Department of Electrical & Computer Engineering

AIR UNIVERSITY

Complex Engineering Problem

Digital Image Processing

Instructor Name

Mam Zoafshan

Submitted by

Rafia Arshad 210275
Muhammad Burhan Ahmed 210287
Sadia Yaqoob 210284

5/28/2024

Contents

Image Segmentation using Matlab GUI

Objectives:

The objectives of this complex engineering problem are:

- To understand and implement different image segmentation techniques.
- To evaluate the performance of these techniques on a set of diverse test images.
- To provide visual demonstrations of segmented images.
- Conduct a comparative analysis to highlight the strengths and weaknesses of each method.

Software Tools:

- Matlab 2023a
- Test images provided.

1 Introduction

Image segmentation
It is a critical process in digital image processing, essential for tasks such as object detection, recognition, and image analysis. The main objective of image segmentation is to divide an image into meaningful regions to facilitate easier analysis and interpretation of the content. Various segmentation techniques have been developed, each with distinct advantages and limitations. This report examines four prominent segmentation methods:

1. Thresholding
2. Region Growing
3. Watershed and Multilevel Thresholding
4. Edge Detection

By implementing and analyzing these techniques, this report aims to demonstrate their effectiveness and applicability to different types of images.

2 Applications of Image Segmentation:

Some applications of Image Segmentation are:

- **Medical Imaging:** In medical imaging, image segmentation is used for identifying and delineating anatomical structures such as organs, tissues, tumors, and abnormalities. It aids in diagnosis, treatment planning, and surgical navigation.
- **Object Recognition and Tracking:** Image segmentation is crucial for object recognition and tracking in computer vision applications. It enables systems to detect and localize objects within images or videos, facilitating tasks such as object counting, classification, and motion analysis.
- **Digital Image Editing:** In graphic design and digital image editing software, segmentation is utilized for tasks such as background removal, image compositing, selective editing, and image enhancement. It allows users to isolate and manipulate specific regions or objects within images.
- **Industrial Quality Control:** Image segmentation is used in industrial applications for quality control, defect detection, and inspection. It enables the automated analysis of product surfaces, identifying defects, anomalies, or irregularities.

3 Deliverables:

The deliverables of our project include:

- Implementation and evaluation of four image segmentation techniques:
  1. Thresholding.
  2. Region Growing.
  3. Watershed & Multilevel thresholding.
  4. Edge Detection.
- MATLAB code containing source code for each image segmentation technique and the Graphical User Interface (GUI) application.
- Different images for validation and evaluation of segmentation algorithms under varied conditions.
- Development of an interactive GUI application facilitating user interaction, parameter adjustments, and real-time visualization of segmentation results.

