Computer Vision

Name : Rishit Maurya

Roll No. : 224ME1001

Course : M.Tech

Branch : Robotics and Automation

Year : 1st Year

Camera and Imaging : Module 1 & Module 2

Objective

In this module, we aim to project a 3D cube onto a 2D screen using basic lens formulas. The cube is positioned in a 3D space, and its projection is computed using a pinhole camera model.

Concepts Used

- **3D Transformations:** The cube is translated to a new position.
- **Pinhole Camera Model:** Used to project 3D points onto a 2D plane.
- **Perspective Projection:** The projection formula involves dividing by the depth (z-coordinate).
- **Lens Formula used:**

$$\frac{1}{f} = \frac{1}{d_o} + \frac{1}{d_i}$$

  where:

  - f is the focal length of the camera
  - d_o is the distance of the object from the camera
  - d_i is the distance of the image from the camera (screen)

MATLAB Code

```
% Define the vertices of the cube in 3D space (centered at origin initially)
cubeVertices = [
    -0.5 -0.5 0.5;  % Vertex 1
     0.5 -0.5 0.5;  % Vertex 2
     0.5  0.5 0.5;  % Vertex 3
    -0.5  0.5 0.5;  % Vertex 4
    -0.5 -0.5 1.5;  % Vertex 5
     0.5 -0.5 1.5;  % Vertex 6
     0.5  0.5 1.5;  % Vertex 7
    -0.5  0.5 1.5   % Vertex 8
];

% Translate the cube to a new position
cubeTranslation = [0, 2, 3]; % [x, y, z]
cubeVertices = cubeVertices + cubeTranslation;

% Define the pinhole camera position
cameraPosition = [0, 0, 0]; % Camera at origin

% Pinhole camera parameters
focalLength = 2; % Distance of the pinhole from the image plane (units)
```

```
22
23 % Initialize matrix for projected points
24 projectedVertices = zeros(size(cubeVertices, 1), 2);
25
26 % Apply the lens formula for each vertex
27 for i = 1:size(cubeVertices, 1)
28     x_rel = cubeVertices(i, 1) - cameraPosition(1);
29     y_rel = cubeVertices(i, 2) - cameraPosition(2);
30     z_rel = cubeVertices(i, 3) - cameraPosition(3);
31
32     if z_rel > 0
33         x_proj = focalLength * (x_rel / z_rel);
34         y_proj = focalLength * (y_rel / z_rel);
35         projectedVertices(i, :) = [x_proj, y_proj];
36     else
37         projectedVertices(i, :) = [NaN, NaN];
38     end
39 end
40
41 % Plot the 3D cube in its translated position
42 figure;
43 subplot(1, 2, 1);
44 hold on;
45 grid on;
46 title('3D␣Cube␣in␣World␣Coordinates');
47 xlabel('X');
48 ylabel('Y');
49 zlabel('Z');
50 axis equal;
51 view(3);
52
53 plot3(cubeVertices(:, 1), cubeVertices(:, 2), cubeVertices(:, 3),
       'ro', 'MarkerSize', 8, 'LineWidth', 2);
54
55 edges = [
56     1 2; 2 3; 3 4; 4 1;
57     5 6; 6 7; 7 8; 8 5;
58     1 5; 2 6; 3 7; 4 8
59 ];
60
61 for i = 1:size(edges, 1)
62     plot3([cubeVertices(edges(i, 1), 1), cubeVertices(edges(i, 2)
          , 1)], ...
63           [cubeVertices(edges(i, 1), 2), cubeVertices(edges(i, 2)
              , 2)], ...
64           [cubeVertices(edges(i, 1), 3), cubeVertices(edges(i, 2)
              , 3)], 'k-', 'LineWidth', 1);
65 end
66
67 plot3(cameraPosition(1), cameraPosition(2), cameraPosition(3), 'b
     ^', 'MarkerSize', 10, 'LineWidth', 2);
```

```
text(cameraPosition(1), cameraPosition(2), cameraPosition(3), ' Camera', 'VerticalAlignment', 'bottom');
hold off;

subplot(1, 2, 2);
hold on;
axis equal;
grid on;
title('Projection of Cube Vertices onto 2D Screen');
xlabel('X (image plane)');
ylabel('Y (image plane)');

plot(projectedVertices(:, 1), projectedVertices(:, 2), 'ro', 'MarkerSize', 8, 'LineWidth', 2);

for i = 1:size(edges, 1)
    plot([projectedVertices(edges(i, 1), 1), projectedVertices(edges(i, 2), 1)], ...
         [projectedVertices(edges(i, 1), 2), projectedVertices(edges(i, 2), 2)], 'k-', 'LineWidth', 1);
end

hold off;
```

Output

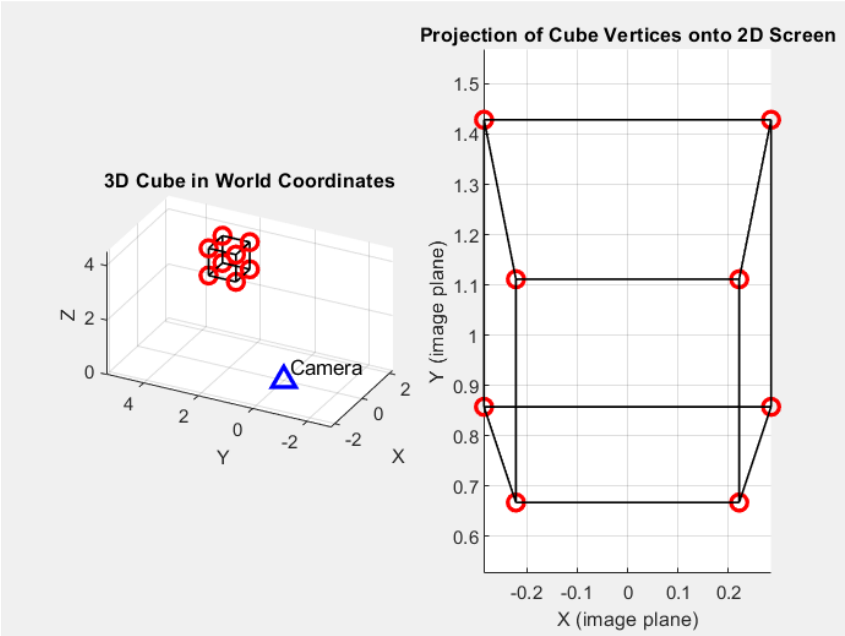

Figure 1: Projection of 3D Object to 2D Screen

Camera and Imaging : Module 3

Grayscale Image

A grayscale image is a digital image in which each pixel represents a different shade of gray, ranging from black to white. Unlike colored images, which use three color channels (Red, Green, and Blue), grayscale images contain only intensity information.

How Grayscale Images Are Created Using Thresholding

Grayscale images can be obtained by analyzing the intensity values of an image and applying a thresholding technique. Instead of using weighted RGB conversion, a simple approach is to set pixel values based on their brightness:

- Convert the image to a single intensity channel by computing the average of the Red, Green, and Blue components:

$$I = \frac{R + G + B}{3}$$

- Apply a threshold T to determine whether a pixel should be dark or bright:

$$I(x, y) = \begin{cases} 255, & \text{if } I(x, y) > T \\ 0, & \text{otherwise} \end{cases}$$

  where:

  - $I(x, y)$ is the intensity of the pixel at coordinates (x, y).
  - T is the threshold value (typically in the range 0-255).

Advantages of Threshold-Based Grayscale Conversion

- **Simplified Processing:** Thresholding removes color complexity and focuses on pixel intensity.
- **Useful for Binarization:** It helps in applications such as document scanning and object detection by creating clear black-and-white images.
- **Reduces Noise:** By ignoring subtle intensity variations, thresholding can eliminate minor noise in images.

MATLAB Code to Convert an RGB Image to Grayscale

```
% Clear workspace and command window
clc;
clear;
close all;

% Prompt user to select an image
[file, path] = uigetfile({'*.jpg;*.png;*.bmp;*.tif', 'Image␣Files
    '}, 'Select␣an␣Image');
```

```
if isequal(file, 0)
    disp('No image selected. Exiting...');
    return;
end

% Read the selected image
imagePath = fullfile(path, file);
img = imread(imagePath);

% Get image dimensions
[rows, cols, channels] = size(img);

% If the image is already grayscale, no conversion needed
if channels == 1
    grayscaleImg = img;
else
    % Convert to grayscale manually using the average method
    grayscaleImg = zeros(rows, cols, 'uint8'); % Initialize
       grayscale image

    for i = 1:rows
        for j = 1:cols
            R = img(i, j, 1);
            G = img(i, j, 2);
            B = img(i, j, 3);

            % Compute grayscale intensity using average method
            grayscaleImg(i, j) = uint8((R + G + B) / 3);
        end
    end
end

% Define threshold value (set to 128 by default)
threshold = 128;

% Apply thresholding manually
binaryImg = zeros(rows, cols, 'uint8'); % Initialize binary image

for i = 1:rows
    for j = 1:cols
        if grayscaleImg(i, j) > threshold
            binaryImg(i, j) = 255; % White pixel
        else
            binaryImg(i, j) = 0;   % Black pixel
        end
    end
end

% Display original and processed images
figure;

```

```
subplot(1, 3, 1);
imshow(img);
title('Original Image');

subplot(1, 3, 2);
imshow(grayscaleImg);
title('Grayscale Image');

subplot(1, 3, 3);
imshow(binaryImg);
title('Thresholded Image');

disp('Image processing completed.');
```

Output

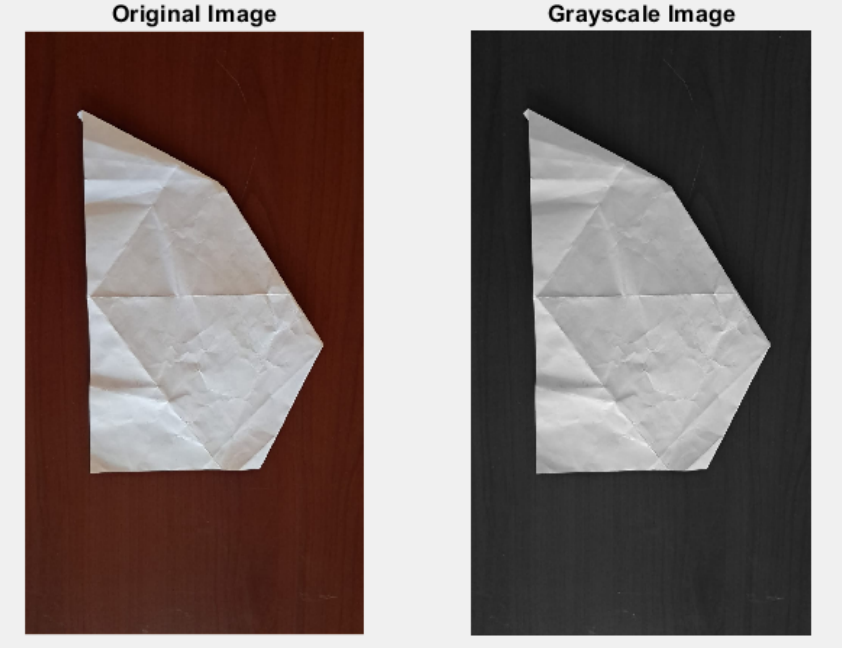

Figure 2: Conversion of RGB Image to Greyscale Image

Salt and Pepper Noise in Grayscale Images

Salt and Pepper Noise is a type of impulse noise that randomly replaces some pixel values in an image with either **black (0)** or **white (255)**, creating a speckled appearance. This noise typically occurs due to transmission errors, sensor defects, or image corruption.

Characteristics of Salt and Pepper Noise Salt and Pepper Noise can be identified by the following properties:

- Appears as **random white and black pixels** scattered throughout the image.
- Affects only a **certain percentage of pixels**, leaving the rest of the image unchanged.

- More prominent in images with low contrast.
- Often caused by **bit errors** in transmission or **faulty pixel sensors**.

