Digital Image Processing

Experiment#1	Introduction to MATLAB Digital Image Processing Toolbox. To get familiar with some simple commands related to reading and displaying images using MATLAB.
Experiment#2	To get familiar with commands related to writing images on disk. To get familiar with image classes and conversion between image classes.
Experiment#3	To get familiar with matrix manipulation commands. To use matrix manipulation commands for rotating and compressing images.
Experiment#4	To get familiar with basic intensity transformation functions. To get familiar with M-file programming.
Experiment#5	To get familiar with histogram processing of images.
Experiment#6	To get familiar with histogram matching, local histogram processing and filtering.
Experiment#7	To get familiar with Linear and Non-Linear Filtering.
Experiment#8	To get familiar with morphological operations

Experiment#1: Introduction to MATLAB Digital Image Processing Toolbox. To get familiar with some simple commands related to reading and displaying images using MATLAB.

1. **Experiment Text**
    1. **Digital Image Representation**
    2. **Reading Images in MATLAB**
    3. **Displaying Images**
2. **Lab Exercise**
3. **Exercise Questions**

Experiment Text

1. Digital Image Representation

A digital image may be defined as a two dimensional function f (x, y), where x and y are spatial co-ordinates and the value of f at any value of x and y is called the intensity of image at that point. An image when captured may be continuous with respect to x and y co-ordinates and also in intensity. Converting such image to digital form means, that the co-ordinates as well as intensity is digitized. Digitizing the co-ordinates is called sampling, while digitizing intensity is called quantization. When co-ordinates x and y and intensity values of f are all finite, discrete quantities, the image is called digital image. In this lab we will use MATLAB for digital image processing. MATLAB has a special toolbox for dealing with images known as Image Processing Toolbox.

Let us suppose an image f (x, y) is sampled such that it has M rows and N columns. We say that this image is of size M x N where x and y can take positive integer values. Although normally the values of x and y start from (0,0) but in MATLAB they always start from (1,1) since MATLAB cannot have zero index. The first value in the parenthesis i.e. x represents row while the second value y represents column e.g. notation (1,2) is used to signify the second sample along the first row. One another difference is that MAT LAB image processing toolbox uses the notation (r, c) instead of (x, y) where r and c represent row and column respectively. A typical digital image is shown in Figure 1.

$$f = \begin{bmatrix} f(1,1) & f(1,2) & \dots & f(1,N) \\ f(2,1) & f(2,2) & \dots & f(2,N) \\ . & . & & . \\ . & . & & . \\ . & . & & . \\ f(M,1) & f(M,2) & \dots & f(M,N) \end{bmatrix}$$

A digital image is represented as a matrix in MATLAB. A typical image matrix in MAT-LAB is shown below.

2. Reading Images in MATLAB

Images are read in MATLAB using command

```
>> imread('filename')
```

where 'filename' is full name of an image file including extension. e.g.

```
>> f = imread('cameraman.tif');
```

reads the content of image 'cameraman.tif' in variable f. The above command reads an image from the current directory of the MATLAB. The path of the current directory of MATLAB is normally shown on top of command Window. Figure 2 shows different sections of a MATLAB window.

```
f=imread('cameraman.tif')

130  127  130  127  126  125  126  135  129  121  122  122  130  132  128  123  132  131
.
.
.

117  133  130  113
```

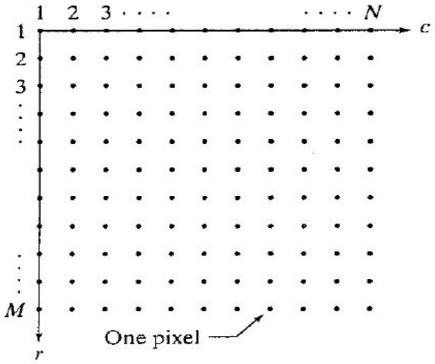
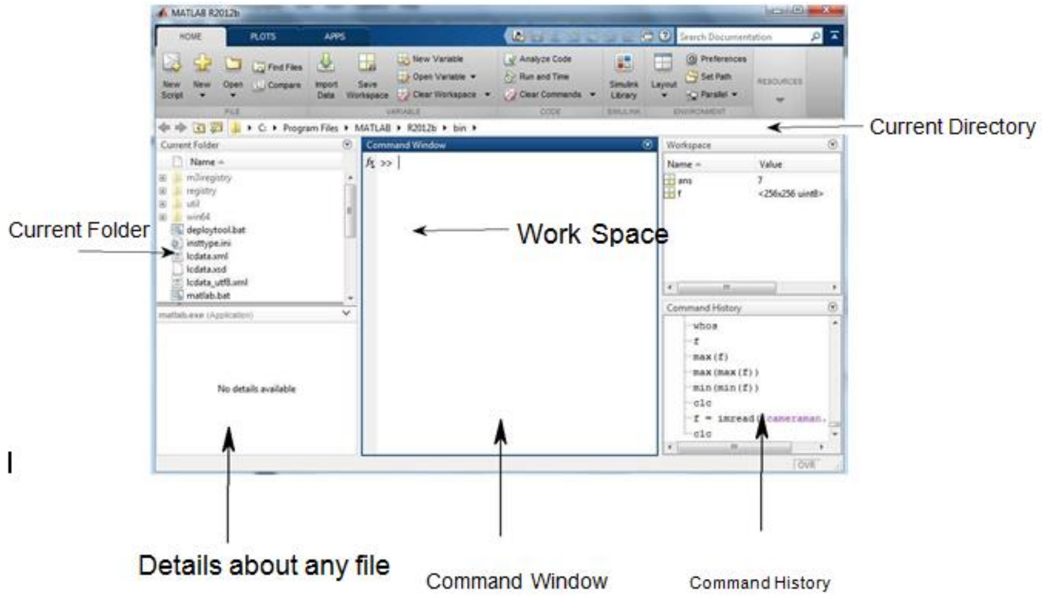


Figure 1: A typical digital image



```
whos
f
max(f)
max(max(f))
min(min(f))
clc
f = imread('cameraman.
clc
```

Figure 2: MATLAB Window

If you want to read file from a specific directory then you need to give the full path of that directory with 'filename' e.g. the following command reads an image from D: drive from folder named as images.

```
>> f = imread ('D:\images\cameraman.tif');
```

```
t=imread('E:\ImageProcessing\cameraman.tif')
```

```
124  118  119  119  126  126  120  123  118  111  120  116  118  116  113  108  117  120
.
.
.
137  153  150  133
```

The size of the image can be read using the following command.

```
>> size(f)
```

```
>> size_of_image_f=size(f)
```

```
size_of_image_f =

  256   256
```

This command gives you the number of rows and columns of the image. You can also use the following command

```
>> [M N ] = size(f)
```

where the number of rows and columns will be saved in M and N respectively.

```
>>[M N]=size(f)
```

```
>> [M N]=size(f)
M =
  256
N =
  256
>> [M N R]=size(f)
M =
  256
N =
  256
R =
    1
```

The function whos can be used to get additional information about an image. For instance

```
>> whos
```

will give you details about f e.g. it will tell you the size, number of bytes taken and class of the image.

```
>> whos
  Name                 Size          Bytes         Class    Attributes
  M                    1x1           8             double
  N                    1x1           8             double
  R                    1x1           8             double
  ans                  1x256         256           uint8
  f                    256x256       65536         uint8
  g                    256x256       65536         uint8
  size_of_image_f       1x2          16            double
  t                    256x256       65536         uint8

>> whos f
  Name         Size          Bytes         Class    Attributes
  f            256x256       65536         uint8
```

3 Displaying Images

Images are shown in MATLAB by using command imshow e.g.

```
>> imshow(f)
```

will display the image read in variable f.

```
>>imshow(f);
```

The command

```
>> imshow(f, [low high])
```

displays all the pixels as black which are less than or equal to low and all values as white which are greater than or equal to high. All the values in between are displayed as intermediate intensity values using the default number of levels.

```
>> imshow(f,[0 255])
```

```
>> imshow(f,[50 255])
```

```
>> imshow(f,[150 255])
```

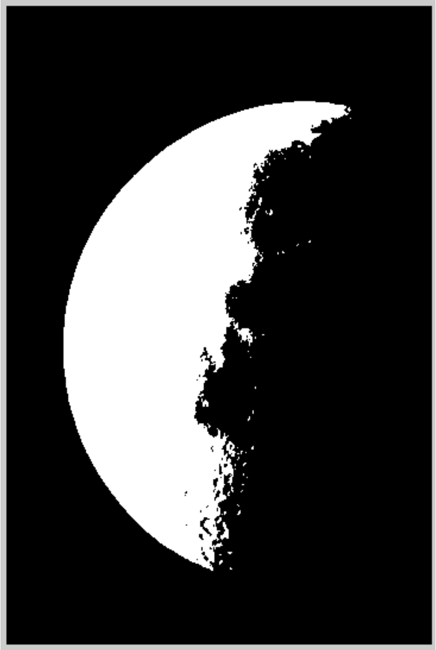
```
>> imshow(f,[0 230])
```

```
>> imshow(f,[0 150])
```

```
>> imshow(f,[50 150])
```

If you want to show images in separate figures, the command `figure` can be used. e.g. the following commands display two images in separate figure. If you have more than two figures you can use the command `figure` again.

```
>> figure
imshow('moon.tif')
figure
f=imread('moon.tif');
imshow(f,[127,128])
```



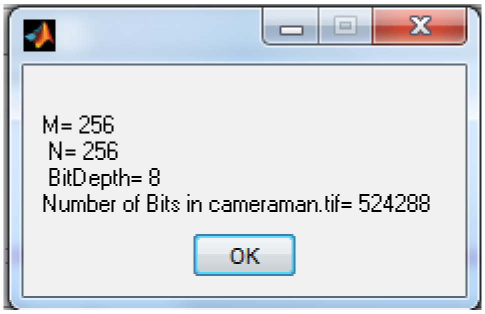
```
>> min(f)
min(min(f))
ans =
  Columns 1 through 18
   64   66   63   64   63   62   61   63   61   61   62   62   63   64   62   63   63   62
   …………………….
  Columns 253 through 256
   26   27   21   24
ans =
    7
>> max(f)
max(max(f))
ans =
  Columns 1 through 18
  166  168  166  169  169  174  188  187  181  194  204  173  170  203  196  175  193  178
  ……………..
  Columns 253 through 256
  177  173  172  170
```

```
ans =
  253
```

EXERCISE QUESTIONS

Q1 : Calculate the number of bits required to represent one intensity level in image 'cameraman.tif'. (Hint: The total size of an image in bits is MxNxk where k represents the number of bits required for one intensity level.)

```
f=imread('cameraman.tif');
info=imfinfo('cameraman.tif');
k=info.BitDepth;
[M N]=size(f);
total_imbits=M*N*k;
output=sprintf('M= %d \n N= %d \n BitDepth= %d \nNumber of Bits in cameraman.tif= %d',M,N,k,total_imbits);
msgbox(output);
```



```
M= 256
 N= 256
 BitDepth= 8
Number of Bits in cameraman.tif= 524288
```

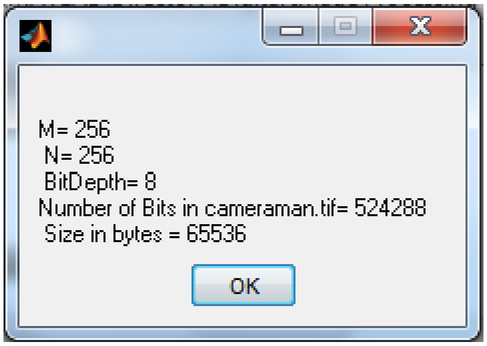
```
clc
f=imread('cameraman.tif');
[m n]=size(f)
whos f
bit_depth=8*((m*n)/65536)
tot_bits=m*n*bit_depth
```

```
m =
   256
n =
   256
```

```
  Name        Size        Bytes      Class    Attributes

  f           256x256     65536      uint8
bit_depth =
    8
tot_bits =
     524288
```

Q2 : Now recalculate the size of image in bytes from k. Does this result tally with the result given by whos? (Hint: Size in bytes = (MxNxk)/8))

```
>> f=imread('cameraman.tif');
info=imfinfo('cameraman.tif');
k=info.BitDepth;
[M N]=size(f);
total_imbits=M*N*k;
size_in_bytes=total_imbits/8;
output=sprintf('M= %d \n N= %d \n BitDepth= %d \nNumber of Bits in cameraman.tif= %d \n Size in bytes = %d ',M,N,k,total_imbits,size_in_bytes);
msgbox(output);
```



```
M= 256
 N= 256
 BitDepth= 8
Number of Bits in cameraman.tif= 524288
 Size in bytes = 65536
```

```
>> f=imread('cameraman.tif');
[m n]=size(f)
whos f
bit_depth=8*((m*n)/65536)
tot_bits=m*n*bit_depth;
tot_bytes=tot_bits/8
```

```
whos f
tot_bytes =
      65536
  Name       Size            Bytes          Class    Attributes
  f          256x256          65536         uint8
```

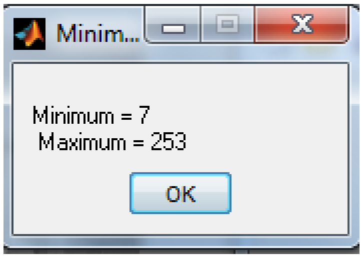
Q3 : Find the minimum and maximum intensity value used in image 'cameraman.tif'. (Hint: The intensity values are stored in 'f', and minimum and maximum values can be found using MATLAB built in commands `min` and `max`.

```
>> f=imread('cameraman.tif');
minimum_values=min(f);
mini=min(min(f))
maxi=max(max(f))
mini =
    7
maxi =
  253
```

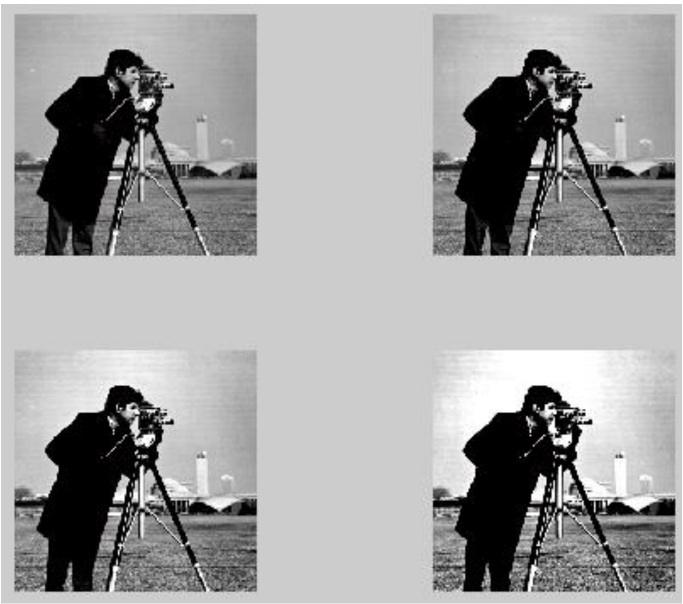
Q4 : Now use some value of `low` and `high` in command `imshow(f, [low high])` based on the minimum and maximum value found in Q3. You can show many images with different values of `low` and `high` on same figure using command `subplot`. Use different values of `low` and `high` and see the changes in the image.

```
>> f=imread('cameraman.tif');
subplot(1,2,1)
imshow(f);
minimum_values=min(f);
mini=min(min(f))
maxi=max(max(f))
subplot(1,2,2);
imshow(f,[mini maxi])
h=sprintf('Minimum = %d \n Maximum = %d ', mini,maxi);
msgbox(h,'Minimum/Maximum Value')
figure
subplot(2,2,1);
```

```
imshow(f,[20 235])
subplot(2,2,2);
imshow(f,[40 215])
subplot(2,2,3);
imshow(f,[60 195])
subplot(2,2,4);
imshow(f,[80 175])
mini =
    7
maxi =
  253
```



```
Minimum = 7
 Maximum = 253
```



Experiment#2: To get familiar with commands related to writing images on disk. To get familiar with image classes and conversion between image classes.

1. **Experiment Text**
    i. **Writing Images**
    ii. **Image Classes**
    iii. **Image Types**
    iv. **Conversion Between Classes**
2. **Lab Exercise**
3. **Exercise Questions**

EXPERIMENT TEXT

1 Writing Images

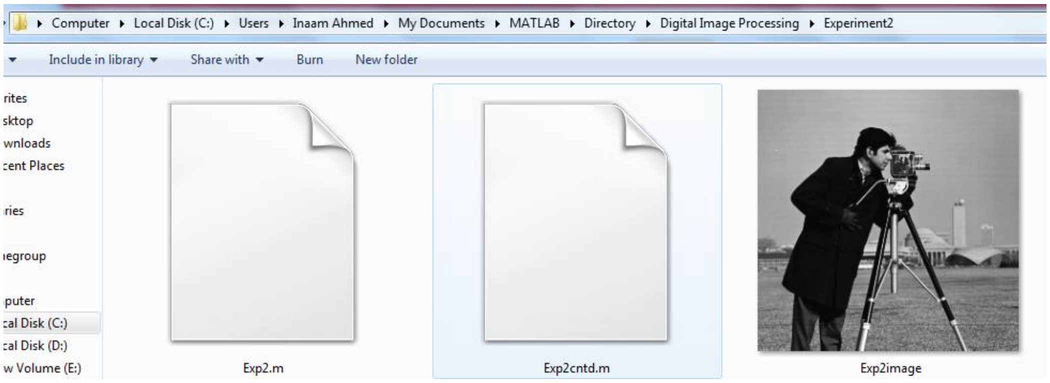
Images can be written to disk using command `imwrite(f,‘filename’)` e.g.

```
>> imwrite(f,‘cameraman1.tif’)
```

will save the contents of variable `f` in the form of an image named `cameraman1.tif'. The variable `f` should have contents of a valid image. (Hint: You can use command `f = imread()` to load contents of an image into `f`).

```
>> f=imread('cameraman.tif');
```

```
imshow(f)
imwrite(f,'Exp2image.tif');
```

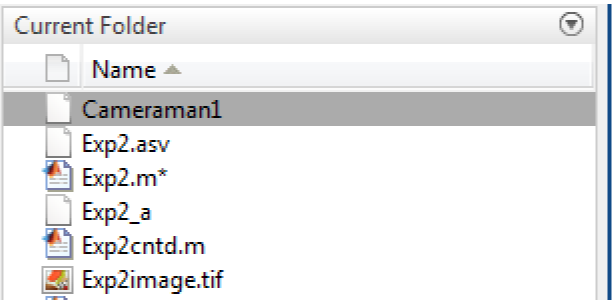


In the command shown above `filename' should have a recognized file format extension. Alternatively, the desired format can be specified explicitly with a third input argument. e.g.

```
>> imwrite(f,'cameraman1','tif')
```

saves the contents of f in .tif format in file named cameraman1.

```
imwrite(f,'Cameraman1','tif');
```



If `filename' contains no path information, imwrite will save the image in the current directory. If you want to save the image in another directory, you need to give full path with `filename'. e.g. the following command will save the image in drive D.

```
>> imwrite(f,‘D:\cameraman1.tif’)
imwrite(f,'E:ImageProcessing\Exp2image.tif');
```

To see the detailed information of an image, the command imfinfo filename can be used where filename is the name of the image stored on the disk. e.g. the following command will give detailed information about image cameraman.tif

```
>> imfinfo cameraman.tif
>> struct=imfinfo('cameraman.tif')
struct =
Filename: [1x67 char]
FileModDate: '04-Dec-2000 13:57:54'
FileSize: 65240
Format: 'tif'
FormatVersion: []
Width: 256
```

```
Height: 256
BitDepth: 8
ColorType: 'grayscale'
.
.
.
ImageDescription: [1x112 char]
```

The information fields displayed by imfinfo can be captured into a so-called structure variable that can be used for subsequent computations. Using the preceding image as an example, and letting K denote the structure variable, we use the syntax

```
>> K = imfinfo('cameraman.tif');
```

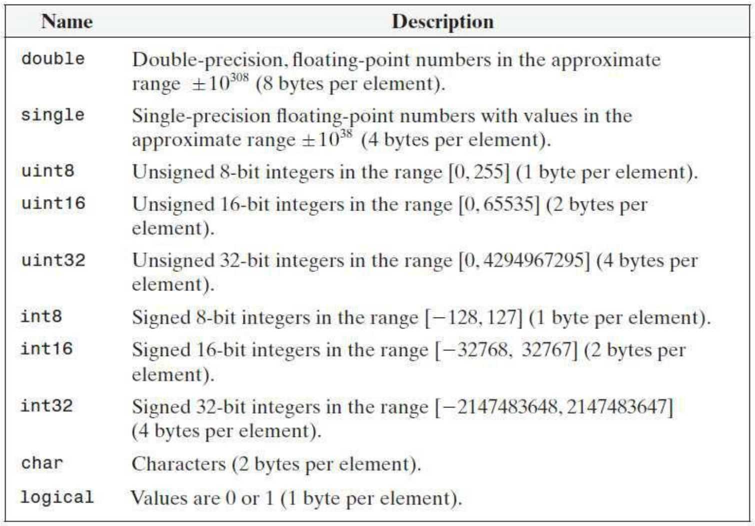
to store into variable K all the information generated by command imfinfo. The information generated by imfinfo is appended to the structure variable by means of fields, separated from K by a dot. For example, the image height and width are now stored in structure fields K.Height and K.Width which can be accessed using the following command.

```
>> K.Height
```

```
>> struct.Format
struct.Offset
struct.BitDepth
ans =
tif
ans =
     64872
ans =
   8
```

2 Image Classes

Although we work with integer coordinates, the intensities of pixels are not restricted to be integers in MATLAB. Figure 1 lists the various classes supported by MATLAB and the Image Processing Toolbox for representing pixel values. Classes **uint8 and logical** are used extensively in image processing, and they are the usual classes encountered when reading images from image file formats such as TIFF or JPEG. These classes use 1 byte to represent each pixel. Some scientific data sources, such as medical imagery, require more dynamic range than is provided by uint8, so the **uint16 and int16** classes are used often for such data. These classes use 2 bytes for each array element. The classes double and single are used for computationally intensive operations such as the Fourier transform. The int8, uint32, and int32 classes, although supported by the toolbox, are not used commonly for image processing. There are a quite a few images available in the MATLAB toolbox. A list is given in Figure 2.



Name	Description
`double`	Double-precision, floating-point numbers in the approximate range $\pm 10^{308}$ (8 bytes per element).
`single`	Single-precision floating-point numbers with values in the approximate range $\pm 10^{38}$ (4 bytes per element).
`uint8`	Unsigned 8-bit integers in the range [0, 255] (1 byte per element).
`uint16`	Unsigned 16-bit integers in the range [0, 65535] (2 bytes per element).
`uint32`	Unsigned 32-bit integers in the range [0, 4294967295] (4 bytes per element).
`int8`	Signed 8-bit integers in the range [−128, 127] (1 byte per element).
`int16`	Signed 16-bit integers in the range [−32768, 32767] (2 bytes per element).
`int32`	Signed 32-bit integers in the range [−2147483648, 2147483647] (4 bytes per element).
`char`	Characters (2 bytes per element).
`logical`	Values are 0 or 1 (1 byte per element).

Figure 1: Classes used for Images in MATLAB

3 Image Types

MATLAB toolbox supports four types of images Gray-scale, Binary, Indexed and RGB. Most monochrome image processing operations are carried out using binary or gray-scale images, so our initial focus is on these two image types.

3.1 Gray-scale Images

A gray-scale image is a data matrix whose values represent shades of gray. When the elements of a gray-scale image are of class uint8 or uint16, they have integer values in the range [0, 255] or [0, 65535], respectively. If the image is of class double or single, the values are floating-point numbers. Values of double and single gray-scale images normally are scaled in the range [0, 1], although other ranges can be used.

3.2 Binary Images

Binary images have a very specific meaning in MATLAB. A binary image is a logical array of 0s and 1s. Thus, an array of 0s and 1s whose values are of data class, say, uint8, is not considered a binary image in MATLAB. A numeric array is converted to binary using function logical. Thus, if A is a numeric array consisting of 0s and 1s, we create a logical array B using the statement

```
       AT3_1m4_01.tif              AT3_1m4_02.tif
       AT3_1m4_03.tif              AT3_1m4_04.tif
       AT3_1m4_05.tif              AT3_1m4_06.tif
       AT3_1m4_07.tif              AT3_1m4_08.tif
       AT3_1m4_09.tif              AT3_1m4_10.tif
           autumn.tif                     bag.png
            blobs.png                   board.tif
        cameraman.tif                   canoe.tif
             cell.tif                  circbw.tif
          circles.png      circlesBrightDark.png
          circuit.tif                   coins.png
     coloredChips.png          concordaerial.png
concordorthophoto.png                   eight.tif
           fabric.png                football.jpg
           forest.tif             gantrycrane.png
            glass.png                  greens.jpg
          hestain.png                    kids.tif
      liftingbody.png                    logo.tif
              m83.tif                   mandi.tif
             moon.tif                     mri.tif
         office_1.jpg                office_2.jpg
         office_3.jpg                office_4.jpg
         office_5.jpg                office_6.jpg
            onion.png                  paper1.tif
            pears.png                 peppers.png
         pillsetc.png                    pout.tif
             rice.png                  saturn.png
           shadow.tif              snowflakes.png
            spine.tif                    tape.png
         testpat1.png                    text.png
             tire.tif                  tissue.png
            trees.tif      westconcordaerial.png
westconcordorthophoto.png
```

B = logical (A)

cameraman=imread('cameraman.tif');

b=logical(cameraman)

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

.

.

.

```
1    1    1    1    1    1    1    1    1    1    1    1    1    1    1
```

If A contains elements other than 0s and 1s, the logical function converts all nonzero quantities to logical 1s and all entries with value 0 to logical 0s. Using relational and logical operators also results in logical arrays. To test if an array is of class logical we use the islogical function : islogical(C) If C is a logical array, this function returns a 1. Otherwise it returns a 0.

```
>> islogical(b)
ans =
    1
>> whos b
  Name          Size                    Bytes           Class      Attributes


  b             256x256                 65536           logical
```

An image is characterized by both a class and a type. For instance, a statement discussing a “uint8 gray-scale image” is simply referring to a gray-scale (type) image whose pixels are of class uint8 [1].

4 Conversion Between Classes

Converting images from one class to another is a common operation. When converting between classes, keep in mind the value ranges of the classes being converted (See Figure 1).

The general syntax for class conversion is

B = class_name(A)

where class name is one of the names in the first column of Figure 1. For example, suppose that A is an array of class uint8. A double-precision array, B, is generated by the command B = double(A).

```
>> cameraman=imread('cameraman.tif');
>> whos cameraman
  Name           Size          Bytes   Class    Attributes


  cameraman   256x256       65536    uint8
```

```
>> b=double(cameraman);
>> whos b
  Name       Size             Bytes  Class     Attributes


  b          256x256          524288  double
```

If C is an array of class double in which all values are in the range [0, 255] (but possibly containing fractional values), it can be converted to an uint8 array with the command D = uint8(C). If an array of class double has any values outside the range [0, 255] and it is converted to class uint8 in the manner just described, MATLAB converts to 0 all values that are less than 0, and converts to 255 all values that are greater than 255. Numbers in between are rounded to the nearest integer. Thus, proper scaling of a double array so that its elements are in the range [0, 255] is necessary before converting it to uint8.

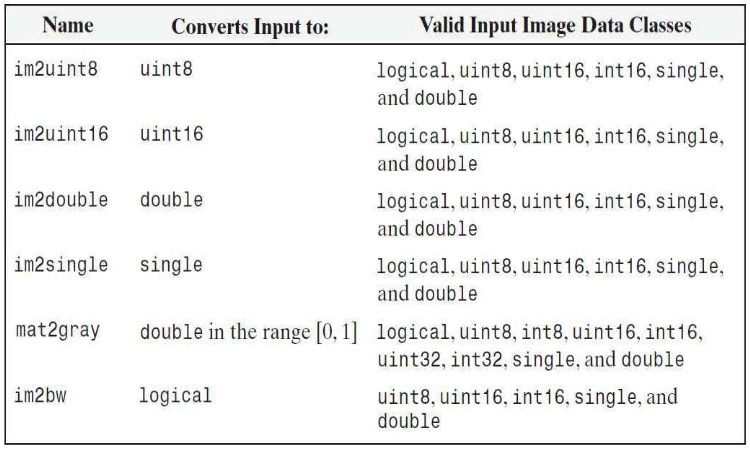
```
>> u=1000*rand(3,3)
u =
  529.4502   260.7776   998.9270
  931.9208   453.7062   502.2118
  332.4853     1.2837    80.8436

  Name      Size           Bytes      Class     Attributes
  u           3x3              72       double
>> un=uint8(u)
un =
  255   255   255
  255   255   255
  255     1     81
```

Converting any of the numeric data classes to logical creates an array with logical 1s in locations where the input array has nonzero values, and logical 0s in places where the input array contains 0s.

```
>> u=1000*rand(3,3)
u1=u-min(min(u))-1
l=logical(u)
u =

  334.2572    42.4156   241.4016
  914.4002   400.0491   296.6317
```

```
  590.3546   872.6598   470.5874

u1 =

  290.8416   -1.0000     197.9860
  870.9846   356.6335   253.2161
  546.9390   829.2442   427.1718

l =

     1      1      1
     1      1      1
     1      1      1
```

The toolbox provides specific functions that perform the scaling and other book keeping necessary to convert images from one class to another. The functions are given in Figure 3.