4 Implementation of Graphic User Interface (GUI) in MATLAB

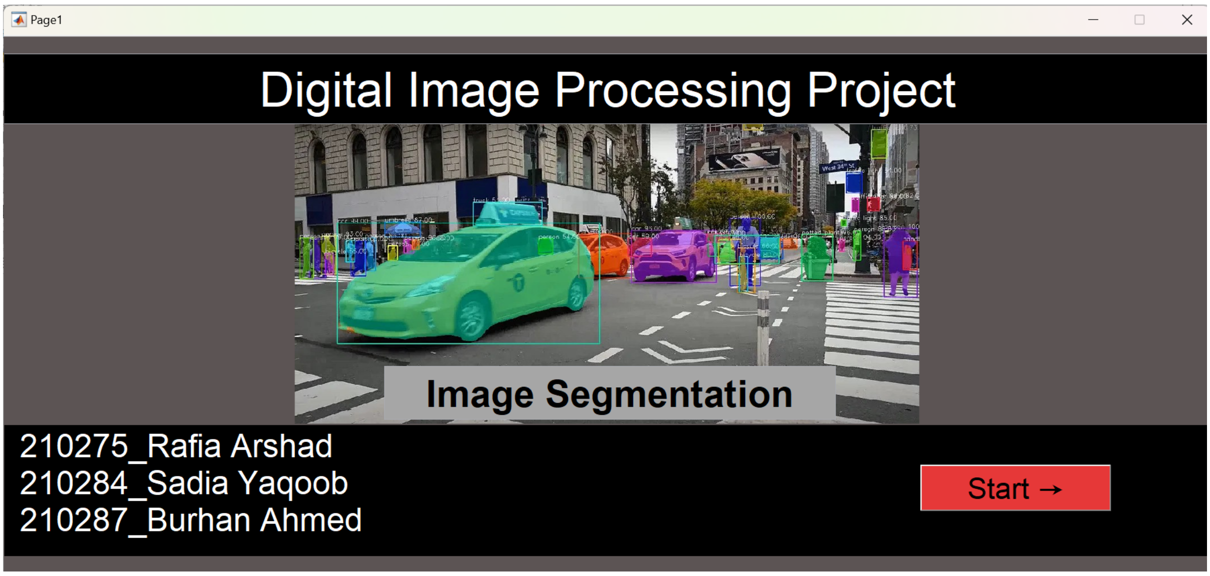


Figure 1: Main Page of Image Segmentation GUI.

