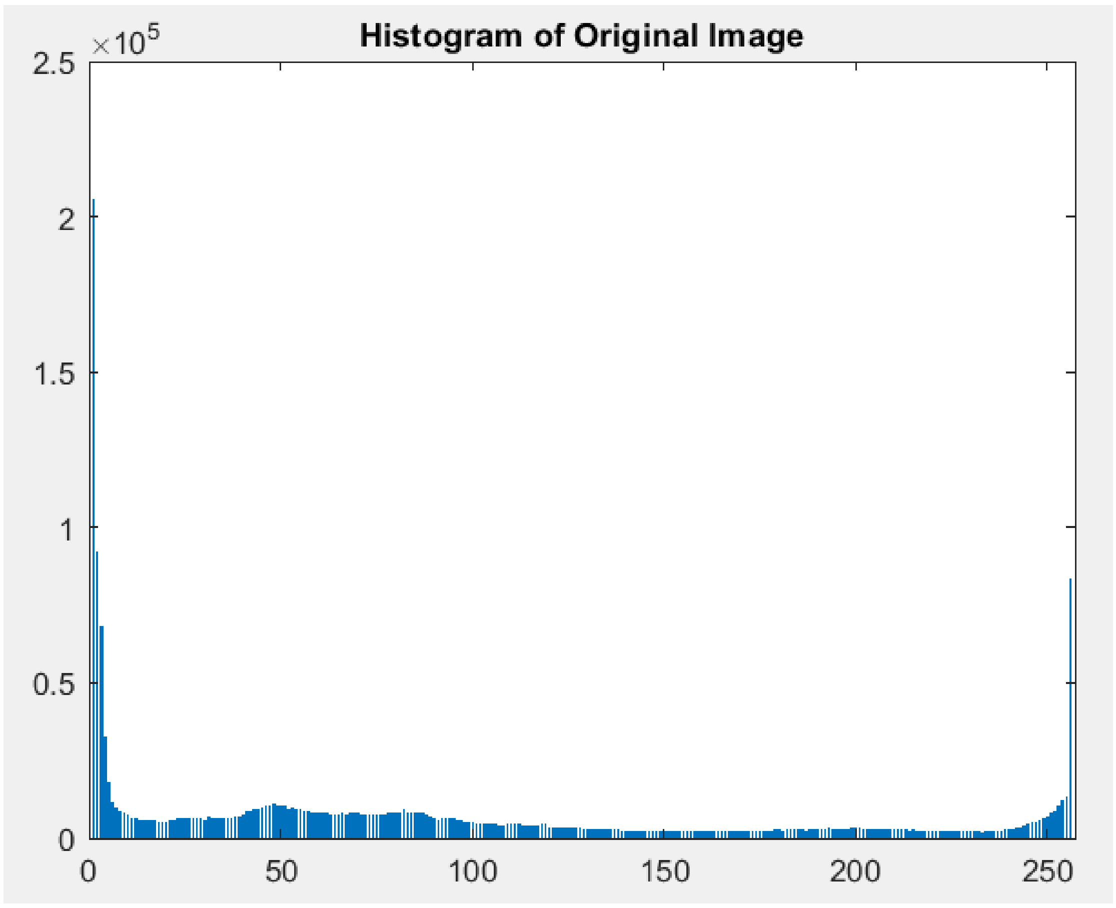
1)

a) Read and display an image of your choice

```
originalImage = imread('butterfly.jpg');
imshow(originalImage);
```

b)

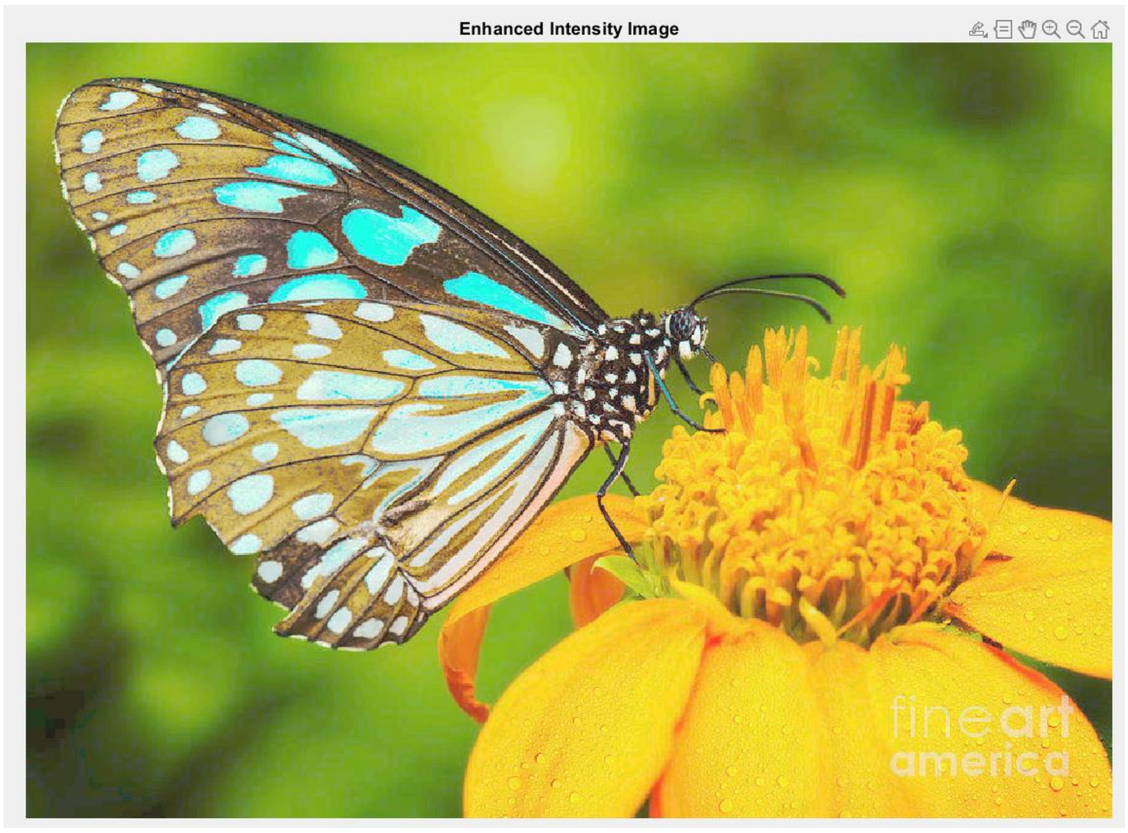
```
histogram = imhist(originalImage);
figure;
bar(histogram);
title('Histogram of Original Image');
```

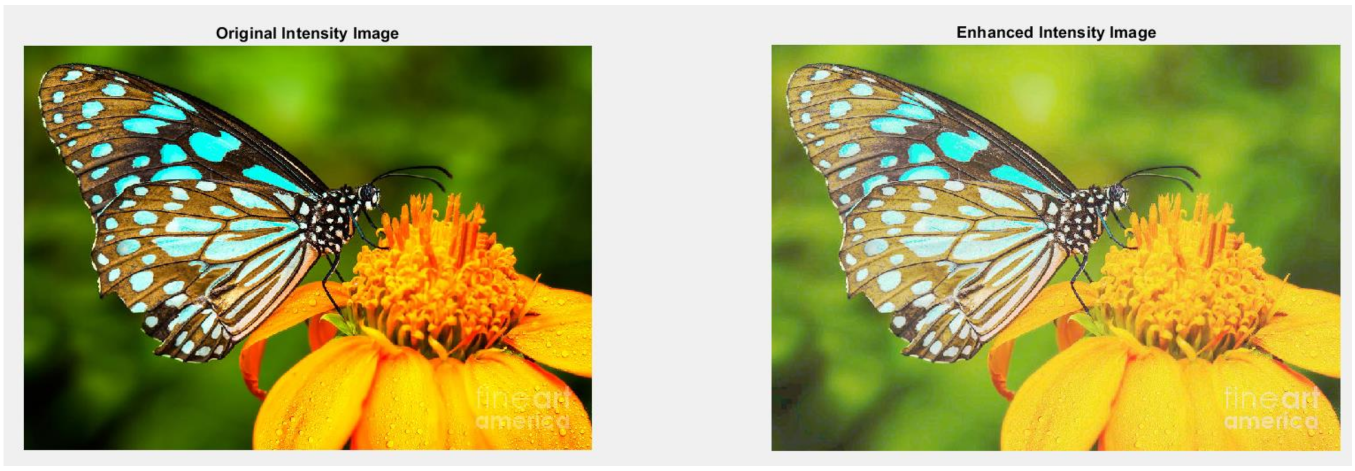


c)