Mathematical Representation Mathematically, Salt and Pepper Noise can be expressed as:

$$I'(x,y) = \begin{cases} 0, & \text{with probability } P_{\text{black}} \\ 255, & \text{with probability } P_{\text{white}} \\ I(x,y), & \text{otherwise} \end{cases}$$

where:

- $I(x,y)$ is the original grayscale pixel intensity.
- $I'(x,y)$ is the noisy pixel intensity.
- P_{black} and P_{white} are the probabilities of black and white noise, respectively.

Removal of Salt and Pepper Noise Several filtering techniques can be used to reduce Salt and Pepper Noise:

- **Median Filter:** Replaces each pixel with the median of its neighboring pixel values, effectively removing outliers.
- **Adaptive Filters:** Modify the filtering approach based on local noise density.
- **Morphological Operations:** Techniques like dilation and erosion can help clean up noise in binary and grayscale images.

MATLAB Code to add Salt and Pepper noise to Greyscale

```
% Read a grayscale image from the user
[file, path] = uigetfile({'*.jpg;*.png;*.bmp', 'Image Files (*.
    jpg, *.png, *.bmp)'});
if isequal(file, 0)
    disp('User selected Cancel');
    return;
end
img = imread(fullfile(path, file));

% Convert to grayscale if not already
dimensions = size(img);
if numel(dimensions) == 3
    img = rgb2gray(img);
end

% Add salt and pepper noise to the image
noisy_img = imnoise(img, 'salt & pepper', 0.02);

% Save the noisy image
imwrite(noisy_img, 'saltpeppergreyscaleimage.png');
```

```
20
21 % Display the original and noisy images
22 figure;
23 subplot(1,2,1), imshow(img), title('Original Grayscale Image');
24 subplot(1,2,2), imshow(noisy_img), title('Noisy Image (Salt &
     Pepper)');
```

Output

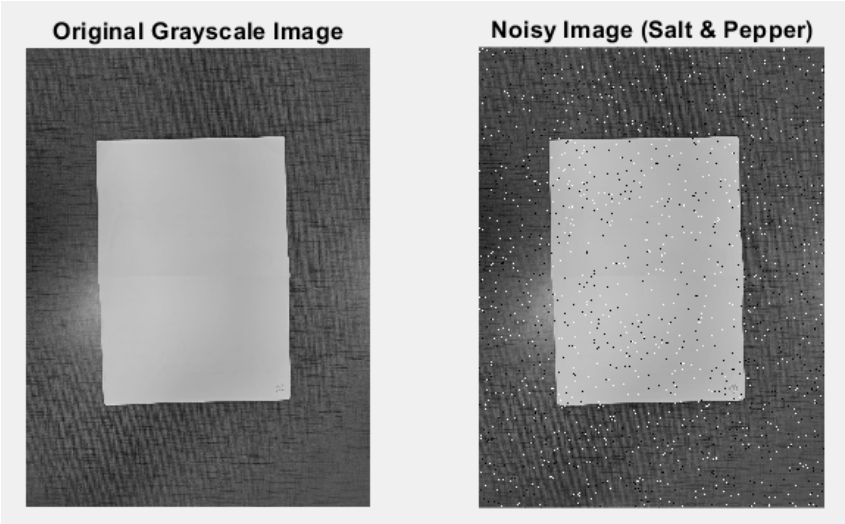

Figure 3: Add Salt and Perpper noise to greyscale image

Camera and Imaging : Module 4

Binary Images

Binary images are images that contain only two possible intensity values for each pixel: **black (0)** and **white (255)**. These images are often used in **object detection, segmentation, and pattern recognition**.

How Binary Images Are Generated

Binary images are created using **thresholding**, a technique that converts grayscale images into black and white based on a threshold value. The process follows these steps:

- Convert the original color image to **grayscale**.
- Set a **threshold value** (e.g., 128).
- Compare each pixel's intensity with the threshold:
  - If pixel intensity $I(x, y)$ is greater than the threshold, set it to **white (255)**.
  - Otherwise, set it to **black (0)**.

The thresholding formula is:

$$B(x, y) = \begin{cases} 255, & \text{if } I(x, y) > T \\ 0, & \text{if } I(x, y) \leq T \end{cases}$$

where:

- $I(x, y)$ is the grayscale pixel intensity.
- T is the threshold value (commonly **128**).
- $B(x, y)$ is the binary image output.

Geometric Properties of Binary Images

Once a binary image is obtained, we can analyze key geometric properties such as **centroid, orientation, and roundedness**.

Centroid The **centroid** (x_c, y_c) is the geometric center of the binary object, calculated as:

$$x_c = \frac{\sum xI(x, y)}{\sum I(x, y)}, \quad y_c = \frac{\sum yI(x, y)}{\sum I(x, y)}$$

where:

- $I(x, y)$ is **1** for object pixels and **0** for background.
- x_c and y_c represent the **mean position of all white pixels** in the binary object.

Orientation The **orientation** θ of the object describes its angle relative to the horizontal axis. It is given by:

$$\theta = \frac{1}{2} \tan^{-1} \left(\frac{2M_{11}}{M_{20} - M_{02}} \right)$$

where:

- $M_{pq} = \sum x^p y^q I(x, y)$ are **image moments**.
- M_{11} is the **mixed moment**.
- M_{20} and M_{02} are second-order central moments.

Roundedness The **roundedness** (or **eccentricity**) measures how circular an object is. It is defined as:

$$R = \frac{\lambda_{\min}}{\lambda_{\max}}$$

where:

- $\lambda_{\max}$ and $\lambda_{\min}$ are the **largest and smallest eigenvalues** of the covariance matrix.
- A value of $R \approx 1$ indicates a nearly circular object, while smaller values indicate elongated shapes.

Applications of Binary Images

Binary images are widely used in:

- **Edge Detection:** Used in **Canny, Sobel**, and **Prewitt** edge detectors.
- **Character Recognition:** Optical Character Recognition (OCR) converts text to binary for processing.
- **Medical Imaging:** Used in **X-ray** and **MRI segmentation**.
- **Object Detection:** Helps in detecting shapes, patterns, and contours.

MATLAB Code for Binary Image Conversion

```
% Step 1: Select an image file
[filename, pathname] = uigetfile({'*.jpg;*.png;*.bmp;*.tif', 'Image Files (*.jpg, *.png, *.bmp, *.tif)'}, 'Select a Grayscale Image');

% Check if the user selected a file
if isequal(filename, 0)
    disp('No file selected. Exiting...');
    return;
end

% Step 2: Read the selected image
```

```
imagePath = fullfile(pathname, filename);
grayImage = imread(imagePath);

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

% Step 3: Apply a custom threshold to create a binary image
threshold = 100;  % You can adjust this threshold as needed
binaryImage = grayImage > threshold;

savePath = fullfile(pathname, 'binaryimage.png');
imwrite(binaryImage, savePath);
disp(['Modified image saved to: ', savePath]);

% Get image dimensions
[rows, cols] = size(binaryImage);

% Initialize area and moment sums
area = 0;
sum_x = 0;
sum_y = 0;

% Initialize second moment variables
Mxx = 0;
Myy = 0;
Mxy = 0;

% Compute area, centroid sums, and second moments
for i = 1:rows
    for j = 1:cols
        if binaryImage(i, j) == 1  % Object pixel
            area = area + 1;   % Count object pixels
            sum_x = sum_x + j; % Sum x-coordinates
            sum_y = sum_y + i; % Sum y-coordinates
        end
    end
end

% Compute centroid coordinates
centroid_x = sum_x / area;
centroid_y = sum_y / area;

% Compute second moments relative to centroid
for i = 1:rows
    for j = 1:cols
        if binaryImage(i, j) == 1  % Object pixel
            x_rel = j - centroid_x;  % Distance from centroid (x)
            y_rel = i - centroid_y;  % Distance from centroid (y)

```

```
            Mxx = Mxx + y_rel^2;
            Myy = Myy + x_rel^2;
            Mxy = Mxy + x_rel * y_rel;
        end
    end
end

% Compute orientation angle    using:
theta = 0.5 * atan2(2 * Mxy, Mxx - Myy);
theta_degrees = rad2deg(theta);  % Convert to degrees

% Compute eigenvalues of second moment matrix
trace = Mxx + Myy;
determinant = (Mxx * Myy) - (Mxy^2);
lambda1 = (trace + sqrt(trace^2 - 4*determinant)) / 2; % Larger
    moment
lambda2 = (trace - sqrt(trace^2 - 4*determinant)) / 2; % Smaller
    moment

% Compute roundedness as the ratio of the smallest to largest
    second moment
roundedness = lambda2 / lambda1;

% Display results in command window
fprintf('Computed Area: %d\n', area);
fprintf('Centroid ( x  ,  ): (%.2f, %.2f)\n', centroid_x,
    centroid_y);
fprintf('Orientation Angle (  ): %.2f degrees\n', theta_degrees);
fprintf('Roundedness: %.4f\n', roundedness);

% Show the binary image with centroid marked
imshow(binaryImage);
title('Binary Image with Centroid and Orientation');
hold on;
plot(centroid_x, centroid_y, 'r+', 'MarkerSize', 10, 'LineWidth',
    2);

% Display the area and orientation as text on the image
text(10, 20, sprintf('Area: %d', area), 'Color', 'white', '
    FontSize', 12, 'FontWeight', 'bold');
text(10, 60, sprintf('Orientation: %.2 f  ', theta_degrees), '
    Color', 'white', 'FontSize', 12, 'FontWeight', 'bold');
text(10, 100, sprintf('Centroid: (%.2f, %.2f)', centroid_x,
    centroid_y), 'Color', 'white', 'FontSize', 12, 'FontWeight', '
    bold');
text(10, 140, sprintf('Roundedness: %.4f', roundedness), 'Color',
    'white', 'FontSize', 12, 'FontWeight', 'bold');

% Compute the slope of the orientation line
slope = tan(theta);

```

```
% Format the equation of the principal axis
equation_text = sprintf('y = %.2f(x - %.2f) + %.2f', slope,
    centroid_x, centroid_y);

% Display the equation on the image
text(10, 180, equation_text, 'Color', 'white', 'FontSize', 12, '
    FontWeight', 'bold');

% Plot orientation line
length_line = 190; % Length of orientation line
x1 = centroid_x - length_line * cos(theta);
y1 = centroid_y - length_line * sin(theta);
x2 = centroid_x + length_line * cos(theta);
y2 = centroid_y + length_line * sin(theta);
plot([x1, x2], [y1, y2], 'g-', 'LineWidth', 2);

hold off;
```

Output

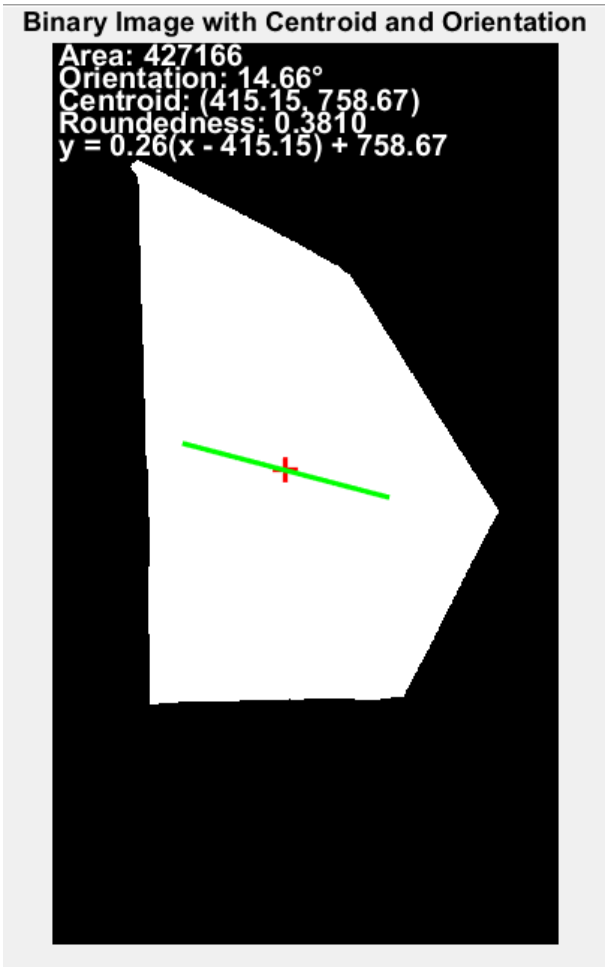

Figure 4: Conversion of greyscale to binary and calculating Area, Orientation, Centroidand Roundedness

Object Counting in a Binary Image

Object counting in binary images involves identifying and labeling distinct connected components. This process is achieved using **Connected Component Labeling (CCL)**, a fundamental technique in image processing.