Name	Converts Input to:	Valid Input Image Data Classes
`im2uint8`	`uint8`	`logical`, `uint8`, `uint16`, `int16`, `single`, and `double`
`im2uint16`	`uint16`	`logical`, `uint8`, `uint16`, `int16`, `single`, and `double`
`im2double`	`double`	`logical`, `uint8`, `uint16`, `int16`, `single`, and `double`
`im2single`	`single`	`logical`, `uint8`, `uint16`, `int16`, `single`, and `double`
`mat2gray`	`double` in the range [0, 1]	`logical`, `uint8`, `int8`, `uint16`, `int16`, `uint32`, `int32`, `single`, and `double`
`im2bw`	`logical`	`uint8`, `uint16`, `int16`, `single`, and `double`

Figure 3: Conversion functions between different classes

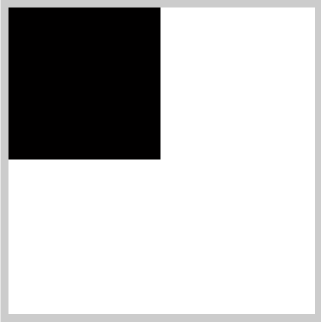
Function im2uint8, for example, creates a unit8 image after detecting the data class of the input and performing all the necessary scaling for the toolbox to **recognize the data as valid image data**. For example, consider the following image f of class double, which could be the result of an intermediate computation:

```
>> f=[-0.5 0.5;0.75 1.5]

f =

   -0.5000    0.5000
    0.7500    1.5000
>> whos f
  Name      Size            Bytes    Class     Attributes

  f         2x2                     32    double
```

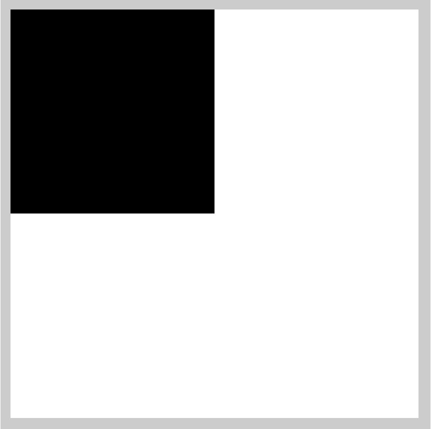
Performing the conversion

```
>> g=double(f)

g =

   -0.5000    0.5000
    0.7500    1.5000

>> whos g

  Name      Size            Bytes  Class     Attributes
  g         2x2                32  double

>> g1= im2double(f)

g1 =


   -0.5000     0.5000
    0.7500     1.5000

>> whos g1

  Name      Size            Bytes  Class     Attributes

  g1        2x2                32  double

>> h=uint8(f)

h =

    0    1
    1    2

>> whos h

  Name      Size            Bytes   Class

  h         2x2                 4         uint8

>> h=im2uint8(f)

h =

    0     128

  191   255


>> h2=uint8(round(f*255))

h2 =

    0   128
  191   255


>> j= [-0.5 50; 255 755]
```

```
j =
   -0.5000     50.0000
  255.0000   755.0000

  Name      Size           Bytes   Class

  j         2x2                  32       double

imshow(j,'InitialMagnification',1500)
```

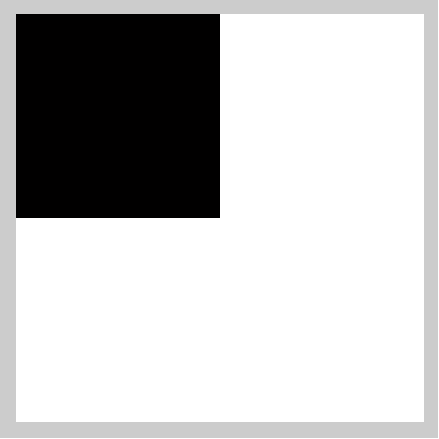


```
>> d= double (j)
d =
   -0.5000    50.0000
  255.0000   755.0000
>> imshow(d,'InitialMagnification',2000)
```



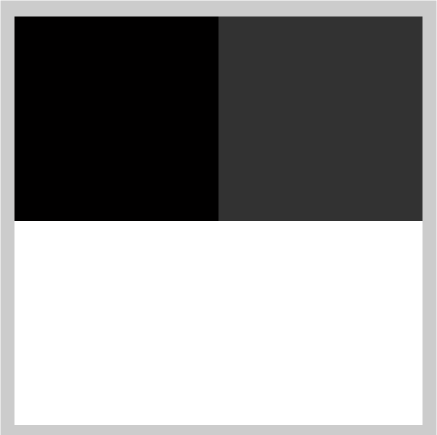
```
>> doub=im2double(j)
doub =
   -0.5000   50.0000
```

```
  255.0000   755.0000
```



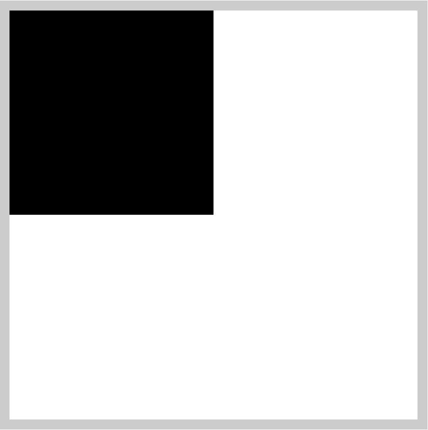
From which we see that function im2uint8 sets to 0 all values in the input that are less than 0, sets to 255 all values in the input that are greater than 1, and multiplies all other values by 255. Rounding the results of the multiplication to the nearest integer completes the conversion.

```
>> ui=uint8(j)
ui =
    0     50
  255   255
```



```
>> j= [-0.5 50; 255 755]
j =
   -0.5000     50.0000
  255.0000   755.0000
>> uint =im2uint8(j)
uint =
```

```
   0   255
 255   255
```

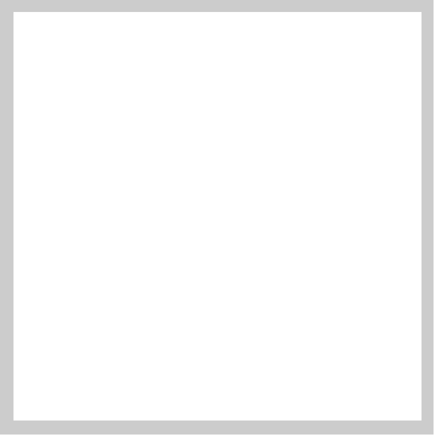


As an illustration, consider the following example

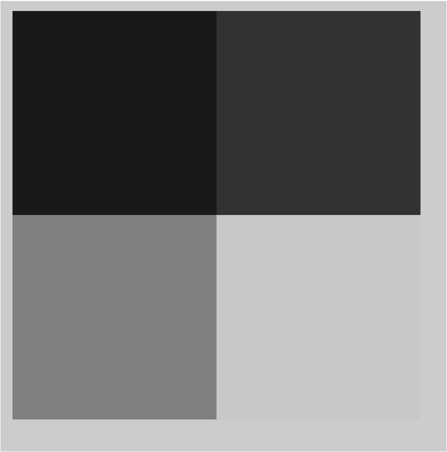
```
>>  h = uint8([25 50; 128 200]);
>>  g = im2double(h)
g =
  0.0980  0.1961
  0.4706  0.7843
```

from which we infer that the conversion when the input is of class uint8 is done simply by dividing each value of the input array by 255. If the input is of class uint16 the division is by 65535.

```
>> h=uint8 ([25 50;128 200])
h =
   25   50
  128   200
>> g1=double(h)
g1 =
    25    50
   128   200
```



```
g2 =


    0.0980    0.1961
    0.5020    0.7843
```



Function im2double converts an input to class double. If the input is of class uint8, uint16, or logical, function im2double converts it to class double with values in the range [0, 1]. If the input is of class single, or is already of class double, im2double returns an array that is of class double, but is numerically equal to the input. For example, if an array of class double results from computations that yield values outside the range [0, 1], inputting this array into im2double will have no effect. Function mat2gray can be used to convert an array of any of the classes to a double array with values in the range [0, 1].

Toolbox function mat2gray converts an image of any of the classes to an array of class double scaled to the range [0, 1]. The calling syntax is

```
g = mat2gray(A, [Amin, Amax])
```

where image g has values in the range 0 (black) to 1 (white). The specified parameters, Amin and Amax, are such that values less than Amin in A become 0 in g, and values greater than Amax in A correspond to 1 in g. Sets the values of Amin and Amax to the actual minimum and maximum values in A

```
>> f= [-0.5 50;345 90]
f =
   -0.5000    50.0000
  345.0000   90.0000
>> whos f
  Name           Size               Bytes             Class
  f                  2x2              32                  double
>> g1=double (f)
g1 =
   -0.5000   50.0000
  345.0000   90.0000

>> whos g1
  Name           Size               Bytes             Class     Attributes
  g1                2x2              32                  double

>> g2=im2double (f)
g2 =
   -0.5000      50.0000
  345.0000     90.0000

>> whos g2
  Name      Size           Bytes       Class     Attributes
  g2          2x2             32          double

>> g3=mat2gray (f)
g3 =
        0       0.1462
```

```
   1.0000     0.2619

>> whos g3
  Name      Size           Bytes    Class     Attributes
  g3          2x2               32      double
>> g4=mat2gray(f,[50 90])
g4 =
     0     0
     1     1
>> whos g4
  Name      Size                      Bytes              Class     Attributes
  g4          2x2                       32                  double

>> f=[-0.5 50;345 90]
f =
   -0.5000    50.0000
  345.0000   90.0000

>> g=f-min(f(:))
g =
        0        50.5000
  345.5000   90.5000

>> g=g/max(g(:))
g =
        0    0.1462
    1.0000    0.2619

>> whos g
  Name      Size                      Bytes              Class     Attributes
  g         2x2                       32                  double
```

```
>> mat2gray(f) % g and ans are same here
ans =
        0        0.1462
    1.0000     0.2619
```

The syntax

```
g = mat2gray(A)
```

The second syntax of mat2gray is a very useful tool because it scales the entire range of values in the input to the range [0, 1], independently of the class of the input, thus eliminating clipping.

Function im2double fails to create valid image data [0 --- 1] when input is in double and single in that case function mat2gray() is used.

```
>> f=uint8([-0.5 50;345 90])
f =
    0   50
  255   90

>> whos f
  Name      Size            Bytes     Class    Attributes
  f             2x2             4           uint8
>> g1=double(f)
g1 =
     0    50
   255    90

>> whos g1
  Name      Size            Bytes     Class     Attributes
  g1          2x2                32      double

>> g2=im2double(f)
g2 =
        0       0.1961
    1.0000    0.3529
```

```
>> whos g2
  Name      Size            Bytes   Class     Attributes
  g2        2x2               32      double

>> g3=mat2gray(f)
g3 =
         0       0.1961
    1.0000    0.3529

>> whos g3
  Name      Size            Bytes   Class     Attributes
  g3          2x2               32     double

>> g4=mat2gray(f,[50 90])
g4 =
     0     0
     1     1

>> whos g4
  Name      Size            Bytes   Class     Attributes
  g4        2x2               32        double

>> g=f-min(f(:))
g =
    0   50
  255   90

>> g=g/max(g(:))
g =
    0    0
    1    0
```

```
>> whos g
  Name      Size            Bytes   Class    Attributes
  g           2x2               4       uint8
```

```
>> s=mat2gray(f)
s =
         0       0.1961
    1.0000    0.3529
```

Finally, we consider conversion to class logical. (Recall that the Image Processing Tool-box treats logical matrices as binary images.) Function logical converts an input array to a logical array. In the process, nonzero elements in the input are converted to 1s, and 0s are converted to 0s in the output. An alternative conversion procedure that often is more useful is to use a relational operator, such as >, with a threshold value. For example, the syntax

g = f>t

Produces a logical matrix containing 1s wherever the elements of f are greater than t and 0s elsewhere

Toolbox function im2bw performs this thresholding operation in a way that automatically scales the specified threshold in different ways, depending on the class of the input image. The syntax is

im2bw(f,T)

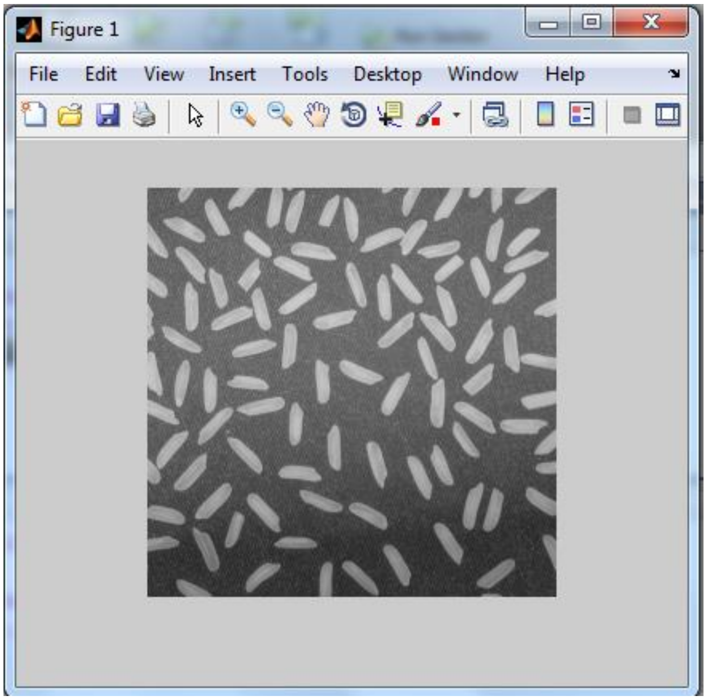
Values specified for the threshold T must be in the range [0, 1] , regardless of the class of the input. The function automatically scales the threshold value according to the input image class. For example, if f is uint8 and T is 0.4, then im2bw thresholds the pixels in f by comparing them to 255 * 0.4 = 102.

```
>> f = uint8 ([ 25 50; 128 200])


f =
   25    50
  128   200
>> s = im2double(f)


s =
    0.0980    0.1961
    0.5020    0.7843
```

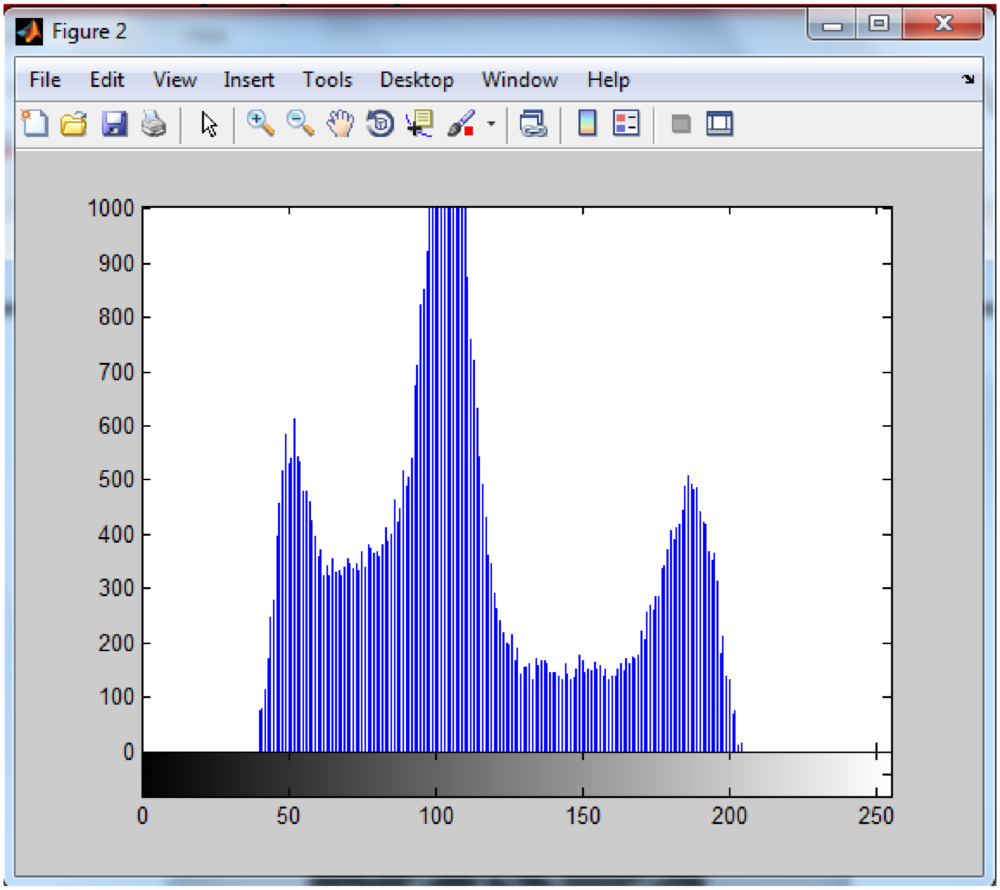
```
>> s = mat2gray(f)
s =
        0          0.1429
   0.5886        1.0000
>> g = f>150

g =


    0     0
    0     1


>> whos g
  Name      Size            Bytes           Class      Attributes


  g         2x2                 4                   logical
>> g = im2bw(f,0.5)

g =


    0     0
    1     1
```

LAB EXERCISE

```
y=imread('rice.png');
figure
imshow(y);
```

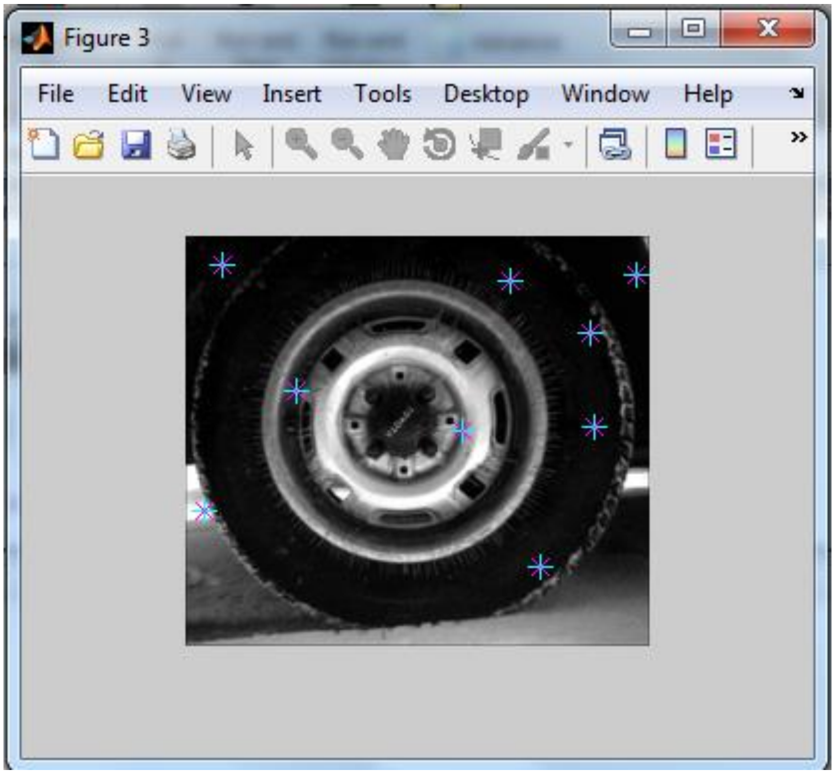


```
figure
imhist(y);
```



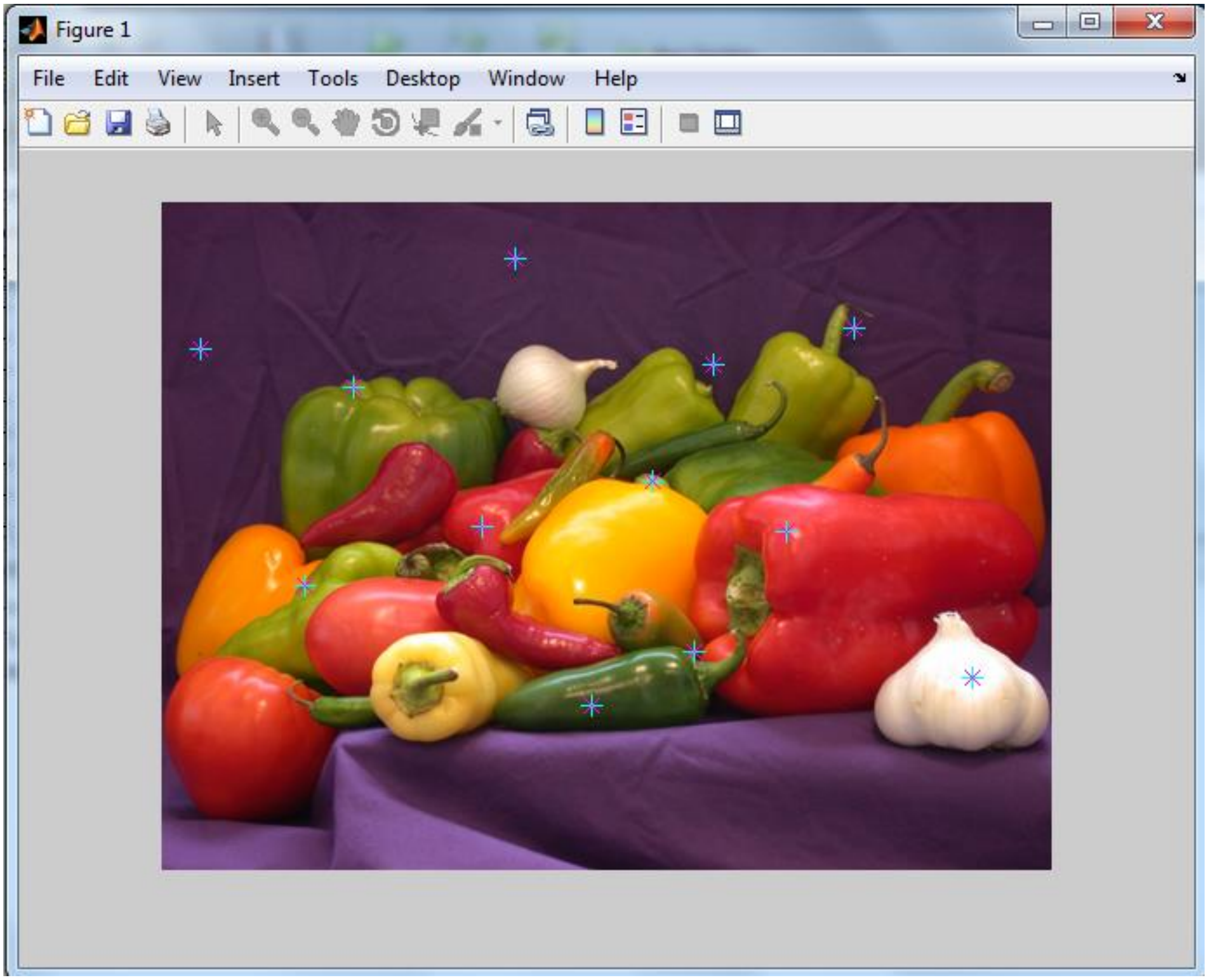
```
>> mean=mean2(y)
```

```
mean =
  111.2468
>> std=std2(y)
std =
   42.4935
>> corellation_of_y_f=corr2(y,f)
corellation_of_y_f =
    0.0968
>> tire=imread('tire.tif');
figure
impixel(tire)
```



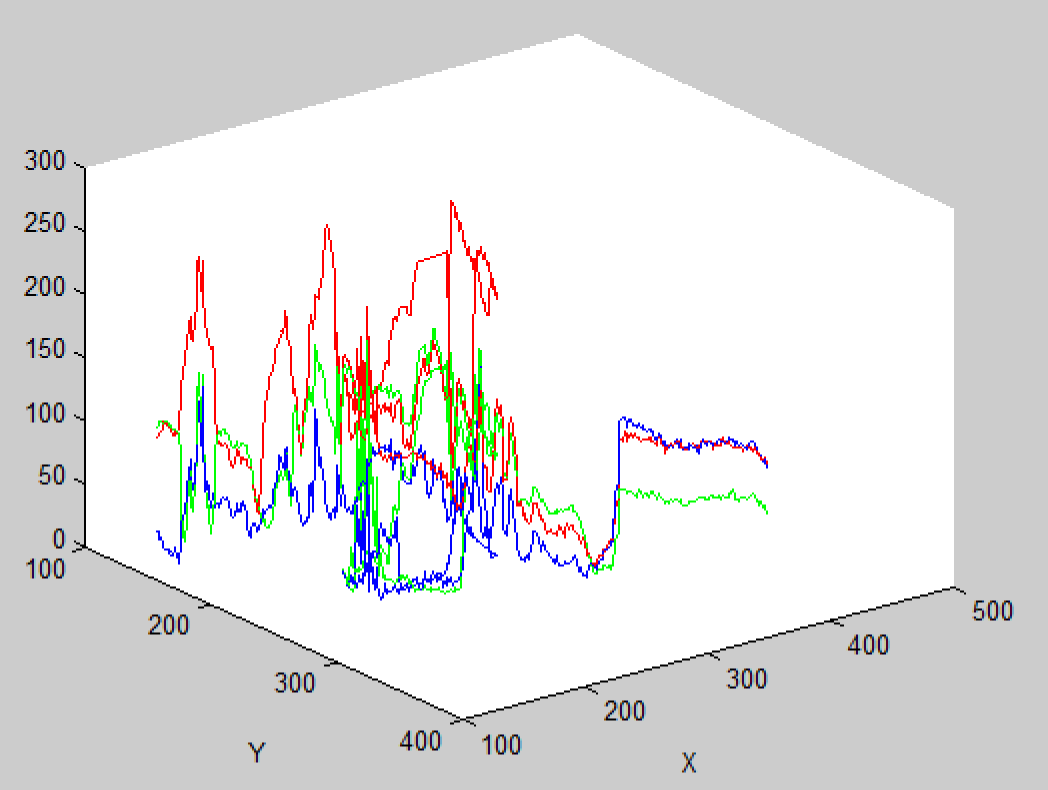
```
ans =
    24    24    24
    86    86    86
    13    13    13
     1     1     1
     6     6     6
```

```
    1     1     1

  255   255   255

    8     8     8

    7     7     7

>> tire=imread('peppers.png');

figure

impixel(tire)
```

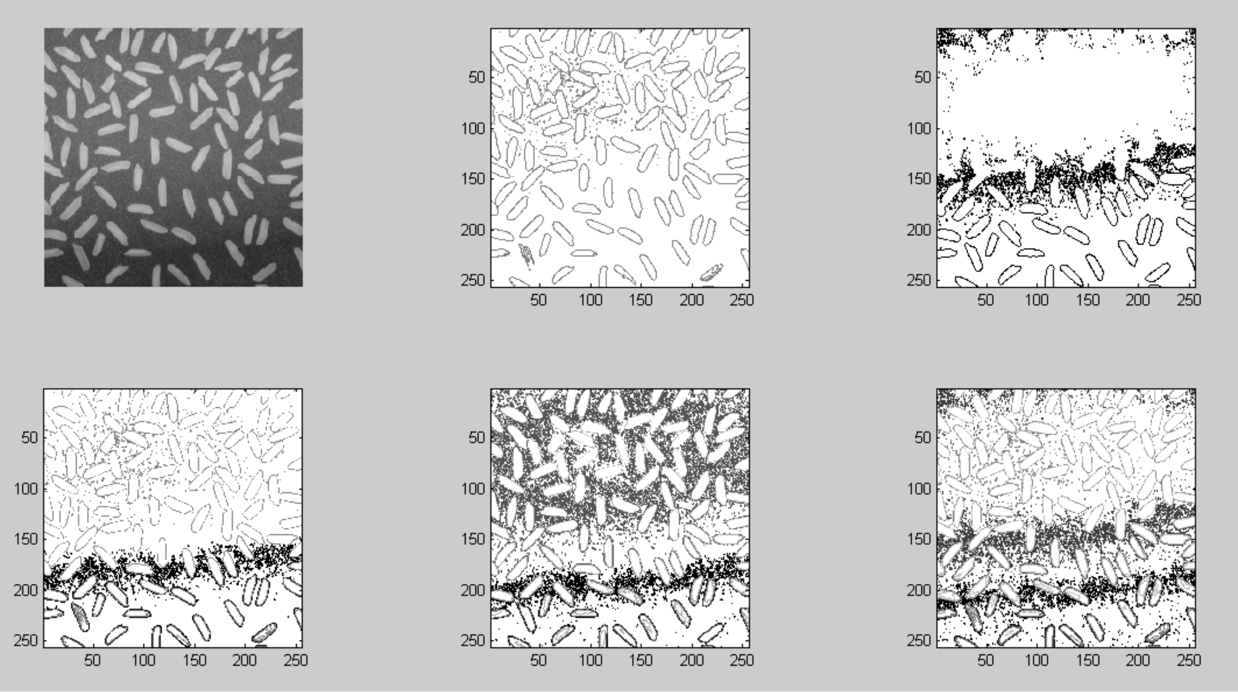


```
ans =

  131   127     8

  214    51    51

  249    66    58

   83    49     7

   50    62    32

  138   140     5
```

```
  195   182    90
   71    40    65
   67    36    67
   75    47    76
   66    35    62
  254   254   254
   70    40    71
>>imshow('peppers.png')
Improfile
```

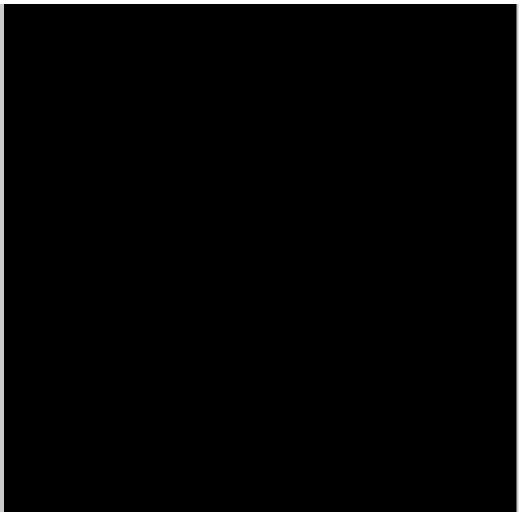


```
>> I=imread('rice.png')
subplot(2,3,1)
imshow(I)
subplot(2,3,2)
imcontour(I,1)
subplot(2,3,3)
imcontour(I,2)
subplot(2,3,4)
imcontour(I,3)
subplot(2,3,5)
imcontour(I,4)
subplot(2,3,6)
imcontour(I,5)
```

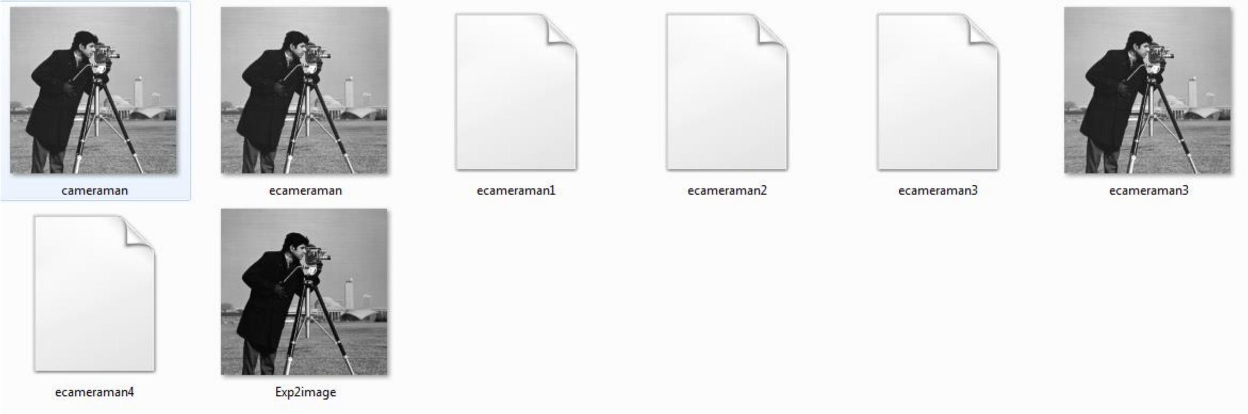


```
>> cameraman=imread('cameraman.tif');
b=logical(cameraman);
disp('B is logical or not : ');
islogical(b)
B is logical or not :
ans =
     1
>>whos cameraman
>> whos cameraman
  Name            Size            Bytes  Class    Attributes
  cameraman      256x256           65536  uint8
imshow(cameraman)
>> g=[-0.5 0.5;0.75 1.5]
y=im2uint8(g)
s=im2uint16(g)
g =
   -0.5000    0.5000
    0.7500    1.5000
y =
```

```
    0    128
  191    255
s =
     0          32768
  49151         65535
c=double(cameraman)/65535;
imshow(c)
```

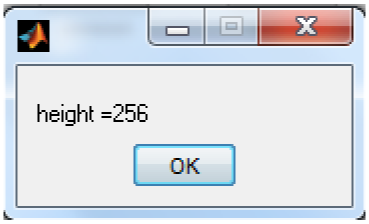


```
>> f=imread('cameraman.tif');
subplot(1,2,1);
imshow(f)
f=f+20;
imwrite(f,'E:\ImageProcessing\ecameraman.tif');
imwrite(f,'E:\ImageProcessing\ecameraman1','jpeg');
imwrite(f,'E:\ImageProcessing\ecameraman2','png');
imwrite(f,'E:\ImageProcessing\ecameraman3','gif');
imwrite(f,'E:\ImageProcessing\ecameraman4','bmp');
```



E:\ImageProcessing

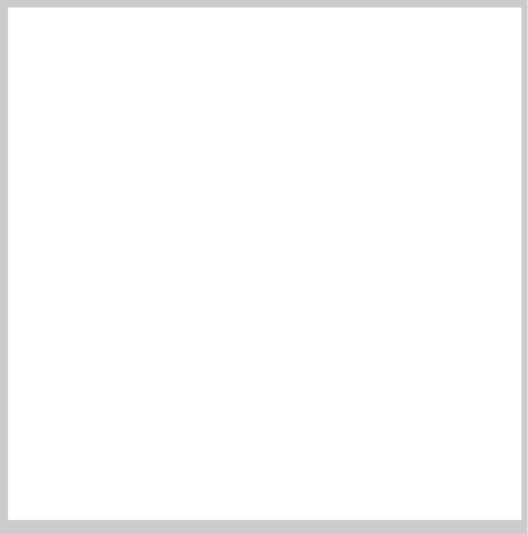
```
>> h=imfinfo('cameraman.tif');
g=sprintf('height =%d ',h.Height)
msgbox(g);
g =
height =256
```



height =256

```
>> figure
```

```
log=logical(f);
imshow(log);
```



```
>> figure
log=logical(f);
imshow(log);
>> d=im2double(f);
title('Tool Box Functions')
subplot(1,3,1)
imshow(d)
xlabel('double')
>> figure
d=im2double(f);
title('Tool Box Functions')
subplot(1,3,1)
imshow(d)
xlabel('double')
g=im2uint16(d);
subplot(1,3,2)
imshow(g)
xlabel('uint16')
l=im2single(d);
subplot(1,3,3)
```

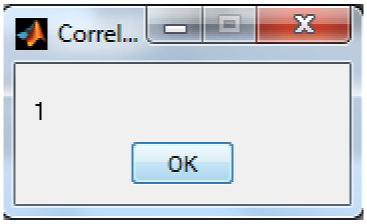
```
imshow(l)
xlabel('single')
whos
  Name                Size                Bytes  Class     Attributes


  I                   256x256             65536  uint8
…………………………………………………………………
  std                 1x1                 8  double
```

EXERCISE QUESTIONS

Q1 : Read any image of class uint8 from MATLAB built in images. Convert it to double using the command given above and show it using imshow. Do you see any difference in the original image and the image converted to double? Why there is a difference?

```
f=imread('cameraman.tif');
whos f
subplot(1,3,1)
imshow(f)
xlabel('uint8')
d=double(f)
d(1:10)
f(1:10)
subplot(1,3,2)
imshow(d)
```

```
xlabel('double(f)')
d=d/255;
subplot(1,3,3)
imshow(d)
xlabel('double(f)/255')
d(1:10)
f(1:10)
msgbox(num2str(corr2(f,d)),'Correlation');
```

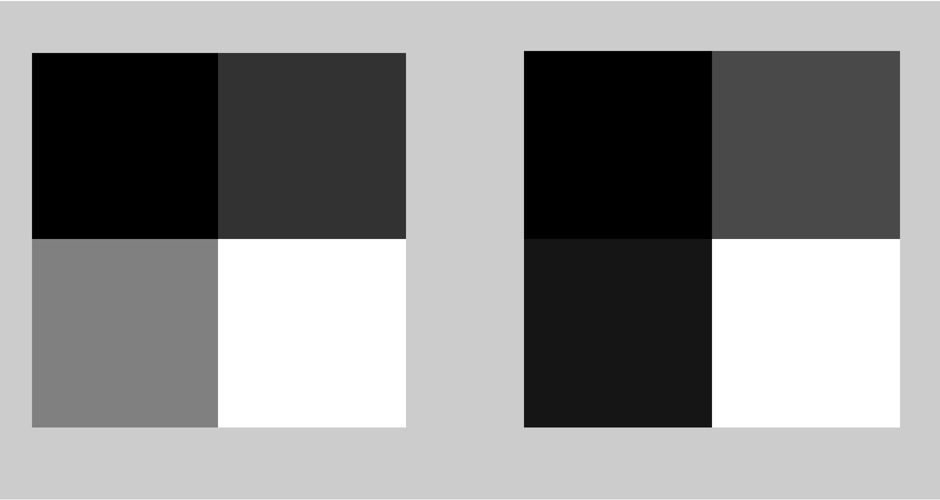


1

Q2 : Convert the above matrix h into type double using im2double and using mat2gray. Do you see any difference in the output? How does the function mat2gray scale the values in between maximum and minimum values of matrix h?

```
h =
    0    50
  128   255
g =
         0    0.1961
    0.5020    1.0000
c =
    0   40
```

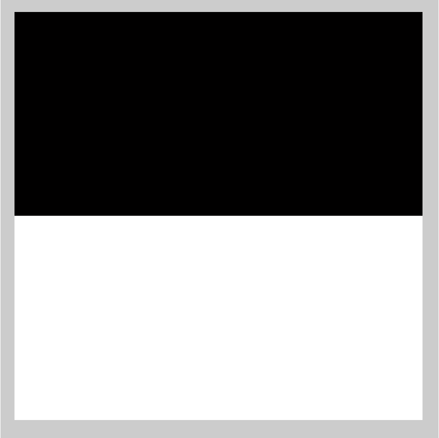
```
   12  140
q =
        0    0.2857
    0.0857    1.0000
  Name                    Size              Bytes  Class     Attributes
  I                       256x256            65536  uint8
  ………………………………………………………….
  y                       2x2                   4  uint8
```



Q3 : Let us suppose we wish to convert the following small, double image (h = [1 2 ; 3 4];) to binary such that values 1 and 2 become 0 and the other two values become 1. Use atleast two methods to convert it into binary. (Hint: convert the values of f between 0 and 1 before converting it to binary). After converting it into logical (binary) image convert it to uint8. Now convert this uint8 image to double without changing the values.