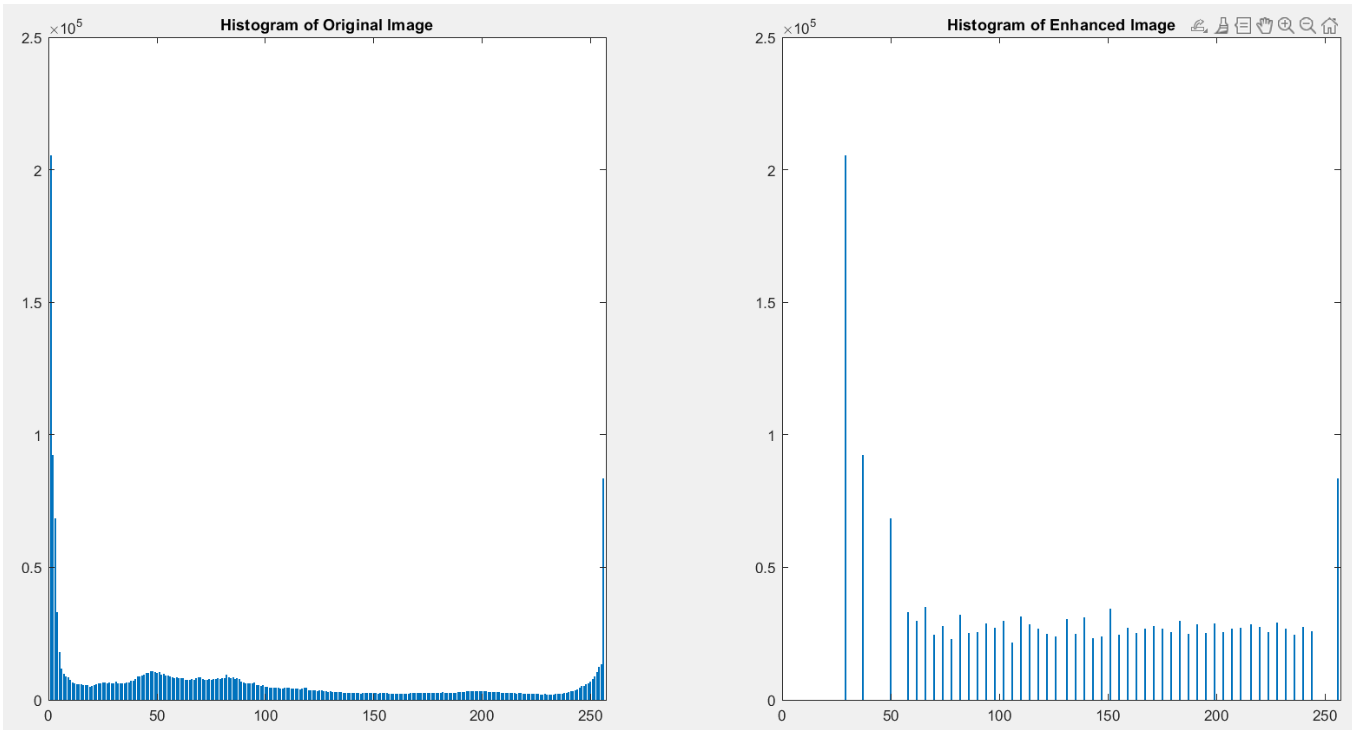
```
% Read the intensity image (grayscale)
intensityImage = imread('butterfly.jpg');

% Calculating the histogram of original intensity image
originalHistogram = imhist(intensityImage);

% Apply histogram equalization to enhance contrast
enhancedImage = histeq(intensityImage);

% Calculate the histogram of the enhanced image
enhancedHistogram = imhist(enhancedImage);

% Display the original and enhanced intensity images side by
side
figure;
subplot(1,2,1);
imshow(intensityImage);
title('Original Intensity Image');
```

```
subplot(1,2,2);
imshow(enhancedImage);
title('Enhanced Intensity Image');

% Display the original and enhanced histograms
figure;
subplot(1,2,1);
bar(originalHistogram);
title('Histogram of Original Image');

subplot(1,2,2);
bar(enhancedHistogram);
title('Histogram of Enhanced Image');
```





```
% Display the original and enhanced histograms
figure;
subplot(1,2,1);
bar(originalHistogram);
title('Histogram of Original Image');

subplot(1,2,2);
bar(enhancedHistogram);
title('Histogram of Enhanced Image');
```



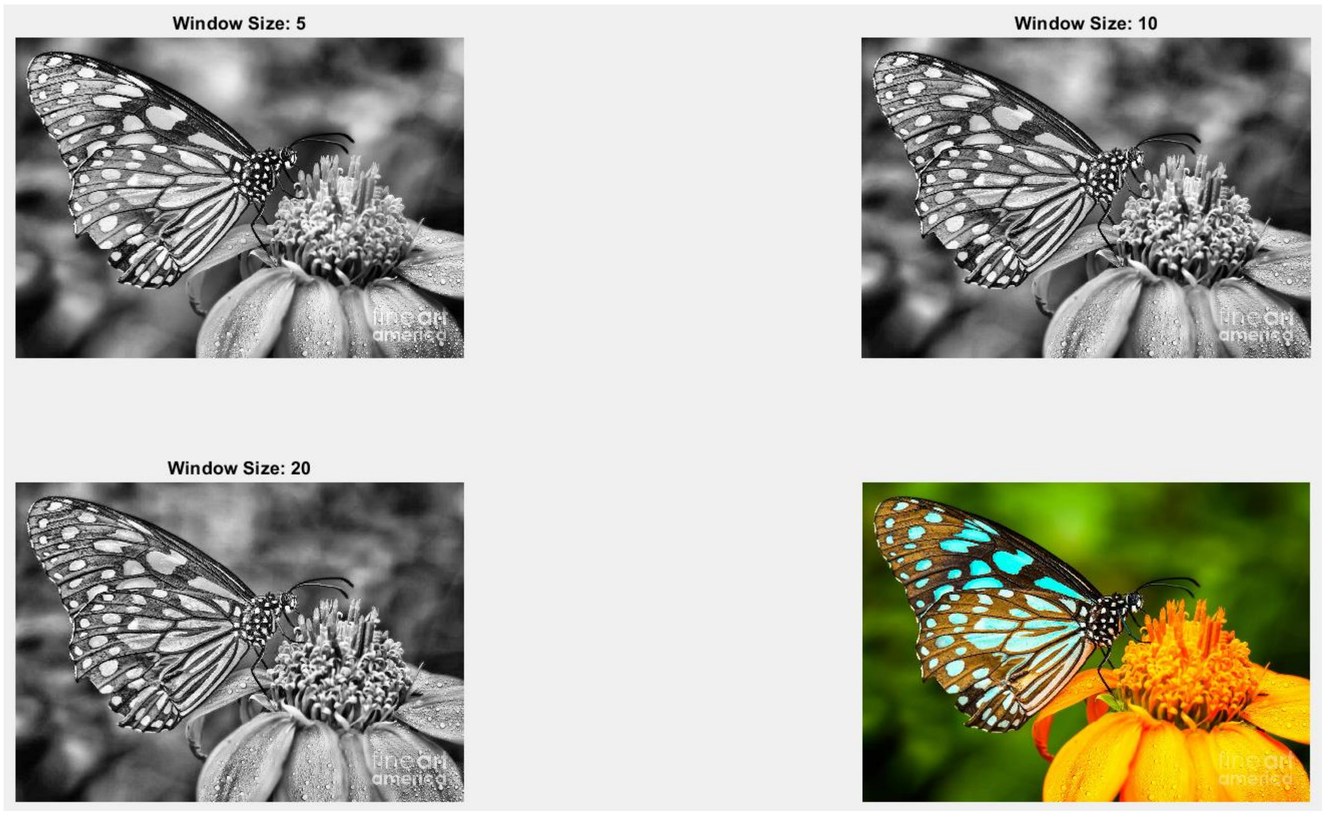
Original Image histogram

- Represents how pixel intensities are distributed in the original image. Has peaks and valleys, indicating areas of high and low intensity.
- Reflects the image's inherent contrast and brightness characteristics.
- Enhanced Image Histogram (After Histogram Equalization)

Enhanced image histogram

- The result of a process that redistributes pixel intensities to create a more uniform distribution.
- Aims to spread out the intensities across the entire range, effectively maximizing the use of the available intensity levels.
- Leads to a more balanced representation of intensities, resulting in improved image contrast.

