the original and enhanced images arises from the fact that histogram equalization alters the distribution of pixel intensities, resulting in a more even spread of intensity values in the enhanced image. This procedure effectively enhances the image's overall contrast by expanding the range of pixel values, rendering darker regions even darker and brighter regions even brighter. Consequently, the enhanced image exhibits a more uniform histogram in contrast to the original image, where pixel intensities may be clustered within a specific range.


Original Image


Histogram of Original Image
Frequency
Pixel Intensity
0
2000
4000
6000
8000
10000
12000
14000
0
50
100
150
200
250


Enhanced Image


Histogram of Enhanced Image
×10^4
Frequency
Pixel Intensity
0
0.5
1
1.5
2
2.5
3
0
50
100
150
200
250

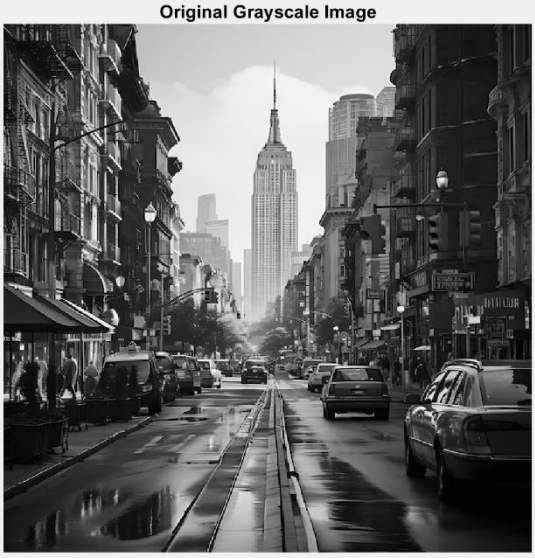
481 Students: ***(10/0)*** Apply a local enhancement approach on this image and show your results. Before you start, consider how your image might call for a particular window size. For fun, you might want to try a few different window sizes. One student actually put the window size in a loop from 1 to the image size and showed the results in a video. The gauntlet has been thrown. ☺

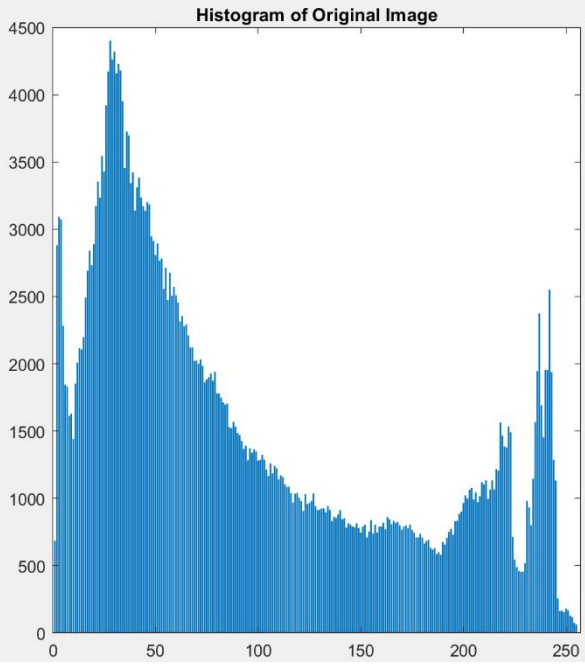
CODE:

```
% grayscale the original image
nyc_image = imread('nyc_view2.jpg');
gray_image = rgb2gray(nyc_image);
hist_gray = imhist(gray_image);

figure;
subplot(1,2,1);
imshow(gray_image);
title('Original Grayscale Image');

subplot(1,2,2);
bar(hist_gray);
title('Histogram of Original Image');

% Window sizes
window_sizes = [16, 32, 64, 128];

% Apply local enhancement using the window size list
for i = 1:length(window_sizes)
    window_size = window_sizes(i);

    %Calling the 'ALE' function to apply local enhancement
    enhanced_image = ALE(gray_image, window_size);

    figure;
    subplot(1,2,1);
    imshow(enhanced_image);
    title(['Enhanced Grayscale Image (Window Size = ' num2str(window_size) ')']);

    hist_enhanced = imhist(enhanced_image);
    subplot(1,2,2);
    bar(hist_enhanced);
    title(['Histogram of Enhanced Image'])

end

% writing a function to apply adapthisteq with varied window sizes
function enhanced_image = ALE(image, window_size)
    enhanced_image = adapthisteq(image, 'NumTiles', [window_size, window_size]);
end
```

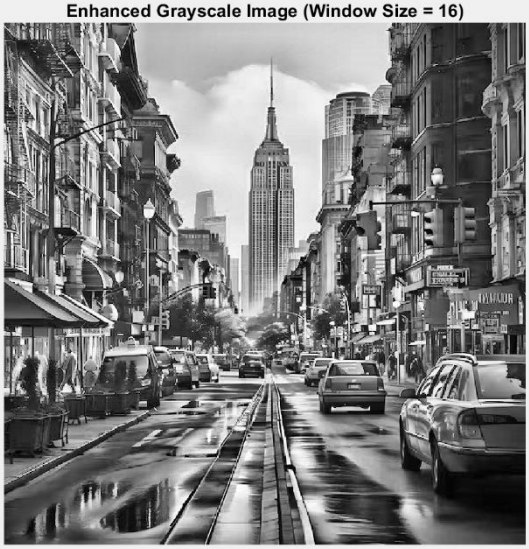
O/P:

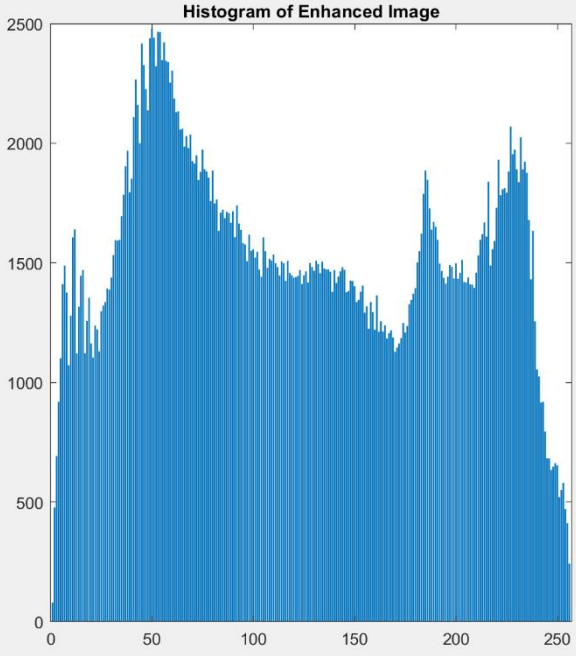
Original Grayscale Image



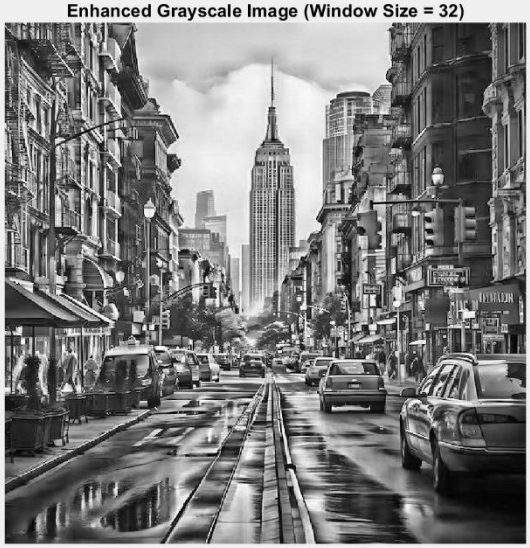


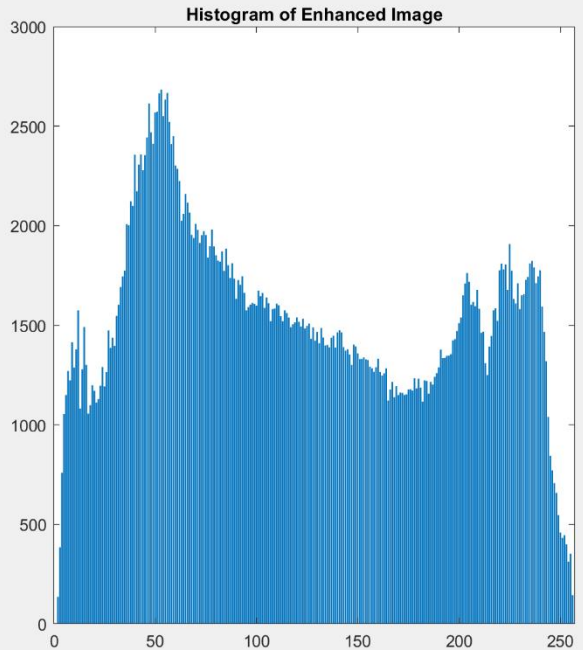
Enhanced Grayscale Image (Window Size = 16)