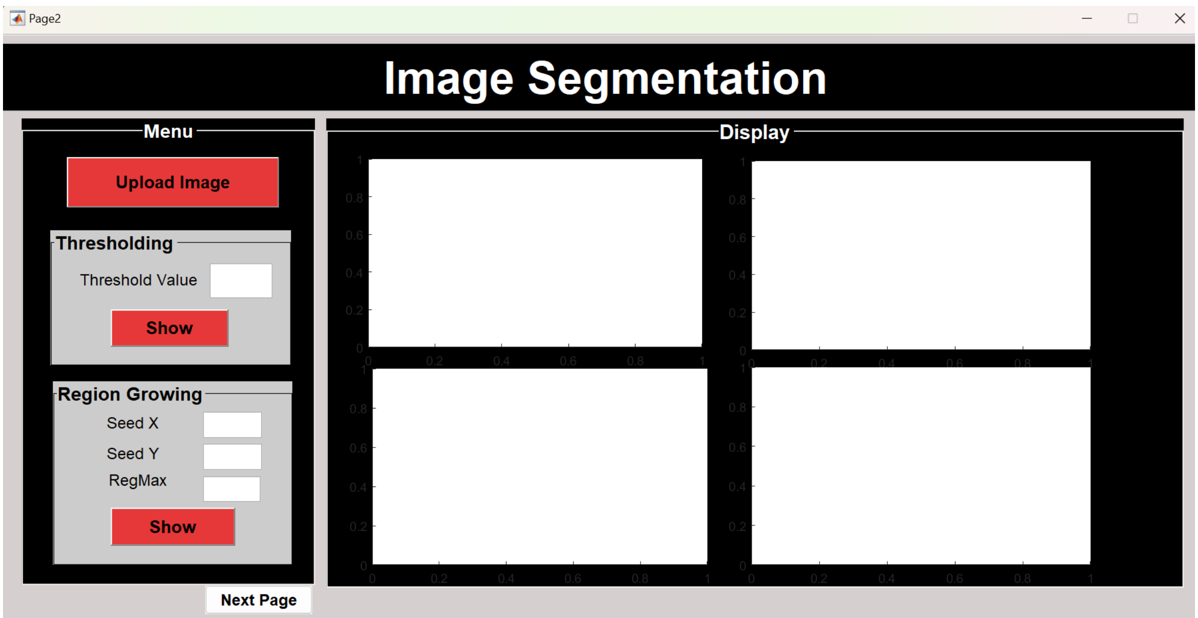


Figure 2: First Menu Page of GUI

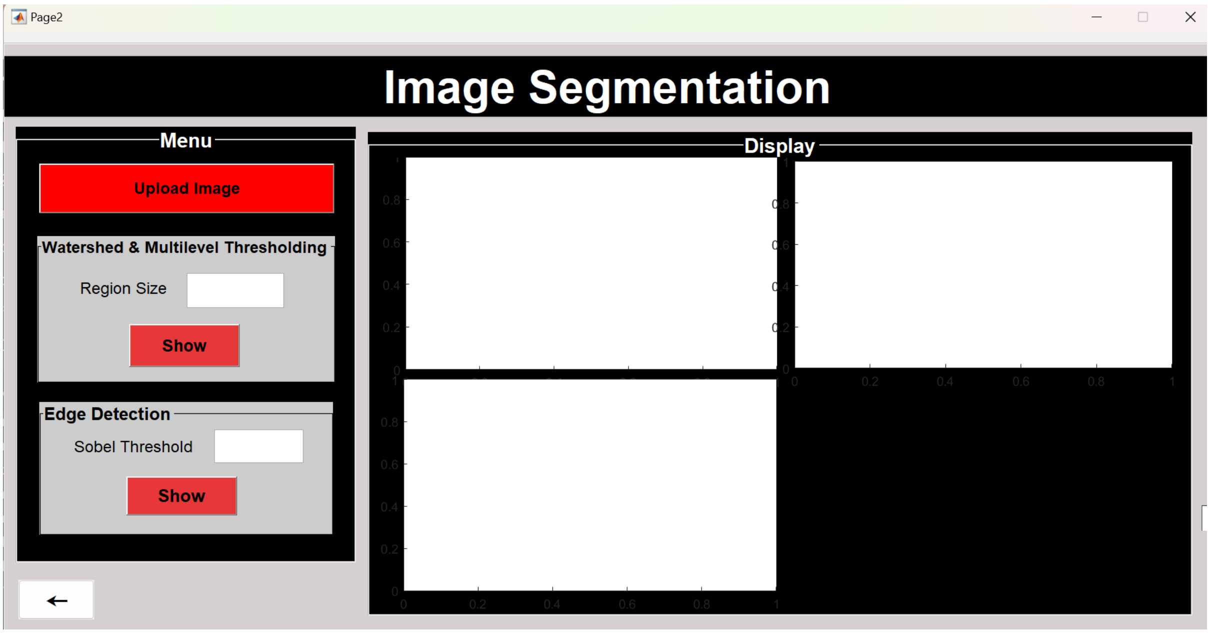


Figure 3: Second Menu Page of GUI

5 Implementation of Image Segmentation using various techniques

5.1 Thresholding

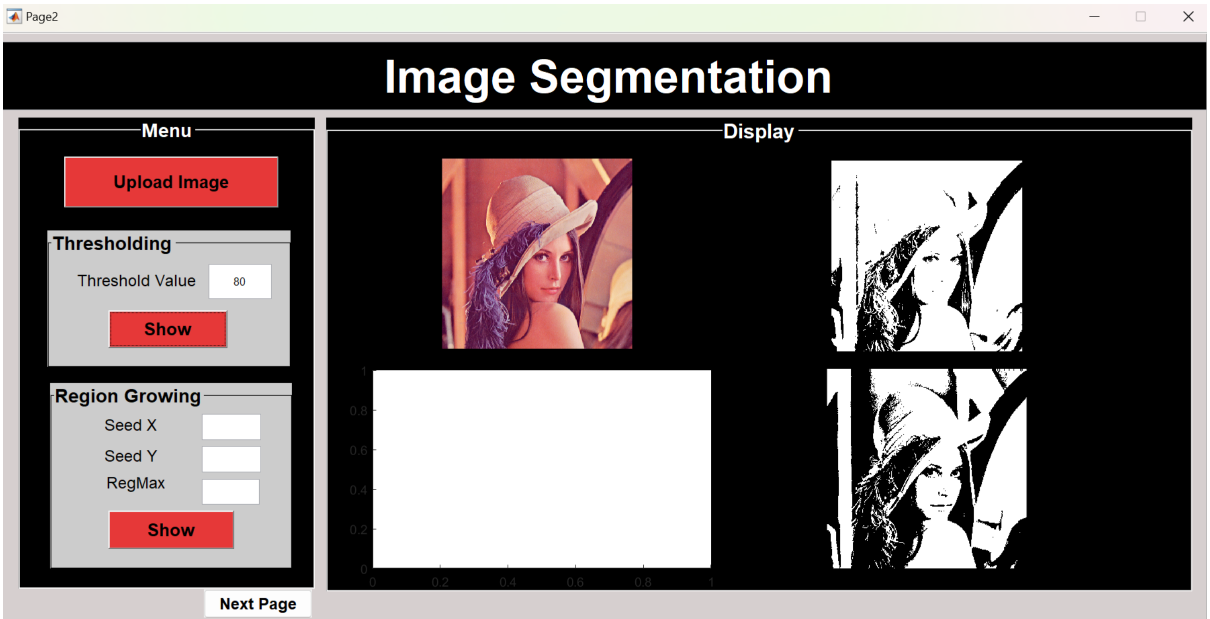
In this function, **button2_Callback**, is designed to perform image thresholding using both a simple user-defined threshold and Otsu's method, displaying the results on a (GUI). The function begins by retrieving the image data, converting it to gray scale if necessary. It then applies a simple thresholding method, where the threshold value is obtained from user input, creating a binary image based on this threshold. A histogram of pixel intensities is then calculated, and the probability distribution of these intensities is determined. The function computes the mean intensity of the image and uses Otsu's method to find the optimal threshold that maximizes the variance between the background and foreground. A binary image is also created using this optimal threshold. Finally, both the binary images (from the simple threshold and Otsu's method) are displayed on the GUI for comparison.