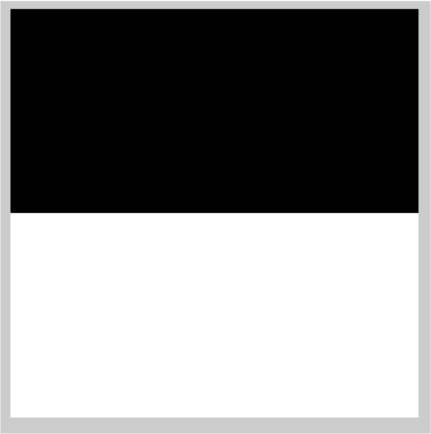
```
>> h=[1 2;3 4]

h =

     1     2
     3     4
>> l=h>2

l =
```

```
     0     0
     1     1
>> imshow(l,'InitialMagnification',2000)
```



```
>> h

h =

     1     2
     3     4

>> v=uint8(h)

v =

    1    2
    3    4
>> fun2=im2bw(v,0.0098)

fun2 =

     0     0
```

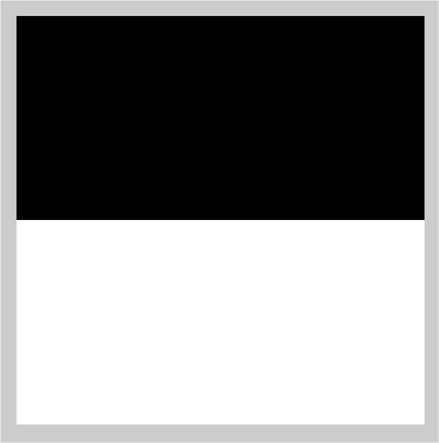
```
     1     1
>> fun2=im2bw(v,2.5/255)

fun2 =

     0     0
     1     1
>> imshow(fun2,'InitialMagnification',2000)
```



```
>> h = [ 1 2 ; 3 4]

h =

     1     2
     3     4
>> g = im2bw(h,0.6) %fails

g =

     1     1
     1     1

>> g = im2bw(h,0.1) % fails
```

```
g =


     1     1
     1     1

>> g = im2bw(h,0) %fails

g =


     1     1
     1     1

>> s = im2double(h) %fails

s =


     1     2
     3     4

>> s = mat2gray(h) %agreed

s =


         0    0.3333
    0.6667    1.0000
>> g = im2bw(s,0.4)

g =


     0     0
     1     1
```

```
>> imshow(g,'InitialMagnification',2000)
```



Experiment#3: To get familiar with matrix manipulation commands. To use matrix manipulation commands for rotating and compressing images.

1. **Experiment Text**
    i. **Array Indexing**
        Vector Indexing
        Matrix Indexing
2. **Lab Exercise**
3. **Exercise Questions**

EXPERIMENT TEXT

1 Array Indexing

1.1 Vector Indexing

An array of dimension 1xN is called a row vector. The elements of such a vector can be accessed using a single index value (also called a subscript). Thus, v(1) is the first element of vector v, v(2) is its second element, and so forth. Vectors can be formed in MATLAB by enclosing the elements, separated by spaces or commas, within square brackets. For example,

```
>> v = [1 3 5 7 9]
```

will generate a vector v and v(2) gives the second element of vector v. A row vector is converted to a column vector (and vice versa) using the transpose operator ('). e.g. the following command will store transpose of v in w.

```
>> w = v’
```

```
>> v=[1 3 5 7 9]

v =

    1     3     5     7     9

>> v(1)

ans =

    1
>> v(5)

ans =

    9

>> w=v'
```

```
w =

     1
     3
     5
     7
     9
```

To access blocks of elements, we use MATLAB's colon notation. For example, to access the second through four elements of V we write v(2:4). To access all the elements from third to last we write V(3:END).

```
>> v(2:4)

ans =

     3     5     7
>> v(2:end)

ans =

     3     5     7     9
>> v([1 4 5])

ans =

     1     7     9

>> v([1 4 9]) % Error
Index exceeds matrix dimensions.
```

1.2 Indexing Matrices

Matrices can be represented conveniently in MATLAB as a sequence of row vectors enclosed by square brackets and separated by semicolons. For example

```
>> A = [1 2 3; 4 5 6; 7 8 9]
```

Will create matrix A

```
>> a= [1 2 3;4 5 6;7 8 9]

a =

     1     2     3
     4     5     6
     7     8     9
```

To extract the element in the second row, third column of matrix A, we write A(2,3).

```
>> a= [1 2 3;4 5 6;7 8 9]

a =

     1     2     3
     4     5     6
     7     8     9

>> a(2,3)

ans =

     6
```

A submatrix of A can be extracted by specifying a vector of values for both the row and the column indices. For example, the following statement extracts the submatrix of A containing rows 1 and 2 and columns 1, 2, and 3:

```
>> T2 = A([1 2], [1 2 3])
```

```
>> T1=a([1 2],[1 2 3])

T1 =

     1     2     3
     4     5     6
```

The preceding statement could be written also as T2 = A(1:2, 1:3).

```
>> T2=a (1:2,1:3)

T2 =

     1     2     3
     4     5     6
```

A colon in the row index position is shorthand notation for selecting all rows. e.g. the following command will select third column and all rows.

```
>> T2 = A (:,3)
```

```
>> T2=a (:,3)

T2 =
```

```
     3
     6
     9
```

Similarly this statement will extract the second row.

```
>> T2 = A (2,:)
```

```
>> T3=a(2,:)

T3 =

     4     5     6
```

The keyword END, when it appears in the row index position, is shorthand notation for the last row. When END appears in the column index position, it indicates the last column. For example, the following statement finds the element in the last row and last column of A:

```
>> T2 = A (END,END)
>> t=a (end,end)

t =

     9
```

and the following statement gives the last row as first row and first row as second row.

```
>> T2 = A([END 1],:)
```

```
>> T2=a([end 1],:)

T2 =

     7     8     9
     1     2     3

>> T4=a(end,:)

T4 =

     7     8     9
>> T5=a(:,end)

T5 =

     3
     6
```

```
     9
>> T6=a([end 1],:)

T6 =

     7     8     9
     1     2     3
>> T6=a([end 1],:)

T6 =

     7     8     9
     1     2     3
>> T8=a([end 1],1)

T8 =

     7
     1
>> T9=a(:,[1 end])

T9 =

     1     3
     4     6
     7     9
>> T9=a(:,[2 end])

T9 =

     2     3
     5     6
     8     9

>> T10=a(:,[end 2])

T10 =

     3     2
     6     5
     9     8
>> a

a =

     1     2     3
     4     5     6
     7     8     9
```

```
>> rot90(a)

ans =

     3     6     9
     2     5     8
     1     4     7

>> rot90(a,2)

ans =

     9     8     7
     6     5     4
     3     2     1

>> rot90(a,4)

ans =

     1     2     3
     4     5     6
     7     8     9
```

An image can be cropped by selecting only a few rows and columns. e.g. the following command S = F(50:200,50:200) can crop the cameraman.tif image. How should we select the row and column number for cropping? (Hint: See the total number of rows and columns of the image).

```
>> I=imread('cameraman.tif');
c =
174  172  176  175  177  180  178  179  180  178  179  176  175  179  174  176  176  178
.
.
.
139 139  149  142  135  125  128
```

```
>> I=imread('cameraman.tif');
J=imcrop(I,[[44.5 35.5 167 159]]);
imshow(J)
```

```
c=I(50:200,50:200)
imshow(c)
```

```
>> a
a =
     1     2     3
     4     5     6
     7     8     9
>> B=flipud(a)
B =
     7     8     9
     4     5     6
     1     2     3
>> a
a =
     1     2     3
     4     5     6
     7     8     9
>> C=flipdim(a,1)
C =
     7     8     9
     4     5     6
     1     2     3
>> a
a =
     1     2     3
```

```
     4     5     6
     7     8     9
>> D=flipdim(a,2)
D =
     3     2     1
     6     5     4
     9     8     7
```

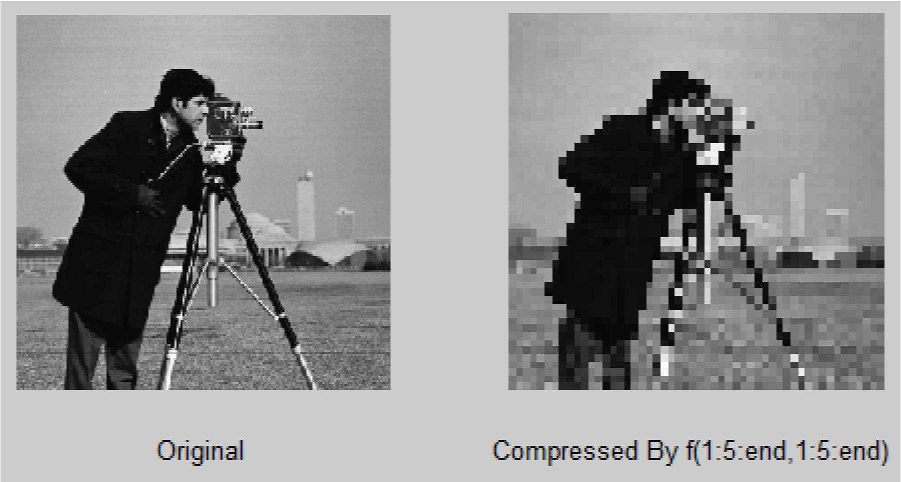
An image can be compressed by skipping some of the rows and columns in the original image (It will degrade the original image though). The following command makes the size of an image 1/4 by skipping every second row and column.

```
>> fs = f(1:2:end, 1:2:end);
```

```
>> f=imread('cameraman.tif');
comp=f(1:2:end,1:2:end);
subplot(1,2,1)
imshow(f)
xlabel('Original')
subplot(1,2,2)
imshow(comp)
xlabel('Compressed By f(1:2:end,1:2:end) ')
```

```
>> f=imread('cameraman.tif');
comp=f(1:5:end,1:5:end);
```

```
subplot(1,2,1)
imshow(f)
xlabel('Original')
subplot(1,2,2)
imshow(comp)
xlabel('Compressed By f(1:5:end,1:5:end) ')
```



LAB EXERCISE

CROPPING

```
>> I=imread('cameraman.tif');
J=imcrop(I,[[44.5 35.5 167 159]]);
imshow(J)
```

```
c=I(50:200,50:200)
imshow(c)
```

QUESTIONS:

Q1: What do the command v(1:2:end) and v(end:-2:1) do? Now use these commands to create a vector w which have same elements as v but in reverse order.

```
>> v= [1 3 5 7 9]
v =
    1     3     5     7     9
>> v (1:1:end)
ans =
    1     3     5     7     9
>> v (1:2:end)
ans =
    1     5     9
>> v (1:3:end)
ans =
    1     7
>> v (1:4:end)
ans =
    1     9
>> v (1:5:end)
ans =
    1
>> v (1:6:end)
ans =
    1
>> v(1:-1:end)
```

```
ans =

  Empty matrix: 1-by-0

>> v(1:-4:end)

ans =

  Empty matrix: 1-by-0

>> v(end:1:1)

ans =

  Empty matrix: 1-by-0

>> v(end:-1:1)

ans =

     9     7     5     3     1

>> v(end:-2:1)

ans =

     9     5     1

>> v(end:-3:1)

ans =

     9     3

>> v(end:-4:1)

ans =

     9     1

>> v(end:-5:1)

ans =

     9
```

b)

```
>> w=v(end:-1:1)

w =

     9     7     5     3     1
```

Q2 : Create a 5x5 matrix A with any elements. Now make another matrix B from A such that the first and last rows of A and B are exchanged, similarly the second and second last rows are also exchanged. (Hint use END to access last row and then move backwards).

```
>> v=sort(magic(5))

v =

     4     5     1     2     3
```

```
    10     6     7     8     9
    11    12    13    14    15
    17    18    19    20    16
    23    24    25    21    22
>> w=v([end:-1:1],:)
w =
    23    24    25    21    22
    17    18    19    20    16
    11    12    13    14    15
    10     6     7     8     9
     4     5     1     2     3
v =

     4     5     1     2     3
    10     6     7     8     9
    11    12    13    14    15
    17    18    19    20    16
    23    24    25    21    22

>> w1=v(:,[end :-1:1])

w1 =

     3     2     1     5     4
     9     8     7     6    10
    15    14    13    12    11
    16    20    19    18    17
    22    21    25    24    23

>> flipud(v) %flip up to down
ans =
    23    24    25    21    22
```

```
    17    18    19    20    16
    11    12    13    14    15
    10     6     7     8     9
     4     5     1     2     3
>> fliplr(v) %flip right left
ans =
     3     2     1     5     4
     9     8     7     6    10
    15    14    13    12    11
    16    20    19    18    17
    22    21    25    24    23
>> flipdim(v,1)
ans =
    23    24    25    21    22
    17    18    19    20    16
    11    12    13    14    15
    10     6     7     8     9
     4     5     1     2     3
>> flipdim(v,2)
ans =
     3     2     1     5     4
     9     8     7     6    10
    15    14    13    12    11
    16    20    19    18    17
    22    21    25    24    23
```

Q3 : Read the cameraman.tif image from MATLAB (this image is in the form of a matrix) and use the same procedure used in **Q2** to exchange the rows. Now show both images (original and exchanged on the same graph. What does the exchanged image show?

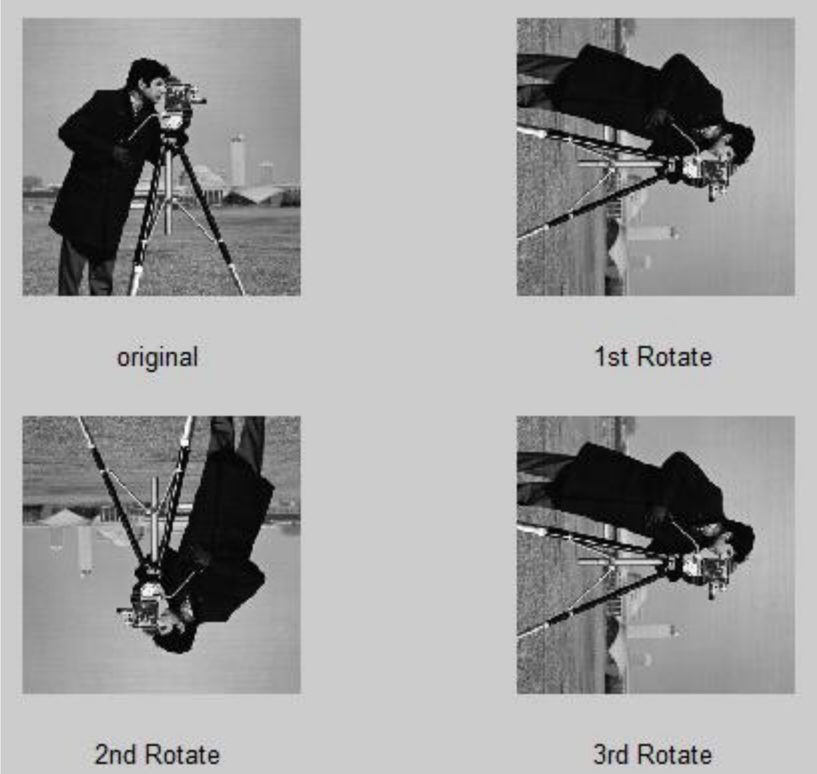
```
>> a=imread('cameraman.tif');
b=a([end:-1:1],:);
c=a(:,[end:-1:1]);
subplot(1,3,1)
```

```
imshow(a)
xlabel('original')
subplot(1,3,2)
imshow(b)
xlabel('Exchanged Row wise')
subplot(1,3,3)
imshow(c)
xlabel('Exchanged Column wise')
```

Q4 : Use subplot to show four images on a figure. The first image should be the actual cameraman.tif image. The second image should be the 90 degree clockwise (CW) rotation of first one. The third image should be the 90 degree CW rotation of second one and the last image should be 90 degree CW rotation of third image.

```
>> y=imread('cameraman.tif');
title('Using Matrix Indexing')
subplot(2,2,1)
imshow(y)
xlabel('original')
rot=y(end:-1:1,:);
rot1=rot';
subplot(2,2,2)
imshow(rot1);
xlabel('1st Rotate')
rot=rot1(end:-1:1,:);
rot2=rot';
subplot(2,2,3)
```

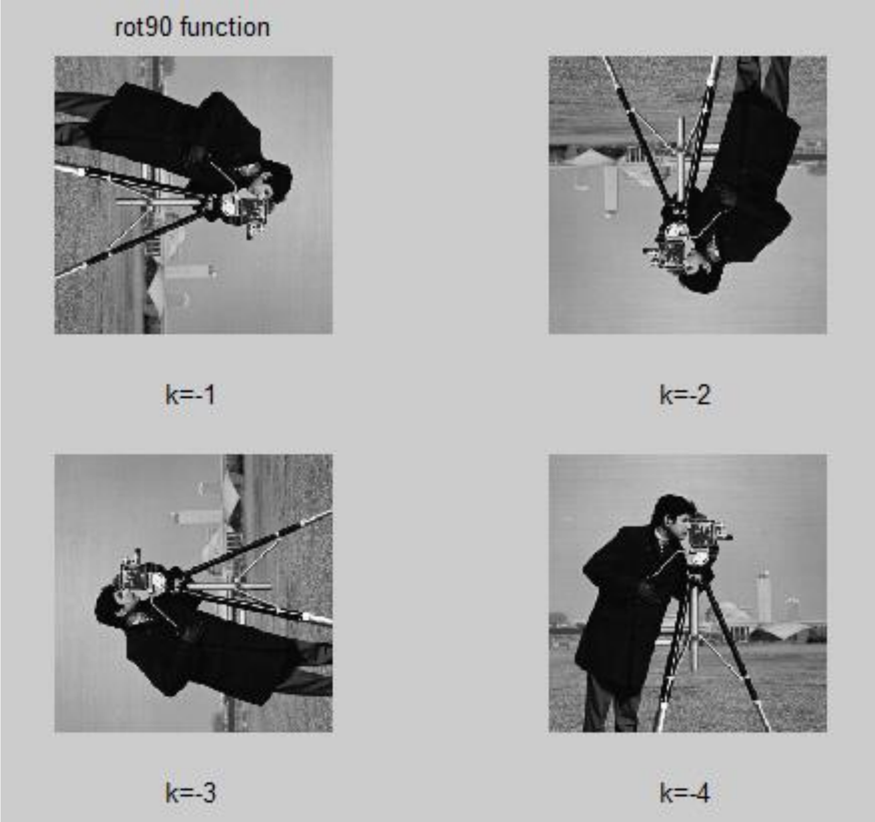
```
imshow(rot2);
xlabel('2nd Rotate')
rot=rot2(end:-1:1,:);
rot3=rot';
subplot(2,2,4)
imshow(rot1);
xlabel('3rd Rotate')
```



Using rot90 function

```
clc
a=rot90(y,-1);
b=rot90(y,-2);
c=rot90(y,-3);
d=rot90(y,-4);
figure('name','rot90 function');
subplot(2,2,1);
imshow(a);
title('rot90 function');
xlabel('k=1');
```

```
subplot(2,2,2);
imshow(b);
xlabel('k=2');
subplot(2,2,3);
imshow(c);
xlabel('k=3');
subplot(2,2,4);
imshow(d);
xlabel('k=4');
```



Q5: Compare the size of `FS` with `F`. Plot both the images using subplot. Do you see much degradation? Now create two more images with increments of 5 and 10 instead of 2 and plot all four images (Original, with increment of 2, 5, 10) on same figure using subplot.

```
clc
y=imread('cameraman.tif');
title('Using Matrix Indexing')
```

```
subplot(2,2,1)
imshow(y);
xlabel('original')
ylabel('No Compression');
fs2=y(1:2:end,1:2:end);
subplot(2,2,2)
imshow(fs2)
x=sprintf(num2str(size(fs2)));
xlabel(x)
ylabel('y(1:2:end,1:2:end)');
fs5=y(1:5:end,1:5:end);
subplot(2,2,3)
imshow(fs5)
x=sprintf(num2str(size(fs5)));
xlabel(x)
ylabel('y(1:5:end,1:5:end)')
fs10=y(1:10:end,1:10:end);
subplot(2,2,4)
imshow(fs10)
x=sprintf(num2str(size(fs10)));
xlabel(x)
ylabel('y(1:10:end,1:10:end)')
```


No Compression
original
y(1:2:end,1:2:end)
128 128
y(1:5:end,1:5:end)
52 52
y(1:10:end,1:10:end)
26 26

Experiment#4: To get familiar with basic intensity transformation functions and to get familiar with M-file programming.

1. **Experiment Text**
   - i. **Intensity Transformation**
   - ii. **M-Files**
2. **Lab Exercise**
3. **Exercise Questions**

1 Intensity Transformation

1.1 Background

The term spatial domain refers to the image plane and any spatial operation refers to directly manipulating the pixels of an image. There are two important categories of spatial domain processing, intensity transformation and spatial filtering. The spatial processes are denoted by the expression

$$g(x,y) = T\left[f(x,y)\right] \qquad (1)$$

where f (x, y) is the input image, g(x, y) is the output image and T is an operator applied on f over a specific neighborhood defined around (x, y). If the neighborhood is a matrix of size 1x1 then it is called intensity transformation otherwise spatial filtering.

1.2 Intensity Transformation

The intensity transformation can be denoted as

$$s = T(r) \qquad (2)$$

where r is the intensity of the input image, s is the intensity of the output image and T is the operation performed on the input image.

1.2.1 Power Law transformation

Function imadjust is the basic tool for transforming grey scale images. It has the syntax

```
>> g = imadjust(f, [low-in high-in], [low-out high-out], gamma)
```

This function maps the intensity values of image f to new values in g such that the values in between low-in and high-in map to values low-out and high-out. Values below low-in map to low-out and those above high-in map to high-out. The value of gamma decides how would the intensities be mapped (Refer to class lectures for more details about gamma).

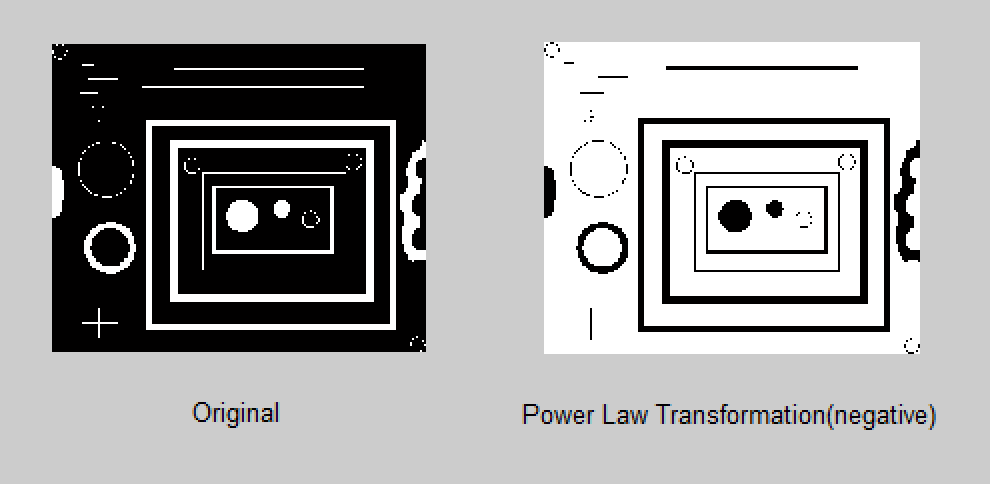
The input image can be of class uint8, uint16 or double and the output image will have the same class as input. All the inputs to imadjust (except f and gamma) are specified between 0 and 1, regardless of class of f . If class of f is uint8, imadjust multiplies the supplied values with 255 to get the actual values to use. Using empty matrix ([ ]) for [low-in high-in] and low-out high-out will result in default values [0 1]. If gamma is omitted then it is gets default value of 1. The following command can be used to take negative of an image.

```
>> g = imadjust(f, [0 1] [1 0], 1)
```

Now read image blobs.png and take its negative using imadjust (Remember to change class of image before applying imadjust).

```
clc
i=imread('blobs.png');
whos i
f=im2uint8(i);
whos f
powerlaw=imadjust(f,[0 1],[1 0],1);
figure('Name','Power Law Transformation')
subplot(1,2,1)
imshow(f);
xlabel('Original')
subplot(1,2,2)
imshow(powerlaw);
xlabel('Power Law Transformation(negative)')
```

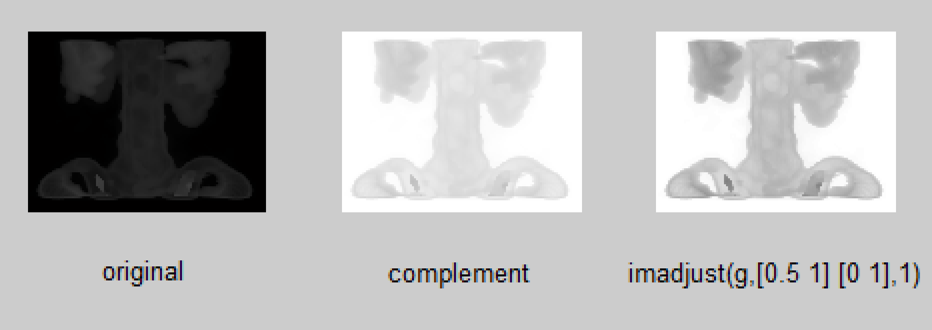
```
  Name        Size             Bytes  Class      Attributes

  i         272x329           89488  logical

  Name        Size             Bytes  Class    Attributes

  f         272x329           89488  uint8
```



Now read another image spine.tif and plot both image and its negative on same figure. Which image shows the object more clear? Why?

```
clc
i=imread('spine.tif');
whos i
g= imadjust(i, [0 1], [1 0] ,1);
h=imadjust(g, [0.5 1], [0 1] ,1);
subplot(1,3,1)
imshow(i)
xlabel('original')
subplot(1,3,2)
imshow(g)
xlabel('complement')
subplot(1,3,3)
imshow(h)
x=sprintf('imadjust(g,[0.5 1] [0 1],1)')
xlabel(x)
```

```
Name        Size              Bytes   Class     Attributes

i         367x490              179830  uint8
```

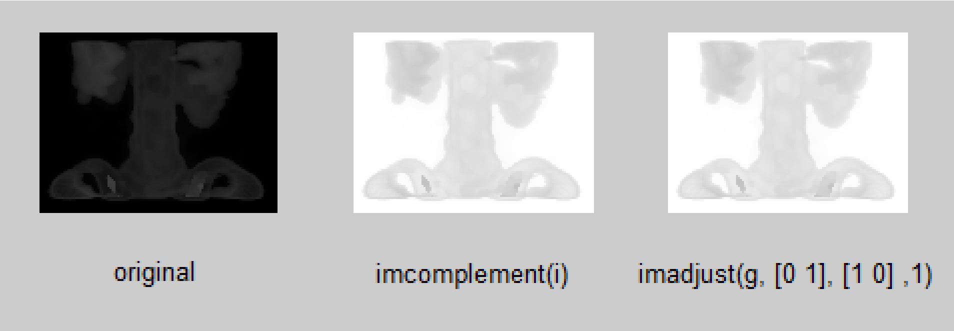


The negative of an image can also be taken using the command

```
>> g = imcomplement(f)
```

Now take complement of spine.tif and save it in g. Now apply the command,

```
clc
i=imread('spine.tif');
g=imcomplement(i);
h=imadjust(i, [0 1], [1 0] ,1);
subplot(1,3,1)
imshow(i)
xlabel('original')
subplot(1,3,2)
imshow(g)
xlabel('imcomplement(i)')
subplot(1,3,3)
imshow(h)
x=sprintf('imadjust(g, [0 1], [1 0] ,1)')
xlabel(x)
```



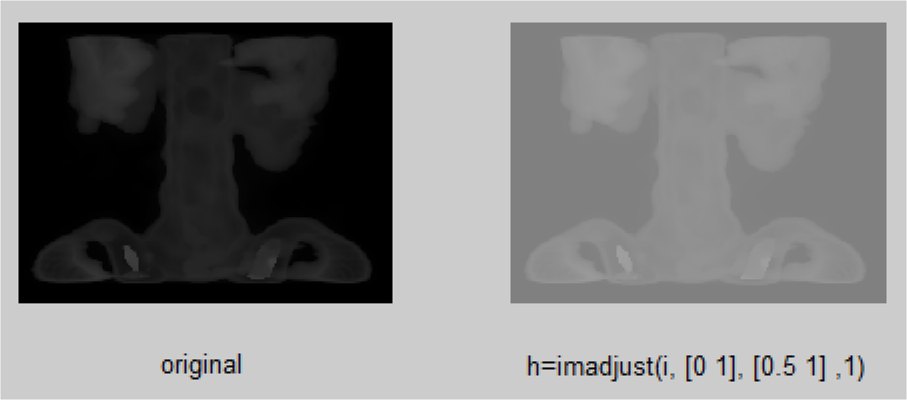
```
>> h = imadjust(g, [0.5 1] [0 1], 1)
clc
i=imread('spine.tif');
whos i
h=imadjust(i, [0.5 1], [0 1] ,1);
subplot(1,2,1)
imshow(i)
xlabel('original')
subplot(1,2,2)
imshow(h)
xlabel(' h=imadjust(i, [0.5 1], [0 1] ,1) ')
```