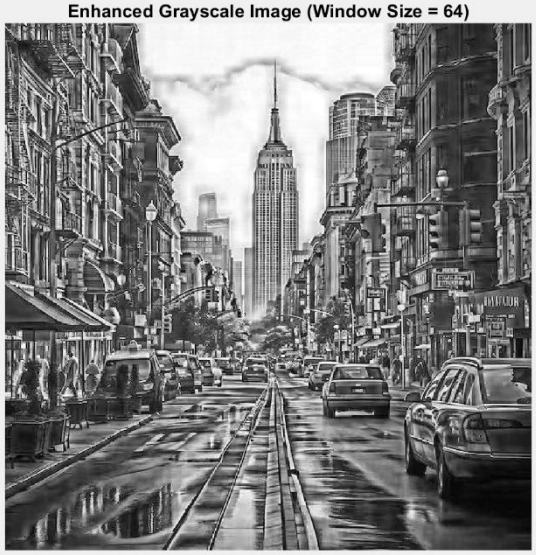


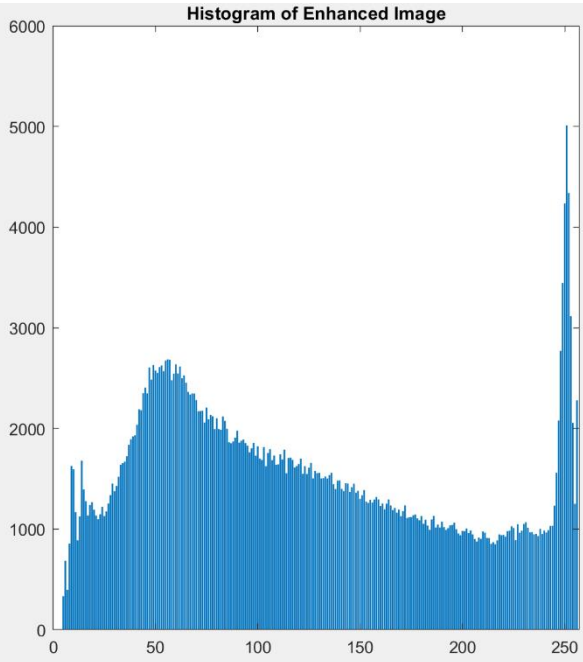
Enhanced Grayscale Image (Window Size = 32)

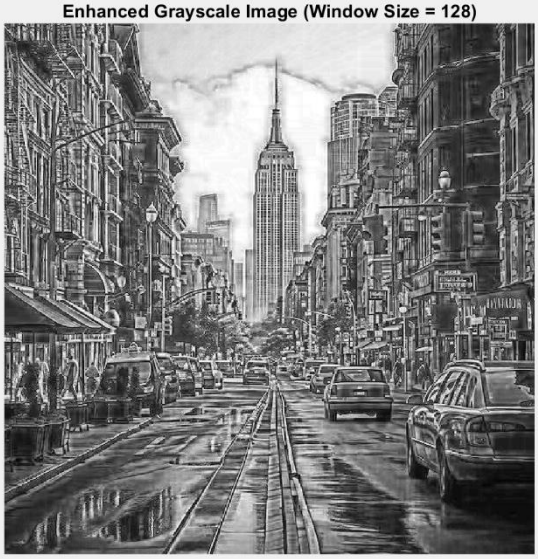


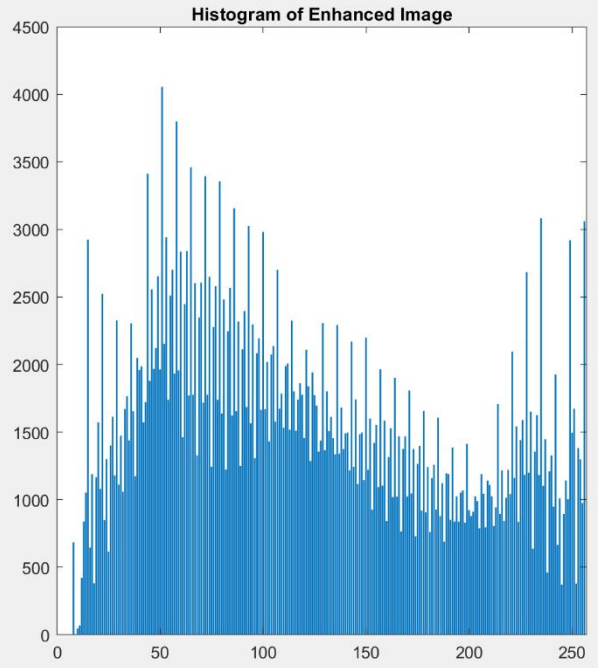


Enhanced Grayscale Image (Window Size = 64)









Filtering: (*25/15*)

a) **481 Students: (*10/0)*** write a function to perform filtering. Do not use any of the prewritten functions; your function should process the image pixel by pixel and explicitly calculate the sum of products for each pixel in the image. You can use whatever simplifying assumptions you want for pixels on the border of the image. State those assumptions.

I've created the customFilter function to filter an input image using a specified kernel. First, I determine the sizes of the input image and kernel. Then, I calculate the border size for valid convolution by dividing the kernel size by 2. I initialize an empty matrix to store the filtered image. I use nested loops to iterate through each pixel in the input image and, for each pixel, calculate the sum of products between the image and the kernel elements in its neighborhood. I ensure the coordinates are within the image bounds and handle boundary conditions. The result is stored in the filtered_image matrix. After filtering, I normalize and clip pixel values to the 0-255 range. This function allows for various image processing tasks, such as blurring, sharpening, edge detection, and more, by providing different kernels.