5.1.1 Source Code:

```
function button2_Callback(hObject, eventdata, handles)
% hObject    handle to button2 (see GCBO)
% eventdata  reserved - to be defined in a future version of
   MATLAB
% handles    structure with handles and user data (see GUIDATA)
a=getappdata(0,'a');

```

```
if size(a, 3) == 3
  img_gray = rgb2gray(a);
else
  img_gray = a;
end
val=str2num(get(handles.Threshinput,'string'));
% Apply simple thresholding
simple_threshold = val;
binary_img_simple = img_gray > simple_threshold;

[rows, cols] = size(img_gray);

histogram = zeros(256, 1);
for i = 1:rows
  for j = 1:cols
    pixel_value = img_gray(i, j);
    histogram(pixel_value + 1) = histogram(pixel_value + 1) + 1;
  end
end

total_pixels = sum(histogram);

probability = histogram / total_pixels;

mean_intensity = 0;
for i = 1:256
  mean_intensity = mean_intensity + i * probability(i);
end

max_variance = 0;
optimal_threshold = 0;

for threshold = 1:255
  weight_background = sum(probability(1:threshold));
  weight_foreground = sum(probability(threshold + 1:end));

  if weight_background == 0 || weight_foreground == 0
    continue;
  end

  mean_background = 0;
  for i = 1:threshold
    mean_background = mean_background + i * probability(i);
  end
  mean_background = mean_background / weight_background;

  mean_foreground = 0;
  for i = threshold + 1:256
    mean_foreground = mean_foreground + i * probability(i);
  end
  mean_foreground = mean_foreground / weight_foreground;
```

```matlab

  variance = weight_background * weight_foreground *
     (mean_background - mean_foreground).^2;

  if variance > max_variance
    max_variance = variance;
    optimal_threshold = threshold;
  end
end

% Apply Otsu's threshold
binary_img_otsu = img_gray > optimal_threshold;

setappdata(0,'filename',binary_img_simple)
axes(handles.axes2);
imshow(binary_img_simple);
title('Simple Threshold');

setappdata(0,'filename',binary_img_otsu)
axes(handles.axes3);
imshow(binary_img_otsu);
title('Simple Threshold');
title('Otsu''s Threshold (Custom)');
```