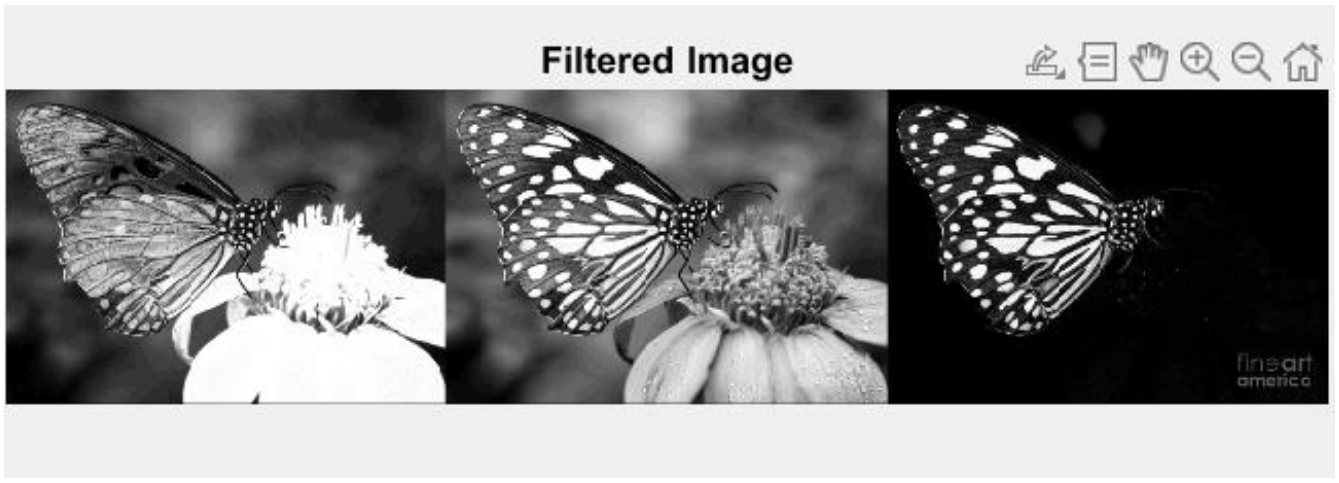
In summary, the histograms differ because the original image's histogram shows the inherent intensity distribution, while the enhanced image's histogram reflects the equalization process that spreads out intensities for improved contrast.

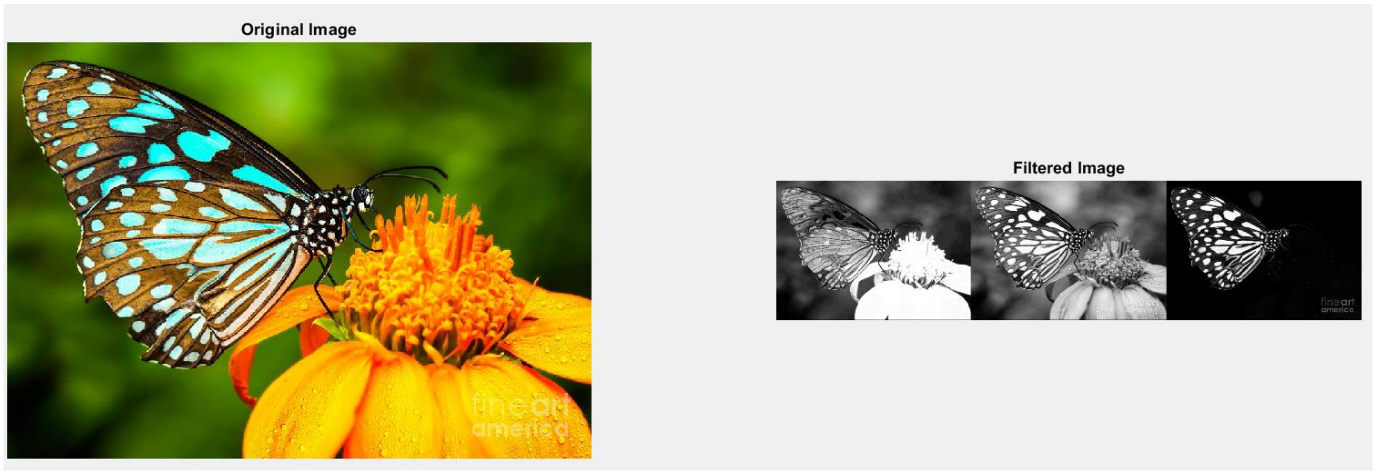


2)A

```
% Read the image

originalImage = imread('butterfly.jpg');


% Define a filter kernel (for example, a simple averaging filter)

filterKernel = ones(3, 3) / 9;  % 3x3 averaging filter


% Call the custom image filtering function

filteredImage = customImageFilter(originalImage, filterKernel);


% Display the filtered image

figure;

imshow(filteredImage, []);

title('Filtered Image');


% Custom Image Filtering Function

function filteredImage = customImageFilter(image, filterKernel)

   % Get the dimensions of the image and kernel

   [imageHeight, imageWidth] = size(image);

   [kernelHeight, kernelWidth] = size(filterKernel);


   % Calculate the border padding for the kernel

   padSize = floor((kernelHeight - 1) / 2);


   % Initialize the output image

   filteredImage = zeros(imageHeight, imageWidth);


   % Iterate through the image
```

```
  for y = 1:imageHeight
    for x = 1:imageWidth
      % Initialize the sum of products
      productSum = 0;

      % Iterate through the kernel
      for ky = 1:kernelHeight
        for kx = 1:kernelWidth
          % Calculate the corresponding pixel in the image
          px = x - padSize + kx;
          py = y - padSize + ky;

          % Check if the pixel is within the image bounds
          if px > 0 && px <= imageWidth && py > 0 && py <= imageHeight
            % Calculate the product and accumulate
            productSum = productSum + image(py, px) * filterKernel(ky, kx);
          end
        end
      end

      % Store the result in the filtered image
      filteredImage(y, x) = productSum;
    end
  end
endend
```





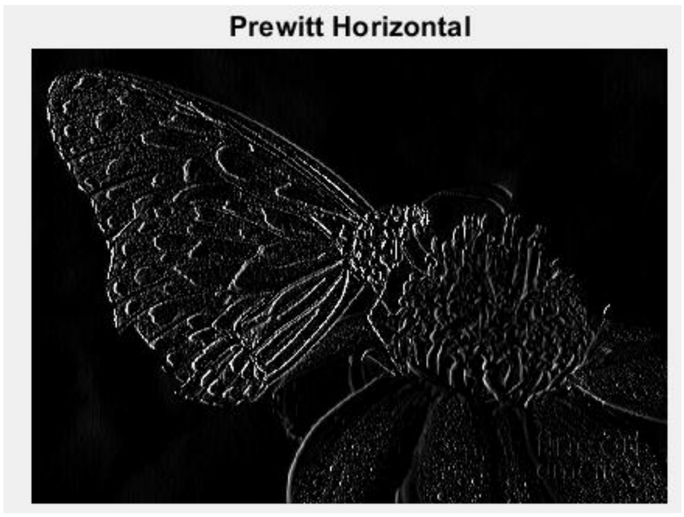
```
[[          % Check if the pixel is within the image bounds
            if px > 0 && px <= imageWidth && py > 0 && py <= imageHeight
              % Calculate the product and accumulate
              productSum = productSum + image(py, px) * filterKernel(ky, kx);
            end       ]]
```

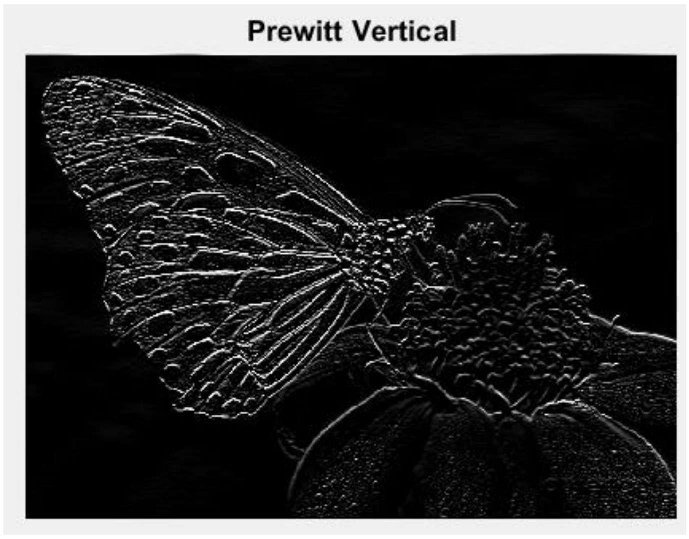
Explanation :