The MATLAB script provided follows these steps to count objects in a binary image:

1. **Binary Image Input:** The user selects a binary image where objects are represented by **white (1)** pixels and the background by **black (0)** pixels.
2. **Connected Component Labeling (CCL):** The script applies a **flood fill** algorithm using a stack-based **Breadth-First Search (BFS)** to label connected components.
3. **4-Connected Neighborhood:** The algorithm considers four possible neighbors (up, down, left, right) for connectivity.
4. **Object Counting:** The number of distinct labels assigned corresponds to the number of objects in the binary image.

Connectedness and Neighbors in Binary Images

Connectivity defines how pixels are grouped into components based on their adjacency. It determines whether neighboring pixels belong to the same object.

4-Connected Neighborhood

A pixel (x, y) is 4-connected if it shares an edge with another pixel. The four neighbors are:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & X & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

where:

- X is the reference pixel.
- The 1s represent valid neighbors (up, down, left, right).

8-Connected Neighborhood A pixel is 8-connected if it shares either an edge or a corner with another pixel. The eight neighbors are:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & X & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

This connectivity is more inclusive and can merge diagonal objects into a single component.

6-Connected Neighborhood (Hexagonal Grid) In a hexagonal pixel grid, each pixel has six neighbors:

$$\begin{bmatrix} 1 & 1 & \\ 1 & X & 1 \\ & 1 & 1 \end{bmatrix}$$

This connectivity is mainly used in specialized imaging applications like crystal structure analysis.

Flood Fill Algorithm for Labeling Objects The script uses a **stack-based flood fill algorithm** to label objects. The process follows these steps:

1. Start with an unvisited object pixel.
2. Assign a new label and push the pixel onto the stack.
3. Pop a pixel from the stack, label it, and check its neighbors.
4. Add all valid, unvisited neighbors to the stack.
5. Repeat until the stack is empty.

Applications of Connected Component Labeling Connected Component Labeling is used in:

- **Medical Image Processing:** Counting blood cells, tumor segmentation.
- **Automated Inspection Systems:** Detecting defects in industrial products.
- **Optical Character Recognition (OCR):** Identifying letters and digits.
- **Surveillance and Object Tracking:** Detecting objects in binary motion masks.

MATLAB Code to count objects in Binary image

```
1
2
3  % Load binary image (User can uncomment the next line to load an
       actual image)
4  [filename, pathname] = uigetfile({'*.jpg;*.png;*.bmp;*.tif', '
       Image Files (*.jpg, *.png, *.bmp, *.tif)'}, 'Select a Binary
       Image');
5
6  % Check if the user selected a file
7  if isequal(filename, 0)
8      disp('No file selected. Exiting...');
9      return; % Exit the script if no file is selected
10 end
11
12 % Construct the full file path
13 imagePath = fullfile(pathname, filename);
14
15 % Read the selected binary image
16 binary_image = imread(imagePath);
```

```
17
18  % Compute the Euclidean distance transform of the inverted binary
        image
19  D = bwdist(~binary_image);
20
21  % Invert the distance map to identify local minima
22  D = -D;
23
24  % Suppress small minima to prevent over-segmentation
25  D = imhmin(D, 2);
26
27  % Apply the watershed transform for segmentation
28  L = watershed(D);
29
30  % Ensure the background remains zero after watershed
31  L(~binary_image) = 0;
32
33  % Convert the labeled regions to a binary image
34  L = imbinarize(L);
35
36  % Remove small objects with an area less than 500 pixels
37  L = bwareaopen(L, 500);
38
39  % Assign random colors to different segmented regions
40  RGB = label2rgb(L, 'jet', 'k', 'shuffle');
41
42  % Display the segmented image with random colors
43  imshow(RGB);
44
45  % Identify connected components in the binary image
46  CC = bwconncomp(L);
47
48  % Compute the centroid of each connected component
49  S = regionprops(CC, 'Centroid');
50
51  % Display the number of detected cells in the title
52  title("Number of Cells: " + num2str(numel(S)));
53
54  % Hold the current figure for plotting centroids
55  hold on;
56
57  % Loop through each detected object and plot its centroid
58  for k = 1:numel(S)
59      centroid = S(k).Centroid; % Get the centroid coordinates
60      plot(centroid(1), centroid(2), 'r+', 'MarkerSize', 10, '
            LineWidth', 2); % Mark the centroid with a red cross
61  end
62
63  % Release the hold on the current figure
64  hold off;
```

Output

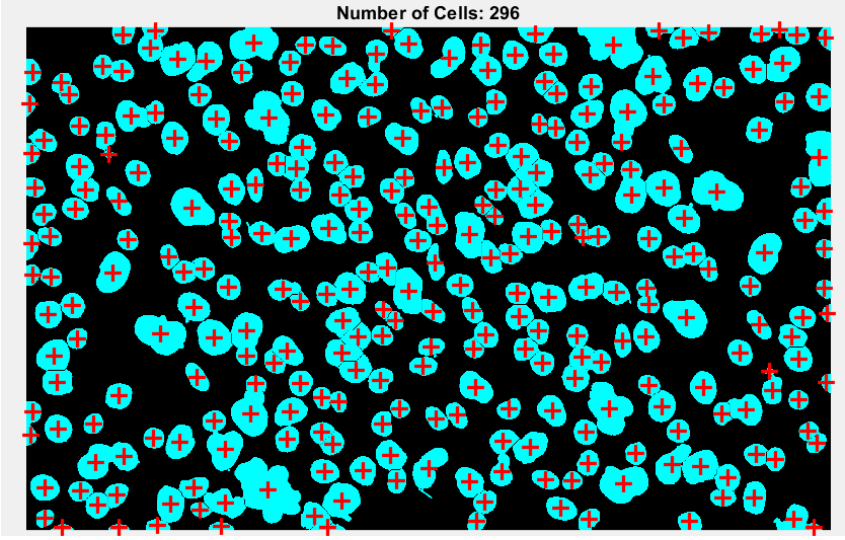

Figure 5: Counting objects in binary image

Iterative Modification Algorithm

Iterative modification algorithms apply a set of transformation rules to a binary image based on pixel neighborhoods. The process iteratively modifies pixel values according to predefined conditions.

Notation and Neighborhood Definition A neighborhood set S is defined, which can be any combination of predefined sets such as $**N+1$, $N0$, $N-1$, $N-2**$. Each pixel is represented as (i, j), and its neighborhood configuration is used to determine modifications.

Let:

- $a_{ij} = 1$ if the neighborhood of pixel (i, j) belongs to S.
- b_{ij} represent the current value of the pixel (either **0** or **1**).
- c_{ij} be the new value assigned to the pixel based on a_{ij} and b_{ij}.

Possible Input Cases Since each pixel has two possible states (**0 or 1**) and its neighborhood also has two possible states, the possible input combinations are:

$$(a_{ij}, b_{ij}) = (0, 0), (0, 1), (1, 0), (1, 1)$$

For each input pair, the output c_{ij} is assigned a binary value (either **0** or **1**). Since there are four input cases, the total number of possible transformation rules is:

$$2^4 = 16$$

Each combination defines a different **iterative modification algorithm**.

These algorithms are widely used in **image processing**, particularly in **noise removal, shape refinement, and skeletonization** of binary images.

Conditions for Growing an Image : Image growth refers to expanding object regions by modifying pixel values iteratively. For a pixel to be added to the object (change from 0 to 1), the following conditions must hold:

- $a_{ij} = 1$ (the neighborhood satisfies the condition for growth).
- $b_{ij} = 0$ (the current pixel is part of the background).
- $c_{ij} = 1$ (the pixel is transformed into part of the object).

Thus, the transformation rule for growing an image can be expressed as:

$$(a_{ij}, b_{ij}) = (1, 0) \Rightarrow c_{ij} = 1$$

This means that whenever a background pixel is adjacent to an object pixel (as determined by the neighborhood S), it gets converted into an object pixel in the next iteration.

MATLAB Code to grow object in Binary image

```
clc; clear; close all;

% Load binary image (User can uncomment the next line to load an
   actual image)
[filename, pathname] = uigetfile({'*.jpg;*.png;*.bmp;*.tif', '
   Image Files (*.jpg, *.png, *.bmp, *.tif)'}, 'Select a Binary
   Image');

% Check if the user selected a file
if isequal(filename, 0)
    disp('No file selected. Exiting...');
    return;
end

% Step 2: Read the selected image
imagePath = fullfile(pathname, filename);
binary_image = imread(imagePath);


% Define number of growth iterations
num_iterations = 50;

% Display original image
figure;
subplot(1, 2, 1);
imshow(binary_image, 'InitialMagnification', 'fit');
title('Original Image');

% Get image size
```

```
[rows, cols] = size(binary_image);

% Perform iterative modification (growing image)
for iter = 1:num_iterations
    new_image = binary_image;  % Copy of original image

    % Iterate over each pixel (skip borders to avoid indexing issues)
    for i = 2:rows-1
        for j = 2:cols-1
            % If current pixel is 0, check if it has any 1 in the neighborhood
            if binary_image(i, j) == 0
                % Check 8-neighborhood manually
                if binary_image(i-1, j) == 1 || binary_image(i+1, j) == 1 || ...
                   binary_image(i, j-1) == 1 || binary_image(i, j+1) == 1 || ...
                   binary_image(i-1, j-1) == 1 || binary_image(i-1, j+1) == 1 || ...
                   binary_image(i+1, j-1) == 1 || binary_image(i+1, j+1) == 1
                    new_image(i, j) = 1;  % Expand the white region
                end
            end
        end
    end

    % Update the binary image for the next iteration
    binary_image = new_image;
end

% Display final grown image after all iterations
subplot(1, 2, 2);
imshow(binary_image, 'InitialMagnification', 'fit');
title(['Final Image (After ' num2str(num_iterations) ' Iterations)']);
```

Output

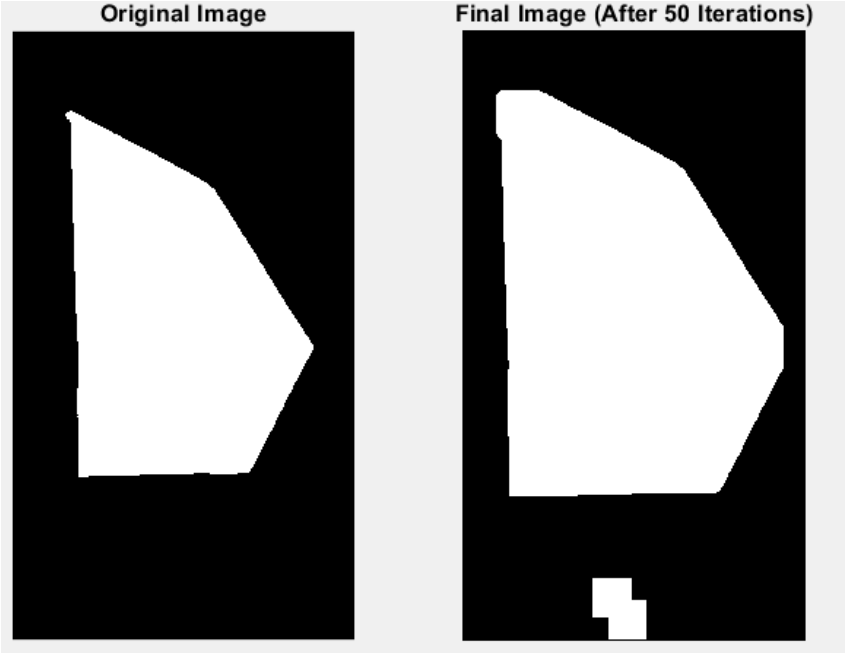

Figure 6: Apply Growing property to binary image

Conditions for Thinning an Image : Thinning selectively removes pixels from object boundaries while ensuring connectivity is maintained. For a pixel to be removed (change from **1** to **0**), the following conditions must hold:

- $a_{ij} = 1$ (the neighborhood satisfies the thinning condition).
- $b_{ij} = 1$ (the current pixel is part of the object).
- $c_{ij} = 0$ (the pixel is transformed into the background).

