```
classdef imageproject < matlab.apps.AppBase

    % Properties that correspond to app components
    properties (Access = public)

    properties (Access = private)
        img % Description
        img2 % Description
        noisyImage
    end



    % Callbacks that handle component events
    methods (Access = private)

        % Button pushed function: SELECTButton
        function SELECT(app, event)
            [file, path] = uigetfile('*.*');
            if isequal(file,0)
               figure(app.UIFigure);
                return ;
            end
            figure(app.UIFigure);
           app.img = imread(fullfile(path,file));
           imshow(app.img,'Parent',app.UIAxes);
           title(app.UIAxes, "INPUT IMAGE");
        end

        % Button pushed function: convert2grayButton
        function convert(app, event)
            app.img2 = rgb2gray(app.img);
            imshow(app.img2,'Parent',app.UIAxes2);
            title(app.UIAxes2, "OUTPUT IMAGE");
```

```
        end


        % Button pushed function: RESETButton
        function reset(app, event)
            cla(app.UIAxes, 'reset'); % Clear all graphics objects and reset
properties
    set(app.UIAxes, 'Visible', 'off'); % Optionally turn off visibility to
indicate it's cleared
           cla(app.UIAxes2, 'reset');
    set(app.UIAxes2, 'Visible', 'off');


    cla(app.UIAxes3, 'reset');
    set(app.UIAxes3, 'Visible', 'off');




        end


        % Button pushed function: pointenhanscementButton
        function histogram(app, event)
            grayImage = rgb2gray(app.img); % Ensure grayscale
     % Convert image to double precision if necessary
    if ~isa(grayImage, 'double')
        grayImage = im2double(grayImage); % Convert to double for processing
    end


    equalizedImage = histeq(grayImage);


    % Display the equalized image
    imshow(equalizedImage, 'Parent', app.UIAxes2);


    % Display the histogram
    figure;
    imhist(equalizedImage);
```

```
title('Histogram of Equalized Image');
    end

    % Button pushed function: AverageButton
    function average(app, event)

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

avg = fspecial('average');

% Applying the avg filter
favg = imfilter(app.img, avg, 'replicate');



imshow(favg, 'Parent',app.UIAxes2);
title(app.UIAxes2, "Average filter");

    end

    % Button pushed function: SharpningButton
    function sharp(app, event)

if size(app.img, 3) == 3
    app.img = rgb2gray(app.img);
end

sharp = fspecial('unsharp');



fsharp = imfilter(app.img, sharp, 'replicate');
```

```
    imshow(fsharp, 'Parent',app.UIAxes2);
    title(app.UIAxes2, "Sharping filter");
        end

        % Value changed function: AddNoiseDropDown
        function noise(app, event)
            value = app.AddNoiseDropDown.Value;
            % Callback function for dropdown menu to add noise



    if size(app.img, 3) == 3
        grayImage = rgb2gray(app.img);
    else
        grayImage = app.img;
    end

    % Add noise based on the selected type
    switch value
        case 'Gaussian'
            noisyImage = imnoise(grayImage, 'gaussian', 0, 0.01);
        case 'Salt & Pepper'
            noisyImage = imnoise(grayImage, 'salt & pepper', 0.02);
        case 'Speckle'
            noisyImage = imnoise(grayImage, 'speckle', 0.04);
        case 'Uniform'
            noisyImage = grayImage + uint8(25 * (2 * rand(size(grayImage)) -
1));
        case 'Periodic'
            [rows, cols] = size(grayImage);
            periodicNoise = 64 * sin(2 * pi * 0.2 * (1:cols));
            noisyImage = grayImage + uint8(repmat(periodicNoise, rows, 1));
```

```
        case 'Salt'
            noisyImage = grayImage;
            numSalt = round(numel(grayImage) * 0.02);
            saltIdx = randperm(numel(grayImage), numSalt); % Random
            noisyImage(saltIdx) = 255;
        case 'Pepper'
            noisyImage = grayImage;
            numPepper = round(numel(grayImage) * 0.02);
