Programme-1

Load, display, resize, crop, and save images; convert images between color spaces (RGB, grayscale, binary).

1. **Load the image**

img = imread('IMG_20241007_120658.jpg');

2. **Display the original image** figure,
imshow(img);
title('Original Image');

Original Image

3. **Resize the image**

resized_img = imresize(img, [300 400]); % Resize to 300x400 pixels figure, imshow(resized_img);
title('Resized Image');

Resized Image

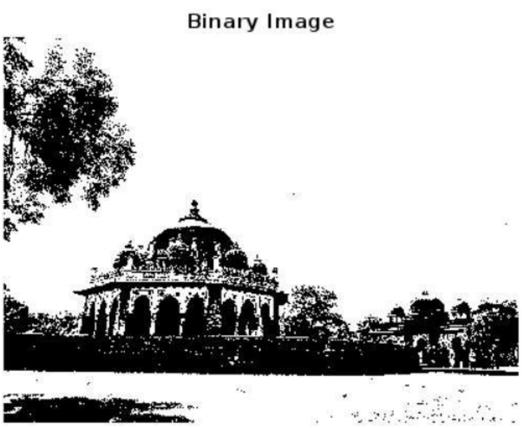
4. **Crop the image**

cropped_img = imcrop(img, [50 50 200 200]); % Crop with [x, y, width, height] figure, imshow(cropped_img);
title('Cropped Image');

5. **Convert to Grayscale** gray_img = rgb2gray(img); figure, imshow(gray_img); title('Grayscale Image');

6. **Convert to Binary (Black & White)**

bw_img = imbinarize(gray_img); % Convert grayscale to binary
figure, imshow(bw_img);
title('Binary Image');



<u>**Programme- 2**</u>

<u>**Perform histogram equalization on grayscale images, analyze contrast changes, and plot histograms for comparison.**</u>

1. **Load the image**

```
img = imread('IMG_20241007_120658.jpg');
```

2. **Convert to grayscale if not already**

```
gray_img = rgb2gray(img);
```

3. **Perform histogram equalization**

```
equalized_img = histeq(gray_img);
```

4. **Compute histograms**

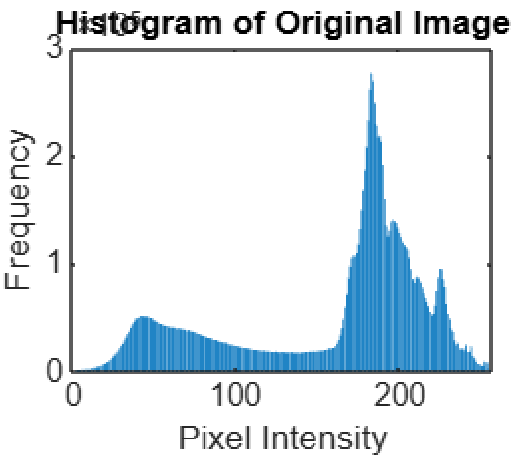
```
orig_hist = imhist(gray_img); % Histogram of original image
eq_hist = imhist(equalized_img); % Histogram of equalized
image
```

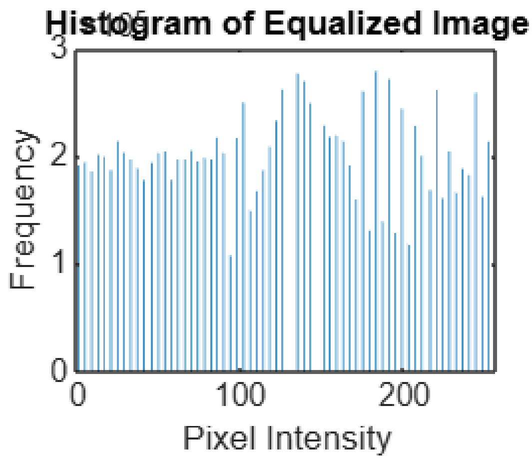
5. **Display the original and equalized images**

```
figure;
subplot(2,2,1);
imshow(gray_im
g);
title('Original Grayscale Image');

figure;
subplot(2,2,2);
imshow(equalized_img);
title('Histogram Equalized Image');
```

6. Plot histograms

```
figure;
subplot(2,2,3);
bar(orig_hist);
title('Histogram of Original Image');
xlabel('Pixel Intensity');
ylabel('Frequency');

figure;
subplot(2,2,4);
bar(eq_hist);
title('Histogram of Equalized
Image'); xlabel('Pixel Intensity');
ylabel('Frequency');
```





