Rohit Raibagkar
gd4139

Intro. to Digital Image Processing

ECE 5690
Fall 2017

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: 12/13/2017

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 affine matrix transformation to shear or rotate image, intensity and power transformation to enhance image quality. Also, to remove the noise, I have experimented with box kernels and gaussian kernels. To sharpen the image, Laplacian and unmask sharpening techniques are discussed. Further, Detailed Fourier domain analysis is carried out to study the effects of frequency changes and frequency bands on image processing. To study the effects of higher frequencies, high boost and high-frequency-emphasis techniques are discussed in 9th chapter. Noise is again big issue as mentioned. I experimented with mean and order statistic 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 second chapter presents the problem statement assigned. Third chapter focuses on methods in geometric transformations and its results on various images. The 4th chapter discusses intensity transformations in detail with results. Next, 5th chapter discusses advanced intensity transformations. Next, 6th chapter introduces and discusses spatial transformations in detail. 7th chapter deals with advanced spatial domain operations such as Fuzzy sets and Sobel operators to detect edge of an image. 8th chapter delves more in details of frequency domain 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 image noise reduction and image enhancement 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.

- Compute and display Histogram of Image.

Histogram is important tool to determine probability distribution of pixels and estimate the noise level present in the image. I have successfully coded function to compute and display histogram of an image and managed to get histogram equalization of an image, to enhance the image details.

- Geometric transformations on image (zooming, shrinking, rotation, shear).

Geometric transformations to modify and highlight details of an image. I have coded horizontal shear, vertical shear and rotations of image using affine matrix transformation.

- Adjust the image contrast and brightness through intensity transformations.

I carried out intensity transformations such as negatives, logarithms, power-law, piecewise linear transformations, histogram techniques.

- Image filtering in both spatial and frequency domain and results from different filters.
- Noise reduction using Mean filters (arithmetic, harmonic, contrharmonic) and orderstatistic filters.
- Processing the given images with specific issues.

3. Geometric Transformations

In many imaging systems, detected images are subject to geometric distortion introduced by perspective irregularities wherein the position of the camera(s) with respect to the scene alters the apparent dimensions of the scene geometry. Applying an affine transformation to a uniformly distorted image can correct for a range of perspective distortions by transforming the measurements from the ideal coordinates to those used. (For example, this is useful in satellite imaging where geometrically correct ground maps are desired).

The general affine transformation is commonly written in homogeneous coordinates:

$$\begin{vmatrix} x_2 \\ y_2 \end{vmatrix} = A \times \begin{vmatrix} x_1 \\ y_1 \end{vmatrix} + B$$

By defining only the *B* matrix, this transformation can carry out pure translation:

$$A = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}, B = \begin{vmatrix} b_1 \\ b_2 \end{vmatrix}$$

Pure rotation uses the *A* matrix and is defined as (for positive angles being clockwise rotations):

$$A = \begin{vmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{vmatrix}, B = \begin{vmatrix} 0 \\ 0 \end{vmatrix}$$

Similarly, pure scaling is:

$$A = \begin{vmatrix} a_{11} & 0 \\ 0 & a_{22} \end{vmatrix}, B = \begin{vmatrix} 0 \\ 0 \end{vmatrix}$$

The transformations are applied to the images using affine matrices. The affine matrices modify (x, y) coordinates of the pixel and sets the pixel to the new value, thus affecting the complete image.

In MATLAB, Horizontal shear, Vertical shear, and Rotation is achieved by affine matrix transformation. The command I used was:

For horizontal shear:

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

For vertical shear:

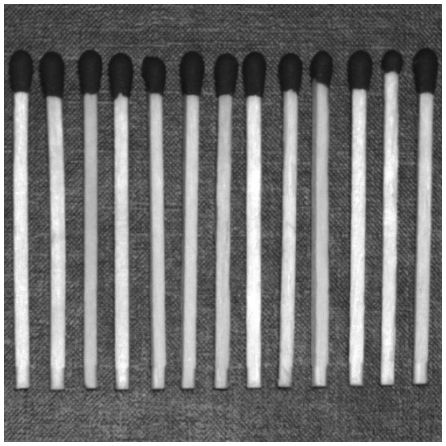
```
sampleArray(uint64(x + imProcess.shearConstant*y), y) = imProcess.img(x,y);
```

For Rotation:

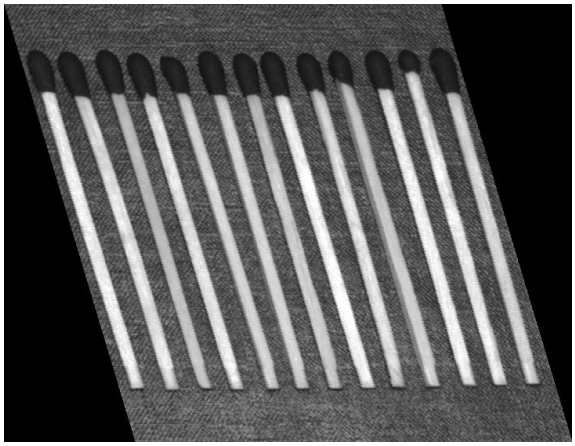
```
xPrime = uint64((x*cos(imProcess.theta)) - (y*sin(imProcess.theta)) +
size(imProcess.img, 1)*sqrt(2));
yPrime = uint64((x*sin(imProcess.theta)) + (y*cos(imProcess.theta)) +
size(imProcess.img, 1)*sqrt(2));
sampleArray(xPrime, yPrime) = imProcess.img(x,y);
```

Results:

Horizontal Shear

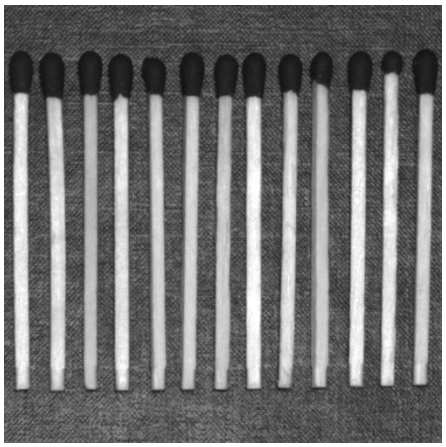


Original Image

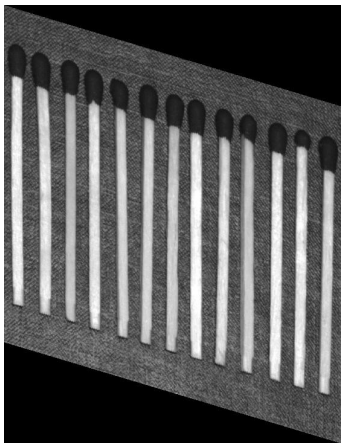


Horizontal shear with shear factor 0.3

Vertical Shear

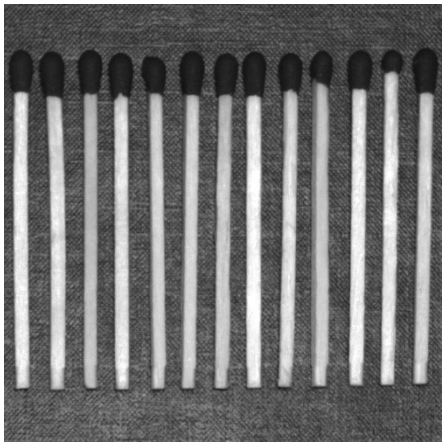


Original Image

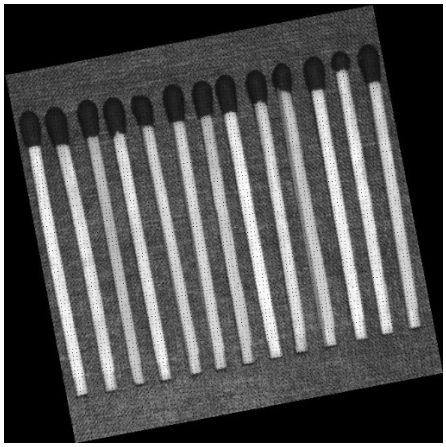


Vertical Shear with shear factor 0.3

Rotation



Original Image



Rotation by 10.8^0

4. Intensity Transformations

Image arithmetic applies one of the standard arithmetic operations or a logical operator to two or more images. The operators are applied in a pixel-by-pixel fashion which means that the value of a pixel in the output image depends only on the values of the corresponding pixels in the input images. Hence, the images normally should be of the same size. One of the input images may be a constant value, for example when adding a constant offset to an image.