            pepperIdx = randperm(numel(grayImage), numPepper); % Random
            noisyImage(pepperIdx) = 0;
        otherwise
            noisyImage = grayImage; % invalid choise
    end

    imshow(noisyImage, 'Parent', app.UIAxes2);
    title(app.UIAxes2, ['Image with ', value, ' Noise']);
app.noisyImage = noisyImage; % Store the noisy image



        end


        % Callback function
        function log(app, event)





        end


        % Button pushed function: LogarithmButton
        function Log(app, event)

      if size(app.img, 3) == 3
```

```
        grayImage = rgb2gray(app.img);
    else
        grayImage = app.img;
end
% Set parameters
sigma = 2.0;
kernelSize = 2 * ceil(3 * sigma) + 1;

% Create filter
logFilter = fspecial('log', kernelSize, sigma);

filteredImage = imfilter(double(grayImage) , logFilter, 'replicate');

imshow(filteredImage, "Parent" , app.UIAxes2);
title(app.UIAxes2, ['Logarithmic Filter']);
        end

        % Button pushed function: PowerlawButton
        function PowerLaw(app, event)
            if size(app.img, 3) == 3
    grayImage = rgb2gray(app.img);
else
    grayImage = app.img;
end

gamma = 0.5;
normalizedImage = double(grayImage) / 255;
powerLawImage = normalizedImage .^ gamma;

% Scale the result back to [0, 255] and convert to uint8
powerLawImage = uint8(powerLawImage * 255);
```

```
imshow(powerLawImage, "Parent", app.UIAxes2);
title(app.UIAxes2, ['Power Law Transformation']);
        end

        % Button pushed function: LaplacianButton
        function Laplacian(app, event)
            if size(app.img, 3) == 3
    grayImage = rgb2gray(app.img);
else
    grayImage = app.img;
end
laplacianFilter = fspecial('laplacian', 0.2);
% Apply filter
filteredImage = imfilter(double(grayImage), laplacianFilter, 'replicate');
imshow(filteredImage, "Parent", app.UIAxes2);
title(app.UIAxes2, 'Laplacian Filter');
        end

        % Button pushed function: LaplacianofGaussianButton
        function LaplacianOfGaussian(app, event)
            if size(app.img, 3) == 3
    grayImage = rgb2gray(app.img);
else
    grayImage = app.img;
end
sigma = 2.0;
kernelSize = 2 * ceil(3 * sigma) + 1;
% Gaussian filter
gaussianFilter = fspecial('gaussian', kernelSize, sigma);
% Laplacian filter
laplacianFilter = fspecial('laplacian', 0.2); % 0.2 is the default alpha value
% Apply Gaussian
smoothedImage = imfilter(double(grayImage), gaussianFilter, 'replicate');
```

```
% Laplacian of Gaussian (LoG) filter
filteredImage = imfilter(smoothedImage, laplacianFilter, 'replicate');


imshow(filteredImage, "Parent", app.UIAxes2);
title(app.UIAxes2, 'Laplacian of Gaussian Filter');
        end


        % Button pushed function: MedianButton
        function median(app, event)
            if size(app.img, 3) == 3
    grayImage = rgb2gray(app.img);
else
    grayImage = app.img;
end
filterSize = 3; % 3x3


filteredImage = medfilt2(grayImage, [filterSize filterSize]);


imshow(filteredImage, "Parent", app.UIAxes2);
title(app.UIAxes2, 'Median Filter');
        end


        % Button pushed function: GaussianButton
        function Gaussian(app, event)
            if size(app.img, 3) == 3
    grayImage = rgb2gray(app.img);
else
    grayImage = app.img;
end


sigma = 2.0;
kernelSize = 2 * ceil(3 * sigma) + 1;
```

```
% Create filter
gaussianFilter = fspecial('gaussian', kernelSize, sigma);

% Apply Gaussian filter with correlation
correlatedImage = imfilter(double(grayImage), gaussianFilter, 'corr',
'replicate');

% Apply Gaussian filter with convolution
convolutionImage = imfilter(double(grayImage), gaussianFilter, 'conv',
'replicate');

imshowpair(correlatedImage, convolutionImage, 'montage');
title('Gaussian Filter: Correlation (left) and Convolution (right)');

        end

        % Button pushed function: GradientfilterButton
        function Gradient(app, event)
            % Convert to grayscale if the image is RGB
if size(app.img, 3) == 3
    grayImage = rgb2gray(app.img);