Programme-3

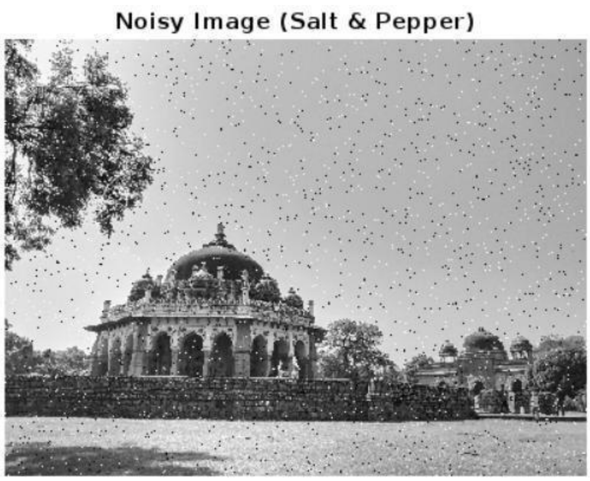
Apply mean, median, and Gaussian filters to noisy images, comparing results across filter types.

1. **Load the image**

```
img = imread('IMG_20241007_120658.jpg');
gray_img = rgb2gray(img); % Convert to grayscale if not already
figure, imshow(gray_img);
title('Original Grayscale Image');
```

2. **Add salt & pepper noise**

```
noisy_img = imnoise(gray_img, 'salt & pepper', 0.02);
figure, imshow(noisy_img);
title('Noisy Image (Salt & Pepper)');
```



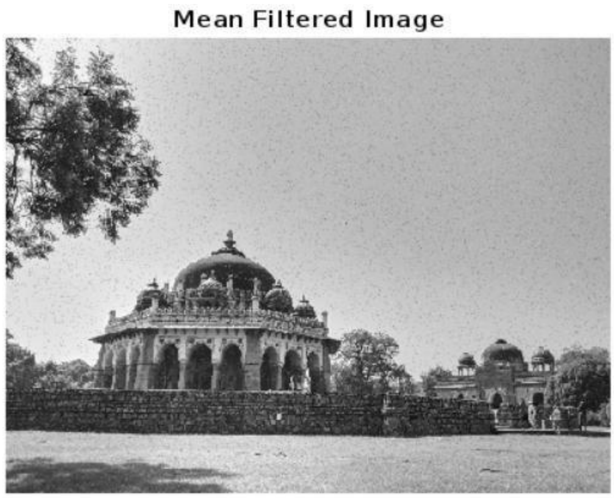
3. **Apply Mean filter (Average filter)**

mean_filter = fspecial('average', [3 3]);

mean_filtered = imfilter(noisy_img, mean_filter, 'replicate');

figure, imshow(mean_filtered);

title('Mean Filtered Image');



4. **Apply Median filter**

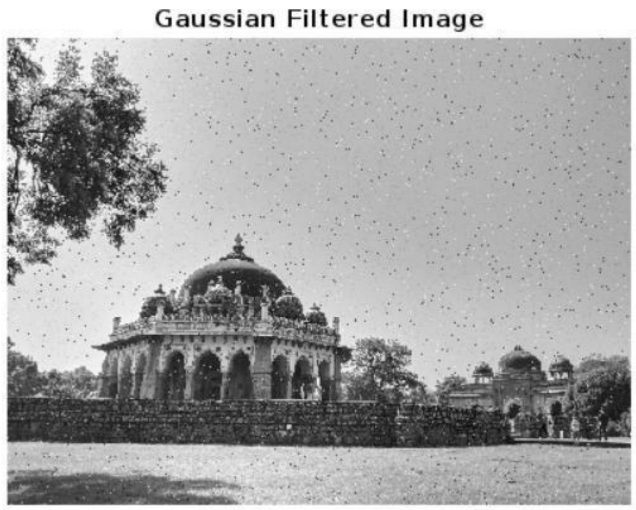
median_filtered = medfilt2(noisy_img, [3 3]);

figure, imshow(median_filtered);

title('Median Filtered Image');

5. **Apply Gaussian filter**

```
gaussian_filter = fspecial('gaussian', [3 3], 0.5);
gaussian_filtered = imfilter(noisy_img, gaussian_filter, 'replicate');
figure, imshow(gaussian_filtered);
title('Gaussian Filtered Image');
```