CODE:

```
nyc_image = imread('nyc_view2.jpg');
gray_image = rgb2gray(nyc_image);
kernel =  [1/9, 1/9, 1/9;1/9, 1/9, 1/9;1/9, 1/9, 1/9];
kernel_2 =  [1/16, 1/8, 1/16;1/8, 1/4, 1/8;1/16, 1/8, 1/16];
kernel_6 = [0,0,0;0,1,0;0,0,0];
kernel_7 = [-1,0,1;-1,0,1;-1,0,1];
kernel_8 = [-1,-1,-1;0,0,0;1,1,1];
```

```
filtered_image = customFilter(gray_image,kernel);
filtered_image_2 = customFilter(gray_image,kernel_2);
filtered_image_6 = customFilter(gray_image,kernel_6);
filtered_image_7 = customFilter(gray_image,kernel_7);
filtered_image_8 = customFilter(gray_image,kernel_8);

figure;
imshow(filtered_image, 'DisplayRange', [0, 255]);
title('Box Blur Filter');

figure;
imshow(filtered_image_2,'DisplayRange', [0, 255]);
title('Gaussian Filter');

figure;
imshow(filtered_image_6,'DisplayRange', [0, 255]);
title('Point Filter');

figure;
imshow(filtered_image_7,'DisplayRange', [0, 255]);
title('Prewitt Horizontal Filter');

figure;
imshow(filtered_image_8,'DisplayRange', [0, 255]);
title('Prewitt Vertical Filter');

function filtered_image = customFilter(image, kernel)
    [image_height, image_width] = size(image);
    [kernel_height, kernel_width] = size(kernel);

    border_height = floor(kernel_height / 2);
    border_width = floor(kernel_width / 2);

    filtered_image = zeros(image_height, image_width);

    for y = 1:image_height
        for x = 1:image_width
            sum_of_products = 0;

            for ky = 1:kernel_height
                for kx = 1:kernel_width

                    xi = x + kx - border_width;
                    yi = y + ky - border_height;

                    if xi >= 1 && xi <= image_width && yi >= 1 && yi <= image_height
```

```
                sum_of_products = sum_of_products + image(yi, xi) * kernel(ky, kx);
            end
        end
    end
    % Assign the result to the filtered image
    filtered_image(y, x) = sum_of_products;
  end
end
% Normalize and clip pixel values
filtered_image = min(max(filtered_image, 0), 255);
end
```

O/P:

Gaussian Filter

Point Filter

Prewitt Horizontal Filter

Prewitt Vertical Filter

b) **All Students:** perform filtering on the image of your choice using a Prewitt filter, a Sobel filter, a point filter, and the blurring filter. Use built-in functions to do the filtering. **481 students:** use both your function and a library function so that you can compare your results to the library results.

CODE:

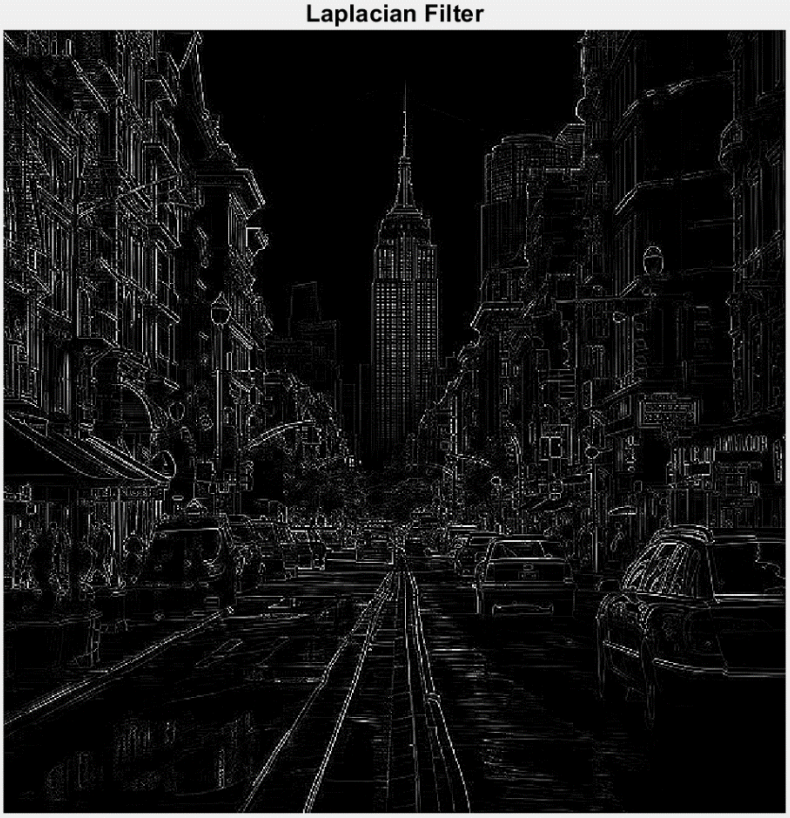
```
nyc_image = imread('nyc_view2.jpg');
gray_image = rgb2gray(nyc_image);

% Define kernel matrices
kernel = 1/9 * ones(3, 3);
kernel_2 = fspecial('gaussian', [3 3], 0.5);
kernel_3 = [0 1 0; 1 -4 1; 0 1 0];
kernel_4 = fspecial('sobel');
kernel_5 = kernel_4';
kernel_6 = [0 0 0; 0 1 0; 0 0 0];
kernel_7 = [1 0 -1; 1 0 -1; 1 0 -1];
kernel_8 = kernel_7';

% Apply filters using imfilter
filtered_image = imfilter(gray_image, kernel);
filtered_image_2 = imfilter(gray_image, kernel_2);
filtered_image_3 = imfilter(gray_image, kernel_3);
filtered_image_4 = imfilter(gray_image, kernel_4);
filtered_image_5 = imfilter(gray_image, kernel_5);
filtered_image_6 = imfilter(gray_image, kernel_6);
filtered_image_7 = imfilter(gray_image, kernel_7);
filtered_image_8 = imfilter(gray_image, kernel_8);

% Display filtered images
figure;
imshow(filtered_image, 'DisplayRange', [0, 255]);
title('Box Blur Filter');

figure;
imshow(filtered_image_2, 'DisplayRange', [0, 255]);
title('Gaussian Filter');

figure;
imshow(filtered_image_3, 'DisplayRange', [0, 255]);
title('Laplacian Filter');

figure;
imshow(filtered_image_4, 'DisplayRange', [0, 255]);
```

```
title('Sobel Horizontal Filter');

figure;
imshow(filtered_image_5, 'DisplayRange', [0, 255]);
title('Sobel Vertical Filter');

figure;
imshow(filtered_image_6, 'DisplayRange', [0, 255]);
title('Point Filter');

figure;
imshow(filtered_image_7, 'DisplayRange', [0, 255]);
title('Prewitt Horizontal Filter');

figure;
imshow(filtered_image_8, 'DisplayRange', [0, 255]);
title('Prewitt Vertical Filter');
```