else
    grayImage = app.img;
end

prewittFilterH = fspecial('prewitt'); % Horizontal Prewitt filter
prewittFilterV = prewittFilterH';     % Vertical Prewitt filter

sobelFilterH = fspecial('sobel');    % Horizontal Sobel filter
sobelFilterV = sobelFilterH';        % Vertical Sobel filter

prewittImageH = imfilter(double(grayImage), prewittFilterH, 'replicate');
prewittImageV = imfilter(double(grayImage), prewittFilterV, 'replicate');
```

```
sobelImageH = imfilter(double(grayImage), sobelFilterH, 'replicate');
sobelImageV = imfilter(double(grayImage), sobelFilterV, 'replicate');

% Combine Horizontal and Vertical
prewittCombined = prewittImageH + prewittImageV;
sobelCombined = sobelImageH + sobelImageV ;

% Prewitt filters
subplot(2, 3, 1), imshow(prewittImageH, []), title('Prewitt (Horizontal)');
subplot(2, 3, 2), imshow(prewittImageV, []), title('Prewitt (Vertical)');
subplot(2, 3, 3), imshow(prewittCombined, []), title('Prewitt Filter');

% Sobel filters
subplot(2, 3, 4), imshow(sobelImageH, []), title('Sobel (Horizontal)');
subplot(2, 3, 5), imshow(sobelImageV, []), title('Sobel (Vertical)');
subplot(2, 3, 6), imshow(sobelCombined, []), title('Sobel Filter');
        end

        % Value changed function: LowpassDropDown
        function lowpassFourier(app, event)

    % Get the selected filter type from the dropdown
    value = app.LowpassDropDown.Value;

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

    % Perform Fourier Transform
    F = fft2(double(grayImage));
```

```
Fshift = fftshift(F);

% Get image size and create meshgrid for frequency domain
[M, N] = size(grayImage);
[x, y] = meshgrid(-floor(N/2):floor(N/2)-1, -floor(M/2):floor(M/2)-1);

% Set parameters for filters
D0 = 30;        % Cutoff frequency
n = 2;          % Order for Butterworth filter
filteredImage = grayImage; % Initialize in case of default

% Apply the selected filter
switch value
    case 'Ideal'
        % Ideal Low-Pass Filter
        idealFilter = double(sqrt(x.^2 + y.^2) <= D0);
        F_filtered = Fshift .* idealFilter;
        filteredImage = real(ifft2(ifftshift(F_filtered)));
        titleText = 'Ideal Low-Pass Filter';

    case 'Butterworth'
        % Butterworth Low-Pass Filter
        butterworthFilter = 1 ./ (1 + (sqrt(x.^2 + y.^2) / D0).^(2 * n));
        F_filtered = Fshift .* butterworthFilter;
        filteredImage = real(ifft2(ifftshift(F_filtered)));
        titleText = 'Butterworth Low-Pass Filter';

    case 'Gaussian'
        % Gaussian Low-Pass Filter
        gaussianFilter = exp(-(x.^2 + y.^2) / (2 * D0^2));
        F_filtered = Fshift .* gaussianFilter;
        filteredImage = real(ifft2(ifftshift(F_filtered)));
        titleText = 'Gaussian Low-Pass Filter';
```

```
    otherwise
        % Default to original image if unexpected value
        filteredImage = grayImage;
        titleText = 'Original Image';
end

% Scale the intensity of the filtered image for display
filteredImage = mat2gray(filteredImage); % Normalize to [0, 1] range

% Display the filtered image in the specified UIAxes
imshow(filteredImage, 'Parent', app.UIAxes2);
title(app.UIAxes2, titleText);



    end

    % Value changed function: HighpassDropDown
    function HighpassFourier(app, event)
        value = app.HighpassDropDown.Value;

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

% Perform Fourier Transform
F = fft2(double(grayImage));
Fshift = fftshift(F);

% Get image size and create meshgrid for frequency domain
```

```
[M, N] = size(grayImage);
[x, y] = meshgrid(-floor(N/2):floor(N/2)-1, -floor(M/2):floor(M/2)-1);

% High-Pass Filter Parameters
D0 = 30;       % Cutoff frequency for all filters
n = 2;         % Order for Butterworth filter

% Apply the selected high-pass filter
switch value
    case 'Ideal'
        % Ideal High-Pass Filter
        idealFilter = double(sqrt(x.^2 + y.^2) > D0);
        F_filtered = Fshift .* idealFilter;
        filteredImage = real(ifft2(ifftshift(F_filtered)));