Thus, the transformation rule for thinning can be expressed as:

$$(a_{ij}, b_{ij}) = (1, 1) \Rightarrow c_{ij} = 0$$

This means that whenever an object pixel satisfies the predefined thinning condition, it is removed in the next iteration.

MATLAB Code for thinning object in Binary image

```
clc;
clear;
close all;

% Load binary image (User can uncomment the next line to load an
    actual image)
[filename, pathname] = uigetfile({'*.jpg;*.png;*.bmp;*.tif', '
    Image␣Files␣(*.jpg,␣*.png,␣*.bmp,␣*.tif)'}, 'Select␣a␣Binary␣
    Image');
```

```
7
8  % Check if the user selected a file
9  if isequal(filename, 0)
10     disp('No file selected. Exiting...');
11     return;
12 end
13
14 % Step 2: Read the selected image
15 imagePath = fullfile(pathname, filename);
16 binary_image = imread(imagePath);
17
18 % Define number of thinning iterations
19 num_iterations = 250;
20
21 % Display original image
22 figure;
23 subplot(1, 2, 1);
24 imshow(binary_image, 'InitialMagnification', 'fit');
25 title('Original Image');
26
27 % Get image size
28 [rows, cols] = size(binary_image);
29
30 % Perform iterative thinning
31 for iter = 1:num_iterations
32     new_image = binary_image;  % Copy of original image
33
34     % Iterate over each pixel (skip borders to avoid indexing
          issues)
35     for i = 2:rows-1
36         for j = 2:cols-1
37             % If current pixel is 1, check its neighborhood
38             if binary_image(i, j) == 1
39                 % Count number of 1's in 8-neighborhood
40                 neighbor_count = binary_image(i-1, j) +
                      binary_image(i+1, j) + ...
41                                  binary_image(i, j-1) +
                                       binary_image(i, j+1) + ...
42                                  binary_image(i-1, j-1) +
                                       binary_image(i-1, j+1) + ...
43                                  binary_image(i+1, j-1) +
                                       binary_image(i+1, j+1);
44
45                 % If a 1-pixel has more than 2 and less than 6
                      neighbors, remove it
46                 if neighbor_count >= 2 && neighbor_count <= 6
47                     new_image(i, j) = 0;  % Thin the image
48                 end
49             end
50         end
51     end
```

```
52
53      % Update the binary image for the next iteration
54      binary_image = new_image;
55  end
56
57  % Display final thinned image after all iterations
58  subplot(1, 2, 2);
59  imshow(binary_image, 'InitialMagnification', 'fit');
60  title(['Final␣Image␣(After␣' num2str(num_iterations) '␣Iterations
    )']);
```

Output

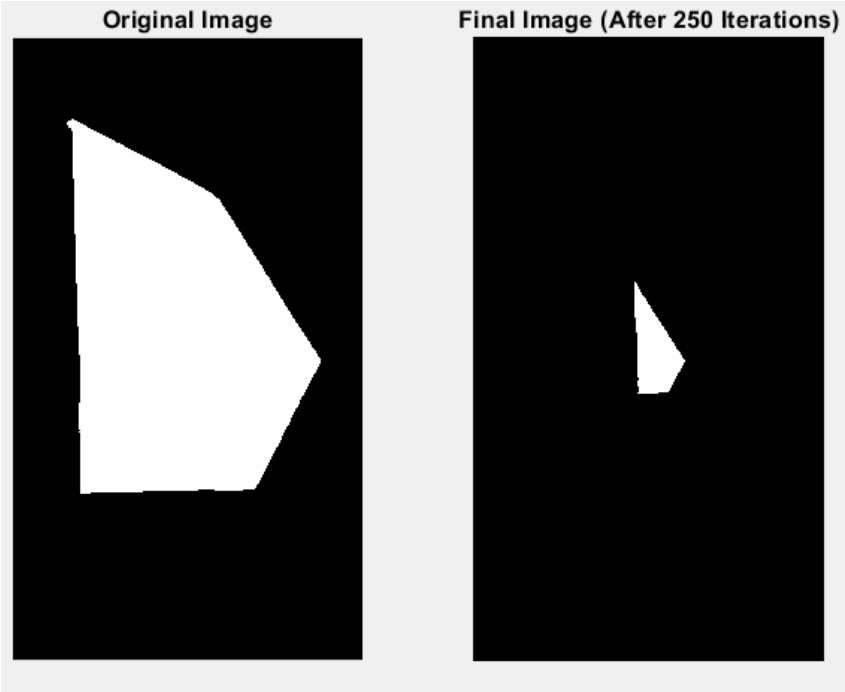

Figure 7: Apply Thinning property to binary image

Camera and Imaging: Module 5

Gaussian Smoothing

Gaussian smoothing is a widely used technique in image processing to reduce noise and blur an image while preserving important structures. It is implemented using a Gaussian filter, which is based on the Gaussian function.

Mathematical Definition The fuzzy filter can be formalized using the Gaussian function. The Gaussian function in the discrete domain is defined as:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where:

- σ is the standard deviation of the Gaussian distribution.
- x, y represent the spatial coordinates in the image.
- $\frac{1}{2\pi\sigma^2}$ ensures that the filter is normalized so that the total sum of the weights remains **1**.

Effect of Standard Deviation (σ) The larger the value of σ, the broader the Gaussian filter becomes. This means:

- A small σ results in a narrow Gaussian, causing minimal blurring.
- A large σ results in a wide Gaussian, leading to stronger blurring.

Choosing the Filter Size An important question arises: what should be the size of the filter? Since the Gaussian function theoretically extends to infinity, we need a practical way to select a finite filter size.

A good rule of thumb is:

$$K \approx 2\sigma$$

where $K \times K$ represents the size of the Gaussian filter. This choice ensures that most of the energy of the Gaussian function is captured within the filter.

Applications of Gaussian Smoothing Gaussian smoothing is extensively used in:

- **Preprocessing for Edge Detection**: Reducing noise before applying edge detection algorithms like **Canny Edge Detector**.
- **Image Denoising**: Removing high-frequency noise while preserving structures.
- **Blurring Effects**: Used in graphics and vision applications to create soft blur effects.

Gaussian smoothing provides a controlled way of removing noise while ensuring smooth transitions in an image.

MATLAB Code for Gaussian Smoothing in Greyscale image

```
clc; clear; close all;

% Step 1: Select an Image from the User
[filename, pathname] = uigetfile({'*.jpg;*.png;*.bmp;*.tif', 'Image Files'});
if filename == 0
    disp('No image selected. Exiting...');
    return;
end

% Step 2: Read and Convert the Image to Grayscale
img = imread(fullfile(pathname, filename));
if size(img, 3) == 3
    img = rgb2gray(img);
end
img = double(img); % Convert to double for calculations
[M, N] = size(img);

% Step 3: Compute the Fourier Transform and Shift Zero Frequency to Center
F = fft2(img);
F_shifted = fftshift(F);

% Step 4: Create a Gaussian Low-Pass Filter in Frequency Domain
D0 = 50; % Cutoff frequency (adjustable)
[X, Y] = meshgrid(1:N, 1:M);
center_x = ceil(N/2);
center_y = ceil(M/2);
D = sqrt((X - center_x).^2 + (Y - center_y).^2);

% Gaussian function: H = e^(-D^2 / (2*D0^2))
H = exp(-(D.^2) / (2 * D0^2));

% Step 5: Apply the Gaussian Filter in Frequency Domain
F_filtered = F_shifted .* H;

% Step 6: Compute the Inverse Fourier Transform
F_inv_shifted = ifftshift(F_filtered);
img_filtered = abs(ifft2(F_inv_shifted));

% Step 7: Normalize the Image for Display
img_filtered = uint8(img_filtered * 255 / max(img_filtered(:)));

% Step 8: Display the Results
figure;
subplot(1, 3, 1), imshow(uint8(img)), title('Original Image');
subplot(1, 3, 2), imshow(log(1 + abs(F_shifted)), []), title('Fourier Transform');
```

```
46 subplot(1, 3, 3), imshow(img_filtered), title('Gaussian␣Smoothed␣
       Image');
47
48 disp('Gaussian␣smoothing␣applied␣successfully.');
```

Output

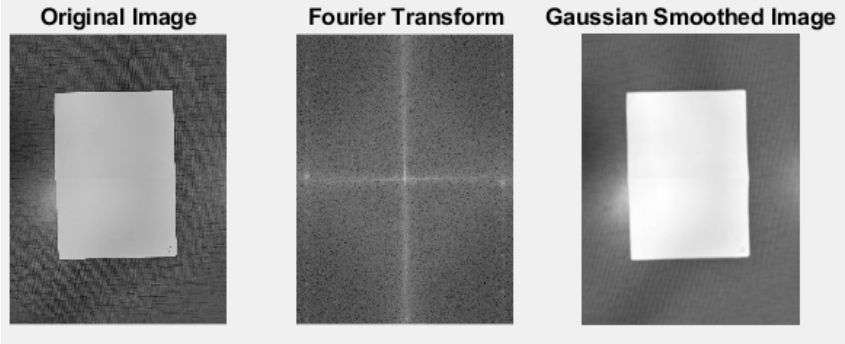

Figure 8: Gaussian Smoothing

Impulse Filter

Impulse filters are designed to handle **impulse noise**, which appears as sudden, sharp intensity variations in an image, commonly caused by transmission errors, sensor defects, or environmental factors. Impulse noise is often classified into two types:

- **Salt-and-Pepper Noise**: This occurs when random pixels are set to either **minimum (black, 0)** or **maximum (white, 255)** intensity values.
- **Random Impulse Noise**: In this case, noisy pixels take on arbitrary intensity values rather than just black or white.

Delta Function in Impulse Filtering The **delta function**, often referred to as the **impulse function**, plays a crucial role in impulse noise filtering. It is defined mathematically as:

$$\delta(x) = \begin{cases} 1, & \text{if } x = 0 \\ 0, & \text{otherwise} \end{cases}$$

This function represents an idealized **point source** or an **impulse**, meaning it has value **1 at a single point** and **0 elsewhere**. In image processing, the delta function is used to model **impulse noise**, where certain pixels exhibit extreme intensity values while the surrounding pixels remain unaffected.

Role of the Delta Function in Filtering Impulse filters are designed to **suppress the effect of impulse noise**, which is often modeled as:

$$g(i,j) = f(i,j) + \eta(i,j)$$

where:

- $g(i,j)$ is the observed noisy image.
- $f(i,j)$ is the original image.
- $\eta(i,j)$ represents the impulse noise, which follows the delta function behavior, affecting only a few random pixels.

Since impulse noise is **localized** (affecting only specific pixels), the delta function helps analyze how an image responds to sudden changes in pixel intensity. Many filtering techniques, such as the **median filter** and **adaptive impulse filters**, are designed to detect and remove these delta-function-like noise components.

Applications of Impulse Filters Impulse filters are widely used in:

- **Medical Imaging**: Removing artifacts from X-ray and MRI images.
- **Satellite Image Processing**: Eliminating sensor noise from remote sensing images.
- **Computer Vision**: Preprocessing images before applying edge detection and segmentation algorithms.

By effectively reducing impulse noise, these filters help enhance image clarity while preserving important details.