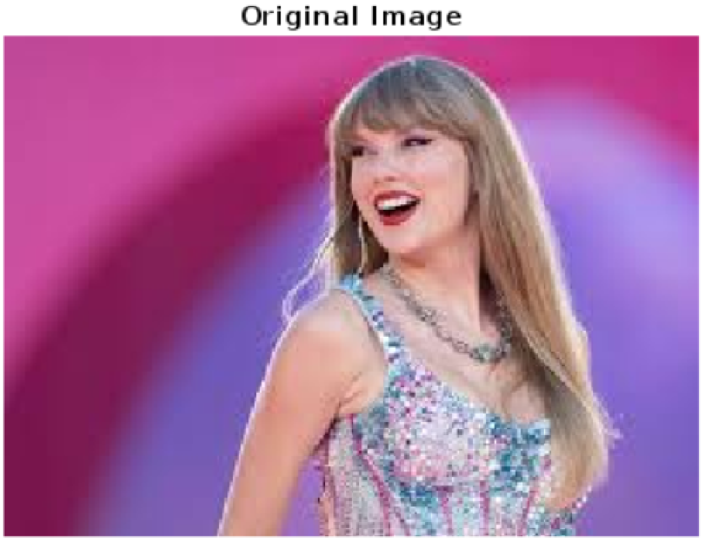


Programme-4

Apply high-pass and Laplacian filters to sharpen image details, analyze before and after images.

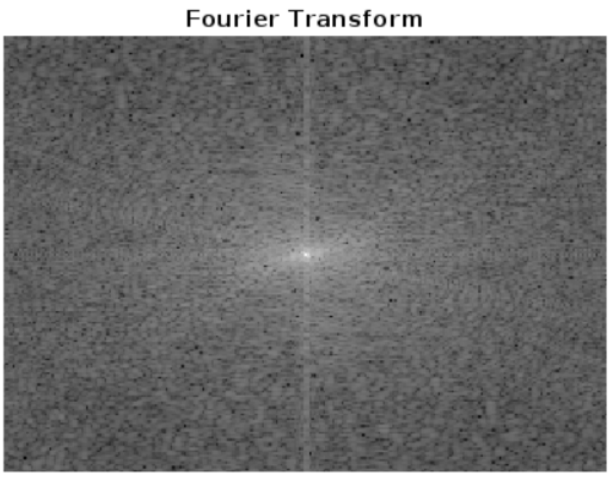
1. **Load Image and Convert to Grayscale**

```
img = imread('IMG_20241007_120658.jpg'); % Replace with your image
figure, imshow(img, []), title('Original Image');
img = rgb2gray(img); % Convert to grayscale if it's RGB
img = double(img); % Convert to double for processing
```



2. **Fourier Transform and Shift**

```
F = fft2(img);
F_shifted = fftshift(F); % Shift zero frequency to center
figure, imshow(log(1 + abs(F_shifted)), []), title('Fourier Transform');
```



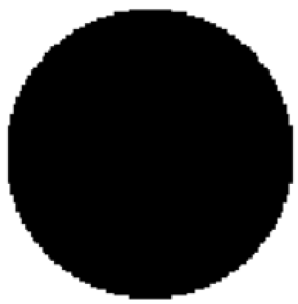
3. **Get Image Size**

```
[M, N] = size(img);
u = 0:M-1;
v = 0:N-1;
[U, V] = meshgrid(v - N/2, u - M/2); % Frequency coordinates
```

4. **Create High-Pass Filter (HPF) - Circular mask**

```
D0 = 50; % Cutoff frequency
HPF = double(sqrt(U.^2 + V.^2) > D0);
figure, imshow(HPF, []), title('High-Pass Filter Mask');
```

High-Pass Filter Mask



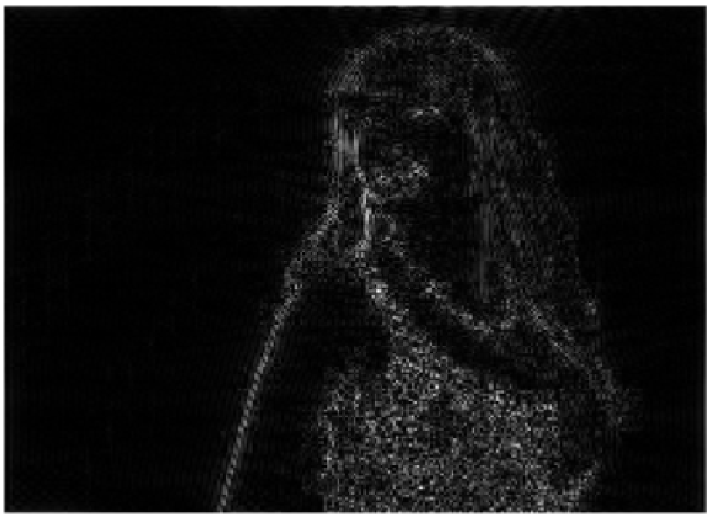
5. **Apply High-Pass Filter in Frequency Domain**

```
F_HPF = F_shifted .* HPF;
```

6. **Inverse FFT for HPF Image**

```
F_HPF_shifted = ifftshift(F_HPF);
img_HPF = abs(ifft2(F_HPF_shifted));
figure, imshow(img_HPF, []), title('High-Pass Filtered Image');
```

High-Pass Filtered Image



7. **Apply Laplacian Filter in Spatial Domain**

```
laplacian_filter = [0 -1 0; -1 4 -1; 0 -1 0];
img_Laplacian = imfilter(img, laplacian_filter, 'replicate');
figure, imshow(img_Laplacian, []), title('Laplacian Filtered Image');
```

Laplacian Filtered Image

8. **Sharpened Image by Adding Laplacian to Original**

```
img_Sharpened = img + img_Laplacian;
figure, imshow(img_Sharpened, []), title('Sharpened Image');
```

Sharpened Image

Programme-5

Use Fourier Transform to switch to frequency domain, apply low-pass and high-pass filters, and then convert back to spatial domain.