Can you see that your object has become more prominent now? This command expands the grey scale region between 0.5 and 1 to 0 and 1. Now what if you change the positions of scaling, e.g. if you execute the command,

```
>> h = imadjust(g, [0 1] [0.5 1], 1)
```

What do you see now? Is the new image washed out? Why?

```
clc
i=imread('spine.tif');
whos i
h=imadjust(i, [0 1], [0.5 1] ,1);
subplot(1,2,1)
imshow(i)
xlabel('original')
subplot(1,2,2)
imshow(h)
xlabel(' h=imadjust(i, [0 1], [0.5 1] ,1) ')
```



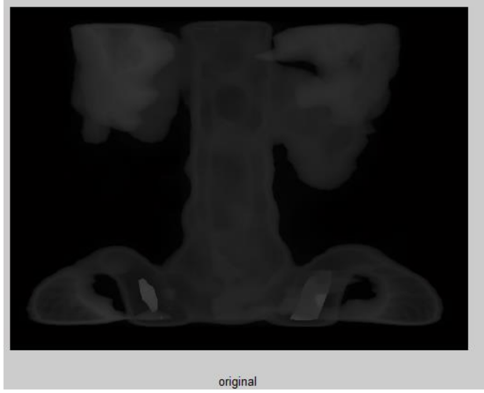
The image can be also be made more brighter or darker by changing the value of *gamma*. A value of *gammma* greater than one compresses the lower end of intensities (makes the image more darker) while a value less than one makes the stretches the lower end of intensities (makes the image more brighter).

```
clc
i=imread('spine.tif');
whos i
h=imadjust(i, [0 1], [0.5 1] ,0.5);
```

```
subplot(1,2,1)
imshow(i)
xlabel('original')
subplot(1,2,2)
imshow(h)
xlabel(' h=imadjust(i, [0 1], [0.5 1] ,0.5) ')
```

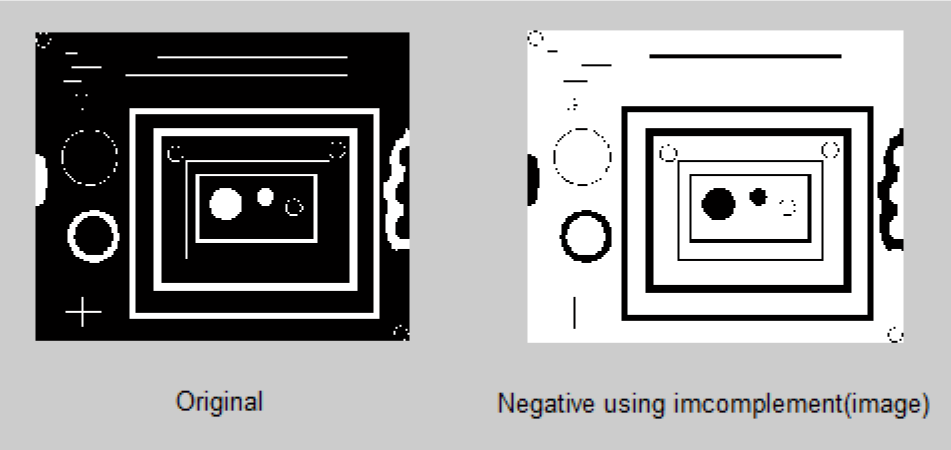
```
clc
i=imread('spine.tif');
whos i

h1=imadjust(i, [0 1], [0.5 1] ,0.5);
h2=imadjust(i, [0 1], [0.5 1] ,0.9);
h3=imadjust(i, [0 1], [0.5 1] ,1.5);
h4=imadjust(i, [0 1], [0.5 1] ,1.9);

imshow(i)
xlabel('original')
figure
subplot(2,2,1)
imshow(h1)
xlabel(' h1=imadjust(i, [0 1], [0.5 1] ,0.5) ')

subplot(2,2,2)
imshow(h2)
```

```
xlabel(' h2=imadjust(i, [0 1], [0.5 1] ,0.9) ')


subplot(2,2,3)
imshow(h3)
xlabel(' h3=imadjust(i, [0 1], [0.5 1] ,1.5) ')


subplot(2,2,4)
imshow(h4)
xlabel(' h4=imadjust(i, [0 1], [0.5 1] ,1.9) ')
```

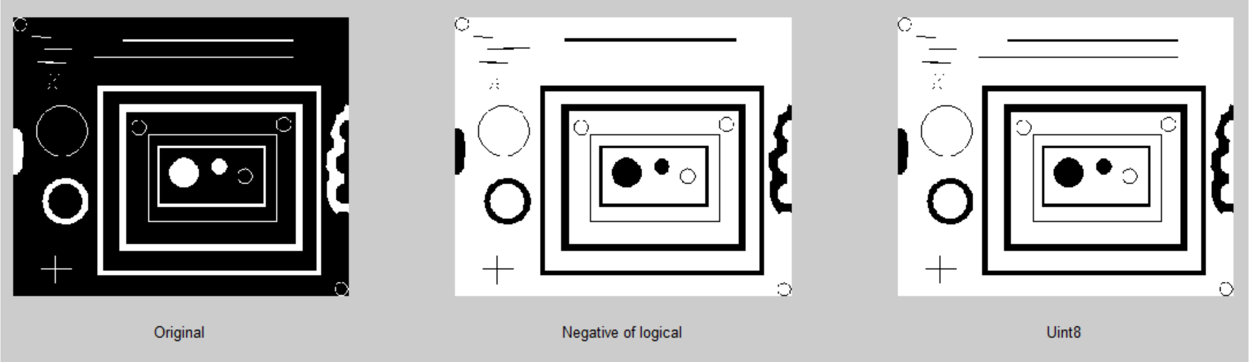


```
clc
```

```
i=imread('blobs.png');
neg=imcomplement(i);
subplot(1,2,1)
imshow(i);
xlabel('Original')
subplot(1,2,2)
imshow(neg);
xlabel('Negative using imcomplement(image)')
```



```
clc
i=imread('blobs.png');
f=im2uint8(i);
neg=imcomplement(i);
neg_con=imcomplement(f);
subplot(1,3,1)
imshow(i);
xlabel('Original')
subplot(1,3,2)
imshow(neg);
xlabel('Negative of logical')
subplot(1,3,3)
imshow(neg_con)
xlabel('Uint8')
```



1.2.2 Logarithmic Transformations

Logarithmic transformations are implemented using the expression

```
>> g = c * log (1 + double (f))
```

The shape of the logarithmic curve is similar to the $gamma < 1$ curve, however, the shape of $gamma$ curve is variable while the shape of logarithmic curve is fixed. While dealing with logarithmic transformation, it is always desirable to bring the resulting compressed values back to the original range. For 8 bits, the easiest way to do in MATLAB is to use this expression

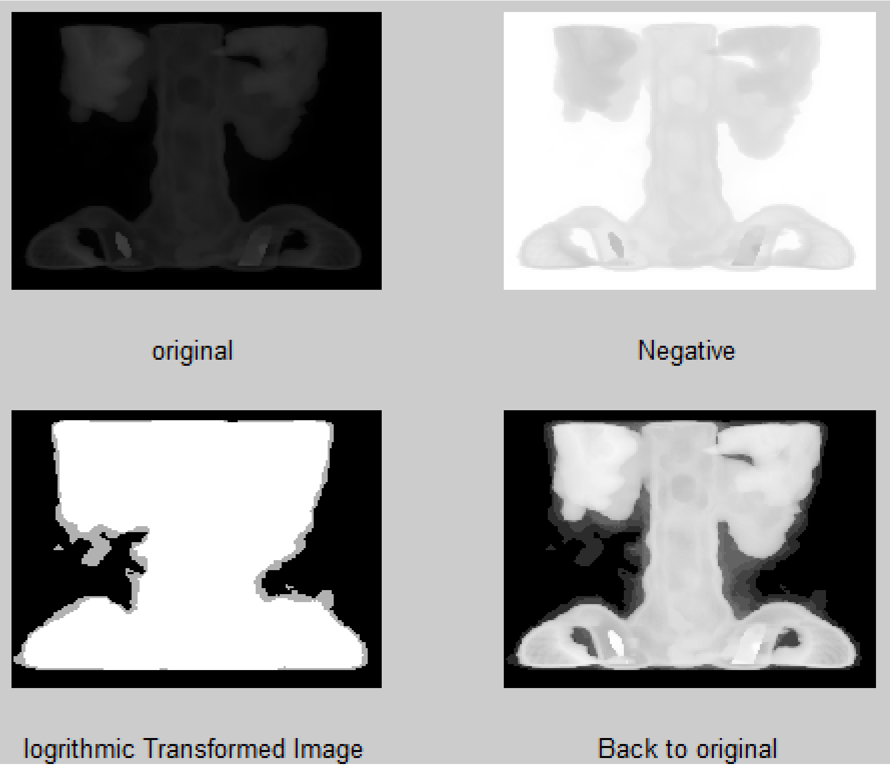
```
>> gs = im2uint8(mat2gray(g))
```

Where g is the logarithmic image. The function `mat2gray` converts the image to range [0,1] while function `im2uint8` converts it to range [0,255].

```
clc
clear
i=imread('spine.tif');
g=imcomplement(i);
c=input('Enter C = ');
lg=c*log(1+double(i));
subplot(2,2,1)
imshow(i)
xlabel('original')
subplot(2,2,2)
imshow(g)
xlabel('Negative')
subplot(2,2,3)
imshow(lg)
```

```
xlabel('logrithmic Transformed Image')
gs=im2uint8(mat2gray(lg));
subplot(2,2,4)
imshow(gs)
xlabel('Back to original')
whos
```

```
Enter C = 1
  Name          Size                    Bytes          Class     Attributes

  c             1x1                     8              double
  g             367x490                 179830         uint8
  gs            367x490                 179830         uint8
  i             367x490                 179830         uint8
  lg            367x490                 1438640        double
```



2 M-Files

There are two types of M-files in MATLAB, script files and function files. Script files simply execute a series of MATLAB statements while function files can accept arguments and can produce one or more outputs. M-files can be created by right clicking on current folder section and selecting 'new file' and then selecting function or script file. As mentioned earlier script files will have a sequence of commands which can be execute by either typing the name of script file at command prompt or by using the 'Run' button at top of M-file.

Function files have a function definition line at the start of M -file which has the following syntax

```
Function [a, b] = function_name(c, d)
```

Where c and d are the inputs to the file while a & b are the output of the file. The function name and filename should always be same.

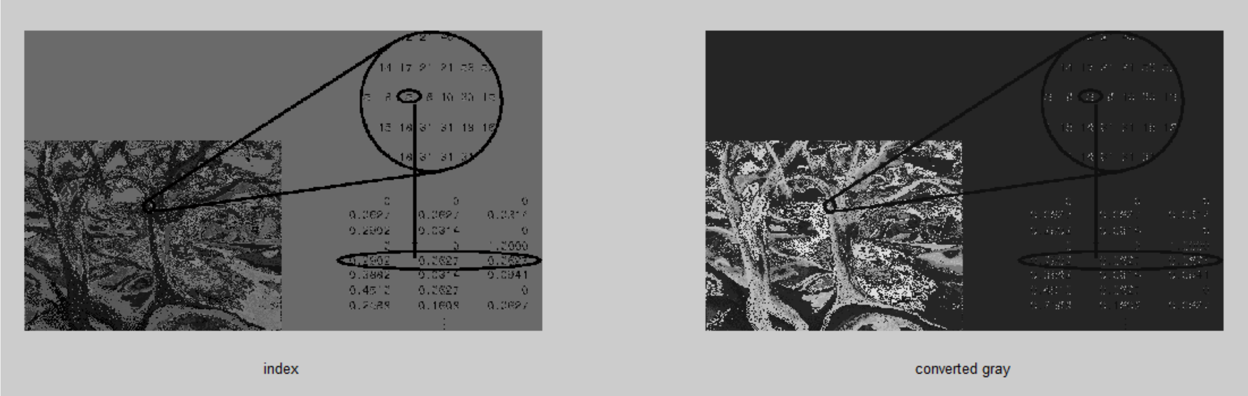
MATLAB EXERCISE

Color Conversion

Index to Gray

Y=ind2gray(x,map),   'map' is gray scale map

```
clc
clear
x=imread('chimage9.tif');
imfinfo('chimage9.tif')
y=ind2gray(x,colormap);
imwrite(y,'chimage9y.tif')
imfinfo('chimage9y.tif')
subplot(1,2,1)
imshow(x)
xlabel('index')
subplot(1,2,2)
imshow(y);
xlabel('converted gray')
```



index

converted gray

Gray to Index

[y map]=gray2index(x,level)

```
clc
clear
x=imread('cameraman.tif');
imfinfo('cameraman.tif')
y=gray2ind(x,128);
imwrite(y,'cameramany.tif')
imfinfo('cameramany.tif')
subplot(1,2,1)
imshow(x)
xlabel('Gray')
subplot(1,2,2)
imshow(y);
xlabel('converted indexed')
```

RGB to graycale

Y=rgb2gray(x)

```
clc
clear
x=imread('peppers.png');
imfinfo('peppers.png')
y=rgb2gray(x);
imwrite(y,'peppersy.png')
imfinfo('peppers.png')
subplot(1,2,1)
imshow(x)
xlabel('RGB')
subplot(1,2,2)
imshow(y);
xlabel('converted Gray')
```

RGB to Index

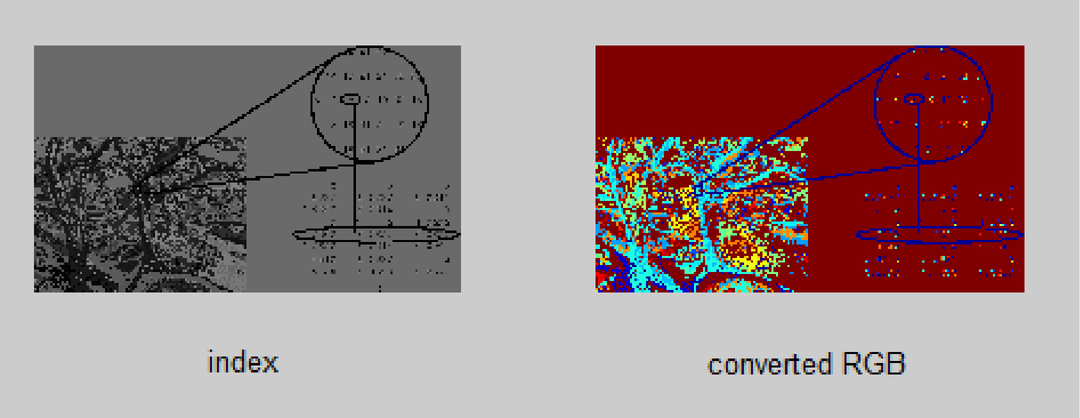
[y map]=rgb2ind(x)

```
clc
clear
x=imread('peppers.png');
imfinfo('peppers.png')
[y,map]=rgb2ind(x,128);
imwrite(y,'peppersy.png')
imfinfo('peppers.png')
subplot(1,2,1)
imshow(x)
xlabel('RGB')
subplot(1,2,2)
imshow(y);
xlabel('converted Index')
```

Index to RGB

Y=ind2rgb(x)

```
clc
clear
x=imread('chimage9.tif');
imfinfo('chimage9.tif')
y=ind2rgb(x,colormap);
imwrite(y,'chimage9y.tif')
imfinfo('chimage9y.tif')
subplot(1,2,1)
imshow(x)
xlabel('index')
subplot(1,2,2)
imshow(y);
xlabel('converted RGB')
```



Gray to BW

X=im2bw(x,map,level) , level is threshold value always b/w [0,1] regardless of class of input

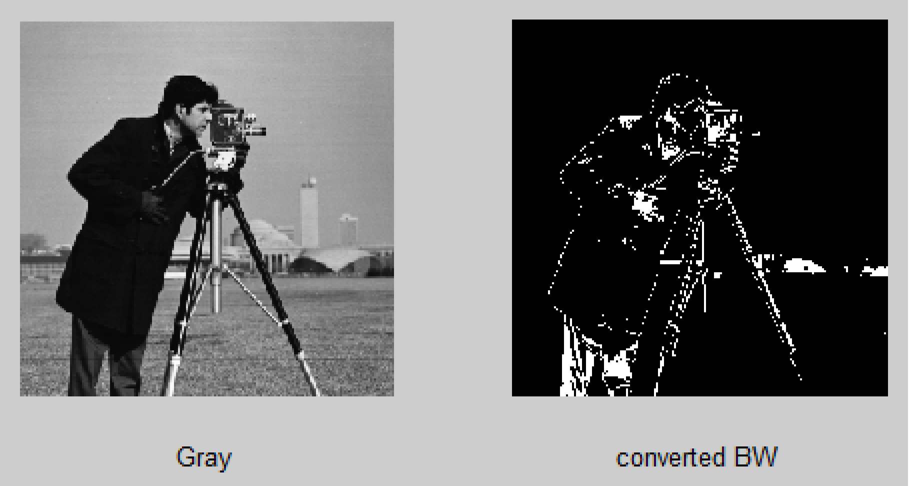
(0 – black 1—white) Even if class in uint16 in this case each threshold value is multiplied by 255 65535.

Threshold in different ways depends on the class of image, if it is 0.5 the pixel value lies in midway to black and white.

Uint8 → 0-255

Uint16 → 0-65535

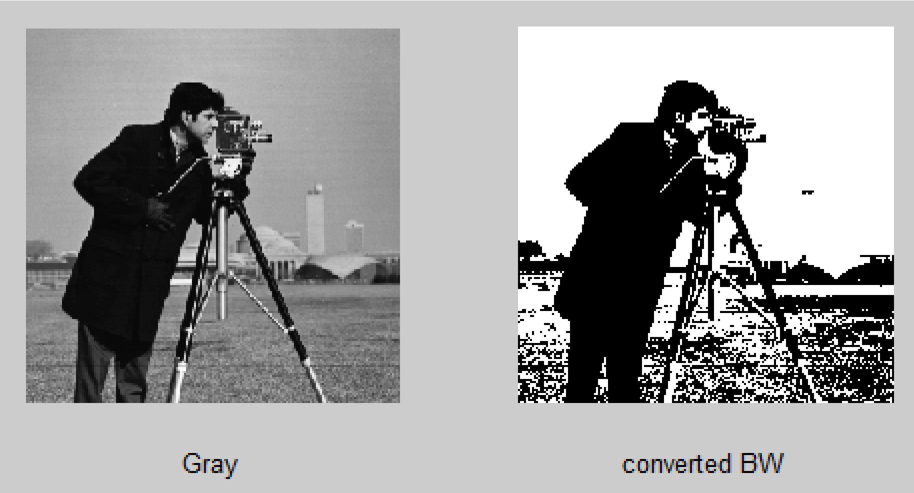
```
clc
clear
x=imread('cameraman.tif');
imfinfo('cameraman.tif')
y=im2bw(x,colormap,0.5);
imwrite(y,'cameramany.tif')
imfinfo('cameramany.tif')
subplot(1,2,1)
imshow(x)
xlabel('Gray')
subplot(1,2,2)
imshow(y);
xlabel('converted BW')
```



Im2bw(f,t) , if f belongs to uint8 class and t=0.4 in this case im2bw threshold the pixel in f by comparing them to

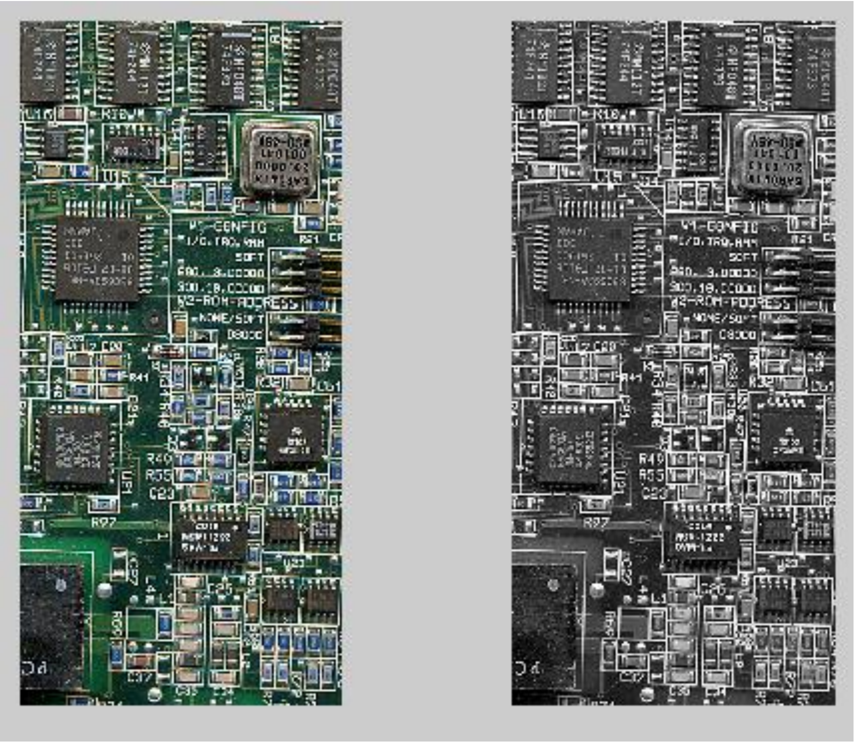
255*0.4=102

```
clc
clear
x=imread('cameraman.tif');
imfinfo('cameraman.tif')
y=im2bw(x,0.5);
imwrite(y,'cameramany.tif')
imfinfo('cameramany.tif')
subplot(1,2,1)
imshow(x)
xlabel('Gray')
subplot(1,2,2)
imshow(y);
xlabel('converted BW')
```



Example:

```
i=imread('board.tif');
j=rgb2gray(i);
subplot(1,2,1)
imshow(i)
subplot(1,2,2)
imshow(j)
```



Intensity Transformation

%without converting image type

EXERCISE QUESTIONS:

Q1 : Read image spine.tif, take its compliment and save in g. Now apply imadjust on this image with different values of $gamma$ and show all the figures on the same plot. Also show the value of $gamma$ on top of each figure.

(Hint: Use command title to show value of $gamma$ on top of figure. You can use $\backslash gamma$ to show the symbol γ in MATLAB.).

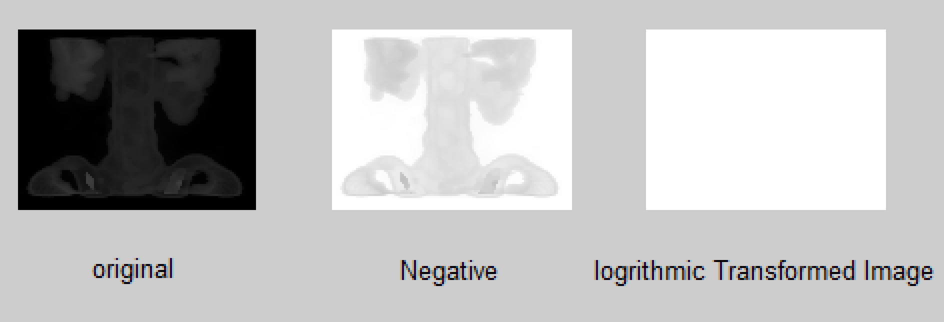
```
clc
i=imread('spine.tif');
g=imcomplement(i);
subplot(2,3,1)
imshow(i)
title('original')
subplot(2,3,2)
imshow(g)
title('complement')
```

```
a=imadjust(g, [0 1], [0 1] ,0.5);
subplot(2,3,3)
imshow(a)
x=sprintf('gamma=0.5')
title(x)
b=imadjust(g, [0 1], [0 1] ,0.1);
subplot(2,3,4)
imshow(b)
title('gamma=0.1')
c=imadjust(g, [0 1], [0 1] ,1.5);
subplot(2,3,5)
imshow(c)
title('gamma=1.5')
d=imadjust(g, [0 1], [0 1] ,2.5);
subplot(2,3,6)
imshow(d)
title('gamma=2.5')
```

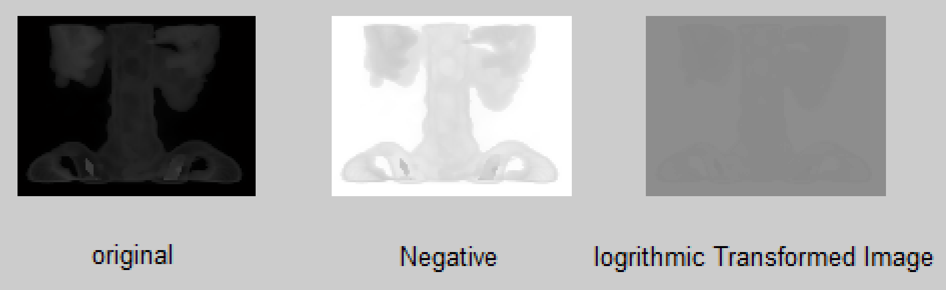
Q2 : Read image spine.tif, take its compliment and save in g. Now apply logarithmic transform on g and show both original and transformed image on same figure.

```
clc
i=imread('spine.tif');
g=imcomplement(i);
c=input('Enter C = ')
lg=c*log(1+double(g));
subplot(1,3,1)
imshow(i)
xlabel('original')
subplot(1,3,2)
imshow(g)
xlabel('Negative')
subplot(1,3,3)
imshow(lg)
xlabel('logrithmic Transformed Image')
```

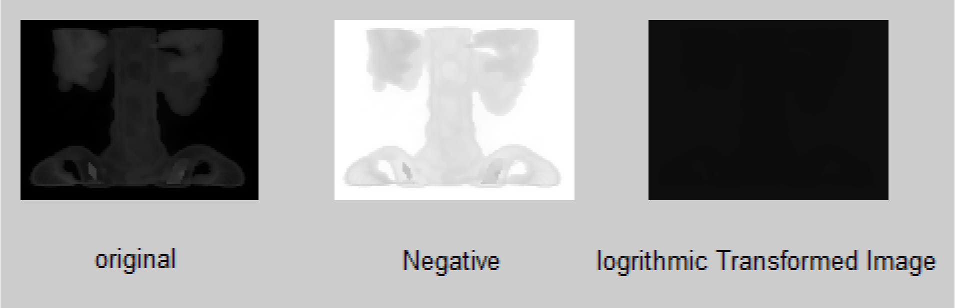
C=1



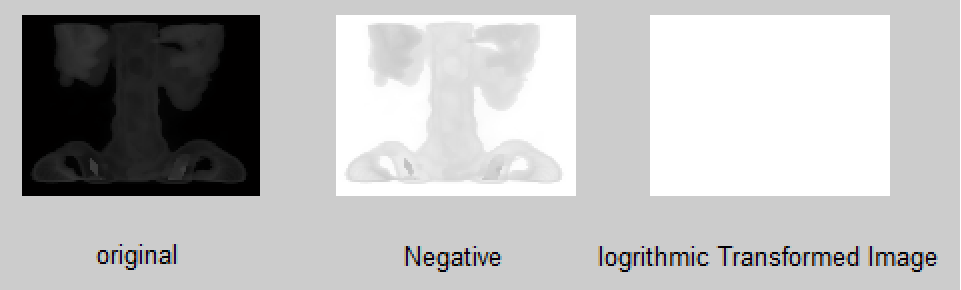
C=0.1



C=0.01



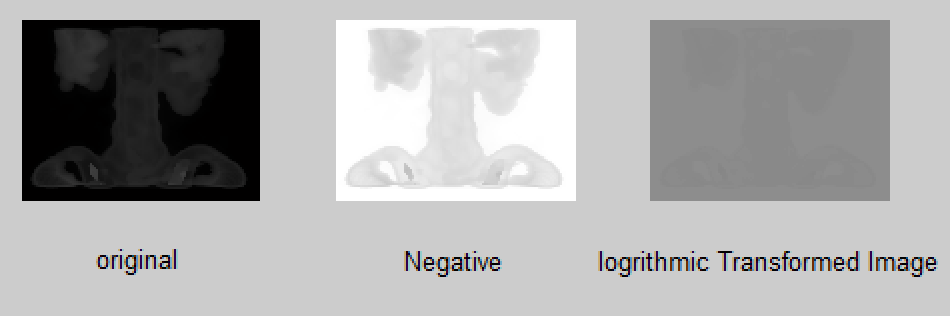
C=0.9

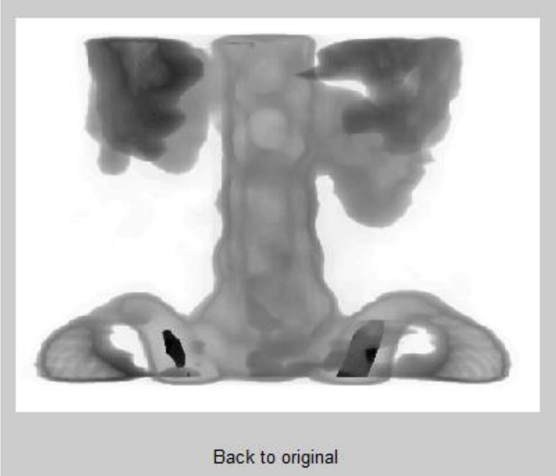


Returning to original Image

```
clc
i=imread('spine.tif');
g=imcomplement(i);
c=input('Enter C = ')
lg=c*log(1+double(g));
subplot(1,3,1)
imshow(i)
xlabel('original')
subplot(1,3,2)
imshow(g)
xlabel('Negative')
subplot(1,3,3)
imshow(lg)
xlabel('logrithmic Transformed Image')
 gs=im2uint8(mat2gray(lg))
figure
imshow(gs)
xlabel('Back to original')
```

C=0.1



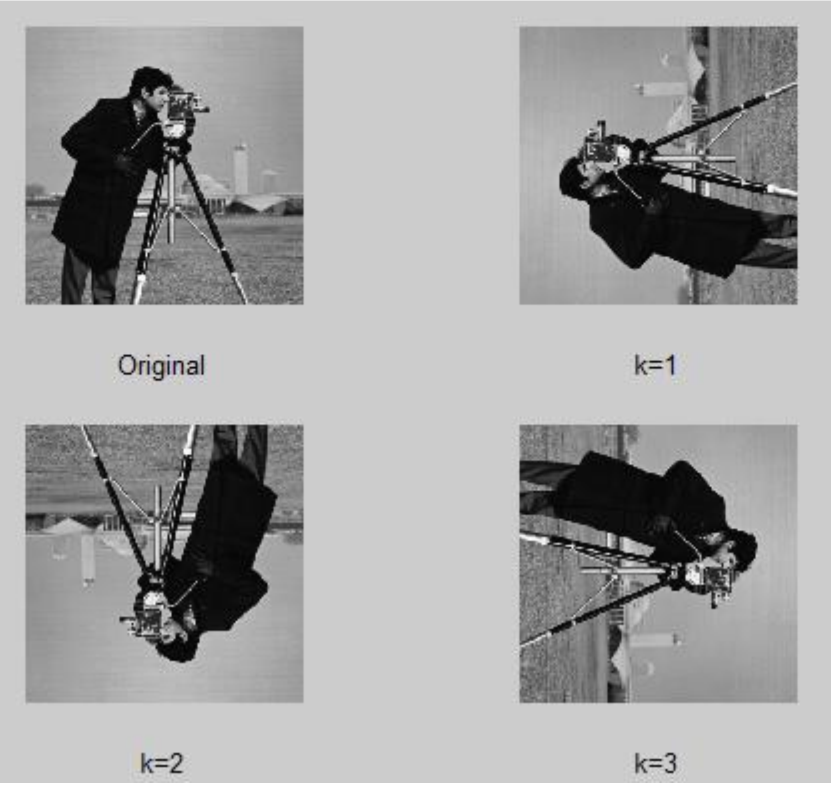


Q3: Create a Script M-file named 'Rotation.m'. Now write a code in this file which should ask the user to give name of an image file and should show four images on a single figure where the first image should be original image and other images should be 90 degree rotated compared to previous images (Hint: Use command 'input' to get input from user).

```
clc
x=input('Enter Name of Image','s');
i=imread (x)
a=rot90 (i,1);
b=rot90 (i,2);
c=rot90 (i,3);
figure('name','Rotation function Ouput');
subplot(2,2,1);
imshow(i);
xlabel('Original');
subplot(2,2,2);
```

```
imshow(a);
xlabel('k=1');
subplot(2,2,3);
imshow(b);
xlabel('k=2');
subplot(2,2,4);
imshow(c);
xlabel('k=3');
```

```
Enter Name of Image: cameraman.tif
```



Q4 : Now create another Script file 'gammatransform.m' which takes image from user as input and plots four images on one figure for different values of gamma. It should also write the value of gamma on top of each image.

```
clc
x=input('Enter Name of Image : ','s');
i=imread(x);
```

```
subplot(2,2,1)
imshow(i)
title('original')


a=imadjust(i, [0 1], [0 1] ,0.1);
subplot(2,2,2)
imshow(a)
x=sprintf('gamma=0.1')
title(x)

b=imadjust(i, [0 1], [0 1] ,0.9);
subplot(2,2,3)
imshow(a)
x=sprintf('gamma=0.9')
title(x)

c=imadjust(i, [0 1], [0 1] ,2.5);
subplot(2,2,4)
imshow(b)
title('gamma=2.5')
```

```
Enter Name of Image : cameraman.tif
```

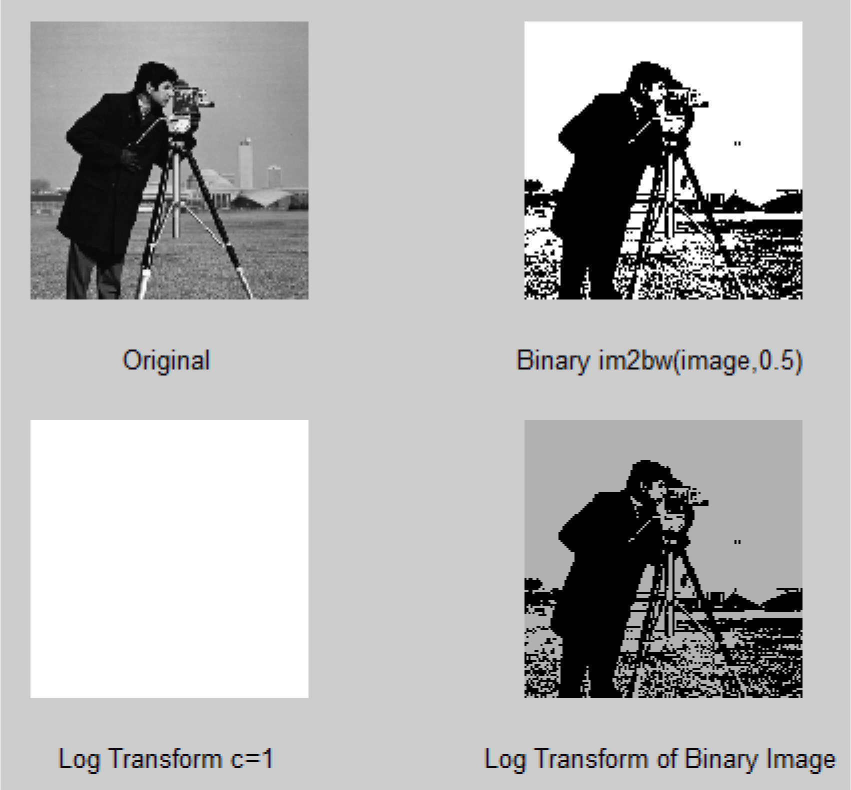
Q5 : Create a Function M-file (give any name) which should accept image file as input argument and should convert the image into binary and logarithmic and return both images as output.

