INDEX

SL. No.	Title	Page no.
1	Write a program to find neighbors of 4,8 and Diagonal point	1-2
2	Write a program to remove noise from an image using Median filter and Mean Filter.	2-3
3	Write a program to detect the edge of an image using Canny edge detection Algorithm.	
4	Write a program for the implementation of image sharpening filters and Edge Detection using Gradient Filters (i.e. Laplacian, Sobel, Prewitt, Roberts)	
5	Write a program to apply histogram equalization to enhance the contrast of an image.	
6	Write a program to apply Apply Gaussian smoothing to blur an image. Apply Gaussian smoothing with a specified standard deviation (sigma)	
7	Write a program to add random noise to an image and display the noisy image.	
8	Write a program to crop a region of interest (ROI) from an image.	
9	Write a program to detect and remove salt-and-pepper noise from an image.	
11	Write a program to apply Laplacian of Gaussian (LoG) edge detection to an image	

GROUP-B

1. 4 neighbors , 8 neighbors and Diagonal neighbors.

INPUT:

FIND_NEIGHBOURS.M

```
function find_neighbors(matrix, row, col)
    % Get the size of the matrix
    [rows, cols] = size(matrix);
        % Define the directions for 4-neighbors, 8-neighbors, and diagonal neighbors
    directions_4 = [0, 1; 1, 0; 0, -1; -1, 0];
    directions_8 = [0, 1; 1, 1; 1, 0; 1, -1; 0, -1; -1, -1; -1, 0; -1, 1];
    directions_diagonal = [-1, -1; -1, 1; 1, -1; 1, 1];
        % Function to find neighbors based on given directions
    function neighbors = find_based_on_directions(directions)
        neighbors = [];
        for i = 1:size(directions, 1)
            new_row = row + directions(i, 1);
            new_col = col + directions(i, 2);
            % Check if the new row and column indices are within bounds
            if new_row >= 1 && new_row <= rows && new_col >= 1 && new_col <= cols
                % Append the neighbor value
                neighbors = [neighbors, matrix(new_row, new_col)];
            end
        end
    end

    % Get the 4-neighbors, 8-neighbors, and diagonal neighbors
    four_neighbors = find_based_on_directions(directions_4);
    eight_neighbors = find_based_on_directions(directions_8);
    diagonal_neighbors = find_based_on_directions(directions_diagonal);

    % Display the results
    fprintf('4-neighbors of point (%d, %d):\n', row, col);
    disp(four_neighbors);
    fprintf('8-neighbors of point (%d, %d):\n', row, col);
    disp(eight_neighbors);
    fprintf('Diagonal neighbors of point (%d, %d):\n', row, col);
    disp(diagonal_neighbors);
end
```

MAIN_SCRIPT.M

```
% Get matrix input from the user
rows = input('Number of rows: ');
cols = input('Number of columns: ');
fprintf('Enter the matrix values row by row, with values separated by spaces:\n');
matrix = zeros(rows, cols);
```

```
for i = 1:rows
    row_values = input(sprintf('Enter values for row %d: ', i), 's');
    matrix(i, :) = str2num(row_values); %#ok<ST2NM>
end
% Get the row and column of the point of interest
row = input('Enter the row index of the point of interest: ');
col = input('Enter the column index of the point of interest: ');
% Find the neighbors of the specified point
find_neighbors(matrix, row, col);
```

OUTPUT:

```
>> main_script
Number of rows: 3
Number of columns: 3
Enter the matrix values row by row, with values separated by spaces:
Enter values for row 1: 1 2 3
Enter values for row 2: 4 5 6
Enter values for row 3: 7 8 9
Enter the row index of the point of interest: 2
Enter the column index of the point of interest: 2
4-neighbors of point (2, 2):
     6     8     4     2

8-neighbors of point (2, 2):
     6     9     8     7     4     1     2     3

Diagonal neighbors of point (2, 2):
     1     3     7     9
```

EXPLANATION:

This MATLAB code consists of two scripts: `FIND_NEIGHBOURS.M` and `MAIN_SCRIPT.M`. `FIND_NEIGHBOURS.M` defines a function to find and display the 4-neighbors, 8-neighbors, and diagonal neighbors of a specified point in a given matrix. It uses predefined directions to check adjacent cells and ensures they are within matrix bounds before collecting their values. `MAIN_SCRIPT.M` prompts the user to input the size and values of a matrix row by row, then asks for the coordinates of a point of interest. Finally, it calls the `find_neighbors` function to find and print the neighbors of the specified point using the defined directions.

2. Remove noise from an image using Median filter and Mean Filter.

INPUT:

```
% Read the input image
input_image = imread('G:/4TH YEAR/Image_Processing/Pictures/puppy.jpeg'); % Replace 'noisy_image.jpg' with your image file

% Convert the image to grayscale if it's a color image
if size(input_image, 3) == 3
    gray_image = rgb2gray(input_image);
else
    gray_image = input_image;
end

% Display the original image
figure;
```

```
subplot(1, 3, 1);
imshow(gray_image);
title('Original Image');
% Apply Median Filter
median_filtered_image = medfilt2(gray_image);
% Display the Median Filtered image
subplot(1, 3, 2);
imshow(median_filtered_image);
title('Median Filtered Image');
% Apply Mean Filter (using a 3x3 averaging filter)
mean_filter = fspecial('average', [3 3]);
mean_filtered_image = imfilter(gray_image, mean_filter);
% Display the Mean Filtered image
subplot(1, 3, 3);
imshow(mean_filtered_image);
title('Mean Filtered Image');
% Save the filtered images if needed
imwrite(median_filtered_image, 'median_filtered_image.jpg');
imwrite(mean_filtered_image, 'mean_filtered_image.jpg');
```

OUTPUT:

EXPLANATION:

The provided code snippet operates on an input image, first verifying if it's in color and converting it to grayscale if necessary. It proceeds to display the original image alongside its median and mean filtered variants. The median filter is applied using 'medfilt2', while the mean filter is implemented using a 3x3 averaging filter created with 'fspecial' and applied via 'imfilter'. The resulting filtered images are then saved as 'median_filtered_image.jpg' and 'mean_filtered_image.jpg'.

3. Detecting the edge of an image using Canny edge detection Algorithm.

INPUT:

```
% Read the image
originalImage = imread('D:\matlab installation\programs\images\new1.jpg');
% Convert the image to grayscale if it is not already
if size(originalImage, 3) == 3
grayImage = rgb2gray(originalImage);
```

```
else
grayImage = originalImage;
end
% Display the original grayscale image
figure;
subplot(1, 2, 1);
imshow(grayImage);
title('Original Grayscale Image');
% Apply Canny edge detection
edgeDetectedImage = edge(grayImage, 'Canny');
% Display the edge-detected image
subplot(1, 2, 2);
imshow(edgeDetectedImage);
title('Canny Edge Detected Image');
```

OUTPUT:

EXPLANATION:

The code snippet begins by reading an image file and converting it to grayscale if it's not already in that format. It proceeds to display the original grayscale image alongside its Canny edge-detected version. The edge detection is performed using the 'edge' function with the Canny method.

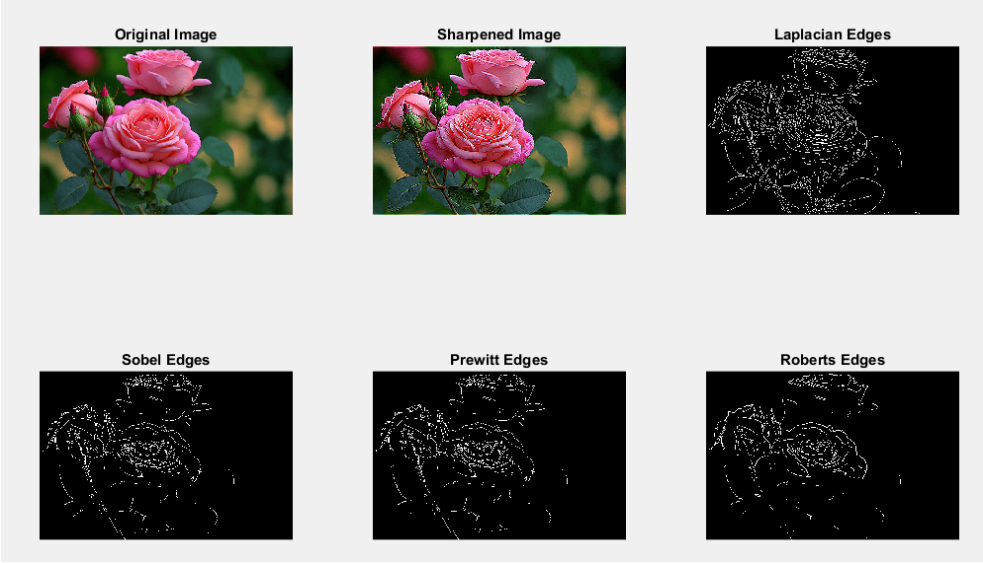
4. Image sharpening filters and Edge Detection using Gradient Filters (i.e. Laplacian, Sobel, Prewitt, Roberts)

INPUT:

```
function main()
    image_path = 'G:/4TH YEAR/Image_Processing/Pictures/flowers.jpg';
    img = imread(image_path);
    sharpened_img = image_sharpening(img);
    [laplacian_edges, sobel_edges, prewitt_edges, roberts_edges] = edge_detection(img);
    figure('Position', [100, 100, 1200, 800]);
    subplot(2, 3, 1);
    imshow(img);
    title('Original Image');
    axis off;
    subplot(2, 3, 2);
```

```
    imshow(sharpened_img);
    title('Sharpened Image');
    axis off;
    subplot(2, 3, 3);
    imshow(laplacian_edges);
    title('Laplacian Edges');
    axis off;
    subplot(2, 3, 4);
    imshow(sobel_edges);
    title('Sobel Edges');
    axis off;
    subplot(2, 3, 5);
    imshow(prewitt_edges);
    title('Prewitt Edges');
    axis off;
    subplot(2, 3, 6);
    imshow(roberts_edges);
    title('Roberts Edges');
    axis off;
end
function sharpened_img = image_sharpening(img)
    sharpening_kernel = [0, -1, 0; -1, 5, -1; 0, -1, 0];
    sharpened_img = imfilter(img, sharpening_kernel);
end
function [laplacian_edges, sobel_edges, prewitt_edges, roberts_edges] = edge_detection(img)
    gray_img = rgb2gray(img);
    laplacian_edges = edge(gray_img, 'log');
    sobel_edges = edge(gray_img, 'sobel');
    prewitt_edges = edge(gray_img, 'prewitt');
    roberts_edges = edge(gray_img, 'roberts');
end
```

OUTPUT:



EXPLANATION:

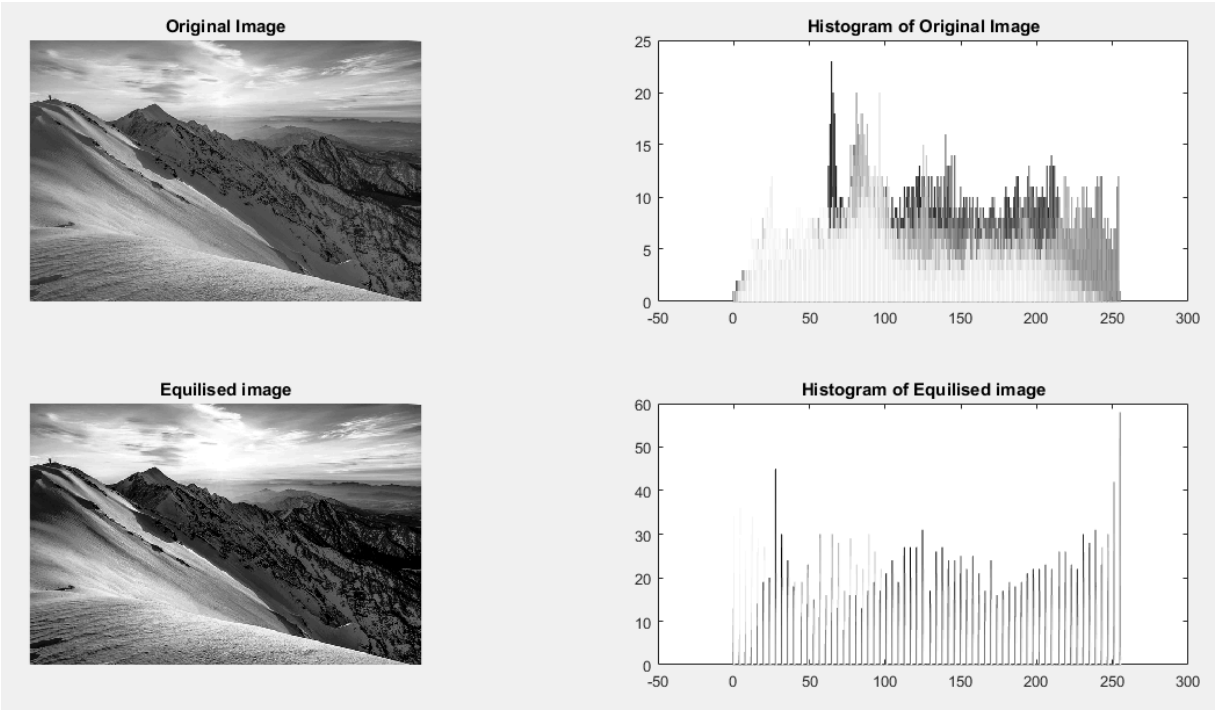
This MATLAB code performs image sharpening and edge detection on an input image of flowers. It first loads the image and applies image sharpening using a predefined kernel to enhance the details in the image. Then, it performs edge detection using Laplacian, Sobel, Prewitt, and Roberts operators to detect edges of different orientations in the grayscale version of the image. Finally, it displays the original image, sharpened image, and the detected edges using each operator in separate subplots of a single figure for visual comparison. The image processing operations are implemented using MATLAB's built-in functions like imread, imfilter, edge, and imshow. The subplot function is used to arrange multiple images in a single figure window.