Although image arithmetic is the simplest form of image processing, there is a wide range of applications. A main advantage of arithmetic operators is that the process is very simple and therefore fast. Single-point processing is a simple method of image enhancement. This technique determines a pixel value in the enhanced image dependent only on the value of the corresponding pixel in the input image. The process can be described with the *mapping function.*

The intensity transformations I carried out on images are Image Brightness, Contrast Stretch, Image Negatives, Image Thresholding, Intensity Boost, Intensity Pass, Image Blending.

4.1 Image Brightness

I used pixel addition method to enhance image brightness. In its most straightforward implementation, this operator takes as input two identically sized images and produces as output a third image of the same size as the first two, in which each pixel value is the sum of the values of the corresponding pixel from each of the two input images. More sophisticated versions allow more than two images to be combined with a single operation. A common variant of the operator simply allows a specified constant to be added to every pixel. It is given by equation:

$$Q(i,j) = P_1(i,j) + C$$

I used addition method, which was mapping the complete addition to `uint8` (unsigned 8 bit integer). The code in nested `for` loops is as given:

```
function imProcess = brightnessControl (imProcess)

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

end
```

Original Image

Enhanced brightness to intensity 80

4.2 Contrast Stretch

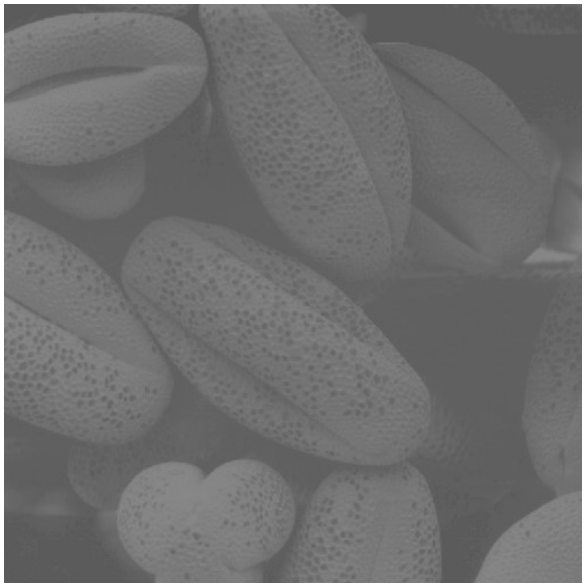
Contrast stretching (often called normalization) is a simple image enhancement technique that attempts to improve the contrast in an image by stretching the range of intensity values it contains to span a desired range of values, *e.g.* the the full range of pixel values that the image type concerned allows. It differs from the more sophisticated histogram equalization in that it can only apply a *linear* scaling function to the image pixel values. As a result, the enhancement is less harsh. (Most implementations accept a graylevel image as input and produce another graylevel image as output.)

The simplest sort of normalization then scans the image to find the lowest and highest pixel values currently present in the image. Call these *c* and *d*. Then each pixel *P* is scaled using the following function:

$$P_{out} = (P_{in} - c)\left(\frac{b-a}{d-c}\right) + a$$

The formula to implement the function in MATLAB is:

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



Original Image of low contrast pollen

Enhanced contrast with contrast stretching

4.3 Image Negatives

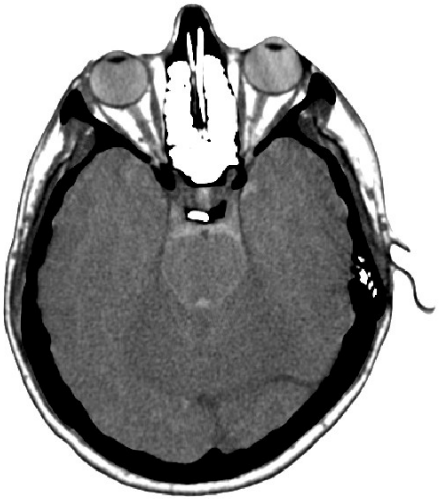
Logical NOT or *invert* is an operator which takes a binary or graylevel image as input and produces its photographic negative, *i.e.* dark areas in the input image become light and light areas become dark. The resulting value for each pixel is the input value subtracted from *255:*

$$Q(i,j) = 255 - P(i,j)$$

Image negative formula coded by me is:

```
imProcess.imgProcessed(x,y,z) =  uint16(((2^imProcess.imgInfo.BitDepth)-1) -
                                 imProcess.img(x,y,z));
```

Original CT Image



Negative of CT Image

4.4 Image Thresholding

The input to a thresholding operation is typically a grayscale or color image. In simple implementations, the segmentation is determined by a single parameter known as the *intensity threshold*. In a single pass, each pixel in the image is compared with this threshold. If the pixel's intensity is higher than the threshold, the pixel is set to, say, white in the output. If it is less than the threshold, it is set to black.

The MATLAB code for thresholding is:

```
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
```

Original Image

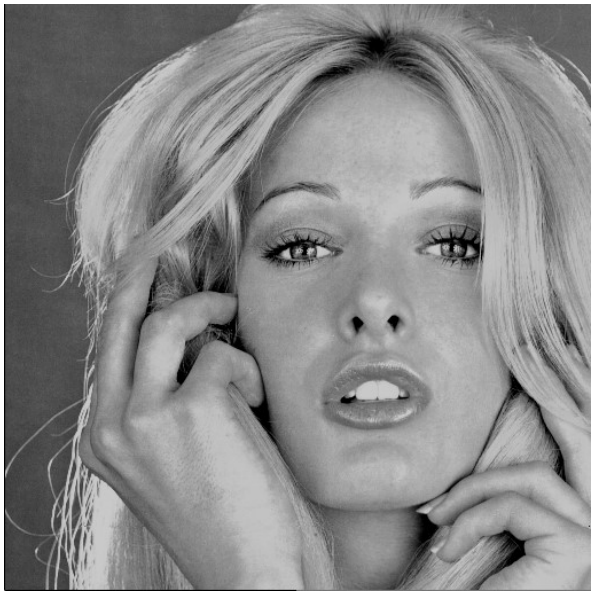
Thresholded image with threshold = 100

4.5 Intensity Boost

In the intensity boost, the intensity values between given band of intensities are boosted to the required value while others are kept unchanged.

The following snippet explains the intensity boost operation:

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

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



Original image

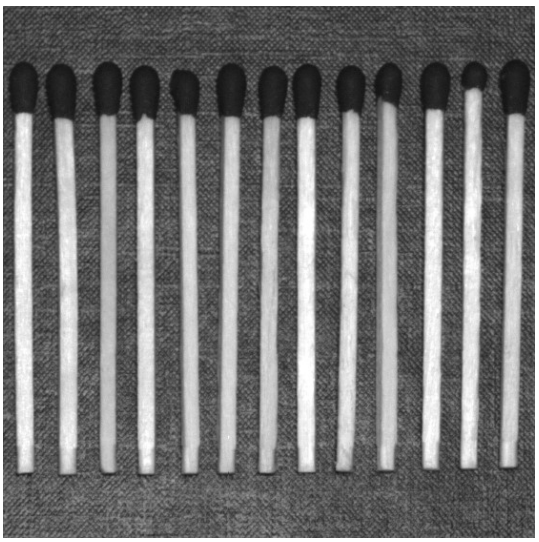
Image boosted with intensity values between 150 and 180 to intensity 240

4.6 Intensity Pass

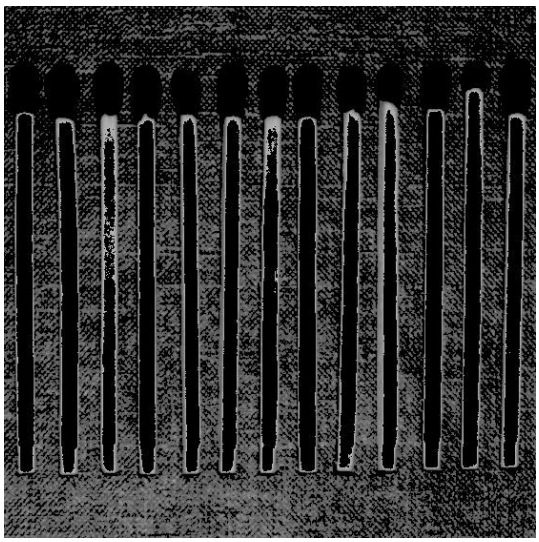
In the intensity pass, only the values between given band are passed and other values are set to zero.