O/P:

Gaussian Filter

Laplacian Filter

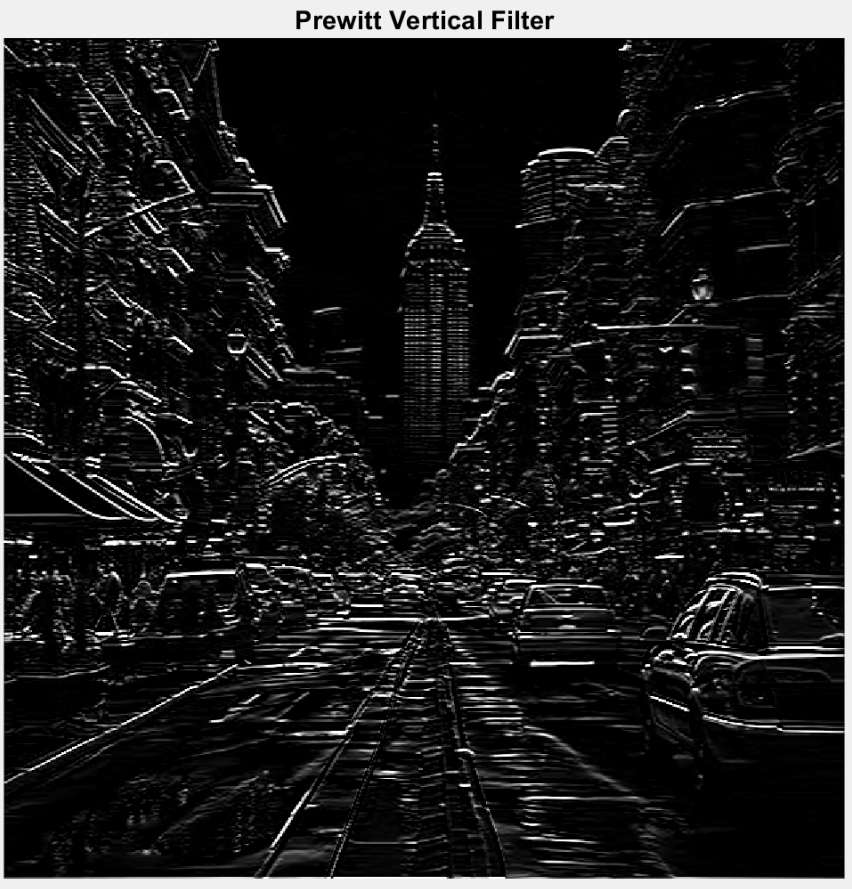


Sobel Horizontal Filter

Sobel Vertical Filter

Point Filter

Prewitt Horizontal Filter



Bit Plane Splicing ***(15/0)***

Bit place splicing (**https://en.wikipedia.org/wiki/Bit_plane**) is a simple form of frequency analysis in which the frequencies are defined by the bits representing the intensity of the pixels. Write a program to perform bit-place splicing on an image such that you can generate a figure similar to the one shown in the Wikipedia article: your original image and each of the 8 bit planes in it. Perhaps the key lesson is that each bit-position represents a different binary image.

Then, "assemble" the original image by successively adding bit planes to the most significant bit plane. You will have 7 new images, which will be the combination of bit planes 7 and 6; 7 and 6 and 5; 7 and 6 and 5 and 4; … all bit planes, which should be the original image. State which bit plane you feel you could stop at and still get a good visual match with the original image.

```
originalImage = imread('nyc_view2.jpg');
originalImage = rgb2gray(originalImage);

figure;
imshow(originalImage);
title('Original Image');
```