Function:

```
function [ bin logrithmic bin_logrithmic ] = Q5( image )
bin=im2bw(image,0.5);
logrithmic=1*log(1+double(image));
bin_logrithmic=1*log(1+double(bin));
end
```

```
i=imread('cameraman.tif');
[bin logrithmic bin_logrithmic]=Q5(i);

subplot(2,2,1)
imshow(i)
xlabel('Original')
subplot(2,2,2)
```

```
imshow(bin)
xlabel('Binary im2bw(image,0.5)')
subplot(2,2,3)
imshow(logrithmic)
xlabel('Log Transform c=1')
subplot(2,2,4)
imshow(bin_logrithmic)
xlabel('Log Transform of Binary Image')
```

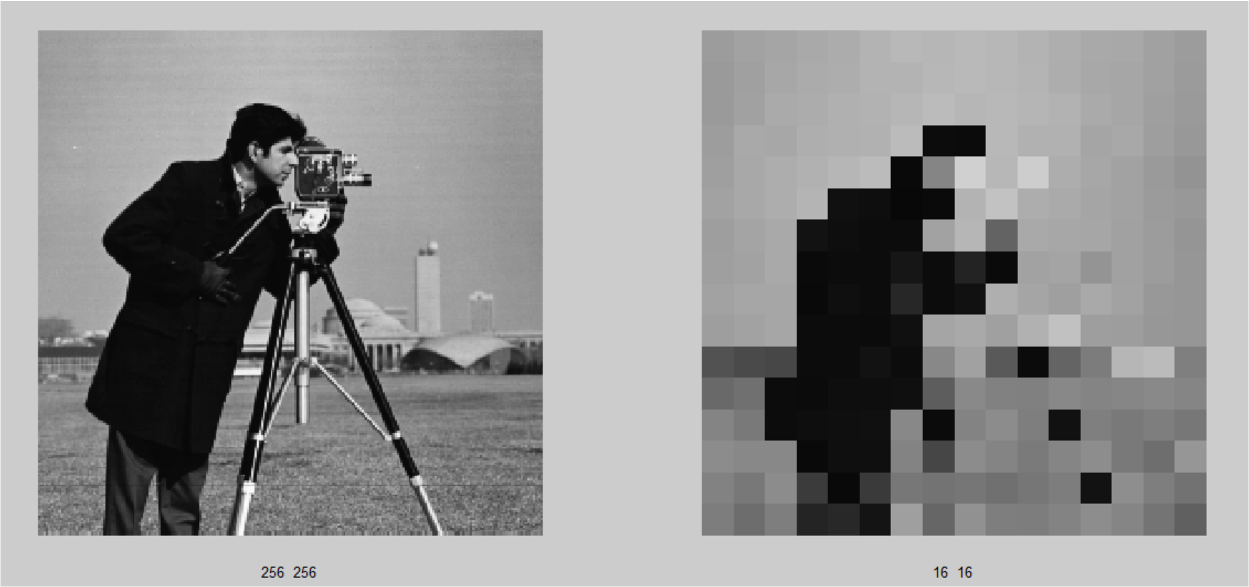


Q6 : Create a Function m-file which accepts an image as input, compresses it 16 times, displays the original and compressed image on a single figure and also returns compressed image as output.

Function M File

```
function [ comp ] = Q6( image )
subplot(1,2,1)
imshow(image)
x=sprintf(num2str(size(image)));
xlabel(x)
```

```
comp=image(1:16:end,1:16:end);
subplot(1,2,2)
imshow(comp)
x=sprintf(num2str(size(comp)));
xlabel(x)
end
>> i=imread('cameraman.tif');
>> [compressed]=Q6(i);
```



Experiment#5: To get familiar with histogram processing of images.

1. **Experiment Text**
    i. **Histogram Processing**
2. **Lab Exercise**
3. **Exercise Questions**

EXPERIMENT TEXT

1 Histogram Processing

1.1 Background

The histogram of a digital image with L possible intensity levels is a discrete function written as

$$h(r_k) = n_k$$

Where r_k is the kth intensity level and n_k is the number of pixels in the image with intensity levels r_k. The value of L for uint8 image is 256, 65536 for uint16 image and 1 for a double image. If we divide all the elements of $h(r_k)$ with total number of pixels in an image we get a normalized histogram. i.e.

$$\begin{aligned} p(r_k) &= \frac{h(r_k)}{MN} \\ &= \frac{n_k}{MN} \end{aligned}$$

Where we recognize that $p(r_k)$ is the probability of occurrence of intensity level r_k.

1.2 Image Histogram

The core function that deals with image histograms in MATLAB is imhist, which has the following syntax.

```
h = imhist(f, b)
```

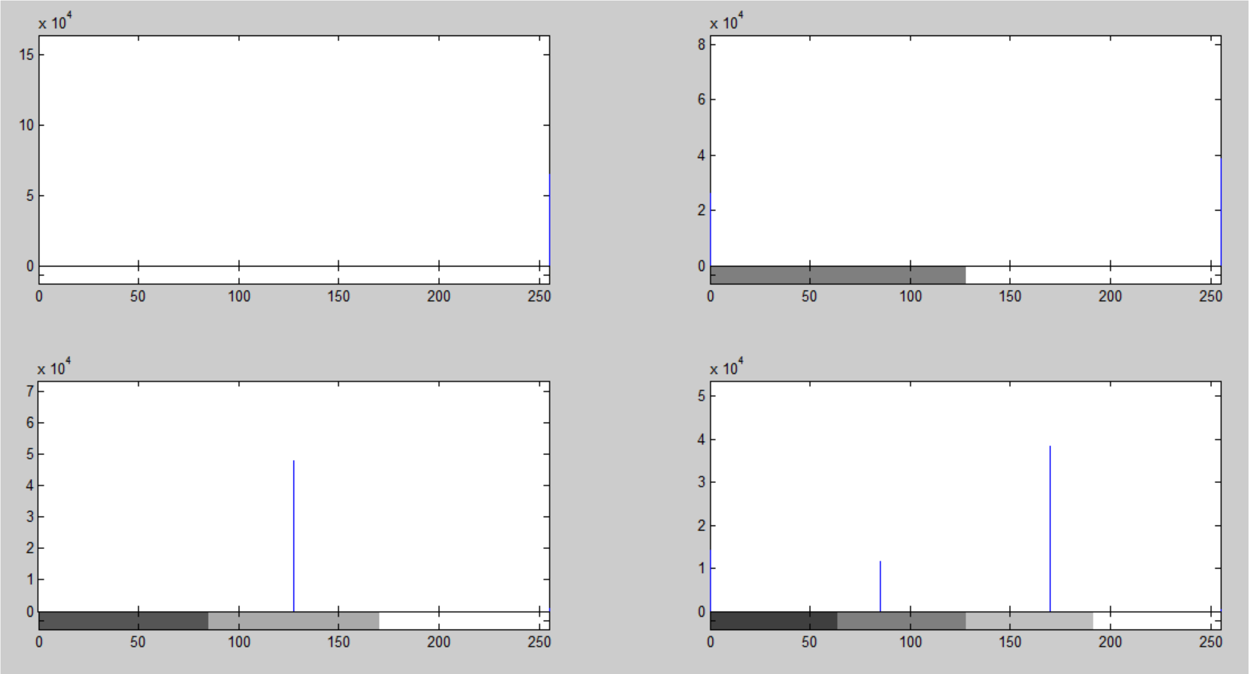
Where h is the histogram, f is the input image and b is the number of bins used in histogram. Using imhist(f,b) without h will plot the histogram of image f. The default value of b is 256. A bin is subdivision of intensity scales e.g. if for uint8 image you use b = 2, then the intensity scale is subdivided into two ranges; 0 to 127 and 128 to 255. The resulting histogram will have two values, h (1) equal to number of pixels in the range [0,127] and h (2) equal to the number of pixels in the range [128,255].

```
clc
b=1;
f=imread('cameraman.tif');
h=imhist(f,b);
subplot(2,2,1)
imhist(f,b); %without h plotting the histogram otherwise save histogram into h and don't plot it
b=2;
f=imread('cameraman.tif');
h=imhist(f,b)
subplot(2,2,2)
imhist(f,b);
b=3;
f=imread('cameraman.tif');
h=imhist(f,b)
subplot(2,2,3)
imhist(f,b);
b=4;
f=imread('cameraman.tif');
h=imhist(f,b)
subplot(2,2,4)
imhist(f,b);
```

Output

```
h =

     26477
     39059
```

```
h =


      16032
      48331
       1173



h =


      14538
      11939
      38531
        528
```



An image can be normalized by using the expression

```
h1 = imhist(f, b)/numel(f )
```

where `numel(f)` gives the number of elements in an image.

```
clc
f=imread('cameraman.tif');

numel(f)

imhist(f)/numel(f)


ans =


       65536



ans =


         0
         0
         0
         0
         0
         0
         0
    0.0001
    0.0065
    0.0225
.
.
.
.
.
.
.
    0.0000
```

```
0
0
```

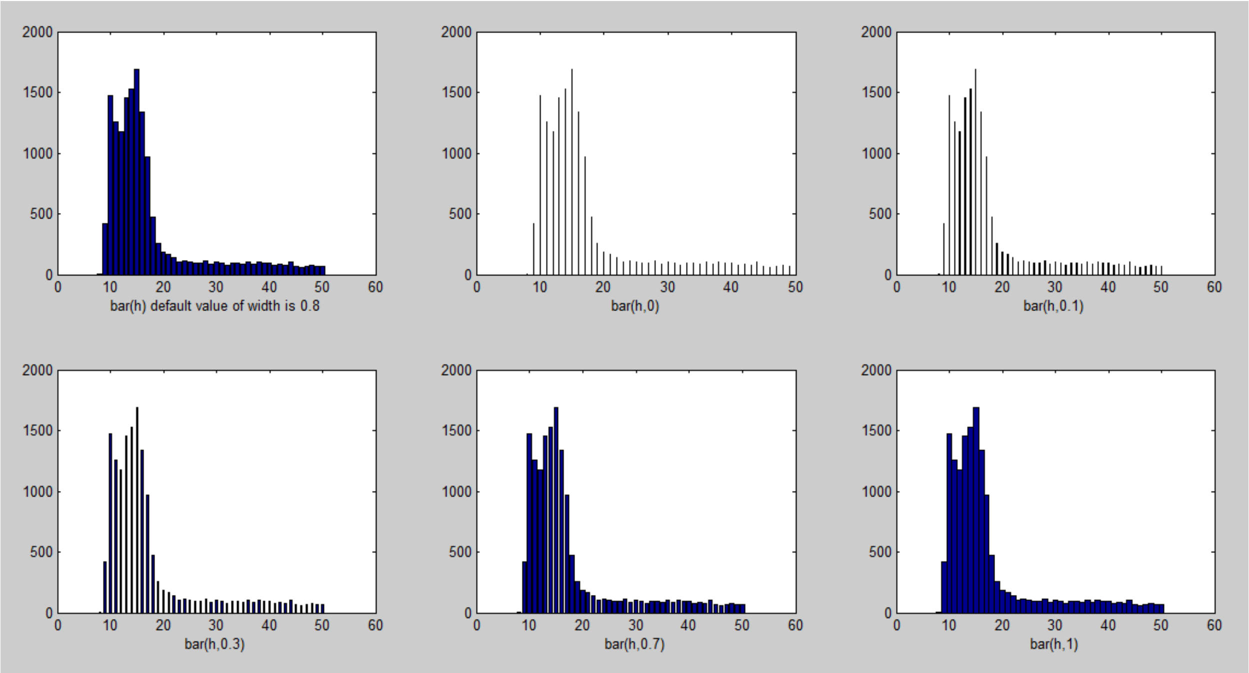
Histograms can also be plotted using some other commands e.g. histograms can be plotted using bar graphs. For this purpose we can use the function

`bar(horz, v, width)`

where v is a row vector with actual points that need to be plotted, horz contains the increments of the horizontal scale and width is a number between 0 and 1. If horz is omitted, horizontal axis is divided from 0 to length(v). When width is 1, the bars touch, when it is 0 then bars are just vertical lines. The default value of width is 0.8. The following commands show histogram using bar graph with increments of 10.

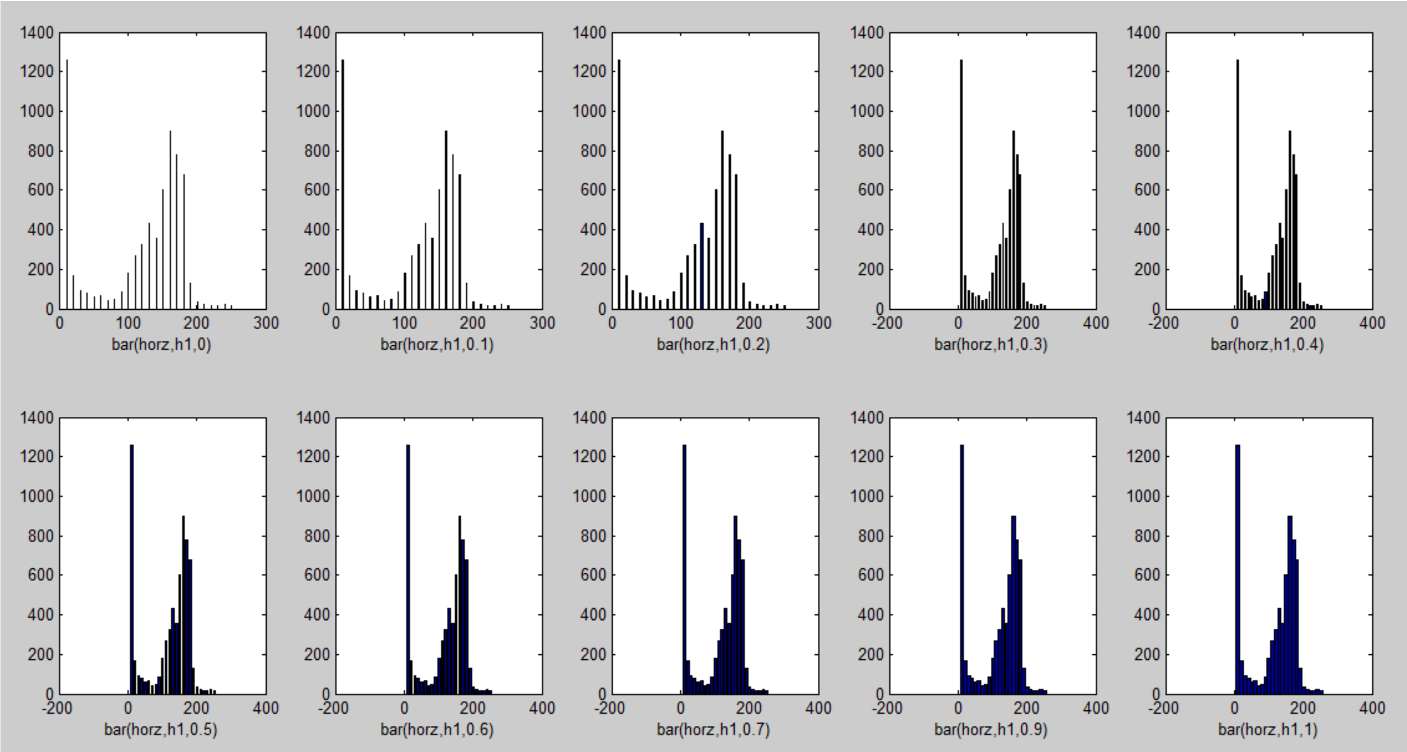
```
clc
f=imread('cameraman.tif');
f1= imhist(f);
h=f1(1:50)
subplot(2,3,1)
bar(h); % default value of width is 0.8
x=sprintf('bar(h) default value of width is 0.8');
xlabel(x)

pause

subplot(2,3,2)
bar(h,0)
x=sprintf('bar(h,0)');
xlabel(x)

pause

subplot(2,3,3)
bar(h,0.1)
x=sprintf('bar(h,0.1)');
```

```
xlabel(x)

pause
subplot(2,3,4)
bar(h,0.3)
x=sprintf('bar(h,0.3)');
xlabel(x)

pause
subplot(2,3,5)
bar(h,0.7)
x=sprintf('bar(h,0.7)');
xlabel(x)

pause
subplot(2,3,6)
bar(h,1)
x=sprintf('bar(h,1)');
xlabel(x)
```



```
h = imhist(f)
h1 = h(1:10:256)
horz = 1:10:256
subplot(2,5,1)
bar(horz,h1,0)
x=sprintf('bar(horz,h1,0)')
xlabel(x)
subplot(2,5,2)
bar(horz,h1,0.1)
x=sprintf('bar(horz,h1,0.1)')
xlabel(x)
subplot(2,5,3)
bar(horz,h1,0.2)
x=sprintf('bar(horz,h1,0.2)')
xlabel(x)
subplot(2,5,4)
bar(horz,h1,0.3)
x=sprintf('bar(horz,h1,0.3)')
```

```
xlabel(x)
subplot(2,5,5)
bar(horz,h1,0.4)
x=sprintf('bar(horz,h1,0.4)')
xlabel(x)
subplot(2,5,6)
bar(horz,h1,0.5)
x=sprintf('bar(horz,h1,0.5)')
xlabel(x)
subplot(2,5,7)
bar(horz,h1,0.6)
x=sprintf('bar(horz,h1,0.6)')
xlabel(x)
subplot(2,5,8)
bar(horz,h1,0.7)
x=sprintf('bar(horz,h1,0.7)')
xlabel(x)
subplot(2,5,9)
bar(horz,h1,0.9)
x=sprintf('bar(horz,h1,0.9)')
xlabel(x)
subplot(2,5,10)
bar(horz,h1,1)
x=sprintf('bar(horz,h1,1)')
xlabel(x)
```

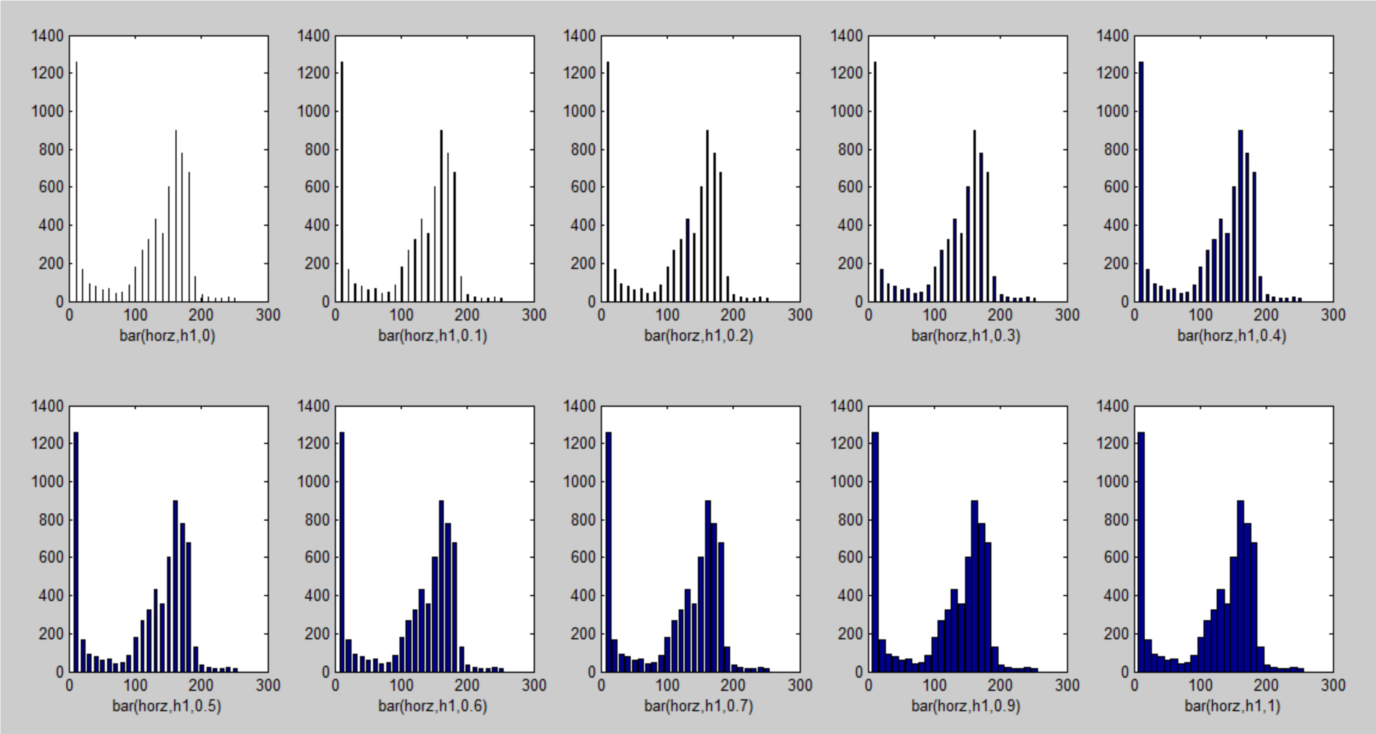


where the command axis is used to set the range of x and y axis. You can change values of horz and width to see what effect it would have on the image.

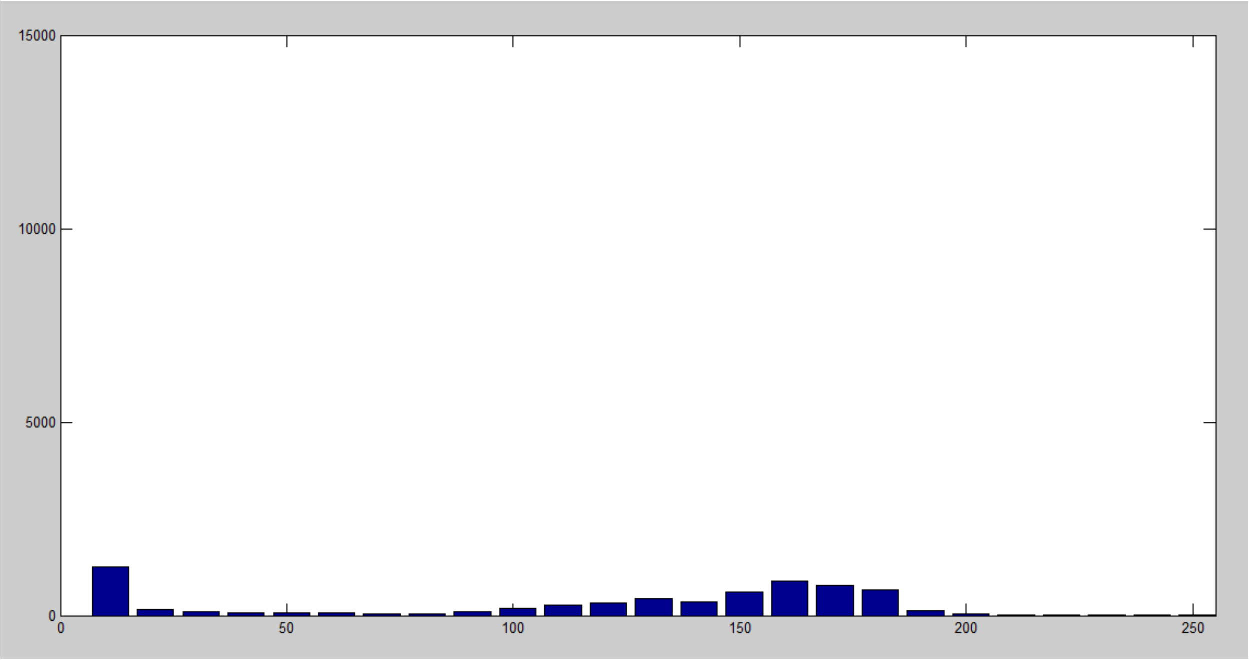
```
h = imhist(f)
h1 = h(1:10:256)
horz = 1:10:256
subplot(2,5,1)
bar(horz,h1,0)
axis ([ 0 300 0 1400]);
x=sprintf('bar(horz,h1,0)')
xlabel(x)
subplot(2,5,2)
bar(horz,h1,0.1)
axis ([ 0 300 0 1400]);
x=sprintf('bar(horz,h1,0.1)')
xlabel(x)
subplot(2,5,3)
bar(horz,h1,0.2)
axis ([ 0 300 0 1400]);
```

```
x=sprintf('bar(horz,h1,0.2)')
xlabel(x)
subplot(2,5,4)
bar(horz,h1,0.3)
axis ([ 0 300 0 1400]);
x=sprintf('bar(horz,h1,0.3)')
xlabel(x)
subplot(2,5,5)
bar(horz,h1,0.4)
axis ([ 0 300 0 1400]);
x=sprintf('bar(horz,h1,0.4)')
xlabel(x)
subplot(2,5,6)
bar(horz,h1,0.5)
axis ([ 0 300 0 1400]);
x=sprintf('bar(horz,h1,0.5)')
xlabel(x)
subplot(2,5,7)
bar(horz,h1,0.6)
axis ([ 0 300 0 1400]);
x=sprintf('bar(horz,h1,0.6)')
xlabel(x)
subplot(2,5,8)
bar(horz,h1,0.7)
axis ([ 0 300 0 1400]);
x=sprintf('bar(horz,h1,0.7)')
xlabel(x)
subplot(2,5,9)
bar(horz,h1,0.9)
axis ([ 0 300 0 1400]);
x=sprintf('bar(horz,h1,0.9)')
xlabel(x)
```

```
subplot(2,5,10)
bar(horz,h1,1)
axis ([ 0 300 0 1400]);
x=sprintf('bar(horz,h1,1)')
xlabel(x)
```

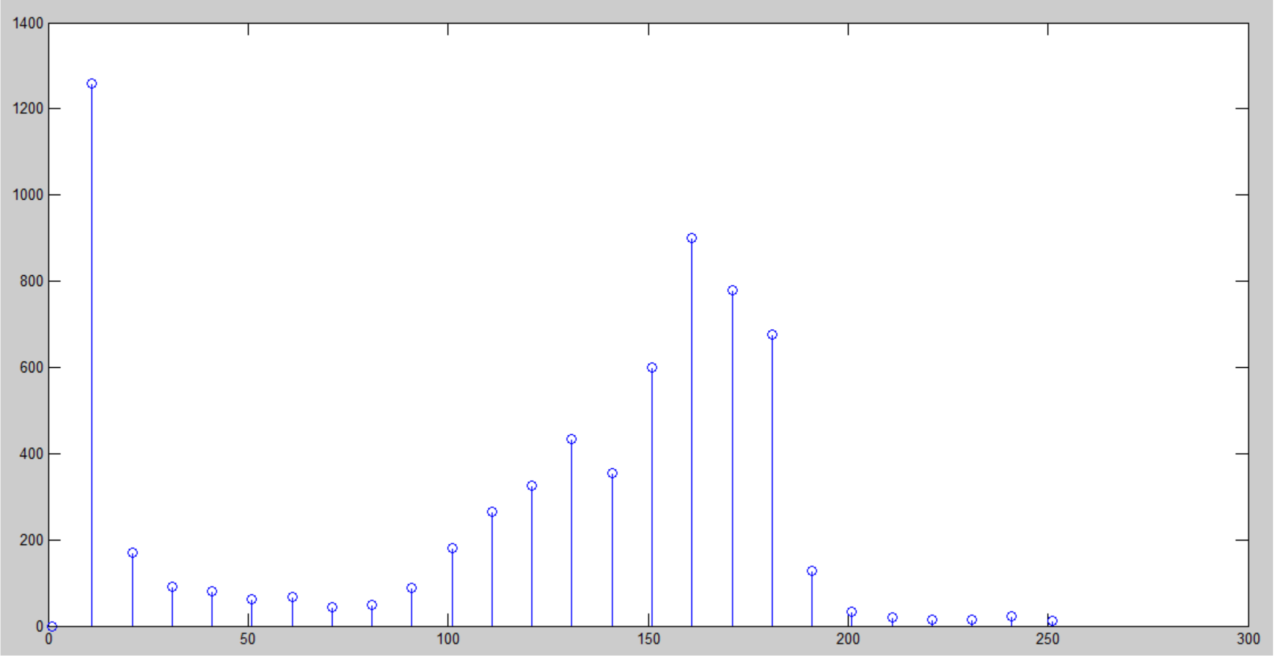


```
h = imhist(f)
h1 = h(1:10:256)
horz = 1:10:256
bar(horz,h1)
axis ([ 0 255 0 15000]);
```



Another way to show histogram is using stem function which has similar syntax as bar but does not have width parameter.

```
clc
h = imhist(f)
h1 = h(1:10:256)
horz = 1:10:256
stem(horz,h1)
axis ([ 0 300 0 1400]);
```



1.3 Histogram Equalization

Histogram equalization is a process to increase the contrast of an image by changing the histogram of an image such that the histogram of new image covers a wide range of intensity levels. Suppose that we have an image with intensity levels between [0,L-1] and let $p_r(r)$ be its probability density function. Suppose we perform the following transformation to obtain an output image

$$s = T(r) = \int_0^r p_r(w)dw$$

It can be shown (refer to lectures) that using this transformation function generates an image whose intensity levels are equally likely. The net result is an image with increased dynamic range, which will have a high contrast. For discrete quantities we deal with summations and the above transform can be written as

$$\begin{aligned} s_k &= T(r_k) \\ &= \sum_{j=1}^{k} p_r(r_j) \\ &= \sum_{j=1}^{k} n_j/MN \qquad (1) \end{aligned}$$

where s_k is the intensity value in the output image corresponding to r_k intensity in the input image.