1. **Read image and convert to grayscale**

```
img = imread('IMG_20241007_120658.jpg');  % Replace with your image
figure, imshow(img, []), title('Original Image');
img = rgb2gray(img); % Convert to grayscale if it's RGB
img = double(img); % Convert to double for processing
```

Original Image

2. **Fourier Transform and shift**

```
F = fft2(img);
F_shifted = fftshift(F); % Centering the frequency components
```

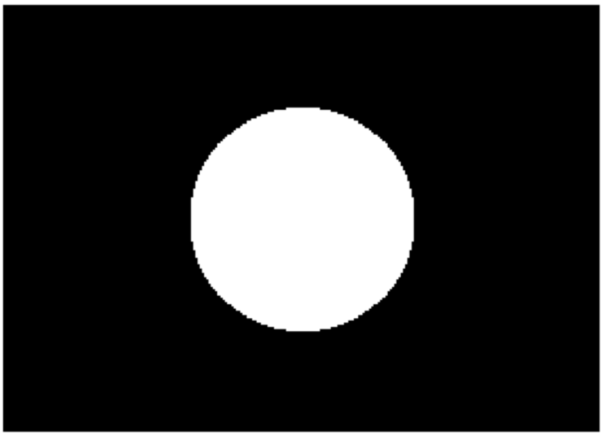
3. **Get image size**

```
[M, N] = size(img);
u = 0:M-1;
v = 0:N-1;
[U, V] = meshgrid(v - N/2, u - M/2); % Frequency coordinates
```

4. **Create Low-Pass Filter (LPF) - Circular mask**

```
D0 = 50; % Cutoff frequency
LPF = double(sqrt(U.^2 + V.^2) <= D0);
figure, imshow(LPF, []), title('Low-Pass Filter Mask');
```

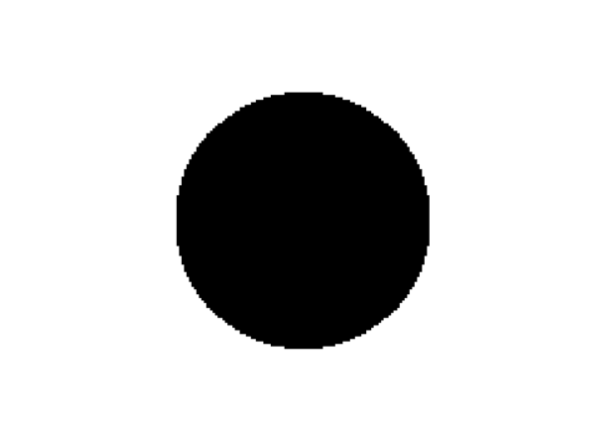
Low-Pass Filter Mask



5. **Create High-Pass Filter (HPF) - Circular mask**

```
HPF = double(sqrt(U.^2 + V.^2) > D0);
figure, imshow(HPF, []), title('High-Pass Filter Mask');
```

High-Pass Filter Mask



6. **Apply filters**

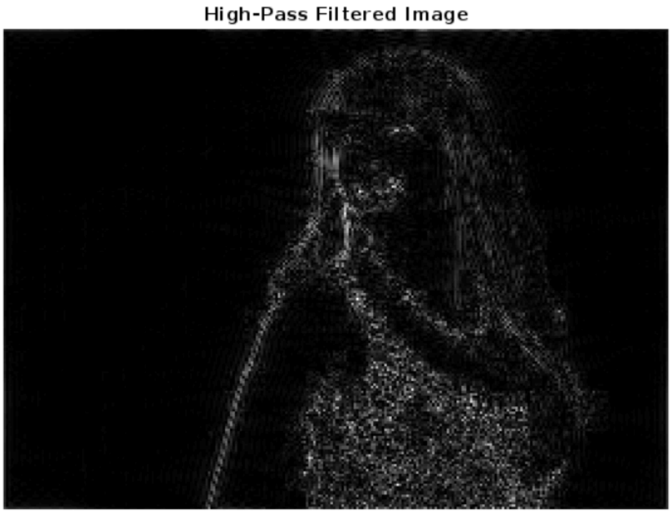
```
F_LPF = F_shifted .* LPF; % Low-pass filtered frequency domain
F_HPF = F_shifted .* HPF; % High-pass filtered frequency domain
```