The code for intensity pass:

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

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



Original Image



Intensity clipped below 80 and above 100

4.7 Image Blending

This operator forms a blend of two input images of the same size. the value of each pixel in the output image is a linear combination of the corresponding pixel values in the input images. The resulting image is calculated using the formula:

$$Q(i,j) = X \times P_1(i,j) + (1 - X) \times P_2(i,j)$$

The MATLAB Code for blend calculation is:

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

Two images of same size

Blended with blend factor 0.6

5. Advanced Intensity Transformations

5.1 Power Transform

The exponential operator is a point process where the mapping function is an exponential curve. This means that each pixel intensity value in the output image is equal to a basis value raised to the value of the corresponding pixel value in the input image. Which basis number is used depends on the desired degree of compression of the dynamic range. To enhance the visibility of a normal photograph, values just above *1* are suitable. For display, the image must be scaled such that the maximum value becomes *255* (assuming an 8-bit display). The resulting image is given by

$$Q(i,j) = c\, b^{P(i,j)}$$

The code snippet for power transform:

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

Original Image

Transformed Image with K = 1.25 & Gamma = 1.05

5.2 Log Transform

The logarithmic operator is a simple point processor where the *mapping function* is a logarithmic curve. Each pixel value is replaced with its logarithm. Most implementations take either the *natural logarithm* or the *base 10 logarithm*. However, the basis does not influence the shape of the logarithmic curve, only the scale of the output values which are scaled for display on an 8-bit system. Hence, the basis does not influence the degree of compression of the dynamic range. The logarithmic mapping function is given by:

$$Q(i,j) = c\, \log(1 + |P(i,j)|)$$

Code snippet for log transform:

```
imProcess.imgProcessed(x,y,z) = uint8(log(1 + double(imProcess.img(x,y,z))));
```

Original Image

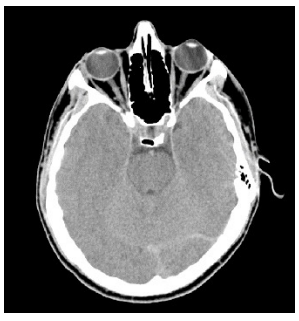
Log transform with constant K = 55

5.3 Bit-Plane Slicing

Bit-plane slicing give T(r) = 1 if the given intensity values are between n^{th} bit and $n\text{-}1^{th}$ bit. In other words, it gives the bit level intensity contribution in image. If we slice image to 8^{th} bit, then values between 2^8 -1 = 255 and 2^7 = 128 are mapped to one and other are mapped to zero. The same process is applied to other bit level intensities.

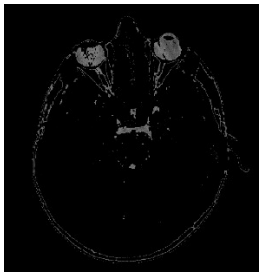
Code snippet for Bit-Plane slicing:

```
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
```



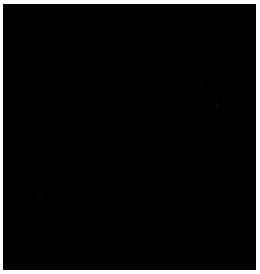
Original image of CT

8^{th} bit slice

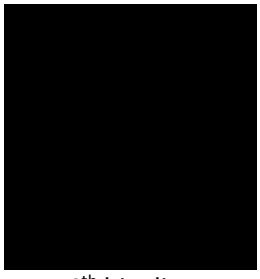


7^{th} bit slice

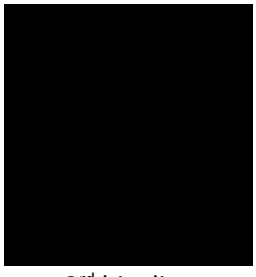
6^{th} bit slice



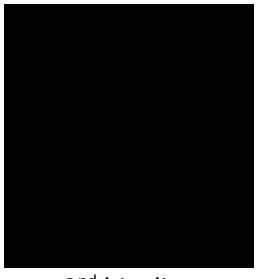
5^{th} bit slice



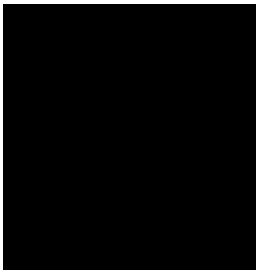
4^{th} bit slice



3^{rd} bit slice



2^{nd} bit slice



1^{st} bit slice

6. Spatial Filtering

6.1 Histogram

Histogram modeling techniques (*e.g.* histogram equalization) provide a sophisticated method for modifying the dynamic range and contrast of an image by altering that image such that its intensity histogram has a desired shape. Unlike contrast stretching, histogram modeling operators may employ *non-linear* and *non-monotonic* transfer functions to map between pixel intensity values in the input and output images. Histogram equalization employs a monotonic, non-linear mapping which re-assigns the intensity values of pixels in the input image such that the output image contains a uniform distribution of intensities (*i.e.* a flat histogram). This technique is used in image comparison processes (because it is effective in detail enhancement) and in the correction of non-linear effects introduced by, say, a digitizer or display system.

Code Snippet:

This function takes input image and estimates the values of each pixel intensities.

```
if pixelvals(n,:) == imProcess.img(x,y);
      hist_data(n,:) = hist_data(n,:) + 1;
      imProcess.norm_hist_data(n,:) =
       (hist_data(n,:))/(numrows(imProcess.img)*numcols(imProcess.img));
end
```

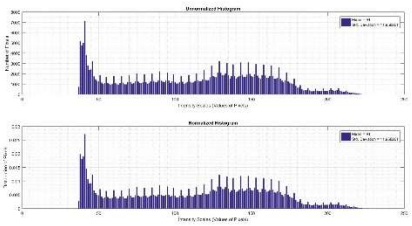
This function takes histogram array as input and equalizes pixel values as histogram equalization:

```
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
```

This function replaces the original image pixel value with histogram equalized pixel value:

```
if imProcess.img(x,y) == pixelvals(n,:)
      imProcess.imgProcessed(x,y) = newPixelVals(n,:);
end
```

Pirate



Its Histogram

Histogram equalized Image

6.2 Smoothing

6.2.1 Box Kernels

Box kernels used in reducing the amount of intensity variation between one pixel and the next. It is often used to reduce noise in images. The idea of mean filtering is simply to replace each pixel value in an image with the mean (average) value of its neighbors, including itself. This has the effect of eliminating pixel values which are unrepresentative of their surroundings. Mean filtering is usually thought of as a convolution filter. Like other convolutions it is based around a kernel, which represents the shape and size of the neighborhood to be sampled when calculating the mean.

5 X 5 Box Kernels

0.0400	0.0400	0.0400	0.0400	0.0400
0.0400	0.0400	0.0400	0.0400	0.0400
0.0400	0.0400	0.0400	0.0400	0.0400
0.0400	0.0400	0.0400	0.0400	0.0400
0.0400	0.0400	0.0400	0.0400	0.0400

6.2.2 Gaussian Kernels

The Gaussian smoothing operator is a 2-D convolution operator that is used to blur images and remove detail and noise. In this sense it is similar to the mean filter, but it uses a different kernel that represents the shape of a Gaussian shape.

7 X 7 Gaussian Kernel with k = 2 & sigma = 2:

0.2108	0.3938	0.5730	0.6493	0.5730	0.3938	0.2108
0.3938	0.7358	1.0705	1.2131	1.0705	0.7358	0.3938
0.5730	1.0705	1.5576	1.7650	1.5576	1.0705	0.5730
0.6493	1.2131	1.7650	2.0000	1.7650	1.2131	0.6493
0.5730	1.0705	1.5576	1.7650	1.5576	1.0705	0.5730
0.3938	0.7358	1.0705	1.2131	1.0705	0.7358	0.3938
0.2108	0.3938	0.5730	0.6493	0.5730	0.3938	0.2108

In the program, the image is padded by suitable zeros and then every pixel is modified using kernel at its center.

Box Kernel:

```
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) = uint8(imProcess.arraySum);
```

Gaussian Kernel:

```
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));
```

Application:

```
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) = uint8(imProcess.arraySum);
```

Original

5 X 5 Box Kernel

9 X 9 Box Kernel

Original

5 X 5 Gaussian with k = 2 & Sigma = 2

7 X 7 Gaussian with k = 2 & Sigma = 2

6.3 Sharpening

6.3.1 Laplacian

The Laplacian is measure of the 2nd spatial derivative of an image. The Laplacian of an image highlights regions of rapid intensity change and is therefore often used for edge detection (see zero crossing edge detectors). The Laplacian is often applied to an image that has first been smoothed with something approximating a Gaussian smoothing filter to reduce its sensitivity to noise, and hence the two variants will be described together here. The operator normally takes a single gray level image as input and produces another gray level image as output.