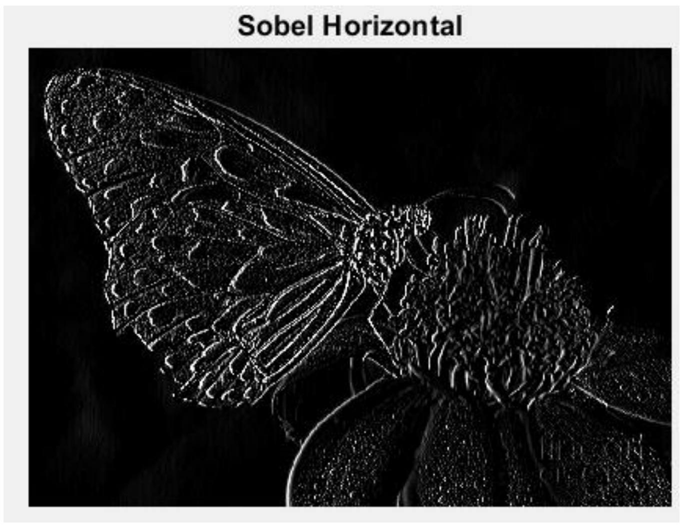
For pixels on the border of the image, we assume that any pixels outside the image boundary are treated as if they have a value of 0. This is a common approach for simplicity when performing convolution-based filtering operations on images.

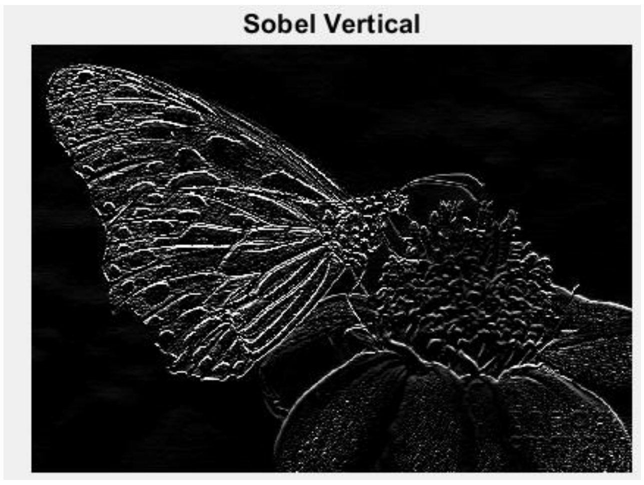
The function takes an input image and a filter kernel and processes the image pixel by pixel, explicitly calculating the sum of products for each pixel using the given filter kernel.

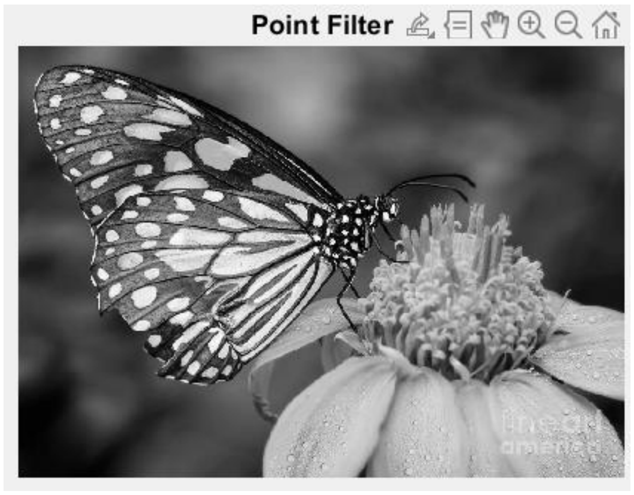
2 B)