MATLAB Code for Impulse filter in Greyscale image

```
% Read a grayscale image from the user
[file, path] = uigetfile({'*.jpg;*.png;*.bmp', 'Image Files (*.jpg, *.png, *.bmp)'});
if isequal(file, 0)
    disp('User selected Cancel');
    return;
end
img = imread(fullfile(path, file));

% Convert to grayscale if not already
dimensions = size(img);
if numel(dimensions) == 3
    img = rgb2gray(img);
end

% Define the impulse (delta) filter
impulse_filter = [0 0 0; 0 1 0; 0 0 0];

% Apply the convolution operation
filtered_img = imfilter(img, impulse_filter, 'conv');

% Display the original and filtered images
figure;
subplot(1,2,1), imshow(img), title('Original Grayscale Image');
subplot(1,2,2), imshow(filtered_img), title('Filtered Image (Impulse Filter)');
```

Output

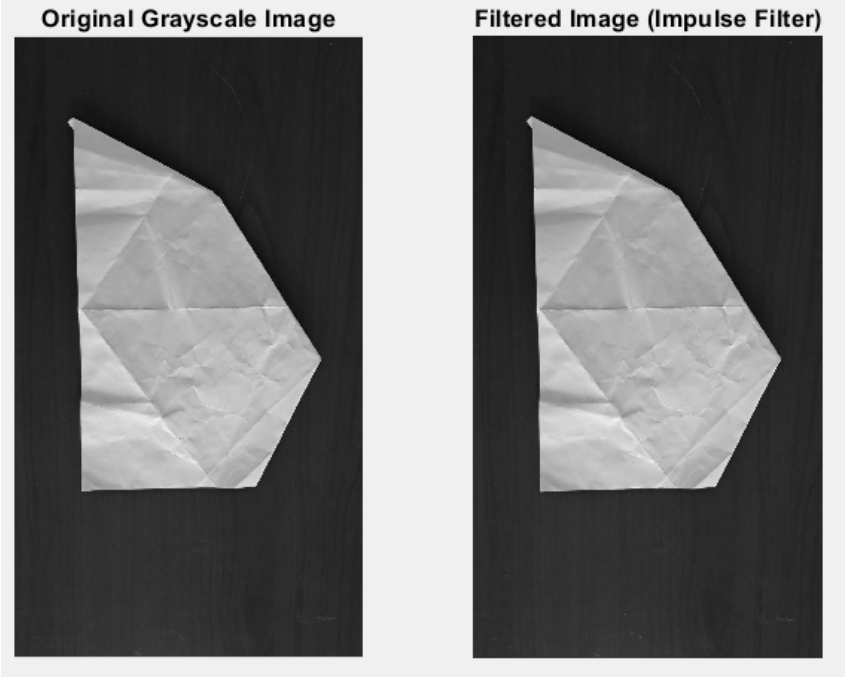

Figure 9: Apply Impulse filter to Greyscale image

Median Filter

Median filtering is a widely used technique for noise removal in image processing. Unlike linear filters, which compute a weighted sum of pixel values, median filtering follows a **non-linear approach** to preserve edges while reducing noise.

How Median Filtering Works For each pixel in the image, the median filter performs the following steps:

- Collect all the intensity values within a **KxK window** centered at the pixel, including its own value.
- Sort these intensity values in **ascending order**.
- Select the **middle value** from the sorted list, which is the median.
- Assign this median value as the new intensity for the pixel.

Since the median represents the **middle** value, it is less affected by extreme noise pixels, making it highly effective in removing **impulse noise** such as **salt-and-pepper noise** while preserving important image structures.

Advantages of the Median Filter

- **Preserves Edges**: Unlike averaging filters, the median filter does not blur sharp edges.

- **Effective for Impulse Noise**: It removes isolated extreme values without affecting normal pixels.
- **Simple and Efficient**: Can be implemented efficiently, especially with optimized sorting techniques.

Applications of Median Filtering

- **Medical Imaging**: Reducing noise in X-ray, CT, and MRI images.
- **Digital Photography**: Removing speckle and salt-and-pepper noise from photographs.
- **Remote Sensing**: Enhancing satellite and aerial imagery.

MATLAB Code for Median filter in Greyscale image

```
% MATLAB program to apply median filtering on a grayscale image

% Read the grayscale image from the user
[file, path] = uigetfile({'*.jpg;*.png;*.bmp','Image Files (*.jpg, *.png, *.bmp)'});
if isequal(file,0)
    disp('User selected Cancel');
    return;
end
img = imread(fullfile(path, file));

% Convert to grayscale if the image is not already grayscale
if size(img, 3) == 3
    img = rgb2gray(img);
end

% Display original image
figure;
subplot(1,2,1);
imshow(img);
title('Original Image');

% Apply median filtering with a 10x10 mask
filtered_img = medfilt2(img, [3 3]);

% Display filtered image
subplot(1,2,2);
imshow(filtered_img);
title('Median Filtered Image (10x10)');
```

Output :

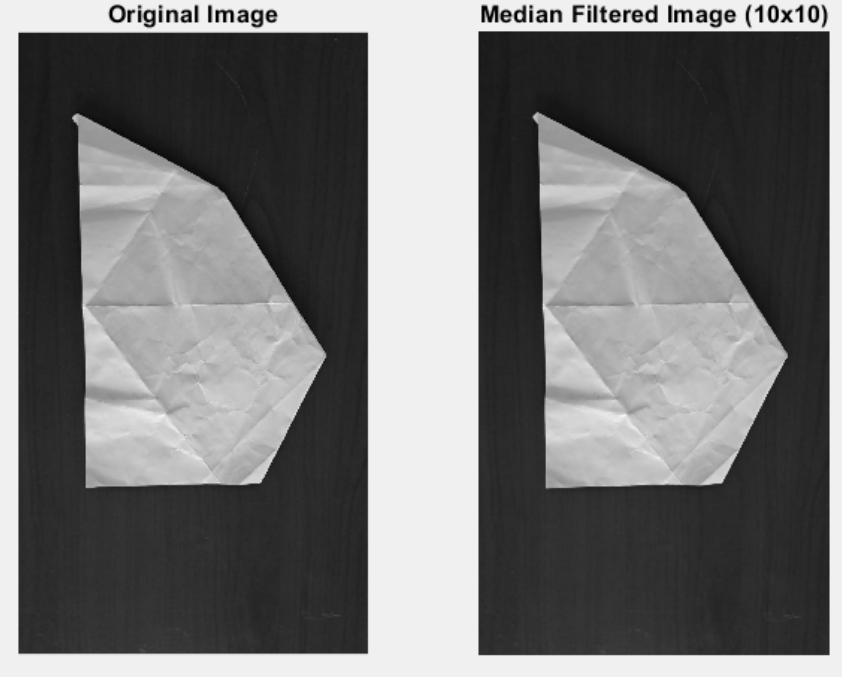

Figure 10: Apply Median filter to Greyscale image

Single Value Decomposition (SVD) Method

Overview
The **Single Value Decomposition (SVD)** is a powerful matrix factorization technique widely used in image processing, signal processing, and machine learning. It decomposes a matrix into three distinct matrices, capturing essential information while reducing noise and redundancy.

Mathematical Representation of SVD : Given a matrix A of size $m \times n$, the SVD decomposition is defined as:

$$A = U\Sigma V^T$$

where:

- U is an $m \times m$ orthogonal matrix, whose columns are the **left singular vectors** of A.
- Σ is an $m \times n$ diagonal matrix containing **singular values** of A, arranged in descending order.
- V^T is the transpose of an $n \times n$ orthogonal matrix, where columns of V are the **right singular vectors** of A.

Steps in Applying SVD :

1. **Matrix Decomposition:** Decompose matrix A into U, Σ, and V^T.

2. **Retain Dominant Singular Values:** The values in Σ represent the importance of each component. Keeping only the largest singular values approximates the original matrix with reduced noise.

3. **Reconstruct the Matrix:** Approximate A using a subset of singular values:

$$A_k = U_k \Sigma_k V_k^T$$

where k is the number of retained singular values.

Applications of SVD in Image Processing :

- **Image Compression:** Reduces storage requirements by approximating the image using fewer singular values.
- **Noise Reduction:** Eliminates noise by discarding small singular values associated with insignificant details.
- **Pattern Recognition:** Analyzes features in images for recognition and classification.

Benefits of SVD :

- **Optimal Approximation:** Provides the best low-rank approximation of a matrix.
- **Noise Resilience:** Helps separate noise from useful data.
- **Dimensionality Reduction:** Efficiently represents data in lower dimensions.

The **Single Value Decomposition** technique offers a robust approach for handling large datasets in image and signal processing, making it essential in fields like computer vision and machine learning.

MATLAB Code for Single Value Decomposition Method

```
% Single Value Decomposition without using built-in svd function
clc;
clear;
close all;

% Select an image from the user
[file, path] = uigetfile({'*.jpg;*.jpeg;*.png;*.bmp', 'Image␣
   Files'}, 'Select␣a␣Grayscale␣Image');
if isequal(file, 0)
    disp('User␣canceled␣image␣selection');
    return;
end

% Read and convert image to grayscale
imagePath = fullfile(path, file);
img = imread(imagePath);
if size(img, 3) == 3
```

```
    img = rgb2gray(img); % Convert to grayscale if RGB
end
img = double(img);

% Get dimensions
[m, n] = size(img);

% Step 1: Compute A * A'
AAT = img * img';

% Step 2: Compute eigenvalues and eigenvectors of A * A'
[eigVec_U, eigVal_U] = eig(AAT);

% Sort eigenvalues and eigenvectors
[eigVal_U, sortIdx] = sort(diag(eigVal_U), 'descend');
eigVec_U = eigVec_U(:, sortIdx);

% Step 3: Calculate singular values
singularValues = sqrt(eigVal_U);

% Step 4: Compute V from A' * A
ATA = img' * img;
[eigVec_V, eigVal_V] = eig(ATA);

% Sort eigenvalues and eigenvectors
[eigVal_V, sortIdx] = sort(diag(eigVal_V), 'descend');
eigVec_V = eigVec_V(:, sortIdx);

% Step 5: Construct Sigma matrix (Handle rectangular images)
Sigma = zeros(m, n); % m x n matrix

% Adjust assignment for rectangular images
for i = 1:min(m, n)
    Sigma(i, i) = singularValues(i);
end

% Step 6: Compute U and V matrices
U = eigVec_U;
V = eigVec_V;

% Step 7: Reconstruct the image
reconstructedImg = U * Sigma * V';

% Display Original and Reconstructed Images
figure;
subplot(1, 2, 1);
imshow(uint8(img));
title('Original Image');

subplot(1, 2, 2);
imshow(uint8(reconstructedImg));
```

```
68 title('Reconstructed Image using SVD');
69
70 disp('SVD Decomposition Complete');
```

Output :

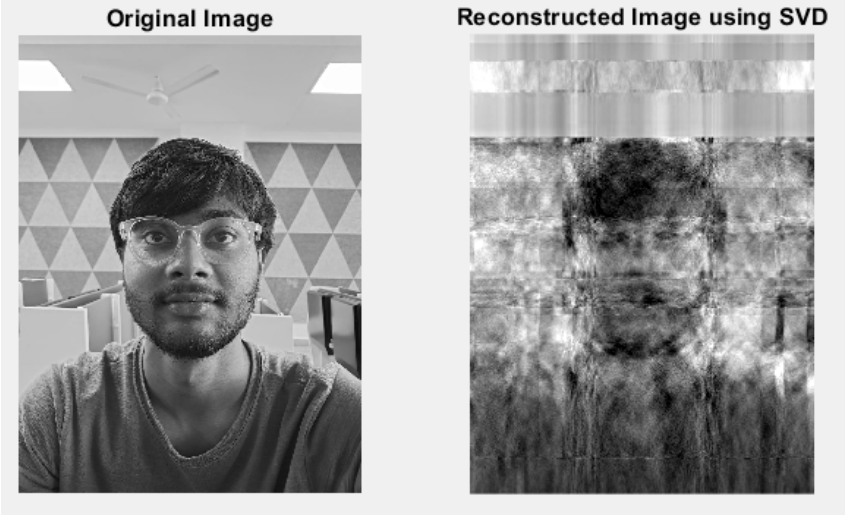

Figure 11: Single Valued Decomposition image

Camera and Imaging: Module 6

Low-Pass Filter

Overview

A low-pass filter smooths an image by removing high-frequency details, making it useful for **blurring** and **noise reduction**.

Fourier Transform and Filtering : In the frequency domain, **low frequencies** represent smooth intensity variations, while **high frequencies** correspond to sharp edges. Low-pass filtering removes the high-frequency components by cutting out the outer regions of the Fourier transform.

Effect on Images : Applying an **inverse Fourier transform** after filtering results in a **blurred** image. Stronger low-pass filtering increases blurring but may introduce **block artifacts** due to harsh frequency cut-offs.