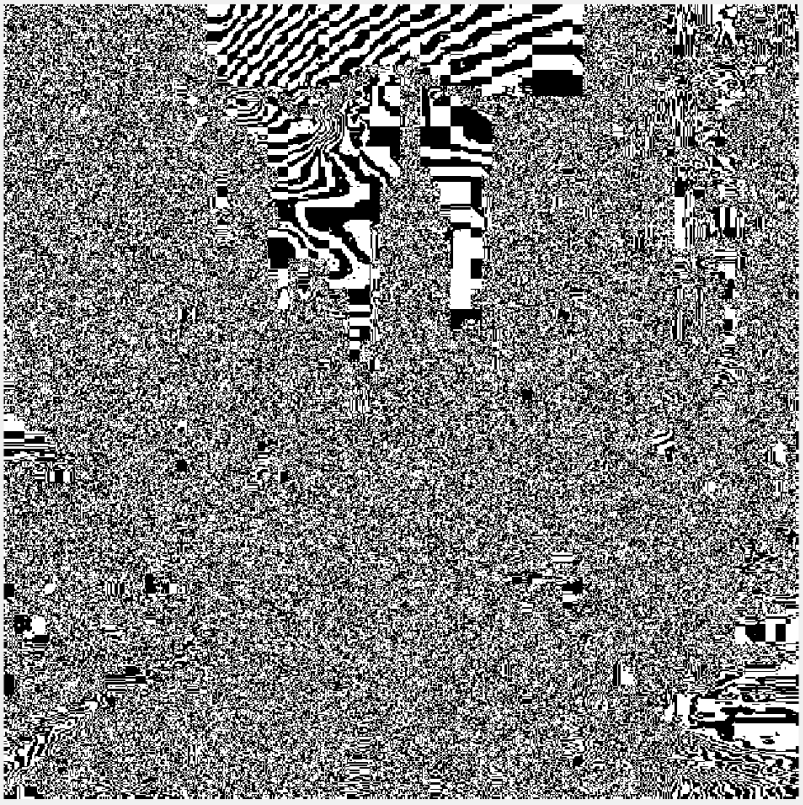
```
[height, width] = size(originalImage);
bitPlanes = cell(1, 8);
for bit = 1:8
    mask = bitget(originalImage, bit);

    bitPlanes{bit} = uint8(mask * 255);
end

for bit = 1:8
    figure;
    imshow(bitPlanes{bit});
    title(['Bit Plane ', num2str(bit)]);
end

reconstructedImage = bitPlanes{8};
reconstructedImages = cell(1, 7);

for bit = 7:-1:1
    reconstructedImage = reconstructedImage + bitPlanes{bit};
    reconstructedImages{bit} = reconstructedImage;
end

for bit = 1:7
    figure;
    imshow(reconstructedImages{bit});
    title(['Reconstructed Image (Bit Planes 7-', num2str(bit), ')']);
end
```

The code processes an input grayscale image, decomposes it into its eight binary bit planes (from least to most significant), and then reassembles the original image by summing the bit planes in order from the most significant bit to the least significant bit. It provides a visual representation of the individual bit planes and their cumulative impact on the final image. The code is a useful tool for understanding the role of different bits in representing image details and can be employed for educational or analytical purposes in image processing.

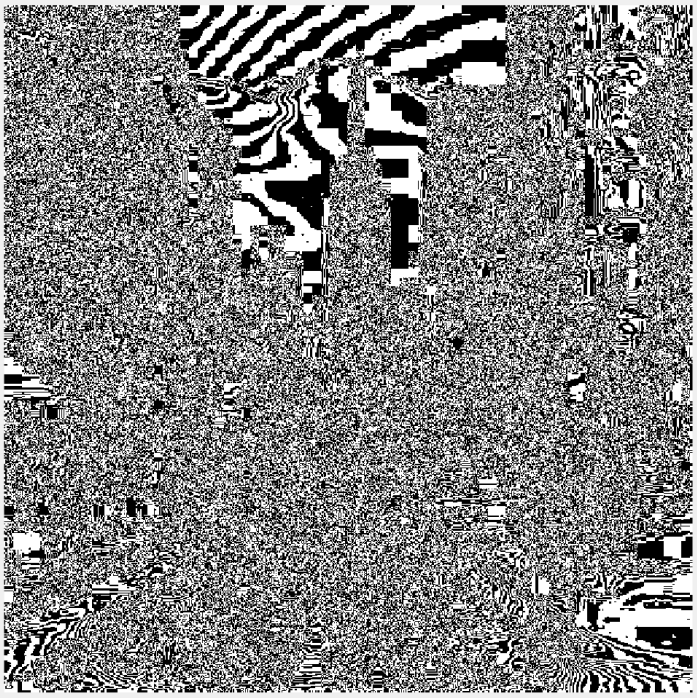
O/P:

Original Image

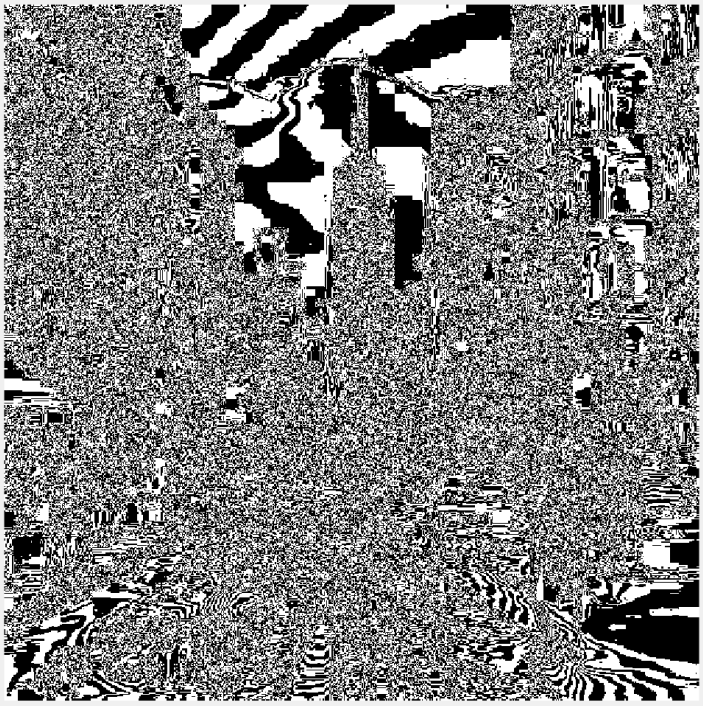
Bit Plane 1



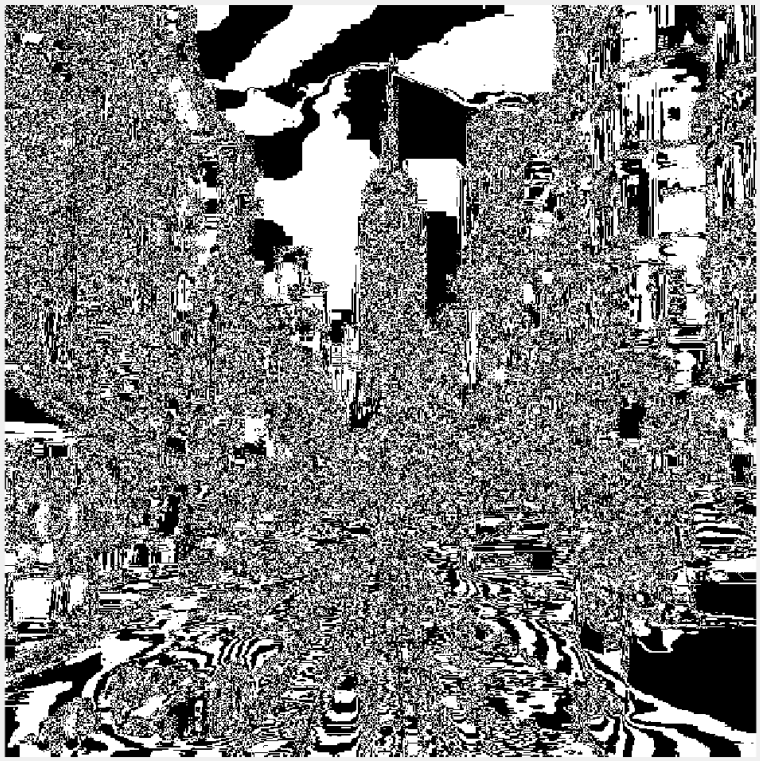
Bit Plane 2



Bit Plane 3



Bit Plane 4



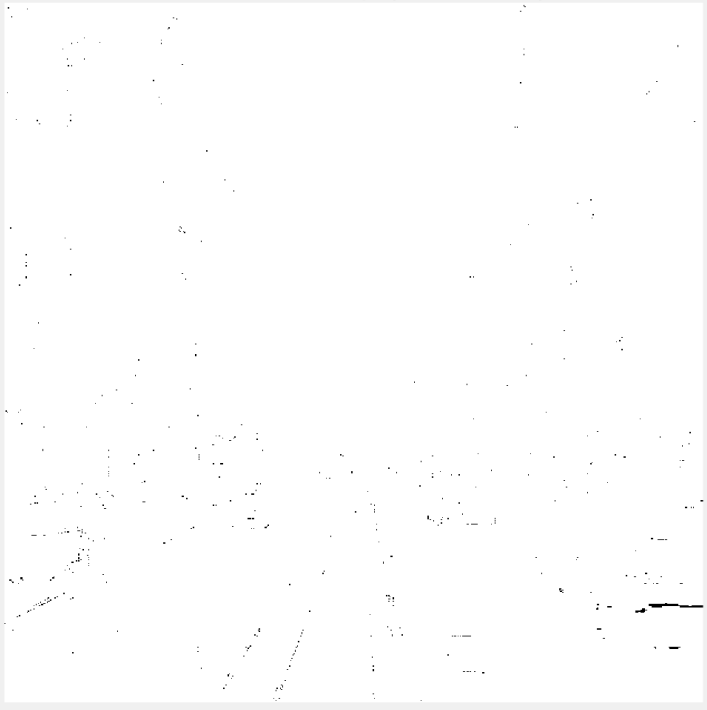
Bit Plane 5

Bit Plane 6

Bit Plane 7

Bit Plane 8

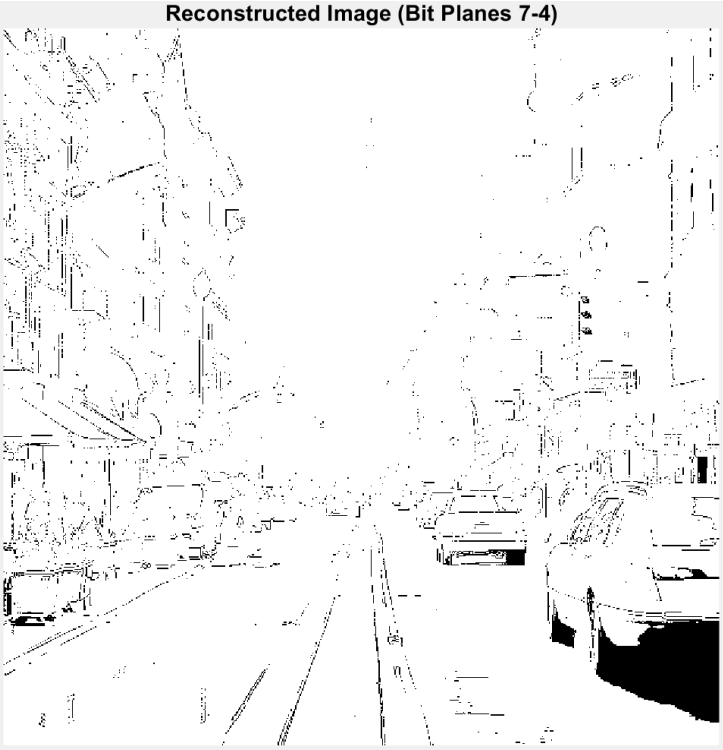
Reconstructed Image (Bit Planes 7-1)