5. Apply histogram equalization to enhance the contrast of an image.

INPUT:

```
img = imread('G:/4TH YEAR/Image_Processing/Pictures/mountain.jpg');

if(size(img,3)==3)
   img_gray=rgb2gray(img);
else
   img_gray=img;
end
%Compute the histogram of image
[histogram_val1,intensity_val1] = imhist(img_gray);
img_eq=histeq(img_gray);
[histogram_val2,intensity_val2]=imhist(img_eq);
subplot(2,2,1);imshow(img_gray);title('Original Image');
subplot(2,2,2); hist(img_gray,[0:255]);title('Histogram of Original Image');
subplot(2,2,3); imshow(img_eq);title('Equilised image');
subplot(2,2,4); hist(img_eq,[0:255]);
title('Histogram of Equilised image');
```

OUTPUT:



EXPLANATION:

This MATLAB code reads an image of a mountain from the specified path and processes it to enhance its contrast using histogram equalization. Initially, it checks if the image is in color (i.e., has three channels). If so, it converts the image to grayscale using rgb2gray; otherwise, it uses the image as is. It then computes the histogram of the grayscale image to understand the distribution of pixel intensities using imhist. Next, it applies histogram equalization with histeq to improve the contrast of the image by redistributing the intensity values more uniformly. The histograms before and after equalization are computed using imhist again. Finally, the code displays the original grayscale image and its histogram, along with the equalized image and its histogram, in a 2x2 grid of subplots for visual comparison. This allows us to observe how the histogram equalization process enhances the image's contrast by spreading out the most frequent intensity values.

6. Apply Gaussian smoothing to blur an image. Apply Gaussian smoothing with a specified standard deviation (sigma).

INPUT:

```
% Read the image
img = imread('Circles.jpg');
% Define the standard deviation (sigma) for Gaussian filter
sigma = 10;  % Increased sigma for more noticeable blur
% Create a Gaussian filter using fspecial
h = fspecial('gaussian', [5 5], sigma);
% Apply Gaussian smoothing using imfilter
blurred_img = imfilter(img, h);
% Option 1: Set the figure title after creating the figure (with zoomed-in comparison)
figure;
set(gcf, 'Name', 'Blur Comparison');  % Set figure title
% Create a 1x3 subplot layout to accommodate all images
subplot(121); imshow(img); title('Original Image');
subplot(122); imshow(blurred_img); title(sprintf('Blurred Image (sigma = %f)', sigma));
```

OUTPUT:

EXPLANATION:

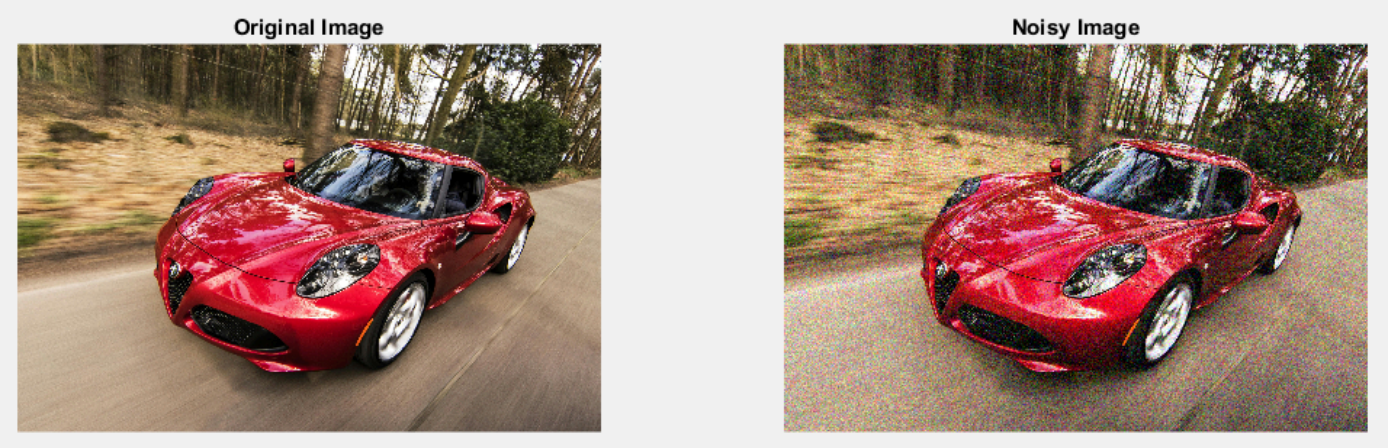
This MATLAB code applies Gaussian smoothing to an image 'Circles.jpg' to create a blurred version of the original image. Firstly, the image is read using the imread function, and a standard deviation (sigma) of 10 is defined for the Gaussian filter, which determines the amount of blur applied. The Gaussian filter is created using the fspecial function with a 5x5 kernel size and the specified sigma value. Next, the imfilter function is used to convolve the Gaussian filter with the original image, resulting in a blurred version of the image. Finally, the code creates a figure titled "Blur Comparison" and displays both the original and blurred images side by side in a 1x2 subplot layout, allowing for a visual comparison of the effect of Gaussian smoothing on the image.

7. Add random noise to an image and display the noisy image.

INPUT:

```
function main()
    % Define the image path
    image_path = 'G:/4TH YEAR/Image_Processing/Pictures/mountain.jpg';

    % Load the image
    img = imread(image_path);

    % Check if the image was loaded successfully
    if isempty(img)
        disp('Error loading image.');
        return;
    end

    % Convert the image to double precision for adding noise
    img_double = im2double(img);

    % Add Gaussian noise to the image
    noisy_img = imnoise(img_double, 'gaussian', 0, 0.01); % Mean = 0, Variance = 0.01

    % Display the original image and the noisy image
    figure('Position', [100, 100, 1200, 600]);
    % Display the original image
    subplot(1, 2, 1);
    imshow(img);
    title('Original Image');
    axis off;

    % Display the noisy image
    subplot(1, 2, 2);
    imshow(noisy_img);
    title('Noisy Image');
    axis off;
end
```

OUTPUT:



EXPLANATION:

This MATLAB program reads an image from a specified file path, adds Gaussian noise to the image, and displays both the original and noisy images for comparison. The program starts by defining the image path and loading the image using imread. It then checks if the image was loaded successfully. To add noise, the image is first converted to double precision using im2double, which is necessary for accurate noise addition. Gaussian noise with a mean of 0 and variance of 0.01 is added to the image using the imnoise function. Finally, the original image and the noisy image are displayed side by side in a figure window with two subplots, allowing for a clear visual comparison between the two images. The imshow function is used to display the images, and the title function is used to label each subplot accordingly.

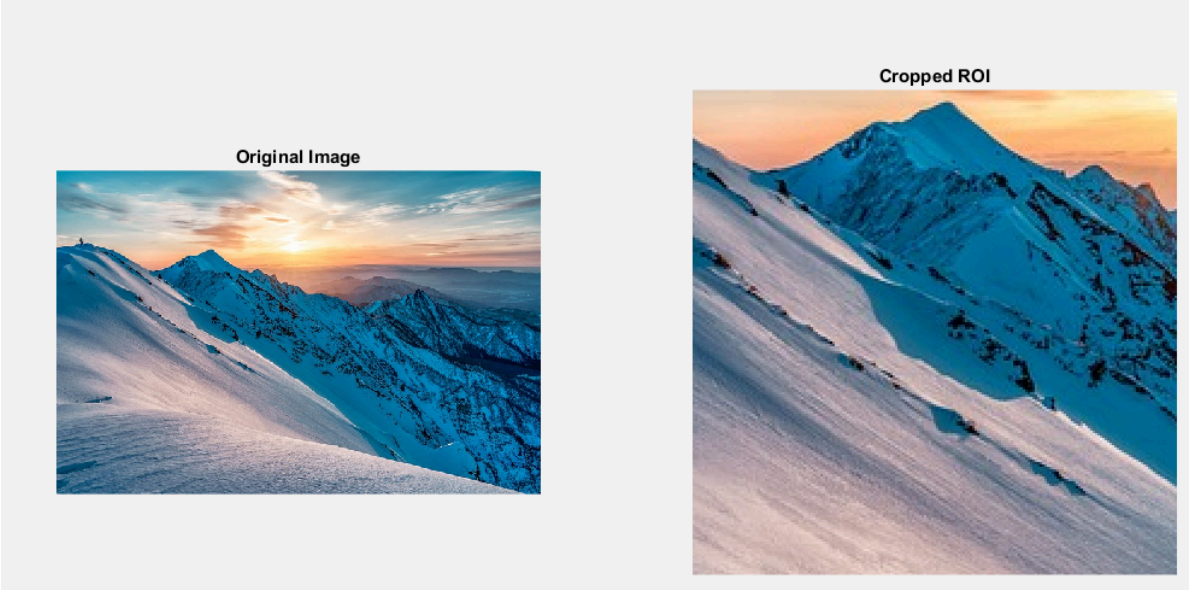
8. Crop a region of interest (ROI) from an image.

INPUT:

```
function main()
    % Define the image path
    image_path = 'G:/4TH YEAR/Image_Processing/Pictures/mountain.jpg';
    % Define the coordinates of the top-left and bottom-right corners of the ROI
    top_left = [100, 100];
    bottom_right = [300, 300];

    % Crop the region of interest from the image
    crop_roi(image_path, top_left, bottom_right);
end

function crop_roi(image_path, top_left, bottom_right)
    % Load the image
    img = imread(image_path);
    % Check if the image was loaded successfully
    if isempty(img)
        disp('Error loading image.');
        return;
    end

    % Extract the region of interest (ROI)
    roi = img(top_left(2):bottom_right(2), top_left(1):bottom_right(1), :);
```

```
    % Display the original image and the cropped ROI
    figure('Position', [100, 100, 1200, 600]);
    % Display the original image
    subplot(1, 2, 1);
    imshow(img);
    title('Original Image');
    axis off;

    % Display the cropped ROI
    subplot(1, 2, 2);
    imshow(roi);
    title('Cropped ROI');
    axis off;
end
```

OUTPUT:



EXPLANATION:

This MATLAB code is designed to load an image, crop a specified region of interest (ROI) from it, and display both the original and cropped images. The main function sets the path to the image file and defines the coordinates for the top-left and bottom-right corners of the ROI. It then calls the crop_roi function, passing the image path and ROI coordinates as arguments. Inside crop_roi, the image is read using imread, and a check is performed to ensure the image is loaded correctly. The specified ROI is extracted from the image using matrix indexing. Both the original image and the cropped ROI are displayed side by side in a figure window with two subplots, labeled "Original Image" and "Cropped ROI" respectively, using imshow and title. This setup allows for a visual comparison between the original image and the specific section that was cropped.