Applications :

- **Noise Reduction** – Suppresses high-frequency noise.
- **Image Blurring** – Used in preprocessing for vision tasks.
- **Medical Imaging** – Smooths MRI and CT scans.

MATLAB Code for Low Pass Filter

```
clc; clear; close all;

% Select an image file from user
[filename, pathname] = uigetfile({'*.jpg;*.jpeg;*.png;*.bmp;*.tif
    ','Image Files'});
if filename == 0
    disp('No image selected. Exiting...');
    return;
end

% Read the image and convert to grayscale if necessary
img = imread(fullfile(pathname, filename));
if size(img, 3) == 3
    img = rgb2gray(img);
end

img = double(img); % Convert to double for calculations
[M, N] = size(img);

% Compute the Fourier Transform manually
F = fft2(img);
F_shifted = fftshift(F); % Centering the Fourier spectrum

% Define a low-pass filter (circular mask)
```

```
D0 = 50; % Cutoff frequency (adjustable)
[X, Y] = meshgrid(1:N, 1:M);
center_x = ceil(N/2);
center_y = ceil(M/2);
D = sqrt((X - center_x).^2 + (Y - center_y).^2);

% Create low-pass filter mask
H = double(D <= D0);

% Apply the filter
F_filtered = F_shifted .* H;

% Compute the inverse Fourier Transform manually
F_inv_shifted = ifftshift(F_filtered); % Shift back
img_filtered = abs(ifft2(F_inv_shifted)); % Compute inverse FFT

% Normalize the image for proper display
img_filtered = uint8(img_filtered * 255 / max(img_filtered(:)));

% Save the modified image
savePath = fullfile(pathname, 'rishit_lowscale.jpeg');
imwrite(img_filtered, savePath);
disp(['Modified image saved to: ', savePath]);

% Display results
figure;
subplot(1, 3, 1), imshow(uint8(img)), title('Original Image');
subplot(1, 3, 2), imshow(log(1 + abs(F_shifted)), []), title('
    Fourier Transform');
subplot(1, 3, 3), imshow(img_filtered), title('Low-Pass Filtered
    Image');

disp('Low-pass filtering applied successfully.');
```

Output :

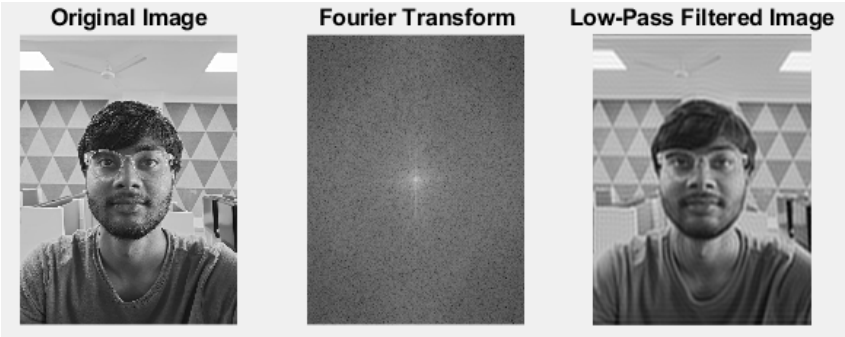

Figure 12: Apply Low Pass filter to Greyscale image

High-Pass Filter

Overview

A high-pass filter enhances **sharp details** by preserving high-frequency components while removing low-frequency background information. It is useful for **edge detection** and **feature enhancement**.

Fourier Transform and Filtering : In the frequency domain, **low frequencies** represent smooth variations, while **high frequencies** correspond to sharp transitions. High-pass filtering removes the low-frequency components by eliminating the central region of the Fourier transform.

Effect on Images : Applying an **inverse Fourier transform** after filtering results in an image with enhanced edges and fine details, making textures and object boundaries more prominent. Excessive filtering may introduce **noise amplification**.

Applications :

- **Edge Detection** – Highlights object boundaries.
- **Feature Enhancement** – Emphasizes textures and fine details.
- **Medical Imaging** – Enhances structures in X-rays and MRIs.

MATLAB Code for High Pass Filter

```
clc; clear; close all;

% Select an image file from user
[filename, pathname] = uigetfile({'*.jpg;*.jpeg;*.png;*.bmp;*.
    tif', 'Image Files'});
if filename == 0
    disp('No image selected. Exiting...');
    return;
end

% Read the image and convert to grayscale if necessary
img = imread(fullfile(pathname, filename));
if size(img, 3) == 3
    img = rgb2gray(img);
end

img = double(img); % Convert to double for calculations
[M, N] = size(img);

% Compute the Fourier Transform manually
F = fft2(img);
F_shifted = fftshift(F); % Centering the Fourier spectrum

% Define a high-pass filter (circular mask)
D0 = 50; % Cutoff frequency (adjustable)
```

```
[X, Y] = meshgrid(1:N, 1:M);
center_x = ceil(N/2);
center_y = ceil(M/2);
D = sqrt((X - center_x).^2 + (Y - center_y).^2);

% Create high-pass filter mask
H = double(D > D0);

% Apply the filter
F_filtered = F_shifted .* H;

% Compute the inverse Fourier Transform manually
F_inv_shifted = ifftshift(F_filtered); % Shift back
img_filtered = abs(ifft2(F_inv_shifted)); % Compute inverse FFT

% Normalize the image for proper display
img_filtered = uint8(img_filtered * 255 / max(img_filtered(:)));

% Save the modified image
savePath = fullfile(pathname, 'somesh_highscale.jpeg');
imwrite(img_filtered, savePath);
disp(['Modified image saved to: ', savePath]);

% Display results
figure;
subplot(1, 3, 1), imshow(img), title('Original Image');
subplot(1, 3, 2), imshow(log(1 + abs(F_shifted)), []), title('
   Fourier Transform');
subplot(1, 3, 3), imshow(img_filtered), title('High-Pass Filtered
   Image');

disp('High-pass filtering applied successfully.');
```

Output :

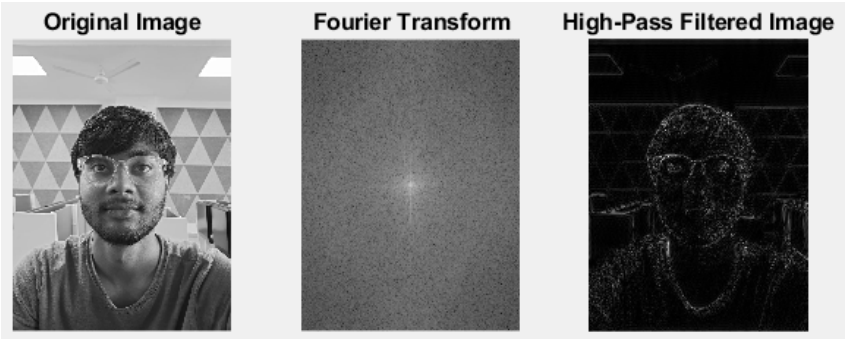

Figure 13: Apply High Pass filter to Greyscale image

Hybrid Image

Overview

A hybrid image is created by combining a **low-pass filtered** image with a **high-pass filtered** image. This results in an image that appears different depending on the viewing distance.

Formation of Hybrid Images :

- A **low-pass filter** removes high frequencies, leaving a **blurred** image with only coarse details.
- A **high-pass filter** removes low frequencies, preserving only fine details.
- Averaging these two images creates a hybrid image that exhibits **distance-dependent perception**.

Perception Mechanism :

- **Close-up View**: The human eye detects fine details (high frequencies), making the high-pass filtered image dominant.
- **Distant View**: High frequencies fade due to the **point-spread function** of the eye's lens, revealing the low-pass filtered image.

Applications :

- **Art and Optical Illusions** – Creating images with multiple interpretations.
- **Visual Perception Studies** – Understanding how the human eye processes frequencies.
- **Image Compression** – Efficiently encoding images for different viewing conditions.

MATLAB Code for Hybrid Image

```
% Hybrid Image Creation in MATLAB

% Read Low-Pass and High-Pass Images
low_pass_img = im2double(imread('rishit_lowscale.jpeg')); % Replace with user input
high_pass_img = im2double(imread('somesh_highscale.jpeg')); % Replace with user input

% Ensure images are grayscale and of same size
if size(low_pass_img,3) == 3
    low_pass_img = rgb2gray(low_pass_img);
end
if size(high_pass_img,3) == 3
    high_pass_img = rgb2gray(high_pass_img);
end
if size(low_pass_img) ~= size(high_pass_img)
```

```
    error('Images must be of the same size');
end

% Create Hybrid Image by Averaging
hybrid_image = (low_pass_img + high_pass_img) / 2;

% Display the Images
figure;
subplot(1,3,1); imshow(low_pass_img); title('Low-Pass Image');
subplot(1,3,2); imshow(high_pass_img); title('High-Pass Image');
subplot(1,3,3); imshow(hybrid_image); title('Hybrid Image');
```

Output :

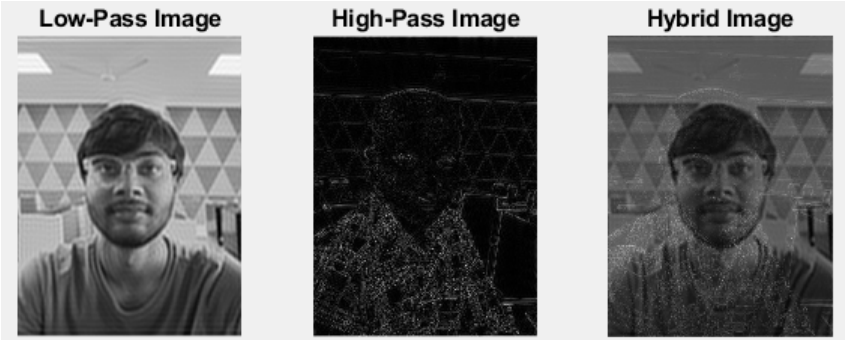

Figure 14: Hybrid Image

Wiener Deconvolution

Overview

Wiener deconvolution is a technique used to **restore blurred images** while suppressing noise. It improves upon simple deconvolution by incorporating a noise-signal ratio (NSR) to prevent excessive noise amplification.

Mathematical Formulation The Wiener deconvolution formula is given by:

$$F_{\text{restored}}(u,v) = \frac{F_{\text{observed}}(u,v)}{H(u,v)} \cdot \frac{|H(u,v)|^2}{|H(u,v)|^2 + \text{NSR}}$$

where:

- $F_{\text{restored}}(u,v)$ is the recovered image in the frequency domain.
- $F_{\text{observed}}(u,v)$ is the blurred image in the frequency domain.
- $H(u,v)$ represents the blur kernel (point spread function) in the frequency domain.
- **NSR** is the **noise-to-signal ratio**, defined as the ratio of noise power to signal power.

Effect of Noise-to-Signal Ratio (NSR) :

- When **NSR is high** (high noise), the denominator increases, reducing amplification of noise.
- When **H(u,v) is small** (motion blur affects certain frequencies), Wiener deconvolution prevents excessive boosting of those frequencies.