Reconstructed Image (Bit Planes 7-2)

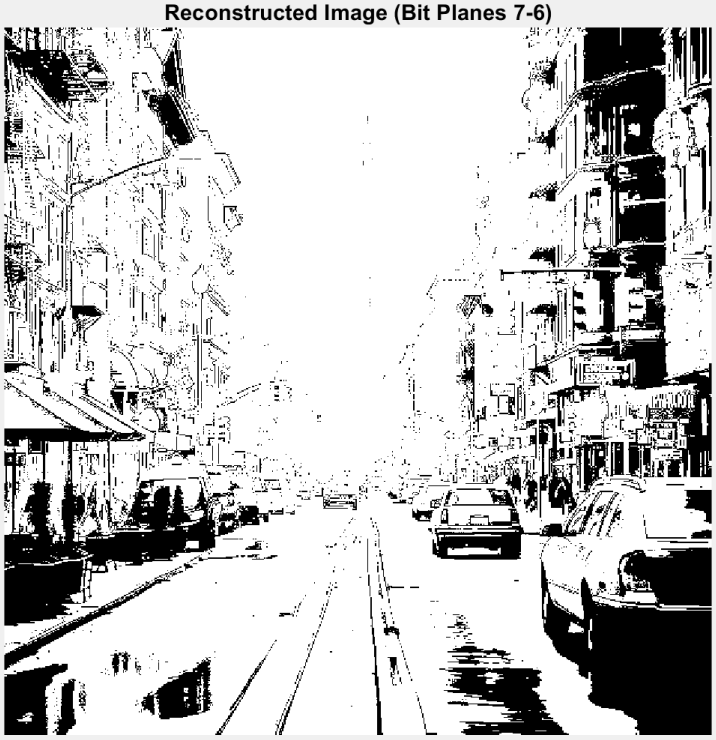
Reconstructed Image (Bit Planes 7-3)

Reconstructed Image (Bit Planes 7-4)

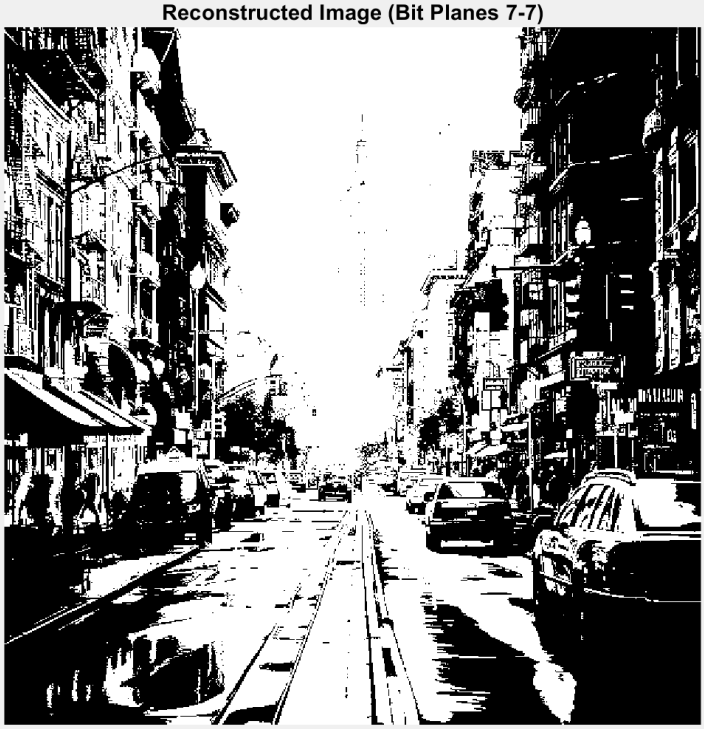


Reconstructed Image (Bit Planes 7-5)

Reconstructed Image (Bit Planes 7-6)



Reconstructed Image (Bit Planes 7-7)



General submission instructions:

(a) Be kind to your aging, over-worked professor and submit only a single document. This can be pdf, MS Word, OpenOffice, etc. Do not submit a zip file.

(b) Your single document should include the input image for your problem, if required, and answers to each of the sub-problems (text, image or both, as appropriate). Your document should also include code that you wrote to generate your answers.

(c) You may use any images you like for the programming; I encourage you to use images that might be useful/interesting for your final project.

(d) Feel free to use whatever functions MatLab supplies. Also feel free to write your own, if you are so inclined; it will take more time, but you will gain a deeper understanding of the material.

(e) Point values for each question are indicated as ***(x/y)*** in which ***x*** is the point value for 481 students and ***y*** is the point value for 381 students.