        titleText = 'Ideal High-Pass Filter';

    case 'Butterworth'
        % Butterworth High-Pass Filter
        butterworthFilter = 1 ./ (1 + (D0 ./ sqrt(x.^2 + y.^2)).^(2 * n));
        F_filtered = Fshift .* butterworthFilter;
        filteredImage = real(ifft2(ifftshift(F_filtered)));
        titleText = 'Butterworth High-Pass Filter';

    case 'Gaussian'
        % Gaussian High-Pass Filter
        gaussianFilter = 1 - exp(-(x.^2 + y.^2) / (2 * D0^2));
        F_filtered = Fshift .* gaussianFilter;
        filteredImage = real(ifft2(ifftshift(F_filtered)));
        titleText = 'Gaussian High-Pass Filter';

    otherwise
        % In case of unexpected value, default to the original image
        filteredImage = grayImage;
```

```
            titleText = 'Original Image';
    end

    % Display the filtered image in the specified UIAxes
    imshow(filteredImage, 'Parent', app.UIAxes2);
    title(app.UIAxes2, titleText);

        end

        % Button pushed function: LaplacianFButton
        function LaplacianFourier(app, event)
% Convert the image to grayscale if it is RGB
grayImage = rgb2gray(app.img);

% Transform to Fourier Domain
Fshift = fftshift(fft2(double(grayImage)));

% Create Laplacian Filter in Frequency Domain
[M, N] = size(grayImage);
[x, y] = meshgrid(-floor(N/2):floor(N/2)-1, -floor(M/2):floor(M/2)-1);
laplacian = -(x.^2 + y.^2);
laplacianFilter = laplacian / max(abs(laplacian(:))); % Normalize

% Apply the Laplacian Filter
filteredImage = real(ifft2(ifftshift(Fshift .* laplacianFilter)));
filteredImage = mat2gray(filteredImage);

% Add Filtered Image to Original
enhancedImage = mat2gray(im2double(grayImage) + filteredImage);

% Display Fourier Magnitude Spectrum
figure;
imshow(log(1 + abs(Fshift)), []);
```

```
title('Fourier Spectrum');

% Display Results in App
imshow(filteredImage, [], 'Parent', app.UIAxes3);
title(app.UIAxes3, 'Laplacian Filter Applied');
imshow(enhancedImage, [], 'Parent', app.UIAxes2);
title(app.UIAxes2, 'Original + Laplacian');

        end

        % Button pushed function: AliasingButton
        function Aliasing(app, event)

   % Convert the image to grayscale if it is RGB
if size(app.img, 3) == 3
    grayImage = rgb2gray(app.img);
else
    grayImage = app.img;
end

% Fourier Transform
F = fft2(double(grayImage));
Fshift = fftshift(F);  % Shift zero frequency to the center

% Display the Spectrum
magnitudeSpectrum = log(1 + abs(Fshift));  % Compute the log of the magnitude
spectrum

% Show the Spectrum in a Figure
figure;
imshow(magnitudeSpectrum, []);
title('Fourier Aliasing Spectrum');
```

```
        end

        % Value changed function: RemoveNoiseDropDown
        function noiseRemoval(app, event)
            value = app.RemoveNoiseDropDown.Value;

    % Convert noisy image to double for filtering
    imgdouble = double(app.noisyImage);

    % Apply filter based on the selected type
    switch value
        case 'Geometric mean'
            geometricMeanFilter = fspecial('average', [3 3]);
            filteredImage = exp(imfilter(log(imgdouble + 1),
geometricMeanFilter, 'replicate')) - 1;

        case 'Harmonic mean'
            filteredImage = 9 ./ imfilter(1 ./ (imgdouble + eps), ones(3, 3),
'replicate');

        case 'Contra-harmonic mean'
            Q = 1.5;
            filteredImage = imfilter(imgdouble.^(Q + 1), ones(3, 3),
'replicate')./ (imfilter(imgdouble.^Q, ones(3, 3), 'replicate') + eps);

        case 'Median'
            filteredImage = medfilt2(imgdouble, [3 3]);

        case 'Max'
            filteredImage = ordfilt2(imgdouble, 9, ones(3, 3));

        case 'Min'
            filteredImage = ordfilt2(imgdouble, 1, ones(3, 3));
```

```
    case 'Midpoint'
        maxFiltered = ordfilt2(imgdouble, 9, ones(3, 3));
        minFiltered = ordfilt2(imgdouble, 1, ones(3, 3));
        filteredImage = (maxFiltered + minFiltered) / 2;

 case 'Band reject'
%Fourier Transform
[M, N] = size(imgdouble);
F = fft2(imgdouble);
Fshift = fftshift(F);
