Rohit Raibagkar
gd4139

Advanced Digital Image Processing & Applications

ECE 7680
Winter 2018

Report of Image Processing Term Project

College of Engineering

Declaration: The project and the report are my own work
I contributed 100% of my own project.
Date: 04/19/2018

Abstract

I address importance of image enhancement in research. The image obtained from cameras or any CCD is corrupted with noise. The image may lose important features during acquisition. The best image quality is always obtained with focal zoom that digital zoom. But focal zooming is not always possible. The project and report is an attempt to focus on the different image processing and enhancement techniques. The techniques involve morphological transformation to process image in desired direction. The image also has some random noise. The morphological techniques also discuss dome of the noise reduction techniques. While processing image, it is important sometimes to extract feature or pattern. Image segmentation techniques such as point, line and edge detection are also experimented. Further, Detailed segmentation procedure is also explained and experimented. Noise is again big issue as mentioned. I experimented with morphological filters with real world images. The results are discussed. In appendix, complete MATLAB GUI preview, code and functions, GUI features are discussed for user reference. Also, results with approximate parameters are discussed at end of each chapter.

Table of Contents

1. Introduction

The image processing is very important in fields where output signal quality is very low. These fields include bio-medical, astronomy, Industry, Printing Industry. The image is nothing but array of values obtained from array of sensors. But due to some noise, corruption, interference, focal mis-alignments or many other reasons, Image quality is deteriorated. To extract the important features and enhance desired features, Image Processing is very important. The image processing involves right from operation on co-ordinates of the pixel to frequency of pixel. We are going to study the results of project in steps.

The 2nd chapter presents the problem statement assigned. 3rd chapter focuses on detailed discussion for methods in morphological operations and its results on various images. The 4th chapter discusses morphological transformations for noise reduction in detail with results. Next, 5th chapter discusses image segmentation with point, line and edge detection. Next, 6th chapter introduces Thresholding operation for noise reduction. 7th chapter deals with combination of all the segmentation techniques to segment objects. 8th chapter delves more in details of noise reduction analysis. 9th chapter deals with band filtering of images. 10th chapter explains the results on real world noise reduction in images. Appendix focuses on MATLAB GUI features and function.

2. Problem Statement

The approach towards the solution of morphological image processing and image segmentation is below:

- Creating MATLAB GUI and class.

I have decided to use MATLAB as my image processing and algorithm development software. Creating MATLAB GUI, creating `imProcessing` class and coding functions for image processing was first task.

- Building Morphological image processing functions.

Building morphological operations such as erosion, dilation, hit-or-miss transform, boundary extraction, convex hull, thinning-thickening, region filling, skeletons pruning and watershed segmentation. These functions were built and combined with each other to perform advanced operations.

- Using morphological techniques for noise reduction.

Morphological functions such as top-hat and bottom-hat, opening, closing are built by combining the basic morphological functions.

- Extract objects of interest using morphological techniques.

Objects of interest are extracted using morphological gradient, boundary extraction, region of connected components.

- Extract objects of interests using watershed techniques.
- Other features and applications to the given images and images of interest.
- Processing the given images with specific issues.

3. Morphological image processing

3.1 Erosion

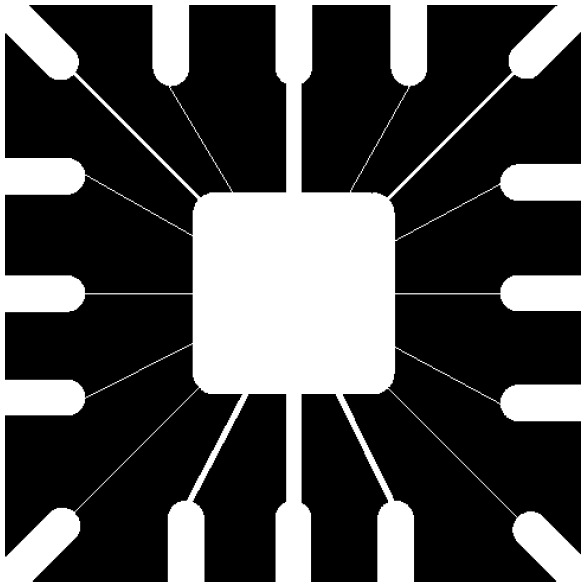
The Erosion is process in which Pixels beyond the image border are assigned the maximum value afforded by the data type. The Algorithm is given by:

g[x,y] = min{W{ f[x,y] }} = erode(f, W).

The execution in GUI is loading image in window and processing it in Erosion. For loading image, following function is used:

```
function imProcess = getImg (imProcess) % this function helps to get image...

            [imProcess.fileName, imProcess.pathName, imProcess.filterIndex] =
uigetfile('*.*', 'Please Select Input Image');
            imProcess.img = imread(imProcess.fileName);  % the command takes image and
stores in to global variable..
            imProcess.imgInfo = imfinfo(imProcess.fileName);
            imProcess.imgProcessed = double(zeros(size(imProcess.img),
class(imProcess.img)));
            clc;
            %disp(imProcess.imgInfo);

end
```

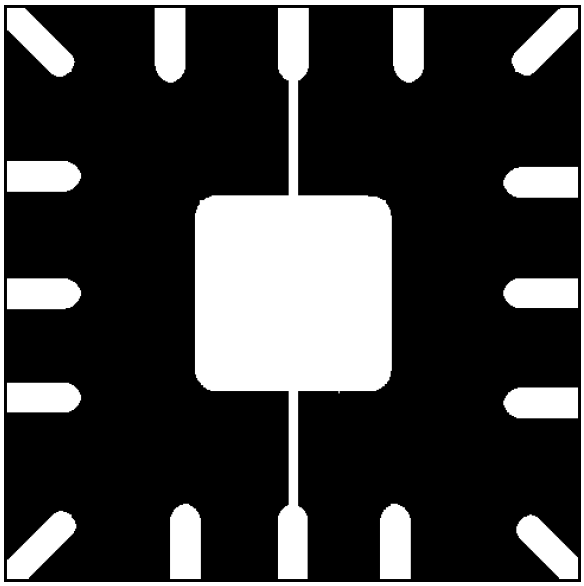
The function will load image of Binary Wirebond mask as:



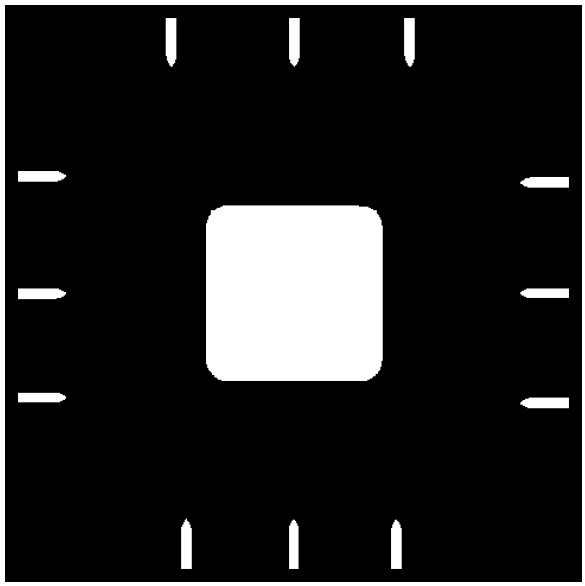
The erosion is given by following function:

```
      function imProcess = erosion(imProcess)

            % This function performs erosion of images....

            %imProcess = imProcess.getKernelSize; % gets size of the kernel...
            boxKernel = ones(imProcess.kernelSize); % generates box kernel of obtained
size...
            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]); % pads new image...
            %boxKernel = rot90(boxKernel, 2);
            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));
                        imProcess = imProcess.orderStatFiltParamtr(boxKernel .*
double(piece));
                        imProcess.imgProcessed(x, y, z) = imProcess.minIntensity;

                    end
                end
            end

            if imProcess.imgInfo.BitDepth == 1; % checks if given image is binary
image...
                imProcess.imgProcessed = logical(imProcess.imgProcessed); % converts
uint8 array to logical array.
            end

        end
```

The Result:



Erosion by 11 X 11 SE



Erosion by 45 X 45 SE

3.2 Dilation

The Dilation is given by process in which Pixels beyond the image border are assigned the minimum value afforded by the data type.

The Algorithm is given by:

g[x,y] = max{W{ f[x,y] }} = dilate(f, W).

Again, the function

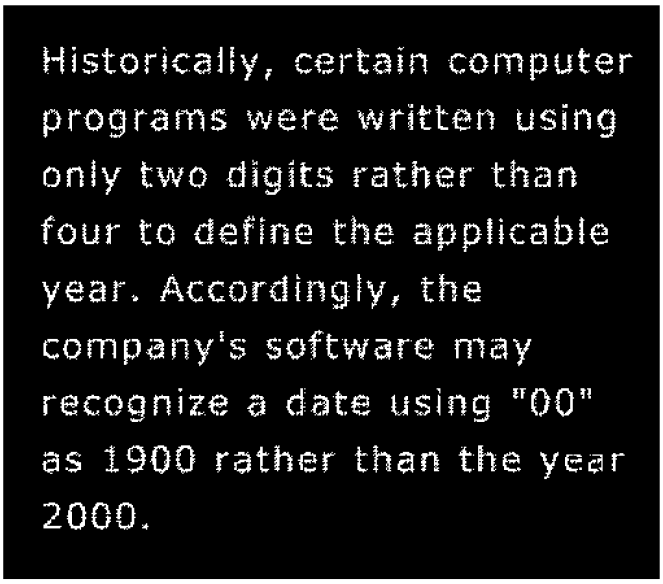
```
function imProcess = getImg (imProcess) % this function helps to get image...
```

Will load the image.

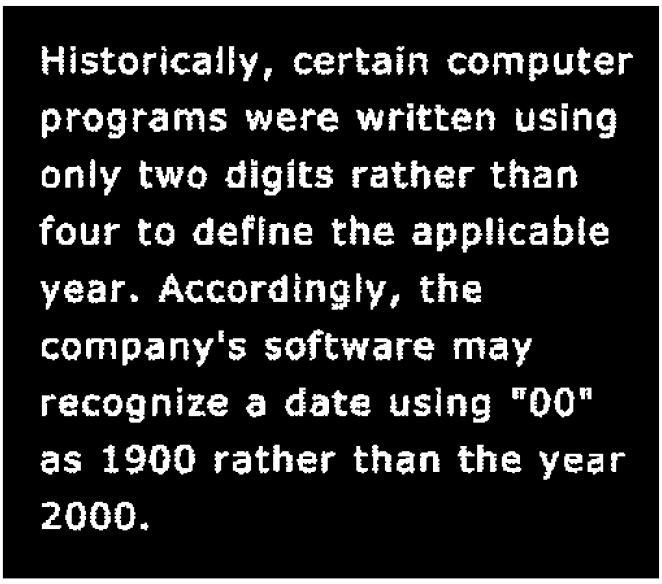
The dilation operation is given by following code:

```
        function imProcess = dilation (imProcess)

            % This function performs dilation on the images....

            %imProcess = imProcess.getKernelSize; % gets size of the kernel...
            boxKernel = ones(imProcess.kernelSize); % generates box kernel of obtained
size...
            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]); % pads new image...
            boxKernel = rot90(boxKernel, 2);
            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));
                        imProcess = imProcess.orderStatFiltParamtr(boxKernel .*
double(piece));
                        imProcess.imgProcessed(x, y, z) = imProcess.maxIntensity;

                    end
                end
            end

            if imProcess.imgInfo.BitDepth == 1; % checks if given image is binary
image...
                imProcess.imgProcessed = logical(imProcess.imgProcessed); % converts
uint8 array to logical array.
            end

        end
```

The result of the dilation operation is given by:

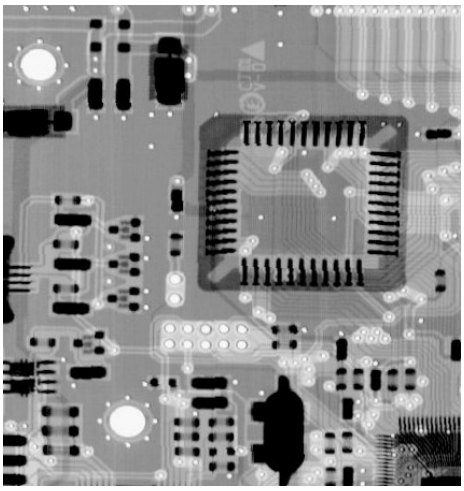


Broken text 1019 x 889 pixels binary image

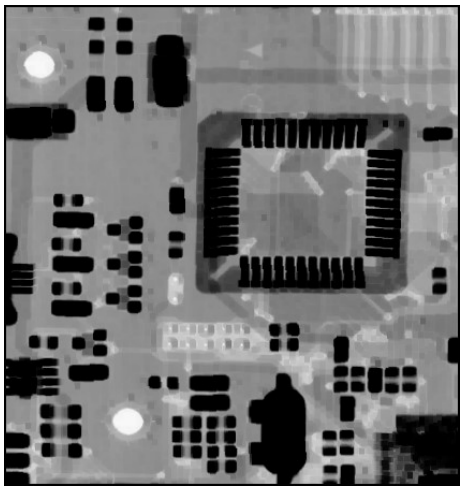


Dilation by 3 x 3 SE

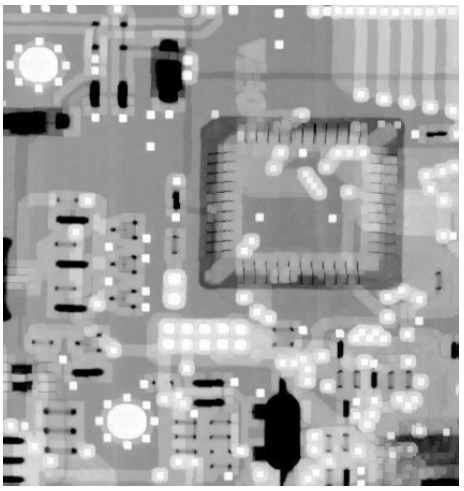
Similarly, dilation and erosion operation on grayscale image are given by:



448 X 425 Gray-scale X-ray of circuit board



Erosion by 5 x 5 SE



Dilation by 5 x 5 SE

3.3 Opening

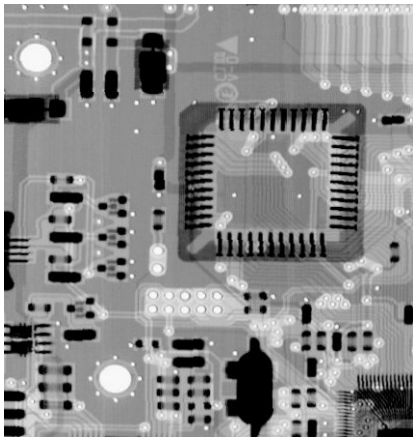
The Opening of an image is given by the following code:

```
      function imProcess = opening (imProcess)

            % This function performs both morphological and grayscale
            % opening of images.

            imProcess.auxImg = imProcess.img;    % storing main image in auxiliary image
variable...
            imProcess = imProcess.erosion;       % performing erosion on main image...
            imProcess.img = imProcess.imgProcessed; % storing processed image into main
image property for next dilation operation...
            imProcess = imProcess.dilation;      % performing dlation operation...
            imProcess.img = imProcess.auxImg;    % storing original image in main image
property...
            %imProcess = imProcess.showOriginal;
            %figure;
            %imProcess = imProcess.showProcessed;

        end
```

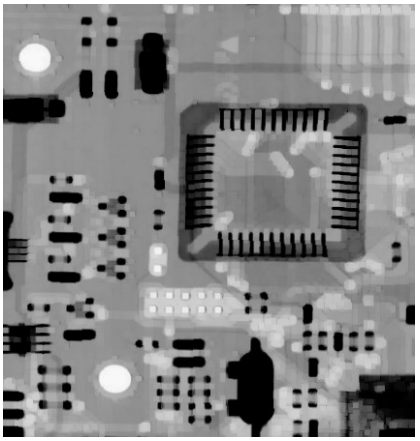
The result of the opening is given by:

Noisy fingerprint 315 x 228 pixels, binary

Opening by 3 x 3 SE



448 X 425 Gray-scale X-ray of circuit board



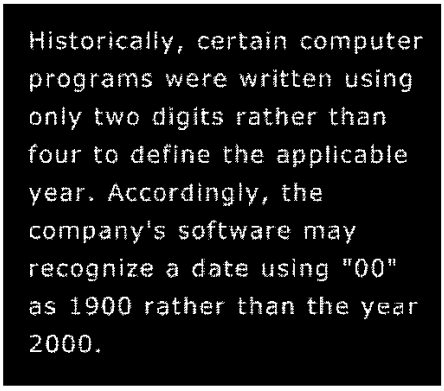
Opening by 5 x 5 SE

3.4 Closing

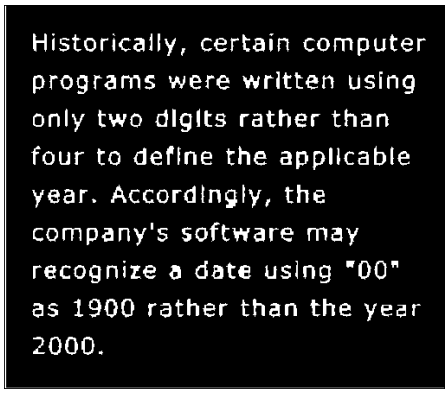
The closing of the image is given by following code:

```
        function imProcess = closing (imProcess)

            % This function performs both morphological and grayscale
            % closing of images.

            imProcess.auxImg = imProcess.img;    % storing main image in auxiliary
property...
            imProcess = imProcess.dilation;      % performing dilation on main
image...
            imProcess.img = imProcess.imgProcessed; % storing processed image
into main image property for next erosion operation...
            imProcess = imProcess.erosion;       % performing erosion operation...
            imProcess.img = imProcess.auxImg;    % storing original image in main
image property...

        end
```

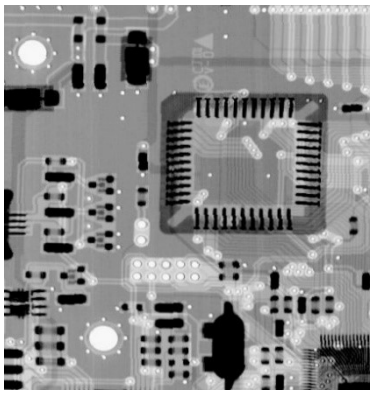
The result of the Closing operation is given by:



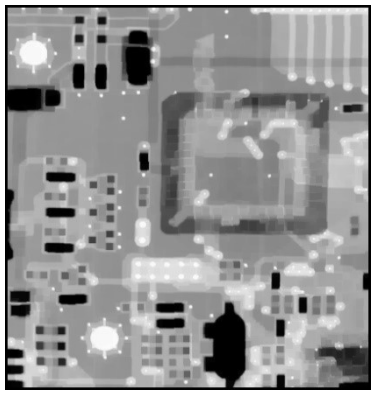
Broken text 1019 x 889 pixels binary image



Closing by 5 x 5 se



448 X 425 Gray-scale X-ray of circuit board



Closing by 7 x 7 SE

3.5 Boundary Extraction

The Boundary Extraction of the image is given by following code:

```
function imProcess = boundaryExtraction (imProcess)

      % This function extracts boundary from images.

      imProcess = imProcess.erosion;
      imProcess.imgProcessed = double(imProcess.img) - double(imProcess.imgProcessed);

end
```

The result of the Boundary Extraction operation is given by:

Lincoln : 221 x 269 binary

Boundary extraction by 3 x 3 SE

Boundary extraction by 7 x 7 SE

Cameraman 512 x 512 pixel, grayscale

Boundary extraction by 3 x 3 SE

Boundary extraction by 7 x 7 SE

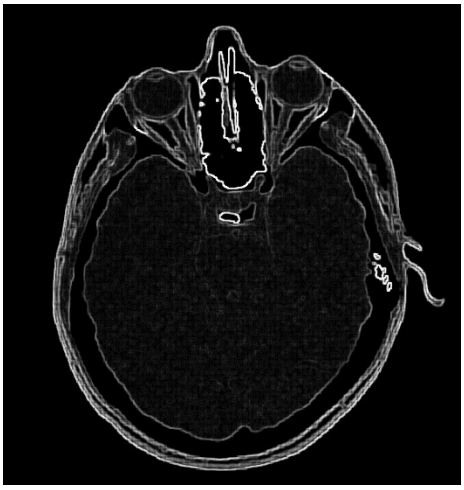
3.6 Morphological Gradient

The Morphological Gradient of the image is given by following code:

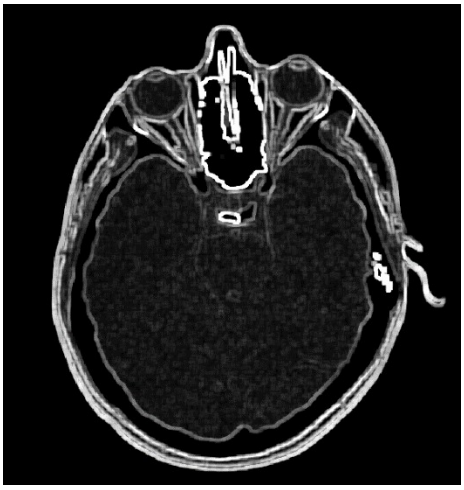
```
function imProcess = morphologicalGradient (imProcess)

      % This function gets morphological gradient of an image...

      imProcess = imProcess.dilation;
      imProcess.auxImg = imProcess.imgProcessed;
      imProcess = imProcess.erosion;
      imProcess.imgProcessed = imProcess.auxImg - imProcess.imgProcessed;

end
```

The result of the Morphological Gradient Operation operation is given by:

Head CT scan 512 x 512 pixels



Morph. Gradient using 3 x 3 SE



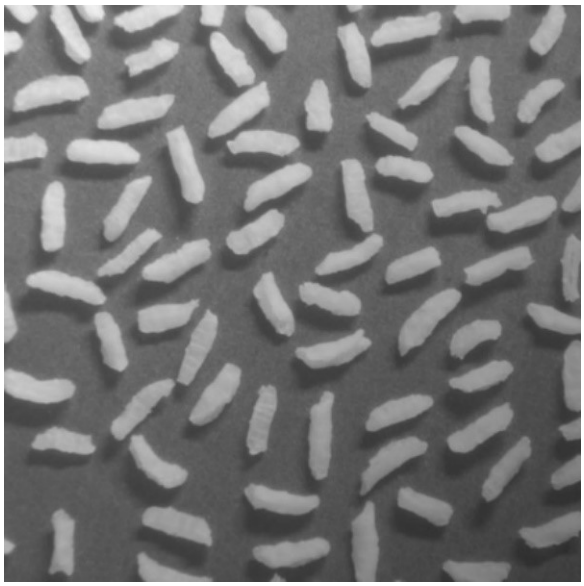
Morph. Gradient using 5 x 5 SE

3.7 Top-hat Transform

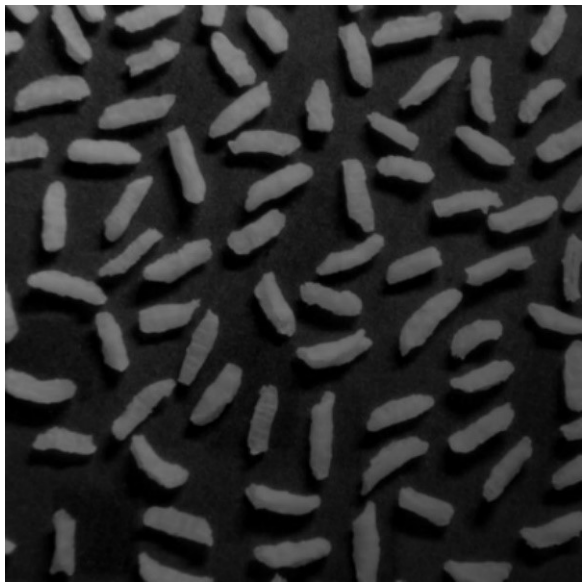
The Top-Hat transform of the image is given by following code:

```
function imProcess = topHatTransform (imProcess)

      % this function performs the tophat transformation of an
      % image...

      imProcess = imProcess.opening;
      imProcess.imgProcessed = double(imProcess.img) - imProcess.imgProcessed;

end
```