7. **Inverse FFT (Low-Pass)**

```
F_LPF_shifted = ifftshift(F_LPF);
img_LPF = abs(ifft2(F_LPF_shifted));
figure, imshow(img_LPF, []), title('Low-Pass Filtered Image');
```

8. **Inverse FFT (High-Pass)**

```
F_HPF_shifted = ifftshift(F_HPF);
img_HPF = abs(ifft2(F_HPF_shifted));
figure, imshow(img_HPF, []), title('High-Pass Filtered Image');
```



Programme-6

Apply Wiener filter to noisy images, compare restored images with original and noisy versions.

1. **Read and convert image to grayscale**

```
img = im2double(imread('flower.jpg')); % You can replace with any image
```

2. **Add Gaussian noise**

```
noisy_img = imnoise(img, 'gaussian', 0, 0.01); % Mean=0, Variance=0.01
```

3. **Define filter window size**

```
window_size = 5;
half_window = floor(window_size / 2);
```

4. **Pad image to handle border pixels**

```
padded_img = padarray(noisy_img, [half_window, half_window], 'symmetric');
```

5. **Initialize filtered image**

```
filtered_img = zeros(size(noisy_img));
```

6. **Compute local mean and variance manually**

```
[M, N] = size(noisy_img);
for i = 1:M
   for j = 1:N
      % Extract local region
      local_window = padded_img(i:i+window_size-1, j:j+window_size-1);

      % Compute local mean and variance
      local_mean = mean(local_window(:));
      local_variance = var(local_window(:));

      % Noise variance estimation (Assuming Gaussian noise variance is known)
      noise_variance = 0.01;

      % Apply Wiener filtering formula
      filtered_img(i, j) = local_mean + (max(local_variance - noise_variance, 0) /
local_variance) * (noisy_img(i, j) - local_mean);
   end
end
```

7. **Display images**

```
figure;
imshow(img), title('Original Image');
figure;
imshow(noisy_img), title('Noisy Image');
figure;
```

```
imshow(filtered_img), title('Manually Filtered Image');
```

Original Image

Noisy Image

Manually Filtered Image

Programme-7

Apply Sobel, Prewitt, and Canny edge detectors, comparing sensitivity and accuracy of edge maps.

1. **Read the input image**

```
image = imread('image.jpg');
grayImage = rgb2gray(image);
```

2. **Apply edge detection algorithms to generate edge maps**

```
cannyEdges = edge(grayImage, 'Canny');
sobelEdges = edge(grayImage, 'Sobel');
prewittEdges = edge(grayImage, 'Prewitt');
```

3. **Read Ground Truth Image**

```
groundTruth = imread('ground_truth.png');
```

4. **Perform the comparative analysis for each edge detection method**

```
[resultsCanny] = evaluateEdgeDetection(cannyEdges, groundTruth);
[resultsSobel] = evaluateEdgeDetection(sobelEdges, groundTruth);
[resultsPrewitt] = evaluateEdgeDetection(prewittEdges, groundTruth);
```

5. **Display results**

```
fprintf('Canny - Sensitivity: %.4f, Accuracy: %.4f\n', ...
  resultsCanny.sensitivity,  resultsCanny.accuracy);
fprintf('Sobel - Sensitivity: %.4f,  Accuracy: %.4f\n', ...
  resultsSobel.sensitivity,  resultsSobel.accuracy);
fprintf('Prewitt - Sensitivity: %.4f,  Accuracy: %.4f\n', ...
  resultsPrewitt.sensitivity,  resultsPrewitt.accuracy);
```