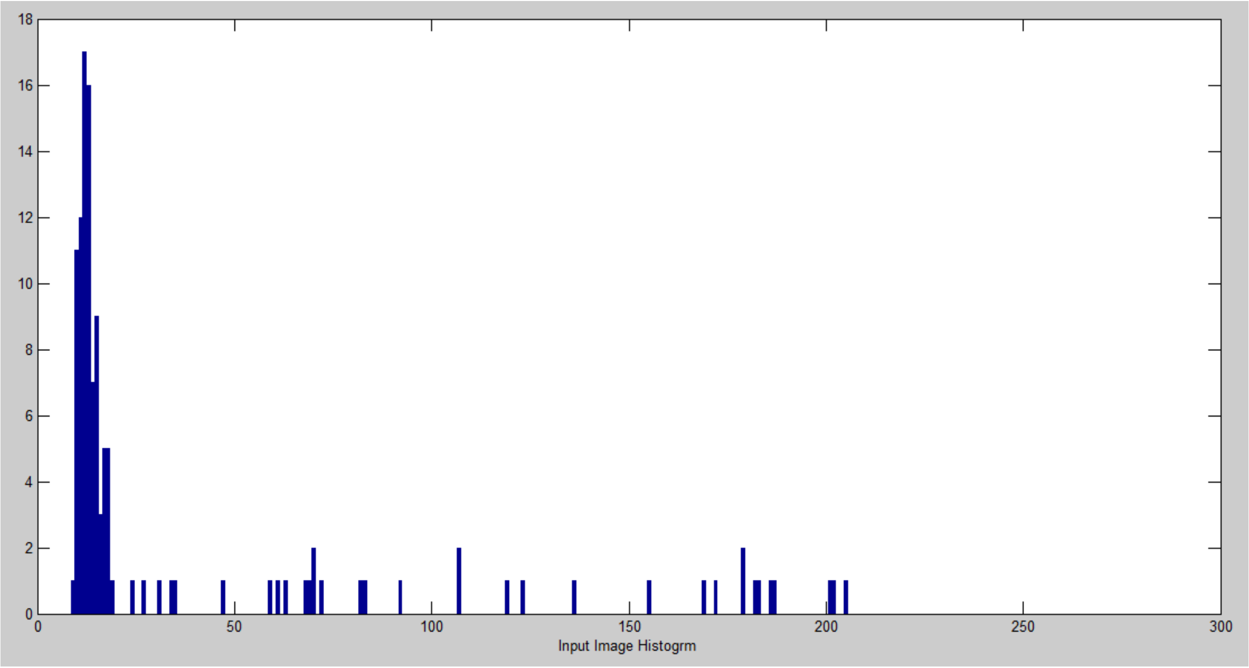
Histogram equalization is performed in MATLAB using the function `histeq` which has the syntax

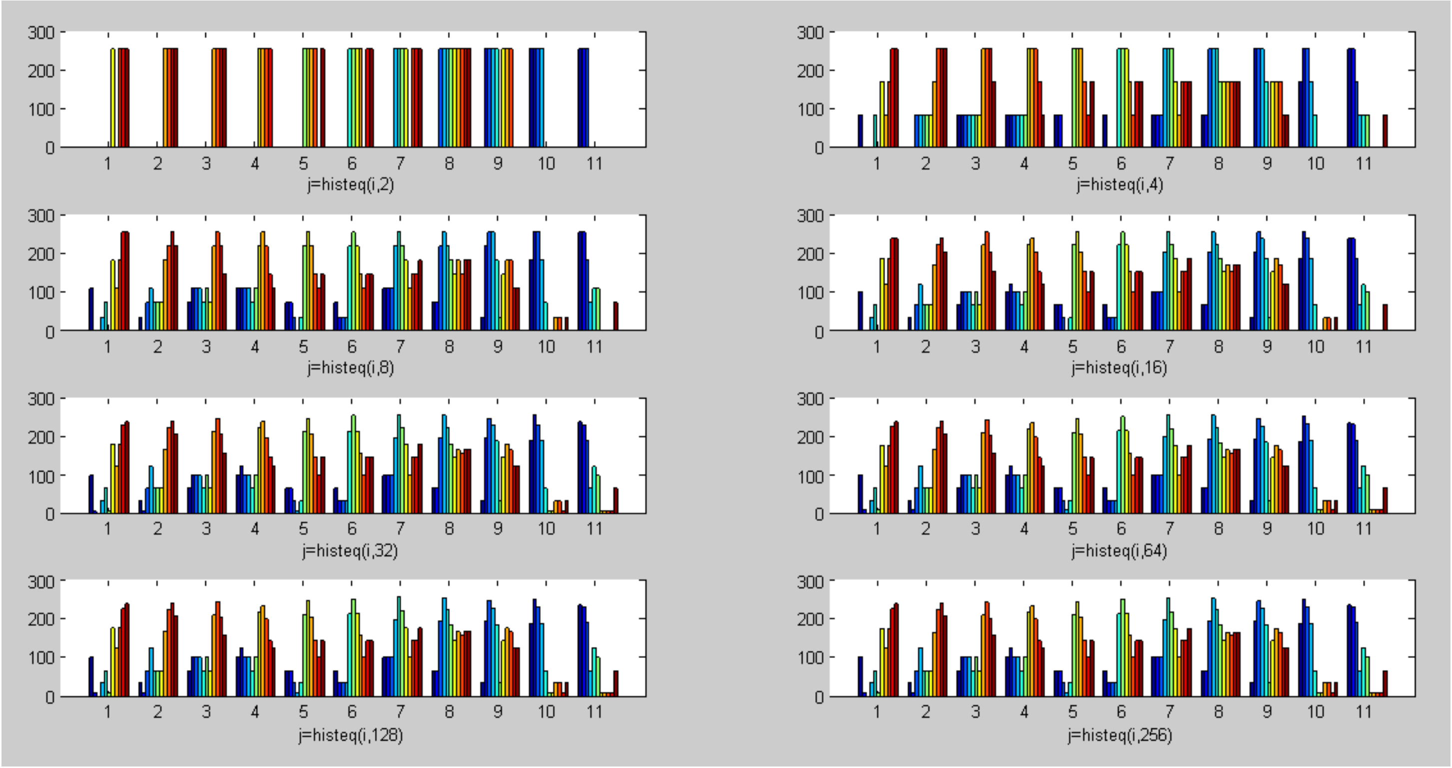
```
>> g = histeq(f,nlev);
```

where f is the input image and nlev is the number of intensity level for the output image. For equation 1, the number of output intensity levels (nelv) should be equal to the input intensity levels. The default value of nlev is 64.

```
%g=histeq(f,nlev)
clc
f=imread('cameraman.tif');
i=f(100:110,100:110)
j=imhist(i);
bar(j)
xlabel('Input Image Histogrm')
figure
j=histeq(i,2);
subplot(4,2,1)
bar(j,1)
l=sprintf('j=histeq(i,2)')
xlabel(l)
j=histeq(i,4);
subplot(4,2,2)
bar(j,1)
l=sprintf('j=histeq(i,4)')
xlabel(l)

j=histeq(i,8);
subplot(4,2,3)
bar(j,1)
l=sprintf('j=histeq(i,8)')
xlabel(l)

j=histeq(i,16);
```

```
subplot(4,2,4)
bar(j,1)
l=sprintf('j=histeq(i,16)')
xlabel(l)

j=histeq(i,32);
subplot(4,2,5)
bar(j,1)
l=sprintf('j=histeq(i,32)')
xlabel(l)

j=histeq(i,64);
subplot(4,2,6)
bar(j,1)
l=sprintf('j=histeq(i,64)')
xlabel(l)

j=histeq(i,128);
subplot(4,2,7)
bar(j,1)
l=sprintf('j=histeq(i,128)')
xlabel(l)

j=histeq(i,256);
subplot(4,2,8)
bar(j,1)
l=sprintf('j=histeq(i,256)')
xlabel(l)
```





```
clc
i=imread('cameraman.tif');
figure
imshow(i)
```

```
xlabel('Input Image')
figure
j=histeq(i,2);
subplot(2,4,1)
imshow(j)
l=sprintf('j=histeq(i,2)')
xlabel(l)
j=histeq(i,4);
subplot(2,4,2)
imshow(j)
l=sprintf('j=histeq(i,4)')
xlabel(l)

j=histeq(i,8);
subplot(2,4,3)
imshow(j)
l=sprintf('j=histeq(i,8)')
xlabel(l)

j=histeq(i,16);
subplot(2,4,4)
imshow(j)
l=sprintf('j=histeq(i,16)')
xlabel(l)

j=histeq(i,32);
subplot(2,4,5)
imshow(j)
l=sprintf('j=histeq(i,32)')
xlabel(l)

j=histeq(i,64);
```

```
subplot(2,4,6)
imshow(j)
l=sprintf('j=histeq(i,64)')
xlabel(l)

j=histeq(i,128);
subplot(2,4,7)
imshow(j)
l=sprintf('j=histeq(i,128)')
xlabel(l)

j=histeq(i,256);
subplot(2,4,8)
imshow(j)
l=sprintf('j=histeq(i,256)')
xlabel(l)
```

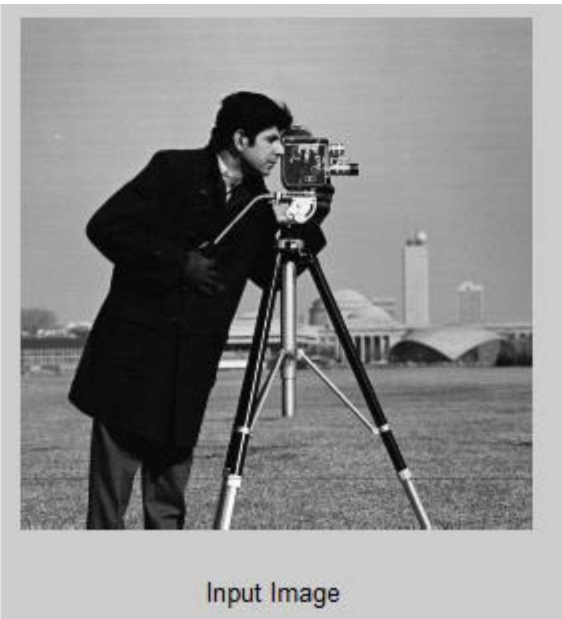
As explained earlier that the transformation function (T (r)) for making a high contrast image is simply the sum of normalised histogram values. The transformation function can be obtained using the MATLAB cumsum function. e.g.

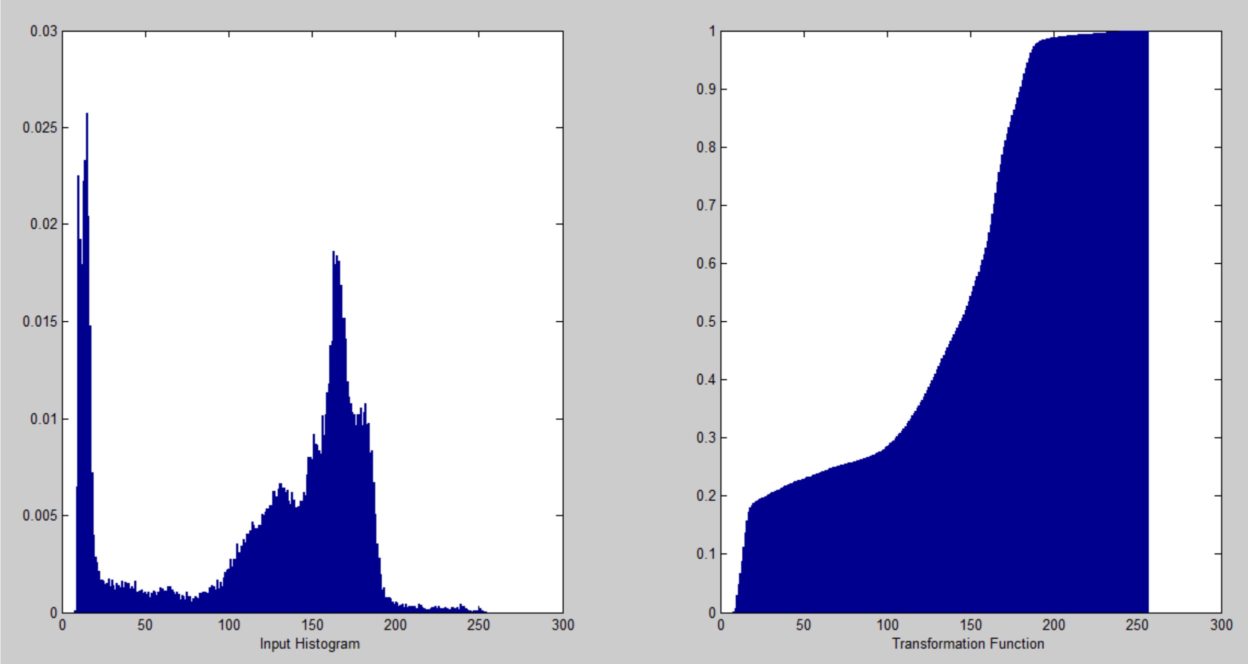
```
>>  hnorm = imhist(f)./numel(f);
>>  cdf = cumsum(hnorm);
```

where cdf is the transformation function.

```
clc
i=imread('cameraman.tif');
figure
imshow(i)
xlabel('Input Image')
figure
subplot(1,2,1)
tr=imhist(i)./numel(i);
bar(tr)
xlabel('Input Histogram')
subplot(1,2,2)
```

```
im_cdf=cumsum(tr);
bar(im_cdf)
xlabel('Transformation Function')
```



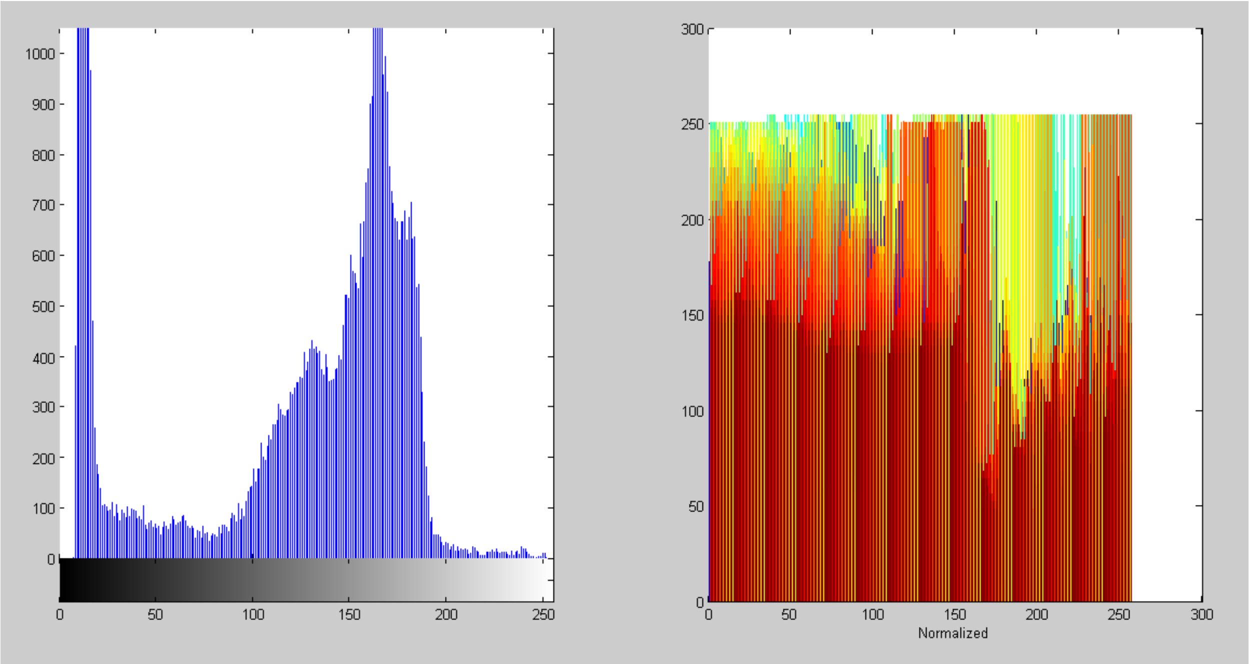


LAB EXERCISE

EXERCISE QUESTIONS

Q1 : Read image cameraman.tif, show the image, show its histogram using imhist and its normalised histogram using bar. All the images should be on one figure and you should use a script file.

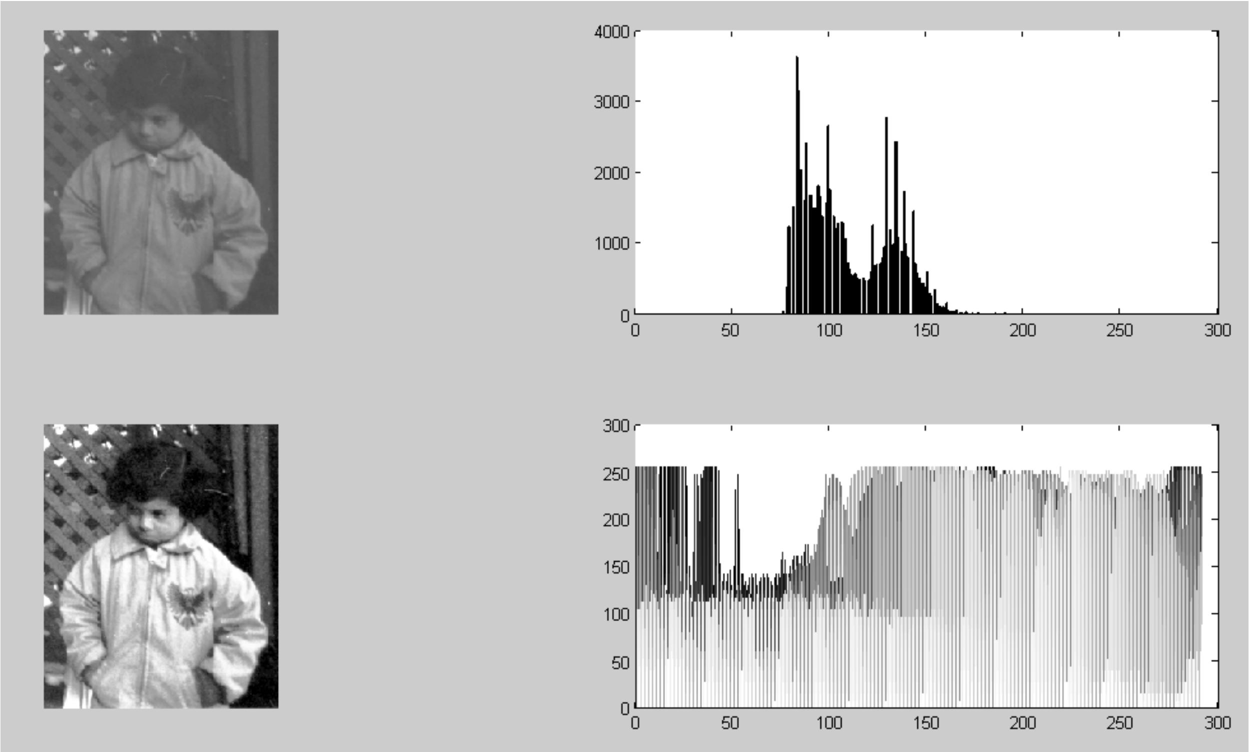
```
clc
f=imread('cameraman.tif');
he=histeq(f);
subplot(1,2,1)
imhist(f)
subplot(1,2,2)
bar(he)
xlabel('Normalized')
```



Q2 : Read image pout.tif, show the image, its histogram, equalized image and equalized histogram, all on one figure using a script file. Now repeat the above steps for another image circuit.tif.

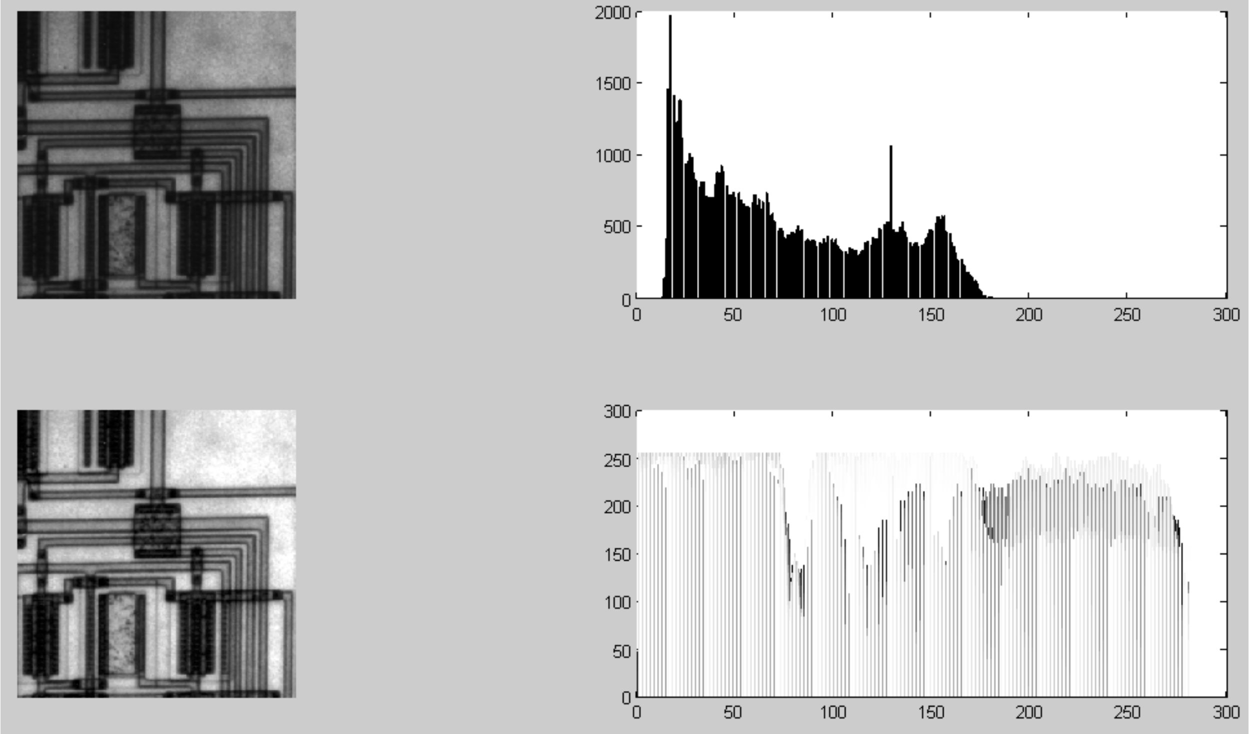
```
clc
f=imread('pout.tif');
histogram=imhist(f);
```

```
eq=histeq(f);
subplot(2,2,1)
imshow(f)
subplot(2,2,2)
bar(histogram)
subplot(2,2,3)
imshow(eq)
subplot(2,2,4)
bar(eq)
```



```
clc
f=imread('circuit.tif');
histogram=imhist(f);
eq=histeq(f);
subplot(2,2,1)
imshow(f)
subplot(2,2,2)
bar(histogram)
```

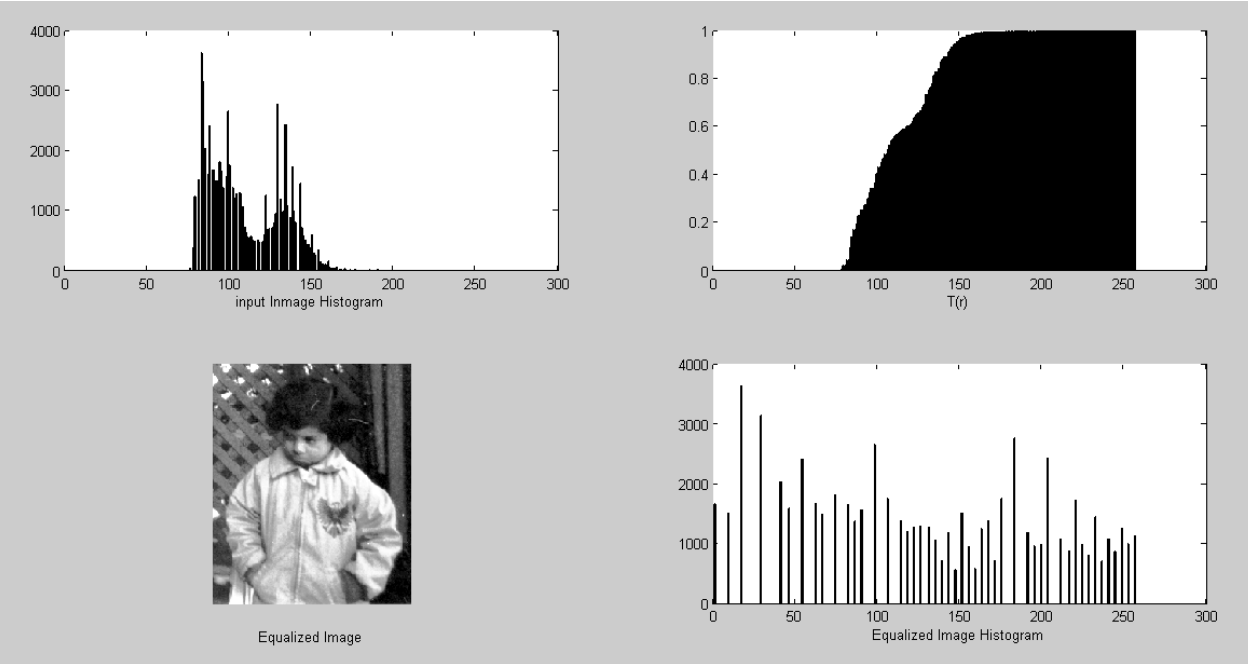
```
subplot(2,2,3)
imshow(eq)
subplot(2,2,4)
bar(eq)
```



Q3 : For Q2 and for image pout.tif show its histogram, the transformation function and histogram of transformed image on one figure.

```
clc
f=imread('pout.tif');
subplot(2,2,1)
bar(imhist(f))
xlabel('input Inmage Histogram')
nor=imhist(f)/numel(f);
tr=cumsum(nor);
subplot(2,2,2)
bar(tr)
xlabel('T(r)')
subplot(2,2,3)
```

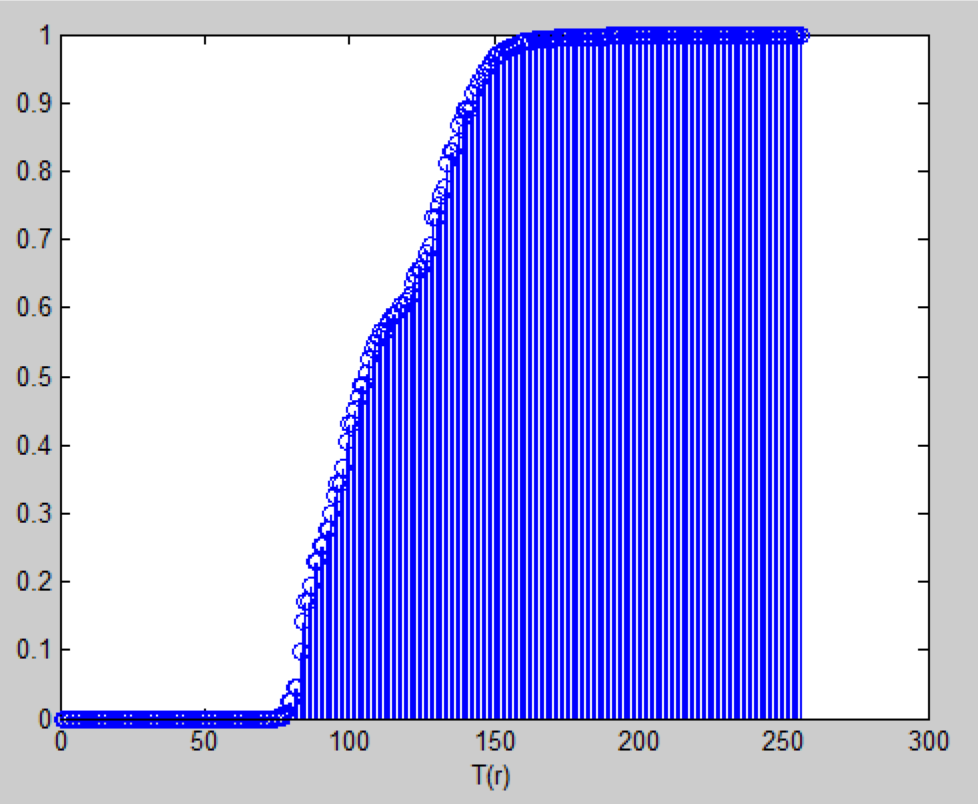
```
x=histeq(f);
imshow(x)
xlabel('Equalized Image')
subplot(2,2,4)
bar(imhist(x));
xlabel('Equalized Image Histogram')
```

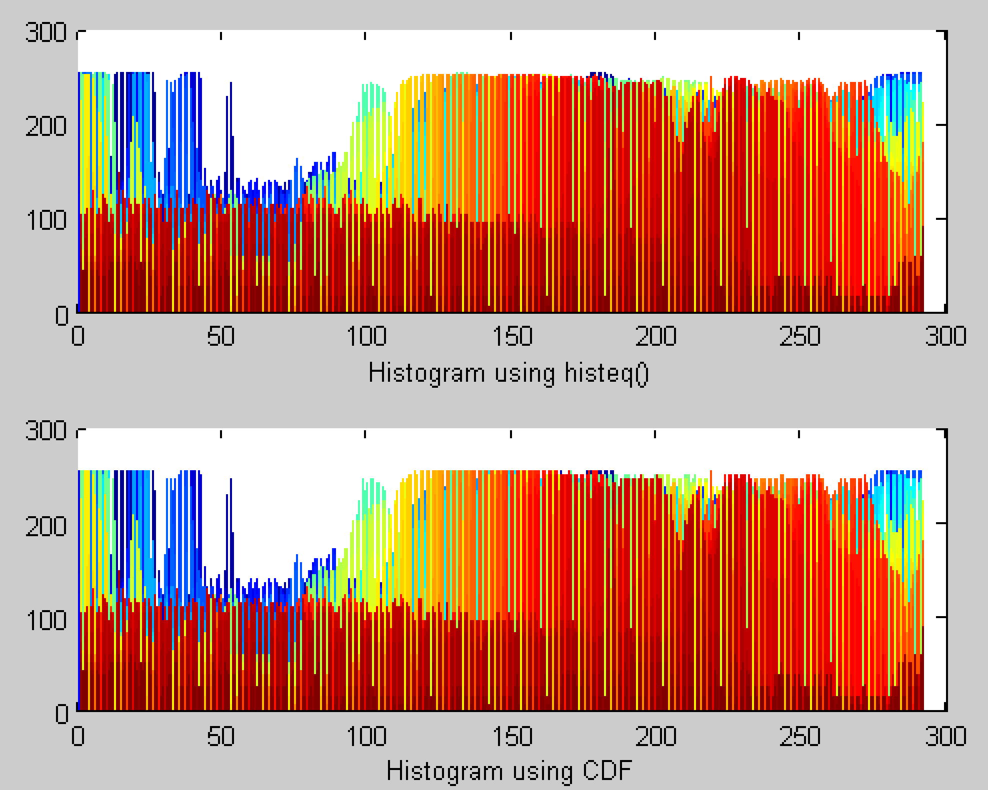


Q4 : For image pout.tif use cdf to convert it into high contrast image without using histeq. Now compare your result with histeq result. (Hint : Use for loop to map input intensities to output intensities and consult example 3.5 of your lecture notes).

```
clc
f=imread('pout.tif');
nor=imhist(f)./numel(f)
cdf=cumsum(nor);
stem(cdf);
xlabel('T(r)')
h=histeq(f,256);
figure
subplot(2,1,1)
```

```
bar(h)
xlabel('Histogram using histeq()');
for i=1:256
   x(round(cdf(i))+1) =nor(i);
end
subplot(2,1,2)
bar(x)
xlabel('Histogram using CDF');
```




300
200
100
0
0
50
100
150
200
250
300
Histogram using histeq()
300
200
100
0
0
50
100
150
200
250
300
Histogram using CDF

Experiment#6: To get familiar with histogram matching, local histogram processing and filtering.