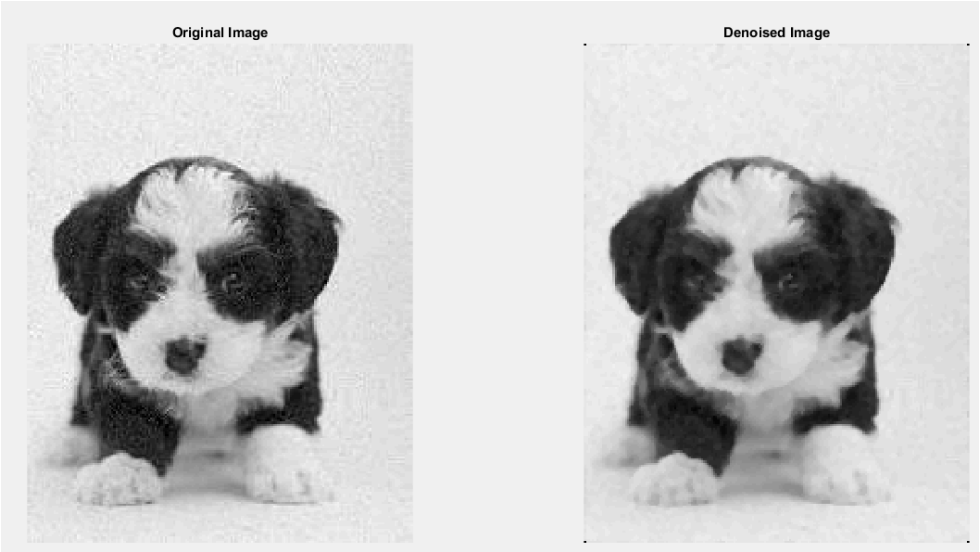
9. Detects and removes salt-and-pepper noise from an image.

INPUT:

```
function main()
    % Define the path to the image
    image_path = 'G:/4TH YEAR/Image_Processing/Pictures/puppy.jpeg';
```

```
    % Define the filter size for the median filter
    filter_size = 3;  % You can adjust the filter size as needed

    % Call the function to remove salt-and-pepper noise and display the results
    remove_salt_and_pepper_noise(image_path, filter_size);
end

function remove_salt_and_pepper_noise(image_path, filter_size)
    % Read the image from the provided path
    image = imread(image_path);

    % If the image is colored, apply the filter to each channel separately
    if size(image, 3) == 3
        denoised_image = zeros(size(image), 'uint8');
        for i = 1:3
            denoised_image(:,:,i) = medfilt2(image(:,:,i), [filter_size, filter_size]);
        end
    else
        % Apply a median filter to remove salt-and-pepper noise for grayscale images
        denoised_image = medfilt2(image, [filter_size, filter_size]);
    end

    % Display the original and denoised images side by side
    figure('Position', [100, 100, 1200, 600]);
    subplot(1, 2, 1);
    imshow(image);
    title('Original Image');
    axis off;

    subplot(1, 2, 2);
    imshow(denoised_image);
    title('Denoised Image');
    axis off;
end
```

OUTPUT:



EXPLANATION:

This MATLAB program reads an image from a specified file path, applies a median filter to remove salt-and-pepper noise, and displays the original and denoised images side by side for comparison. The `main` function specifies the image path and the filter size, and calls the `remove_salt_and_pepper_noise` function. Inside this function, the image is loaded using `imread`. If the image is colored (has three channels), the median filter `medfilt2` is applied to each channel individually; otherwise, it is applied directly to the grayscale image. The `medfilt2` function is used with the specified filter size to perform the noise reduction. Finally, the original and denoised images are displayed in a figure window with two subplots using `imshow` and labeled appropriately using `title`, with the axes turned off for a cleaner presentation.

10. Detect circles in an image using the Hough transform, Calculate and display the perimeter and radius of each detected circle.

INPUT:

```
% Read the image
image = imread('CirclesNotes.jpg'); % Replace 'Circles.jpg' with your image file name

% Check if the image is already in grayscale
if size(image, 3) == 3
  grayImage = rgb2gray(image); % Convert to grayscale if the image is in RGB
else
  grayImage = image; % Image is already in grayscale
end
% Apply noise reduction (optional, adjust sigma as needed)
sigma = 1;  % Adjust sigma for your image noise level
grayImage = imgaussfilt(grayImage, sigma);
% Apply contrast enhancement (optional, adjust clipLimit for best results)
grayImage = adapthisteq(grayImage, 'clipLimit', 0.02);
% Apply edge detection with Canny filter (adjust thresholds if needed)
lowThreshold = 0.001;  % Experiment with these values
highThreshold = 0.1;
edges = edge(grayImage, 'canny', [lowThreshold highThreshold], 2);
```

```
% Detect circles using Hough Transform with refined parameters
minRadius = 20;  % Adjust based on expected minimum circle size
maxRadius = 80;  % Adjust based on expected maximum circle size
sensitivity = 0.9;  % Adjust sensitivity for best balance

[centers, radii, metric] = imfindcircles(edges, [minRadius maxRadius], ...
                                         'Sensitivity', sensitivity);
% Display the original image
imshow(image);
hold on;
% Plot detected circles with high metric values
if ~isempty(centers)
  % Filter circles by high metric (more accurate)
  goodCircles = metric >= mean(metric);
  viscircles(centers(goodCircles, :), radii(goodCircles), 'EdgeColor', 'b');
  % Calculate and display properties for each detected circle
  numCircles = sum(goodCircles);
  for i = 1:numCircles
    radius = radii(i);
    perimeter = 2 * pi * radius;
    fprintf('Circle %d: Radius = %.2f, Perimeter = %.2f\n', i, radius, perimeter);
    % Annotate the image with radius and perimeter
    text(centers(i, 1), centers(i, 2), sprintf('R=%.2f, P=%.2f', radius, perimeter), ...
        'Color', 'yellow', 'FontSize', 10, 'FontWeight', 'bold');
  end
else
  disp('No circles detected.');
end

hold off;
```

OUTPUT:

EXPLANATION:

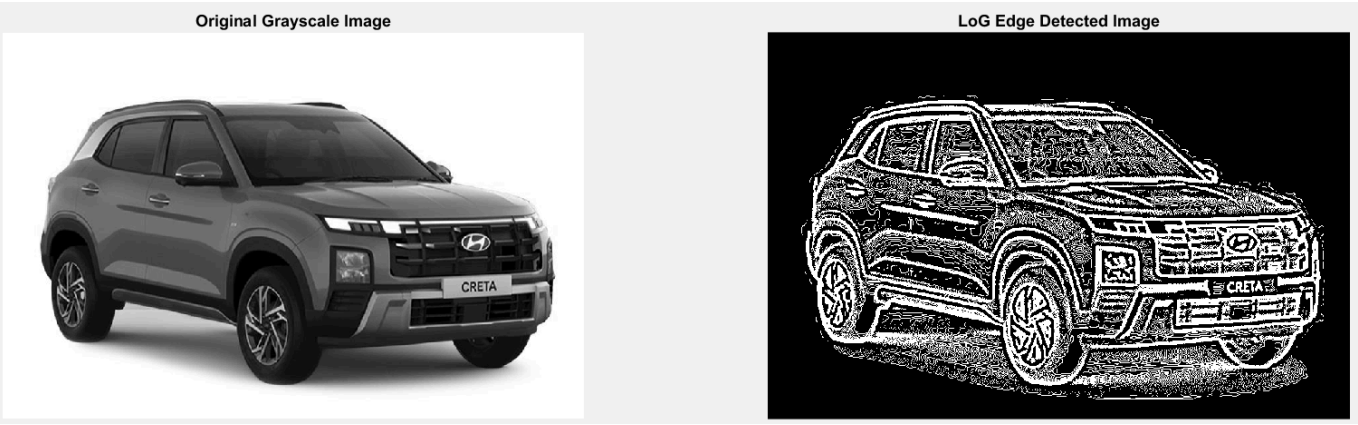
This program reads an image, converts it to grayscale if needed, and applies noise reduction and contrast enhancement (optional). Then, it uses an edge detection filter to highlight the outlines of potential circles. The Hough transform, a powerful circle detection technique, is used to find circles within the image based on specified size ranges and sensitivity. Finally, the program displays the original image with detected circles highlighted in blue. It also calculates and displays the radius and perimeter of each well-detected circle (based on a metric threshold) and annotates the image with this information.

11. Apply Laplacian of Gaussian (LoG) edge detection to an image.

INPUT:

```
% Read the image
originalImage = imread('D:\matlab installation\programs\images\car.jpg');
% Convert the image to grayscale if it is not already
if size(originalImage, 3) == 3
        grayImage = rgb2gray(originalImage);
else
        grayImage = originalImage;
end
% Display the original grayscale image
figure;
subplot(1, 2, 1);
imshow(grayImage);
title('Original Grayscale Image');
% Apply Gaussian filter to smooth the image
smoothedImage = imgaussfilt(grayImage, 2); % Adjust the sigma as needed
% Apply Laplacian filter
laplacianFilter = fspecial('log', [5, 5], 0.5); % Adjust the size and sigma as needed
logImage = imfilter(smoothedImage, laplacianFilter, 'replicate');
% Threshold the image to get binary edges
threshold = 0.05; % Adjust the threshold as needed
edgeDetectedImage = logImage > threshold;
% Display the edge-detected image
subplot(1, 2, 2);
imshow(edgeDetectedImage);
title('LoG Edge Detected Image');
```

OUTPUT:



EXPLANATION:

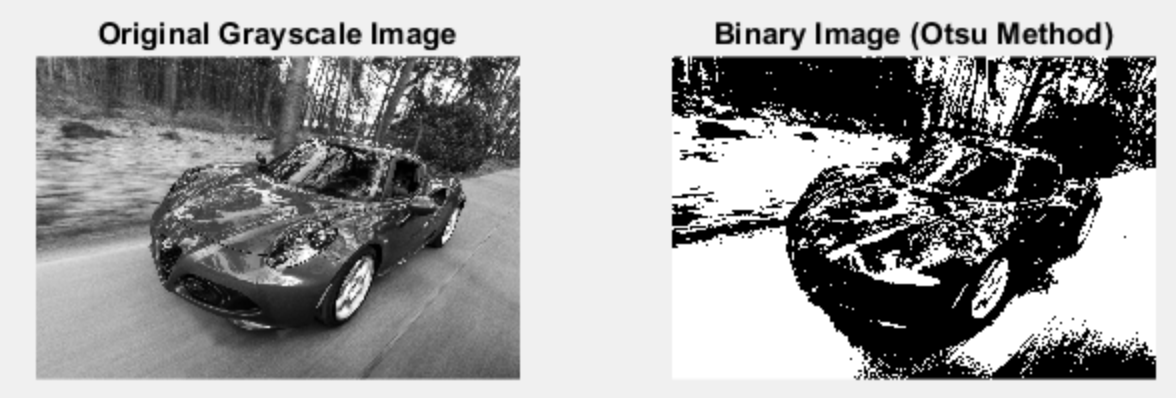
The provided code reads an image, converts it to grayscale if necessary, and displays the original grayscale image. It then applies a Gaussian filter to smooth the image, followed by a Laplacian of Gaussian (LoG) filter to detect edges. The result is thresholded to create a binary edge-detected image, which is then displayed alongside the original image.