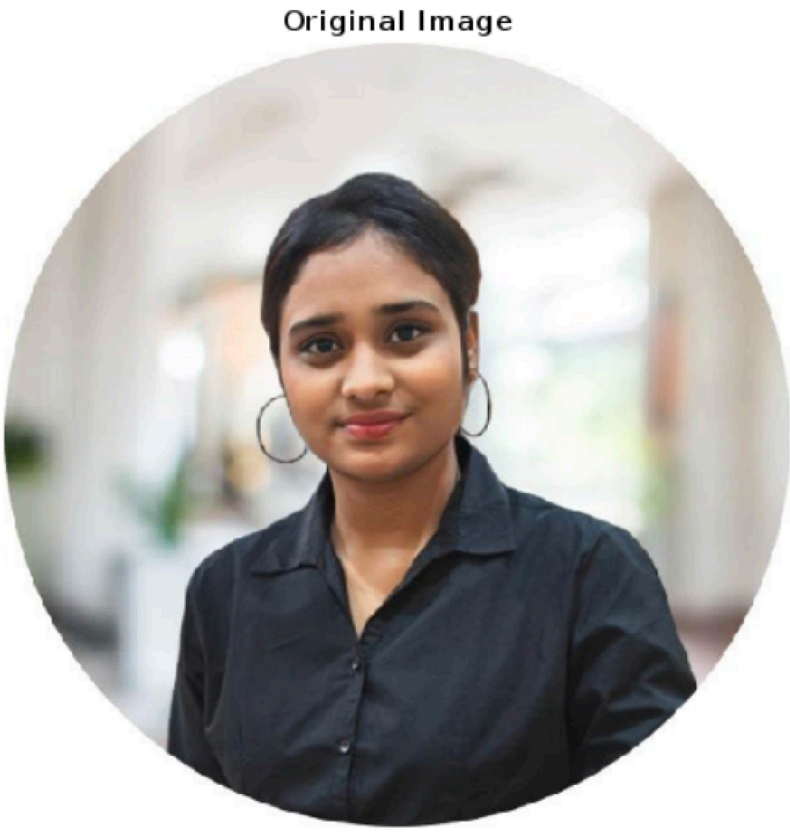
6. **Function to calculate the performance metrics for edge detection**

```
function results = evaluateEdgeDetection(detectedEdges, groundTruth)
  % True positives (TP)
  TP = sum(sum(detectedEdges == 1 & groundTruth == 1));

  % False positives (FP)
  FP = sum(sum(detectedEdges == 1 & groundTruth == 0));

  % False negatives (FN)
  FN = sum(sum(detectedEdges == 0 & groundTruth == 1));

  % True negatives (TN)
  TN = sum(sum(detectedEdges == 0 & groundTruth == 0));

  % Sensitivity (Recall or True Positive Rate)
  sensitivity = TP / (TP + FN);
```

```
  % Accuracy
  accuracy = (TP + TN) / (TP + FP + FN + TN);

  % Store results in a structure
  results.sensitivity = sensitivity;
  results.accuracy = accuracy;
```

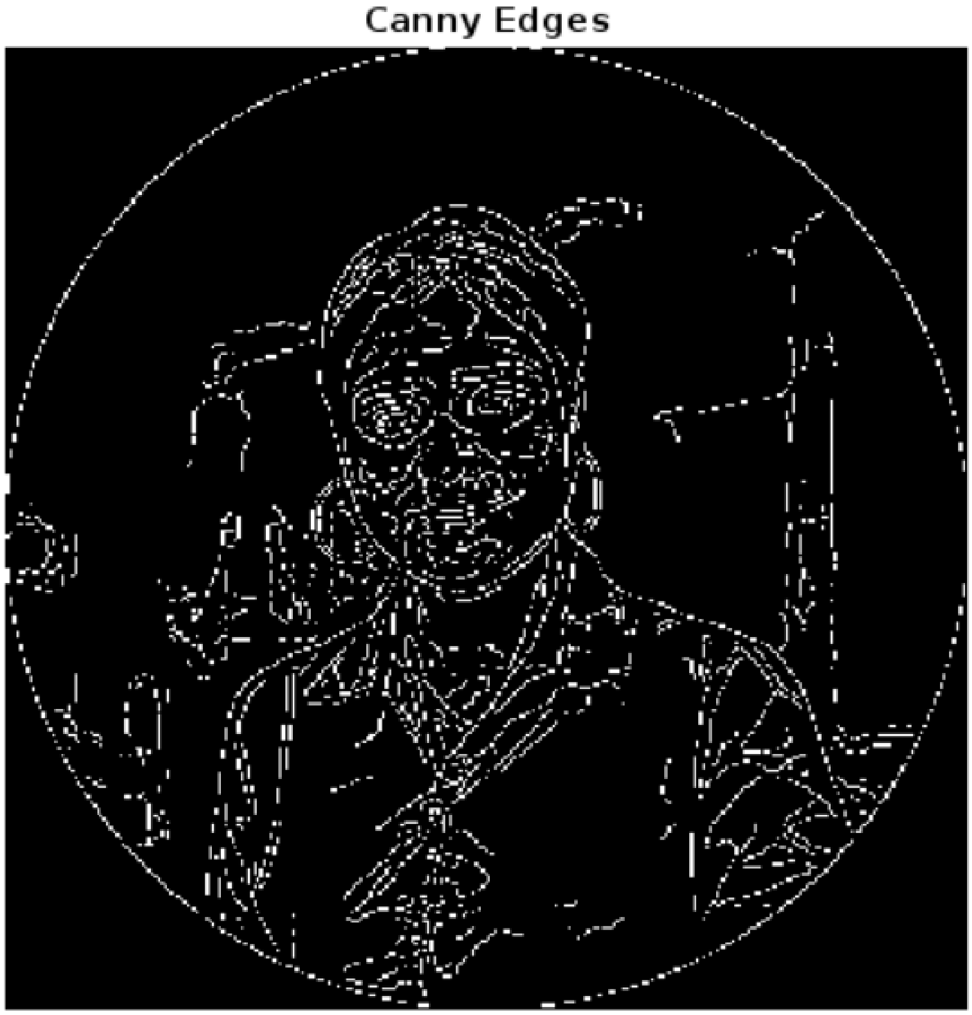
7. **Save the generated ground truth images**

```
imwrite(cannyEdges, 'ground_truth_canny.png');
imwrite(sobelEdges, 'ground_truth_sobel.png');
imwrite(prewittEdges, 'ground_truth_prewitt.png');

end
```