The result of the Top-Hat Operation operation is given by:



Rice shaded 600 x 600 pixels



Top-hat by 81 x 81 SE

3.8 Bottom-Hat Transform

The Bottom-Hat transform of the image is given by following code:

```
function imProcess = bottomHatTransform (imProcess)

      % this function performs the bottomhat transformation of an
      % image...

      imProcess = imProcess.closing;
      imProcess.imgProcessed = imProcess.imgProcessed - double(imProcess.img);

end
```

The result of the Bottom-Hat Operation operation is given by:

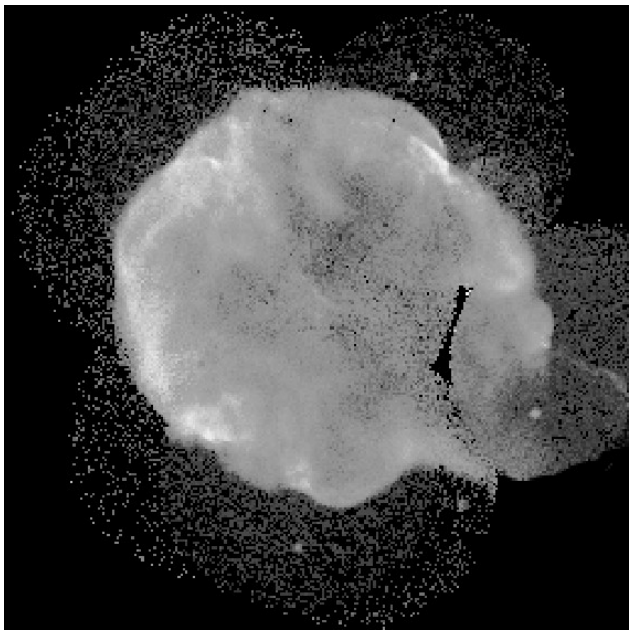
Cameraman 512 x 512 Pixels

Bottom-Hat by 13 x 13 SE

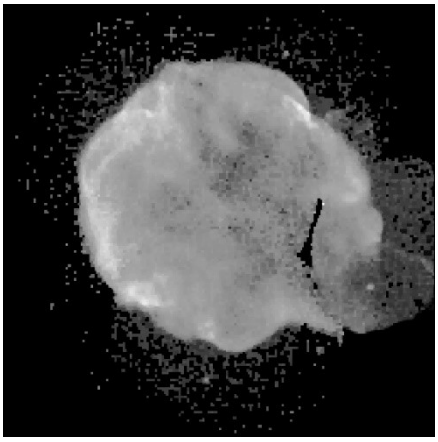
4. Morphological Processing for Noise Reduction

The Morphological noise reduction of the image by smoothing is given by following code:

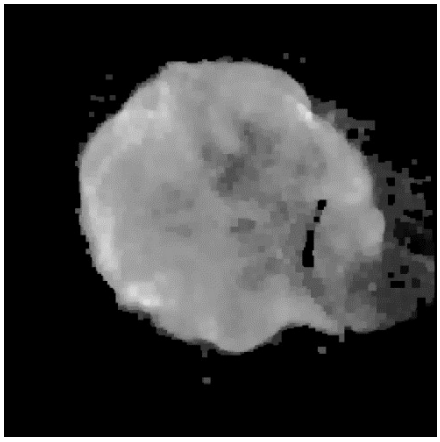
```
function imProcess = morphSmoothing (imProcess)

      % The algorithm performs Morphological smoothin on the given
      % image. First it performs opening on the given input image,
      % then it performs closing on the result of opening.

      imProcess = imProcess.opening;
      imProcess.img = imProcess.imgProcessed;
      imProcess = imProcess.closing;


end
```



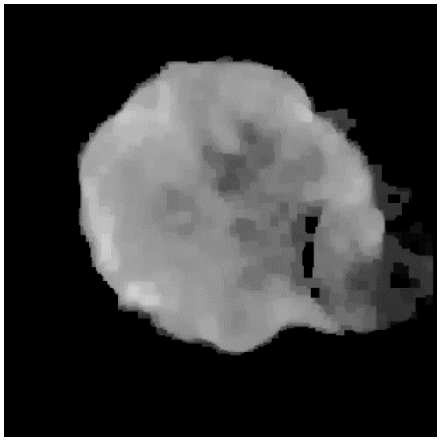
Cygnus loop supernova 566 X 566
Corrupted by noise



Morphological smoothing with S.E. of radius 1 (3 X 3)



Morphological smoothing with S.E. of radius 3 (7 X 7)



Morphological smoothing with S.E. of radius 5 (11 X 11)

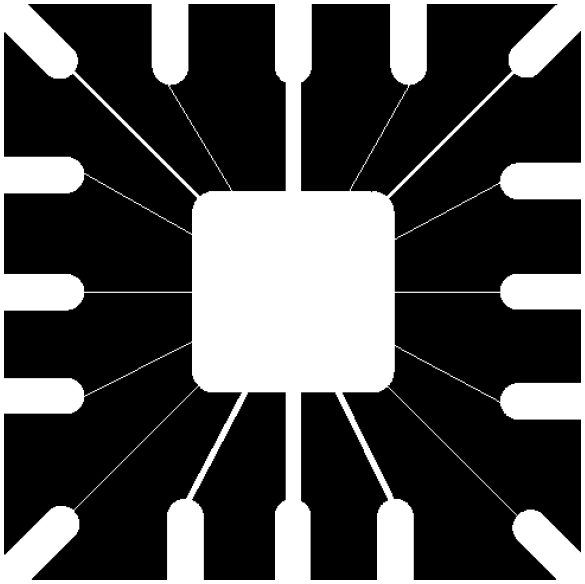
5. Image Segmentation 1 : Point, Line and Edge Detection

5.1 Line Detection: Horizontal, +45^0, Vertical, -45^0

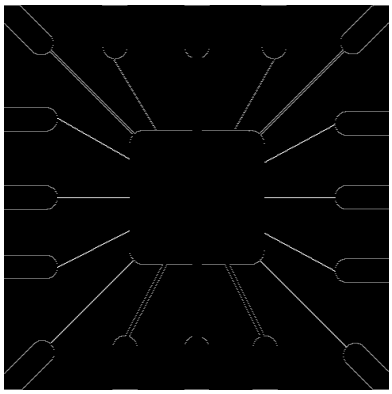
The Line Detection of the image is given by following code:

```
function imProcess = lineDetection (imProcess)

      % The function detects horizontal, vertical, +45 degree, -45
      % degree angle lines in the input image.

      lineAngle = {'Horizontal Line Detection', '+45 degree Line Detection', 'Vertical
Line Detection', '-45 degree Line Detection'};
      [s, v] = listdlg('ListString', lineAngle, 'SelectionMode', 'single',...
      'ListSize', [250 100], 'InitialValue', [1], 'Name', 'Line Detection Kernel',...
      'PromptString', 'Please select type of Kernel for line detection :',...
      'OKString', 'Select', 'CancelString', 'Cancel');

      if v == 1;

            switch s;
                  case 1;
                        boxKernel = [-1 -1 -1; 2 2 2; -1 -1 -1];
                  case 2;
                        boxKernel = [2 -1 -1; -1 2 -1; -1 -1 2];
                  case 3;
                        boxKernel = [-1 2 -1; -1 2 -1; -1 2 -1];
                  case 4;
                        boxKernel = [-1 -1 2; -1 2 -1; 2 -1 -1];
                  otherwise
                        % do nothing
            end
      end

      newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2 (size(boxKernel, 2)
- 1)/2]);

      for z = 1:1:size(newImage, 3);
            for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                  for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
                  2)-1));
                        imProcess = imProcess.sumOfArray((boxKernel) .* double(piece));
                        imProcess.imgProcessed(x,y,z) = uint8(imProcess.arraySum);
                  end
            end
      end

      if imProcess.imgInfo.BitDepth == 1; % checks if given image is binary image...
                  imProcess.imgProcessed = logical(imProcess.imgProcessed); % converts
uint8 array to logical array.
      end

end
```

The result is given by:

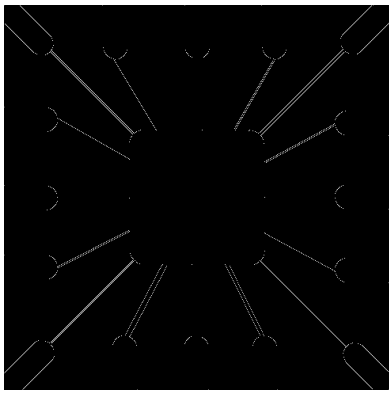


486 X 486 Wire bond Mask (Binary)



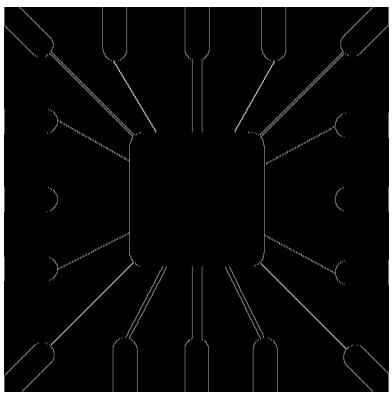
Horizontal line detection with Kernel

−1	−1	−1
2	2	2
−1	−1	−1



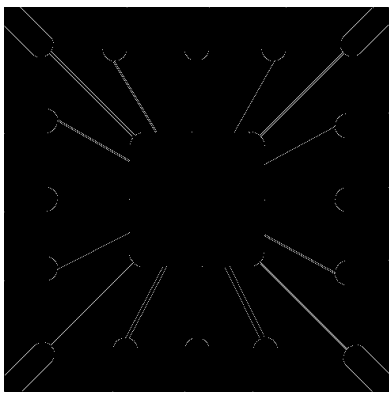
+45^0 line detection with Kernel

2	−1	−1
−1	2	−1
−1	−1	2



Vertical line detection with Kernel

−1	2	−1
−1	2	−1
−1	2	−1



- 45^0 line detection with Kernel

−1	−1	2
−1	2	−1
2	−1	−1

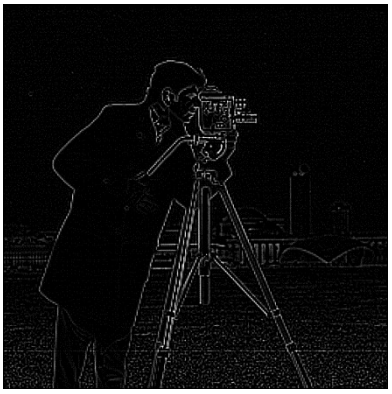
5.2 Laplacian: Type 1, Type 2, Type 3, Type 4

The Laplacian Operation on the Image is gien by the following code:

```
function imProcess = laplacianOperator (imProcess)

      imProcess = imProcess.createLaplacianKernelFilter;
      newImage = padarray(imProcess.img, [(size(imProcess.laplacianKernel, 1) - 1)/2
(size(imProcess.laplacianKernel, 2) - 1)/2]);
      imProcess.laplacianKernel = rot90(imProcess.laplacianKernel, 2);

      for z = 1:1:size(newImage, 3);
            for x = 1:1:(size(newImage, 1) - (size(imProcess.laplacianKernel, 1)-1));
                  for y = 1:1:(size(newImage, 2) - (size(imProcess.laplacianKernel, 2)-
            1));
                        piece = newImage(x:x+(size(imProcess.laplacianKernel, 1)-1),
                  y:y+(size(imProcess.laplacianKernel, 2)-1));
                        imProcess = imProcess.sumOfArray(imProcess.laplacianKernel .*
                  double(piece));
                        imProcess.imgProcessed(x,y,z) = imProcess.arraySum;
                  end
            end
      end

end
```

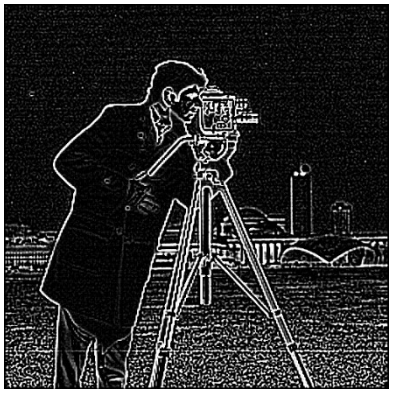
The result of Type 1 Laplacian is given by:

Cameraman 512 x 512



Type 1 Laplacian filter : 3 x 3

```
1      1      1
1     -8      1
1      1      1
```



Type 1 Laplacian filter : 5 x 5

```
1      1      1      1      1
1      1      1      1      1
1      1    -24      1      1
1      1      1      1      1
1      1      1      1      1
```

The result of Type 2 Laplacian is given by:

Cameraman 512 x 512

Type 2 Laplacian filter : 3 x 3

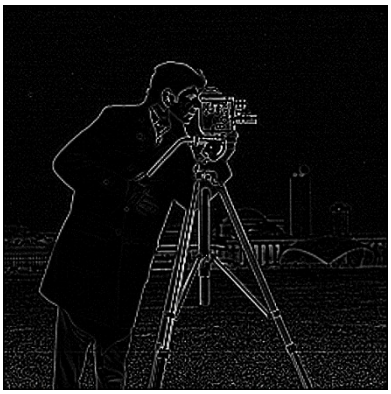
$$\begin{matrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{matrix}$$

Type 2 Laplacian filter : 5 x 5

$$\begin{matrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & 24 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{matrix}$$

The result of Type 3 Laplacian is given by:

Cameraman 512 x 512



Type 3 Laplacian filter : 5 x 5

$$\begin{matrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & -8 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{matrix}$$

Type 3 Laplacian filter : 7 x 7

$$\begin{matrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & -12 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{matrix}$$

The result of Type 4 Laplacian is given by:

Cameraman 512 x 512

Type 4 Laplacian filter : 5 x 5

$$\begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ -1 & -1 & 8 & -1 & -1 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}$$

Type 4 Laplacian filter : 7 x 7

$$\begin{bmatrix} 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & -1 & -1 & 12 & -1 & -1 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{bmatrix}$$

5.3 Gradients: Robert's Cross, Prewitt Operator, Sobel Operator

The Gradient Operation on the Image is gien by the following code:

```
function imProcess = gradientOperator (imProcess)
      % The function operates gradient on the given image. The
      % gradient operator extracts horizontal component gX, vertical
      % component gY, vector and angle alpha.

      gradOperator = {'Roberts Cross Gradient', 'Prewitt Operator', 'Sobel Operator'};
            [s, v] = listdlg('ListString', gradOperator, 'SelectionMode', 'single',...
            'ListSize', [250 100], 'InitialValue', [1], 'Name', 'Gradient Operator',...
            'PromptString', 'Please select Gradient operator :',...
            'OKString', 'Select', 'CancelString', 'Cancel');

      if v == 1;
            switch s;
                  case 1;
                        gradX = [-1 0; 0 1];    gradY = [0 -1; 1 0];
                  case 2;
                        gradX=[-1 -1 -1; 0 0 0; 1 1 1]; gradY = [-1 0 1; -1 0 1; -1 0 1];
                  case 3;
                        gradX=[-1 -2 -1; 0 0 0; 1 2 1]; gradY = [-1 0 1; -2 0 2; -1 0 1];
                  otherwise
                        % do nothing
            end
      end

      boxKernel = ones(3);
      imProcess.arrayOfAlpha = double(zeros(size(imProcess.img)));
      newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2 (size(boxKernel, 2)
      - 1)/2]);

      for z = 1:1:size(newImage, 3);
            for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                  for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));

                        if s == 1
                              imProcess = imProcess.sumOfArray(double(piece(2:3,
                        2:3)).*gradX);    gX = imProcess.arraySum;
                              imProcess = imProcess.sumOfArray(double(piece(2:3,
                        2:3)).*gradY);    gY = imProcess.arraySum;

                        elseif s == 2 || s == 3
                              imProcess = imProcess.sumOfArray(double(piece).*gradX);
                        gX = imProcess.arraySum;
                              imProcess = imProcess.sumOfArray(double(piece).*gradY);
                        gY = imProcess.arraySum;
                        end

                        imProcess.imgProcessed(x, y, z) = sqrt((gX^2)+(gY^2));
                        imProcess.arrayOfAlpha(x, y, z) = atan(gY/gX);

                  end
            end
      end

end
```

The result of Gradient operation is given by:

Building 600 x 600 pixels

Robert's cross gradients

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

Prewitt operator

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

Sobel operator

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

5.4 Kirsch Compass Mask

The code for Kirsch compass mask is given by:

```
function imProcess = kirschCompassMask (imProcess)
      kirsch = {'North Direction Mask', 'North-West Direction Mask',...
                'West Direction Mask', 'South-West Direction Mask'...
                'South Direction Mask', 'South-East Direction Mask',...
                'East Direction Mask', 'North-East Direction Mask'};

      [s, v] = listdlg('ListString', kirsch, 'SelectionMode', 'single',...
            'ListSize', [250 120], 'InitialValue', [1], 'Name', 'Kirsch Compass Mask',...
                'PromptString', 'Please select Compass mask :',...
                'OKString', 'Select', 'CancelString', 'Cancel');

      if v == 1;

            switch s;

                  case 1; % North directional mask is selected...
                        boxKernel = [-3 -3 5; -3 0 5; -3 -3 5];
                  case 2; % North-West directional mask is selected...
                        boxKernel = [-3 5 5; -3 0 5; -3 -3 -3];
                  case 3; % West direction mask is selected...
                        boxKernel = [5 5 5; -3 0 -3; -3 -3 -3];
                  case 4; % South-West direction mask is selected...
                        boxKernel = [5 5 -3; 5 0 -3; -3 -3 -3];
                  case 5; % South direction mask is selected...
                        boxKernel = [5 -3 -3; 5 0 -3; 5 -3 -3];
                  case 6; % South-East direction mask is selected...
                        boxKernel = [-3 -3 -3; 5 0 -3; 5 5 -3];
                  case 7; % East direction mask is selected...
                        boxKernel = [-3 -3 -3; -3 0 -3; 5 5 5];
                  case 8; % North-East direction mask is selected...
                        boxKernel = [-3 -3 -3; -3 0 5; -3 5 5];

            otherwise
                  % do nothing
            end
      end

      newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2 (size(boxKernel, 2)
- 1)/2]);

      for z = 1:1:size(newImage, 3);
            for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                  for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));

                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
                  2)-1));
                        imProcess = imProcess.sumOfArray(double(piece).*boxKernel);
                        imProcess.imgProcessed(x, y, z) = imProcess.arraySum;

                  end
            end
      end

end
```

The result of Kirsch compass operation is given by:

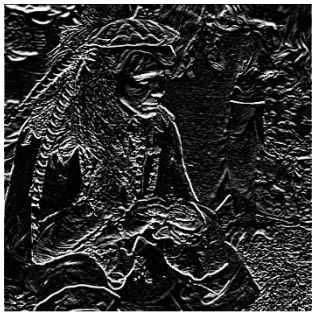
Pirate : 512 x 512 pixels

North Compass

−3	−3	5
−3	0	5
−3	−3	5

North-West Compass

−3	5	5
−3	0	5
−3	−3	−3



West Compass

5	5	5
−3	0	−3
−3	−3	−3

South-West Compass

5	5	−3
5	0	−3
−3	−3	−3

South Compass

5	−3	−3
5	0	−3
5	−3	−3

South-East Compass

−3	−3	−3
5	0	−3
5	5	−3

East Compass

−3	−3	−3
−3	0	−3
5	5	5

North-East Compass

−3	−3	−3
−3	0	5
−3	5	5

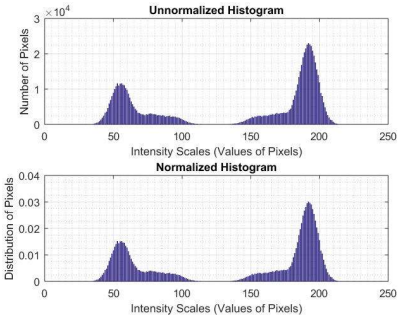
6. Thresholding for Noise Reduction

The code for Kirsch compass mask is given by:

```
function imProcess = basicGlobalThresholding (imProcess)
	% The function performs global thresholding operation on the
	% input image.

	[imProcess, avgIntensity] = imProcess.avgImgIntensity(imProcess.img);

	promptDlg = sprintf('Average Image intensity is %f.\n\nSelect an initial estimate for the
global threshold:', avgIntensity);

	prompt = {promptDlg, 'Enter delta threshold:'};
	title = 'Global Thresholding Parameters';
	num_lines = [1 60];
	default_ans = {'128', '10'};
	options.Resize='on';    options.WindowStyle='normal';   options.Interpreter='tex';
	thresholdParameters = inputdlg(prompt, title, num_lines, default_ans, options);
	initialThreshold = str2double(thresholdParameters{1,:});
	delThreshold = str2double(thresholdParameters{2,:});

	delTa = 255; newThreshold = 0;
	gH = double(zeros(size(imProcess.img)));
	gL = double(zeros(size(imProcess.img)));

	while delTa > delThreshold

		for z = 1:1:size(imProcess.img, 3)
			for x = 1:1:size(imProcess.img, 1)
				for y = 1:1:size(imProcess.img, 2)

					if imProcess.img (x, y, z) > initialThreshold;
						gH(x, y, z) = imProcess.img(x, y, z);
					elseif imProcess.img (x, y, z) <= initialThreshold;
						gL(x, y, z) = imProcess.img(x, y, z);
					end
				end
			end
		end

		imProcess = imProcess.sumOfArray(gH);
		meanH = imProcess.arraySum/(size(gH, 1)*size(gH, 2));
		imProcess = imProcess.sumOfArray(gL);
		meanL = imProcess.arraySum/(size(gL, 1)*size(gL, 2));
		newThreshold = (meanH + meanL)/2;
		delTa = abs(initialThreshold - newThreshold)
		initialThreshold = newThreshold;
	end

	for z = 1:1:size(imProcess.img, 3)
		for x = 1:1:size(imProcess.img, 1)
			for y = 1:1:size(imProcess.img, 2)
				if imProcess.img (x, y, z) > newThreshold;

					imProcess.imgProcessed(x, y, z) = 255;

				elseif imProcess.img (x, y, z) <= newThreshold;

					imProcess.imgProcessed(x, y, z) = 0;
				end

			end
		end
	end

end
```

The result of Thresholding operation is given by:

Noisy fingerprint



Its histogram

Segmentation using global thresholding. Notice that noise is completely removed.

7. Segmentation Using Morphological Watersheds

The segmentation using watershed algorithm is carried out by step-by-step sequence explained below:

1. Find out the Gradient Magnitude of the image. The gradient magnitude can be found out by procedure explained in section 5.3. The function

   ```
   function imProcess = gradientOperator (imProcess)
   ```

   Gives the gradient of the image.