12. Apply Global Thresholding on an image using Otsu Method.

INPUT:

```
% Read the input image
originalImage = imread('G:/4TH YEAR/Image_Processing/Pictures/car.jpg'); % Replace 'input_image.jpg' with your image file
% Convert the image to grayscale if it is not already
if size(originalImage, 3) == 3
    grayImage = rgb2gray(originalImage);
else
    grayImage = originalImage;
end
% Apply Otsu's method to find the global threshold
threshold = graythresh(grayImage);
% Convert the threshold to the range of the image data type
thresholdValue = threshold * 255;
% Apply the threshold to the grayscale image to create a binary image
binaryImage = imbinarize(grayImage, threshold);
% Display the original grayscale image
figure;
subplot(1, 2, 1);
imshow(grayImage);
title('Original Grayscale Image');
% Display the binary image
subplot(1, 2, 2);
imshow(binaryImage);
title('Binary Image (Otsu Method)');
% Explanation of the code
```

OUTPUT:



EXPLANATION:

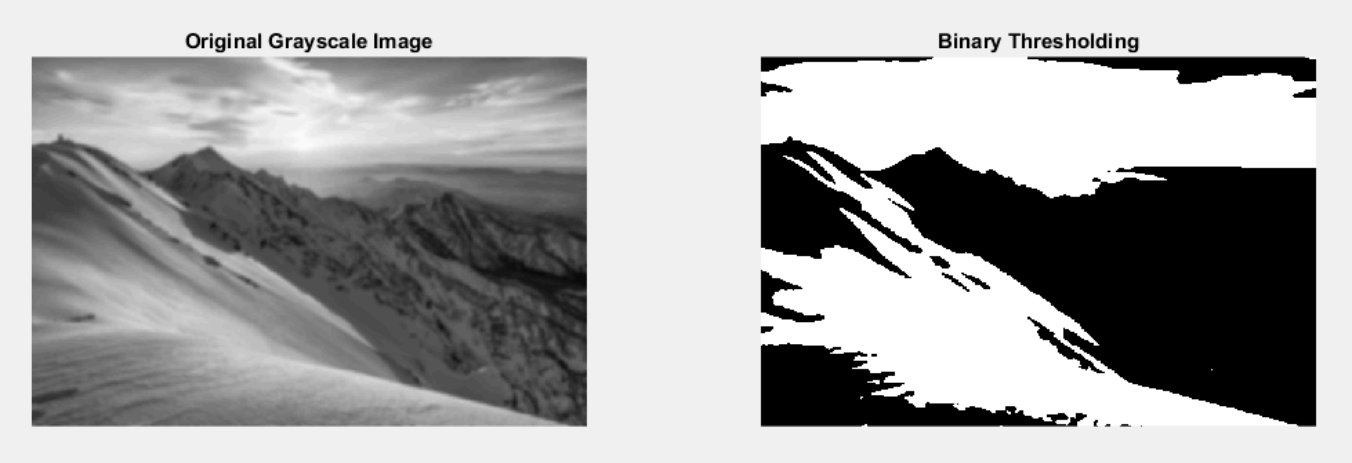
This MATLAB code reads an input image and converts it to grayscale if it is not already. It then applies Otsu's method to determine the optimal global threshold for separating the foreground and background. The `graythresh` function computes the threshold value, which is used to convert the grayscale image into a binary image using `imbinarize`. Finally, the original grayscale image and the resulting binary image are displayed side by side.

13. Apply binary thresholding to segment an image into foreground and background regions.

INPUT:

```
function main()
    % Load an image
    image_path = 'G:/4TH YEAR/Image_Processing/Pictures/mountain.jpg';
    image = imread(image_path);
    % Convert the image to grayscale
    gray_image = rgb2gray(image);
    % Apply Gaussian blur to reduce noise and smooth the image
    gray_image = imgaussfilt(gray_image, 2);  % Standard deviation of 2
    % Apply binary thresholding
    % Choose a threshold value; typically 127 is a good starting point for 8-bit grayscale images
    threshold_value = 127;
    binary_image = imbinarize(gray_image, threshold_value / 255);
    % Display the original and binary thresholded images
    figure('Position', [100, 100, 1200, 600]);
    % Original grayscale image
    subplot(1, 2, 1);
    imshow(gray_image);
    title('Original Grayscale Image');
    axis off;
    % Binary thresholded image
    subplot(1, 2, 2);
    imshow(binary_image);
    title('Binary Thresholding');
    axis off;
end
```

OUTPUT:



EXPLANATION:

This MATLAB program processes an image to convert it to grayscale, applies Gaussian blur to reduce noise, and then performs binary thresholding to create a binary image. The `main` function loads the image from a specified file path using `imread` and converts it to grayscale using `rgb2gray`. To reduce noise and smooth the image, a Gaussian blur is applied with `imgaussfilt`, using a standard deviation of 2. Binary thresholding is then performed with `imbinarize`, using a threshold value of 127, normalized to the range [0, 1]. The original grayscale image and the resulting binary image are displayed side by side using `imshow` in a figure window with two subplots, labeled appropriately using `title`, and the axes are turned off for a cleaner presentation. This allows for a visual comparison of the grayscale and thresholded images.

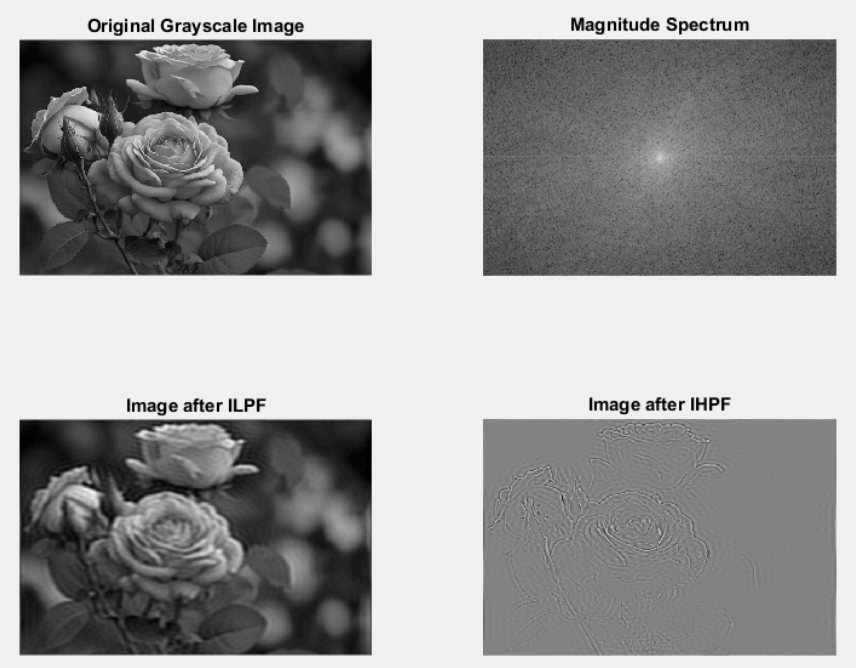
14. Apply Ideal Low Pass Filter (ILPF) and Ideal High Pass Filter (IHPF) for image smoothing.

INPUT:

```
% Read the image
originalImage = imread('G:/4TH YEAR/Image_Processing/Pictures/flowers.jpg');

% Convert the image to grayscale if it is not already
if size(originalImage, 3) == 3
        grayImage = rgb2gray(originalImage);
else
        grayImage = originalImage;
end
% Display the original grayscale image
figure;
subplot(2, 3, 1);
imshow(grayImage);
title('Original Grayscale Image');
% Compute the Fourier Transform of the image
F = fft2(double(grayImage));
F_shifted = fftshift(F);
% Compute the magnitude spectrum of the Fourier Transform
magnitude_spectrum = log(1 + abs(F_shifted));
% Display the magnitude spectrum
subplot(2, 3, 2);
imshow(magnitude_spectrum, []);
title('Magnitude Spectrum');
% Define the cutoff frequency for the filters
cutoff_frequency = 50; % Adjust this value as needed
% Create ideal low pass filter (ILPF)
[rows, cols] = size(grayImage);
[x, y] = meshgrid(1:cols, 1:rows);
center_x = floor(cols / 2) + 1;
center_y = floor(rows / 2) + 1;
ILPF = double(sqrt((x - center_x).^2 + (y - center_y).^2) <= cutoff_frequency);
% Apply ILPF to the magnitude spectrum
filtered_image_ILPF = ifftshift(F_shifted .* ILPF);
filtered_image_ILPF = real(ifft2(filtered_image_ILPF));
```

```
% Display the image after applying ILPF
subplot(2, 3, 4);
imshow(filtered_image_ILPF, []);
title('Image after ILPF');
% Create ideal high pass filter (IHPF)
IHPF = 1 - ILPF;
% Apply IHPF to the magnitude spectrum
filtered_image_IHPF = ifftshift(F_shifted .* IHPF);
filtered_image_IHPF = real(ifft2(filtered_image_IHPF));
% Display the image after applying IHPF
subplot(2, 3, 5);
imshow(filtered_image_IHPF, []);
title('Image after IHPF');
% Optional: Save the filtered images
imwrite(uint8(filtered_image_ILPF), 'filtered_image_ILPF.jpg');
imwrite(uint8(filtered_image_IHPF), 'filtered_image_IHPF.jpg');
```

OUTPUT:



EXPLANATION:

The provided MATLAB script performs a series of image processing tasks on a given grayscale image. First, it reads and converts the image to grayscale if necessary, displaying the original grayscale image. The script then computes the Fourier Transform of the image and shifts the zero-frequency component to the center. It calculates and displays the magnitude spectrum of the Fourier Transform. Next, the script creates an Ideal Low Pass Filter (ILPF) and an Ideal High Pass Filter (IHPF) based on a specified cutoff frequency. These filters are applied to the magnitude spectrum to produce filtered images, which are then inversely transformed back to the spatial domain. The resulting images, one highlighting low-frequency components (ILPF) and the other highlighting high-frequency components (IHPF), are displayed. Finally, the script optionally saves these filtered images to the disk. This process effectively demonstrates the application of frequency domain filtering to emphasize or suppress certain spatial features in the image.

15. Compute feature vector of an image using GLCM (Gray-Level Co-occurrence Matrix) or HOG (Histogram of Oriented Gradients) methods

INPUT:

```
% Load an image
image = imread('G:/4TH YEAR/Image_Processing/Pictures/flowers.jpg'); % Replace 'example.jpg' with your image file

% Convert image to grayscale if it is not already
if size(image, 3) == 3
    grayImage = rgb2gray(image);
else
    grayImage = image;
end
% GLCM (Gray-Level Co-occurrence Matrix) Method
% Compute the GLCM
glcm = graycomatrix(grayImage, 'Offset', [0 1; -1 1; -1 0; -1 -1]);
% Derive statistics from GLCM
stats = graycoprops(glcm, {'contrast', 'correlation', 'energy', 'homogeneity'});
% Convert stats to a feature vector
glcmFeatureVector = [stats.Contrast, stats.Correlation, stats.Energy, stats.Homogeneity];
% Display GLCM feature vector
disp('GLCM Feature Vector:');
disp(glcmFeatureVector);
% HOG (Histogram of Oriented Gradients) Method
% Compute the HOG feature vector
[featureVector, hogVisualization] = extractHOGFeatures(grayImage);
% Display HOG feature vector
disp('HOG Feature Vector:');
disp(featureVector);

% Visualize HOG features
figure;
imshow(grayImage);
hold on;
plot(hogVisualization);
title('HOG Features');
hold off;
```

OUTPUT:

EXPLANATION:

This MATLAB code loads an image, checks if it's grayscale, and converts it if not. It then computes texture features using both the Gray-Level Co-occurrence Matrix (GLCM) and Histogram of Oriented Gradients (HOG) methods. For GLCM, it calculates statistical properties such as contrast, correlation, energy, and homogeneity from the co-occurrence matrix, producing a feature vector. The code also extracts HOG features, which capture the gradient information in the image, generating another feature vector. These feature vectors provide quantitative representations of texture and structural characteristics of the image. Finally, it visualizes the HOG features overlaid on the grayscale image. This code demonstrates a comprehensive approach to texture analysis in image processing, combining both GLCM and HOG methods to capture different aspects of texture information.