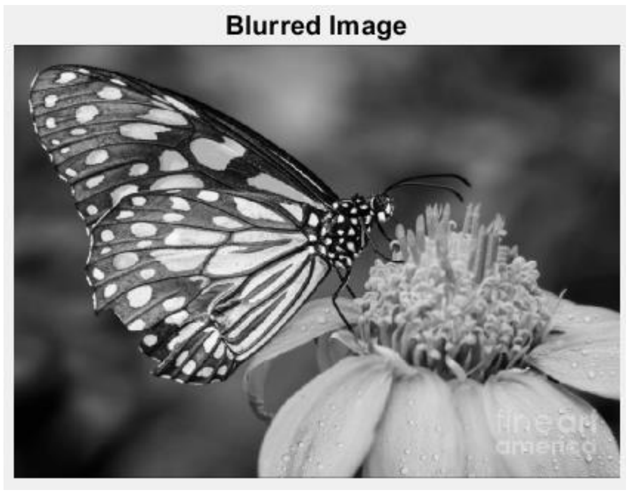










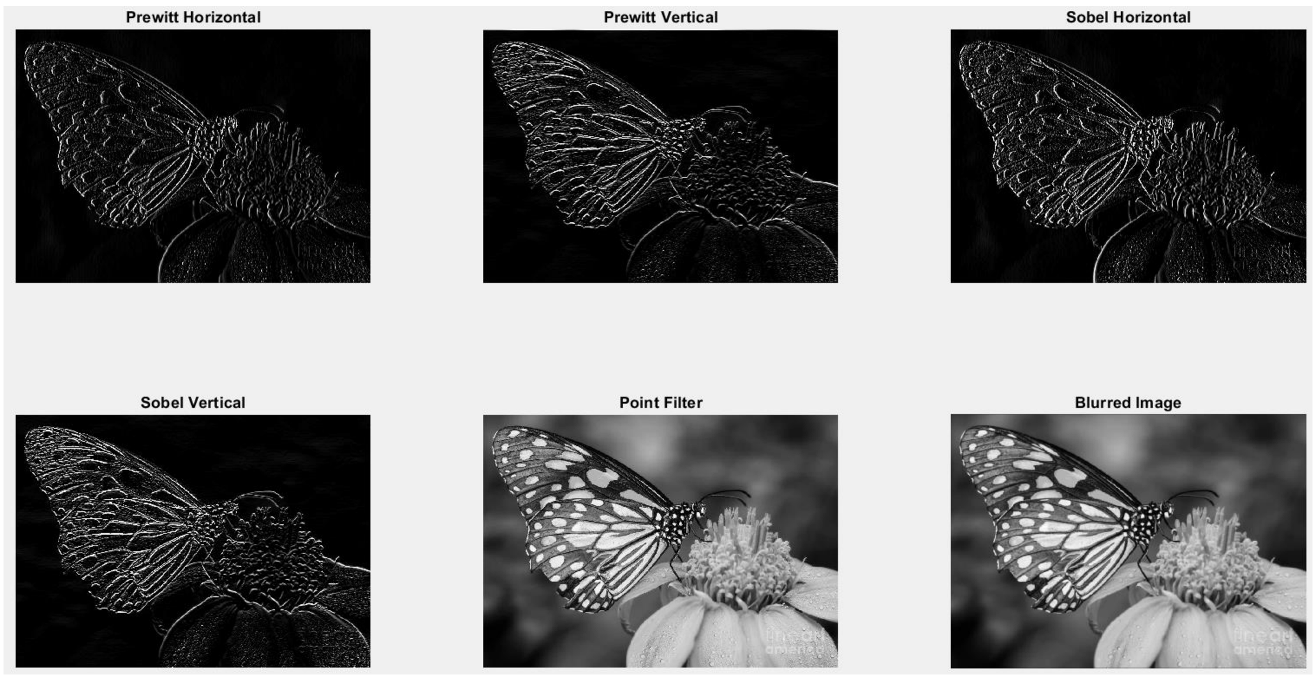


% Read an image of your choice

originalImage = imread('butterfly.jpg');

```
% Convert the image to grayscale
if size(originalImage, 3) == 3
   originalImage = rgb2gray(originalImage);
end

% Define Prewitt filter kernels for horizontal and vertical edges
prewittHorizontal = [-1, 0, 1; -1, 0, 1; -1, 0, 1];
prewittVertical = [-1, -1, -1; 0, 0, 0; 1, 1, 1];

% Define Sobel filter kernels for horizontal and vertical edges
sobelHorizontal = [-1, 0, 1; -2, 0, 2; -1, 0, 1];
sobelVertical = [-1, -2, -1; 0, 0, 0; 1, 2, 1];

% Define a point filter kernel (e.g., identity filter)
pointFilter = [0, 0, 0; 0, 1, 0; 0, 0, 0];

% Define a blurring filter kernel (e.g., 3x3 Gaussian blur)
blurFilter = fspecial('gaussian', [3, 3], 1);

% Apply filters using built-in function (imfilter)

% Prewitt filter
prewittHorizontalResult = imfilter(originalImage, prewittHorizontal);
prewittVerticalResult = imfilter(originalImage, prewittVertical);

% Sobel filter
sobelHorizontalResult = imfilter(originalImage, sobelHorizontal);
sobelVerticalResult = imfilter(originalImage, sobelVertical);

% Point filter (Identity)
pointFilterResult = imfilter(originalImage, pointFilter);
```

```
% Blurring filter

blurredImageResult = imfilter(originalImage, blurFilter);


% Display each result separately

figure;


subplot(2, 3, 1), imshow(prewittHorizontalResult), title('Prewitt Horizontal');

subplot(2, 3, 2), imshow(prewittVerticalResult), title('Prewitt Vertical');

subplot(2, 3, 3), imshow(sobelHorizontalResult), title('Sobel Horizontal');

subplot(2, 3, 4), imshow(sobelVerticalResult), title('Sobel Vertical');

subplot(2, 3, 5), imshow(pointFilterResult), title('Point Filter');

subplot(2, 3, 6), imshow(blurredImageResult), title('Blurred Image');
```



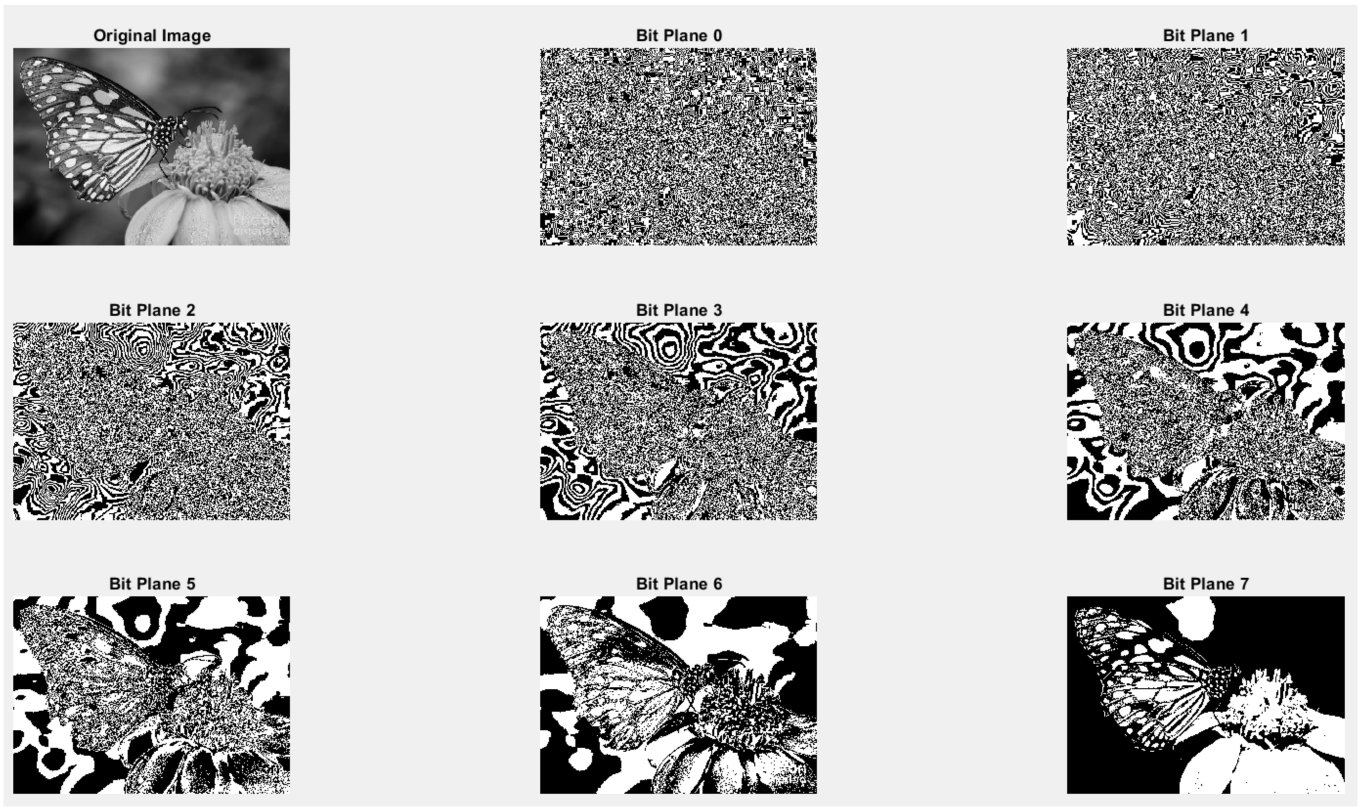
3)

```
% Load the input image
inputImage = imread('butterfly.jpg');



% Convert the image to grayscale
if size(inputImage, 3) == 3
   inputImage = rgb2gray(inputImage);
end

% Display the original image
figure;
subplot(3, 3, 1);
imshow(inputImage);
title('Original Image');

% Split the image into its bit planes
bitPlanes = cell(1, 8);
for i = 1:8
   bitPlanes{i} = bitget(inputImage, i);
   subplot(3, 3, i + 1);
   % Convert the logical bit plane to grayscale
   imshow(bitPlanes{i}, []);
   title(['Bit Plane ' num2str(i-1)]); % Displayed bit planes from 0 to 7
end

% Create a directory to save the bit planes as images
bitPlaneDir = 'bit_planes/';
if ~exist(bitPlaneDir, 'dir')
   mkdir(bitPlaneDir);
```

```
end

% Save the bit planes as individual images
for i = 1:8
   imwrite(bitPlanes{i}, [bitPlaneDir 'bit_plane_' num2str(i-1) '.jpg']);
end

% Assemble the original image by successively adding bit planes
assembledImage = zeros(size(inputImage), 'uint8');
assembledImages = cell(1, 7);

for i = 7:-1:1
   assembledImage = assembledImage + (bitPlanes{i} * 2^(i-1)); % Element-wise addition
   assembledImages{i} = assembledImage;
end

% Create a directory to save the assembled images
assembledDir = 'assembled_images/';
if ~exist(assembledDir, 'dir')
   mkdir(assembledDir);
end

% Save the assembled images as individual images
for i = 1:7
   imwrite(assembledImages{i}, [assembledDir 'assembled_image_' num2str(i-1) '.jpg']);
end

% Display and save the grayscale versions of assembled images
for i = 1:7
   figure;
   % Convert the logical assembled image to grayscale
```

```
    grayAssembledImage = mat2gray(assembledImages{i}); % Convert to grayscale
    imshow(grayAssembledImage);
    title(['Assembled Image ' num2str(i-1)]);

    % Save the grayscale image
    imwrite(grayAssembledImage, [assembledDir 'assembled_image_gray_' num2str(i-1) '.jpg']);

    % Detect noise in the bit plane
    noisyPixels = detectNoise(grayAssembledImage);
    fprintf('Noise in Assembled Image %d: %.2f%%\n', i-1, noisyPixels * 100);
end

% Function to detect noise in an image
function noisyPixels = detectNoise(image)
    [h, w] = size(image);
    noisyCount = 0;

    for x = 2:h
        for y = 2:w
            pixel = image(x, y);
            neighbors = [image(x-1, y), image(x, y-1), image(x-1, y-1)];
            similarity = sum(neighbors == pixel);

            % If the pixel is the same as at least two of the three adjacent pixels, it is not noise
            if similarity >= 2
                noisyCount = noisyCount + 1;
            end
        end
    end

    noisyPixels = noisyCount / (h * w);
```

end



The Bit Plane Resultant Image

Splitting the Image into Bit Planes:

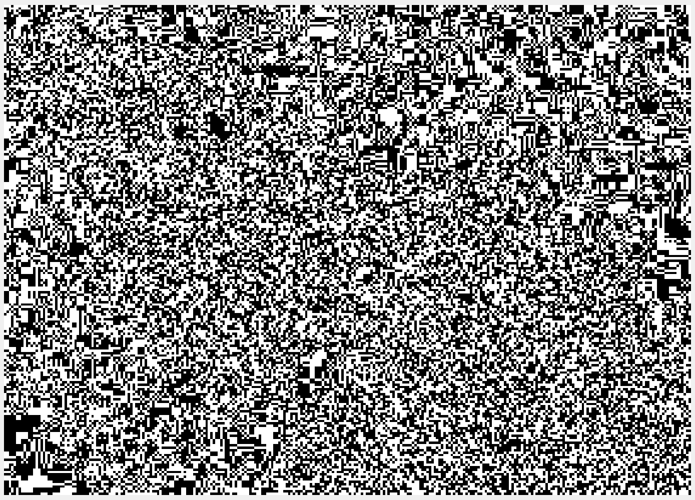
Bit planes are created by breaking down the original image into its binary components.