2. Find the opening of the image using SE of any desired radius. The Opening operation is explained in section 3.3. The function

   ```
   function imProcess = opening (imProcess)
   ```

   Gives Opening of the image. Here SE of 41 X 41 is used. The opening by reconstruction can be carried out using Reconstruction option in the function.

3. Find the closing of the opening operation carried out in step 2. The closing is explained in section 3.4. The function

   ```
   function imProcess = closing (imProcess)
   ```

   Gives Closing of the image. The Closing by reconstruction can be carried out using Reconstruction option in the function.

4. Find out the regional maxima of the image. The regional maxima can be found out by the thresholding operation on the image. If the maxima is not accurate, then find out the modified maxima of the image with closing of maxima followed by erosion of the maxima and opening of the previous result. The erosion process is explaind in section 3.1. The function

   ```
   function imProcess = erosion(imProcess)
   ```

   Gives Erosion of the image.

5. Find out the watershed ridges of the image. The function

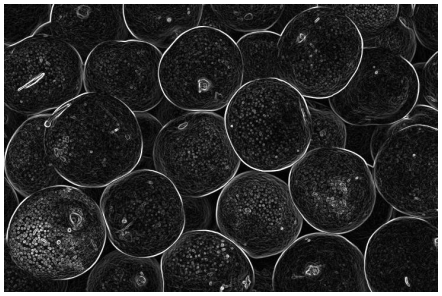
   ```
   function imProcess = getWatershed(imProcess)
   ```

   Gives watershed ridgs of the image.

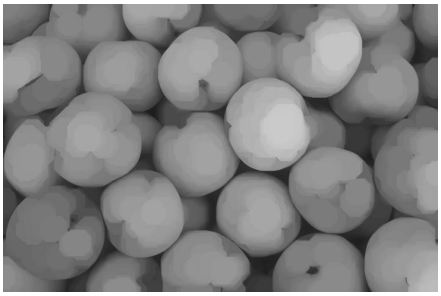
6. Combine the original image, watershed ridge lines, regional maxima together. The three things can be combined together or can be infused in each slice of the image as RGB Component with adjusted alpha parameter for clear visualization.

The segmentation operation result is given below:

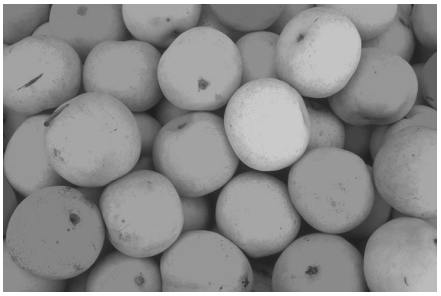
Grayscale image of pears.



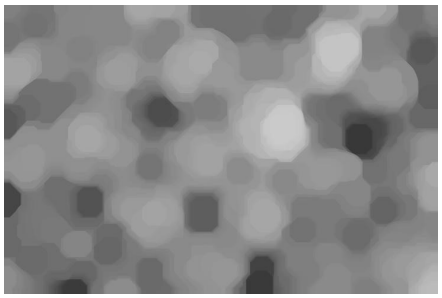
1. Gradient of image



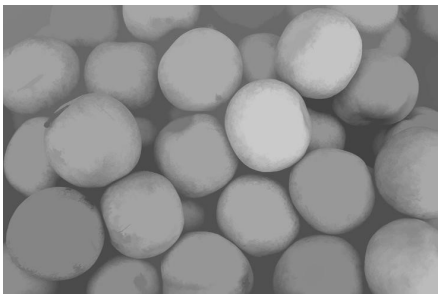
2. Opening with SE of radius 20



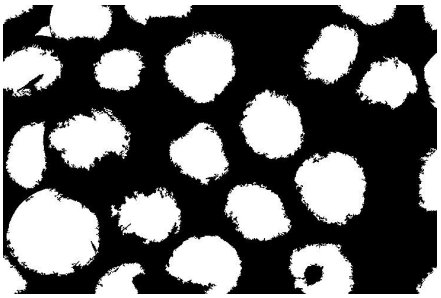
3. Opening by reconstruction



4. Closing of opening



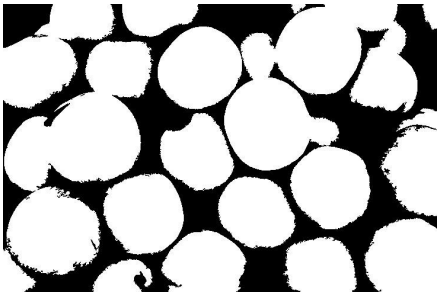
5. Closing of opening by reconstruction



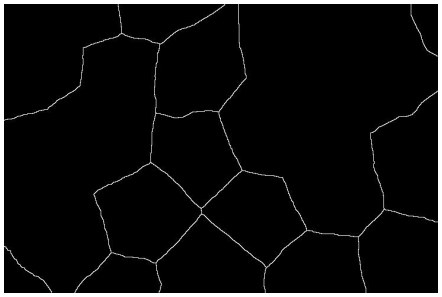
6. Regional Maxima of (5)

7. Regional maxima superimposed on original image

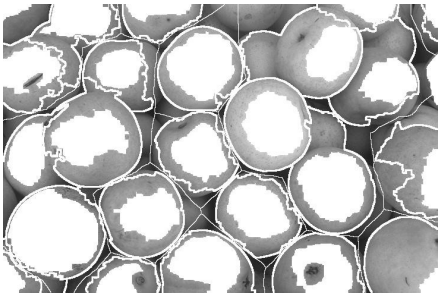
8. Modified maxima superimposed on original image



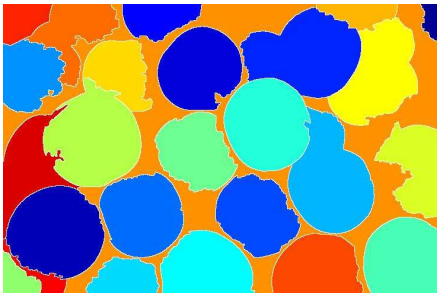
9. Opening & Closing of regional maxima with thresholding



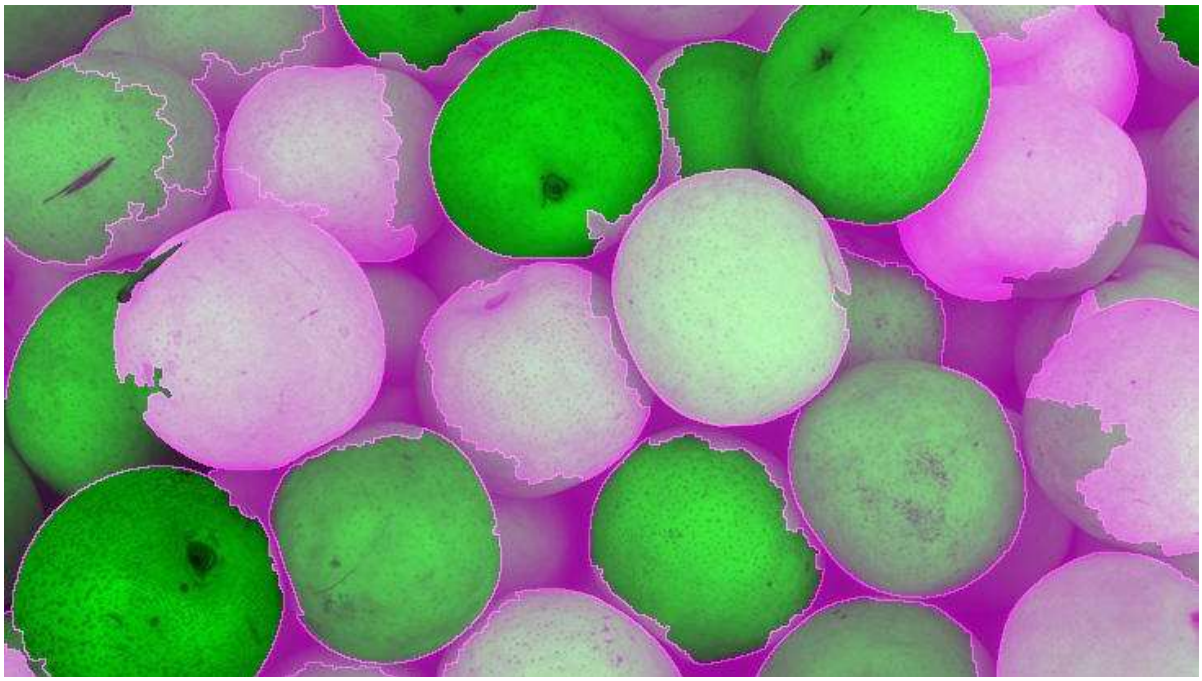
10. Ridge lines watershed



11. Combining original, Ridge lines, Maxima



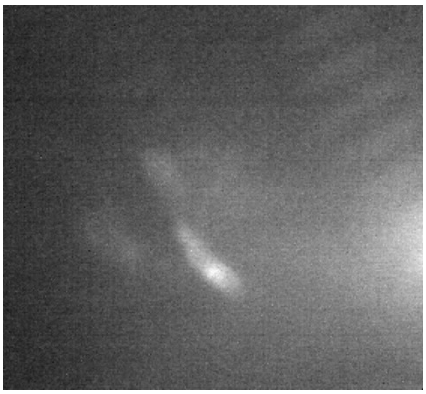
12. Colored watershed objects



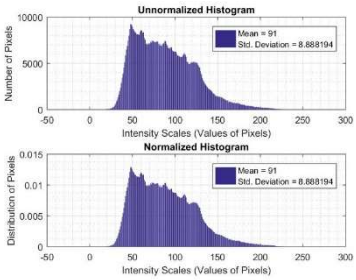
Result of Image Segmentation using morphological watersheds

8. Noise Reduction Problems

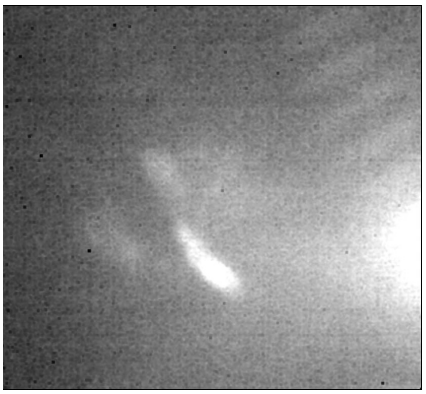
8.1 Image 1



Original Image

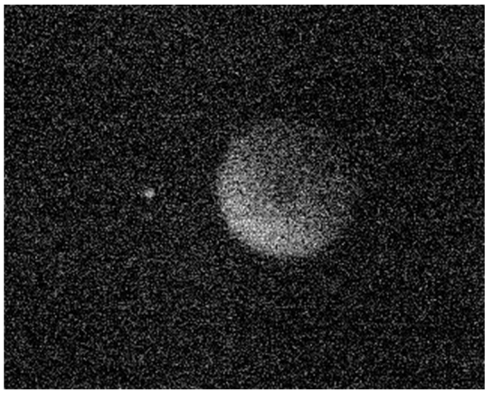


Histogram

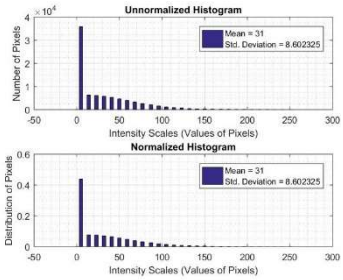


Result after Smoothing by box filter of size 3 X 3
Sharpening by Type 4 Laplacian of size 3 X 3
Noise reduction by Opening with SE of 11 X 11 followed by closing by same SE of 11 X 11
Power transform with K = 1.25 and Gamma = 1.025

8.2 Image 2



Original Image

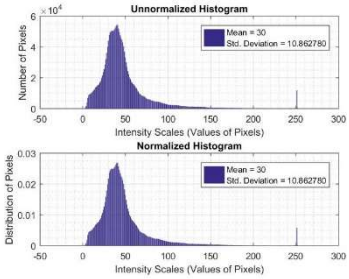


Histogram

Result after Smoothing by gaussian filter of size 5 X 5 with K : 2 and σ : 2
Sharpening by Type 3 Laplacian of size 3 X 3 Followed by Top-Hat transform subtracted from image.

8.3 Image 3

Original Image

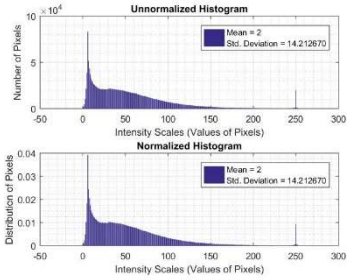


Histogram

Result after Thresholding with moving averages intensity value of 65.
Power transform with K = 1.25 and Gamma = 1.125
Noise reduction by morphological smoothing.

8.4 Image 4

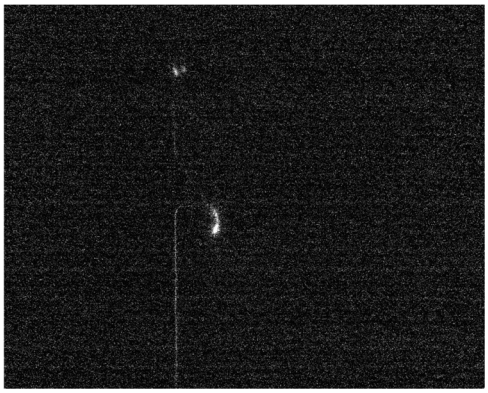
Original Image



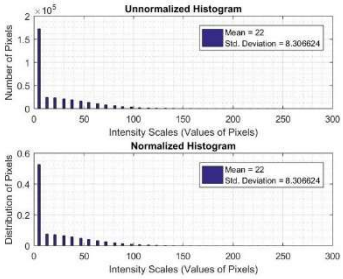
Histogram

Result after Thresholding by Otsu's method with intensity value of 90.
Smoothing with Gaussian kernel of size 5 X 5 with K = 1.25 and σ = 1.25
Opening of image followed by closing of the result of opening
Power transform with K = 1.25 and Gamma = 1.05

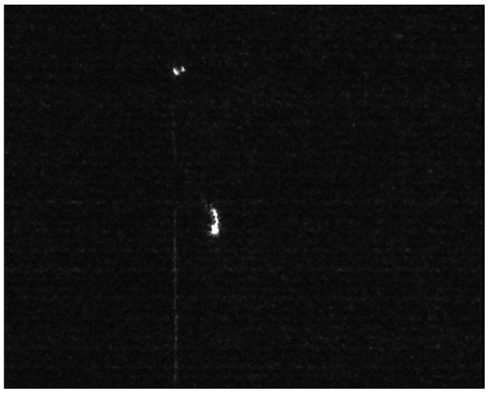
8.5 Image 5



Original Image

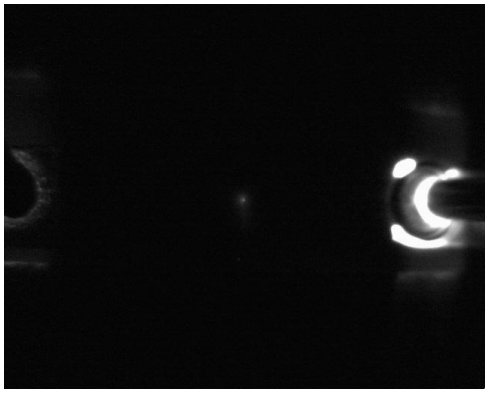


Histogram

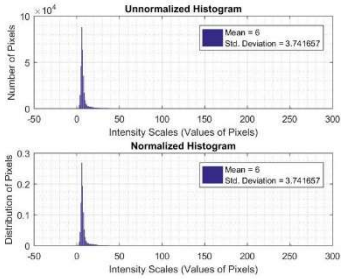


Result after Laplacian of Gaussian filter of size 3x3 with K = 1.25 and σ = 1.25
Mask = Image – Closing of image
Image = Image + mask.
Power transform with K = 1.25 and gamma = 1.15

8.6 Image 6



Original Image

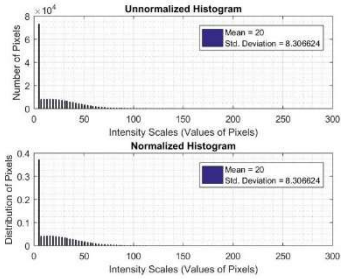


Histogram

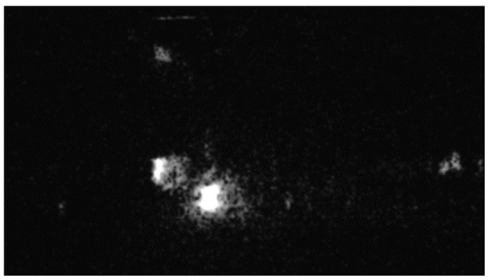
Result after Thresholding with intensity pass of 0-100.
Removal of large object using connected component.
Morphological smoothin using 7 X 7 SE
Power transform with K = 1.25 and gamma = 1.125.

8.7 Image 7

Original Image

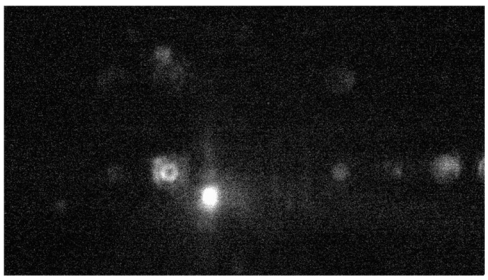


Histogram

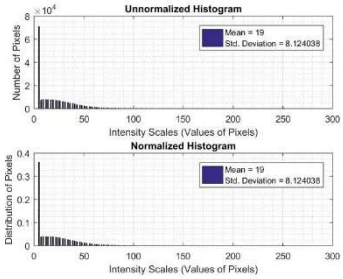


Result after Noise Reduction by Addition of mask from image
Mask = Image – (Top-Hat + Bottom-Hat)
Smoothing by Gaussian filter of size 5 X 5 with K : 1.25 and Sigma : 1.25
Power Transform with K = 1.25 and Gamma = 1.125

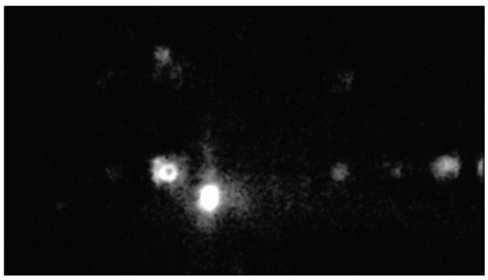
8.8 Image 8



Original Image

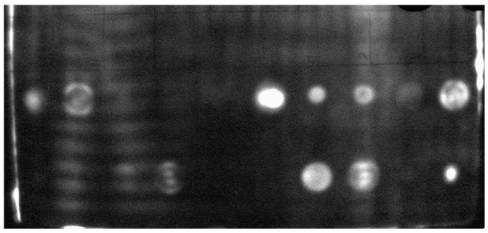


Histogram

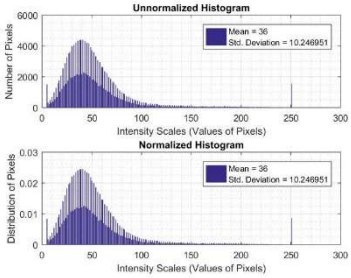


Result after Noise Reduction by Addition of mask from image
Mask = Image – (Top-Hat + Bottom-Hat)
Smoothing by Gaussian filter of size 5 X 5 with K : 1.25 and Sigma : 1.25
Power Transform with K = 1.00 and Gamma = 1.075

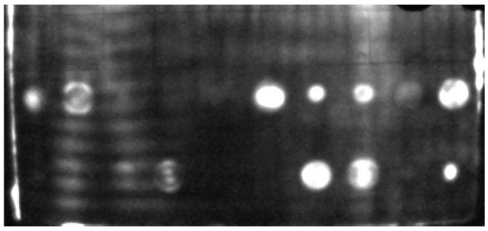
8.9 Image 9



Original Image

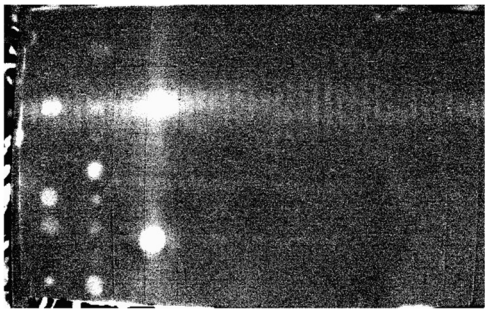


Histogram

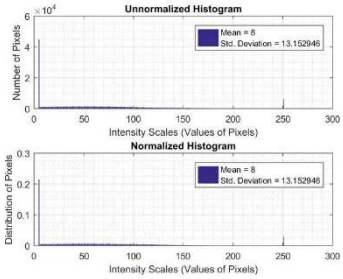


Result after Smoothing by Gaussian filter of size 5 X 5 with K = 1.5 and Sigma = 1.5
Opening of image with radius 7.
Power Transform with K = 0.75 and Gamma = 1.125

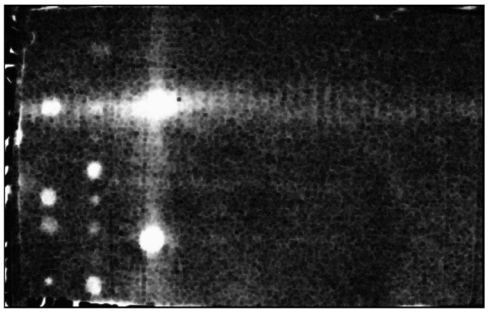
8.10 Image 10



Original Image



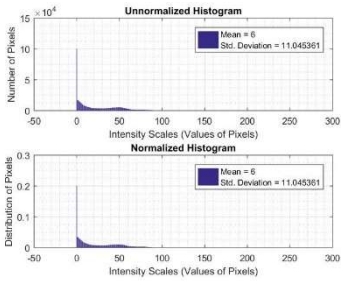
Histogram



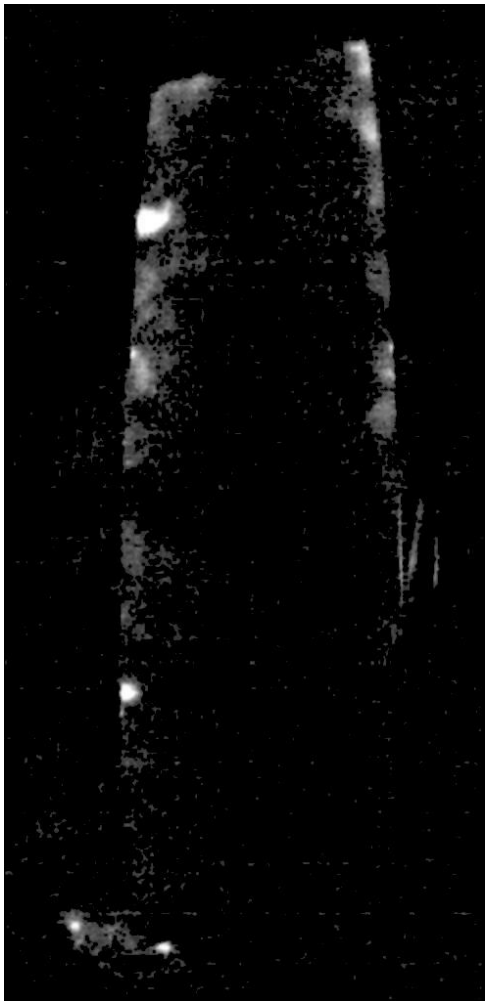
Result after Geometric mean filter of size 3 X 3
Finding the mask using Otsu's thresholding method.