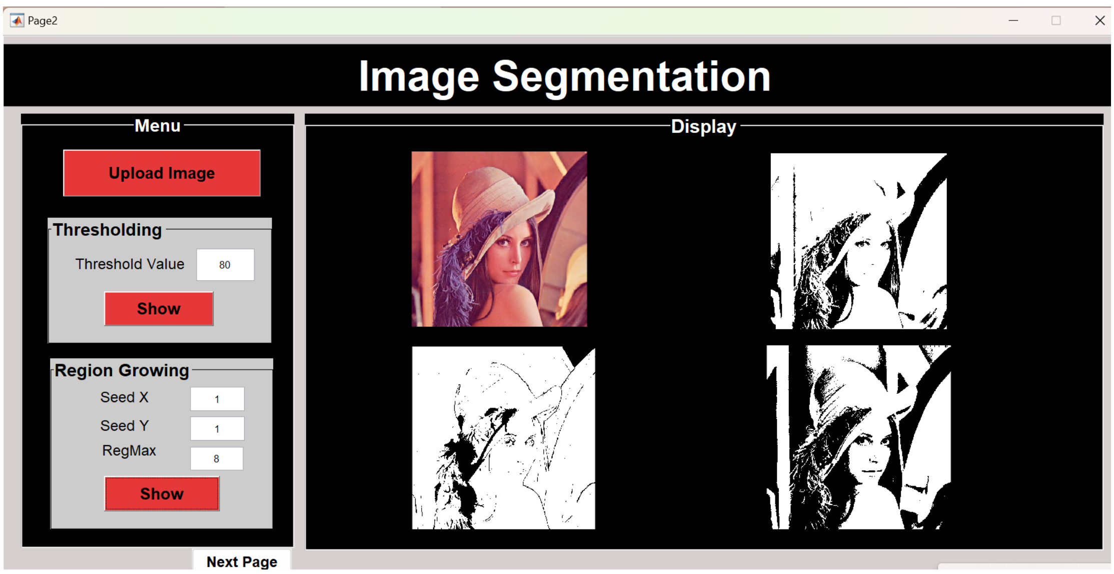
5.2 Region Growing

In this function, **button3_Callback** implements a region growing technique to segment an image based on user-defined seed coordinates and a maximum intensity difference threshold. It retrieves the image data from the application data, converts it to grayscale if necessary, and initializes a binary mask to mark the segmented region. Using 8-connectivity, it iteratively expands the region by adding neighboring pixels that satisfy the intensity similarity criterion, updating the mask accordingly. The process continues until no more eligible pixels remain in the queue. Finally, the resulting segmented image, represented by the binary mask, is displayed in the GUI's axes4 for visualization and further analysis.

5.2.1 Source Code:

```matlab
function button3_Callback(hObject, eventdata, handles)
% hObject    handle to button3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

a=getappdata(0,'a');

if size(a, 3) == 3
  img_gray = rgb2gray(a);
else
  img_gray = a;
end

seed_x=str2double(get(handles.seedx,'string'));
seed_y=str2double(get(handles.seedy,'string'));
reg_maxdist=str2double(get(handles.regmax,'string'));

% Initialize binary mask (all zeros)
    mask = false(size(img_gray));
    mask(seed_y, seed_x) = 1;

    % neighborhood offsets (8-connectivity)
    dx = [-1, -1, 0, 1, 1, 1, 0, -1];
    dy = [0, 1, 1, 1, 0, -1, -1, -1];

    queue = [seed_x, seed_y];

    while ~isempty(queue)
        % Get current pixel coordinates
        current_x = queue(1);
        current_y = queue(2);
        queue(1:2) = [];

```

```
        for k = 1:8
            neighbor_x = current_x + dx(k);
            neighbor_y = current_y + dy(k);

            % Check if neighbor is within image bounds
            if neighbor_x >= 1 && neighbor_x <= size(img_gray, 2) && ...
               neighbor_y >= 1 && neighbor_y <= size(img_gray, 1)

                % Calculate intensity difference
                intensity_diff = abs(double(img_gray(current_y, current_x)) - double(img_gray(neighbor_y, neighbor_x)));

                % Add neighbor to region if within similarity threshold
                if intensity_diff <= reg_maxdist && ~mask(neighbor_y, neighbor_x)
                    mask(neighbor_y, neighbor_x) = 1;
                    queue(end+1:end+2) = [neighbor_x, neighbor_y]; % Enqueue
                end
            end
        end
    end

    segmented_image = uint8(mask) * 255;

setappdata(0,'filename',segmented_image);
axes(handles.axes4);
imshow(segmented_image);
```