A cell array bit Planes is initialized to store the 8-bit planes.

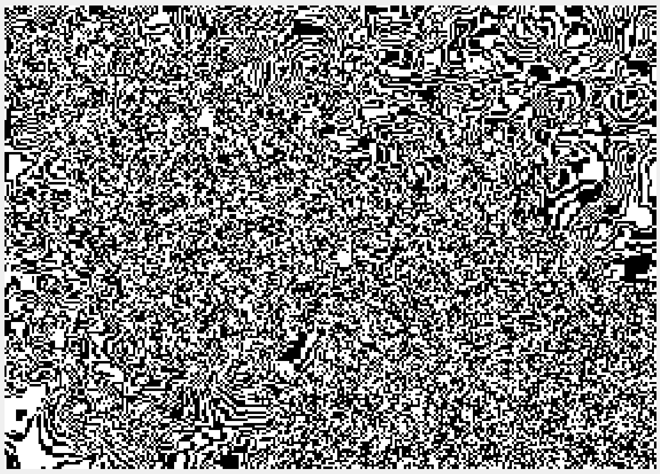
A loop from 1 to 8 iterates through each bit plane.

Each bit plane is displayed in a subplot, and titles are added to label them as "Bit Plane 0" to "Bit Plane 7

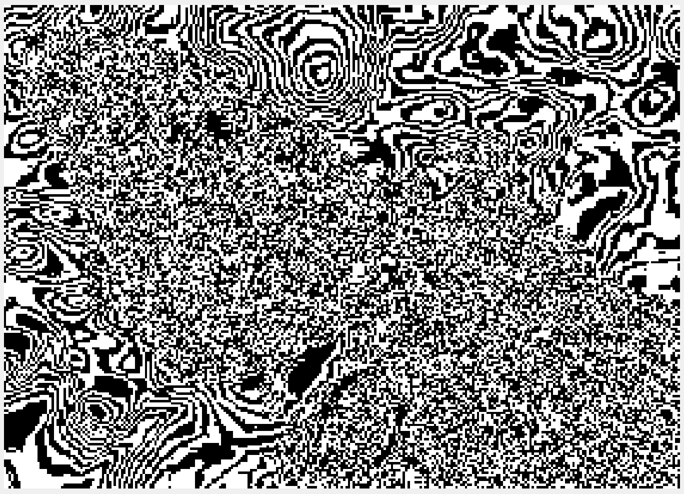
Bit Plane 0



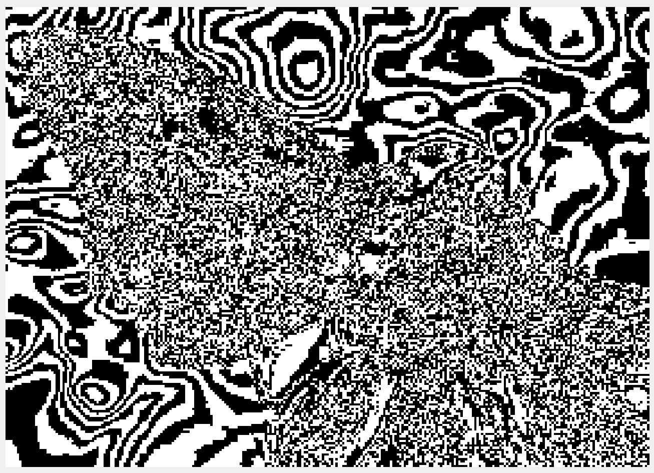
Bit Plane 1



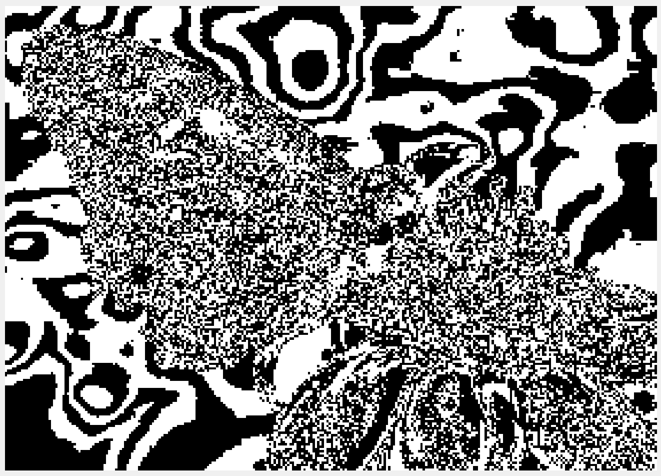
Bit Plane 2