8.11 Image 11

Original Image

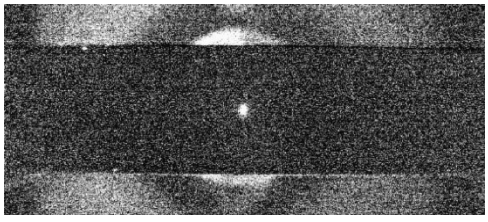


Histogram

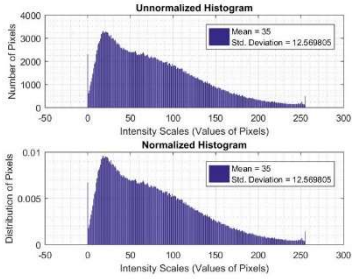


Result after Multiple global thresholding with k_1 = 25 and k_2 = 180
Smoothing with Gaussian filter of size 3 X 3 with K = 1 and Sigma = 1.25
Power Transform with K = 1.25 and Gamma = 1.025

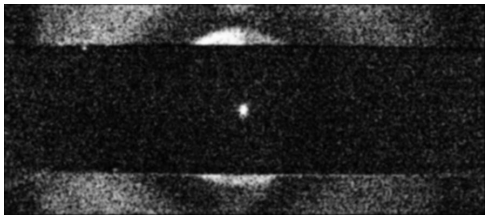
8.12 Image 12



Original Image

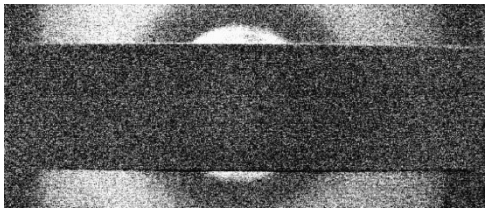


Histogram

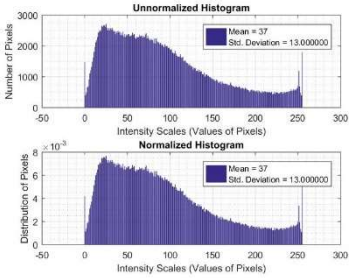


Result after Top-Hat transform followed by bottom hat transform with radius of 11 (23 X 23)
Smoothing by 5 X 5 Gaussian filter with K = 2 and Sigma = 2

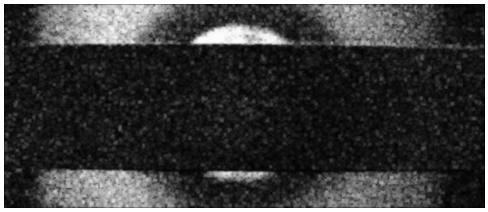
8.13 Image 13



Original Image

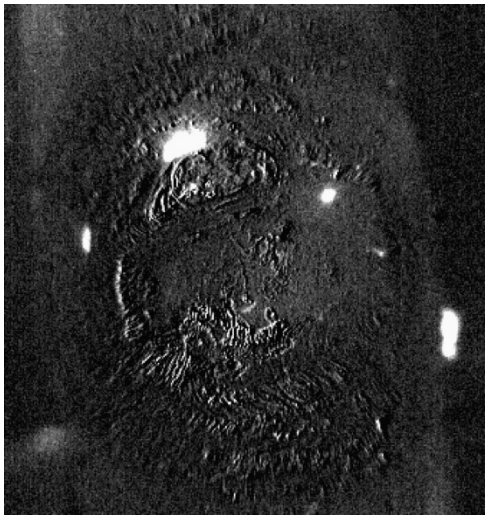


Histogram

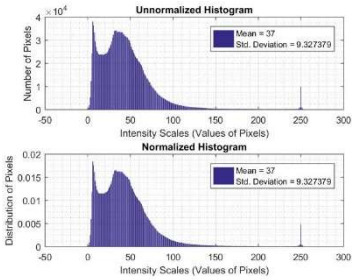


Result after Top-Hat transform followed by bottom hat transform with radius of 11 (23 X 23)
Smoothing by 5 X 5 Gaussian filter with K = 2 and Sigma = 2

8.14 Image 14



Original Image

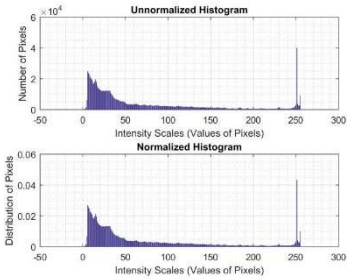


Histogram

Result after basic global thresholding with intensity 37
Morphological Smoothing with SE of Size 3 X 3

8.15 Image 15

Original Image



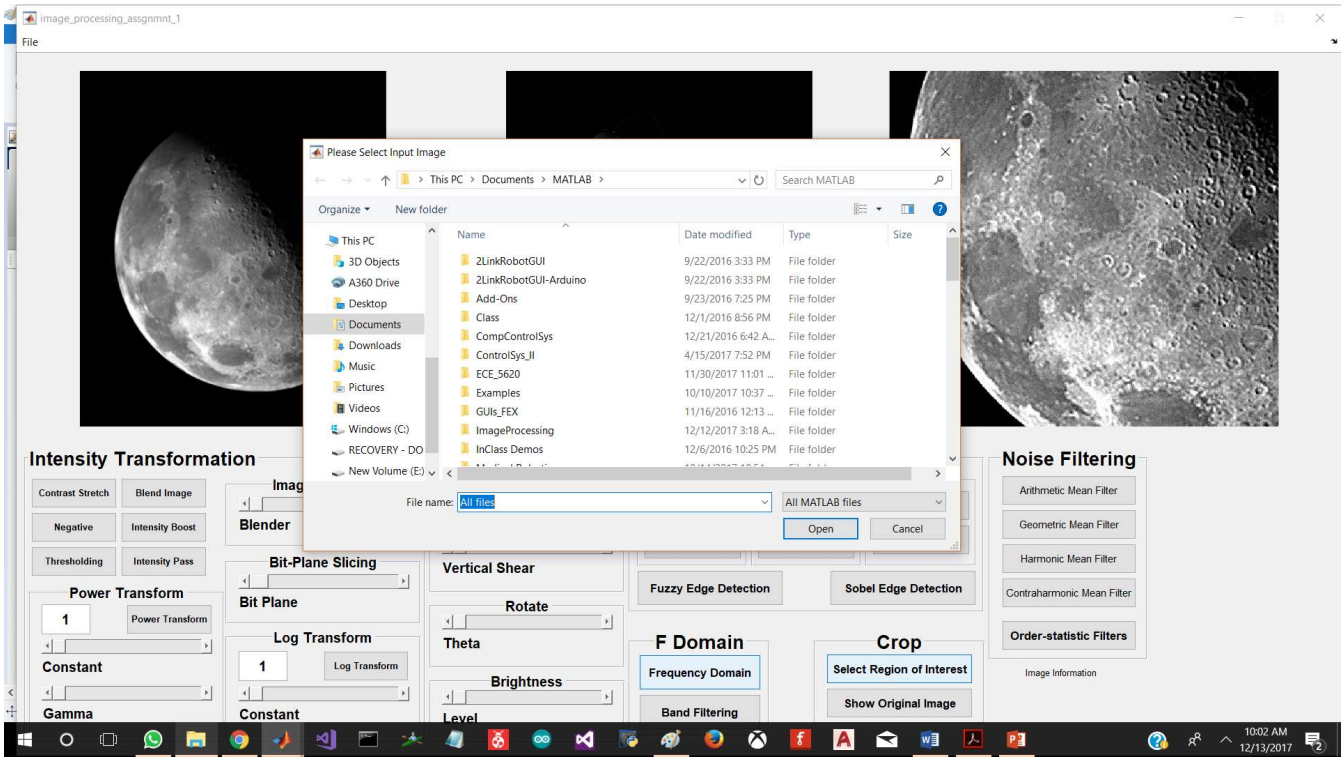
Histogram

Result after Top-Hat transform followed by closing with SE 7 X 7 Smoothing with Gaussian kernel of Size 3 X 3, K = 1 and Sigma = 1.25

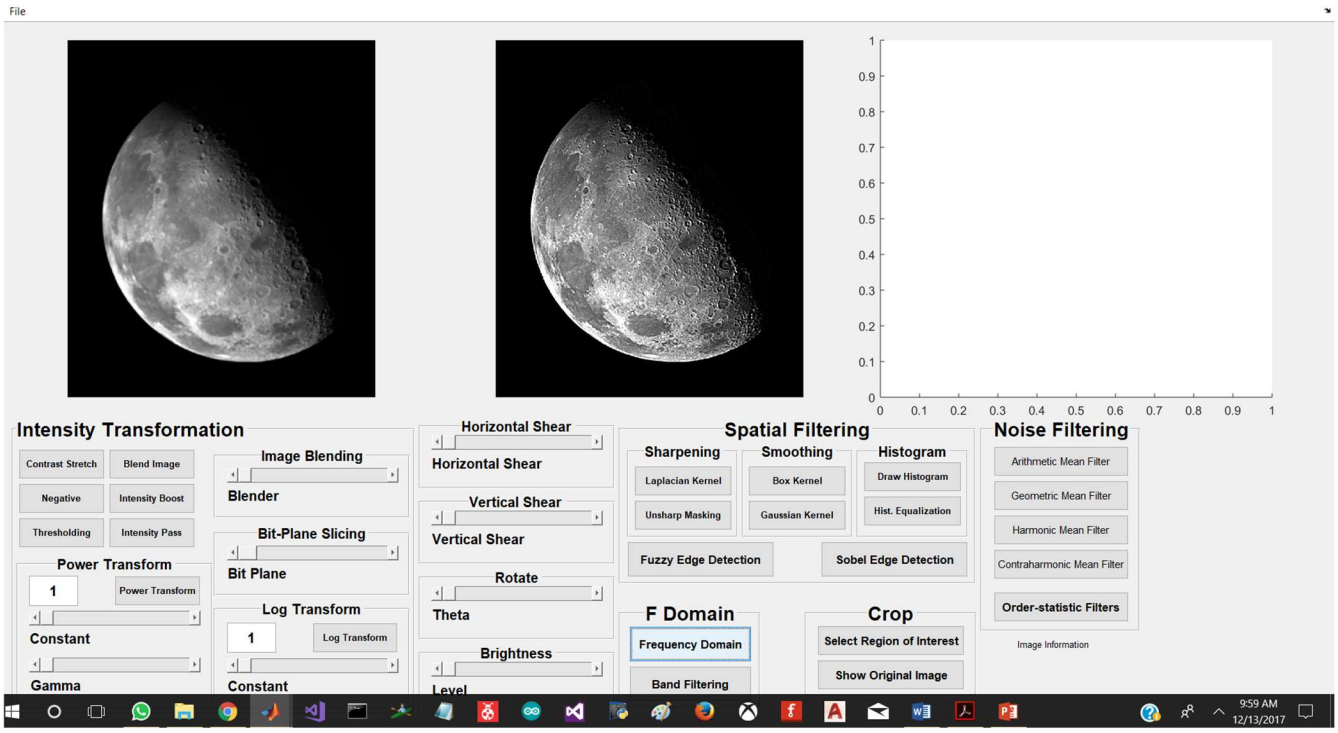
9. Appendix

9.1 MATLAB GUI

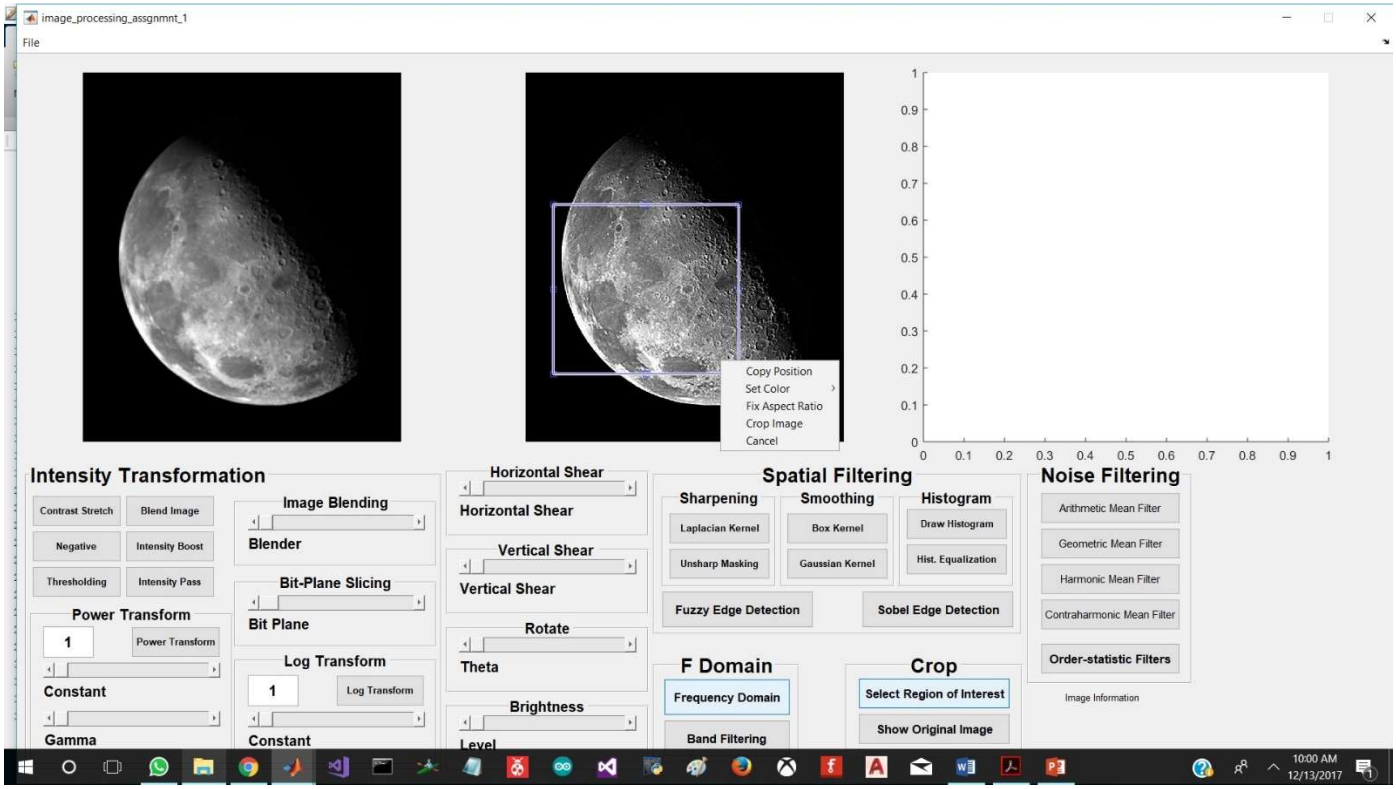
Below are the screenshots of MATLAB GUI:



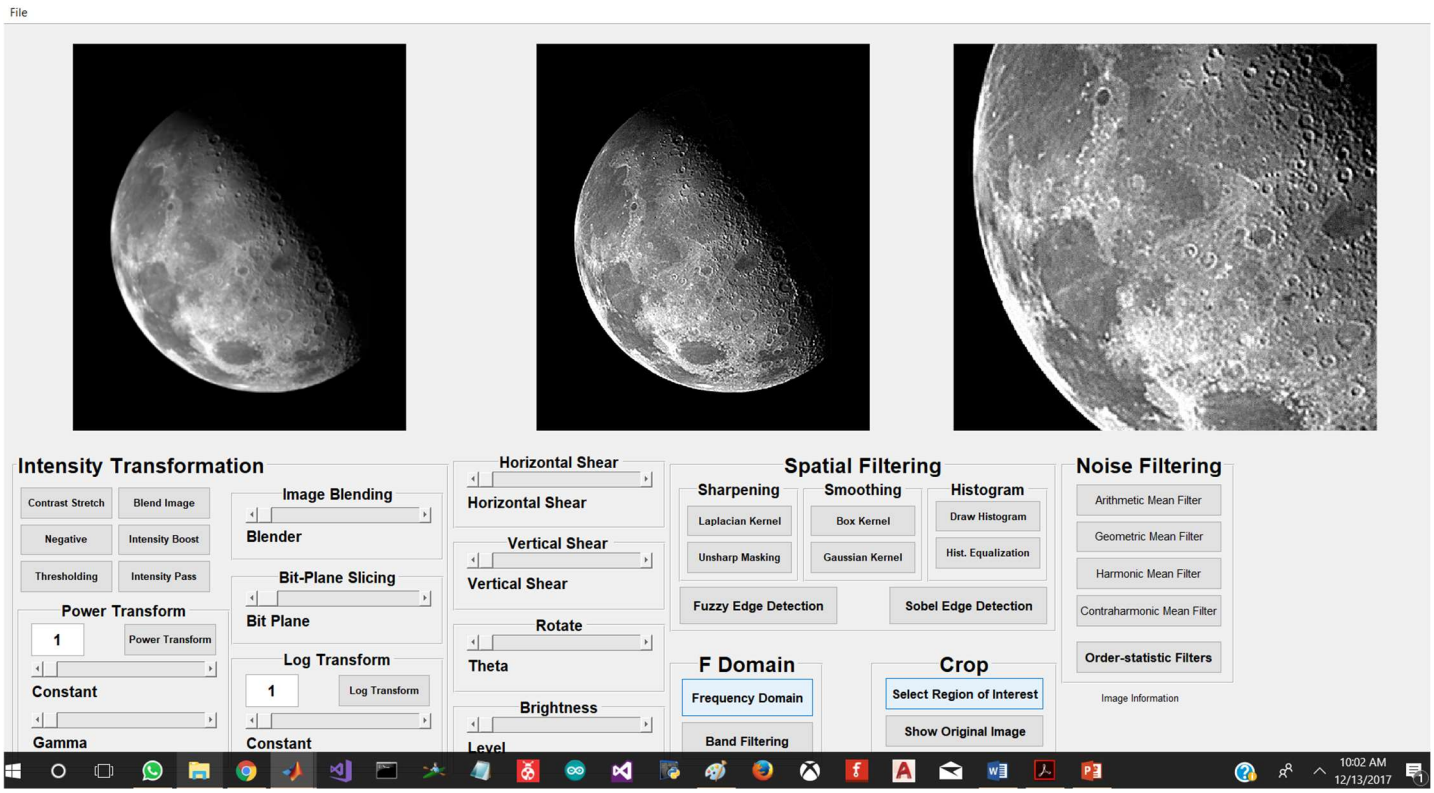
Opening the image



Processed Image



Selecting region of Interest



Viewing ROI in another window

9.2 MATLAB GUI Callback Code

```
function varargout = image_processing_assgnmnt_1(varargin)
% IMAGE_PROCESSING_ASSGNMNT_1 MATLAB code for image_processing_assgnmnt_1.fig
%      IMAGE_PROCESSING_ASSGNMNT_1, by itself, creates a new IMAGE_PROCESSING_ASSGNMNT_1
or raises the existing
%      singleton*.
%
%      H = IMAGE_PROCESSING_ASSGNMNT_1 returns the handle to a new
IMAGE_PROCESSING_ASSGNMNT_1 or the handle to
%      the existing singleton*.
%
%      IMAGE_PROCESSING_ASSGNMNT_1('CALLBACK',hObject,eventData,handles,...) calls the
local
%      function named CALLBACK in IMAGE_PROCESSING_ASSGNMNT_1.M with the given input
arguments.
%
%      IMAGE_PROCESSING_ASSGNMNT_1('Property','Value',...) creates a new
IMAGE_PROCESSING_ASSGNMNT_1 or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before image_processing_assgnmnt_1_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to image_processing_assgnmnt_1_OpeningFcn via
varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help image_processing_assgnmnt_1

% Last Modified by GUIDE v2.5 19-Apr-2018 22:30:48

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @image_processing_assgnmnt_1_OpeningFcn, ...
                   'gui_OutputFcn',  @image_processing_assgnmnt_1_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before image_processing_assgnmnt_1 is made visible.
function image_processing_assgnmnt_1_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to image_processing_assgnmnt_1 (see VARARGIN)

% Choose default command line output for image_processing_assgnmnt_1
handles.output = hObject;
```

```
imProcessor;    % Running an instance of imProcessor Class....
handles.imProcessing = imProcessor; % Storing objects of imProcessor in handles
variable...

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes image_processing_assgnmnt_1 wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command line.
function varargout = image_processing_assgnmnt_1_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

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

% --------------------------------------------------------------------
function open_image_Callback(hObject, eventdata, handles)
% hObject    handle to open_image (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
axes(handles.main_axes);
handles.imProcessing = handles.imProcessing.getImg;
handles.imProcessing = handles.imProcessing.showOriginal;
% set(handles.image_information, 'String',...
%
matlab.unittest.diagnostics.ConstraintDiagnostic.getDisplayableString(handles.imProcessin
g.imgInfo));
guidata(hObject, handles);

% --------------------------------------------------------------------
function new_image_Callback(hObject, eventdata, handles)
% hObject    handle to new_image (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
cla(handles.main_axes,'reset');
handles.imProcessing = handles.imProcessing.getImg;
handles.imProcessing = handles.imProcessing.showOriginal;
% set(handles.image_information, 'String',...
%
matlab.unittest.diagnostics.ConstraintDiagnostic.getDisplayableString(handles.imProcessin
g.imgInfo));
guidata(hObject, handles);

% --------------------------------------------------------------------
function save_image_Callback(hObject, eventdata, handles)
% hObject    handle to save_image (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.writeNsaveImage;
guidata(hObject, handles);
```

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

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

% --------------------------------------------------------------------
function clear_axes_Callback(hObject, eventdata, handles)
% hObject    handle to clear_axes (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handlesType = {'Analyzer Window', 'Auxialary Window', 'Both Windows'};
[s, v] = listdlg('ListString', handlesType, 'SelectionMode', 'single',...
    'ListSize', [250 100], 'InitialValue', [1], 'Name', 'Window Selection',...
    'PromptString', 'Please select window to clear:',...
    'OKString', 'Select', 'CancelString', 'Cancel');

if v == 1;

    switch s;

        case 1;
            cla(handles.analyzer_window, 'reset');
        case 2;
            cla(handles.auxiliary_window, 'reset');
        case 3;
            cla(handles.analyzer_window, 'reset');
            cla(handles.auxiliary_window, 'reset');
        otherwise;

    end

end

% --------------------------------------------------------------------
function close_GUI_Callback(hObject, eventdata, handles)
% hObject    handle to close_GUI (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
switch questdlg('Are you sure you want to close? All unsaved data will be lost.',...
                  'Close', 'Yes', 'No', 'No');
    case 'Yes';
        close(gcf); % Closes the current figure handle...
    case 'No';
        % ------
end


% --- Executes on button press in draw_negative.
function draw_negative_Callback(hObject, eventdata, handles)
% hObject    handle to draw_negative (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.getNegative;
guidata(hObject, handles);
```

```
% --- Executes on button press in log_transform.
function log_transform_Callback(hObject, eventdata, handles)
% hObject    handle to log_transform (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.getLogTransform;
guidata(hObject, handles);

% --- Executes on button press in power_transform.
function power_transform_Callback(hObject, eventdata, handles)
% hObject    handle to power_transform (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
axes(handles.analyzer_window);
handles.imProcessing.constant = 1;
handles.imProcessing.gam_ma = 1;
handles.imProcessing = handles.imProcessing.getPowerTransform;
guidata(hObject, handles);