MATLAB Code for Wienar Deconvolution Method

```
clc; clear; close all;

% Step 1: Load a Motion Blurred Image
[filename, pathname] = uigetfile({'*.jpg;*.jpeg;*.png;*.bmp;*.tif', 'Image Files'});
if filename == 0
    disp('No image selected. Exiting...');
    return;
end

% Read and convert image to grayscale
blurred_img = imread(fullfile(pathname, filename));
if size(blurred_img, 3) == 3
    blurred_img = rgb2gray(blurred_img);
end
blurred_img = double(blurred_img); % Convert to double
[M, N] = size(blurred_img);

% Step 2: Generate Motion Blur Kernel (PSF - Point Spread Function)
L = 20;  % Motion length (adjust for stronger/weaker blur)
theta = 30; % Motion direction in degrees

% Create the Motion Blur Kernel (Manually Constructed in Frequency Domain)
[X, Y] = meshgrid(1:N, 1:M);
center_x = ceil(N/2);
center_y = ceil(M/2);
D = (L * (cosd(theta) * (X - center_x) + sind(theta) * (Y - center_y))) / N;

% Manually Compute sinc(D) = sin(pi * D) / (pi * D)
sinc_D = ones(size(D)); % Initialize to 1 (for D = 0)
nonzero_indices = (D ~= 0);
sinc_D(nonzero_indices) = sin(pi * D(nonzero_indices)) ./ (pi * D(nonzero_indices));

% Create the blur kernel
H = sinc_D .^ 2;  % Squaring to approximate motion blur PSF

% Step 3: Compute Fourier Transform of the Blurred Image
F_blurred = fft2(blurred_img);
H_shifted = fftshift(H); % Shift PSF to center
```

```
39
40 % Step 4: Define NSR (Noise-to-Signal Ratio)
41 NSR = 0.01; % Experiment with values (lower = more sharpening,
       higher = noise suppression)
42
43 % Step 5: Wiener Deconvolution (Applying the Filter)
44 % W = (1 / H) * (|H|^2 / (|H|^2 + NSR))
45 H_abs_sq = abs(H_shifted).^2;
46 Wiener_Filter = (1 ./ H_shifted) .* (H_abs_sq ./ (H_abs_sq + NSR)
       );
47
48 % Apply the Wiener filter in the frequency domain
49 F_recovered = F_blurred .* Wiener_Filter;
50
51 % Step 6: Compute Inverse Fourier Transform to obtain the
       restored image
52 recovered_img = real(ifft2(F_recovered));
53
54 % Step 7: Normalize and Convert to uint8 for Display
55 recovered_img = recovered_img - min(recovered_img(:)); % Shift to
        non-negative values
56 recovered_img = recovered_img / max(recovered_img(:)) * 255; %
       Scale to 0-255
57 recovered_img = uint8(recovered_img);
58
59 % Step 8: Display the Results
60 figure;
61 subplot(1, 3, 1), imshow(uint8(blurred_img)), title('Motion␣
       Blurred␣Image');
62 % subplot(1, 3, 2), imshow(log(1 + abs(fftshift(F_blurred))), []
       , title('Fourier Transform of Blurred Image');
63 subplot(1, 3, 3), imshow(recovered_img), title('Restored␣Image␣(
       Wiener␣Deconvolution)');
64
65 disp('Wiener␣deconvolution␣applied␣successfully.');
```

Output :

Figure 15: Apply Weinar Deconvolution to restore motion blur image

Features and Boundaries: Module 1 & Module 2

Roberts Filter for Edge Detection

Overview

The **Roberts filter** is one of the simplest and earliest edge detection operators. It detects edges by computing the gradient of an image using discrete differentiation. This method highlights regions of rapid intensity change, which correspond to edges.

Mathematical Formulation : The Roberts filter uses two 2×2 convolution kernels:

$$G_x = \begin{bmatrix} +1 & 0 \\ 0 & -1 \end{bmatrix}, \quad G_y = \begin{bmatrix} 0 & +1 \\ -1 & 0 \end{bmatrix}$$

where:

- G_x detects edges along the diagonal from **top-left to bottom-right**.
- G_y detects edges along the diagonal from **top-right to bottom-left**.

Gradient Magnitude Calculation : The edge strength is determined using:

$$G = \sqrt{(G_x * I)^2 + (G_y * I)^2}$$

where I is the input image, and $*$ denotes convolution. Alternatively, an approximation can be used:

$$G \approx |G_x * I| + |G_y * I|$$

Properties and Characteristics

- **Simple and efficient** – Uses small kernels, making it computationally fast.
- **Sensitive to noise** – Since it uses small gradients, it may detect noise as edges.
- **Detects diagonal edges** effectively.

Applications

- **Image Processing** – Basic edge detection in grayscale images.
- **Medical Imaging** – Enhancing X-ray and MRI edges.
- **Barcode and Text Recognition** – Identifying character outlines in scanned documents.

MATLAB Code for Roberts filter in Greyscale

```
% MATLAB Program to compute gradient magnitude and orientation
   using Roberts filter (without built-in functions)

% Select an image from the user
[file, path] = uigetfile({'*.jpg;*.jpeg;*.png;*.bmp;*.tif', '
   Image Files'});
```

```
if isequal(file,0)
    disp('User canceled the operation.');
    return;
end
imgPath = fullfile(path, file);
img = imread(imgPath);

% Convert to grayscale manually if needed
if size(img, 3) == 3
    grayImg = 0.2989 * img(:,:,1) + 0.5870 * img(:,:,2) + 0.1140 * img(:,:,3);
else
    grayImg = img;
end

grayImg = double(grayImg);

% Define Roberts operator manually
Gx_kernel = [1 0; 0 -1];
Gy_kernel = [0 1; -1 0];

% Get image dimensions
[row, col] = size(grayImg);
Gx = zeros(row, col);
Gy = zeros(row, col);

% Apply Roberts filter manually (without imfilter)
for i = 1:row-1
    for j = 1:col-1
        Gx(i, j) = sum(sum(Gx_kernel .* grayImg(i:i+1, j:j+1)));
        Gy(i, j) = sum(sum(Gy_kernel .* grayImg(i:i+1, j:j+1)));
    end
end

% Compute gradient magnitude
magnitude = sqrt(Gx.^2 + Gy.^2);

% Compute gradient orientation (in degrees)
orientation = atan2d(Gy, Gx);

% Normalize for display
magnitude = magnitude / max(magnitude(:));
orientation = (orientation - min(orientation(:))) / (max(orientation(:)) - min(orientation(:)));

% Display results
figure;
subplot(1,3,1); imshow(uint8(grayImg)); title('Grayscale Image');
subplot(1,3,2); imshow(magnitude); title('Gradient Magnitude');
subplot(1,3,3); imshow(orientation); title('Gradient Orientation');
```

Output :

Figure 16: Apply Roberts filter

Prewitt Filter for Edge Detection

Overview
The **Prewitt filter** is an edge detection operator that computes the gradient of an image intensity function. It is similar to the Sobel filter but uses a simpler averaging method, making it computationally efficient.

Mathematical Formulation : The Prewitt filter uses two 3×3 convolution kernels to approximate the gradient in the x- and y-directions:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix}, \quad G_y = \begin{bmatrix} +1 & +1 & +1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

where:

- G_x detects edges in the **horizontal** direction.
- G_y detects edges in the **vertical** direction.

Gradient Magnitude Calculation The strength of the edges is computed as:

$$G = \sqrt{(G_x * I)^2 + (G_y * I)^2}$$

or an approximation:

$$G \approx |G_x * I| + |G_y * I|$$

Properties and Characteristics

- **Efficient and easy to implement** – Uses integer coefficients for faster computation.
- **Detects both horizontal and vertical edges** in an image.
- **Less sensitive to noise** than the Roberts operator but not as strong as the Sobel filter.

Applications

- **Edge Detection** – Detects object boundaries in images.
- **Traffic Sign Recognition** – Identifying road signs based on edge information.
- **Medical Image Processing** – Enhancing edges in X-rays and CT scans.

MATLAB Code for Prewitt filter in Greyscale

```
% MATLAB Program to compute gradient magnitude and orientation using Prewitt filter (without built-in functions)

% Select an image from the user
[file, path] = uigetfile({'*.jpg;*.jpeg;*.png;*.bmp;*.tif', 'Image Files'});
if isequal(file,0)
    disp('User canceled the operation.');
    return;
end
imgPath = fullfile(path, file);
img = imread(imgPath);

% Convert to grayscale manually if needed
if size(img, 3) == 3
    grayImg = 0.2989 * img(:,:,1) + 0.5870 * img(:,:,2) + 0.1140 * img(:,:,3);
else
    grayImg = img;
end

grayImg = double(grayImg);

% Define Prewitt operator manually
Gx_kernel = [-1 0 1; -1 0 1; -1 0 1];  % Prewitt filter for X-direction
Gy_kernel = [-1 -1 -1; 0 0 0; 1 1 1];  % Prewitt filter for Y-direction

% Get image dimensions
[row, col] = size(grayImg);
Gx = zeros(row, col);
Gy = zeros(row, col);

% Apply Prewitt filter manually (without imfilter)
for i = 2:row-1
    for j = 2:col-1
        Gx(i, j) = sum(sum(Gx_kernel .* grayImg(i-1:i+1, j-1:j+1)));
        Gy(i, j) = sum(sum(Gy_kernel .* grayImg(i-1:i+1, j-1:j+1)));
```

```
    end
end

% Compute gradient magnitude
magnitude = sqrt(Gx.^2 + Gy.^2);

% Compute gradient orientation (in degrees)
orientation = atan2d(Gy, Gx);

% Normalize for display
magnitude = magnitude / max(magnitude(:));
orientation = (orientation - min(orientation(:))) / (max(
    orientation(:)) - min(orientation(:)));

% Display results
figure;
subplot(1,3,1); imshow(uint8(grayImg)); title('Grayscale Image');
subplot(1,3,2); imshow(magnitude); title('Gradient Magnitude');
subplot(1,3,3); imshow(orientation); title('Gradient Orientation'
    );
```

Output :

Figure 17: Apply Prewitt filter

Sobel (3×3) Filter for Edge Detection

Overview

The **Sobel filter** is a widely used edge detection operator that enhances edges by computing the gradient of an image. It is an improvement over the Prewitt filter as it applies more weight to the center pixels, making it more robust against noise.

Mathematical Formulation : The Sobel filter uses two 3×3 convolution kernels to approximate the gradient in the x- and y-directions:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

where:

- G_x detects edges in the **horizontal** direction.
- G_y detects edges in the **vertical** direction.

Gradient Magnitude Calculation : The edge strength is calculated as:

$$G = \sqrt{(G_x * I)^2 + (G_y * I)^2}$$

or an approximation:

$$G \approx |G_x * I| + |G_y * I|$$

Properties and Characteristics :

- **More sensitive to edges** compared to Prewitt due to higher center weight.
- **Effective in detecting both horizontal and vertical edges**.
- **Better noise suppression** than Roberts and Prewitt filters.

MATLAB Code for Sobel (3X3) filter in Greyscale

```
% MATLAB Program to compute gradient magnitude and orientation

% Select an image from the user
[file, path] = uigetfile({'*.jpg;*.jpeg;*.png;*.bmp;*.tif', '
   Image␣Files'});
if isequal(file,0)
    disp('User␣canceled␣the␣operation.');
    return;
end
imgPath = fullfile(path, file);
img = imread(imgPath);

% Convert to grayscale manually if needed
if size(img, 3) == 3
    grayImg = 0.2989 * img(:,:,1) + 0.5870 * img(:,:,2) + 0.1140
       * img(:,:,3);
else
    grayImg = img;
end

grayImg = double(grayImg);

% Define Sobel operator manually
Gx_kernel = [-1 0 1; -2 0 2; -1 0 1];
Gy_kernel = [-1 -2 -1; 0 0 0; 1 2 1];

% Get image dimensions
[row, col] = size(grayImg);
Gx = zeros(row, col);
```

```
Gy = zeros(row, col);

% Apply Sobel filter
for i = 2:row-1
    for j = 2:col-1
        Gx(i, j) = sum(sum(Gx_kernel .* grayImg(i-1:i+1, j-1:j+1)
           ));
        Gy(i, j) = sum(sum(Gy_kernel .* grayImg(i-1:i+1, j-1:j+1)
           ));
    end
end

% Compute gradient magnitude
magnitude = sqrt(Gx.^2 + Gy.^2);

% Compute gradient orientation (in degrees)
orientation = atan2d(Gy, Gx);

% Normalize for display
magnitude = magnitude / max(magnitude(:));
orientation = (orientation - min(orientation(:))) / (max(
   orientation(:)) - min(orientation(:)));

% Display results
figure;
subplot(1,3,1); imshow(uint8(grayImg)); title('Grayscale Image');
subplot(1,3,2); imshow(magnitude); title('Gradient Magnitude');
subplot(1,3,3); imshow(orientation); title('Gradient Orientation'
   );
```

Output :

Figure 18: Appy Sobel (3X3) filter

Sobel (5×5) Filter for Edge Detection

Overview
The **Sobel (5×5) filter** is an extension of the traditional Sobel (3×3) filter, designed to provide better edge detection by considering a larger neighborhood of pixels. The 5×5 kernel provides a smoother gradient approximation, improving edge accuracy and noise reduction.