noiseFrequencies = [0.2, -0.2];
harmonics = 1:3;
W = 2;
[X, Y] = meshgrid(-N/2:N/2-1, -M/2:M/2-1);
D = sqrt(X.^2 + Y.^2);
H = ones(size(Fshift));
for freq = noiseFrequencies
    for h = harmonics
        D0 = h * freq * min(M, N); % Harmonic distance from center
        H = H .* (1 ./ (1 + ((D .* W) ./ (D.^2 - D0^2 + eps)).^2));
    end
end
Ffiltered = Fshift .* H;
filteredImage = real(ifft2(ifftshift(Ffiltered)));



    otherwise
        filteredImage = imgdouble;
end

filteredImageNormalized = mat2gray(filteredImage);
```

```
    % Display the normalized filtered image
    imshow(filteredImageNormalized, 'Parent', app.UIAxes3);
    title(app.UIAxes3, ['Filtered with ', value]);




        end

        % Button pushed function: ErosionButton
        function erosion(app, event)
             % Convert the image to grayscale if it's RGB
    if size(app.img, 3) == 3
        grayImage = rgb2gray(app.img);
    else
        grayImage = app.img;
    end

    % Convert the image to binary
    binaryImage = imbinarize(grayImage, 0.5); % Adjust threshold as needed

    % Display the original binary image
    imshow(binaryImage, 'Parent', app.UIAxes3);
    title(app.UIAxes3, 'Original Binary Image');

% Invert the binary image to expand black pixels
binaryImage = ~binaryImage; % Invert the binary image

    % Create a structuring element
    se = strel('disk', 1 ); % Adjust size and angle as needed

    % Perform erosion
    erodedImage = imerode(binaryImage, se);
    erodedImage = imerode(erodedImage, se);
```

```
erodedImage = ~ erodedImage ;

    % Display the eroded image
    imshow(erodedImage, 'Parent', app.UIAxes2);
    title(app.UIAxes2, 'Erosion Image');
        end

        % Button pushed function: DilationButton
        function dilation(app, event)
            % Convert the image to grayscale if it's RGB
    if size(app.img, 3) == 3
        grayImage = rgb2gray(app.img);
    else
        grayImage = app.img;
    end

    % Convert the image to binary
    binaryImage = imbinarize(grayImage, 0.5); % Adjust threshold as needed

% Display the original binary image
    imshow(binaryImage, 'Parent', app.UIAxes3);
    title(app.UIAxes3, 'Original Binary Image');

% Invert the binary image to expand black pixels
binaryImage = ~binaryImage; % Invert the binary image

% Create a large structuring element
se = strel('disk', 3); % Disk with a large radius (e.g., 15)

% Perform dilation multiple times
dilatedImage = imdilate(binaryImage, se); % First dilation
dilatedImage = imdilate(dilatedImage, se); % Optional: Repeat if needed
```

```
% Invert the image back to restore original orientation
dilatedImage = ~dilatedImage;

    % Display the dilated image
    imshow(dilatedImage, 'Parent', app.UIAxes2);
    title(app.UIAxes2, 'Dilated Image');
        end

        % Button pushed function: BoundaryExtractionButton
        function BoundaryExtraction(app, event)

    % Convert the image to grayscale if it's RGB
    if size(app.img, 3) == 3
        grayImage = rgb2gray(app.img);
    else
        grayImage = app.img;
    end

    % Convert to binary using a specified threshold
    threshold = 0.4; % Adjust threshold between 0 and 1 if needed
    binaryImage = imbinarize(grayImage, threshold);

    % Create a structuring element for erosion
    se = strel('disk', 1); % Small circular structuring element

    % Perform erosion
    erodedImage = imerode(binaryImage, se);

    % Extract the boundary by subtracting the eroded image from the binary image
    boundaryImage = binaryImage - erodedImage;

    % Display the original binary image
    imshow(binaryImage, 'Parent', app.UIAxes3);
```

```
    title(app.UIAxes3, 'Original Binary Image');


    % Display the boundary extracted image

    imshow(boundaryImage, 'Parent', app.UIAxes2);

    title(app.UIAxes2, 'Boundary Extracted Image');

        end


        % Button pushed function: ThresholdingButton

        function Thresholding(app, event)