% --- Executes on button press in stretch_contrast.
function stretch_contrast_Callback(hObject, eventdata, handles)
% hObject    handle to stretch_contrast (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.contrastStretching;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --- Executes on button press in intensity_pass.
function intensity_pass_Callback(hObject, eventdata, handles)
% hObject    handle to intensity_pass (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.intensityPass;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);

% --- Executes on button press in intensity_boost.
function intensity_boost_Callback(hObject, eventdata, handles)
% hObject    handle to intensity_boost (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.intensityBoost;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --- Executes on button press in thresholder.
function thresholder_Callback(hObject, eventdata, handles)
% hObject    handle to thresholder (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.thresholdingImage;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);
```

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

% Hints: get(hObject,'String') returns contents of const_for_LogTransform as text
%        str2double(get(hObject,'String')) returns contents of const_for_LogTransform as
a double
% handles.imProcessing.constant = get(hObject, 'Value');
handles.imProcessing.constant = get(hObject, 'Value');
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function const_for_LogTransform_CreateFcn(hObject, eventdata, handles)
% hObject    handle to const_for_LogTransform (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

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

% Hints: get(hObject,'String') returns contents of const_for_PowerTransform as text
%        str2double(get(hObject,'String')) returns contents of const_for_PowerTransform
as a double
handles.imProcessing.constant = get(hObject, 'Value');
handles.imProcessing.gam_ma = get(hObject, 'Value');
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function const_for_PowerTransform_CreateFcn(hObject, eventdata, handles)
% hObject    handle to const_for_PowerTransform (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

% --- Executes during object creation, after setting all properties.
function image_information_CreateFcn(hObject, eventdata, handles)
% hObject    handle to image_information (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called


% --- Executes on slider movement.
function sldr_4_log_constant_Callback(hObject, eventdata, handles)
% hObject    handle to sldr_4_log_constant (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
```

```
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
axes(handles.analyzer_window);
handles.imProcessing.constant = uint8(get(hObject, 'Value'));
handles.imProcessing = handles.imProcessing.showVariant;
% imshow(get(hObject, 'Value')*handles.imProcessing.imgProcessed);
set(handles.current_log_const, 'String', sprintf('Constant : %f', get(hObject,
'Value')));
guidata(hObject, handles);


% --- Executes during object creation, after setting all properties.
function sldr_4_log_constant_CreateFcn(hObject, eventdata, handles)
% hObject    handle to sldr_4_log_constant (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end


% --- Executes on slider movement.
function sldr_4_power_constant_Callback(hObject, eventdata, handles)
% hObject    handle to sldr_4_power_constant (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
handles.imProcessing.constant = get(hObject, 'Value');
handles.imProcessing = handles.imProcessing.getPowerTransform;
set(handles.current_power_const, 'String', sprintf('Constant : %f',get(hObject,
'Value')));
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function sldr_4_power_constant_CreateFcn(hObject, eventdata, handles)
% hObject    handle to sldr_4_power_constant (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end


% --- Executes on slider movement.
function bit_plane_slicer_Callback(hObject, eventdata, handles)
% hObject    handle to bit_plane_slicer (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
handles.imProcessing.bitPlane = 8 - get(hObject, 'Value');
handles.imProcessing = handles.imProcessing.bitPlaneSlicer;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
```

```
set(handles.bit_plane_indicator, 'String', sprintf('Bit Plane : %d', 8 - get(hObject,
'Value')));
guidata(hObject, handles);


% --- Executes during object creation, after setting all properties.
function bit_plane_slicer_CreateFcn(hObject, eventdata, handles)
% hObject    handle to bit_plane_slicer (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end


% --- Executes during object creation, after setting all properties.
function bit_plane_indicator_CreateFcn(hObject, eventdata, handles)
% hObject    handle to bit_plane_indicator (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called


% --- Executes on slider movement.
function horizontal_shear_Callback(hObject, eventdata, handles)
% hObject    handle to horizontal_shear (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
handles.imProcessing.shearConstant = get(hObject, 'Value');
handles.imProcessing = handles.imProcessing.shearHorizontal;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
set(handles.horizontal_shear_constant, 'String', sprintf('Shear : %f', get(hObject,
'Value')));
guidata(hObject, handles);


% --- Executes during object creation, after setting all properties.
function horizontal_shear_CreateFcn(hObject, eventdata, handles)
% hObject    handle to horizontal_shear (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end


% --- Executes during object creation, after setting all properties.
function horizontal_shear_constant_CreateFcn(hObject, eventdata, handles)
% hObject    handle to horizontal_shear_constant (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called


% --- Executes on slider movement.
function vertical_shear_Callback(hObject, eventdata, handles)
% hObject    handle to vertical_shear (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
handles.imProcessing.shearConstant = get(hObject, 'Value');
handles.imProcessing = handles.imProcessing.shearVertical;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
set(handles.vertical_shear_constant, 'String', sprintf('Shear : %f', get(hObject,
'Value')));
guidata(hObject, handles);


% --- Executes during object creation, after setting all properties.
function vertical_shear_CreateFcn(hObject, eventdata, handles)
% hObject    handle to vertical_shear (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end


% --- Executes during object creation, after setting all properties.
function vertical_shear_constant_CreateFcn(hObject, eventdata, handles)
% hObject    handle to vertical_shear_constant (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called


% --- Executes on slider movement.
function rotate_image_Callback(hObject, eventdata, handles)
% hObject    handle to rotate_image (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
handles.imProcessing.theta = get(hObject, 'Value');
handles.imProcessing = handles.imProcessing.rotateImg;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
set(handles.theta_indicator, 'String', sprintf('Theta : %f', get(hObject,
'Value')*(180/pi)));
guidata(hObject, handles);


% --- Executes during object creation, after setting all properties.
function rotate_image_CreateFcn(hObject, eventdata, handles)
% hObject    handle to rotate_image (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end

% --- Executes during object creation, after setting all properties.
function theta_indicator_CreateFcn(hObject, eventdata, handles)
```

```
% hObject    handle to theta_indicator (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called


% --- Executes on button press in blend_image.
function blend_image_Callback(hObject, eventdata, handles)
% hObject    handle to blend_image (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.getAuxImg;
guidata(hObject, handles);

% --- Executes on slider movement.
function image_blender_Callback(hObject, eventdata, handles)
% hObject    handle to image_blender (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
handles.imProcessing.constant = get(hObject, 'Value');
handles.imProcessing = handles.imProcessing.blendImg;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
set(handles.blender_constant, 'String', sprintf('Blender : %f', get(hObject, 'Value')));
guidata(hObject, handles);


% --- Executes during object creation, after setting all properties.
function image_blender_CreateFcn(hObject, eventdata, handles)
% hObject    handle to image_blender (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end


% --- Executes during object creation, after setting all properties.
function blender_constant_CreateFcn(hObject, eventdata, handles)
% hObject    handle to blender_constant (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called


% --- Executes during object creation, after setting all properties.
function current_log_const_CreateFcn(hObject, eventdata, handles)
% hObject    handle to current_log_const (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called


% --- Executes during object creation, after setting all properties.
function current_power_const_CreateFcn(hObject, eventdata, handles)
% hObject    handle to current_power_const (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called


% --- Executes on slider movement.
```

```
function sldr_4_gamma_Callback(hObject, eventdata, handles)
% hObject    handle to sldr_4_gamma (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
handles.imProcessing.gam_ma = get(hObject, 'Value');
handles.imProcessing = handles.imProcessing.getPowerTransform;
set(handles.current_gamma, 'String', sprintf('Gamma : %f',get(hObject, 'Value')));
guidata(hObject, handles);


% --- Executes during object creation, after setting all properties.
function sldr_4_gamma_CreateFcn(hObject, eventdata, handles)
% hObject    handle to sldr_4_gamma (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end


% --- Executes on slider movement.
function brightness_control_Callback(hObject, eventdata, handles)
% hObject    handle to brightness_control (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of slider
handles.imProcessing.constant = get(hObject, 'Value');
handles.imProcessing = handles.imProcessing.brightnessControl;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
set(handles.brightness_level, 'String', sprintf('Level : %f', get(hObject, 'Value')));
guidata(hObject, handles);

% --- Executes during object creation, after setting all properties.
function brightness_control_CreateFcn(hObject, eventdata, handles)
% hObject    handle to brightness_control (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end


% --- Executes during object creation, after setting all properties.
function brightness_level_CreateFcn(hObject, eventdata, handles)
% hObject    handle to brightness_level (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called


% --- Executes during object creation, after setting all properties.
function current_gamma_CreateFcn(hObject, eventdata, handles)
% hObject    handle to current_gamma (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
```

```
% handles    empty - handles not created until after all CreateFcns called


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


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


% --------------------------------------------------------------------
function histogram_estimation_Callback(hObject, eventdata, handles)
% hObject    handle to histogram_estimation (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
figure;
handles.imProcessing = handles.imProcessing.drawHist;
guidata(hObject, handles);


% --------------------------------------------------------------------
function hist_equalization_Callback(hObject, eventdata, handles)
% hObject    handle to hist_equalization (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
figure;
handles.imProcessing = handles.imProcessing.drawHist;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.myHistEqualization;
guidata(hObject, handles);


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


% --------------------------------------------------------------------
function box_kernel_filter_Callback(hObject, eventdata, handles)
% hObject    handle to box_kernel_filter (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.smoothByBox;
guidata(hObject, handles);


% --------------------------------------------------------------------
function gaussian_kernel_filter_Callback(hObject, eventdata, handles)
% hObject    handle to gaussian_kernel_filter (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.smoothByGauss;
```

```
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


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


% --------------------------------------------------------------------
function laplacian_kernel_Callback(hObject, eventdata, handles)
% hObject    handle to laplacian_kernel (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.sharpenByLaplacian;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --------------------------------------------------------------------
function unsharp_masking_Callback(hObject, eventdata, handles)
% hObject    handle to unsharp_masking (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.unsharpMasking;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --------------------------------------------------------------------
function fuzzy_edge_detection_Callback(hObject, eventdata, handles)
% hObject    handle to fuzzy_edge_detection (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.fuzzyEdgeDetection;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);

% --------------------------------------------------------------------
function sobel_edge_detection_Callback(hObject, eventdata, handles)
% hObject    handle to sobel_edge_detection (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.sobelEdgeDetector;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


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


% --------------------------------------------------------------------
```

```
function frequency_domain_filtering_Callback(hObject, eventdata, handles)
% hObject    handle to frequency_domain_filtering (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.getLowHighPassFilter;
handles.imProcessing = handles.imProcessing.fourierFiltering;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --------------------------------------------------------------------
function frequency_domain_band_filtering_Callback(hObject, eventdata, handles)
% hObject    handle to frequency_domain_band_filtering (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.getBandPassRejectFilter;
handles.imProcessing = handles.imProcessing.fourierBandSelectFiltering;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


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


% --------------------------------------------------------------------
function arithmetic_mean_filter_Callback(hObject, eventdata, handles)
% hObject    handle to arithmetic_mean_filter (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.arithmeticMeanFilter;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --------------------------------------------------------------------
function geometric_mean_filter_Callback(hObject, eventdata, handles)
% hObject    handle to geometric_mean_filter (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.geometricMeanFilter;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --------------------------------------------------------------------
function harmonic_mean_filter_Callback(hObject, eventdata, handles)
% hObject    handle to harmonic_mean_filter (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.harmonicMeanFilter;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);
```

```
% --------------------------------------------------------------------
function contraharmonic_mean_filter_Callback(hObject, eventdata, handles)
% hObject    handle to contraharmonic_mean_filter (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.contraHarmonicMeanFilter;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --------------------------------------------------------------------
function order_statistic_filtering_Callback(hObject, eventdata, handles)
% hObject    handle to order_statistic_filtering (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.orderStatisticsFilter;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


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


% --------------------------------------------------------------------
function crop_image_Callback(hObject, eventdata, handles)
% hObject    handle to crop_image (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.selectROIorImg;
axes(handles.auxiliary_window);
handles.imProcessing = handles.imProcessing.showOriginal;
guidata(hObject, handles);


% --------------------------------------------------------------------
function view_original_Callback(hObject, eventdata, handles)
% hObject    handle to view_original (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showOriginal;
guidata(hObject, handles);


% --------------------------------------------------------------------
function convert2gray_Callback(hObject, eventdata, handles)
% hObject    handle to convert2gray (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
axes(handles.analyzer_window);
[handles.imProcessing, handles.imProcessing.imgProcessed] =
handles.imProcessing.convert2gray(handles.imProcessing.img);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);

% --------------------------------------------------------------------
```

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


% --------------------------------------------------------------------
function morph_erosion_Callback(hObject, eventdata, handles)
% hObject    handle to morph_erosion (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.getKernelSize;
handles.imProcessing = handles.imProcessing.erosion;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);

% --------------------------------------------------------------------
function morph_dilation_Callback(hObject, eventdata, handles)
% hObject    handle to morph_dilation (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.getKernelSize;
handles.imProcessing = handles.imProcessing.dilation;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --------------------------------------------------------------------
function morph_opening_Callback(hObject, eventdata, handles)
% hObject    handle to morph_opening (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.getKernelSize;
handles.imProcessing = handles.imProcessing.opening;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --------------------------------------------------------------------
function morph_closing_Callback(hObject, eventdata, handles)
% hObject    handle to morph_closing (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.getKernelSize;
handles.imProcessing = handles.imProcessing.closing;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --------------------------------------------------------------------
function boundary_extraction_Callback(hObject, eventdata, handles)
% hObject    handle to boundary_extraction (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.getKernelSize;
handles.imProcessing = handles.imProcessing.boundaryExtraction;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);
```

```
% --------------------------------------------------------------------
function morphological_gradient_Callback(hObject, eventdata, handles)
% hObject    handle to morphological_gradient (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.getKernelSize;
handles.imProcessing = handles.imProcessing.morphologicalGradient;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --------------------------------------------------------------------
function top_hat_transform_Callback(hObject, eventdata, handles)
% hObject    handle to top_hat_transform (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.getKernelSize;
handles.imProcessing = handles.imProcessing.topHatTransform;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);

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

handles.imProcessing = handles.imProcessing.getKernelSize;
handles.imProcessing = handles.imProcessing.bottomHatTransform;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --------------------------------------------------------------------
function morph_smoothing_Callback(hObject, eventdata, handles)
% hObject    handle to morph_smoothing (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.getKernelSize;
handles.imProcessing = handles.imProcessing.morphSmoothing;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);

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


% --------------------------------------------------------------------
function line_detection_Callback(hObject, eventdata, handles)
% hObject    handle to line_detection (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.lineDetection;
axes(handles.analyzer_window);
```

```
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --------------------------------------------------------------------
function laplacian_operator_Callback(hObject, eventdata, handles)
% hObject    handle to laplacian_operator (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.laplacianOperator;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);
% --------------------------------------------------------------------
function gradient_operator_Callback(hObject, eventdata, handles)
% hObject    handle to gradient_operator (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.gradientOperator;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --------------------------------------------------------------------
function kirsch_compass_kernels_Callback(hObject, eventdata, handles)
% hObject    handle to kirsch_compass_kernels (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.kirschCompassMask;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


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

handles.imProcessing = handles.imProcessing.marrHildrethEdgeDetection;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);


% --------------------------------------------------------------------
function basic_global_thresholding_Callback(hObject, eventdata, handles)
% hObject    handle to basic_global_thresholding (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
handles.imProcessing = handles.imProcessing.basicGlobalThresholding;
axes(handles.analyzer_window);
handles.imProcessing = handles.imProcessing.showProcessed;
guidata(hObject, handles);
```

9.3 List of functions working in `imProcessor` class object

```
classdef imProcessor

    % The class created for image processing.
    % Detailed explanation goes here

    properties

        img                 % the variable where input image will be stored...
        auxImg
        imgProcessed        % the variable where processed image will be stored...
        arrayOfAlpha        % the array of alpha angles calculated in gradient
        pixelVals

        randNumArray        % array to store random numbers...
        imgInfo             % variable to store image information...

        threshold           % the variable where value of threshold pixel will be
stored...
        constant            % constant multiplier for log and gamma operations...
        gam_ma              % gamma value for power transform...
        bitPlane            % Stores the numbe of bit
        shearConstant       % stores constant for shear
        theta               % stores angle for rotation

        fileName            % property where filename of input image is stored...
        pathName            % property where path of input image is stored...
        filterIndex         % property where filterindex of input image is stored...

        meanVal             % property of imProcessor class where mean of histogram is
stored...
        stdDev              % property of imProcessor class where standard deviaton is
stored...
        hist_data
        norm_hist_data      % property of imProcessor class where normalized histogram
data is saved...

        kernelSize          % property of imProcessor class where size of kernel is
stored...
        arraySum            % property of imProcessor class where sum of array is
stored...
        arrayProduct        % stores product of array elements
        arrayOfDifference   % stores difference of array elements.

        gaussKernel         % property of imProcessor class where gaussian kernel is
stored...
        laplacianKernel     % property of imProcessor class where laplacian kernel is
stored...
        laplacianKernelType % property of imProcessor class where laplacian kernel type
is stored...
        lapOfGauss          % property of imProcessor class where laplacian of Gaussian
is stored

        cutOff              % property where cutoff frequency is stored...
        lowPass             % Low pass filter...
        highPass            % High pass filter...
        laplacianFDomain    % laplacian filter in Frequency domain...

        c0                  % center of band
        W                   % width of band
        bandPass            % band-pass filter
        bandReject          % band-reject filter
```

```
        sortedArray         % Sorted array of subregion
        medianIntensity     % median intensity for order-statistics filter
        minIntensity        % min intensity for order-statistics filter
        maxIntensity        % max intensity for order-statistics filter
        midIntensity        % midpoint intensity for order-statistics filter
        alphaTrimmedArray   % contains alpha-trimmed array output of subregion

    end     % end of imProcessor class properties...

    methods

        function imProcess = getImg (imProcess) % this function helps to get image...

            [imProcess.fileName, imProcess.pathName, imProcess.filterIndex] =
uigetfile('*.*', 'Please Select Input Image');
            imProcess.img = imread(imProcess.fileName);  % the command takes image and
stores in to global variable..
            imProcess.imgInfo = imfinfo(imProcess.fileName);
            imProcess.imgProcessed = double(zeros(size(imProcess.img),
class(imProcess.img)));
            clc;
            %disp(imProcess.imgInfo);

        end

        function imProcess = writeNsaveImage (imProcess) % This finction saves the image

            imgName = sprintf('%s_modified.%s', imProcess.fileName,
imProcess.imgInfo.Format);
            if imProcess.imgInfo.BitDepth > 1
                imwrite(uint8(imProcess.imgProcessed), imgName,
imProcess.imgInfo.Format);
            elseif imProcess.imgInfo.BitDepth == 1
                imwrite(logical(imProcess.imgProcessed), imgName,
imProcess.imgInfo.Format);
            end

        end

        function imProcess = selectROIorImg (imProcess) % this function selects ROI

            regionSelection = {'Select region of interest of Processed Image', 'Select
complete processed Image'};
            [s, v] = listdlg('ListString', regionSelection, 'SelectionMode', 'single',...
                             'ListSize', [250 100], 'InitialValue', [1], 'Name', 'Select
Region',...
                             'PromptString', 'Please select region of interest :',...
                             'OKString', 'Select', 'CancelString', 'Cancel');

            if v == 1

                switch s;
                    case 1;
                        imProcess.img = imcrop(imProcess.imgProcessed);
                    case 2;
                        imProcess.img = imProcess.imgProcessed;
                    otherwise

                end
            end

        end
```

```
        function [imProcess, outputArray] = convert2gray (imProcess, anyArray)

            outputArray = rgb2gray(anyArray);

        end

        function imProcess = showOriginal(imProcess)    % functon to show original
image...

            if imProcess.imgInfo.BitDepth > 1
                imshow(uint8(imProcess.img));
            elseif imProcess.imgInfo.BitDepth == 1
                imshow(logical(imProcess.img));
            end

        end

        function imProcess = showProcessed(imProcess)   % function to show only processed
image...

            if imProcess.imgInfo.BitDepth > 1
                imshow(uint8(imProcess.imgProcessed));
            elseif imProcess.imgInfo.BitDepth == 1
                imshow(logical(imProcess.imgProcessed));
            end

        end

        function imProcess = showVariant(imProcess)     % function to variant in log
transforms...

            if imProcess.constant
                imProcess.constant = uint8(imProcess.constant);
                imshow(imProcess.constant .* uint8(imProcess.imgProcessed));
            end

        end

        function imProcess = thrshldPixlVal (imProcess) % this function gets threshold
value of pixel for thresholding...

            options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
            imProcess.threshold = str2double(inputdlg({'Enter threshold value of
Pixel...'},...
                                        'Threshold Pixel value', [1 50], {'0'},
options));

        end

        function imProcess = getRandNum (imProcess) % gets random number matrix...

            imProcess.randNumArray = rand (numrows(imProcess.img),
numcols(imProcess.img));

        end

        % ================ function for problem statement 1 ======================

        function imProcess = histEstimator (imProcess, anyArray)

            imProcess.pixelVals = unique(anyArray);
```

```
            mySum = 0;
            sumOfProb = zeros(size(imProcess.pixelVals));
            newPixelVals = zeros(size(imProcess.pixelVals));
            imProcess.hist_data = zeros(size(imProcess.pixelVals));
            imProcess.norm_hist_data = zeros(size(imProcess.pixelVals));

            for n = 1:1:numrows(imProcess.pixelVals);
                for z = 1:1:size(anyArray, 3)
                    for x = 1:1:size(anyArray, 1);
                        for y = 1:1:size(anyArray, 2);

                            if imProcess.pixelVals(n,:) == anyArray(x,y,z);
                                imProcess.hist_data(n,:) = imProcess.hist_data(n,:) + 1;
                                imProcess.norm_hist_data(n,:) =
(imProcess.hist_data(n,:))/(numrows(anyArray)*numcols(anyArray));
                            end

                        end
                    end
                end
            end

            imProcess.meanVal = 0;
            imProcess.stdDev = 0;

            for n = 1:1:numrows(imProcess.pixelVals);
                imProcess.meanVal = imProcess.meanVal +
(imProcess.pixelVals(n,:)*imProcess.norm_hist_data(n,:));
            end

            for n = 1:1:numrows(imProcess.pixelVals);
                imProcess.stdDev = imProcess.stdDev +(((imProcess.pixelVals(n,:) -
imProcess.meanVal)^2)*imProcess.norm_hist_data(n,:));
            end

            imProcess.stdDev = sqrt(double(imProcess.stdDev));

        end

        function imProcess = drawHist (imProcess) % Draws histogram

            imProcess = imProcess.histEstimator(imProcess.img);
            subplot(2,1,1);     bar(imProcess.pixelVals, imProcess.hist_data);
            xlabel('Intensity Scales (Values of Pixels)');
            ylabel('Number of Pixels');
            title('Unnormalized Histogram');
            grid on;
            grid minor;
            %legend(sprintf('Mean = %d\nStd. Deviation = %f', imProcess.meanVal,
imProcess.stdDev));
            subplot(2,1,2);     bar(imProcess.pixelVals, imProcess.norm_hist_data);
            xlabel('Intensity Scales (Values of Pixels)');
            ylabel('Distribution of Pixels');
            title('Normalized Histogram');
            grid on;
            grid minor;
            %legend(sprintf('Mean = %d\nStd. Deviation = %f', imProcess.meanVal,
imProcess.stdDev));

        end     % end of Histogram drawing function...