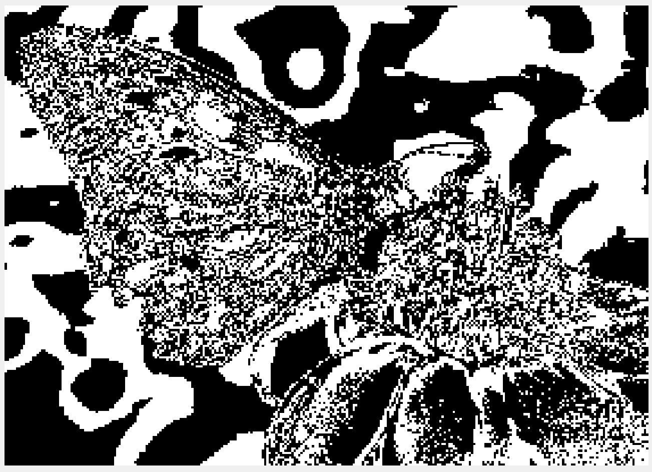
Bit Plane 3

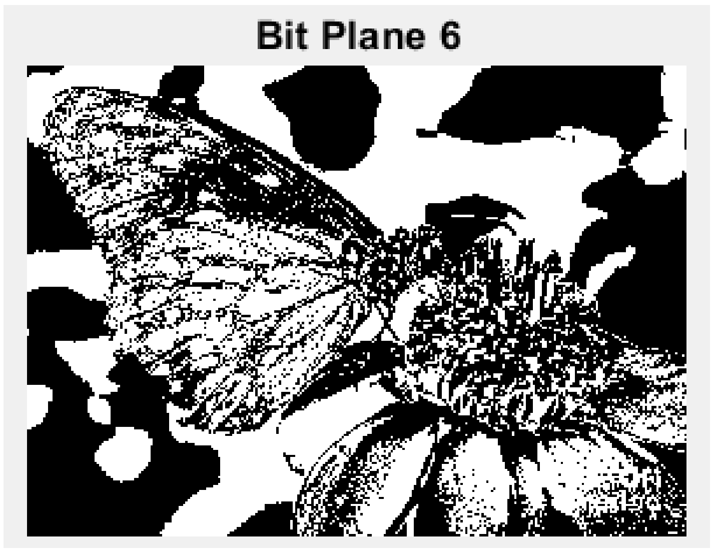


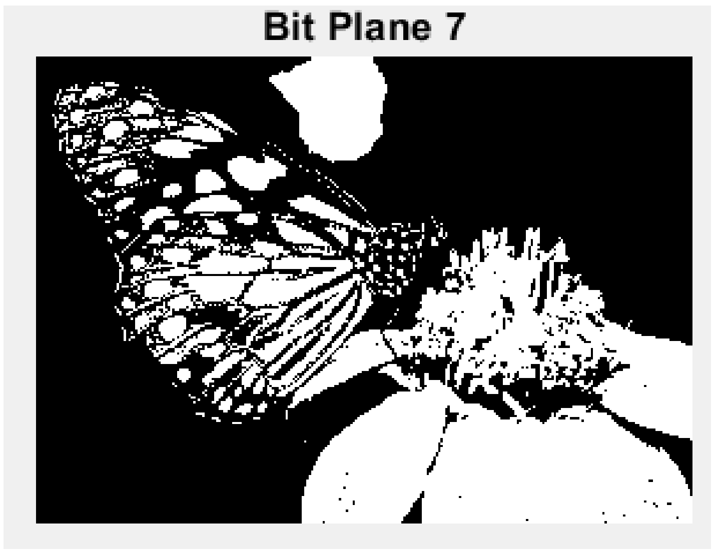
Bit Plane 4



Bit Plane 5







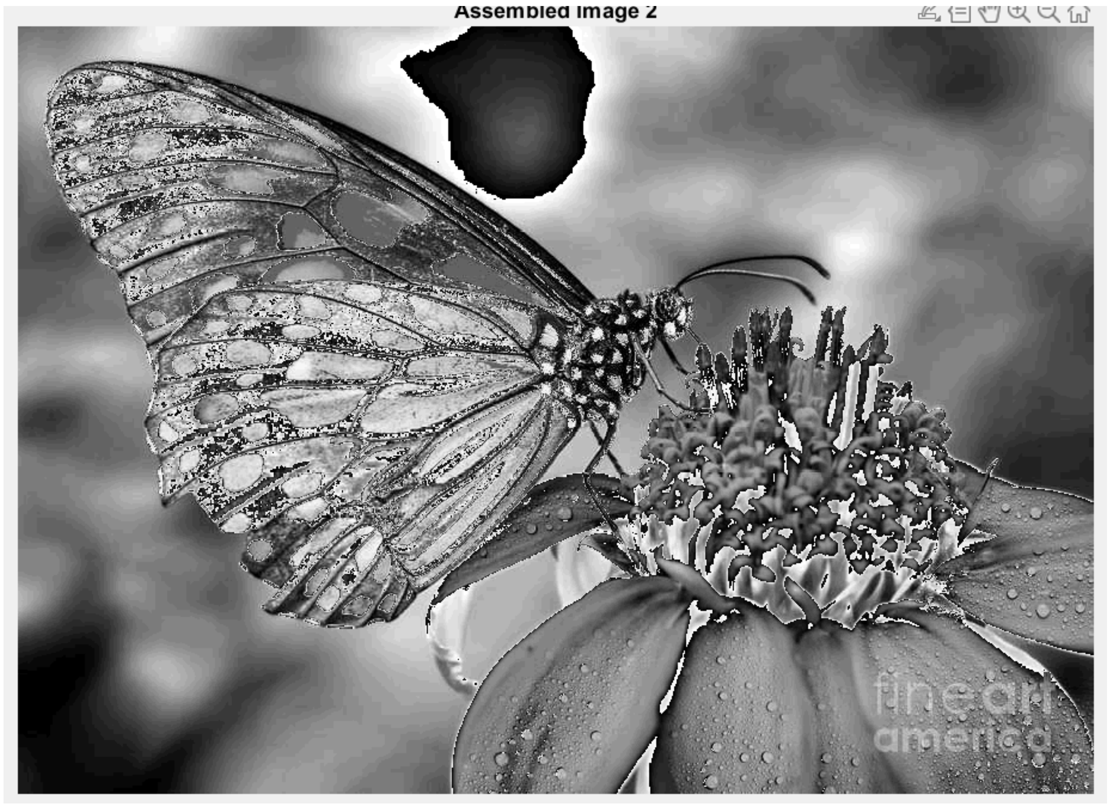
The code assembles the original image by successively adding the bit planes,

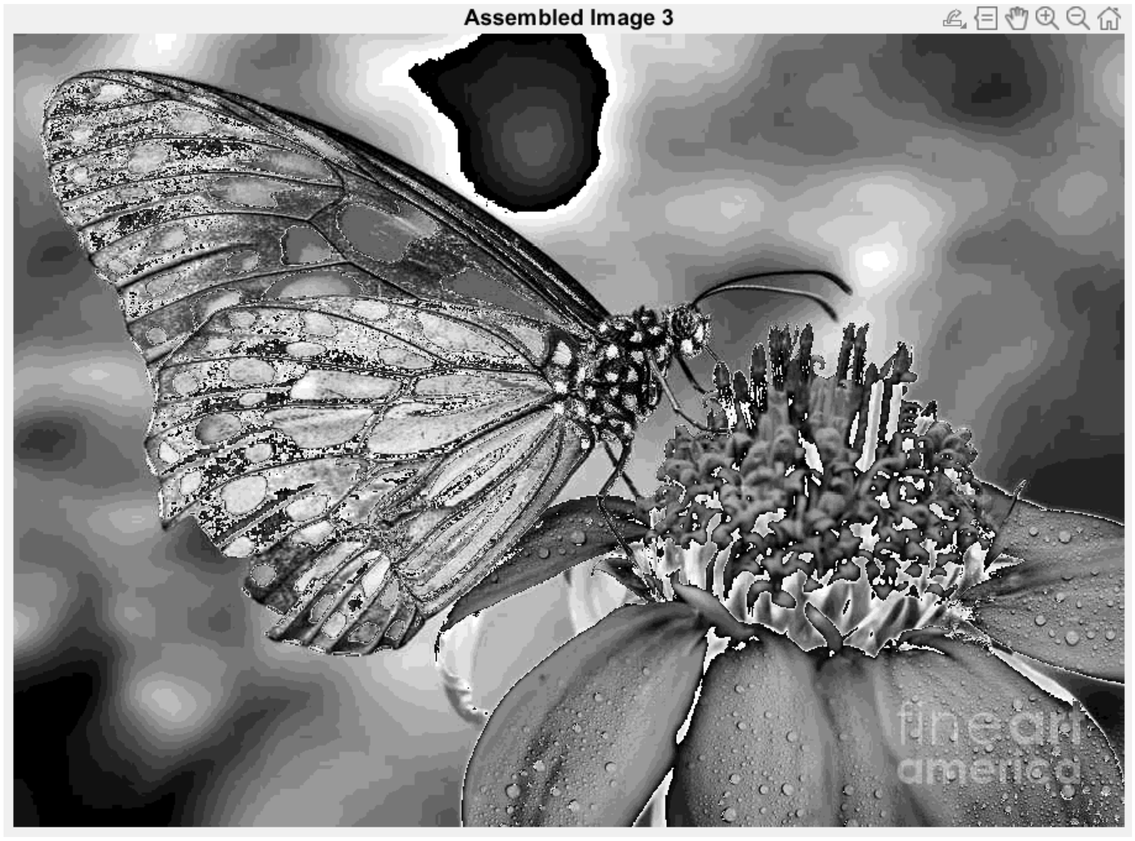
starting from the most significant bit plane (Bit Plane 7) and moving to the least significant bit plane (Bit Plane 1)