 % Convert the image to grayscale if it is RGB

    if size(app.img, 3) == 3

        grayImage = rgb2gray(app.img);

    else

        grayImage = app.img;

    end




    normalizedThreshold = graythresh(grayImage);

    optimalThreshold = normalizedThreshold * 255; % Convert to 8-bit scale
(0-255)


    % Perform thresholding

    segmentedImage = grayImage > optimalThreshold;


    % Display the segmented image

    imshow(segmentedImage, [], 'Parent', app.UIAxes2);

    title(app.UIAxes2, ['Segmented Image (Threshold = '
num2str(round(optimalThreshold)) ')']);


        end


        % Button pushed function: EdgeDetectionButton
```

```
        function SobelEdgeDet(app, event)

    % Convert the image to grayscale if it is RGB
    if size(app.img, 3) == 3
        grayImage = rgb2gray(app.img);
    else
        grayImage = app.img;
    end

    edges = edge(grayImage, 'Sobel');

    imshow(edges, [], 'Parent', app.UIAxes2);
    title(app.UIAxes2, 'Sobel Edge Detection Segmentation');
        end

        % Button pushed function: AdaptiveButton
        function AdaptiveThresholding(app, event)

    % Convert the image to grayscale if it is RGB
    if size(app.img, 3) == 3
        grayImage = rgb2gray(app.img);
    else
        grayImage = app.img;
    end

    % Use adaptive thresholding to segment the image
    adaptiveThresholdImage = adaptthresh(grayImage, 0.5); % Sensitivity
parameter '0.5'
    binaryAdaptiveImage = imbinarize(grayImage, adaptiveThresholdImage); % Apply
the threshold


    % Display Adaptive Thresholding Segmentation Result
```

```
    imshow(binaryAdaptiveImage, [], 'Parent', app.UIAxes2);

    title(app.UIAxes2, 'Adaptive Thresholding Segmentation');


        end


        % Button pushed function: PointDetectionButton

        function pointDet(app, event)


    % Convert the image to grayscale if it is RGB

    if size(app.img, 3) == 3

        grayImage = rgb2gray(app.img);

    else

        grayImage = app.img;

    end


    % -------------------- Point Detection using Laplacian of Gaussian
------------------------

    % Use the Laplacian of Gaussian (LoG) method for point detection

    pointDetectionImage = imfilter(double(grayImage), fspecial('log', 5, 1)); %
Kernel size: 5, Sigma: 1


    % -------------------- Segmentation using Thresholding
------------------------

    % Set a threshold for segmentation

    threshold = 0.1; % Adjust threshold value as needed

    segmentedImage = pointDetectionImage > threshold; % Apply thresholding to
the LoG response


    % Display Point Detection Result (LoG response)

    imshow(pointDetectionImage, [], 'Parent', app.UIAxes2);

    title(app.UIAxes2, 'Point Detection (LoG)');


    % Display Segmented Image (highlight points of interest)

    imshow(segmentedImage, [], 'Parent', app.UIAxes3);
```

```
    title(app.UIAxes3, 'Segmentation Based on Point Detection');

        end

        % Button pushed function: LineDetectionButton
        function LineDet(app, event)

    % Convert the image to grayscale if it is RGB
    if size(app.img, 3) == 3
        grayImage = rgb2gray(app.img);
    else
        grayImage = app.img;
    end

    % Step 1: Apply Sobel Filters in Specific Directions
    horizontalEdges = edge(grayImage, 'Sobel', [], 'horizontal'); % Detect
horizontal lines
    verticalEdges = edge(grayImage, 'Sobel', [], 'vertical'); % Detect vertical
lines

    % Combine horizontal and vertical edges
    combinedEdges = horizontalEdges | verticalEdges;

    % Step 2: Segment the Image Using the Line Mask
    segmentedImage = grayImage .* uint8(combinedEdges); % Use the mask directly
for segmentation

    % Display Detected Edges (Horizontal and Vertical)
    imshow(combinedEdges, [], 'Parent', app.UIAxes2);
    title(app.UIAxes2, 'Line Detection (Horizontal + Vertical)');

    % Display Segmented Image
    imshow(segmentedImage, [], 'Parent', app.UIAxes3);
    title(app.UIAxes3, 'Segmented Image Based on Detected Lines');
```