        % ================== end of function for problem statement 1 ==============
```

```
        % ================== function for histogram equalization ==========

        function imProcess = myHistEqualization (imProcess) % Equalizes the histogram

            mySum = 0;
            pixelvals = unique(imProcess.img);
            sumOfProb = zeros(size(imProcess.norm_hist_data));
            newPixelVals = zeros(size(imProcess.norm_hist_data));
            for n = 1:1:numrows(imProcess.norm_hist_data)
                mySum  = mySum + imProcess.norm_hist_data(n);
                newPixelVals(n,:) = mySum*(2^imProcess.imgInfo.BitDepth -1);
                sumOfProb(n, :) = mySum;
            end

            for n = 1:1:numrows(pixelvals)
                for x = 1:1:numrows(imProcess.img)
                    for y = 1:1:numcols(imProcess.img)
                        if imProcess.img(x,y) == pixelvals(n,:)
                            imProcess.imgProcessed(x,y) = newPixelVals(n,:);
                        end
                    end
                end
            end
            imProcess.imgProcessed = uint8(imProcess.imgProcessed);
            imProcess.showProcessed;

        end

        % ================== end of function for histogram eq.=============

        % ================== Affine transformations =======================

        function imProcess = shearVertical (imProcess) % Vertial Shear

            for z = 1:1:size(imProcess.img, 3);
                for x = 1:1:size(imProcess.img, 1);
                    for y = 1:1:size(imProcess.img, 2);

                        sampleArray(uint64(x + imProcess.shearConstant*y), y) =
imProcess.img(x,y);

                    end
                end
            end

            imProcess.imgProcessed = sampleArray;

        end

        function imProcess = shearHorizontal (imProcess) % Horizontal Shear

            for z = 1:1:size(imProcess.img, 3);
                for x = 1:1:size(imProcess.img, 1);
                    for y = 1:1:size(imProcess.img, 2);

                        sampleArray(x, uint64(imProcess.shearConstant*x + y)) =
imProcess.img(x,y);

                    end
                end
            end

            imProcess.imgProcessed = sampleArray;
```

```
        end

        function imProcess = rotateImg (imProcess) % Image Rotation

            for z = 1:1:size(imProcess.img, 3);
                for x = 1:1:size(imProcess.img, 1);
                    for y = 1:1:size(imProcess.img, 2);

                        xPrime = uint64((x*cos(imProcess.theta)) -
(y*sin(imProcess.theta)) + size(imProcess.img, 1)*sqrt(2));
                        yPrime = uint64((x*sin(imProcess.theta)) +
(y*cos(imProcess.theta)) + size(imProcess.img, 1)*sqrt(2));
                        sampleArray(xPrime, yPrime) = imProcess.img(x,y);

                    end
                end
            end

            imProcess.imgProcessed = sampleArray;

        end

        function imProcess = getAuxImg (imProcess) % gets auxialiary image

            imProcess.auxImg = imProcess.img;
            imProcess = imProcess.getImg;

        end

        function imProcess = blendImg (imProcess) % blends the two images...

            imProcess.imgProcessed = imProcess.constant*imProcess.img + (1-
imProcess.constant)*imProcess.auxImg;

        end

        function imProcess = brightnessControl (imProcess) % brightness control of images

            imProcess.imgProcessed = imProcess.constant + imProcess.img;

        end

        % ================== End of Affine transformations ===============

        % ================== Piecewise linear transformation ==============

        function imProcess = thresholdingImage (imProcess)

            % The function performs thresholding operation on image.

            imProcess = imProcess.thrshldPixlVal;
            imProcess.imgProcessed = imProcess.img;

            for z = 1:1:size(imProcess.imgProcessed, 3);
                for x = 1:1:size(imProcess.imgProcessed, 1);
                    for y = 1:1:size(imProcess.imgProcessed, 2);

                        if imProcess.imgProcessed(x,y,z) >= imProcess.threshold;
                            %imProcess.imgProcessed(x,y,z) = 255;
                        elseif imProcess.imgProcessed(x,y,z) < imProcess.threshold;
                            imProcess.imgProcessed(x,y,z) = 0;
                        end
```

```
                    end
                end
            end

        end

        function imProcess = contrastStretching (imProcess)

            % The function performs contrast stretching operation on Image.

            pixelvals = unique(imProcess.img);
            imProcess.imgProcessed = imProcess.img;

            for z = 1:1:size(imProcess.imgProcessed, 3);
                for x = 1:1:size(imProcess.imgProcessed, 1);
                    for y = 1:1:size(imProcess.imgProcessed, 2);

                        imProcess.imgProcessed(x, y, z) = (imProcess.img(x,y,z) -
min(pixelvals))*(255/(max(pixelvals) - min(pixelvals)));

                    end
                end
            end

        end

        function imProcess = intensityPass (imProcess)

            prompt = {'Enter low intensity value: ',...
                'Enter high intensity value: '};
            title = 'Intensity Band';
            num_lines = [1 60];
            default_ans = {'80', '180'};
            options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
            intensityBand = inputdlg(prompt, title, num_lines, default_ans, options);
            low = str2double(intensityBand{1,:});   high =
str2double(intensityBand{2,:});

            for z = 1:1:size(imProcess.img, 3);
                for x = 1:1:size(imProcess.img, 1);
                    for y = 1:1:size(imProcess.img, 2);

                        if (imProcess.img(x,y,z) >= low) && (imProcess.img(x,y,z) <=
high)

                            imProcess.imgProcessed(x,y,z) = imProcess.img(x,y,z);
                        end

                    end
                end
            end

        end

        function imProcess = intensityBoost (imProcess)

            prompt = {'Enter low intensity value: ','Enter high intensity value: ',
'Enter boost level value:'};
            title = 'Intensity Band';
            num_lines = [1 60];
            default_ans = {'80', '180', '240'};
```

```
            options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
            intensityBand = inputdlg(prompt, title, num_lines, default_ans, options);
            low = str2double(intensityBand{1,:});   high =
str2double(intensityBand{2,:});
            boostLevel = str2double(intensityBand{3,:});

            imProcess.imgProcessed = imProcess.img;

            for z = 1:1:size(imProcess.img, 3);
                for x = 1:1:size(imProcess.img, 1);
                    for y = 1:1:size(imProcess.img, 2);

                        if (imProcess.img(x,y,z) >= low) && (imProcess.img(x,y,z) <=
high)

                            imProcess.imgProcessed(x,y,z) = uint8(boostLevel);
                        end

                    end
                end
            end

        end

        function imProcess = bitPlaneSlicer (imProcess)

            imProcess.imgProcessed = imProcess.img;

            for z = 1:1:size(imProcess.imgProcessed, 3);
                for x = 1:1:size(imProcess.imgProcessed, 1);
                    for y = 1:1:size(imProcess.imgProcessed, 2);

                        if imProcess.imgProcessed(x,y,z) <= ((2^imProcess.bitPlane) - 1)
&& imProcess.imgProcessed(x,y,z) >= ((2^(imProcess.bitPlane-1)) - 1)
                            imProcess.imgProcessed(x,y,z) =
imProcess.imgProcessed(x,y,z);
                        else
                            imProcess.imgProcessed(x,y,z) = 0;
                        end

                    end
                end
            end

        end

        % ================== end of Piecewise linear transformation =======

        % ================== functions for problem statement 4 ============

        function imProcess = getNegative(imProcess)

            for z = 1:1:size(imProcess.img, 3);
                for x = 1:1:size(imProcess.img, 1);
                    for y = 1:1:size(imProcess.img, 2);
                        imProcess.imgProcessed(x,y,z) = 255 - imProcess.img(x,y,z);
                    end
                end
            end

            imProcess.showProcessed;
```

```
        end     % end of getNegative method of imProcessor...

        function imProcess = getLogTransform(imProcess)

            for z = 1:1:size(imProcess.img, 3);
                for x = 1:1:size(imProcess.img, 1);
                    for y = 1:1:size(imProcess.img, 2);
                        imProcess.imgProcessed(x,y,z) = uint8(log(1 +
double(imProcess.img(x,y,z))));
                    end
                end
            end
            imProcess.showProcessed;
        end

        function imProcess = getPowerTransform(imProcess)

            imProcess.imgProcessed = uint8(imProcess.constant .* (double(imProcess.img)
.^ imProcess.gam_ma));
            imProcess.showProcessed;
        end

        % ================= end of problem statement 4 ==================

        % ================= solution of Prob 5 ==========================

        function imProcess = sumOfArray (imProcess, myArray) % this function sums all the
elements in the array...

            imProcess.arraySum = 0;
            for z = 1:1:size(myArray, 3)
                for x = 1:1:size(myArray, 1)
                    for y = 1:1:size(myArray, 2)
                        imProcess.arraySum = imProcess.arraySum + myArray(x,y,z); %sum of
elements
                    end
                end
            end

        end

        function imProcess = getKernelSize (imProcess) % this function gets threshold
value of pixel for thresholding...

            options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
            imProcess.kernelSize = str2double(inputdlg({'Enter size of Kernel (Odd number
only)...'},...
                                                'Kernel Size', [1 50], {'3'}, options));

        end

        function imProcess = smoothByBox(imProcess) % uses box filter to smooth the
image...

            imProcess = imProcess.getKernelSize;
            boxKernel = ones(imProcess.kernelSize);
            imProcess = imProcess.sumOfArray(boxKernel);
            mySum = imProcess.arraySum;
            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]);
            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
```

```
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));
                        imProcess = imProcess.sumOfArray((boxKernel/mySum) .*
double(piece));
                        imProcess.imgProcessed(x,y,z) = imProcess.arraySum;
                    end
                end
            end
            imProcess.showProcessed;
        end

        function imProcess = differenceDetector (imProcess, anyArray)

            rawArray = zeros(size(anyArray, 1), size(anyArray, 2));

            for x = 1:1:size(anyArray, 1)
                for y = 1:1:size(anyArray, 2)

                    rawArray(x,y)= anyArray(x,y) - anyArray(size(anyArray, 1) -
(size(anyArray, 1)-1)/2, size(anyArray, 2) - (size(anyArray, 2)-1)/2);

                end
            end

            imProcess.arrayOfDifference = rawArray;

        end

        function imProcess = fuzzyEdgeDetection (imProcess)

            % The function detects edge of an image using fuzzy edge
            % detection technique.

            imProcess = imProcess.getKernelSize;
            boxKernel = ones(imProcess.kernelSize);
            imProcess.imgProcessed = imProcess.img;
            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]);

            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));
                        imProcess = imProcess.differenceDetector(double(piece) .*
boxKernel);

                        if imProcess.arrayOfDifference(1,2) == 0 &&
imProcess.arrayOfDifference(2,3) == 0;
                            imProcess.imgProcessed(x, y, z) = 255;
                        elseif imProcess.arrayOfDifference(2,3) == 0 &&
imProcess.arrayOfDifference(3,2) == 0;
                            imProcess.imgProcessed(x, y, z) = 255;
                        elseif imProcess.arrayOfDifference(3,2) == 0 &&
imProcess.arrayOfDifference(2,1) == 0;
                            imProcess.imgProcessed(x, y, z) = 255;
                        elseif imProcess.arrayOfDifference(2,1) == 0 &&
imProcess.arrayOfDifference(1,2) == 0;
                            imProcess.imgProcessed(x, y, z) = 255;
                        else
                            imProcess.imgProcessed(x, y, z) = 0;
                        end
```

```
                    end
                end
            end

        end

        function imProcess = sobelEdgeDetector (imProcess)

            % The function detects edge of the image using sobel operator.

            sobelX = [-1 -2 -1;0 0 0;1 2 1];    sobelY = [-1 0 1;-2 0 2;-1 0 1];
            boxKernel = ones(3);

            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]);

            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));
                        imProcess = imProcess.sumOfArray(double(piece).*sobelX);    gX =
imProcess.arraySum;
                        imProcess = imProcess.sumOfArray(double(piece).*sobelY);    gY =
imProcess.arraySum;
                        imProcess.imgProcessed(x, y, z) = sqrt((gX^2)+(gY^2));

                    end
                end
            end

        end

        function imProcess = getGaussKernelSize (imProcess) % selects size of the
gaussian kernel...

            prompt = {'Enter size of the filter (only Odd number):',...
                      'Enter constant, K:', 'Enter Sigma:'};
            title = 'Gaussian Filter Parameters';
            num_lines = [1 50];
            default_ans = {'3', '1', '1'};
            options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
            kernelParameters = inputdlg(prompt, title, num_lines, default_ans, options);
            order = str2double(kernelParameters{1,:});  k =
str2double(kernelParameters{2,:});  sig_ma = str2double(kernelParameters{3,:});

            filterGauss = zeros(order);

            for x = 1:1:size(filterGauss, 1)
                for y = 1:1:size(filterGauss, 2)

                    radius = sqrt((x-(size(filterGauss, 1)-((size(filterGauss, 1)-
1)/2)))^2 + (y-(size(filterGauss, 2)-((size(filterGauss, 2)-1)/2)))^2);
                    filterGauss(x, y) = k * exp(-(radius^2)/(2*sig_ma^2));

                end
            end

            imProcess.gaussKernel = filterGauss;
            disp(imProcess.gaussKernel);
```

```
        end

        function imProcess = smoothByGauss (imProcess) % smooths by the gaussian filter
kernel...

            imProcess = imProcess.getGaussKernelSize;
            imProcess = imProcess.sumOfArray(imProcess.gaussKernel);
            mySum = imProcess.arraySum;
            newImage = padarray(imProcess.img, [(size(imProcess.gaussKernel, 1) - 1)/2
(size(imProcess.gaussKernel, 2) - 1)/2]);
            imProcess.gaussKernel = rot90(imProcess.gaussKernel, 2);

            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(imProcess.gaussKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(imProcess.gaussKernel, 2)-1));
                        piece = newImage(x:x+(size(imProcess.gaussKernel, 1)-1),
y:y+(size(imProcess.gaussKernel, 2)-1));
                        imProcess = imProcess.sumOfArray((imProcess.gaussKernel/mySum) .*
double(piece));
                        imProcess.imgProcessed(x,y,z) = imProcess.arraySum;
                    end
                end
            end

            %imProcess = imProcess.showProcessed;

        end

        function imProcess = createLaplacianKernelFilter (imProcess) % creates laplacian
filter kernel...

            imProcess = imProcess.getKernelSize;
            laplacianKernels = {'Laplacian Kernel : Type 1', 'Laplacian Kernel : Type
2',...
                                'Laplacian Kernel : Type 3', 'Laplacian Kernel : Type
4'};
            [s, v] = listdlg('ListString', laplacianKernels, 'SelectionMode',
'single',...
                             'ListSize', [250 100], 'InitialValue', [1], 'Name',
'Laplacian Kernel',...
                             'PromptString', 'Please select type of Laplacian Kernel
:',...
                             'OKString', 'Select', 'CancelString', 'Cancel');

            if v == 1;

                switch s;

                    case 1; % type B laplacian kernel including diagonal terms (c = -
1)...
                        lapKernel = ones(imProcess.kernelSize);
                        imProcess = imProcess.sumOfArray(lapKernel);
                        total = imProcess.arraySum - 1;
                        lapKernel(size(lapKernel, 1) - (size(lapKernel, 1) - 1)/2,
size(lapKernel, 2) - (size(lapKernel, 2) - 1)/2) = -total;
                        imProcess.laplacianKernel = lapKernel;
                        imProcess.laplacianKernelType = 'B';

                    case 2; % type D laplacian kernel including diagonal terms (c = 1)...
                        lapKernel = ones(imProcess.kernelSize);
                        imProcess = imProcess.sumOfArray(lapKernel);
                        total = imProcess.arraySum - 1;
                        lapKernel = -lapKernel;
```

```
                        lapKernel(size(lapKernel, 1) - (size(lapKernel, 1) - 1)/2,
size(lapKernel, 2) - (size(lapKernel, 2) - 1)/2) = total;
                        imProcess.laplacianKernel = lapKernel;
                        imProcess.laplacianKernelType = 'D';

                    case 3; % type A, normal laplacian kernel (c = -1)...
                        lapKernel = zeros(imProcess.kernelSize);
                        lapKernel(size(lapKernel, 1) - (size(lapKernel, 1) - 1)/2, :) =
1;
                        lapKernel(:, size(lapKernel, 2) - (size(lapKernel, 2) - 1)/2) =
1;
                        imProcess = imProcess.sumOfArray(lapKernel);
                        total = imProcess.arraySum - 1;
                        lapKernel(size(lapKernel, 1) - (size(lapKernel, 1) - 1)/2,
size(lapKernel, 2) - (size(lapKernel, 2) - 1)/2) = -total;
                        imProcess.laplacianKernel = lapKernel;
                        imProcess.laplacianKernelType = 'A';

                    case 4; % type C, other normal laplacian kernel (c = 1)...
                        lapKernel = zeros(imProcess.kernelSize);
                        lapKernel(size(lapKernel, 1) - (size(lapKernel, 1) - 1)/2, :) =
1;
                        lapKernel(:, size(lapKernel, 2) - (size(lapKernel, 2) - 1)/2) =
1;
                        imProcess = imProcess.sumOfArray(lapKernel);
                        total = imProcess.arraySum - 1;
                        lapKernel = -lapKernel;
                        lapKernel(size(lapKernel, 1) - (size(lapKernel, 1) - 1)/2,
size(lapKernel, 2) - (size(lapKernel, 2) - 1)/2) = total;
                        imProcess.laplacianKernel = lapKernel;
                        imProcess.laplacianKernelType = 'C';

                    otherwise
                        disp('Please select Gaussian Kernel of right size ...');
                end

                disp(imProcess.laplacianKernel);

            end

        end

        function imProcess = sharpenByLaplacian(imProcess) % this function sharpens the
image by laplacian kernel...

            imProcess = imProcess.createLaplacianKernelFilter;
            newImage = padarray(imProcess.img, [(size(imProcess.laplacianKernel, 1) -
1)/2 (size(imProcess.laplacianKernel, 2) - 1)/2]);
            imProcess.laplacianKernel = rot90(imProcess.laplacianKernel, 2);

            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(imProcess.laplacianKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(imProcess.laplacianKernel, 2)-
1));
                        piece = newImage(x:x+(size(imProcess.laplacianKernel, 1)-1),
y:y+(size(imProcess.laplacianKernel, 2)-1));
                        imProcess = imProcess.sumOfArray(imProcess.laplacianKernel .*
double(piece));
                        imProcess.imgProcessed(x,y,z) = imProcess.arraySum;
                    end
                end
            end
            figure;
```

```
            imProcess = imProcess.showProcessed;
            switch imProcess.laplacianKernelType;
                case 'A'
                    imProcess.imgProcessed = double(imProcess.img) -
imProcess.imgProcessed;
                case 'B'
                    imProcess.imgProcessed = double(imProcess.img) -
imProcess.imgProcessed;
                case 'C'
                    imProcess.imgProcessed = double(imProcess.img) +
imProcess.imgProcessed;
                case 'D'
                    imProcess.imgProcessed = double(imProcess.img) +
imProcess.imgProcessed;
                otherwise
                    %do nothing
            end

        end

        function imProcess = unsharpMasking(imProcess) % does unsharp masking...

            smoothingMethod = {'Box Kernel', 'Gaussian Kernel'};
            [s, v] = listdlg('ListString', smoothingMethod, 'SelectionMode', 'single',...
                             'ListSize', [250 100], 'InitialValue', [1], 'Name', 'Gaussian
Kernel',...
                             'PromptString', 'Please select type of Kernel for smoothing
:',...
                             'OKString', 'Select', 'CancelString', 'Cancel');
            if v == 1;

                switch s;
                    case 1;
                        imProcess = imProcess.smoothByBox;
                    case 2;
                        imProcess = imProcess.smoothByGauss;
                    otherwise
                        % do nothing
                end
            end

            imProcess.imgProcessed = double(imProcess.img) -
double(imProcess.imgProcessed); % mask
            figure;
            imProcess = imProcess.showProcessed; % show mask
            imProcess.imgProcessed = double(imProcess.img) +
double(imProcess.imgProcessed);

        end

        % ========== Frequency Domain analysis of Image ==================

        function imProcess = getCutOff (imProcess) % this function gets cutoff
frequency...

            options.Resize='on';
            options.WindowStyle='normal';
            options.Interpreter='tex';
            imProcess.cutOff = str2double(inputdlg({'Enter cut-off frequency ...'},...
                                        'Cut-Off Frequency', [1 50], {'.25'},
options));

        end
```

```
        function imProcess = getLowHighPassFilter (imProcess)

            [m,n] = freqspace([2*size(imProcess.img, 1) 2*size(imProcess.img, 2)],
'meshgrid');
            filterMask = zeros(2*size(imProcess.img, 1), 2*size(imProcess.img, 2));

            kernelType = {'Ideal', 'Gaussian', 'Butterworth', 'Laplacian'};
            [s, v] = listdlg('ListString', kernelType, 'SelectionMode', 'single',...
                             'ListSize', [250 100], 'InitialValue', [1], 'Name', 'Filter
Type',...
                             'PromptString', 'Please select type of Filter for processing
:',...
                             'OKString', 'Select', 'CancelString', 'Cancel');

            if v == 1;

                switch s;
                    case 1; % Ideal filter will be selected...

                        imProcess = imProcess.getCutOff; % gets cut off frequency...

                        for x = 1:1:size(filterMask, 1)
                            for y = 1:1:size(filterMask, 2)
                                filterElement = sqrt(m(x,y)^2 + n(x,y)^2);

                                if filterElement <= imProcess.cutOff;
                                    filterMask(x,y) = 1;
                                elseif filterElement > imProcess.cutOff;
                                    filterMask(x,y) = 0;
                                else
                                    filterMask(x,y) = 0;
                                end
                                %lowPassFilter(x,y) = sqrt(m(x,y)^2 + n(x,y)^2);
                            end
                        end

                        imProcess.lowPass = filterMask;
                        imProcess.highPass = 1 - filterMask;

                    case 2; % Gaussian filter will be selected...

                        imProcess = imProcess.getCutOff; % gets cut off frequency...

                        for x = 1:1:size(filterMask, 1)
                            for y = 1:1:size(filterMask, 2)
                                D_square = m(x,y)^2 + n(x,y)^2;
                                filterMask(x, y) = exp(-D_square /
2*(imProcess.cutOff^2));

                            end
                        end

                        imProcess.lowPass = filterMask;
                        imProcess.highPass = 1 - filterMask;

                    case 3; % Butterworth filter will be selected...

                        imProcess = imProcess.getCutOff; % gets cut off frequency...

                        options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
```

```
                        filterOrder = str2double(inputdlg({'Enter order of Butterworth
Filter ...'},...
                            'Order of Filter', [1 50], {'1'}, options)); % order of
filter, n

                        for x = 1:1:size(filterMask, 1)
                            for y = 1:1:size(filterMask, 2)
                                D = sqrt(m(x,y)^2 + n(x,y)^2);
                                filterMask(x, y) = 1/(1 +
(D/imProcess.cutOff)^(2*filterOrder));

                            end
                        end

                        imProcess.lowPass = filterMask;
                        imProcess.highPass = 1 - filterMask;

                    case 4; % Laplacian will be selected...