The two types of 5 X 5 Laplacian Kernels:

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

Another two types of 5 X 5 Laplacian Kernels:

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

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

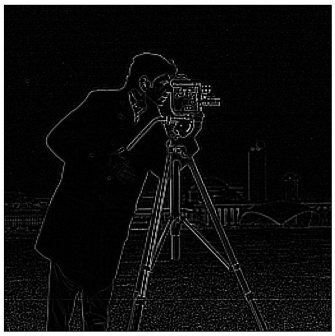
6.3.2 Unsharp Masking

The unsharp mask is a simple sharpening operator which derives its name from the fact that it enhances edges (and other high frequency components in an image) via a procedure which subtracts an unsharp, or smoothed, version of an image from the original image.

Original Image

Image sharpened by 3 X 3 Laplacian

Image sharpened by unsharp masking



Laplacian of Image

Mask of Image

7. Advanced Spatial Filtering

7.1 Fuzzy Edge Detection

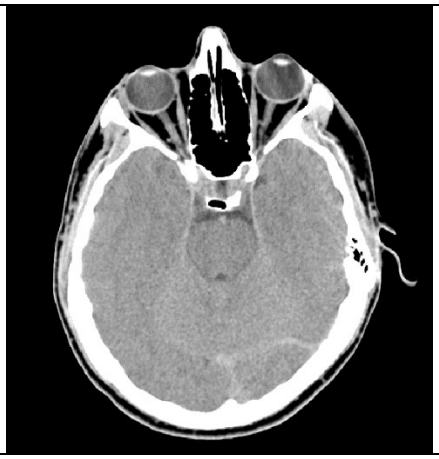
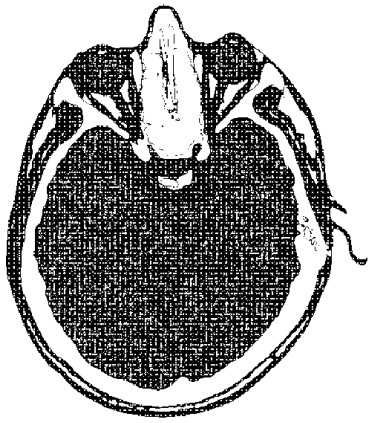
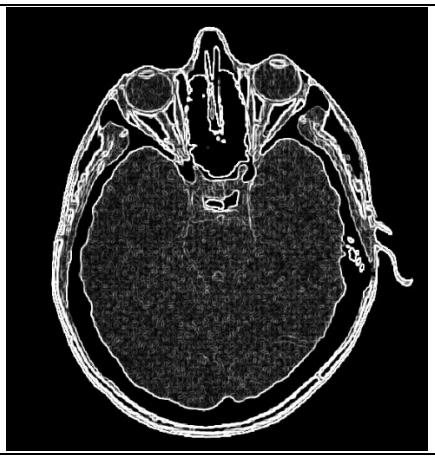
```
d_1    d_2    d_3                'z_1'    'z_2'    'z_3'
d_4     0     d_6                'z_4'    'z_5'    'z_6'
d_7    d_8    d_9                'z_7'    'z_8'    'z_9'
```

Fuzzy detection works on the Distance estimation principle. If neighbourhood distance is zero, then the pixel is belonging to uniform intensity area.

```
if    d_2 == 0 && d_6 == 0
      z_5 = white;
elseif d_6 == 0 && d_8 == 0
      z_5 = white;
elseif d_8 == 0 && d_4 == 0
      z_5 = white;
elseif d_4 == 0 && d_2 == 0
      z_5 = white
else
      z_5 = black;
end
```

7.2 Sobel Edge Detection

The Sobel operator performs a 2-D spatial gradient measurement on an image and so emphasizes regions of high spatial frequency that correspond to edges. Typically, it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image. These kernels are designed to respond maximally to edges running vertically and horizontally relative to the pixel grid, one kernel for each of the two perpendicular orientations. The kernels can be applied separately to the input image, to produce separate measurements of the gradient component in each orientation (call these *Gx* and *Gy*). These can then be combined to find the absolute magnitude of the gradient at each point and the orientation of that gradient.

		
Original	Fuzzy Edge Detection	Sobel Edge Detection

8. Frequency Domain Analysis

Frequency domain analysis gives image enhancement and other image operations in the frequency domain. FFT of image and filter gives effect by just multiplication in frequency domain, thus reducing time and computation effort in convolution and correlation in time domain. The filters in Frequency Domain are given by:

	Low Pass	High Pass
Ideal	H(u,v) = 1 if D(u,v) <= D_0 H(u,v) = 0 if D(u,v) > D_0	1-H(u,v)
Gaussian	$H(u,v) = e^{-D^2(u,v)/2\sigma^2}$	$H(u,v) = 1 - e^{-D^2(u,v)/2\sigma^2}$
Buterworth	$H(u,v) = \frac{1}{1 + [D(u,v) / D_o]} 2n$	$H(u,v) = 1 - \frac{1}{1 + [D(u,v) / D_o]} 2n$

8.1 Smoothing in Frequency Domain

Original

Ideal Filter, 0.25 Cut-Off

Gaussian, 0.25 Cut-Off

Butterworth, 0.25 Cut-Off

Ideal Filter, 0.5 Cut-Off

Gaussian, 0.5 Cut-Off

Butterworth, 0.5 Cut-Off

8.2 Sharpening in Frequency Domain

Ideal Filter, 0.25 Cut-Off

Gaussian, 0.25 Cut-Off

Butterworth, 0.25 Cut-Off, 3rd order

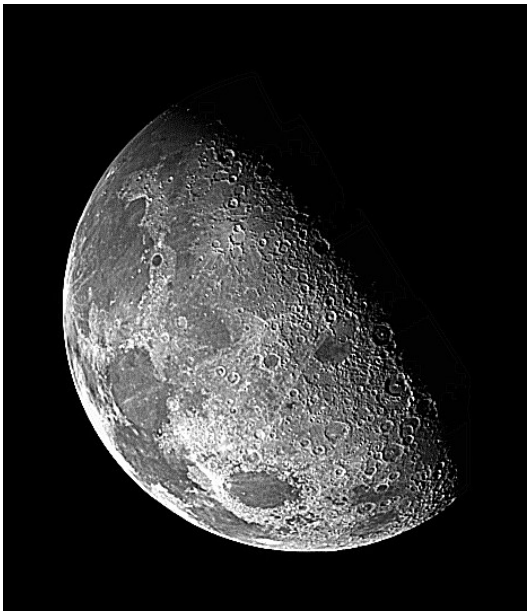
Original	Ideal Filter, 0.5 Cut-Off	Gaussian, 0.5 Cut-Off	Butterworth, 0.5 Cut-Off, 3rd order

8.3 Laplacian in Frequency Domain

Laplacian in frequency domain is given by $H(u,v) = -4\pi^2 D^2(u,v)$

When applied to image, $\nabla^2 f(x,y) = f^{-1}[H(u,v)F(u,v)]$

Original Image



Laplacian

8.4 Unsharp Masking in Frequency Domain

Mask is given by, $g_{mask}(x,y) = f(x,y) - f_{LP}(x,y)$

Which gives $g(x,y) = f(x,y) + kg_{mask}(x,y)$ with k = 1.

Original Image

Unsharp masking with butterworth filter, 0.15 cut-off and order = 2

8.5 High Boost Filtering

When k in unsharp masking is greater than 1, then it becomes high boost filtering.

Original Image

Highboost filtering with butterworth filter, cut-off = 0.25, order = 2, k = 1.25

8.6 High-Frequency-Emphasis Filtering

High frequency emphasis filtering is given by equation

$g(x,y) = F^{-1}\{[1 + kH_{HP}(u,v)]\ F(u,v)\}$ With results below:

Original Image

High-frequency-emphasis filtering with butterworth, cut-off: 0.3, order:2, k: 4

8.7 More General High-Frequency-Emphasis Filtering

$g(x,y) = F^{-1}\{[k_1 + k_2 H_{HP}(u,v)]\, F(u,v)\}$

Original Image

More general High-frequency-emphasis filtering with gaussian, cut-off: 0.2, k1: 1.5, K2: 2

9. Frequency Domain Analysis (Cont.)

7.1 Band Filtering: Band Pass Filtering

Band pass filtering allows only certain band of frequencies around center C_0 and width of band W_0 and blocks all the other frequencies above and below threshold. This filter in useful in unwanted signal rejection in images.

Original Image

Band pass filtering with c0 = 0.25 and W = 0.5

7.2 Band Filtering: Band Reject Filtering

Band reject filter rejects certain band of frequencies within the threshold and passes all the frequencies. Band reject filter is given by: Band reject = 1 – Band Pass.