1. **Experiment Text**
   1. **Histogram Matching**
   2. **Local Histogram Processing**
   3. **Spatial Filtering**
2. **Lab Exercise**
3. **Exercise Questions**

EXPERIMENT TEXT:

1 Histogram Matching

1.1 Background

Histogram equalization produces a transformation function that is adaptive in the sense that it is based on the histogram of the given image. It produces enhancement by spreading the intensities of the input image over a wider range of the intensity scale. We will show in the next section that this does not always lead to successful result. In some applications we want to specify the shape of the histogram that we wish the output image to have. The method used to generate an image that has a specified histogram is called h istogram matching.

Histogram matching can be done using histogram equalisation. Suppose r is the intensity of the input image and z is the intensity of the output image. Let probability density functions of the input and output image be p_R (r) and p_Z (z). We know for histogram equalization

$$s = T(r) = \int_0^r p_r(w)dw$$

where s has uniform probability density function. Let us define a var iable z with the property

$$H(z) = \int_0^z p_z(w)dw = s$$

where z is the intensity of the histogram matched image and p_Z (z) is its histogram. From the preceding two equations it follows that

$$z = H^{-1}(s) = H^{-1}[T(r)]$$

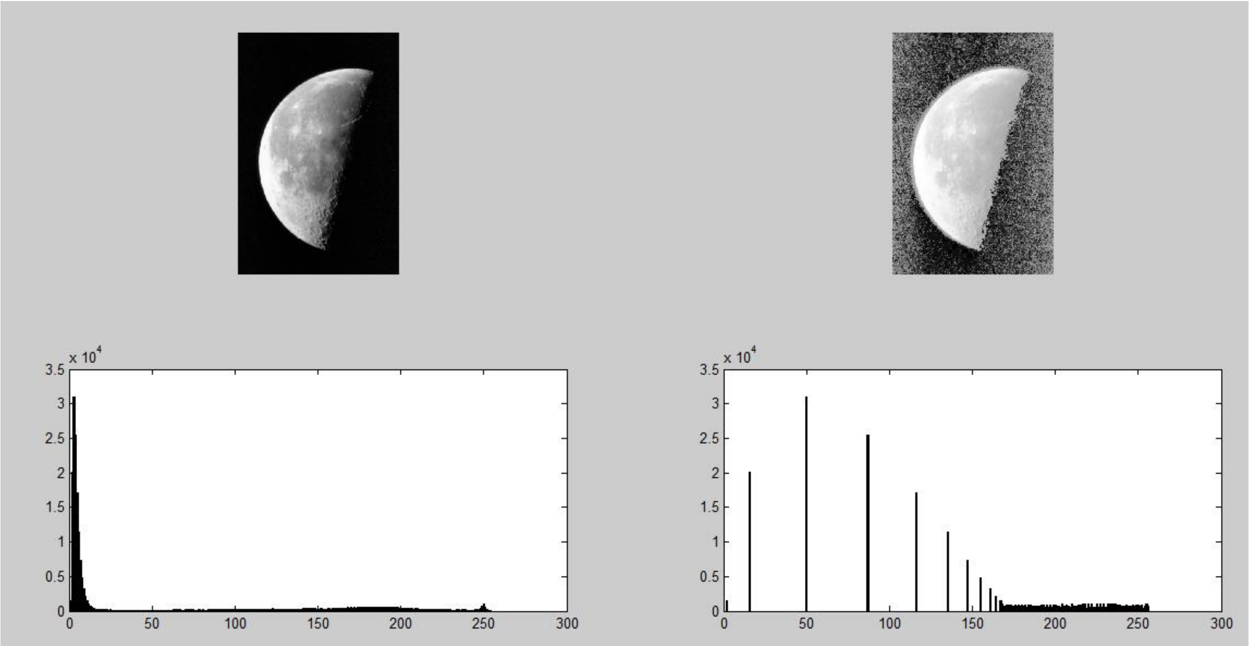
So we can use the preceding equation to find the transformed level `z`. The toolbox uses the following syntax to implement histogram matching

```
g = histeq(f, hspec);
```

where `f` is the input image, `hspec` is the specified histogram (a row vector of intensity values) and `g` is the output image whose histogram matches the specified histogram, `hspec`.

Read the image `moon.tif` in variable `f`. Now create a histogram equalised image `g` from `f` (Refer to Experiment 5 for histogram equalised image). Now show both, original and histogram equalised image on the same figure. The figure will show you that the histogram equalization did not produce a very good result (enhancement) in this case. The reason for this can be seen by studying the histograms of both the images. Now plot image `f`, its histogram, image `g` and its histogram, all on one figure .You will see that most of the intensity levels are shifted to the upper half of the grey scale, giving the image a washed out appearance. The cause of this shift is the large concentration of dark intensities in the original image.

```
f=imread('moon.tif');
g=histeq(f,256);
fh=imhist(f);
gh=imhist(g);
subplot(2,2,1)
imshow(f)
subplot(2,2,2)
imshow(g)
subplot(2,2,3)
bar(fh)
subplot(2,2,4)
bar(gh)
```



The solution to this problem is histogram matching, with the desired histogram having lesser concentration of intensity levels on the lower end of the gray scale and maintaining the general shape of the original histogram. Unfortunately, there is no straight forward method in MATLAB for histogram matching. The user has to develop his own function files to specify what kind of histogram user wants output image to have and it is beyond undergraduate level. But there is another solution to this problem and it is local histogram processing.

2 Local Histogram Processing

Local histogram processing can be used when we want to enhance the contrast of small areas of an image. We saw in the previous section that the global histogram method failed to produce good results when enhancement was needed in small areas. The syntax for local histogram processing in MATLAB is

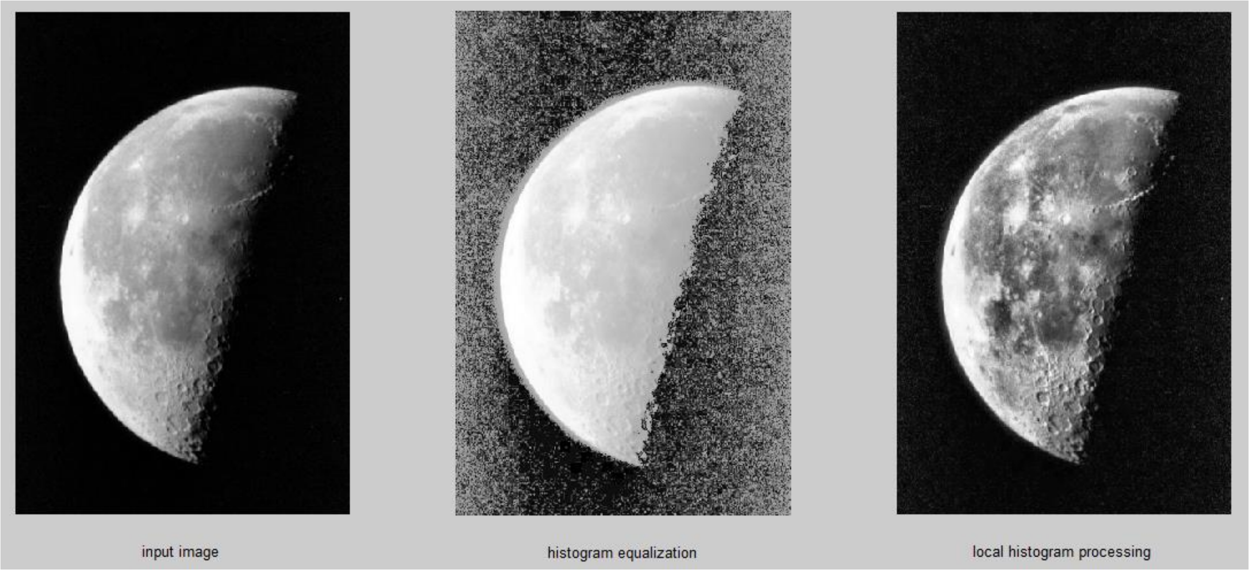
```
g = adapthisteq(f, param1, val1, param2, val2, ...)
```

Now apply the following function on image `moon.tif` and plot both the original and modified image on the same plot.

```
>> g = adpathisteq(f);
```

Do you see the improvement in the new image. Now get another image by using the following command and plot all three images on same figure.

```
clc

f=imread('moon.tif');
subplot(1,3,1)
imshow(f)
xlabel('input image')

g=histeq(f,256);
subplot(1,3,2)
imshow(g)
xlabel('histogram equalization')

gl=adapthisteq(f);
subplot(1,3,3)
imshow(gl)
xlabel('local histogram processing')
```



```
>> g = adpathisteq(f,’NumTiles’,[25 25]);
```

```
clc
```

```
gl=adapthisteq(f,'NumTiles',[25 25]);
subplot(2,3,1)
imshow(gl)
xlabel('NumTiles [25 25]')

gl=adapthisteq(f,'NumTiles',[50 50]);
subplot(2,3,2)
imshow(gl)
xlabel('NumTiles [50 50]')

gl=adapthisteq(f,'NumTiles',[100 100]);
subplot(2,3,3)
imshow(gl)
xlabel('NumTiles [100 100]')

gl=adapthisteq(f,'NumTiles',[150 150]);
subplot(2,3,4)
imshow(gl)
xlabel('NumTiles [150 150]')

gl=adapthisteq(f,'NumTiles',[200 200]);
subplot(2,3,5)
imshow(gl)
xlabel('NumTiles [200 200]')

gl=adapthisteq(f,'NumTiles',[250 250]);
subplot(2,3,6)
imshow(gl)
xlabel('NumTiles [250 250]')
```

Now get another image by using the following command and plot all four images on same figure.

```
>> g = adpathisteq(f,’NumTiles’,[25 25],’ClipLimit’,0.02);
```

```
clc
gl=adapthisteq(f,'NumTiles',[25 25],'ClipLimit',0.1);
subplot(2,3,1)
imshow(gl)
xlabel('NumTiles [25 25] ClipLimit 0.1')

gl=adapthisteq(f,'NumTiles',[25 25],'ClipLimit',0.2);
subplot(2,3,2)
imshow(gl)
xlabel('NumTiles [25 25] ClipLimit 0.2')

gl=adapthisteq(f,'NumTiles',[25 25],'ClipLimit',0.3);
subplot(2,3,3)
```

```
imshow(gl)
xlabel('NumTiles [25 25] ClipLimit 0.3')

gl=adapthisteq(f,'NumTiles',[25 25],'ClipLimit',0.4);
subplot(2,3,4)
imshow(gl)
xlabel('NumTiles [25 25] ClipLimit 0.4')

gl=adapthisteq(f,'NumTiles',[25 25],'ClipLimit',0.5);
subplot(2,3,5)
imshow(gl)
xlabel('NumTiles [25 25] ClipLimit 0.5')

gl=adapthisteq(f,'NumTiles',[25 25],'ClipLimit',0.6);
subplot(2,3,6)
imshow(gl)
xlabel('NumTiles [25 25] ClipLimit 0.6')
```

NumTiles [25 25] ClipLimit 0.1

NumTiles [25 25] ClipLimit 0.2

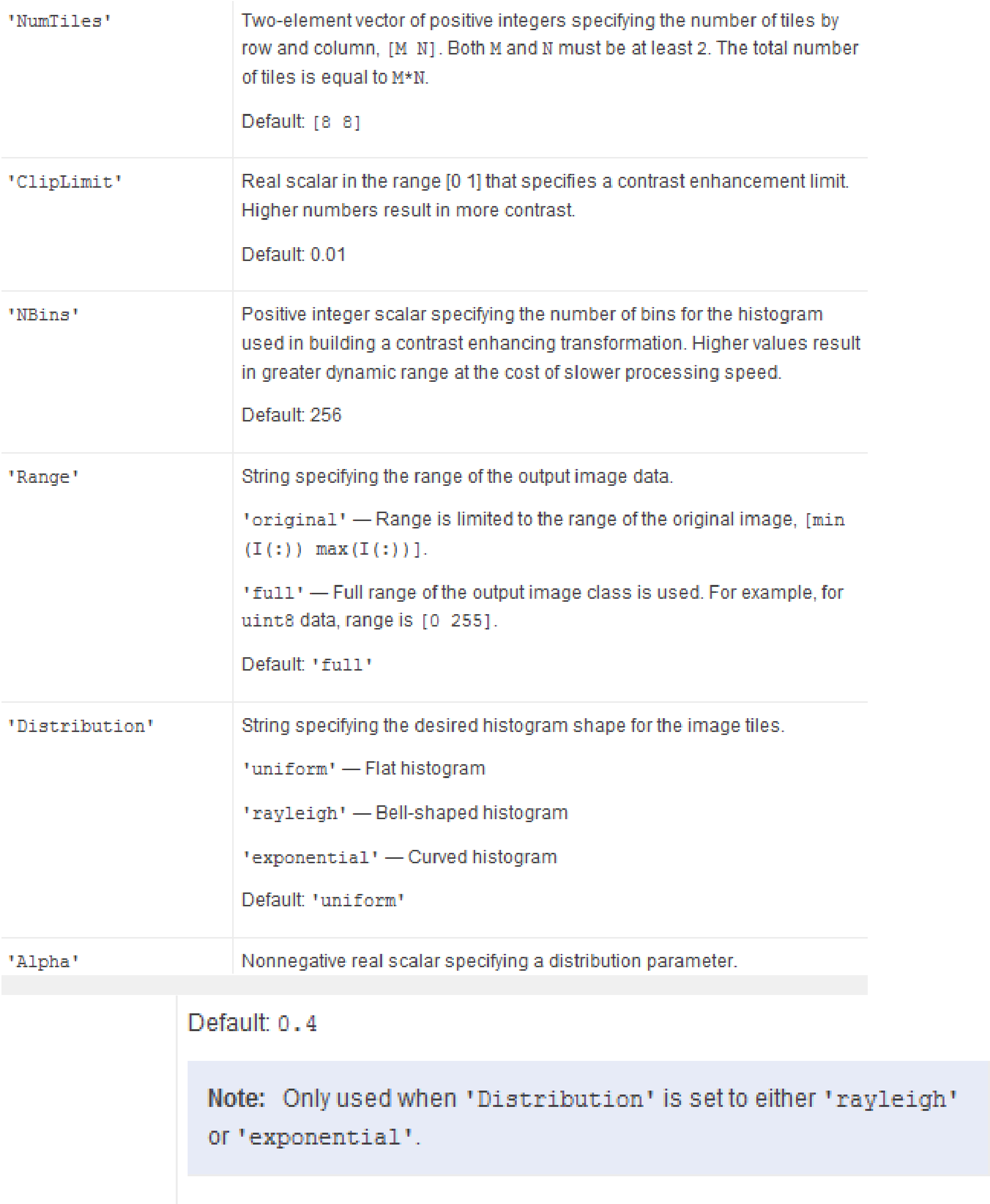
NumTiles [25 25] ClipLimit 0.3

NumTiles [25 25] ClipLimit 0.4

NumTiles [25 25] ClipLimit 0.5

NumTiles [25 25] ClipLimit 0.6

You can see that the contrast improves as we add more parameters. You can explore other parameters by typing `help adapthisteq` on the command prompt.



`'NumTiles'`	Two-element vector of positive integers specifying the number of tiles by row and column, `[M N]`. Both `M` and `N` must be at least 2. The total number of tiles is equal to `M*N`. Default: `[8 8]`
`'ClipLimit'`	Real scalar in the range [0 1] that specifies a contrast enhancement limit. Higher numbers result in more contrast. Default: 0.01
`'NBins'`	Positive integer scalar specifying the number of bins for the histogram used in building a contrast enhancing transformation. Higher values result in greater dynamic range at the cost of slower processing speed. Default: 256
`'Range'`	String specifying the range of the output image data. `'original'` — Range is limited to the range of the original image, `[min(I(:)) max(I(:))]`. `'full'` — Full range of the output image class is used. For example, for `uint8` data, range is `[0 255]`. Default: `'full'`
`'Distribution'`	String specifying the desired histogram shape for the image tiles. `'uniform'` — Flat histogram `'rayleigh'` — Bell-shaped histogram `'exponential'` — Curved histogram Default: `'uniform'`
`'Alpha'`	Nonnegative real scalar specifying a distribution parameter. Default: `0.4` **Note:** Only used when `'Distribution'` is set to either `'rayleigh'` or `'exponential'`.

3 Spatial Filtering

Spatial filtering follows the following steps:

1) Defining a centre point (x,y)

2) Performing an operation that involves only the pixels in predefined neighbourhood

3) Letting the result of that operation be response of the process at that point

4) Repeating the process for every point in the image.

The process of moving the center creates new neighborhood, one for each pixel in the input image. The two terms used for this process are neighborhood operation and spatial filtering.

3.1 Linear Filtering

If the computations performed on the neighborhood of a pixel are linear, it is called linear filtering. The linear operations of interest in this experiment are limited to multiplying each pixel in the neighborhood with a weight and summing the result to obtain the response at point (x,y). The weights are arranged in the form of a matrix also called as **filter, mask, kernel, template or window**. The filtering process consists of simply moving the center of the filter mask w, from point to point in the image f (x, y). At each point (x,y), the response of the filter at that point is the sum of the products of the filter weights and the corresponding neighborhood pixels in the area spanned by the filter mask.

There are two closely related concepts with linear filtering, correlation and convolution. Correlation is the process of moving the filter mask over the image and calculating sum of products at each location as described previously. The mechanics of convolution are exactly the same as correlation except that the filter is first rotated by 180 degree.

The toolbox implements the linear filtering using the following function

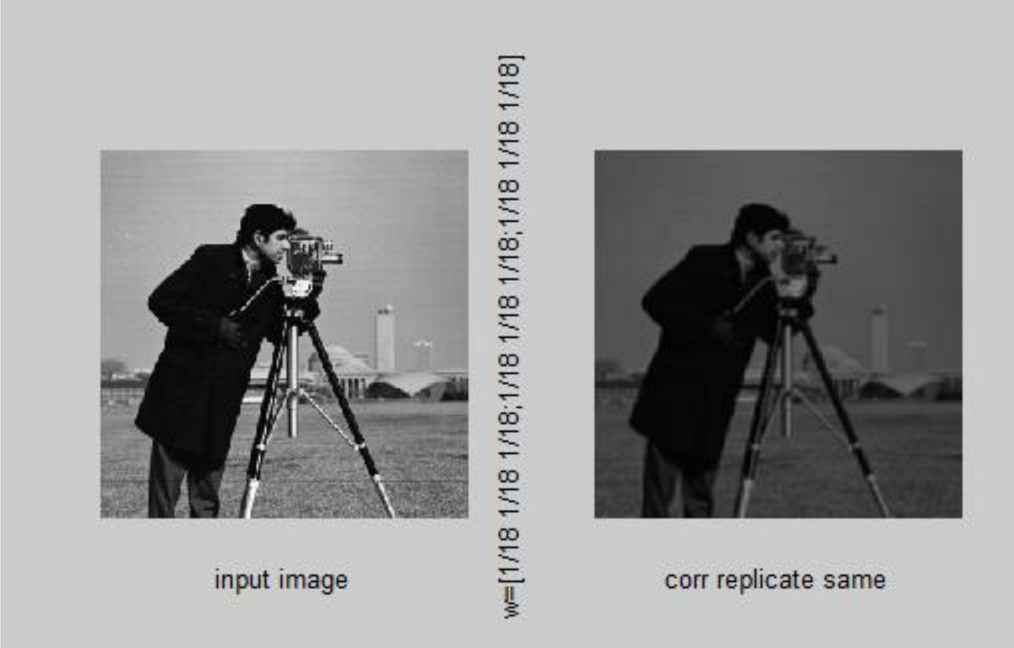
```
g = imfilter(f, w, filteringmode, boundaryoptions, sizeoptions);
```

Where f is the input image, w is the filtering mask and g is the filtered result. Filtering mode is used to specify whether to do correlation ('corr') or convolution ('conv'). The boundary option is about how to pad the boundary of image ('replicate','circular','symmetric',P) while filtering and the size option is used either to save the full output image (padded)('full') or after truncating the padded values ('same').

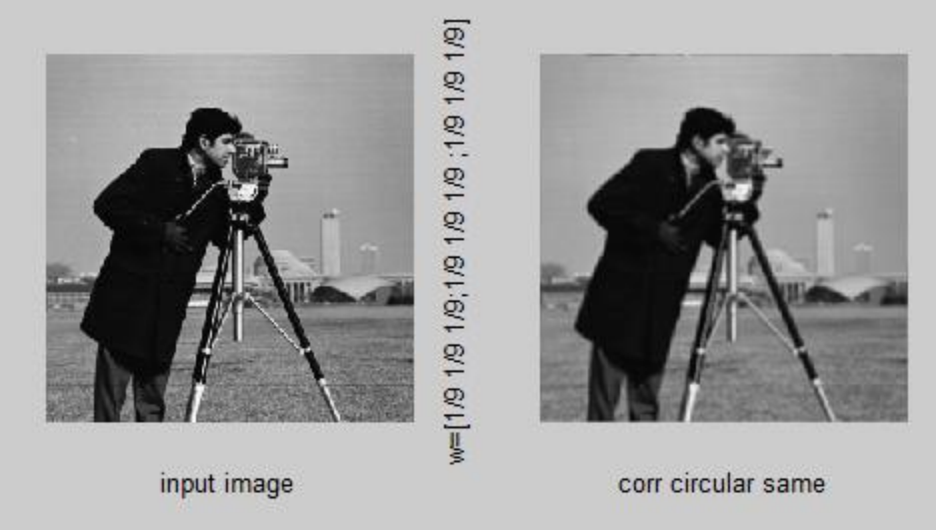
A popular application of linear filtering is blurring which can be achieved using averaging filter. Averaging can be achieved from correlation where after taking sum of products we can divide the result by number of elements of mask. It can also be achieved by having a mask such that each element of the mask is equal to 1/(number of elements of mask).

```
clc
f=imread('cameraman.tif');
w=[1/9 1/9 1/9;1/9 1/9 1/9;1/9 1/9 1/9];
g=imfilter(f,w,'corr','replicate','same');
subplot(1,2,1)
imshow(f)
xlabel('input image')
subplot(1,2,2)
imshow(g)
xlabel('corr replicate same');
```

```
clc
f=imread('cameraman.tif');
w=[1/18 1/18 1/18;1/18 1/18 1/18;1/18 1/18 1/18];
g=imfilter(f,w,'corr','replicate','same');
subplot(1,2,1)
imshow(f)
xlabel('input image')
subplot(1,2,2)
imshow(g)
xlabel('corr replicate same');
ylabel('w=[1/18 1/18 1/18;1/18 1/18 1/18;1/18 1/18 1/18]');
```



```
clc
f=imread('cameraman.tif');
w=[1/9 1/9 1/9;1/9 1/9 1/9;1/9 1/9 1/9];
g=imfilter(f,w,'corr','circular','same');
subplot(1,2,1)
imshow(f)
xlabel('input image')
subplot(1,2,2)
imshow(g)
xlabel('corr circular same');
ylabel('w=[1/9 1/9 1/9;1/9 1/9 1/9 ;1/9 1/9 1/9]')
```



```
clc
f=imread('cameraman.tif');
w=[1/9 1/9 1/9;1/9 1/9 1/9;1/9 1/9 1/9];
g=imfilter(f,w,'corr','symmetric','same');
subplot(1,2,1)
imshow(f)
xlabel('input image')
subplot(1,2,2)
imshow(g)
xlabel('corr symmetric same');
ylabel('w=[1/9 1/9 1/9;1/9 1/9 1/9 ;1/9 1/9 1/9]')
```

```
clc
f=imread('cameraman.tif');
w=[1/9 1/9 1/9;1/9 1/9 1/9;1/9 1/9 1/9];
g=imfilter(f,w,'corr', [0],'same');
subplot(1,2,1)
imshow(f)
xlabel('input image')
subplot(1,2,2)
imshow(g)
xlabel('corr [0] same');
ylabel('w=[1/9 1/9 1/9;1/9 1/9 1/9 ;1/9 1/9 1/9]')
```

```
clc
f=imread('cameraman.tif');
w=[1/9 1/9 1/9;1/9 1/9 1/9;1/9 1/9 1/9];
g=imfilter(f,w,'corr', [1],'same');
subplot(1,2,1)
imshow(f)
xlabel('input image')
subplot(1,2,2)
```

```
imshow(g)
xlabel('corr [1] same');
ylabel('w=[1/9 1/9 1/9;1/9 1/9 1/9 ;1/9 1/9 1/9]')
```

```
clc
f=imread('cameraman.tif');
w=[1/9 1/9 1/9;1/9 1/9 1/9;1/9 1/9 1/9];
g=imfilter(f,w,'corr', [128],'same');
subplot(1,2,1)
imshow(f)
xlabel('input image')
subplot(1,2,2)
imshow(g)
xlabel('corr [128] same');
ylabel('w=[1/9 1/9 1/9;1/9 1/9 1/9 ;1/9 1/9 1/9]')
```

```
clc
f=imread('cameraman.tif');
w=[1/9 1/9 1/9;1/9 1/9 1/9;1/9 1/9 1/9];
g=imfilter(f,w,'corr', [255],'same');
subplot(1,2,1)
imshow(f)
xlabel('input image')
subplot(1,2,2)
imshow(g)
xlabel('corr [255] same');
ylabel('w=[1/9 1/9 1/9;1/9 1/9 1/9 ;1/9 1/9 1/9]')
```

```
clc
f=imread('cameraman.tif');
w=[1/9 1/9 1/9;1/9 1/9 1/9;1/9 1/9 1/9];
g=imfilter(f,w,'corr', [255],'full');
subplot(1,2,1)
imshow(f)
xlabel('input image')
subplot(1,2,2)
imshow(g)
xlabel('corr [255] full');
ylabel('w=[1/9 1/9 1/9;1/9 1/9 1/9 ;1/9 1/9 1/9]')
```

```
clc
f=imread('cameraman.tif');
w=[1/9 1/9 1/9;1/9 1/9 1/9;1/9 1/9 1/9];
g=imfilter(f,w,'corr', [0],'full');
subplot(1,2,1)
imshow(f)
xlabel('input image')
subplot(1,2,2)
imshow(g)
xlabel('corr [0] full');
ylabel('w=[1/9 1/9 1/9;1/9 1/9 1/9 ;1/9 1/9 1/9]')
```

```
clc
f=imread('cameraman.tif');
w=[1/9 1/9 1/9;1/9 1/9 1/9;1/9 1/9 1/9];
g=imfilter(f,w,'conv', 'replicate' ,'same');
subplot(1,2,1)
imshow(f)
xlabel('input image')
subplot(1,2,2)
imshow(g)
xlabel('conv replicate same');
ylabel('w=[1/9 1/9 1/9;1/9 1/9 1/9 ;1/9 1/9 1/9]')
```

```
clc
f=imread('cameraman.tif');
w=[1/9 1/9 1/9;1/9 1/9 1/9;1/9 1/9 1/9];
g=imfilter(f,w,'conv', 'circular' ,'same');
subplot(1,2,1)
imshow(f)
xlabel('input image')
subplot(1,2,2)
imshow(g)
xlabel('conv circular same');
ylabel('w=[1/9 1/9 1/9;1/9 1/9 1/9 ;1/9 1/9 1/9]')
```

```
clc
f=imread('cameraman.tif');
w=[1/9 1/9 1/9;1/9 1/9 1/9;1/9 1/9 1/9];
g=imfilter(f,w,'conv', 'symmetric' ,'same');
subplot(1,2,1)
imshow(f)
xlabel('input image')
subplot(1,2,2)
imshow(g)
xlabel('conv symmetric same');
ylabel('w=[1/9 1/9 1/9;1/9 1/9 1/9 ;1/9 1/9 1/9]')
```

```
clc
f=imread('cameraman.tif');
w=[1/9 1/9 1/9;1/9 1/9 1/9;1/9 1/9 1/9];
g=imfilter(f,w,'conv', [0] ,'same');
subplot(1,2,1)
imshow(f)
xlabel('input image')
subplot(1,2,2)
imshow(g)
xlabel('conv [0] same');
ylabel('w=[1/9 1/9 1/9;1/9 1/9 1/9 ;1/9 1/9 1/9]')
```

```
clc
```

```
f=imread('cameraman.tif');
w=[1/25 1/25 1/25 1/25 1/25;1/25 1/25 1/25 1/25 1/25;1/25 1/25 1/25 1/25 1/25];
g=imfilter(f,w,'conv', [0] ,'same');
subplot(1,2,1)
imshow(g)
xlabel('conv [0] same')
h=imfilter(f,w,'corr',[0],'same')
subplot(1,2,2)
imshow(h)
xlabel('corr [0] same');
ylabel('[1/25 1/25 1/25 1/25 1/25;1/25 1/25 1/25 1/25 1/25;1/25 1/25 1/25 1/25 1/25]')
```

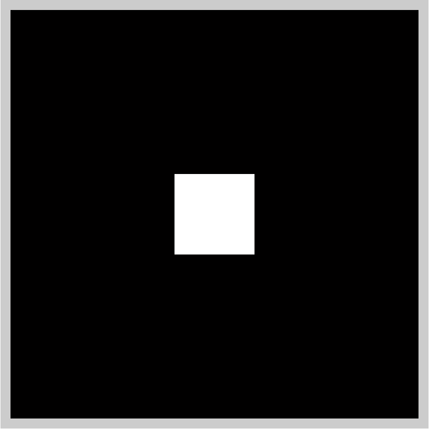
EXERCISE QUESTIONS

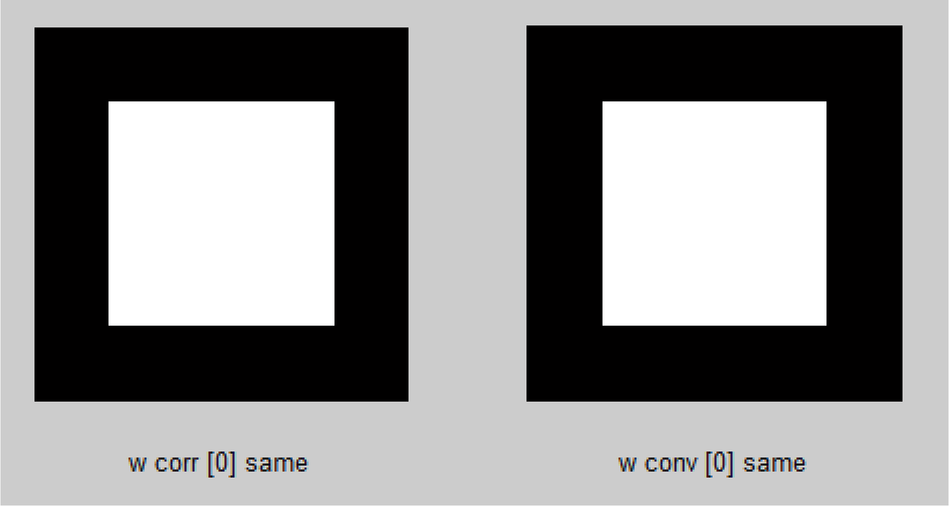
Q1 Make a matrix of size 5x5 such that only the center pixel is 1 and all other values are zero. Now make a filter of size 3x3 with values from 1 to 9 (as discussed during lecture). Now use MATLAB filtering function to do convolution and correlation. Also explore the parameter size options. Do the results match with theory?