                        for x = 1:1:size(filterMask, 1)
                            for y = 1:1:size(filterMask, 2)
                                D_square = m(x,y)^2 + n(x,y)^2;
                                filterMask(x, y) = 1 + 4*(pi^2)*D_square;

                            end
                        end

                        imProcess.laplacianFDomain = filterMask;

                    otherwise
                        % do nothing
                end
            end

        end

        function imProcess = fourierFiltering (imProcess)

            %imProcess.img = im2double(imProcess.img);
            imProcess.img = double(imProcess.img);
            paddedImg = padarray(imProcess.img, [size(imProcess.img, 1)
size(imProcess.img,2)], 'post');

            for x = 1:1:size(paddedImg, 1)
                for y = 1:1:size(paddedImg, 2)
                    paddedImg(x, y) = paddedImg(x, y)*((-1)^(x + y));
                end
            end

            imgFouriered = fft2(paddedImg);
            processType = {'Smoothing', 'Sharpening', 'Use Laplacian', 'Unsharp
Masking',...
                'High-Boost Filtering','High-Frequency-Emphasis Filtering', 'More general
High-Frequency-Emphasis Filtering'};
            [s, v] = listdlg('ListString', processType, 'SelectionMode', 'single',...
                            'ListSize', [250 100], 'InitialValue', [1], 'Name',
'Processing Type',...
                            'PromptString', 'Please select type of processing :',...
                            'OKString', 'Select', 'CancelString', 'Cancel');

            if v == 1;

                switch s;
```

```
                    case 1; % low pass filtering is selected
                        processedImg = imgFouriered .* imProcess.lowPass;
                    case 2; % highpass filtering is selected
                        processedImg = imgFouriered .* imProcess.highPass;
                    case 3; % laplacian is selected
                        processedImg = imgFouriered .* imProcess.laplacianFDomain;
                    case 4; % low pass filtering for unsharp masking is selected...
                        processedImg = imgFouriered .* imProcess.lowPass;
                    case 5; % low pass filtering for highboost filtering is selected...
                        processedImg = imgFouriered .* imProcess.lowPass;
                    case 6; % high pass filtering for high-freq-emphasis filtering is
selected...
                        options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
                        kHiFreq = str2double(inputdlg({'Enter K for High-Frequency-
Emphasis Filtering (K > 1)'},...
                            'High-Freq Emphasizer', [1 60], {'0.25'}, options));
                        processedImg = imgFouriered .* (1 + kHiFreq *
imProcess.highPass);
                    case 7; % high pass filtering for more general high-freq-emphasis
filtering is selected...

                        prompt = {'Enter k1 >= 0, for High-Frequency-Emphasis
Filtering:',...
                            'Enter k2 > 0, for High-Frequency-Emphasis Filtering:'};
                        title = 'High-Freq-Emphasis Parameters';
                        num_lines = [1 60];
                        default_ans = {'0.5', '0.75'};
                        options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
                        kHiFreqPara = inputdlg(prompt, title, num_lines, default_ans,
options);
                        k1 = str2double(kHiFreqPara{1,:});      k2 =
str2double(kHiFreqPara{2,:});

                        processedImg = imgFouriered .* (k1 + (k2 * imProcess.highPass));
                    otherwise
                        % do nothing
                end
            end

            processedImgReconstrctd = ifft2(processedImg);

            for x = 1:1:size(processedImgReconstrctd, 1)
                for y = 1:1:size(processedImgReconstrctd, 2)

                    processedImgReconstrctd(x,y) = processedImgReconstrctd(x, y)*((-
1)^(x+y));
                end
            end

            switch s;
                case 1;
                    imProcess.imgProcessed =
processedImgReconstrctd(1:size(imProcess.img, 1), 1:size(imProcess.img, 2));
                case 2;
                    imProcess.imgProcessed = imProcess.img +
processedImgReconstrctd(1:size(imProcess.img, 1), 1:size(imProcess.img, 2));
                case 3;
                    imProcess.imgProcessed =
processedImgReconstrctd(1:size(imProcess.img, 1), 1:size(imProcess.img, 2));
                case 4;
```

```
                    gMask = imProcess.img - processedImgReconstrctd(1:size(imProcess.img,
1), 1:size(imProcess.img, 2));
                    imProcess.imgProcessed = imProcess.img + gMask;
                case 5;
                    options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
                    kMask = str2double(inputdlg({'Enter K for High-boost Filtering (K >
1)'},...
                        'High-Boost Multiplier', [1 50], {'1.25'}, options));

                    gMask = imProcess.img - processedImgReconstrctd(1:size(imProcess.img,
1), 1:size(imProcess.img, 2));
                    imProcess.imgProcessed = imProcess.img + kMask*gMask;

                case 6;
                    imProcess.imgProcessed =
processedImgReconstrctd(1:size(imProcess.img, 1), 1:size(imProcess.img, 2));
                case 7;
                    imProcess.imgProcessed =
processedImgReconstrctd(1:size(imProcess.img, 1), 1:size(imProcess.img, 2));
            end
        end

        function imProcess = getBandParameters (imProcess)

            prompt = {'Enter the center of the Band, c0: ',...
                'Enter the width of the band, W: '};
            title = 'Band Parameters';
            num_lines = [1 60];
            default_ans = {'0.5', '0.5'};
            options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
            bandPara = inputdlg(prompt, title, num_lines, default_ans, options);
            imProcess.c0 = str2double(bandPara{1,:});      imProcess.W =
str2double(bandPara{2,:});

        end

        function imProcess = getBandPassRejectFilter (imProcess)

            imProcess = imProcess.getBandParameters;
            [m, n] = freqspace([2*size(imProcess.img, 1) 2*size(imProcess.img, 2)],
'meshgrid');
            filterMask = zeros(2*size(imProcess.img, 1), 2*size(imProcess.img, 2));

            filterType = {'Ideal Filter', 'Gaussian Filter', 'Butterworth Filter'};
            [s, v] = listdlg('ListString', filterType, 'SelectionMode', 'single',...
                'ListSize', [250 100], 'InitialValue', [1], 'Name', 'Filter Type',...
                'PromptString', 'Please select type of Filter for processing :',...
                'OKString', 'Select', 'CancelString', 'Cancel');

            if v == 1;

                switch s;

                    case 1; % Ideal filter will be selected

                        for x = 1:1:size(filterMask, 1)
                            for y = 1:1:size(filterMask, 2)
                                filterElement = sqrt(m(x,y)^2 + n(x,y)^2);

                                if (filterElement >= (imProcess.c0 - (imProcess.W / 2)))
&& (filterElement <= (imProcess.c0 + (imProcess.W / 2)))
```

```
                                filterMask(x, y) = 0;
                            else
                                filterMask(x, y) = 1;
                            end

                        end
                    end

                    imProcess.bandReject = filterMask;
                    imProcess.bandPass = 1 - filterMask;

                case 2; % Gaussian filter will be selected

                    for x = 1:1:size(filterMask, 1)
                        for y = 1:1:size(filterMask, 2)
                            D_square = m(x,y)^2 + n(x,y)^2;
                            filterMask(x, y) = 1 - exp(-(((D_square-
imProcess.c0^2)/(imProcess.W*sqrt(D_square)))^2));

                        end
                    end

                    imProcess.bandReject = filterMask;
                    imProcess.bandPass = 1 - filterMask;

                case 3; % Butterworth filter will be selected

                    options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
                    filterOrder = str2double(inputdlg({'Enter order of Butterworth
Filter ...'},...
                        'Order of Filter', [1 50], {'1'}, options)); % order of
filter, n

                    for x = 1:1:size(filterMask, 1)
                        for y = 1:1:size(filterMask, 2)
                            D = sqrt(m(x,y)^2 + n(x,y)^2);
                            filterMask(x, y) = 1/(1 + (((D*imProcess.W)/(D^2-
imProcess.c0^2))^(2*filterOrder)));

                        end
                    end

                    imProcess.bandReject = filterMask;
                    imProcess.bandPass = 1 - filterMask;

                otherwise
                    % do nothing...
            end
        end

        figure;
        imshow(imProcess.bandReject);
        figure;
        imshow(imProcess.bandPass);

    end

    function imProcess = fourierBandSelectFiltering (imProcess)

        %imProcess.img = im2double(imProcess.img);
        imProcess.img = double(imProcess.img);
```

```
            paddedImg = padarray(imProcess.img, [size(imProcess.img, 1)
size(imProcess.img,2)], 'post');

            for x = 1:1:size(paddedImg, 1)
                for y = 1:1:size(paddedImg, 2)
                    paddedImg(x, y) = paddedImg(x, y)*((-1)^(x + y));
                end
            end

            imgFouriered = fft2(paddedImg);

            processType = {'Band Reject Filtering', 'Band Pass Filtering'};
            [s, v] = listdlg('ListString', processType, 'SelectionMode', 'single',...
                             'ListSize', [250 100], 'InitialValue', [1], 'Name',
'Filtering Type',...
                             'PromptString', 'Please select type of Filtering :',...
                             'OKString', 'Select', 'CancelString', 'Cancel');

            if v == 1;

                switch s;
                    case 1; % band reject filtering is selected
                        processedImg = imgFouriered .* imProcess.bandReject;
                    case 2; % band pass filtering is selected
                        processedImg = imgFouriered .* imProcess.bandPass;

                    otherwise
                        % do nothing
                end
            end

            processedImgReconstrctd = ifft2(processedImg);

            for x = 1:1:size(processedImgReconstrctd, 1)
                for y = 1:1:size(processedImgReconstrctd, 2)

                    processedImgReconstrctd(x,y) = processedImgReconstrctd(x, y)*((-
1)^(x+y));
                end
            end

            switch s;
                case 1;
                    imProcess.imgProcessed =
processedImgReconstrctd(1:size(imProcess.img, 1), 1:size(imProcess.img, 2));
                case 2;
                    imProcess.imgProcessed =
processedImgReconstrctd(1:size(imProcess.img, 1), 1:size(imProcess.img, 2));
                    %imProcess.imgProcessed = imProcess.img +
processedImgReconstrctd(1:size(imProcess.img, 1), 1:size(imProcess.img, 2));
                otherwise
                    % do nothing...
            end

        end

        % ============ end of Frequency Domain Analysis ==================

        % ================= end of solution of Prob. 5 ===================

        % ================= Problem 6: Noise reduction ===================

        function imProcess = productOfArray (imProcess, myArray)
```

```
            % The function performs product of elements of the array

            product = 1;

            for x = 1:1:size(myArray, 1);
                for y = 1:1:size(myArray, 2);

                    product = product * myArray(x,y);
                end
            end

            imProcess.arrayProduct = product;

        end

        function imProcess = arithmeticMeanFilter (imProcess) % smoothens local
variations in the image...

            imProcess = imProcess.getKernelSize;
            boxKernel = ones(imProcess.kernelSize);
            myProduct = size(boxKernel, 1) * size(boxKernel, 2);
            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]);
            boxKernel = rot90(boxKernel, 2);    % rotates box kernel by 180 degrees.

            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));
                        imProcess = imProcess.sumOfArray((boxKernel/myProduct) .*
double(piece));
                        imProcess.imgProcessed(x,y,z) = imProcess.arraySum;
                    end
                end
            end

        end

        function imProcess = geometricMeanFilter (imProcess) % smoothens local variations
better than arithmetic mean filter...

            imProcess = imProcess.getKernelSize;
            boxKernel = ones(imProcess.kernelSize);
            myProduct = size(boxKernel, 1) * size(boxKernel, 2);
            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]);
            boxKernel = rot90(boxKernel, 2);
            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));
                        imProcess = imProcess.productOfArray(boxKernel .* double(piece));
                        imProcess.imgProcessed(x,y,z) =
(imProcess.arrayProduct^(1/myProduct));
                    end
                end
            end

        end
```

```
        function imProcess = harmonicMeanFilter (imProcess)

            % The harmonic mean filter works well for for salt noise but
            % fails for pepper noise. It does well also with other types of
            % noise like Gaussian noise...

            imProcess = imProcess.getKernelSize;
            boxKernel = ones(imProcess.kernelSize);
            myProduct = size(boxKernel, 1) * size(boxKernel, 2);
            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]);
            boxKernel = rot90(boxKernel, 2);

            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));
                        imProcess = imProcess.sumOfArray(boxKernel ./ double(piece));
                        imProcess.imgProcessed(x,y,z) = myProduct/imProcess.arraySum;
                    end
                end
            end

        end

        function imProcess = contraHarmonicMeanFilter (imProcess)

            % Here Q, in this case, 'filterOrder' is called as order of the
            % filter. This filter is well suited for reducing or virtually
            % eliminating the effects of salt-and-pepper noise.

            % For positive values of Q, filter eliminates pepper noise. For
            % negative values of Q, the filter eliminates salt noise. It
            % cannot do both simultaneously.

            % Note that contraharmonic reduces to the arithmetic mean
            % filter if Q = 0, and harmonic mean filter if Q = -1.

            prompt = {'Enter size of the filter (only Odd number):', 'Enter order of the
Filter, Q :'};
            title = 'Contraharmonic Mean Filter Parameters';
            num_lines = [1 50];
            default_ans = {'3', '1'};
            options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
            filterParameters = inputdlg(prompt, title, num_lines, default_ans, options);
            filterSize = str2double(filterParameters{1,:}); filterOrder =
str2double(filterParameters{2,:});

            boxKernel = ones(filterSize);
            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]);
            boxKernel = rot90(boxKernel, 2);

            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));
                        region = double(piece) .* boxKernel;
                        imProcess = imProcess.sumOfArray(region .^ filterOrder);
                        denominator = imProcess.arraySum;
```

```
                        imProcess = imProcess.sumOfArray(region .^ (filterOrder + 1));
                        numerator = imProcess.arraySum;
                        imProcess.imgProcessed(x,y,z) = numerator/denominator;
                    end
                end
            end

        end

        function imProcess = orderStatFiltParamtr (imProcess, array)

            % This function sorts the array elements in linear manner. The
            % linear sorted array is used to find out ascending array,
            % median of the array, min value of array, max value of the
            % array, midpoint of the array etc.

            counter = 0;

            for x = 1:1:size(array, 1)
                for y = 1:1:size(array, 2)

                    counter = counter + 1;
                    sortArray(counter) = array(x, y);

                end
            end

            imProcess.sortedArray = sort(sortArray);
            imProcess.medianIntensity = median(imProcess.sortedArray);
            imProcess.minIntensity = min(imProcess.sortedArray);
            imProcess.maxIntensity = max(imProcess.sortedArray);
            imProcess.midIntensity = (imProcess.minIntensity + imProcess.maxIntensity)/2;

        end

        function imProcess = arrayTrimmer (imProcess, arrayInput, trimFactor) % this
function trims the array for alpha-trimmed

            for iteration = 1:1:trimFactor/2

                arrayInput(:,iteration) = 0;
                arrayInput(:, size(arrayInput,2) -(iteration) +1) = 0;

            end

            imProcess.alphaTrimmedArray = arrayInput;
        end

        function imProcess = orderStatisticsFilter (imProcess)

            % The function performs order statistics filtering on given
            % images. Created by Rohit.

            imProcess = imProcess.getKernelSize;
            %imProcess.imgProcessed = imProcess.img;
            boxKernel = ones(imProcess.kernelSize);

            filterType = {'Median Filter', 'Max Filter', 'Min Filter', 'Midpoint Filter',
'Alpha-trimmed Mean Filter'};
            [s, v] = listdlg('ListString', filterType, 'SelectionMode', 'single',...
                             'ListSize', [250 100], 'InitialValue', [1], 'Name', 'Filter
Type',...
```

```
                            'PromptString', 'Please select type of Filter for processing
:',...
                            'OKString', 'Select', 'CancelString', 'Cancel');

            if v == 1;
                switch s;
                    case 5;
                        options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
                        trimFactor = str2double(inputdlg({'Enter trim factor, "d" for
alpha-trimmed filter (only Even number):'},...
                            'Trim Factor', [1 50], {'2'}, options));
                end
            end

            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]);
            boxKernel = rot90(boxKernel, 2);

            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));
                        imProcess = imProcess.orderStatFiltParamtr((boxKernel) .*
double(piece));

                        if v == 1;

                            switch s;
                                case 1; % Median filter is selected...
                                    imProcess.imgProcessed(x,y,z) =
imProcess.medianIntensity;
                                case 2; % Max filter is selected...
                                    imProcess.imgProcessed(x,y,z) =
imProcess.maxIntensity;
                                case 3; % Min filter is selected...
                                    imProcess.imgProcessed(x,y,z) =
imProcess.minIntensity;
                                case 4; % Midpoint filter is selected...
                                    imProcess.imgProcessed(x,y,z) =
imProcess.midIntensity;
                                case 5; % Alpha-trimmed Mean filter is selected...
                                    imProcess =
imProcess.arrayTrimmer(imProcess.sortedArray, trimFactor);
                                    imProcess =
imProcess.sumOfArray(imProcess.alphaTrimmedArray);
                                    imProcess.imgProcessed(x,y,z) = (1/((size(boxKernel,
1)*size(boxKernel, 2))-trimFactor))*imProcess.arraySum;
                                otherwise
                                    % do nothing...
                            end
                        end

                    end
                end
            end

        end

        % ============== end of Problem 6: Noise reduction ==============

        % ============== Morphological Image Processing =================
```

```
        function imProcess = erosion(imProcess)

            % This function performs erosion of images....

            %imProcess = imProcess.getKernelSize; % gets size of the kernel...
            boxKernel = ones(imProcess.kernelSize); % generates box kernel of obtained
size...
            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]); % pads new image...
            %boxKernel = rot90(boxKernel, 2);
            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));
                        imProcess = imProcess.orderStatFiltParamtr(boxKernel .*
double(piece));
                        imProcess.imgProcessed(x, y, z) = imProcess.minIntensity;

                    end
                end
            end

            if imProcess.imgInfo.BitDepth == 1; % checks if given image is binary
image...
                imProcess.imgProcessed = logical(imProcess.imgProcessed); % converts
uint8 array to logical array.
            end

        end

        function imProcess = dilation (imProcess)

            % This function performs dilation on the images....

            %imProcess = imProcess.getKernelSize; % gets size of the kernel...
            boxKernel = ones(imProcess.kernelSize); % generates box kernel of obtained
size...
            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]); % pads new image...
            boxKernel = rot90(boxKernel, 2);
            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));
                        imProcess = imProcess.orderStatFiltParamtr(boxKernel .*
double(piece));
                        imProcess.imgProcessed(x, y, z) = imProcess.maxIntensity;

                    end
                end
            end

            if imProcess.imgInfo.BitDepth == 1; % checks if given image is binary
image...
                imProcess.imgProcessed = logical(imProcess.imgProcessed); % converts
uint8 array to logical array.
            end

        end
```

```
        function imProcess = opening (imProcess)

            % This function performs both morphological and grayscale
            % opening of images.

            imProcess.auxImg = imProcess.img;    % storing main image in auxiliary image
variable...
            imProcess = imProcess.erosion;       % performing erosion on main image...
            imProcess.img = imProcess.imgProcessed; % storing processed image into main
image property for next dilation operation...
            imProcess = imProcess.dilation;      % performing dlation operation...
            imProcess.img = imProcess.auxImg;    % storing original image in main image
property...
            %imProcess = imProcess.showOriginal;
            %figure;
            %imProcess = imProcess.showProcessed;

        end

        function imProcess = closing (imProcess)

            % This function performs both morphological and grayscale
            % closing of images.

            imProcess.auxImg = imProcess.img;    % storing main image in auxiliary
property...
            imProcess = imProcess.dilation;      % performing dilation on main image...
            imProcess.img = imProcess.imgProcessed; % storing processed image into main
image property for next erosion operation...
            imProcess = imProcess.erosion;       % performing erosion operation...
            imProcess.img = imProcess.auxImg;    % storing original image in main image
property...

        end

        function imProcess = boundaryExtraction (imProcess)

            % This function extracts boundary from images.

            imProcess = imProcess.erosion;
            imProcess.imgProcessed = double(imProcess.img) -
double(imProcess.imgProcessed);

        end

        function imProcess = morphologicalGradient (imProcess)

            % This function gets morphological gradient of an image...

            imProcess = imProcess.dilation;
            imProcess.auxImg = imProcess.imgProcessed;
            imProcess = imProcess.erosion;
            imProcess.imgProcessed = imProcess.auxImg - imProcess.imgProcessed;

        end

        function imProcess = topHatTransform (imProcess)

            % this function performs the tophat transformation of an
            % image...

            imProcess = imProcess.opening;
```

```
            imProcess.imgProcessed = double(imProcess.img) - imProcess.imgProcessed;

        end

        function imProcess = bottomHatTransform (imProcess)

            % this function performs the bottomhat transformation of an
            % image...

            imProcess = imProcess.closing;
            imProcess.imgProcessed = imProcess.imgProcessed - double(imProcess.img);

        end

        function imProcess = contrastEnhance (imProcess)

            imProcess = imProcess.topHatTransform;
            a = imProcess.imgProcessed;
            imProcess = imProcess.bottomHatTransform;
            b = imProcess.imgProcessed;
            imProcess.imgProcessed = double(imProcess.img) + a - b;

        end

        function imProcess = morphSmoothing (imProcess)

            % The algorithm performs Morphological smoothin on the given
            % image. First it performs opening on the given input image,
            % then it performs closing on the result of opening.

            imProcess = imProcess.opening;
            imProcess.img = imProcess.imgProcessed;
            imProcess = imProcess.closing;


        end

        % =============== End of Morphological Image processing ===========

        % =============== Image Segmentation I ============================

        function imProcess = lineDetection (imProcess)

            % The function detects horizontal, vertical, +45 degree, -45
            % degree angle lines in the input image.

            lineAngle = {'Horizontal Line Detection', '+45 degree Line Detection',
'Vertical Line Detection', '-45 degree Line Detection'};
            [s, v] = listdlg('ListString', lineAngle, 'SelectionMode', 'single',...
                'ListSize', [250 100], 'InitialValue', [1], 'Name', 'Line Detection
Kernel',...
                'PromptString', 'Please select type of Kernel for line detection :',...
                'OKString', 'Select', 'CancelString', 'Cancel');

            if v == 1;

                switch s;
                    case 1;
                        boxKernel = [-1 -1 -1; 2 2 2; -1 -1 -1];
                    case 2;
                        boxKernel = [2 -1 -1; -1 2 -1; -1 -1 2];
                    case 3;
                        boxKernel = [-1 2 -1; -1 2 -1; -1 2 -1];
```

```
                    case 4;
                        boxKernel = [-1 -1 2; -1 2 -1; 2 -1 -1];
                    otherwise
                        % do nothing
                end
            end

            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]);

            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));
                        imProcess = imProcess.sumOfArray((boxKernel) .* double(piece));
                        imProcess.imgProcessed(x,y,z) = uint8(imProcess.arraySum);
                    end
                end
            end

            if imProcess.imgInfo.BitDepth == 1; % checks if given image is binary
image...
                imProcess.imgProcessed = logical(imProcess.imgProcessed); % converts
uint8 array to logical array.
            end

        end

        function imProcess = laplacianOperator (imProcess)

            imProcess = imProcess.createLaplacianKernelFilter;
            newImage = padarray(imProcess.img, [(size(imProcess.laplacianKernel, 1) -
1)/2 (size(imProcess.laplacianKernel, 2) - 1)/2]);
            imProcess.laplacianKernel = rot90(imProcess.laplacianKernel, 2);

            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(imProcess.laplacianKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(imProcess.laplacianKernel, 2)-
1));
                        piece = newImage(x:x+(size(imProcess.laplacianKernel, 1)-1),
y:y+(size(imProcess.laplacianKernel, 2)-1));
                        imProcess = imProcess.sumOfArray(imProcess.laplacianKernel .*
double(piece));
                        imProcess.imgProcessed(x,y,z) = imProcess.arraySum;
                    end
                end
            end

        end

        function imProcess = gradientOperator (imProcess)