Original Image

Band reject filtering with c0 = 0.25 and W = 0.5

10. Noise Reduction

10.1 Spatial Filtering

10.1.1 Arithmetic Mean Filter

Arithmetic mean filter given by code:

```
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) = uint8(imProcess.arraySum);
```

10.1.2 Geometric Mean Filter

Geometric mean Filter is given by the code:

```
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));
```

10.1.3 Harmonic Mean Filter

Harmonic Mean Filter is given by:

```
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(myProduct/imProcess.arraySum);
```

10.1.4 ContraHarmonic Mean Filter

Contraharmonic mean filter is given by:

```
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) = uint8(numerator/denominator);
```

10.2 Order Statistics Filtering

10.2.1 Median Filter is given by formula:

```
imProcess.medianIntensity = median(imProcess.sortedArray);
```

10.2.2 Max Filter is given by formula:

```
imProcess.maxIntensity = max(imProcess.sortedArray);
```

10.2.3 Min Filter is given by formula:

```
imProcess.minIntensity = min(imProcess.sortedArray);
```

10.2.4 Midpoint Filter is given by formula:

```
imProcess.midIntensity = (imProcess.minIntensity +
imProcess.maxIntensity)/2;
```

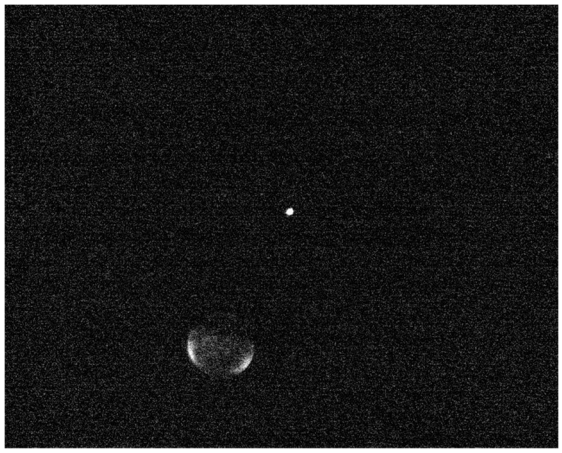
10.2.5 Alpha-Trimmed Mean Filter is given by formula:

Alpha-Trimmed mean filter is given by the formula:

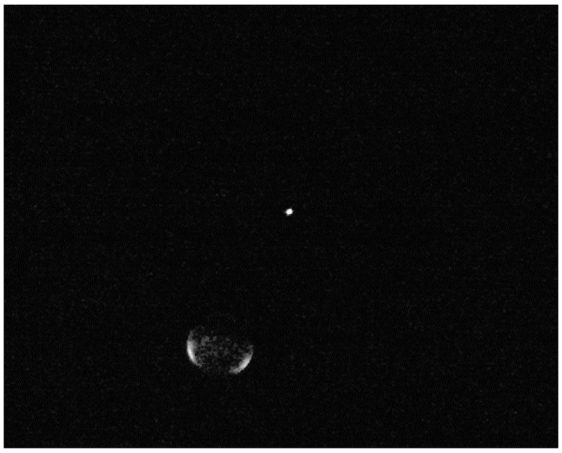
```
for iteration = 1:1:trimFactor/2
     arrayInput(:,iteration) = 0;
     arrayInput(:, size(arrayInput,2) -(iteration) +1) = 0;

end

imProcess.alphaTrimmedArray = arrayInput;
```

11. Problem Solution

Problem Image 1

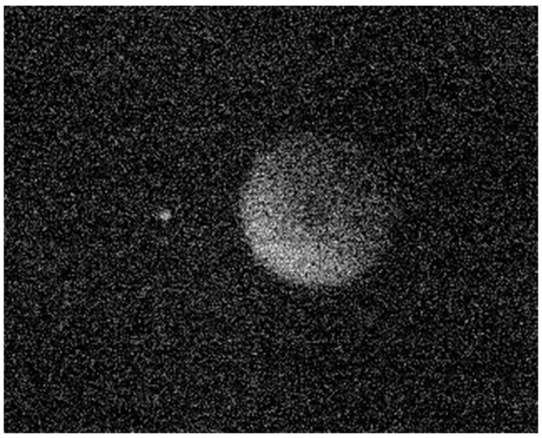


Original Image

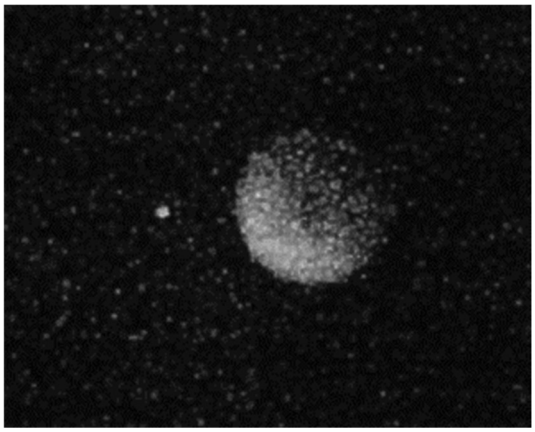


Reduced Salt noise by harmonic mean filter

Problem Image 2

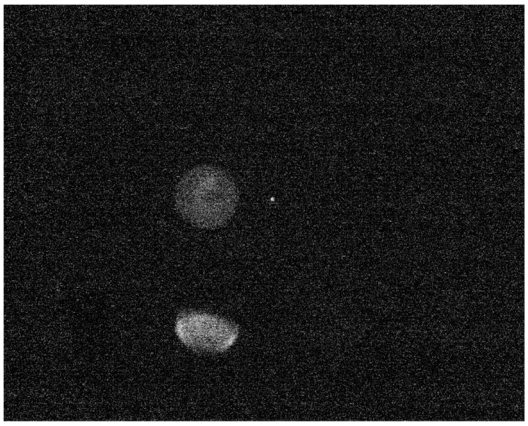


Original Image

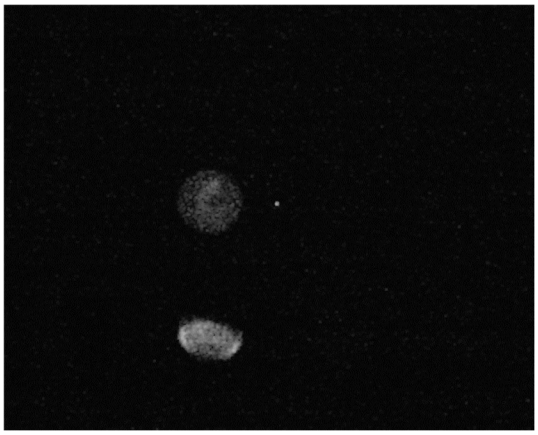


Reduced Salt and Pepper noise by contraharmonic mean filter of size 3 and order -1.25

Problem Image 3

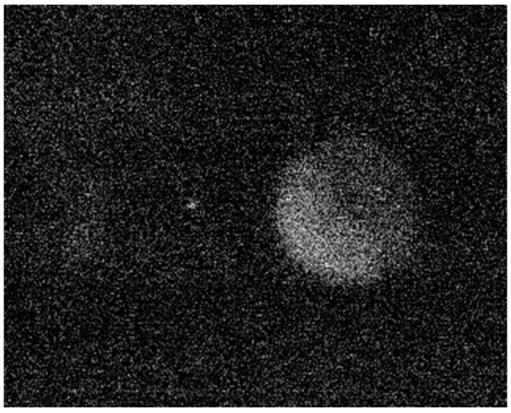


Original Image

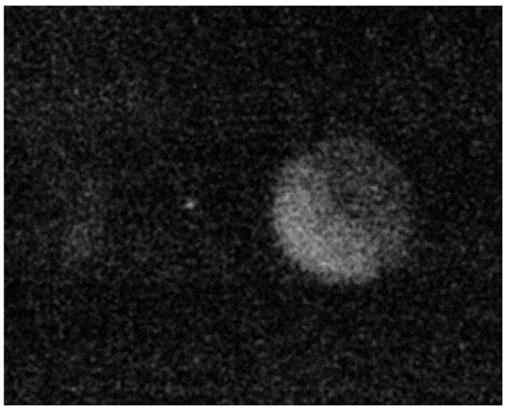


Reduced Salt and Pepper noise by contraHarmonic mean filter of size 3 and order q= -1.25, q= 1.25