Sobel Edges

Prewitt Edges




```
>> p72
Canny - Sensitivity: 0.5407, Accuracy: 0.9543
Sobel - Sensitivity: 0.3314,  Accuracy: 0.9847
Prewitt - Sensitivity: 0.3310,  Accuracy: 0.9848
```

(fig. 1) Comparing sensitivity and accuracy of edge maps

Programme-8

Apply global and adaptive thresholding, followed by morphological operations (dilation, erosion) to isolate objects in images.

Read the image
img = imread('image.jpg');
grayImg = rgb2gray(img);

Global Thresholding (Otsu's Method)
level = graythresh(grayImg);   % Compute global threshold
globalBW = imbinarize(grayImg, level); % Apply threshold
Adaptive Thresholding (Local)
adaptiveBW = imbinarize(grayImg, 'adaptive', 'Sensitivity', 0.6); % Increased sensitivity
Morphological Operations
se = strel('disk', 3); % Structuring element (disk shape)
Dilation (expands object boundaries)
dilatedImg = imdilate(adaptiveBW, se);
Erosion (removes small noise)
erodedImg = imerode(dilatedImg, se);
Additional Morphological Processing
closedImg = imclose(erodedImg, se); % Closing to fill small holes
Edge Detection
edgeImg = edge(grayImg, 'canny'); % Detect edges using Canny method

Save output images
imwrite(globalBW, 'global_threshold.png');
imwrite(adaptiveBW, 'adaptive_threshold.png');
imwrite(dilatedImg, 'dilated.png');
imwrite(erodedImg, 'final_output.png');
imwrite(closedImg, 'closed.png');
imwrite(edgeImg, 'edges.png');

Original Image

(fig. 1)global_threshold

(fig. 2)final_output

(fig. 3)dilated

(fig.4) closed

(fig. 5) adaptive_threshold

Programme-9

Use morphological operations like opening, closing, and boundary extraction to analyze shapes and perform basic object recognition.

1. **Load the image**

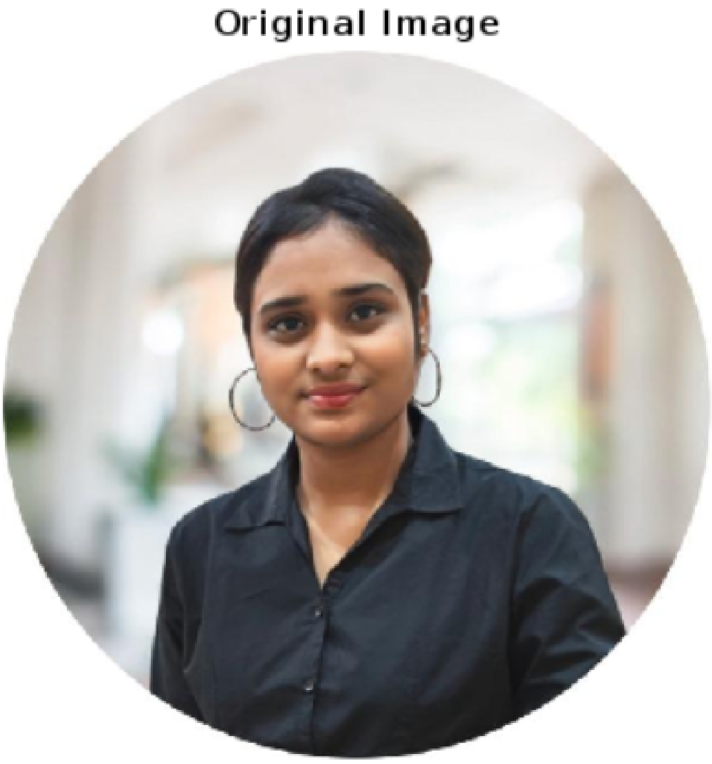
```
img = imread('image.jpg');
gray_img = rgb2gray(img); % Convert to grayscale if not already
binary_img = imbinarize(gray_img); % Convert to binary (black & white)
```