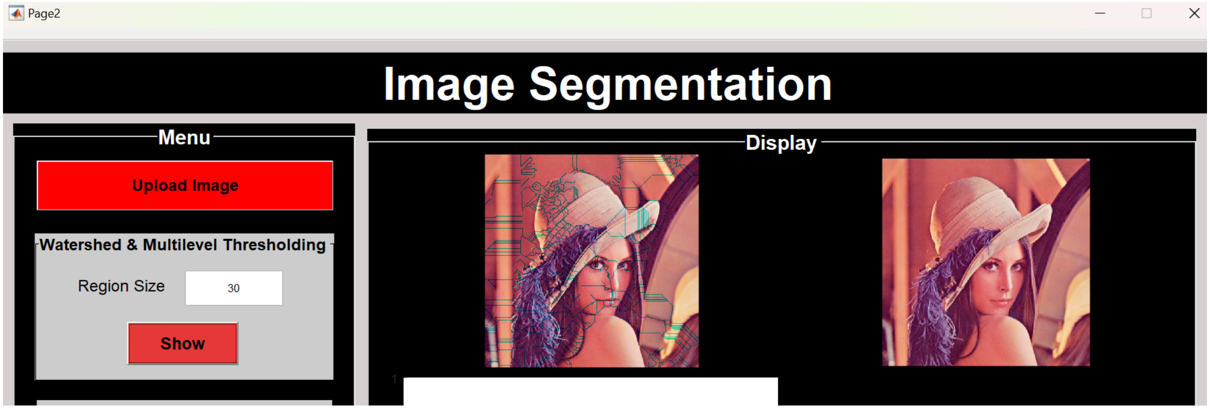


5.3 Watershed and Multilevel Thresholding

This code segment performs image segmentation using the watershed transform technique. Initially, it converts the input image to grayscale if it's not already in grayscale. Then, it applies Otsu's thresholding to binarize the image. Subsequently, it removes small objects from the binary image to eliminate noise. Next, it computes the distance transform of the complement of the binary image to generate a gradient-like image. Using this gradient image, it applies the watershed transform to identify regions or catchment basins. Finally, it assigns each pixel in the original image to a unique catchment basin, effectively segmenting the image into distinct regions. The resulting segmented image is displayed in the fifth axes of the GUI.

5.3.1 Source Code:

```
function button4_Callback(hObject, eventdata, handles)
% hObject    handle to button4 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

a=getappdata(0,'a');
if size(a, 3) == 3
  grayImage = rgb2gray(a);
else
  grayImage = a;
end

threshold = graythresh(grayImage);
binaryImage = imbinarize(grayImage, threshold);

binaryImage = bwareaopen(binaryImage, 50);

distanceTransform = bwdist(~binaryImage);

watershedMask = watershed(-distanceTransform);

segmentedImg = a;
segmentedImg(watershedMask == 0) = 0;

setappdata(0,'filename', segmentedImg);
axes(handles.axes5);
imshow(segmentedImg);
```