Problem Image 4

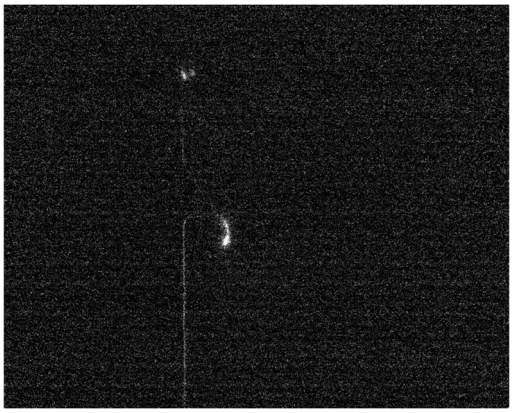


Original Image

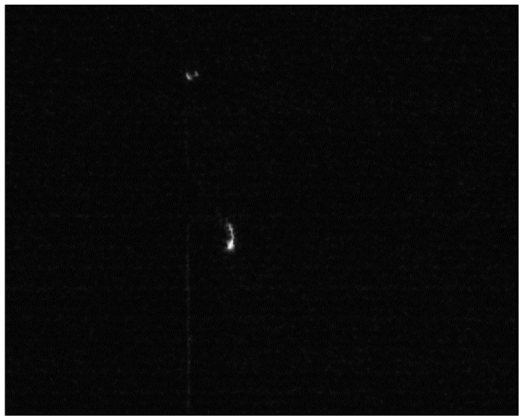


Reduced Salt and Pepper noise Alpha-trimmed mean filter with size = 3 and d = 2

Problem Image 5



Original Image



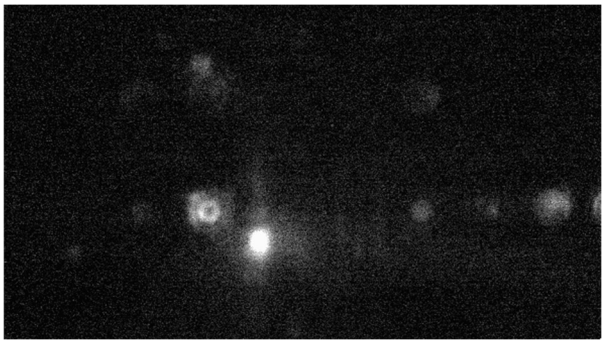
Reduced Salt noise by Harmonic mean filter, size = 3
Contraharmonic mean filter, size =3 & q = -1.25

Problem Image 6

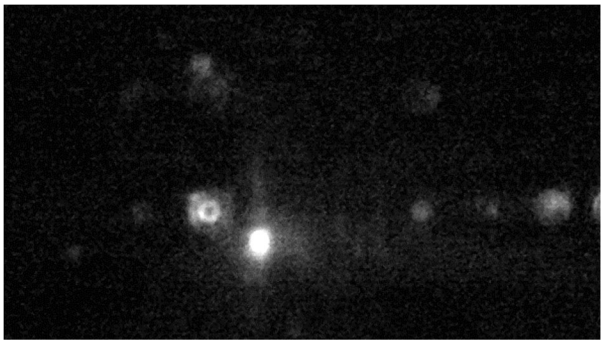
Original Image

Reduced Salt noise by Median filter, size = 3

Problem Image 7

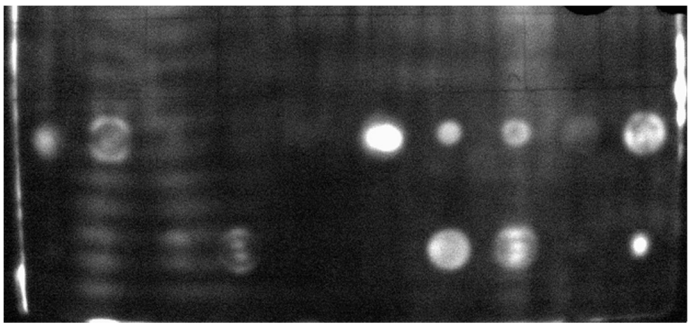


Original Image

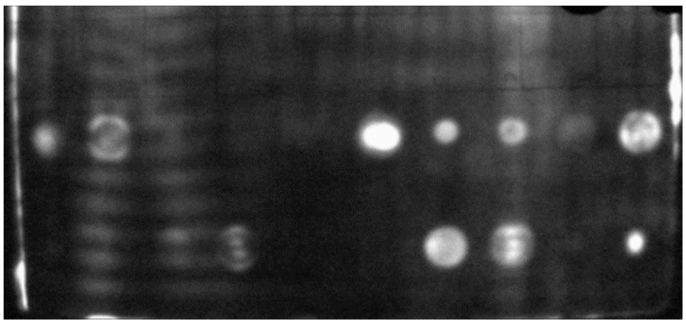


Median filter with size 3, power transform with constant = 0.75 and gamma = 1.075

Problem Image 8

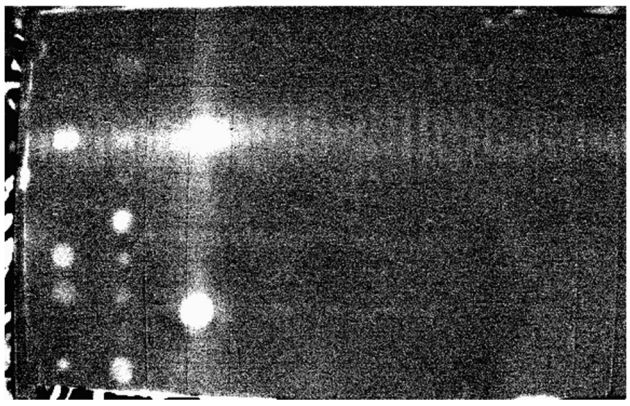


Original Image

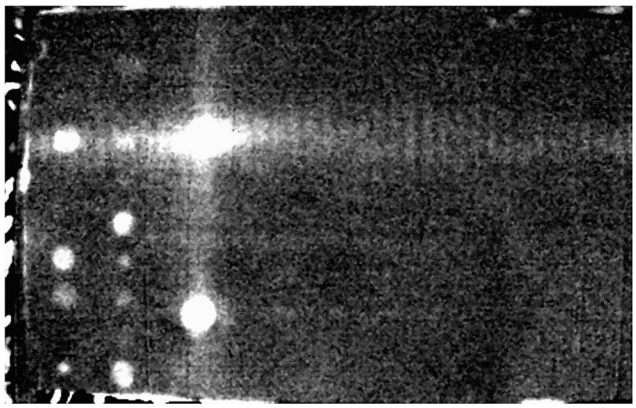


Median filter with size 3, power transform with constant = 0.75 and gamma = 1.075

Problem Image 9



Original Image

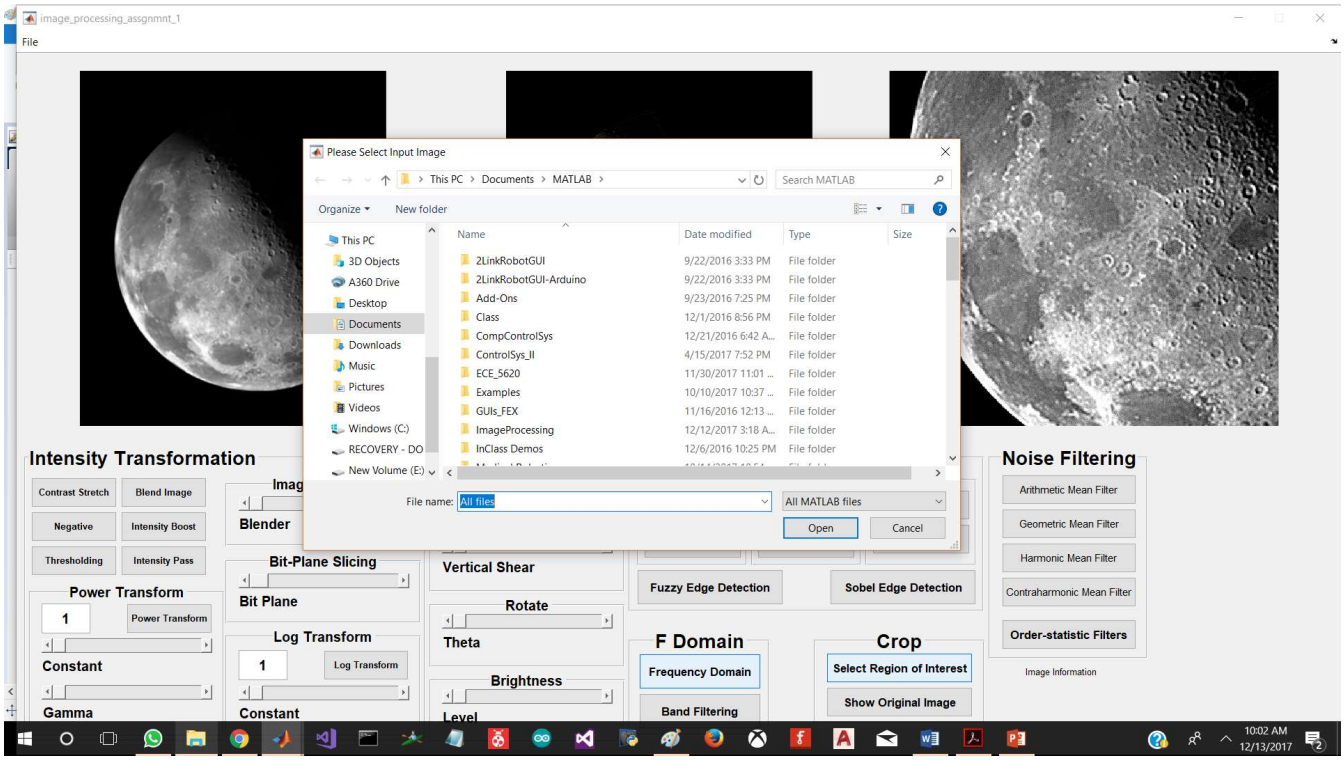


Median filter with size 3, power transform with constant = 0.75 and gamma = 1.075