The assembled images are stored in a cell array assembledImages

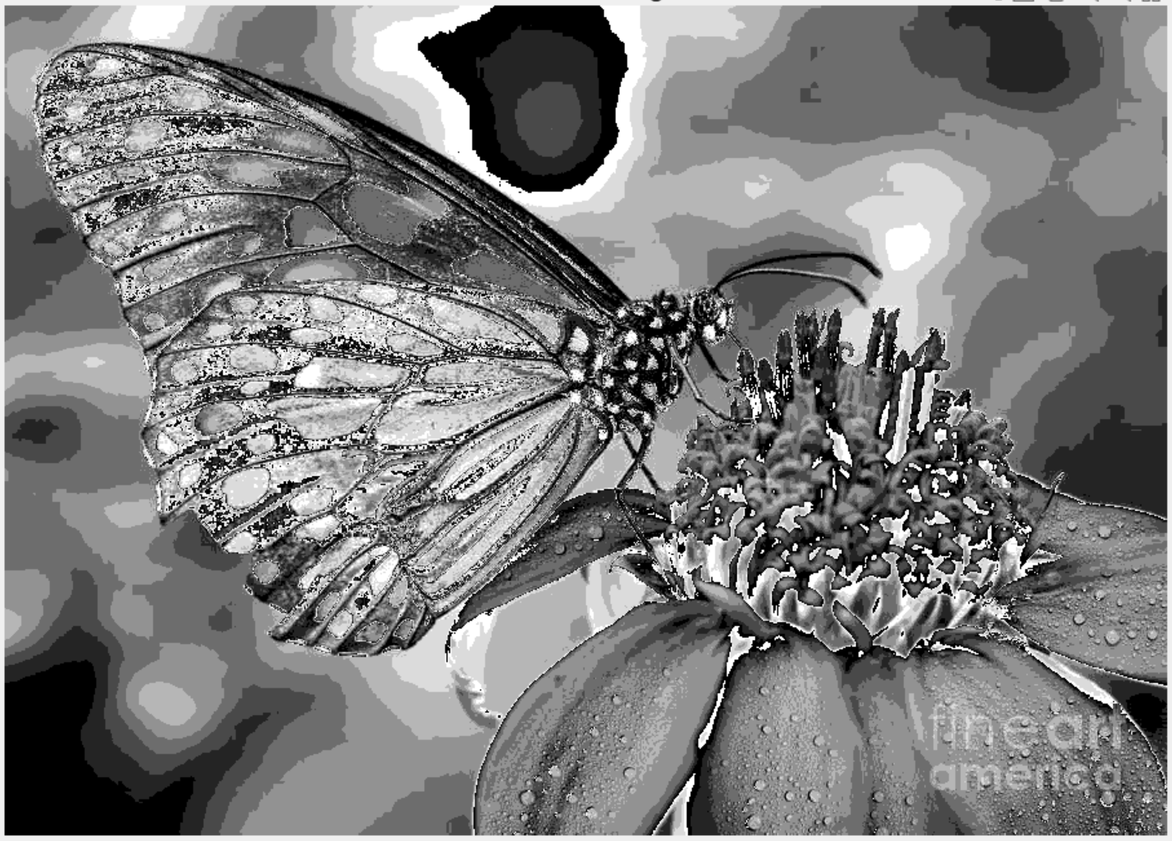
Assembled Image 0

Assembled Image 1

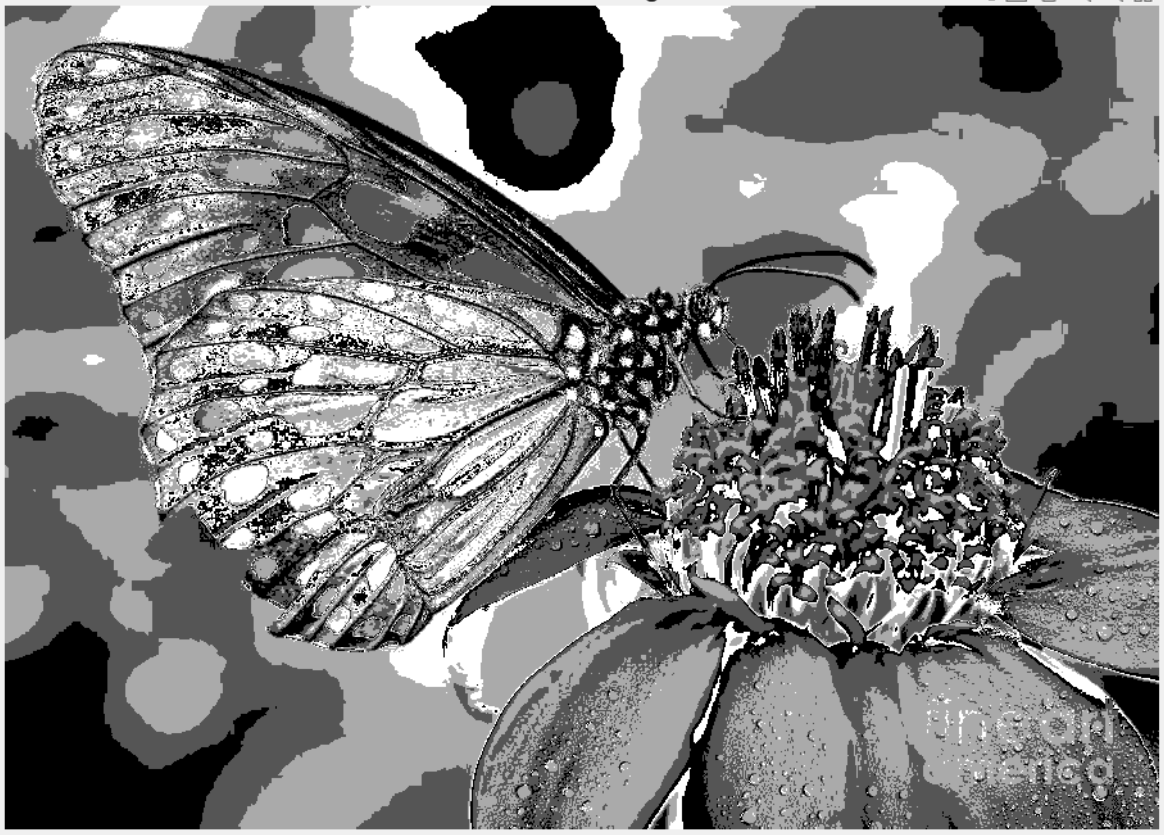

Assembled Image 2
fineart
america

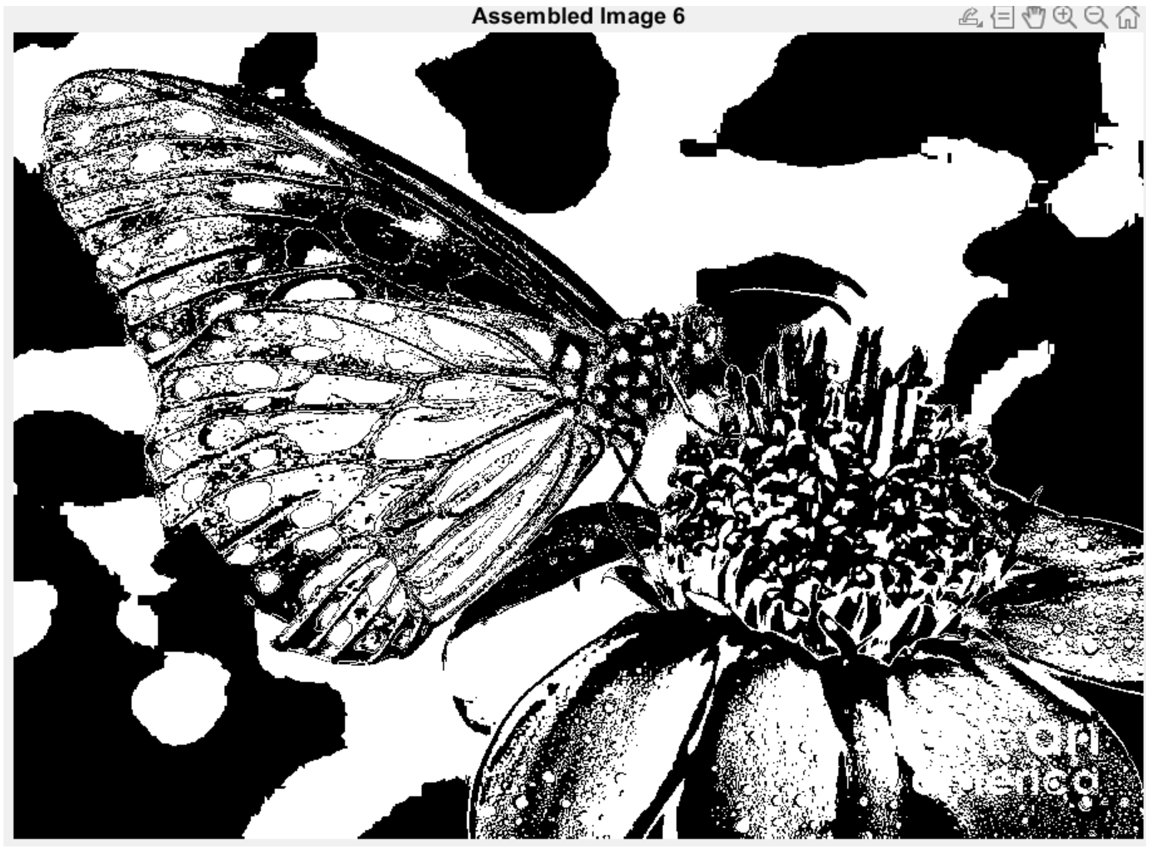

Assembled Image 3
fineart
america

Assembled Image 4



Assembled Image 5





For each grayscale assembled image, the detectNoise function is called to calculate the percentage of noisy pixels

```
Noise in Assembled Image 0: 22.02%
Noise in Assembled Image 1: 33.82%
Noise in Assembled Image 2: 44.55%
Noise in Assembled Image 3: 55.34%
Noise in Assembled Image 4: 67.40%
Noise in Assembled Image 5: 79.25%
Noise in Assembled Image 6: 88.72%
```