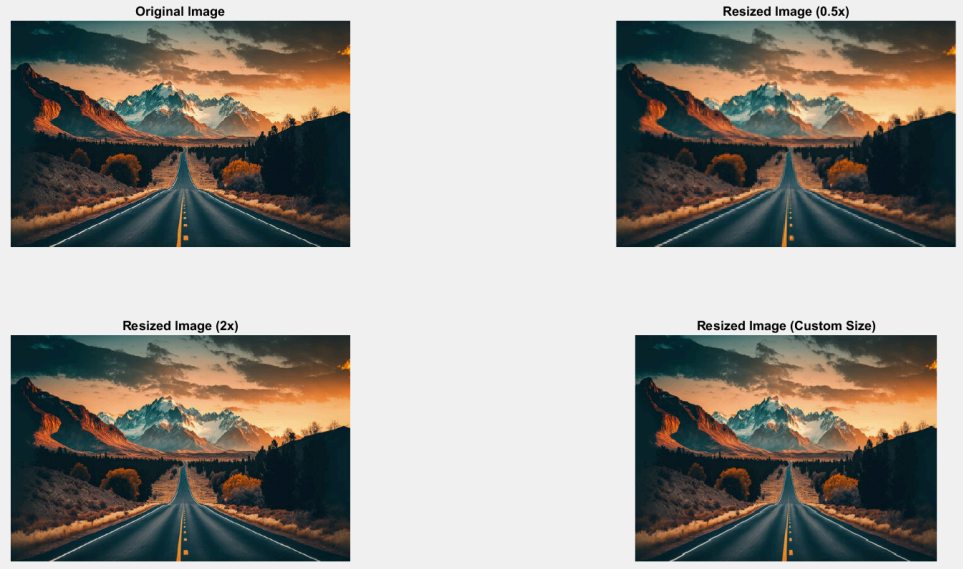
16. Image Resizing (Scaling): Display the color image and its Resized images

INPUT:

```
% Read the color image
originalImage = imread('D:\matlab installation\programs\images\new1.jpg');
% Display the original image
figure; % Create a new figure window
subplot(2, 2, 1); % Create a 2x2 grid and activate the first subplot
imshow(originalImage); % Display the original image
title('Original Image'); % Add a title to the first subplot
% Resize the image to 0.5 times its original size (scaling down)
resizedImage_half = imresize(originalImage, 0.5);
% Display the resized image (scaled down)
subplot(2, 2, 2); % Activate the second subplot
imshow(resizedImage_half); % Display the resized image
title('Resized Image (0.5x)'); % Add a title to the second subplot
% Resize the image to 2 times its original size (scaling up)
resizedImage_double = imresize(originalImage, 2);
% Display the resized image (scaled up)
subplot(2, 2, 3); % Activate the third subplot
```

```
imshow(resizedImage_double); % Display the resized image
title('Resized Image (2x)'); % Add a title to the third subplot
% Resize the image to a specific size [width, height]
newSize = [300, 400]; % Example: 300 pixels height, 400 pixels width
resizedImage_custom = imresize(originalImage, newSize);
% Display the resized image (custom size)
subplot(2, 2, 4); % Activate the fourth subplot
imshow(resizedImage_custom); % Display the resized image
title('Resized Image (Custom Size)'); % Add a title to the fourth subplot
% Optional: Save the resized images
imwrite(resizedImage_half, 'resized_half.jpg');
imwrite(resizedImage_double, 'resized_double.jpg');
imwrite(resizedImage_custom, 'resized_custom.jpg');
```

OUTPUT:



EXPLANATION:

This MATLAB code demonstrates image resizing using the imresize function. Initially, a color image named 'new1.jpg' is read and displayed in a 2x2 grid subplot layout, labeled as the "Original Image." The code then resizes the image to half its original size and displays it in the second subplot, titled "Resized Image (0.5x)." Similarly, the image is resized to twice its original size and displayed in the third subplot, labeled "Resized Image (2x)." Additionally, the image is resized to a custom size of 300 pixels in height and 400 pixels in width, and displayed in the fourth subplot titled "Resized Image (Custom Size)." Lastly, the resized images are optionally saved to disk as 'resized_half.jpg', 'resized_double.jpg', and 'resized_custom.jpg'. Overall, this code provides a simple demonstration of resizing an image to different scales and custom dimensions using MATLAB's imresize function, illustrating how to scale images both up and down efficiently for various applications.

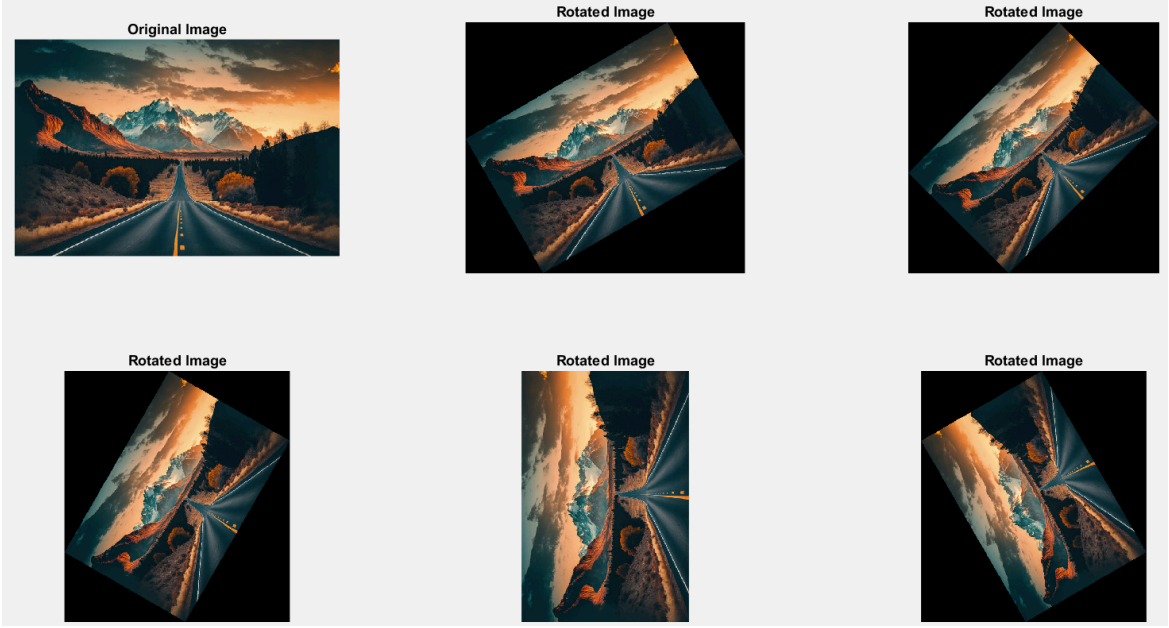
17. Image Rotation: Rotate an image in different angles 30, 45, 60…

INPUT:

```
% Read the color image
originalImage = imread('D:\matlab installation\programs\images\new1.jpg');
% Display the original image
figure;
subplot(2, 3, 1);
```

```
imshow(originalImage);
title('Original Image');
% Define the angles for rotation
angles = [30, 45, 60, 90, 120]; % Add or modify angles as needed
% Rotate and display the images at different angles
for i = 1:length(angles)
rotatedImage = imrotate(originalImage, angles(i));
% Display the rotated image
subplot(2, 3, i+1);
imshow(rotatedImage);
title('Rotated Image');
end
```

OUTPUT:



EXPLANATION:

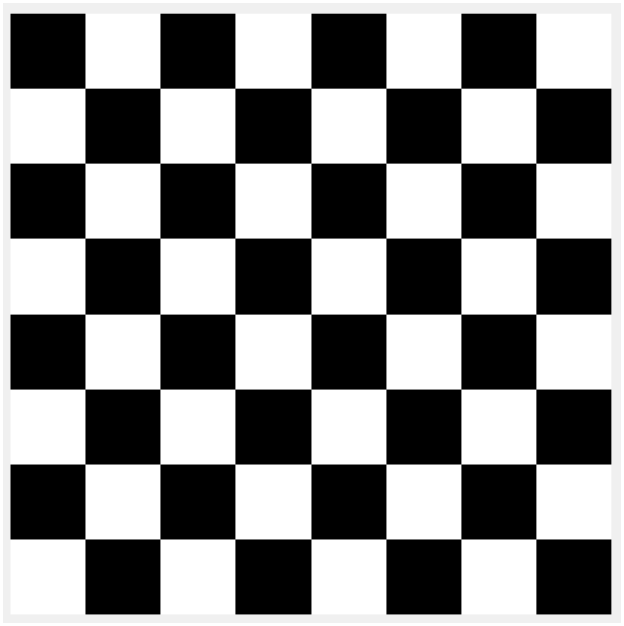
This MATLAB code reads a color image named 'new1.jpg' from a specified file path and displays it as the "Original Image" in a 2x3 subplot layout. It then defines an array of rotation angles (30, 45, 60, 90, and 120 degrees). Using a loop, it iterates through each angle, rotates the original image accordingly using the imrotate function, and displays the rotated images in separate subplots. Each subplot is labeled as "Rotated Image" with the corresponding rotation angle. Overall, the code provides a visual demonstration of how to rotate an image at different angles and display the results using MATLAB's built-in image processing functions.

18. Binary Image simulation. For example

INPUT:

```
a = zeros(100);
b = ones(100);
c =  [a,b,a,b,a,b,a,b; b,a,b,a,b,a,b,a;a,b,a,b,a,b,a,b; b,a,b,a,b,a,b,a;a,b,a,b,a,b,a,b;
b,a,b,a,b,a,b,a;a,b,a,b,a,b,a,b; b,a,b,a,b,a,b,a;];
imshow(c)
```

OUTPUT:



EXPLANATION:

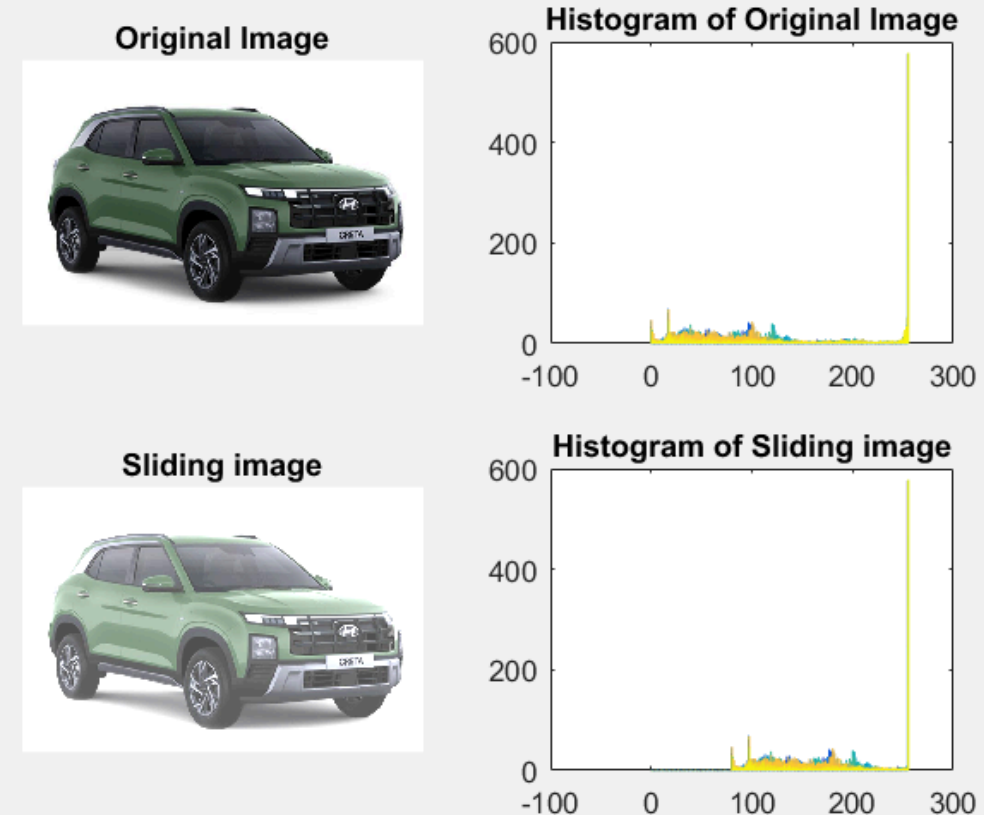
This MATLAB code generates a checkerboard pattern by creating two matrices, 'a' and 'b', consisting of zeros and ones, respectively, each with dimensions of 100x100. The matrices 'a' and 'b' represent alternating black and white squares. Subsequently, these matrices are arranged in a checkerboard pattern using concatenation along rows and columns to form a larger matrix 'c'. Each row of 'c' alternates between 'a' and 'b', creating a checkerboard pattern. Finally, the 'imshow' function is used to display the resulting checkerboard pattern, illustrating a grid of alternating black and white squares across the entire image. Overall, this code efficiently generates and visualizes a checkerboard pattern in MATLAB.