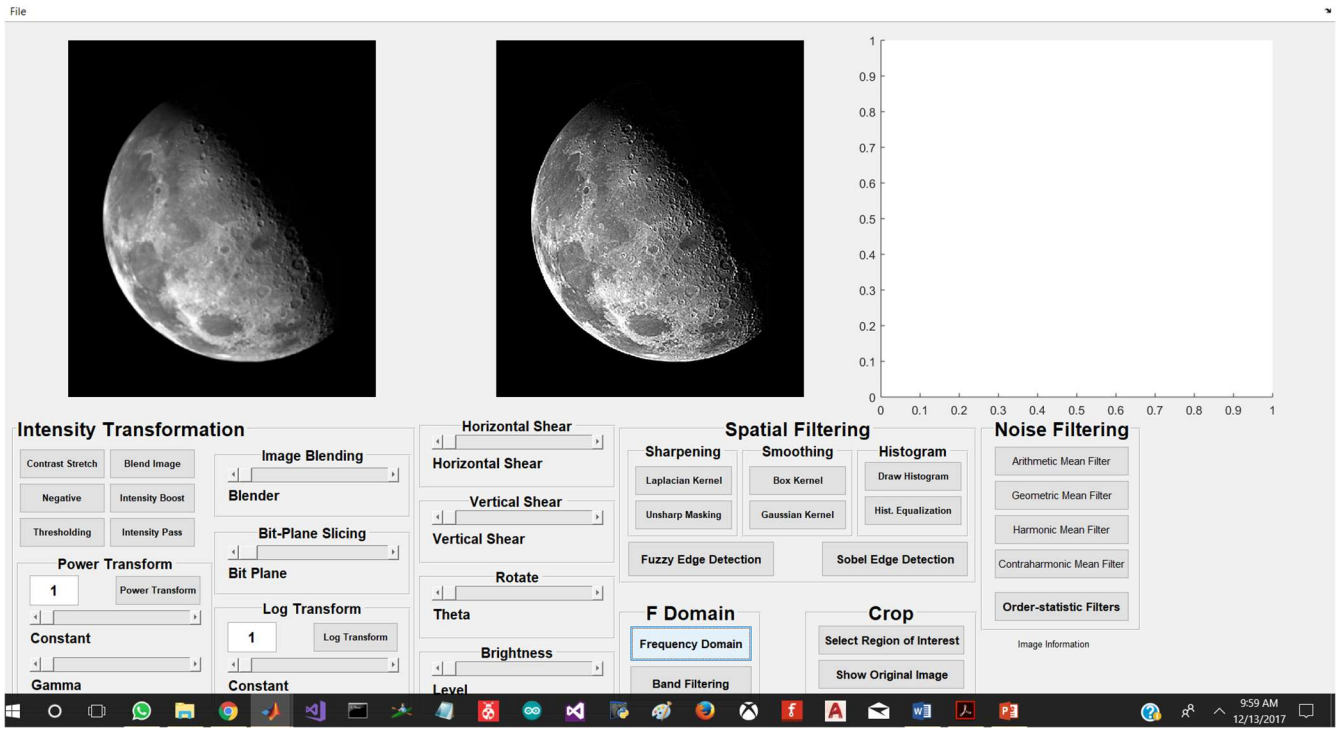
12. Appendix

A. MATLAB GUI.

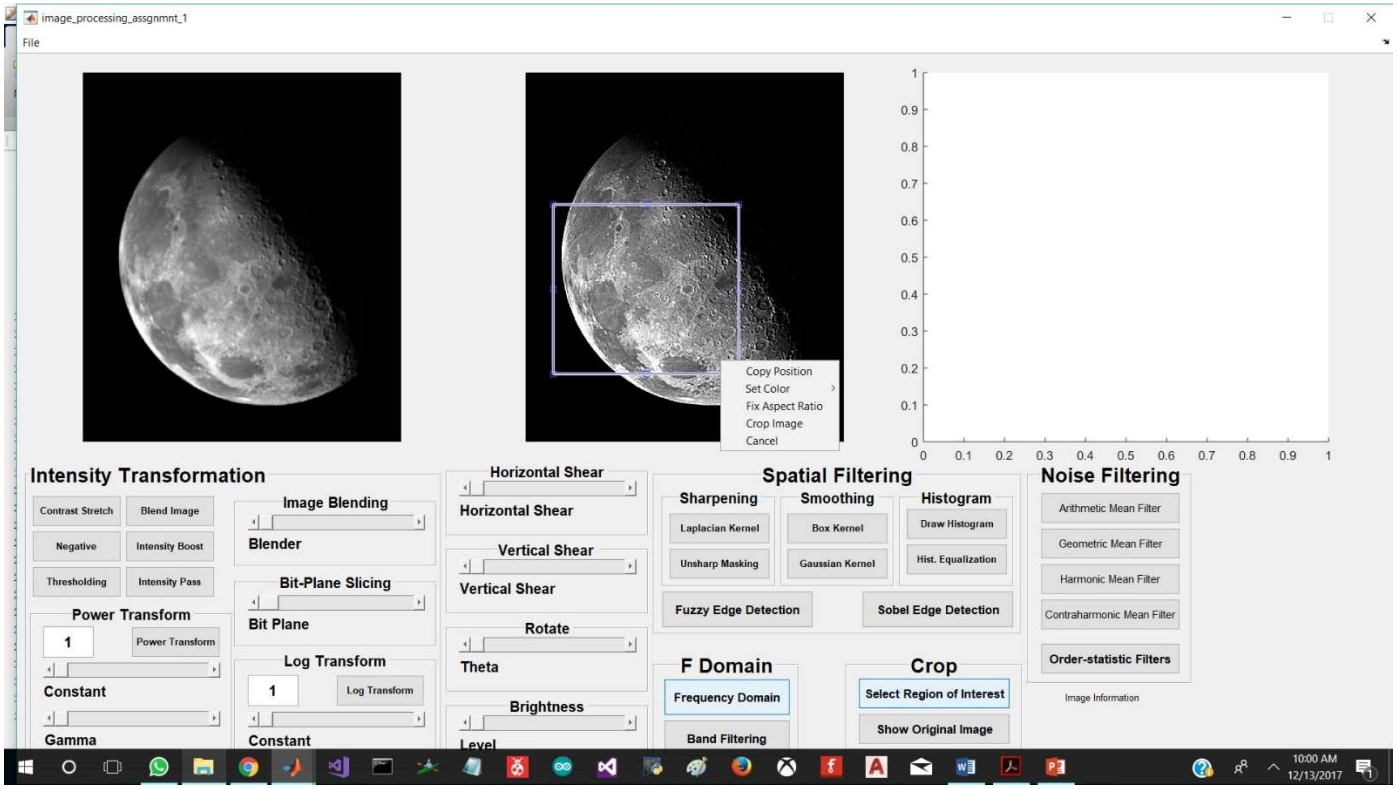
Below are the screenshots of MATLAB GUI:



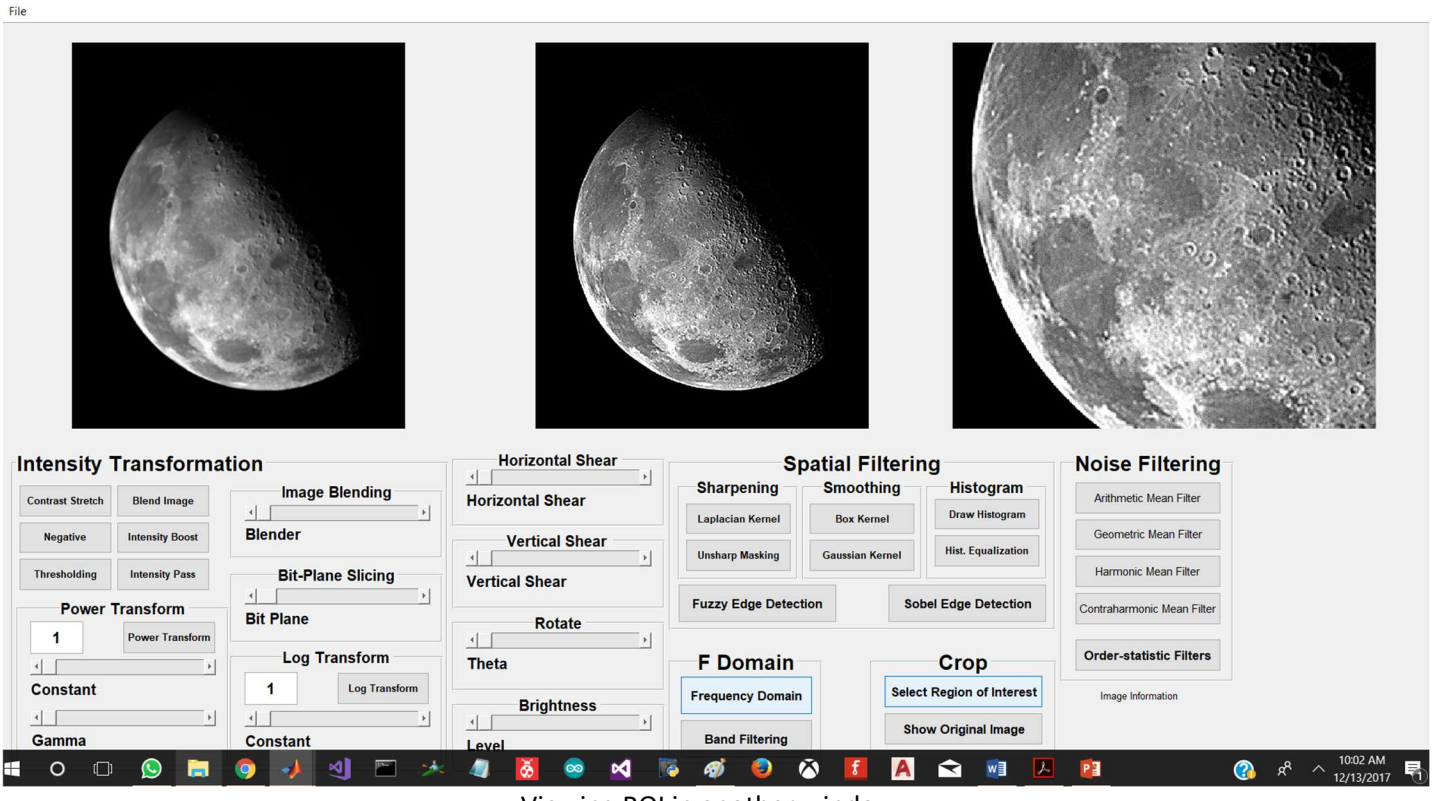
Opening the image



Processed Image



Selecting region of Interest



Viewing ROI in another window

B. MATLAB GUI 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 10-Dec-2017 17:18:06

% 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.imProcessing.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.imProcessing.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_histogram.
function draw_histogram_Callback(hObject, eventdata, handles)
% hObject    handle to draw_histogram (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);
figure;
handles.imProcessing = handles.imProcessing.drawHist;
guidata(hObject, handles);


% --- 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);


% --- Executes on button press in histogram_equalization.
function histogram_equalization_Callback(hObject, eventdata, handles)
% hObject    handle to histogram_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 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


% --- Executes on button press in box_kernel.
function box_kernel_Callback(hObject, eventdata, handles)
% hObject    handle to box_kernel (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);


% --- Executes on button press in sobel_operator_edge_detection.
function sobel_operator_edge_detection_Callback(hObject, eventdata, handles)
% hObject    handle to sobel_operator_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);

% --- Executes on button press in gaussian_kernel.
function gaussian_kernel_Callback(hObject, eventdata, handles)
% hObject    handle to gaussian_kernel (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;
guidata(hObject, handles);
```

```
% --- Executes on button press in laplacian_kernel.
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);


% --- Executes on button press in fuzzy_set_edge_detection.
function fuzzy_set_edge_detection_Callback(hObject, eventdata, handles)
% hObject    handle to fuzzy_set_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);


% --- Executes on button press in unsharp_masking.
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);


% --- Executes on button press in frequency_domain_filtering.
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);


% --- Executes on button press in freq_domain_band_filtering.
function freq_domain_band_filtering_Callback(hObject, eventdata, handles)
% hObject    handle to freq_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);

% --- Executes on button press in arithmetic_mean_filter.
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);
```

```
% --- Executes on button press in geometric_mean_filter.
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);


% --- Executes on button press in harmonic_mean_filter.
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);

% --- Executes on button press in contraharmonic_mean_filter.
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);

% --- Executes on button press in order_statistic_filtering.
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);


% --- Executes on button press in select_region.
function select_region_Callback(hObject, eventdata, handles)
% hObject    handle to select_region (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);


% --- Executes on button press in show_original.
function show_original_Callback(hObject, eventdata, handles)
% hObject    handle to show_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);
```

C. List of Function Working

```
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...
        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...
        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...

        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('All
files', '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 = uint8(zeros(size(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);
            imwrite(imProcess.imgProcessed, imgName, imProcess.imgInfo.Format);

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

            imshow(imProcess.img);

        end

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

            imshow(imProcess.imgProcessed);

        end

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

            if imProcess.constant
                imProcess.constant = uint8(imProcess.constant);
                imshow(imProcess.constant * 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 = drawHist (imProcess) % Draws histogram

            pixelvals = unique(imProcess.img);
            mySum = 0;
            sumOfProb = zeros(size(pixelvals));
            newPixelVals = zeros(size(pixelvals));
            hist_data = zeros(size(pixelvals));
            imProcess.norm_hist_data = zeros(size(pixelvals));

            for n = 1:1:numrows(pixelvals);
                for x = 1:1:numrows(imProcess.img);
                    for y = 1:1:numcols(imProcess.img);

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

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

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

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

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

            subplot(2,1,1);     bar(pixelvals, 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(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)

            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)

            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:numrows(imProcess.img);
                    for y = 1:1:numcols(imProcess.img);
                        imProcess.imgProcessed(x,y,z) = uint16(((2^imProcess.imgInfo.BitDepth)-1) -
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:numrows(imProcess.img);
                    for y = 1:1:numcols(imProcess.img);
                        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) = uint8(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)

            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)

            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;
            %disp(imProcess.gaussKernel);
            imProcess = imProcess.sumOfArray(imProcess.gaussKernel);
            mySum = imProcess.arraySum;
            newImage = padarray(imProcess.img, [(size(imProcess.gaussKernel, 1) - 1)/2
(size(imProcess.gaussKernel, 2) - 1)/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) = uint8(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);
                        total = (myArraySum(lapKernel) - 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);
```

```
                        total = (myArraySum(lapKernel) - 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;
                        total = (myArraySum(lapKernel) - 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;
                        total = (myArraySum(lapKernel) - 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);
                %disp(imProcess.laplacianKernelType);

            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]);

            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) = uint8(imProcess.arraySum);
                    end
                end
            end
            figure;
            imProcess = imProcess.showProcessed;
            switch imProcess.laplacianKernelType;
                case 'A'
                    imProcess.imgProcessed = imProcess.img - imProcess.imgProcessed;
                case 'B'
                    imProcess.imgProcessed = imProcess.img - imProcess.imgProcessed;
                case 'C'
                    imProcess.imgProcessed = imProcess.img + imProcess.imgProcessed;
                case 'D'
                    imProcess.imgProcessed = 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 = imProcess.img - imProcess.imgProcessed; % mask
            figure;
            imProcess = imProcess.showProcessed; % show mask
            imProcess.imgProcessed = imProcess.img + 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);
            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);
            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);

            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]);
            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) = uint8(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]);
            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]);

            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(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]);

            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) = uint8(numerator/denominator);
                    end
                end
            end

        end

        function imProcess = orderStatFiltParamtr (imProcess, array)

            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)

            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]);

            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) =
uint8(imProcess.medianIntensity);
                                case 2; % Max filter is selected...
                                    imProcess.imgProcessed(x,y,z) = uint8(imProcess.maxIntensity);
                                case 3; % Min filter is selected...
                                    imProcess.imgProcessed(x,y,z) = uint8(imProcess.minIntensity);
                                case 4; % Midpoint filter is selected...
                                    imProcess.imgProcessed(x,y,z) = uint8(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 ===============

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

D. 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. 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.