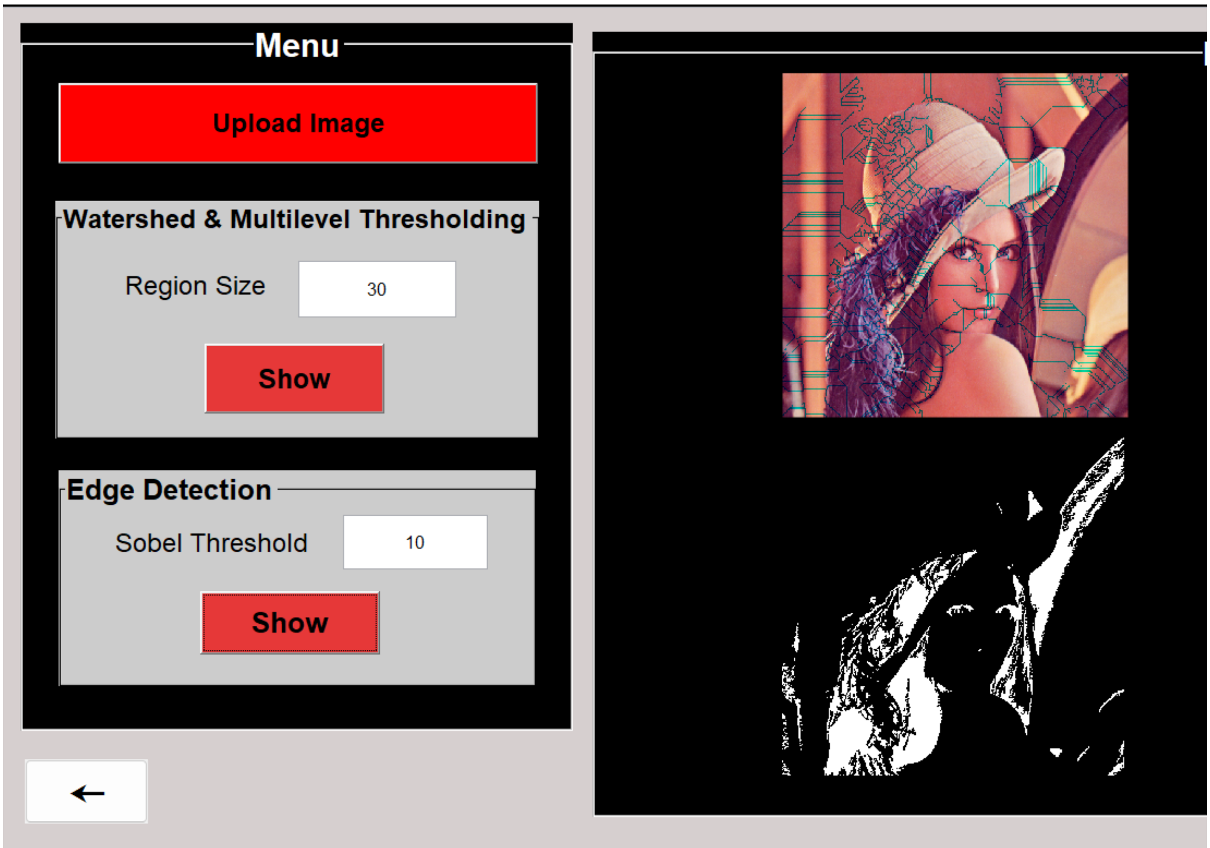
5.4 Edge Detection

This code segment performs edge detection using the Sobel operator on the input image. First, it converts the input image to grayscale if it's in RGB format. Then, it iterates through each pixel of the grayscale image, applying the Sobel operator to compute the gradient magnitude at each pixel. Next, it sets a threshold to binarize the gradient magnitude image, resulting in an initial edge map. Afterward, it fills any holes within the detected edges and clears edge objects touching the image borders to produce the final segmented edge image. Finally, it displays the segmented edge image on the GUI.

5.4.1 Source Code:

```
function button5_Callback(hObject, eventdata, handles)
% hObject    handle to button5 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
a=getappdata(0,'a');
if size(a, 3) == 3
  grayImage = rgb2gray(a);
else
  grayImage = a;
end

% Apply Sobel edge detection
sobelEdgeImage = zeros(size(grayImage));
sobelThreshold = 0.1;

for i = 2:size(grayImage, 1)-1
    for j = 2:size(grayImage, 2)-1

        Gx = grayImage(i-1,j-1) + 2*grayImage(i,j-1) + grayImage(i+1,j-1) ...
             - grayImage(i-1,j+1) - 2*grayImage(i,j+1) - grayImage(i+1,j+1);
```

```
        Gy = grayImage(i-1,j-1) + 2*grayImage(i-1,j) +
           grayImage(i-1,j+1) ...
            - grayImage(i+1,j-1) - 2*grayImage(i+1,j) -
               grayImage(i+1,j+1);

        sobelEdgeImage(i,j) = sqrt(double(Gx)^2 + double(Gy)^2);
    end
end

sobelEdgeImage = sobelEdgeImage > sobelThreshold;

segmentedImage = imfill(sobelEdgeImage, 'holes'); % Fill holes
   in edges
segmentedImage = imclearborder(segmentedImage); % Clear border
   objects

setappdata(0,'filename',segmentedImage);
axes(handles.axes6);
imshow(segmentedImage);
```



6 Comparative Analysis

The following table highlights the strengths and weaknesses of each image segmentation technique evaluated in this report:

Technique	Strengths	Weaknesses
Thresholding	• Simple and fast. • Effective for high-contrast images.	• Not effective for images with varying lighting conditions. • Sensitive to noise.
Region Growing	• Provides accurate segmentation for homogeneous regions. • Intuitive and easy to implement.	• Sensitive to noise. • Dependent on the choice of seed points.
Watershed and Multilevel Thresholding	• Excellent for segmenting complex structures. • Suitable for images with multiple regions and varying intensity gradients. • Can be automated using optimization techniques.	• Prone to over-segmentation. • Computationally intensive. • Requires careful selection of threshold values.
Edge Detection	• Accurate boundary detection. • Useful for applications requiring precise delineation.	• Sensitive to noise. • May miss weak edges.

7 Conclusion

This project explored and implemented five image segmentation techniques, evaluating their performance on a diverse set of images. Each technique has its unique advantages and limitations, making them suitable for different applications. Thresholding is simple and efficient for high-contrast images, while Region Growing and Watershed provide more detailed segmentation at the cost of increased computational complexity. Multilevel Thresholding is effective for images with multiple regions, and Edge Detection excels in detecting precise boundaries. Each technique offers unique strengths and limitations, making them suitable for different applications and scenarios. By understanding the characteristics and performance of these segmentation techniques, one can make decisions when choosing segmentation methods for their specific tasks.