19. Histogram Sliding

INPUT:

```
img = imread('D:\matlab installation\programs\images\car.jpg');
subplot(2,2,1);imshow(img);title('Original Image');
subplot(2,2,2); hist(img, [0:255]); title('Histogram of Original Image');
img=img+80;
subplot(2,2,3); imshow(img);title('Sliding image');
subplot(2,2,4); hist(img, [0:255]); title('Histogram of Sliding image');
```

OUTPUT:



EXPLANATION:

This MATLAB code begins by reading and displaying an image named 'car.jpg' in the first subplot, labeled as the "Original Image." Subsequently, the code generates a histogram of pixel intensities for the original image, depicted in the second subplot, titled "Histogram of Original Image." Next, the image pixel values are increased by 80, simulating a sliding effect, and the modified image is displayed in the third subplot, labeled as the "Sliding image." Finally, a histogram of pixel intensities for the modified image is generated and displayed in the fourth subplot, titled "Histogram of Sliding image." In summary, this code visually demonstrates the effect of adjusting pixel values on an image and illustrates how histogram distributions change accordingly, providing insights into the impact of image processing operations on pixel intensity distributions.

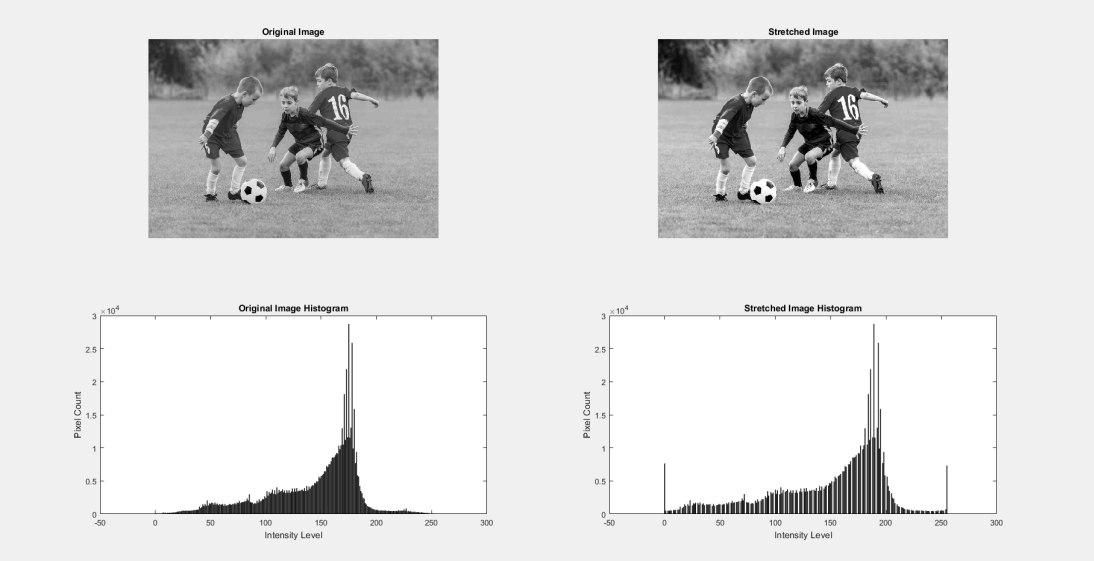
20. Histogram_Stretching

INPUT:

```
% Read the image
I = imread('Circles.jpg');
% Check if it's grayscale (convert to grayscale if RGB)
if ndims(I) == 3
  I = rgb2gray(I);
end
% Perform histogram stretching with imadjust and stretchlim
stretched_image = imadjust(I, stretchlim(I), []);
% **Ensure figure creation for image display**
figure;
% Display original and stretched image
subplot(2,2,1)
imshow(I)
title('Original Image')
subplot(2,2,2)
```

```
imshow(stretched_image)
title('Stretched Image')
% Get histograms (optional)
original_hist = imhist(I);
stretched_hist = imhist(stretched_image);
% Plot histograms (adjust bin count as needed)
subplot(2,2,3)
bar(0:255, original_hist);
xlabel('Intensity Level');
ylabel('Pixel Count');
title('Original Image Histogram');
subplot(2,2,4)
bar(0:255, stretched_hist);
xlabel('Intensity Level');
ylabel('Pixel Count');
title('Stretched Image Histogram');
```

OUTPUT:



EXPLANATION:

This MATLAB code performs histogram stretching on an input image named 'Circles.jpg'. It first checks if the image is grayscale and converts it to grayscale if it's in RGB format. Then, it applies histogram stretching using the imadjust function with the stretchlim function to automatically determine the intensity limits for stretching. The stretched image is displayed alongside the original image in a 2x2 subplot layout, with the original and stretched image titles provided. Additionally, histograms of the original and stretched images are computed and displayed below their respective images, showing the distribution of pixel intensities before and after stretching. This code effectively demonstrates histogram stretching, a technique commonly used in image processing to enhance the contrast of images by spreading out the pixel intensity values over the entire dynamic range.

21. Concept of Digitization - Sampling and Quantization

sampling and quantization in digital image processing

Sampling:

- **Definition**: Converts continuous signals into discrete signals by selecting values at specific intervals.

- **Nyquist-Shannon Theorem**: Sampling frequency must be at least twice the maximum frequency to avoid distortion.
- **Sampling Rate**: Determines the frequency range accurately represented in the digital domain.
- **Aliasing**: Insufficient sampling leads to false frequencies, mitigated by anti-aliasing filters.
- **Applications**: Widely used in audio, image processing, and telecommunications.

Quantization:

- **Definition**: Converts continuous values into a finite set of discrete levels.
- **Quantization Levels**: Determines precision; more levels mean higher fidelity.
- **Quantization Error**: Arises from the difference between the original signal and its quantized representation.
- **Uniform vs. Non-uniform**: Uniform has evenly spaced intervals; non-uniform allows varying step sizes.
- **Applications**: Crucial in compression, ADCs, and reducing storage/transmission bandwidth.

Sampling	Quantization
Digitization of co-ordinate values.	Digitization of amplitude values.
Sampling is done prior to the quantization process.	Quantizatin is done after the sampling process.
It determines the spatial resolution of the digitized images.	It determines the number of grey levels in the digitized images.
It reduces c.c. to a series of tent poles over a time.	It reduces c.c. to a continuous series of stair steps.
A single amplitude value is selected from different values of the time interval to represent it.	Values representing the time intervals are rounded off to create a defined set of possible amplitude values.

22. Concept of Image Enhancement

- **Definition**: Image enhancement improves image quality for specific needs.
- **Brightness and Contrast**: Adjusting brightness and sharpness enhances image appearance.
- **Histogram Equalization**: Balancing pixel intensities creates a more uniform contrast distribution.
- **Noise Reduction**: Removing noise with techniques like median filtering preserves image details.
- **Sharpening**: Enhancing edges and details makes images appear clearer.
- **Color Correction**: Adjusting color balance ensures accurate representation.
- **Spatial Domain**: Directly manipulating pixel values through operations like Gaussian blur enhances image quality.
- **Frequency Domain**: Manipulating frequency components through Fourier transform allows targeted feature enhancement.
- **Multi-Scale Enhancement**: Analyzing images at various resolutions improves detail preservation.
- **Applications**: Image enhancement is vital in fields like medicine, surveillance, and photography for better interpretation and usability.

23. Histogram

- **Definition:**

A histogram is a graph. A graph that shows frequency of anything. Usually histogram have bars that represent frequency of occurring of data in the whole data set. The histogram of a digital image with gray levels in the range $[0, L-1]$ is a discrete function and the histogram function is represented as:

$$H(r_k) = n_k$$

- **Application:**
  - **Image analysis**: Properties of an image can be predicted by the detailed study of the histogram.
  - **Brightness adjustment**: The brightness of the image can be adjusted by having the details of its histogram.
  - **Contrast adjustment**: The contrast of the image can be adjusted according to the need by having details of the x-axis of a histogram.
  - **Image equalization**: Histogram is used for image equalization. Gray level intensities are expanded along the x-axis to produce a high contrast image.
  - **Thresholding**: Histograms are used in thresholding as it improves the appearance of the image. This is mostly used in computer vision.
  - **Image transformation technique detection**: If we have an input and output histogram of an image, we can determine which type of transformation is applied in the algorithm.

2.2 Histogram Processing Techniques

Histogram sliding → used to increase image brightness.

Histogram stretching → used to increase contrast of an image.

Histogram equalization → used to increase contrast of an image.

Note:

Brightness → Emission of light by a particular light source.

Contrast → Max. pixel intensity - Min. pixel intensity

- **Histogram Sliding:**
  - A technique used to adjust the brightness of an image by shifting the histogram towards the right or left.
  - Shifting the histogram towards the right increases image brightness, while shifting it towards the left decreases brightness.
  - The brightness of an image is determined by the intensity of light emitted by its pixels, which is represented by gray level values on the histogram's x-axis.
  - Histograms display the frequency or count of each gray level intensity on the y-axis.
  - To increase brightness using histogram sliding, a constant value is added to all pixel intensities.
  - The decision of how much to shift the histogram depends on the distribution of pixel intensities in the original image.
  - In the provided example, a value of 50 is added to all pixel intensities to shift the histogram towards the whiter portion.
  - If the goal is to decrease brightness, a constant value is subtracted from all pixel intensities.
  - In the example, a value of 80 is subtracted to make the new image darker than the original.
  - The effectiveness of histogram sliding can be validated by examining the resulting histogram, which shows the shifted distribution of pixel intensities.
- **Histogram Stretching:**
  - A technique used to enhance the contrast of an image by expanding the dynamic range of pixel intensities.

    - Contrast = Maximum pixel intensity - Minimum pixel intensity.
    - Histogram stretching aims to cover the entire dynamic range of the histogram, effectively increasing the contrast.
    - By stretching the histogram, the pixel intensities are redistributed to utilize the full available range, resulting in a visually improved image.
    - The contrast of an image can be assessed by examining its histogram to determine if it covers the full dynamic range.
    - In the provided example, the contrast of an image is calculated by finding the difference between the maximum and minimum pixel intensities.
    - The contrast value obtained can indicate whether the image has low or high contrast.
    - Histogram stretching may encounter limitations, particularly when pixel intensities of 0 and 255 are present in the image.
    - In such cases, these extreme pixel values can become the minimum and maximum intensities, affecting the effectiveness of the stretching algorithm.
    - The presence of pixel intensities at the extremes of the dynamic range may lead to issues in accurately stretching the histogram, potentially resulting in suboptimal contrast enhancement.
- **Histogram Equalization:**
    - A technique used to equalize all pixel values in an image, resulting in a uniform flattened histogram.
    - The transformation applied during histogram equalization aims to produce a histogram with equal counts of pixels at each intensity level.
    - Unlike histogram stretching, where the shape of the histogram remains unchanged, histogram equalization alters the shape of the histogram to generate a single, enhanced image.
    - The primary goal of histogram equalization is to increase the dynamic range of pixel values and produce a high-contrast image with a flat histogram.
    - While histogram equalization generally enhances contrast, it's not guaranteed to do so in all cases. There are scenarios where histogram equalization may worsen contrast, leading to decreased image quality.
    - The effectiveness of histogram equalization depends on the distribution of pixel intensities in the original image.
    - In cases where histogram equalization is successful, the resulting image exhibits improved contrast and a more balanced distribution of pixel values.
    - Histogram equalization can be particularly beneficial for enhancing the visual appearance of images with uneven or skewed intensity distributions.
    - During histogram equalization, the overall shape of the histogram changes, distinguishing it from histogram stretching, where the shape remains the same.
    - The decision to apply histogram equalization should consider its potential impact on image quality and whether it aligns with the desired enhancement objectives.

24. Local contrast enhancement

- Local contrast enhancement selectively improves contrast in specific image regions, rather than globally.
- It targets localized areas to enhance visibility of details and textures while maintaining overall image quality.
- Techniques like adaptive histogram equalization (AHE) and contrast-limited adaptive histogram equalization (CLAHE) are commonly used.

- AHE divides the image into regions and adjusts contrast independently in each, allowing for adaptive enhancement.
- CLAHE limits contrast amplification to prevent over-enhancement and artifacts like noise.
- Local spatial filtering methods, such as unsharp masking, adjust pixel values based on neighboring pixels.
- It's effective for images with uneven lighting or varying textures.
- Adjustments should balance detail enhancement and avoiding unnatural effects.
- Local contrast enhancement improves image appearance by enhancing local details.
- It's crucial to tailor adjustments to the image's characteristics and desired outcome.

25. Concept of Image Smoothing

- Image smoothing reduces and suppresses image noise, enhancing image clarity.
- Neighborhood averaging in the spatial domain is a common technique for achieving smoothing.
- Common smoothing filters include average smoothing, Gaussian smoothing, and adaptive smoothing.
- Image sharpening, unlike smoothing, highlights edges by removing blur and enhancing grayscale transitions.
- Smoothing involves weighted summation or integral operations on the neighborhood, while sharpening relies on derivatives or finite differences to enhance edges.