Mathematical Formulation : The Sobel (5×5) filter uses larger convolution kernels to approximate the gradient in the x- and y-directions:

$$G_x = \begin{bmatrix} -2 & -1 & 0 & +1 & +2 \\ -3 & -2 & 0 & +2 & +3 \\ -4 & -3 & 0 & +3 & +4 \\ -3 & -2 & 0 & +2 & +3 \\ -2 & -1 & 0 & +1 & +2 \end{bmatrix}, \quad G_y = \begin{bmatrix} -2 & -3 & -4 & -3 & -2 \\ -1 & -2 & -3 & -2 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ +1 & +2 & +3 & +2 & +1 \\ +2 & +3 & +4 & +3 & +2 \end{bmatrix}$$

where:

- G_x detects edges in the **horizontal** direction.
- G_y detects edges in the **vertical** direction.

Gradient Magnitude Calculation : The edge strength is calculated as:

$$G = \sqrt{(G_x * I)^2 + (G_y * I)^2}$$

or an approximation:

$$G \approx |G_x * I| + |G_y * I|$$

Properties and Characteristics :

- **More precise edge detection** compared to the Sobel (3×3) filter.
- **Larger kernel size** helps smooth out noise while maintaining edge details.
- **Enhanced gradient computation** for better feature extraction.

Applications :

- **High-Resolution Image Processing** – Detecting fine details in large images.
- **Medical Imaging** – Enhancing edges in CT scans and MRI images.
- **Autonomous Navigation** – Edge detection for obstacle detection and path planning.

MATLAB Code for Sobel (5X5) filter in Greyscale

```
% MATLAB Program to compute gradient magnitude and orientation using Sobel 5x5 filter (without built-in functions)

% Select an image from the user
[file, path] = uigetfile({'*.jpg;*.jpeg;*.png;*.bmp;*.tif', 'Image Files'});
if isequal(file,0)
    disp('User canceled the operation.');
    return;
end
imgPath = fullfile(path, file);
img = imread(imgPath);

% Convert to grayscale manually if needed
if size(img, 3) == 3
    grayImg = 0.2989 * img(:,:,1) + 0.5870 * img(:,:,2) + 0.1140 * img(:,:,3);
else
    grayImg = img;
end

grayImg = double(grayImg);

% Define Sobel 5x5 operator manually
Gx_kernel = [-2 -1 0 1 2; -3 -2 0 2 3; -4 -3 0 3 4; -3 -2 0 2 3; -2 -1 0 1 2];
Gy_kernel = [-2 -3 -4 -3 -2; -1 -2 -3 -2 -1; 0 0 0 0 0; 1 2 3 2 1; 2 3 4 3 2];

% Get image dimensions
[row, col] = size(grayImg);
Gx = zeros(row, col);
Gy = zeros(row, col);

% Apply Sobel 5x5 filter manually (without imfilter)
for i = 3:row-2
    for j = 3:col-2
        Gx(i, j) = sum(sum(Gx_kernel .* grayImg(i-2:i+2, j-2:j+2)));
        Gy(i, j) = sum(sum(Gy_kernel .* grayImg(i-2:i+2, j-2:j+2)));
    end
end

% Compute gradient magnitude
magnitude = sqrt(Gx.^2 + Gy.^2);

% Compute gradient orientation (in degrees)
orientation = atan2d(Gy, Gx);
```

```
43
44 % Normalize for display
45 magnitude = magnitude / max(magnitude(:));
46 orientation = (orientation - min(orientation(:))) / (max(
       orientation(:)) - min(orientation(:)));
47
48 % Display results
49 figure;
50 subplot(1,3,1); imshow(uint8(grayImg)); title('Grayscale␣Image');
51 subplot(1,3,2); imshow(magnitude); title('Gradient␣Magnitude');
52 subplot(1,3,3); imshow(orientation); title('Gradient␣Orientation'
       );
```

Output :

Figure 19: Apply Sobel (5X5) filter

Canny Edge Detector

Overview

The **Canny edge detector** is a multi-stage algorithm used to detect edges with high accuracy while suppressing noise. It combines **Gaussian smoothing**, **gradient computation**, and **non-maximum suppression** to extract edges effectively.

Steps in Canny Edge Detection :

1. **Smooth the Image with a 2D Gaussian Filter:** To reduce noise, the image is first convolved with a Gaussian filter.

$$I_{\text{smooth}} = I * G_\sigma$$

   where G_σ is a Gaussian kernel with standard deviation σ.

2. **Compute Image Gradient using the Sobel Operator:** The gradients in both x- and y-directions are obtained using the Sobel filter:

$$G_x = I_{\text{smooth}} * S_x, \quad G_y = I_{\text{smooth}} * S_y$$

3. **Find Gradient Orientation at Each Pixel:** The gradient orientation θ is given by:

$$\theta = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

4. **Compute Laplacian Along the Gradient Direction:** The second derivative of the image along the gradient direction helps refine edges:

$$\frac{\partial^2 I}{\partial n^2}$$

5. **Find Gradient Magnitude:** The edge strength at each pixel is computed as:

$$G = \sqrt{G_x^2 + G_y^2}$$

6. **Find Zero Crossings in Laplacian to Locate Edges:** Edge pixels are identified by detecting zero crossings in the second derivative (Laplacian).

Characteristics of the Canny Edge Detector :

- **Low error rate:** Finds true edges while minimizing false detections.
- **Edge localization:** Precisely detects edge positions.
- **Single-pixel wide edges:** Uses non-maximum suppression to thin edges.
- **Double thresholding:** Distinguishes strong and weak edges.

Applications :

- **Medical Imaging** – Extracting blood vessel structures.
- **Object Detection** – Identifying object boundaries in images.
- **Facial Recognition** – Detecting facial contours in biometric systems.

MATLAB Code for Canny Edge Detector in Greyscale

```
% MATLAB Program for Canny Edge Detection (Without Built-in
   Functions)

clc; clear; close all;

% Select an image from the user
[file, path] = uigetfile({'*.jpg;*.jpeg;*.png;*.bmp;*.tif', '
   Image␣Files'});
if isequal(file,0)
    disp('User␣canceled␣the␣operation.');
    return;
end
imgPath = fullfile(path, file);
```

```
img = imread(imgPath);

% Convert to grayscale manually (if not already grayscale)
if size(img, 3) == 3
    grayImg = 0.2989 * img(:,:,1) + 0.5870 * img(:,:,2) + 0.1140 * img(:,:,3);
else
    grayImg = img;
end

grayImg = double(grayImg);

% Step 1: Apply Gaussian Smoothing
sigma = 1.4;  % Standard deviation for Gaussian filter
filterSize = 5;
[X, Y] = meshgrid(-floor(filterSize/2):floor(filterSize/2), -floor(filterSize/2):floor(filterSize/2));
GaussianFilter = exp(-(X.^2 + Y.^2) / (2 * sigma^2));
GaussianFilter = GaussianFilter / sum(GaussianFilter(:));

% Convolve with Gaussian filter manually
[row, col] = size(grayImg);
smoothedImg = zeros(row, col);
padSize = floor(filterSize / 2);
paddedImg = padarray(grayImg, [padSize padSize], 'replicate');

for i = 1:row
    for j = 1:col
        smoothedImg(i, j) = sum(sum(GaussianFilter .* paddedImg(i:i+filterSize-1, j:j+filterSize-1)));
    end
end

% Step 2: Compute Image Gradient using Sobel Operator
Gx_kernel = [-1 0 1; -2 0 2; -1 0 1];  % Sobel filter for X-direction
Gy_kernel = [-1 -2 -1; 0 0 0; 1 2 1];  % Sobel filter for Y-direction

Gx = zeros(row, col);
Gy = zeros(row, col);

% Apply Sobel filter manually
for i = 2:row-1
    for j = 2:col-1
        Gx(i, j) = sum(sum(Gx_kernel .* smoothedImg(i-1:i+1, j-1:j+1)));
        Gy(i, j) = sum(sum(Gy_kernel .* smoothedImg(i-1:i+1, j-1:j+1)));
    end
end
```

```
56
57 % Step 3: Find Gradient Magnitude at each Pixel
58 magnitude = sqrt(Gx.^2 + Gy.^2);
59
60 % Step 4: Find Gradient Orientation at each Pixel
61 orientation = atan2d(Gy, Gx);
62
63 % Step 5: Compute Laplacian along the Gradient Direction
64 laplacianImg = zeros(row, col);
65 laplacianKernel = [0 1 0; 1 -4 1; 0 1 0];  % 3x3 Laplacian Kernel
66
67 for i = 2:row-1
68     for j = 2:col-1
69         laplacianImg(i, j) = sum(sum(laplacianKernel .* magnitude
               (i-1:i+1, j-1:j+1)));
70     end
71 end
72
73 % Step 6: Find Zero Crossings in Laplacian to find edge location
74 zeroCrossingImg = zeros(row, col);
75
76 for i = 2:row-1
77     for j = 2:col-1
78         if laplacianImg(i,j) == 0
79             zeroCrossingImg(i,j) = 1;
80         else
81             negCount = sum(sum(laplacianImg(i-1:i+1, j-1:j+1) <
                   0));
82             posCount = sum(sum(laplacianImg(i-1:i+1, j-1:j+1) >
                   0));
83
84             if (negCount > 0) && (posCount > 0)
85                 zeroCrossingImg(i,j) = 1;  % Edge detected at
                       zero-crossing
86             end
87         end
88     end
89 end
90
91 % Display results
92 figure;
93 subplot(2,3,1); imshow(uint8(grayImg)); title('Grayscale␣Image');
94 subplot(2,3,2); imshow(uint8(smoothedImg)); title('Gaussian␣
       Smoothed␣Image');
95 subplot(2,3,3); imshow(magnitude, []); title('Gradient␣Magnitude'
       );
96 subplot(2,3,4); imshow(orientation, []); title('Gradient␣
       Orientation');
97 subplot(2,3,5); imshow(laplacianImg, []); title('Laplacian␣
       Response');
```

```
98 subplot(2,3,6); imshow(zeroCrossingImg, []); title('Final Edge
     Detection');
```

Output :

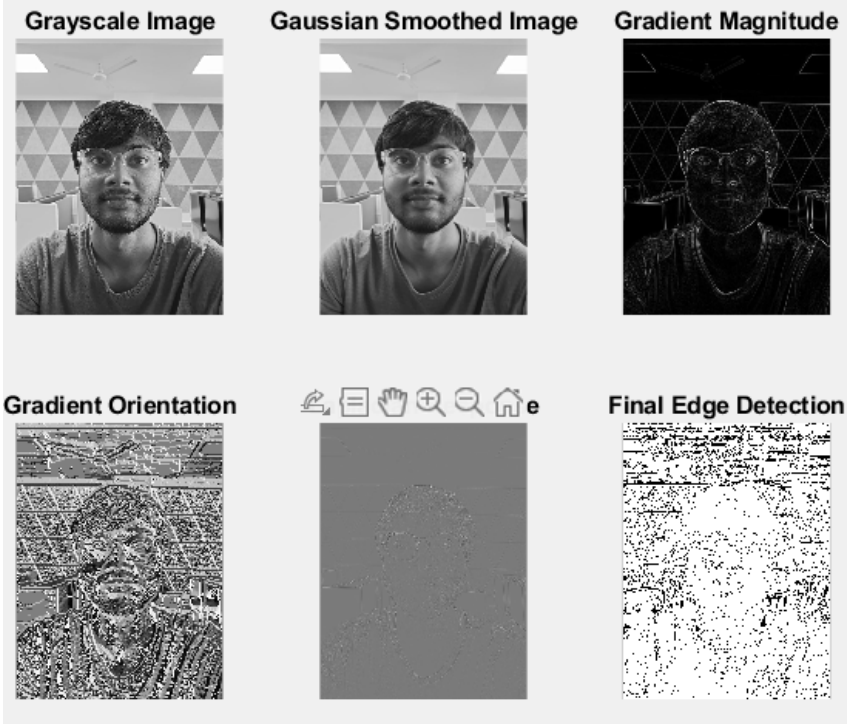

Figure 20: Apply Canny Edge Detection to Greyscale image