2. **Define structuring element**

```
se = strel('disk', 5); % Circular structuring element with radius 5
% Apply morphological opening (removes small noise)
opened_img = imopen(binary_img, se);
% Apply morphological closing (fills small gaps)
closed_img = imclose(binary_img, se);
% Perform boundary extraction (finds object edges)
boundary_img = binary_img - imerode(binary_img, se);
```

3. **Display images separately**

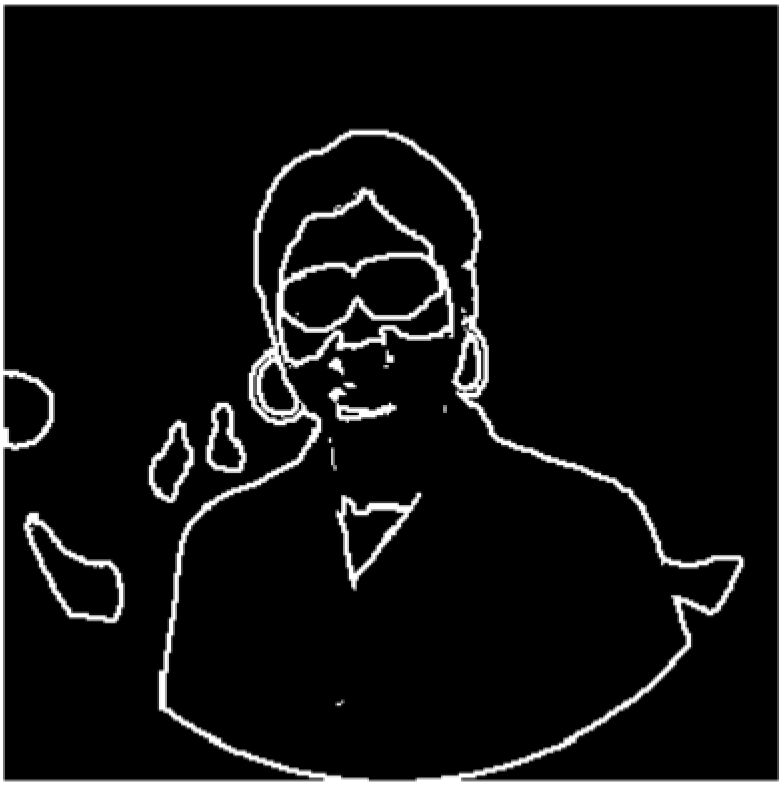
```
figure, imshow(binary_img);
title('Binary Image');
figure, imshow(opened_img);
title('After Morphological Opening');
figure, imshow(closed_img);
title('After Morphological Closing');
figure, imshow(boundary_img);
title('Boundary Extraction');
```



Binary Image

After Morphological Opening

Boundary Extraction



Programme-10

Compress an image using DCT, explore effects of compression on image quality, and analyze the trade-offs between quality and file size.

1. **Load the original color image**

```
img = im2double(imread('image.jpg'));
```

2. **Convert image to YCbCr color space (to process luminance separately)**

```
ycbcr_img = rgb2ycbcr(img);
Y = ycbcr_img(:,:,1); % Luminance channel
Cb = ycbcr_img(:,:,2);
Cr = ycbcr_img(:,:,3);
```

3. **Apply DCT on the luminance channel**

```
dct_Y = dct2(Y);
```

4. **Define compression levels (percentage of coefficients retained)**

```
compression_levels = [0.1, 0.3, 0.5]; % 10%, 30%, and 50% of coefficients retained
```

5. **Process images at different compression levels**

```
for i = 1:length(compression_levels)
  thresh = max(dct_Y(:)) * (1 - compression_levels(i)); % Define threshold
  dct_compressed = dct_Y .* (abs(dct_Y) > thresh); % Zero out small coefficients
  Y_reconstructed = idct2(dct_compressed); % Apply inverse DCT to reconstruct the
luminance channel

  % Reconstruct the color image in YCbCr space
  compressed_img = ycbcr_img;
  compressed_img(:,:,1) = Y_reconstructed; % Replace the Y channel

  % Convert back to RGB color space
  final_img = ycbcr2rgb(compressed_img);

  % Display and save results
  figure, imshow(final_img);
  title(['Reconstructed Image (', num2str(compression_levels(i) * 100), '% coefficients
retained)']);
```

6. **Save compressed image**

```
  filename = ['compressed_' num2str(compression_levels(i) * 100) '.jpg'];
  imwrite(final_img, filename);

end
```

Original Image

Reconstructed Image (10% coefficients retained)

Reconstructed Image (30% coefficients retained)

Reconstructed Image (50% coefficients retained)