26. Concept and types of Filtering

- **Definition:**
  - Filtering, also known as convolving a mask with an image, uses convolution masks to process digital images.
  - Smoothing operations in filtering remove noise and enhance image quality.
  - Filtering techniques can blur, reduce noise, sharpen, and detect edges in images.
  - Filters suppress high frequencies for smoothing and enhance low frequencies for edge detection and image enhancement.
  - The filtering process involves moving a predefined filter mask point by point over the image, applying the mask's relationship to calculate the response at each point (x, y).
- **Types:**
  - Linear Filter:
    - Simplest type of filter where each pixel is replaced by the average of its neighboring pixels.
    - Uses a weighted average instead of a simple average.
  - Median Filter:
    - A non-linear filter that replaces each pixel value with the median of its neighboring pixels.
    - Effective for removing salt-and-pepper noise.
  - Laplacian Filter:
    - Used to detect image edges by highlighting gray level discontinuities.
    - Based on the second spatial derivative of an image:

$$Laplace(f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

    - Uses a single kernel to detect second-order derivatives of image intensity levels, providing faster calculations.

- Gaussian Filter:
  - A 2-D convolution operator used to blur images and remove details and noise.
  - Similar to the mean filter but uses a Gaussian-shaped kernel that weights the center pixels more strongly than the boundaries.
- Neighborhood Average Filter (Mean Filter): Replaces pixel values with the average values of neighboring pixels.
- Box Filter:
  - A spatial domain linear image filter and a type of low-pass filter.
  - Operates similarly to average filtering techniques.
- Frequency Domain Filter:
  - Represents an image as the sum of many sine waves with different frequencies, amplitudes, and directions.
  - Parameters of sine waves are referred to as Fourier coefficients.

27. Image sharpening

- **Image Sharpening Techniques:**
  - Enhance fine details and highlight edges in a digital image.
  - Remove low-frequency components and preserve high-frequency components.
- **High Pass Filters (HPF):**
  - Used for sharpening images by reducing low-frequency components and preserving high-frequency ones.
  - Include Ideal Highpass Filter (IHPF) and Butterworth Highpass Filter (BHPF).
- **Ideal Highpass Filter (IHPF):**
  - Defined by the transformation function $H_{HP}(u,v) = 1 - H_{LP}(u,v)$
  - Passes frequencies outside a circle of radius D_0 and cuts off frequencies within the circle.
- **Butterworth Highpass Filter (BHPF):**
  - Defined by the transformation function **Transformation function:** $H_{HP}(u,v) = 1 - H_{LP}(u,v) = \frac{1}{1+\left[\frac{D_0}{D(u,v)}\right]^{2n}}$
  - Smoothly transitions between high and low frequencies based on the distance from the origin.
- **Sharpening vs. Smoothing:**
  - Sharpening enhances edges and grayscale transitions by using derivatives (gradients) or finite differences.
  - Smoothing is based on weighted summation or integral operations, focusing on noise reduction.

28. Spatial convolution

- **Correlation**
  - Correlation performs the weighted sum of overlapping pixels in the window between the image FFF and the filter HHH.
- **Convolution**
  - Convolution is similar to correlation but involves flipping the filter HHH first before performing the weighted sum.

Correlation

$$G[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[u,v]F[i+u, j+v]$$
$$= H \otimes F$$

Convolution

$$G[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[u,v]F[i-u, j-v]$$
$$G = H \star F$$

Fig. 3.10: Representation of correlation and convolution

29. Gaussian smoothing

- **Definition:**
  - Gaussian Smoothing (Gaussian Blur): A technique used to reduce noise and detail in images by convolving the image with a Gaussian filter kernel.
  - Effect: Blurs the image to make it smoother while preserving important structural features.
- **Gaussian Kernel:**
  - Definition and Shape: A Gaussian kernel is a 2D matrix with values computed based on the Gaussian function, forming a bell-shaped curve where values are highest at the center and decrease towards the edges.
  - Formula: The 2D Gaussian function is defined as: $G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$ where (x, y) are the coordinates in the kernel and σ (sigma) is the standard deviation, which controls the spread of the Gaussian distribution
  - Kernel Size and Truncation:
    - The kernel is centered at the origin (0,0), with values decreasing as the distance from the center increases.
    - The kernel is typically truncated at a certain radius because values far from the center become negligible.
  - Impact of Standard Deviation (σ):
    - The size and amount of smoothing by the Gaussian kernel are determined by its dimensions and the standard deviation (σ).
    - A larger σ value results in a wider kernel and more smoothing.
  - Example of a Gaussian Kernel:
    - A 3 × 3 Gaussian kernel with a standard deviation of 1 is $\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$ equals 1,
    - This kernel is normalized to ensure the sum of all elements with values computed based on the 2D Gaussian function.

30. DoG (Difference of Gaussians)

- **Definition and Purpose:**
  - The Difference of Gaussians (DoG) method enhances edges and detects key points in images.
  - It is obtained by subtracting two Gaussian-blurred versions of an image with different standard deviations.
- **Mathematical Representation**:
  - DoG is represented as: $DoG(x,y) = (G(x,y,\sigma_1) - G(x,y,\sigma_2)$

- $G(x, y, \sigma_1)$ and) $G(x, y, \sigma_2)$ are the Gaussian blurred images with standard deviations σ1 and σ2, respectively.

- **Gaussian Blur:**
  - Gaussian blur is a smoothing filter that convolves the image with a Gaussian kernel.
  - This process reduces noise and detail while preserving edges and important features.
- **Difference Operation:**
  - The difference in DoG refers to subtracting one Gaussian-blurred image from another.
  - This subtraction highlights edges and transitions in intensity within the image.
- **Application and Advantages:**
  - DoG is used in computer vision tasks, particularly in feature detection algorithms like SIFT.
  - It simplifies implementation by allowing edge extraction from stored Gaussian-filtered images without additional computation of image derivatives.

31. LoG (Laplacian of Gaussian)

- **Definition and Purpose:**
  - The Laplacian of Gaussian (LoG) is a method for edge detection in image processing.
  - It combines the smoothing effect of a Gaussian filter with the edge detection capabilities of the Laplacian operator.
- **Gaussian Blur:**
  - The process starts by applying a Gaussian blur to the input image.
  - This helps in reducing noise and detail while preserving important structural features.
- **Laplacian Operator:**
  - The Laplacian operator, denoted by ∇^2 or Δ, is a second-order derivative operator.
  - It measures the rate of change of intensity in an image, highlighting areas of rapid intensity change indicative of edges.
- **Combined Operation:**
  - LoG is obtained by convolving the Gaussian-blurred image with the Laplacian operator.
  - This enhances edges and highlights transitions in intensity, making it useful for edge detection.
- **Mathematical Representation and Applications:**
  - Mathematically represented as: $LoG(x, y) = \nabla^2 G(x, y, \sigma)$ where LoG(x, y) is the pixel value in the resulting image, G(x, y, σ) is the Gaussian-blurred image with standard deviation σ, and ∇^2 represents the Laplacian operator.
  - LoG is widely used in computer vision and image processing for edge detection and feature extraction tasks. It provides a robust method for detecting edges and intensity transitions, essential for various image analysis tasks.

32. Different types of image segmentation techniques

- **Thresholding Techniques:**
  - Definition and Purpose: Thresholding converts a grayscale image to a binary image with pixel values of 0 and 1.
  - Process of Thresholding: It separates the image into two regions based on a threshold value, assigning pixels greater than the threshold as 1 (white).
  - Binary Image Formation: Pixels below or equal to the threshold are assigned 0 (black), creating a binary image.
  - Threshold Selection: The threshold value, critical for segmentation, can be chosen manually, from histograms, or using methods like Otsu's algorithm.

    - Applications and Benefits: Thresholding is used for image segmentation, object detection, and background removal, providing clear separation of objects from the background.
    - Types:
        - Global Thresholding: Defines a single threshold value for the entire image to divide it into two regions, object and background.
        - Variable Thresholding: The threshold value varies over the image region, adapting to local characteristics.
        - Local or Regional Thresholding: Uses multiple thresholds for different regions in the image, with each threshold dependent on the properties of a neighborhood around a point.
        - Multiple-Thresholding: Classifies the image into three regions using two thresholds, suitable for cases with two distinct objects on a background, identified by peaks and valleys in the histogram.
- **Region/Graph-Based Segmentation Techniques:**
    - Graph-Based Segmentation: Techniques like lazy-snapping allow for segmenting an image into foreground and background regions, enhancing the accuracy of object separation.
    - MATLAB Implementation: This segmentation can be performed programmatically using the lazysnapping function or interactively through the Image Segmenter app.
- **Edge-based Techniques:**
    - Edge Detection: Highlights sharp intensity changes using operators like Sobel, Prewitt, and Canny.
    - Gradient-Based Methods: Detects edges by calculating gradient magnitude and direction with derivative operators.
    - Second-Derivative Methods: Uses second-order derivatives, like the Laplacian, to highlight rapid intensity changes.
    - Canny Edge Detector: A multi-step process including noise reduction, gradient calculation, and edge tracking to produce thin, continuous edges.
    - Edge Linking and Boundary Detection: Forms continuous edges by linking points; techniques include Hough Transform and curve fitting.
- **Clustering Techniques:**
    - Clustering Techniques: Used to create a segmented labeled image by applying specific clustering algorithms.
    - K-means Clustering: Segments an image into K clusters using the imsegkmeans function.
- **Deep Learning for Image Segmentation:**
    - Semantic Segmentation with CNNs: Utilizes convolutional neural networks to label every pixel in an image with a class, enabling detailed image understanding.
    - Applications: Includes autonomous driving, industrial inspection, medical imaging, and satellite image analysis.

33. Concept of thresholding

- **Definition and Purpose**: Thresholding converts a grayscale image to a binary image with pixel values of 0 and 1.
- **Process of Thresholding**: It separates the image into two regions based on a threshold value, assigning pixels greater than the threshold as 1 (white).
- **Binary Image Formation**: Pixels below or equal to the threshold are assigned 0 (black), creating a binary image.

- **Threshold Selection**: The threshold value, critical for segmentation, can be chosen manually, from histograms, or using methods like Otsu's algorithm.
- **Applications and Benefits**: Thresholding is used for image segmentation, object detection, and background removal, providing clear separation of objects from the background.
- **Types**:
    - Global Thresholding: Defines a single threshold value for the entire image to divide it into two regions, object and background.
    - Variable Thresholding: The threshold value varies over the image region, adapting to local characteristics.
    - Local or Regional Thresholding: Uses multiple thresholds for different regions in the image, with each threshold dependent on the properties of a neighborhood around a point.
    - Multiple-Thresholding: Classifies the image into three regions using two thresholds, suitable for cases with two distinct objects on a background, identified by peaks and valleys in the histogram.

34. Concept of Grey level thresholding, global/local thresholding

- **Global Thresholding**
    - Definition: Divides an image into object and background regions using a single threshold value TTT. Pixels with intensity greater than TTT are objects; others are background.
    - Application: Suitable for images with consistent lighting and clear object-background contrast.
- **Local or Regional Thresholding**
    - Definition: Uses multiple thresholds based on local image properties to segment regions.
    - Application: Effective for images with varying illumination and complex backgrounds.
- **Gray Level Thresholding**
    - Single-Level: Uses one threshold to segment the image into two regions, similar to global thresholding.
    - Multi-Level: Uses multiple thresholds to segment the image into several regions, ideal for images with multiple objects or intensity levels.

35. Types of edge detection operators with example

- **Global Thresholding**
    - Definition: Divides an image into object and background regions using a single threshold value T. Pixels with intensity greater than T are objects; others are background.
    - Application: Suitable for images with consistent lighting and clear object-background contrast.
- **Local or Regional Thresholding**
    - Definition: Uses multiple thresholds based on local image properties to segment regions.
    - Application: Effective for images with varying illumination and complex backgrounds.
- **Gray Level Thresholding**
    - Single-Level: Uses one threshold to segment the image into two regions, similar to global thresholding.
    - Multi-Level: Uses multiple thresholds to segment the image into several regions, ideal for images with multiple objects or intensity levels.