            % The function operates gradient on the given image. The
            % gradient operator extracts horizontal component gX, vertical
            % component gY, vector and angle alpha.

            gradOperator = {'Roberts Cross Gradient', 'Prewitt Operator', 'Sobel
Operator'};
            [s, v] = listdlg('ListString', gradOperator, 'SelectionMode', 'single',...
                'ListSize', [250 100], 'InitialValue', [1], 'Name', 'Gradient
Operator',...
                'PromptString', 'Please select Gradient operator :',...
```

```
                'OKString', 'Select', 'CancelString', 'Cancel');

            if v == 1;

                switch s;
                    case 1;
                        gradX = [-1 0; 0 1];    gradY = [0 -1; 1 0];
                    case 2;
                        gradX = [-1 -1 -1; 0 0 0; 1 1 1];   gradY = [-1 0 1; -1 0 1; -1 0
1];
                    case 3;
                        gradX = [-1 -2 -1; 0 0 0; 1 2 1];   gradY = [-1 0 1; -2 0 2; -1 0
1];

                    otherwise
                        % do nothing
                end
            end

            boxKernel = ones(3);

            imProcess.arrayOfAlpha = double(zeros(size(imProcess.img)));

            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]);

            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));

                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));

                        if s == 1
                            imProcess = imProcess.sumOfArray(double(piece(2:3,
2:3)).*gradX);    gX = imProcess.arraySum;
                            imProcess = imProcess.sumOfArray(double(piece(2:3,
2:3)).*gradY);    gY = imProcess.arraySum;

                        elseif s == 2 || s == 3
                            imProcess = imProcess.sumOfArray(double(piece).*gradX);    gX
= imProcess.arraySum;
                            imProcess = imProcess.sumOfArray(double(piece).*gradY);    gY
= imProcess.arraySum;
                        end

                        imProcess.imgProcessed(x, y, z) = sqrt((gX^2)+(gY^2));
                        imProcess.arrayOfAlpha(x, y, z) = atan(gY/gX);

                    end
                end
            end


        end

        function imProcess = kirschCompassMask (imProcess)

            kirsch = {'North Direction Mask', 'North-West Direction Mask',...
                'West Direction Mask', 'South-West Direction Mask'...
                'South Direction Mask', 'South-East Direction Mask',...
                'East Direction Mask', 'North-East Direction Mask'};
```

```
            [s, v] = listdlg('ListString', kirsch, 'SelectionMode', 'single',...
                'ListSize', [250 120], 'InitialValue', [1], 'Name', 'Kirsch Compass
Mask',...
                'PromptString', 'Please select Compass mask :',...
                'OKString', 'Select', 'CancelString', 'Cancel');

            if v == 1;

                switch s;

                    case 1; % North directional mask is selected...
                        boxKernel = [-3 -3 5; -3 0 5; -3 -3 5];
                    case 2; % North-West directional mask is selected...
                        boxKernel = [-3 5 5; -3 0 5; -3 -3 -3];
                    case 3; % West direction mask is selected...
                        boxKernel = [5 5 5; -3 0 -3; -3 -3 -3];
                    case 4; % South-West direction mask is selected...
                        boxKernel = [5 5 -3; 5 0 -3; -3 -3 -3];
                    case 5; % South direction mask is selected...
                        boxKernel = [5 -3 -3; 5 0 -3; 5 -3 -3];
                    case 6; % South-East direction mask is selected...
                        boxKernel = [-3 -3 -3; 5 0 -3; 5 5 -3];
                    case 7; % East direction mask is selected...
                        boxKernel = [-3 -3 -3; -3 0 -3; 5 5 5];
                    case 8; % North-East direction mask is selected...
                        boxKernel = [-3 -3 -3; -3 0 5; -3 5 5];

                    otherwise
                        % do nothing
                end
            end

            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]);

            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));

                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1));
                        imProcess = imProcess.sumOfArray(double(piece).*boxKernel);
                        imProcess.imgProcessed(x, y, z) = imProcess.arraySum;

                    end
                end
            end

        end

        function imProcess = laplacianOfGaussian (imProcess)

            % The function creates laplacian of Gaussian in two types.
            % Tyepe 1 and Type 2. The meaning of type here is complementary
            % or negative.


            prompt = {'Enter size of the filter (only Odd number):',...
                      'Enter constant, K:', 'Enter Sigma:'};
            title = 'Gaussian Filter Parameters';
            num_lines = [1 50];
            default_ans = {'3', '1', '1'};
```

```
            options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
            kernelParameters = inputdlg(prompt, title, num_lines, default_ans, options);
            order = str2double(kernelParameters{1,:});  k =
str2double(kernelParameters{2,:});  sig_ma = str2double(kernelParameters{3,:});

            lapOfGaussKernel = {'Laplacian of Gaussian Kernel : Type 1', 'Laplacian of
Gaussian Kernel : Type 2'};

            [s, v] = listdlg('ListString', lapOfGaussKernel, 'SelectionMode',
'single',...
                             'ListSize', [250 100], 'InitialValue', [1], 'Name',
'Laplacian of Gaussian Kernel',...
                             'PromptString', 'Please select type of LOG Kernel :',...
                             'OKString', 'Select', 'CancelString', 'Cancel');

            filterGauss = zeros(order);

            for x = 1:1:size(filterGauss, 1)
                for y = 1:1:size(filterGauss, 2)

                    radius = sqrt((x-(size(filterGauss, 1)-((size(filterGauss, 1)-
1)/2)))^2 + (y-(size(filterGauss, 2)-((size(filterGauss, 2)-1)/2)))^2);
                    filterGauss(x, y) = k * exp(-(radius^2)/(2*sig_ma^2));

                end
            end

            %disp(filterGauss);

            if v == 1;

                switch s;

                    case 1; % The kernel with negative centerweight and positive
neighbours...

                        imProcess = imProcess.sumOfArray(filterGauss);
                        centerWeight = imProcess.arraySum - k;
                        filterGauss(size(filterGauss, 1) - (size(filterGauss, 1) - 1)/2,
size(filterGauss, 2) - (size(filterGauss, 2) - 1)/2) = -centerWeight;
                        imProcess.lapOfGauss = filterGauss;
                        imProcess.laplacianKernelType = 'B';

                    case 2; % The kernel with positive centerweight and negative
neighbours...

                        imProcess = imProcess.sumOfArray(filterGauss);
                        centerWeight = imProcess.arraySum - k;
                        filterGauss = -filterGauss;
                        filterGauss(size(filterGauss, 1) - (size(filterGauss, 1) - 1)/2,
size(filterGauss, 2) - (size(filterGauss, 2) - 1)/2) = centerWeight;
                        imProcess.lapOfGauss = filterGauss;
                        imProcess.laplacianKernelType = 'D';

                    otherwise
                        disp('Please select Gaussian Kernel of right size ...');
                end

                disp(imProcess.lapOfGauss);

            end
```

```
            %imProcess = imProcess.sumOfArray(imProcess.lapOfGauss);
            %disp(imProcess.arraySum);

        end

        function imProcess = marrHildrethEdgeDetection (imProcess)

            % The function uses Marr-Hildreth edge detection algorithm to
            % detect and extract edges from the input image.

            imProcess = imProcess.laplacianOfGaussian;
            %imProcess = imProcess.smoothByGauss;
            %imProcess.img = imProcess.imgProcessed;
            %imProcess = imProcess.createLaplacianKernelFilter;

            choice = questdlg('Would you like to perform thresholding on the output image
??',...
                  'Thresholding of Image', 'Yes', 'No', 'No');

            switch choice;

                case 'Yes';

                    prompt = {'Enter value of thresholding factor in percent:'};
                    title = 'Thresholdig Factor';
                    num_lines = [1 50];
                    default_ans = {'00.00'};
                    options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
                    thresholder = str2double(inputdlg(prompt, title, num_lines,
default_ans, options))/100;

                case 'No';

                    thresholder = 00.00;

                case '';

                    errordlg('Threshold option cannot be empty. Please select proper
thresholding option',...
                        'Error!!', 'modal');
                    return;
            end

            newImage = padarray(imProcess.img, [(size(imProcess.lapOfGauss, 1) - 1)/2
(size(imProcess.lapOfGauss, 2) - 1)/2]);
            absoluteGradient = double(zeros(size(imProcess.img)));

            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(imProcess.lapOfGauss, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(imProcess.lapOfGauss, 2)-1));
                        piece = newImage(x:x+(size(imProcess.lapOfGauss, 1)-1),
y:y+(size(imProcess.lapOfGauss, 2)-1));
                        imProcess = imProcess.sumOfArray(imProcess.lapOfGauss .*
double(piece));
                        absoluteGradient(x, y, z) = imProcess.arraySum;

                    end
                end
            end

            boxKernel = ones(3);
```

```
            newImage = padarray(absoluteGradient, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]);
            peakIntensity = max(max(absoluteGradient));

            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));
                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1)) .* boxKernel;

                        switch choice;

                            case {'Yes', 'No'};

                                if ((piece(1,1)*piece(3,3) < 0)||(piece(1,2)*piece(3,2) <
0)||(piece(1,3)*piece(3,1)<0)||(piece(2,1)*piece(2,3)<0))...
                                        && piece(2,2) >= thresholder * peakIntensity;

                                    imProcess.imgProcessed(x,y,z) = piece(2,2);
                                end

                            case '';
                                errordlg('Threshold option cannot be empty. Please select
proper thresholding option',...
                                    'Error!!', 'modal');
                                break;
                        end

                    end
                end
            end

        end


        function imProcess = cannyEdgeDetector (imProcess)

            % The function detects edge of objects in image using canny
            % algorithm.

            imProcess = imProcess.smoothByGauss;    % Step 1 of Canny. This function
smooths the image.
            imProcess.auxImg = imProcess.img;       % Storing the original image in dummy
variable.

            imProcess.img = double(imProcess.img);  % converting uint8 array to double
array
            imProcess.img = imProcess.imgProcessed; % storing smoothed image to main img
variable for gradient calculation.
            imProcess = imProcess.gradientOperator; % Calculating the gradient of the
image. The magnitude is stored in imProcess.imgProcessed

            imProcess.img = double(imProcess.img);  % converting imProcess.img to double
precision array again.
            imProcess.img = imProcess.imgProcessed; % storing array of gradients in the
imProcess.img for further operations.

            boxKernel = ones(3);

            gN = double(zeros(size(imProcess.img)));

            newImage = padarray(imProcess.img, [(size(boxKernel, 1) - 1)/2
(size(boxKernel, 2) - 1)/2]);
```

```
            for z = 1:1:size(newImage, 3);
                for x = 1:1:(size(newImage, 1) - (size(boxKernel, 1)-1));
                    for y = 1:1:(size(newImage, 2) - (size(boxKernel, 2)-1));

                        piece = newImage(x:x+(size(boxKernel, 1)-1), y:y+(size(boxKernel,
2)-1)) .* boxKernel;
                        normal = imProcess.arrayOfAlpha(x,y,z);

                        if (normal >= (-pi/8) && normal <= (pi/8)) || (normal >= (7*pi/8)
&& normal <= (-7*pi/8))   % Checking for horizontal edge

                            if piece(2,2) < piece(2,1) || piece(2,2) < piece(2,3)
                                gN (x, y, z) = 0;
                            else
                                gN (x, y, z) = piece(2,2);
                            end

                        elseif (normal >= (pi/8) && normal <= (3*pi/8)) || (normal >= (-
7*pi/8) && normal <= (-5*pi/8))     % checking for -45 degree edge

                            if piece(2,2) < piece(3,1) || piece(2,2) < piece(1,3)
                                gN (x, y, z) = 0;
                            else
                                gN (x, y, z) = piece(2,2);
                            end

                        elseif (normal >= (3*pi/8) && normal <= (5*pi/8)) || (normal >=
(-5*pi/8) && normal <= (-3*pi/8))   % checking for vertical edge

                            if piece(2,2) < piece(1,2) || piece(2,2) < piece(3,2)
                                gN (x, y, z) = 0;
                            else
                                gN (x, y, z) = piece(2,2);
                            end

                        elseif (normal >= (5*pi/8) && normal <= (7*pi/8)) || (normal >=
(-3*pi/8) && normal <= (-pi/8))   % checking for +45 degree edge

                            if piece(2,2) < piece(1,1) || piece(2,2) < piece(3,3)
                                gN (x, y, z) = 0;
                            else
                                gN (x, y, z) = piece(2,2);
                            end
                        end

                    end
                end
            end

            gNH = zeros(double(size(gN)));
            gNL = zeros(double(size(gN)));

            prompt = {'Enter High threshold:', 'Enter Low threshold'};
            title = 'Hysterisis Thresholding';
            num_lines = [1 50];
            default_ans = {'160', '40'};
            options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
            kernelParameters = inputdlg(prompt, title, num_lines, default_ans, options);
            TH = str2double(kernelParameters{1,:});            TL =
str2double(kernelParameters{2,:});
```

```
            for x = 1:1:size(gN, 1)
                for y = 1:1:size(gN, 2)

                    if gN(x,y) >= TH
                        gNH(x,y) = gN(x,y);
                    elseif gN(x,y) >= TL
                        gNL(x,y) = gN(x,y);
                    end
                end
            end

            gNL = gNL - gNH;

        end

        function [imProcess, avgIntensity] = avgImgIntensity (imProcess, anyArray)

            imProcess = imProcess.sumOfArray(double(anyArray));
            avgIntensity = imProcess.arraySum/(size(anyArray, 1)*size(anyArray, 2));
            disp(sprintf('Average Intensity:\t%f', avgIntensity));

        end

        function imProcess = basicGlobalThresholding (imProcess)

            % The function performs global thresholding operation on the
            % input image.

            [imProcess, avgIntensity] = imProcess.avgImgIntensity(imProcess.img);

            promptDlg = sprintf('Average Image intensity is %f.\n\nSelect an initial
estimate for the global threshold:', avgIntensity);

            prompt = {promptDlg, 'Enter delta threshold:'};
            title = 'Global Thresholding Parameters';
            num_lines = [1 60];
            default_ans = {'128', '10'};
            options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
            thresholdParameters = inputdlg(prompt, title, num_lines, default_ans,
options);
            initialThreshold = str2double(thresholdParameters{1,:});  delThreshold =
str2double(thresholdParameters{2,:});

            delTa = 255;
            newThreshold = 0;
            gH = double(zeros(size(imProcess.img)));
            gL = double(zeros(size(imProcess.img)));

            while delTa > delThreshold

                for z = 1:1:size(imProcess.img, 3)
                    for x = 1:1:size(imProcess.img, 1)
                        for y = 1:1:size(imProcess.img, 2)

                            if imProcess.img (x, y, z) > initialThreshold;

                                gH(x, y, z) = imProcess.img(x, y, z);

                            elseif imProcess.img (x, y, z) <= initialThreshold;

                                gL(x, y, z) = imProcess.img(x, y, z);
```

```
                        end

                    end
                end
            end

            imProcess = imProcess.sumOfArray(gH);
            meanH = imProcess.arraySum/(size(gH, 1)*size(gH, 2));
            imProcess = imProcess.sumOfArray(gL);
            meanL = imProcess.arraySum/(size(gL, 1)*size(gL, 2));

            newThreshold = (meanH + meanL)/2;
            delTa = abs(initialThreshold - newThreshold)
            initialThreshold = newThreshold;
        end

        for z = 1:1:size(imProcess.img, 3)
            for x = 1:1:size(imProcess.img, 1)
                for y = 1:1:size(imProcess.img, 2)

                    if imProcess.img (x, y, z) > newThreshold;

                        imProcess.imgProcessed(x, y, z) = 255;

                    elseif imProcess.img (x, y, z) <= newThreshold;

                        imProcess.imgProcessed(x, y, z) = 0;

                    end

                end
            end
        end

    end

    function [imProcess, classLow, classHigh] = classDivider (imProcess, anyArray)

        % The function divides input array (image) in two classes. The
        % class low is the array of image with intensity values < k.
        % class high is the array with of image with intensity values
        % higher than k.

        [imProcess, output] = imProcess.avgImgIntensity(imProcess.img);

        promptDlg = sprintf('Average Image intensity is %f.\n\nSelect an initial
estimate for the global threshold:', output);
        prompt = {promptDlg};
        title = 'Global Thresholding Parameters';
        num_lines = [1 60];
        default_ans = {'128'};
        options.Resize='on';    options.WindowStyle='normal';
options.Interpreter='tex';
        seperator = str2double(inputdlg(prompt, title, num_lines, default_ans,
options));

        anyArray = double(anyArray);
        classLow = double(zeros(size(anyArray)));
        classHigh = double(zeros(size(anyArray)));

        for z = 1:1:size(anyArray, 3)
            for x = 1:1:size(anyArray, 1)
                for y = 1:1:size(anyArray, 2)
```

```
                    if anyArray(x, y, z) <= seperator

                        classLow(x, y, z) = anyArray(x, y, z);

                    elseif anyArray(x, y, z) > seperator

                        classHigh(x, y, z) = anyArray(x, y, z);

                    end

                end
            end
        end

    end

    function imProcess = optimumGlobalThresholding (imProcess)

        % The function performs optimum global thresholding operation
        % on the imput imageusing Otsu's method.

        %[imProcess, classLow, classHigh] = imProcess.classDivider (imProcess.img);

        imProcess = imProcess.histEstimator(imProcess.img);

        P1 = double(zeros(size(imProcess.pixelVals)));
        P2 = double(zeros(size(imProcess.pixelVals)));
        mK = double(zeros(size(imProcess.pixelVals)));

        P = 0;
        m = 0;

        for n = 1:1:numrows(imProcess.pixelVals)

            P = P + imProcess.norm_hist_data(n,:);
            P1(n,:) = P;

            m = m + double(imProcess.pixelVals(n,:))*imProcess.norm_hist_data(n,:);
            mK(n,:) = m;

        end

        P2 = 1 - P1;

        SigBK = ((double(imProcess.meanVal).*P1 - mK).^2)./(P1.*P2);
        max(SigBK);

        %disp(imProcess.pixelVals)
        %disp(imProcess.hist_data)
        %disp(imProcess.norm_hist_data)

    end

    % =============== End of Image Segmentation I ====================

end     % end of imProcessor class methods...
end     % end of imProcessor class definition...
```

9.4 List of self-coded function in `imProcessor` class object.

```
function imProcess = getImg (imProcess)
function imProcess = writeNsaveImage (imProcess)
function imProcess = selectROIorImg (imProcess)
function [imProcess, outputArray] = convert2gray (imProcess, anyArray)
function imProcess = showOriginal(imProcess)
function imProcess = showProcessed(imProcess)
function imProcess = showVariant(imProcess)
function imProcess = thrshldPixlVal (imProcess)
function imProcess = getRandNum (imProcess)
function imProcess = histEstimator (imProcess, anyArray)
function imProcess = drawHist (imProcess)
function imProcess = myHistEqualization (imProcess)
function imProcess = shearVertical (imProcess)
function imProcess = shearHorizontal (imProcess)
function imProcess = rotateImg (imProcess)
function imProcess = getAuxImg (imProcess)
function imProcess = blendImg (imProcess)
function imProcess = brightnessControl (imProcess)
function imProcess = thresholdingImage (imProcess)
function imProcess = contrastStretching (imProcess)
function imProcess = intensityPass (imProcess)
function imProcess = intensityBoost (imProcess)
function imProcess = bitPlaneSlicer (imProcess)
function imProcess = getNegative(imProcess)
function imProcess = getLogTransform(imProcess)
function imProcess = getPowerTransform(imProcess)
function imProcess = sumOfArray (imProcess, myArray)
function imProcess = getKernelSize (imProcess)
function imProcess = smoothByBox(imProcess)
function imProcess = differenceDetector (imProcess, anyArray)
function imProcess = fuzzyEdgeDetection (imProcess)
function imProcess = sobelEdgeDetector (imProcess)
function imProcess = getGaussKernelSize (imProcess)
function imProcess = smoothByGauss (imProcess)
function imProcess = createLaplacianKernelFilter (imProcess)
function imProcess = sharpenByLaplacian(imProcess)
function imProcess = unsharpMasking(imProcess)
function imProcess = getCutOff (imProcess)
function imProcess = getLowHighPassFilter (imProcess)
function imProcess = fourierFiltering (imProcess)
function imProcess = getBandParameters (imProcess)
function imProcess = getBandPassRejectFilter (imProcess)
function imProcess = fourierBandSelectFiltering (imProcess)
function imProcess = productOfArray (imProcess, myArray)
function imProcess = arithmeticMeanFilter (imProcess)
function imProcess = geometricMeanFilter (imProcess)
function imProcess = harmonicMeanFilter (imProcess)
function imProcess = contraHarmonicMeanFilter (imProcess)
function imProcess = orderStatFiltParamtr (imProcess, array)
function imProcess = arrayTrimmer (imProcess, arrayInput, trimFactor)
function imProcess = orderStatisticsFilter (imProcess)
function imProcess = erosion(imProcess)
function imProcess = dilation (imProcess)
function imProcess = opening (imProcess)
```

```
function imProcess = closing (imProcess)
function imProcess = boundaryExtraction (imProcess)
function imProcess = morphologicalGradient (imProcess)
function imProcess = topHatTransform (imProcess)
function imProcess = bottomHatTransform (imProcess)
function imProcess = contrastEnhance (imProcess)
function imProcess = morphSmoothing (imProcess)
function imProcess = lineDetection (imProcess)
function imProcess = laplacianOperator (imProcess)
function imProcess = gradientOperator (imProcess)
function imProcess = kirschCompassMask (imProcess)
function imProcess = laplacianOfGaussian (imProcess)
function imProcess = marrHildrethEdgeDetection (imProcess)
function imProcess = cannyEdgeDetector (imProcess)
function [imProcess, avgIntensity] = avgImgIntensity (imProcess, anyArray)
function imProcess = basicGlobalThresholding (imProcess)
function [imProcess, classLow, classHigh] = classDivider (imProcess, anyArray)
function imProcess = optimumGlobalThresholding (imProcess)
```

9.5 GUI Features

1. Open and Save Menu

The menu opens and save the image to be processed and processed Image Respectively.

2. Intensity Transformation panel

The panel consists of buttons like contrast stretch, blend image, negative, intensity boost, thresholding, intensity pass, power transform, image blending, Bit-plane slicing, Log transform. For log transform, contrast and power transform, we can vary the slider and see effects for varying constants.

3. Geometric Transformation panel

The panel includes sliders for shears and rotations.

4. Spatial Filtering Panel

Spatial filtering includes Smoothing, sharpening, Histogram estimation, Box kernel and gaussian kernels options, Edge detection options.

5. F-Domain Buttons

F-domain panel has two buttons, namely frequency domain and band filtering. Frequency domain button includes all the options.

6. Noise Filtering

Noise Filtering is having options like Mean filters and order statistic filters

7. Morphological Image Processing

This menu bar contains functions for morphological operations on the image namely Dilation, Erosion, Opening, Closing, Top-Hat, Bottom-Hat, Thinning, Thickening, Pruning, Convex hull, Skeletons.

8. Image Segmentation

This Menu button has functions such as Line Detection, Laplacian, Gradient Operator, Marr-Hildreth Algorithm, Canny Edge Detector, Basic Global Thresholding, OTSU's Thresholding, Thresholding using moving averages,

9. Edit tools

In edit tools, you can select the portion of the image you want or you can select the complete image you want to re-process.