36. Concept and purpose of Hough transform.

- **Definition and Purpose**:
    - The Hough transform is a feature extraction method used to detect shapes, such as lines and circles, in images. It transforms points from image space to a parameter space.

- **Standard Hough Transform (SHT)**:
  - The hough function implements SHT to detect lines using the parametric representation of a line: ρ=xcos(θ)+ysin(θ)\rho = x \cos(\theta) + y \sin(\theta)ρ=xcos(θ)+ysin(θ). Here, ρ\rhoρ is the distance from the origin to the line along a vector perpendicular to the line, and θ\thetaθ is the angle between the x-axis and this vector.
- **Parameter Space Matrix**:
  - The hough function generates a parameter space matrix, where rows and columns correspond to ρ\rhoρ and θ\thetaθ values, respectively. This matrix is also known as the accumulator space.
- **Image Space to Hough Space Transformation**:
  - In image space, a line y=mx+cy = mx + cy=mx+c is represented by slope mmm and intercept ccc. This line transforms to a point in Hough space. However, to avoid issues with vertical lines (where mmm goes to infinity), polar coordinates (ρ,θ)(\rho, \theta)(ρ,θ) are used instead.
- **Polar Coordinates Representation**:
  - A line is represented by ρ\rhoρ (the length of the segment) and θ\thetaθ (the angle with the x-axis). This line in image space transforms to a point (ρ,θ)(\rho, \theta)(ρ,θ) in Hough space.
- **Constructing the Histogram Array**:
  - The Hough transform constructs a histogram array (M x N matrix) representing the parameter space. MMM corresponds to different values of ρ\rhoρ and NNN to different values of θ\thetaθ.
- **Accumulator Array**:
  - For each parameter combination (ρ,θ)(\rho, \theta)(ρ,θ), the accumulator array is incremented at the corresponding position for each edge pixel in the image that lies close to the line defined by ρ\rhoρ and θ\thetaθ.
- **Intersection Points**:
  - In Hough space, the intersection points of curves (sinusoidal for lines, circles for circles) indicate the presence of a specific geometric shape in the original image space. These points represent the parameters of the detected shapes.
- **Algorithm Steps**:
  - Determine the range of ρ\rhoρ and θ\thetaθ. Typically, θ\thetaθ ranges from 0 to 180 degrees and ρ\rhoρ ranges from -d to d, where d is the diagonal length of the edge.
  - Create a 2D accumulator array initialized to zero.
  - Perform edge detection on the original image.
  - For each edge pixel, calculate ρ\rhoρ for all θ\thetaθ values, update the accumulator array.
  - Iterate over the accumulator to retrieve ρ\rhoρ and θ\thetaθ values where the count exceeds a specified threshold.
- **Applications and Limitations**:
  - The Hough transform is used for detecting lines, circles, and other shapes. It can be computationally complex and is often replaced by learning-based methods for extracting complex features more efficiently.

37. What do you mean by Textural features

- **Definition and Importance**: Texture is a critical characteristic for identifying objects or regions in an image, representing complex visual patterns composed of spatially organized entities.
- **Characteristics of Texture**: Texture encompasses various attributes such as brightness, color, shape, and size, describing the spatial arrangement of these elements in an image or a selected region.
- **Role in Image Description**: Texture provides a detailed description of the spatial arrangement of colors or intensities within an image, enhancing the understanding of its structure and content.

- **Example in Facial Images**: In facial images, texture describes patterns and variations in skin, hair, and facial features, such as skin smoothness, wrinkles, pores, hair texture, and color variations.
- **Applications of Texture Analysis**: Texture analysis is useful for identifying unique features in images, aiding in tasks like facial recognition, emotion detection, and other image-based analyses.
- **Properties of Texture:**
    - Image texture is a function of spatial variation in pixel intensity.
    - Homogeneity: The degree of uniformity or consistency in texture across an image.
    - Directionality: The predominant orientation or directionality of texture elements within an image.
    - Contrast: The difference in intensity or color between adjacent texture elements.
    - Granularity: The size and distribution of texture elements, ranging from fine to coarse.
    - Regularity: The presence or absence of repeating patterns or structures within the texture.
    - Roughness/Smoothness: The perceived roughness or smoothness of the texture surface.
    - Coarseness/Fineness: The degree of detail or complexity in the texture pattern.
    - Depth: The perception of depth or three-dimensionality conveyed by the texture.
    - Anisotropy: The variation in texture properties with respect to different spatial directions.
    - Statistical Properties: Statistical measures such as mean, variance, skewness, and kurtosis of texture elements, used for texture analysis and classification.
- **Different types of texture images**
    - Coarse Texture: Large, irregular patterns with noticeable intensity or color variations. Examples: gravel, rough fabrics, rocky surfaces.
    - Fine Texture: Small, regular patterns with subtle intensity or color variations. Examples: fine fabrics, sandpaper, smooth surfaces with fine details.
    - High Contrast Texture: Sharp transitions between light and dark regions. Examples: checkerboard patterns, high-frequency noise, well-defined edges.
    - Low Contrast Texture: Subtle variations in intensity or color, creating a uniform and smooth appearance. Examples: clear skies, white walls, smooth paper.
    - Regular Texture: Regular, repeating patterns. Examples: bricks, tiles, fabrics with geometric shapes.
    - Irregular Texture: Lack of clear patterns with random variations. Examples: clouds, foliage, natural landscapes.
    - Directional Texture: Predominant orientation or directionality. Examples: wood grain, fur, fabric with distinct directional features.
    - Smooth Texture: Uniform and homogeneous appearance without significant variations. Examples: clear sky, flat wall, polished surface.
    - Rough Texture: Irregularities, bumps, or variations in surface roughness. Examples: gravel, tree bark, rocky terrain.
    - Random Texture: Lack of discernible patterns, with uniform intensity or color distribution. Examples: noise, speckle patterns, water, fog.
    - Granular Texture: Fine granules or particles creating a grainy appearance. Examples: sand, soil, fine-grained materials.
    - Fractal Texture: Self-similar patterns at different scales, complex and intricate structures. Examples: coastlines, mountains, natural phenomena.

38. How features are extracted using GLCM (i.e., step by step operations)

- **Image Preprocessing**: Enhance contrast and remove noise to ensure accurate texture representation.
- **Construct GLCM**: Count occurrences of pixel intensity pairs within a defined neighborhood to form a symmetric matrix.

- **Normalize GLCM**: Divide each element by the sum of all elements to make it invariant to brightness and contrast changes.
- **Calculate Features**: Compute statistical measures like contrast, correlation, energy, and homogeneity from the normalized GLCM.
- **Interpret Features**: Extracted features provide insights into texture properties, such as contrast levels, uniformity, and pixel dependencies, valuable for texture analysis tasks.

39. Basic concept about RGB, CMY, HSI, YCbCr color models

- **RGB Color Model:**
  - **Primary Colors**: RGB (Red, Green, Blue) combines varying intensities of these colors to create a wide range of hues.
  - **Representation**: Colors are represented as three-dimensional vectors with values ranging from 0 to 255 for each component.
  - **Applications**: Widely used in digital displays and graphics for accurate color reproduction.
- **CMY Color Model:**
  - **Complementary Colors**: CMY (Cyan, Magenta, Yellow) subtracts colors from white light to produce hues.
  - **Color Mixing**: Subtractive model used in color printing, where all colors combined at maximum intensity result in black.
  - **Printing**: Essential in color printing processes, combining CMY with black (K) ink for accurate color reproduction.
- **HSI Color Model:**
  - **Hue, Saturation, Intensity**: HSI represents colors based on dominant wavelength (Hue), purity (Saturation), and brightness (Intensity).
  - **Representation**: Colors are represented in a cylindrical color space, offering intuitive controls for color manipulation.
  - **Manipulation**: Used in image processing for intuitive color adjustments, altering hue, saturation, and intensity independently.
- **YCbCr Color Model:**
  - **Color Separation**: YCbCr separates images into luminance (Y) and chrominance (Cb, Cr) components.
  - **Chrominance Subsampling**: Used in video compression to reduce color information while preserving luminance.
  - **Applications**: Widely employed in broadcast television, digital video encoding, and compression algorithms for efficient transmission of high-quality video content.

40. Why is RGB called additive color model and CMY is called subtractive color model?

- **RGB (Additive Color Model):**
  - **Light Addition**: RGB combines red, green, and blue light to create colors, with maximum intensity producing white light.
  - **Digital Displays**: Used in screens where pixels emit light independently, with additive color mixing.
  - **Absence of Light**: Black is the absence of light, achieved by turning off all color channels.
  - **Wavelength Addition**: Each RGB color corresponds to a specific wavelength of light, added together to create different hues.
  - **Example**: Like shining flashlights of different colors (red, green, blue) onto a white surface to create various hues.
- **CMY (Subtractive Color Model):**

- **Ink Absorption**: CMY subtracts cyan, magenta, and yellow pigments from white light to create colors, with maximum absorption producing black.
- **Color Printing**: Used in printing, where inks subtract certain wavelengths of light from white paper.
- **Presence of Pigments**: Black is created by combining maximum amounts of cyan, magenta, and yellow inks.
- **Color Mixing**: Combining paints subtracts wavelengths of light, resulting in different hues.
- **Example**: Mixing cyan, magenta, and yellow paints on a white canvas to create various colors by subtracting light.

41. Basic idea about morphology

- **Image Structure Analysis**: Morphology involves analyzing the geometric structures of images using mathematical operations.
- **Structuring Elements**: Operations are performed using small binary patterns called structuring elements to probe the image.
- **Erosion and Dilation**: Basic operations include erosion (shrinking objects) and dilation (enlarging objects).
- **Opening and Closing**: Opening removes noise, while closing fills gaps, both achieved through combinations of erosion and dilation.
- **Applications**: Used for image segmentation, feature extraction, noise reduction, and object recognition in various fields like medical imaging and computer vision.

42. Meaning/purpose of Dilation and Erosion Operators, Top Hat Filters

- **Dilation and Erosion Operators**:
  - Dilation: Expands the boundaries of objects in an image by adding pixels to their perimeter, effectively enlarging them.
  - Erosion: Shrinks the boundaries of objects by removing pixels from their perimeter, making them smaller.
  - Purpose: Dilation is used to join nearby objects or to fill gaps in broken structures, while erosion helps in separating overlapping objects or removing noise from the image.
  - Structuring Element: Both operations are applied using a structuring element, which defines the neighborhood around each pixel to determine whether it should be added or removed.
  - Applications: Dilation and erosion are fundamental in morphological image processing for tasks like image segmentation, feature extraction, and noise reduction.
- **Top Hat Filters:**
  - Definition: Top Hat filtering is a morphological operation that enhances the brighter or darker regions of an image compared to its surroundings.
  - Purpose: The purpose of a top hat filter is to highlight small, local features or structures in an image that may be overshadowed by larger or more prominent features.
  - Types: There are two types of top hat filters: white top hat (highlighting bright features against a darker background) and black top hat (highlighting dark features against a brighter background).
  - Mathematical Operation: Top hat filtering is achieved by subtracting the original image from its opening (or closing), emphasizing local intensity variations.
  - Applications: Top hat filters are commonly used in image processing tasks such as defect detection, background subtraction, and enhancing small-scale structures or textures in medical and industrial images.

Gaussian noise

```
function main()
    % Define the image path
    image_path = 'G:/4TH YEAR/Image_Processing/Pictures/flowers.jpg';
    % Load the image
    image = imread(image_path);
    % Convert the image to double precision for noise addition
    image = double(image);
     % Add Gaussian noise to the image
    noisy_image = add_gaussian_noise(image);
    % Display the original and noisy images
    figure('Position', [100, 100, 1200, 600]);
    % Original image
    subplot(1, 2, 1);
    imshow(uint8(image));
    title('Original Image');
    axis off;
    % Noisy image
    subplot(1, 2, 2);
    imshow(uint8(noisy_image));
    title('Noisy Image');
    axis off;
end
function noisy_image = add_gaussian_noise(image, mean, std_dev)
    if margin < 2
        mean = 0;
    end
    if margin < 3
        std_dev = 25;
    end
        % Generate Gaussian noise
    noise = mean + std_dev * randn(size(image));
    % Add noise to the image
    noisy_image = image + noise;
    % Clip the values to keep them in valid range
    noisy_image = max(min(noisy_image, 255), 0);
end
```


Original Image


Noisy Image