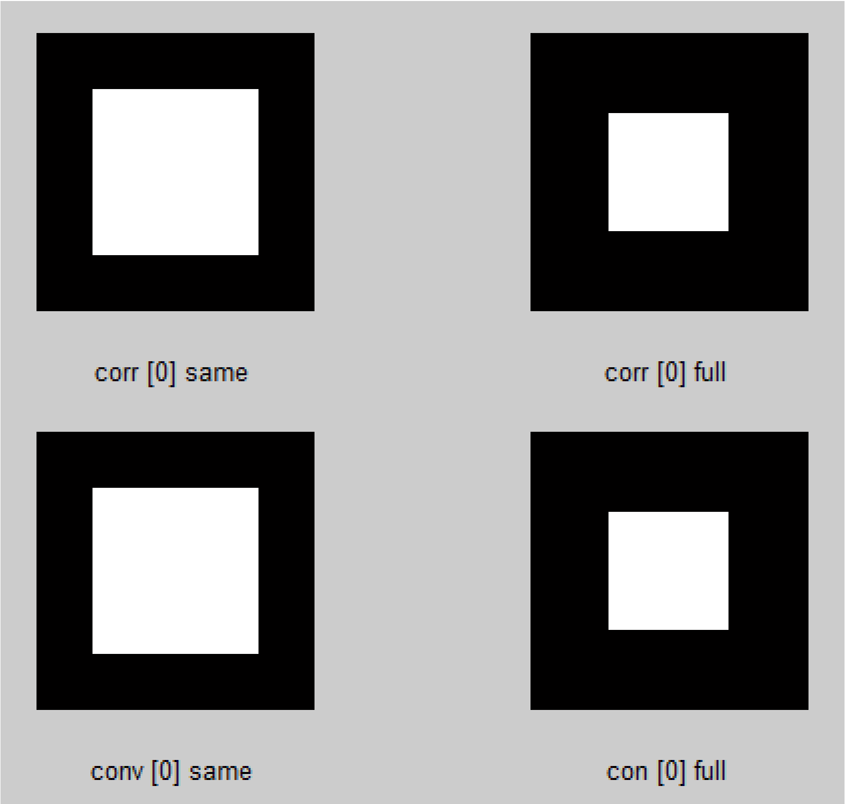
```
clc
I=zeros(5);
siz=size(I);
```

```
for i=1:siz(1)
    for j=1:siz(2)
        if (i==(siz(1)+1)/2 & j==(siz(2)+1)/2 )
            img(i,j)=1;
        else
            img(i,j)=0;
        end
    end
end
im_size=size(img);
imshow(img, 'InitialMagnification', 800);
w=[1 2 3;4 5 6; 7 8 9];
g=imfilter(img,w,'corr',[0],'same');
figure
subplot(1,2,1)
imshow(g)
xlabel('w corr [0] same')
h=imfilter(img,w,'conv',[0],'same');
subplot(1,2,2)
imshow(h)
xlabel('w conv [0] same')

figure
subplot(2,2,1)
gs=imfilter(img,w,'corr',[0],'same');
imshow(gs)
xlabel('corr [0] same')
subplot(2,2,2)
gf=imfilter(img,w,'corr',[0],'full');
imshow(gf)
xlabel('corr [0] full')
subplot(2,2,3)
```

```
hs=imfilter(img,w,'conv',[0],'same');
imshow(hs)
xlabel('conv [0] same')
subplot(2,2,4)
hf=imfilter(img,w,'conv',[0],'full')
imshow(hf)
xlabel('con [0] full')
```



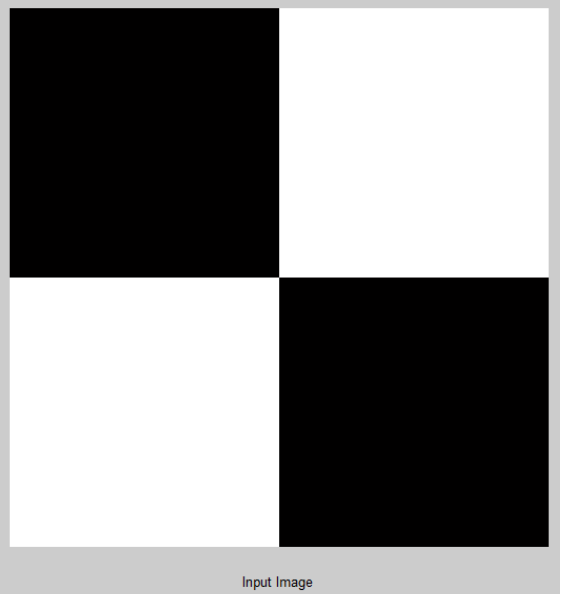


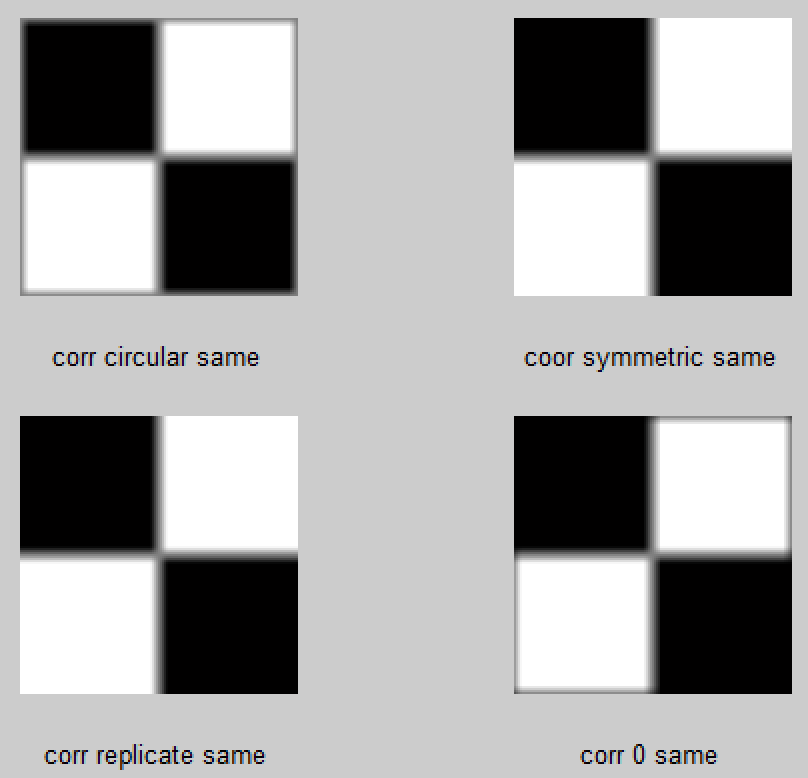


Q2 Create an images f of size 512x512 such that if this image is divided into four squares, the first and last square should be black while second and third squares should be white (Hint: use MATLAB built in functions zeros and ones to create the image). Now Create an averaging mask w of size 31x31 (Hint: Create a matrix of ones of size 31x31. Now divide it by 31^2). Apply the MATLAB filtering operation on the image with different value s of boundary options. Plot all the images on the same figure. What difference do you see with different values of boundary option?

```
clc
x=512/2
img=[zeros(x,x),ones(x,x);ones(x,x),zeros(x,x)];
imshow(img);
xlabel('Input Image')
w=(ones(31,31))./((31)^2);
figure
subplot(2,2,1)
g1=imfilter(img,w,'corr','circular','same');
imshow(g1)
xlabel('corr circular same')
```

```
subplot(2,2,2)
g2=imfilter(img,w,'corr','symmetric','same');
imshow(g2)
xlabel('coor symmetric same')

subplot(2,2,3)
g3=imfilter(img,w,'corr','replicate','same');
imshow(g3)
xlabel('corr replicate same')

subplot(2,2,4)
g4=imfilter(img,w,'corr',0,'same');
imshow(g4)
xlabel('corr 0 same')
```

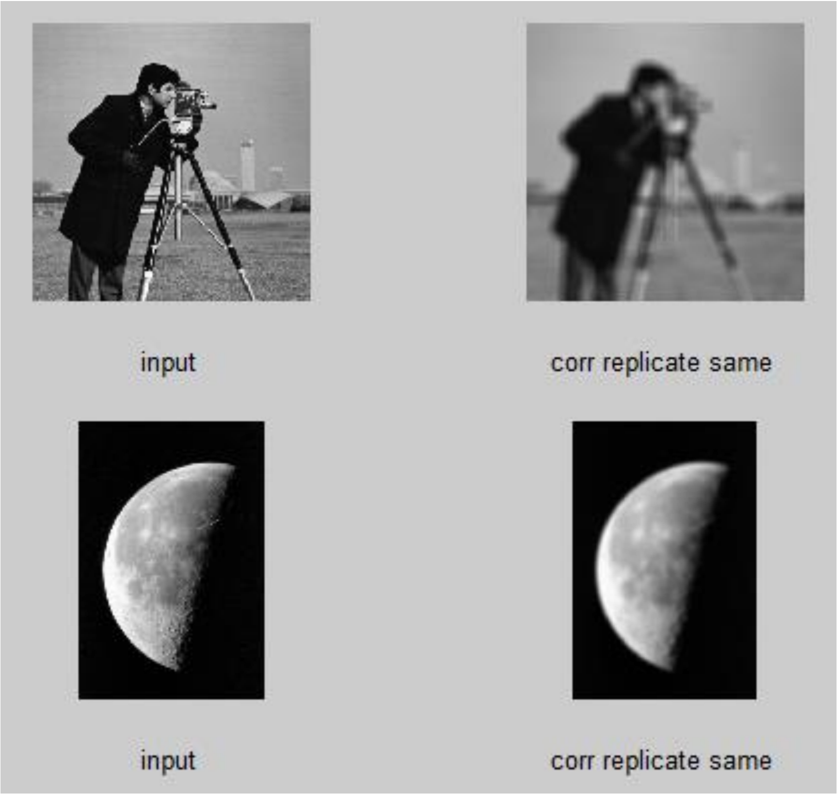




Q3 Now read image moon.tif and cameraman.tif . Make an averaging filter of size 11x11. Apply this filter on both the images and show all four images on same figure. Are the new images blurred?

```
clc
f=imread('cameraman.tif');
i=imread('moon.tif');
w=ones(11,11) / (11)^2;

gf=imfilter(f,w,'corr','replicate','same');
gi=imfilter(i,w,'corr','replicate','same');

subplot(2,2,1)
imshow(f)
xlabel('input')

subplot(2,2,2)
imshow(gf)
xlabel('corr replicate same')
```

```
subplot(2,2,3)
imshow(i)
xlabel('input')

subplot(2,2,4)
imshow(gi)
xlabel('corr replicate same')
```



Experiment#7: To get familiar with Linear and Non-Linear Filtering

1. **Experiment Text**
    i. **Spatial Filtering**
2. **Lab Exercise**
3. **Exercise Questions**

1 Spatial Filtering

In the previous experiment we implemented the filtering operation by manually specifying our filter. MATLAB toolbox supports a number of predefined 2-D linear spatial filters, obtained by using function fspecial, which generates a filter mask w, using the syntax

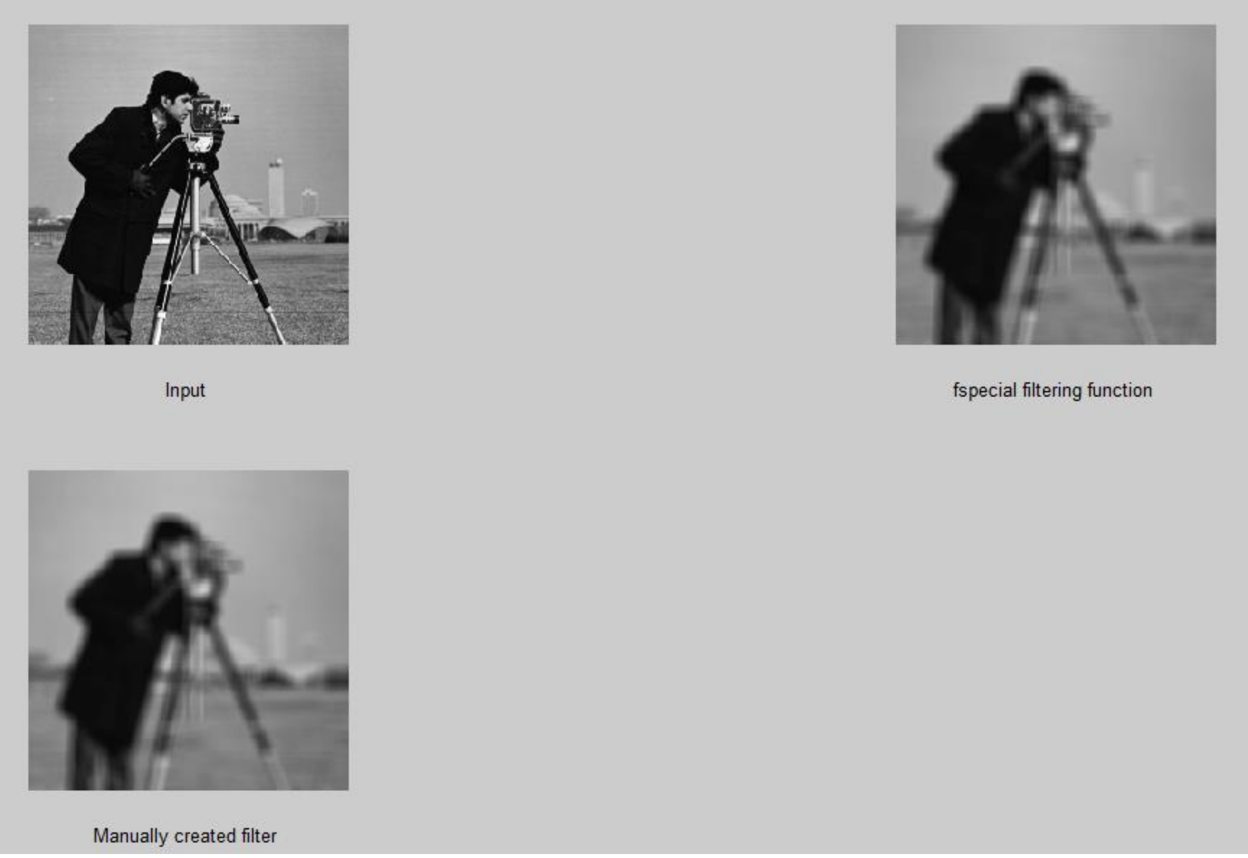
w = fspecial('type', parameters)

where 'type' specifies the filter type, and parameters further define the specified filter.The spatial filters supported by fspecial are given in Figure 1.

Read an image cameraman.tif (f) and create an averaging filter (w) of size 11x11 using fspecial. Create a blurred image (g) using this filter. Now create another another filter w1 of size 11x11 manually (as did in last experiment) and apply it on f to get a new image (g1). Plot f, g and g1 on the same figure. Do you see any difference between g and g1?

```
clc
f=imread('cameraman.tif');
w=fspecial('average',[11 11]);
g=imfilter(f,w,'corr','replicate','same');
w1=ones(11,11)/ 11^2;
 g1=imfilter(f,w1,'corr','replicate','same');
 subplot(2,2,1)
 imshow(f)
 xlabel('Input')
 subplot(2,2,2)
 imshow(g)
 xlabel('fspecial filtering function')
 subplot(2,2,3)
```

```
imshow(g1)
xlabel('Manually created filter')
```



An image can be sharpened using a Laplacian filter. The syntax for using a Laplacian filter is

$$w = fspecial('laplacian', \alpha)$$

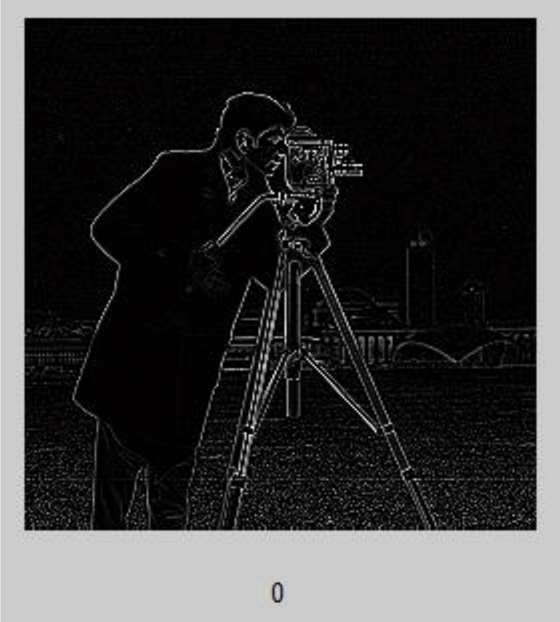
which creates a Laplacian filter of size 3x3? The shape of the filter determined by α whose values are between [0, 1] with a default value of 0.5. The filter is created using value of α by following formula.

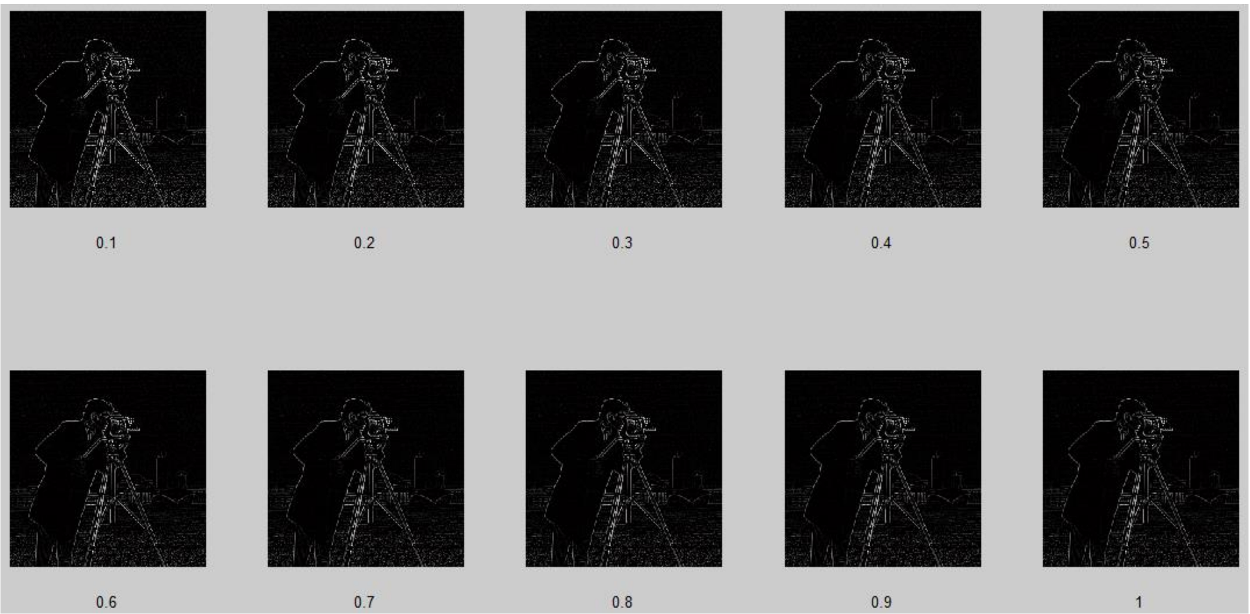
$$\begin{bmatrix} \frac{\alpha}{1+\alpha} & \frac{1-\alpha}{1+\alpha} & \frac{\alpha}{1+\alpha} \\ \frac{1-\alpha}{1+\alpha} & \frac{-4}{1+\alpha} & \frac{1-\alpha}{1+\alpha} \\ \frac{\alpha}{1+\alpha} & \frac{1-\alpha}{1+\alpha} & \frac{\alpha}{1+\alpha} \end{bmatrix}$$

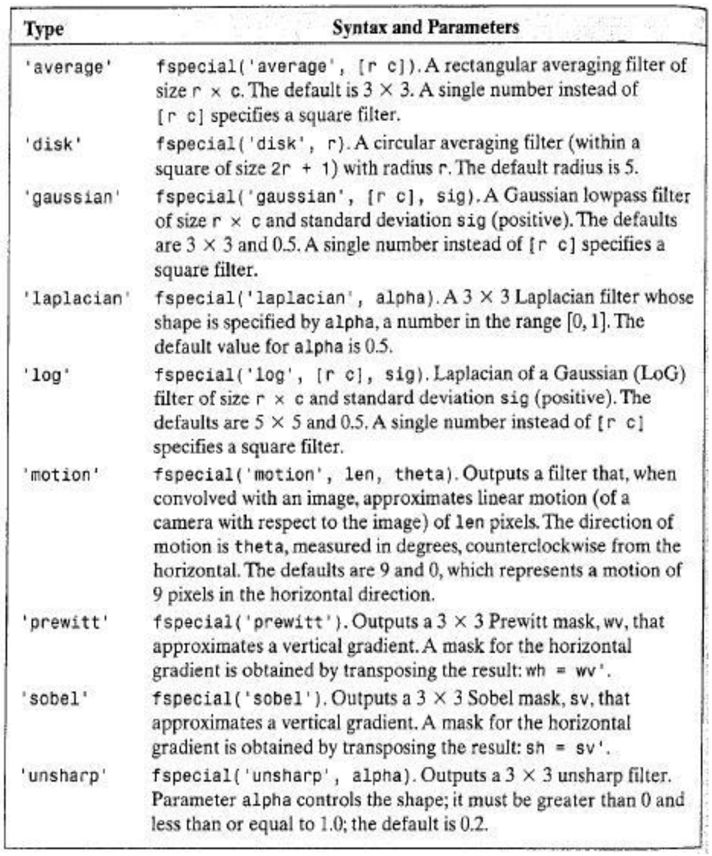
```
clc
imread('cameraman.tif');
w=fspecial('laplacian',0);
```

```
f0=imfilter(f,w,'corr','replicate','same');
imshow(f0)
xlabel('0')
figure
f1=imfilter(f,fspecial('laplacian',0.1),'corr','replicate','same');
f2=imfilter(f,fspecial('laplacian',0.2),'corr','replicate','same');
f3=imfilter(f,fspecial('laplacian',0.3),'corr','replicate','same');
f4=imfilter(f,fspecial('laplacian',0.4),'corr','replicate','same');
f5=imfilter(f,fspecial('laplacian',0.5),'corr','replicate','same');
f6=imfilter(f,fspecial('laplacian',0.6),'corr','replicate','same');
f7=imfilter(f,fspecial('laplacian',0.7),'corr','replicate','same');
f8=imfilter(f,fspecial('laplacian',0.8),'corr','replicate','same');
f9=imfilter(f,fspecial('laplacian',0.9),'corr','replicate','same');
f10=imfilter(f,fspecial('laplacian',1),'corr','replicate','same');
subplot(2,5,1)
imshow(f1)
xlabel('0.1')
subplot(2,5,2)
imshow(f2)
xlabel('0.2')
subplot(2,5,3)
imshow(f3)
xlabel('0.3')
subplot(2,5,4)
imshow(f4)
xlabel('0.4')
subplot(2,5,5)
imshow(f5)
xlabel('0.5')
subplot(2,5,6)
imshow(f6)
xlabel('0.6')
```

```
subplot(2,5,7)
imshow(f7)
xlabel('0.7')
subplot(2,5,8)
imshow(f8)
xlabel('0.8')
subplot(2,5,9)
imshow(f9)
xlabel('0.9')
subplot(2,5,10)
imshow(f10)
xlabel('1')
```







Type	Syntax and Parameters
'average'	fspecial('average', [r c]). A rectangular averaging filter of size r × c. The default is 3 × 3. A single number instead of [r c] specifies a square filter.
'disk'	fspecial('disk', r). A circular averaging filter (within a square of size 2r + 1) with radius r. The default radius is 5.
'gaussian'	fspecial('gaussian', [r c], sig). A Gaussian lowpass filter of size r × c and standard deviation sig (positive). The defaults are 3 × 3 and 0.5. A single number instead of [r c] specifies a square filter.
'laplacian'	fspecial('laplacian', alpha). A 3 × 3 Laplacian filter whose shape is specified by alpha, a number in the range [0, 1]. The default value for alpha is 0.5.
'log'	fspecial('log', [r c], sig). Laplacian of a Gaussian (LoG) filter of size r × c and standard deviation sig (positive). The defaults are 5 × 5 and 0.5. A single number instead of [r c] specifies a square filter.
'motion'	fspecial('motion', len, theta). Outputs a filter that, when convolved with an image, approximates linear motion (of a camera with respect to the image) of len pixels. The direction of motion is theta, measured in degrees, counterclockwise from the horizontal. The defaults are 9 and 0, which represents a motion of 9 pixels in the horizontal direction.
'prewitt'	fspecial('prewitt'). Outputs a 3 × 3 Prewitt mask, wv, that approximates a vertical gradient. A mask for the horizontal gradient is obtained by transposing the result: wh = wv'.
'sobel'	fspecial('sobel'). Outputs a 3 × 3 Sobel mask, sv, that approximates a vertical gradient. A mask for the horizontal gradient is obtained by transposing the result: sh = sv'.
'unsharp'	fspecial('unsharp', alpha). Outputs a 3 × 3 unsharp filter. Parameter alpha controls the shape; it must be greater than 0 and less than or equal to 1.0; the default is 0.2.

Figure 1: Filter Types

1.1 Non Linear Filters

A commonly used tool for generating non linear spatial filters in MATLAB is to use function ordfilt2 which generates order static filters. These are non linear filters whose response is based on ordering the pixels contained in image neighbourhood and then replacing the value of the centre pixel in the neighbourhood with the value determined by the ranking result. The values are arranged in ascending order in this filter. The syntax of the function is

$$g = ordfilt2(f, order, domain)$$

```
clc
clear
x=imread('cameraman.tif')
s=ones(11,11);
g=ordfilt2(x,100,s)
imshow(g)
```

This function creates a new image g by replacing each element of f by order’th element in the sorted set of neighbours specified by non zero elements in domain. Here domain is mxn mask of 1s and 0s which specify the pixel locations in the neighbourhood that are to be used. The pixels in the neighbourhood that correspond to 0s in the domain mask are not included in the computation e.g. to implement a min filter (first element/order 1/0th percentile), the syntax used is

$$g = ordfilt2(f, 1, ones(m, n));$$

```
clc
clear
x=imread('cameraman.tif')
s=ones(3,3);
g=ordfilt2(x,1,s)
imshow(g)
```

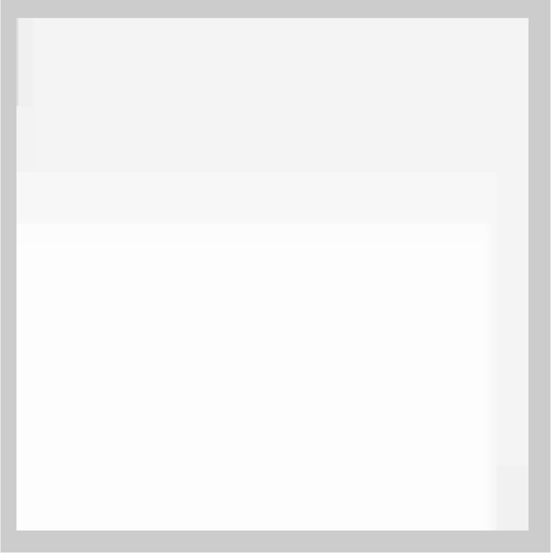
In this example 1 denotes the first element in the ordered set of mn pixels and ones(m, n) creates a mxn matrix consisting of 1s, indicating that all the elements of neighbourhood are to be used. Similarly to implement a max filter, the syntax used is

$$g = ordfilt2(f, m * n, ones(m, n));$$

```
clc
clear
x=imread('cameraman.tif')
s=ones(3,3);
[m n]=size(s);
g=ordfilt2(x,m*n,s)
imshow(g)
```

where m*n specifies the last element (100th percentile) in ordering e.g. if you are using 3x3 neighbourhood then m*n would mean 3*3 i.e. the 9th element in the ordering.

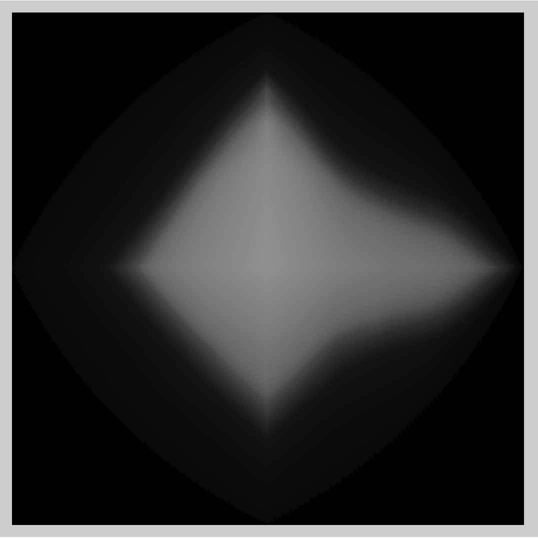
```
clc
clear
x=imread('cameraman.tif')
[m n]=size(x)
s=ones(m,n);
g=ordfilt2(x,m*n,s)
imshow(g)
```



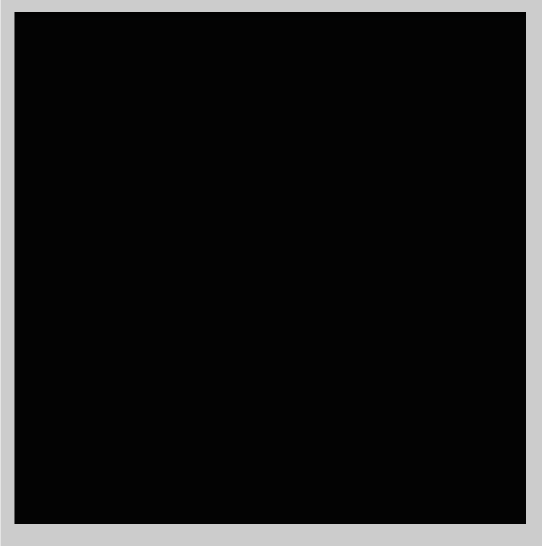
The best known order-static filter in image processing is the median filter which corresponds to 50th percentile.

$$g = ordfilt2(f, median(1:m*n), ones(m,n));$$

```
clc
clear
x=imread('cameraman.tif')
[m n]=size(x)
s=ones(m,n);
md=round((median(1:m*n)))
g=ordfilt2(x,md,s);
imshow(g)
```



```
clc
clear
x=imread('cameraman.tif')
[m n]=size(x)
s=ones(15,15);
md=round((median(1:m*n)))
g=ordfilt2(x,md,s);
imshow(g)
```



where median(1 : m ∗ n) computes the median of the neighbourhood. Because of its special importance, the toolbox provides a special function just for median filter.

$$g = medfilt2(f, [m\ n], padopt);$$

where the parameter [m n] defines the neighborhood over which median needs to be com-puted and padopt specifies the padding options. The default form of this function is

$$g = medfilt2(f);$$

which uses a 3x3 neighborhood and pads the borders with 0s?

```
clc

clear

x=imread('cameraman.tif')

g=medfilt2(x);

imshow(g)
```

```
clc
clear
x=imread('cameraman.tif')
g=medfilt2(x,[5 5]);
imshow(g)
```

Median filtering is widely used to remove salt and pepper noise e. Now we will see how median filtering performs for an image corrupted with salt & pepper noise. An image can be corrupted with salt & pepper noise by using function imnoise which has the following syntax

$$g = imnoise(f,' salt \ \& \ pepper', 0.2);$$

where the last value (0.2) controls the amount of noise added to the image. The value of last parameter is between 0 and 1 where a higher value means more noise.

```
clc
clear
i=imread('cameraman.tif');
g=imnoise(i,'salt & pepper',0.2);
 g1=medfilt2(g);
subplot(1,3,1)
imshow(i)
xlabel('Input Image')
subplot(1,3,2)
imshow(g)
xlabel('Noisy Image')
subplot(1,3,3)
imshow(g1)
xlabel('Removed Noise')
```

MATLAB EXERCISE

EXERCISE QUESTIONS

Q1 Make a script file. Now write the code to read image *moon.tif* and *cameraman.tif* . Make a Laplacian filter with $\alpha = 0$. Apply this filter on both the images to get derivative images . Now subtract the derivative images from original images to get sharpened images and show four images on two figures (one figure for *moon.tif* and *cameraman.tif* each). Each figure should have original and sharpened image. Do you see the sharpness?

```
clc
clear
f=imread('cameraman.tif');
g=imread('moon.tif');
w=fspecial('laplacian',0);
g1=imfilter(g,w,'corr','replicate','same');
f1=imfilter(f,w,'corr','replicate','same');
f_sharp=f-f1;
g_sharp=g-g1;
figure
subplot(1,3,1)
imshow(f)
subplot(1,3,2)
imshow(f1)
subplot(1,3,3)
imshow(f_sharp)

figure
subplot(1,3,1)
imshow(g)
subplot(1,3,2)
imshow(g1)
subplot(1,3,3)
imshow(g_sharp)
```

Contradiction

```
f_sharp=f+f1;
g_sharp=g+g1; %Theory
```

Q2 Now create a Laplacian mask such that the values of mask are same as in last experiment but the signs are reversed. Now apply this mask on both images and get sharpened image (by addition). Show three images (original, with old mask, with new mask) on two figures (for both images). Do you see any improvement with new mask?

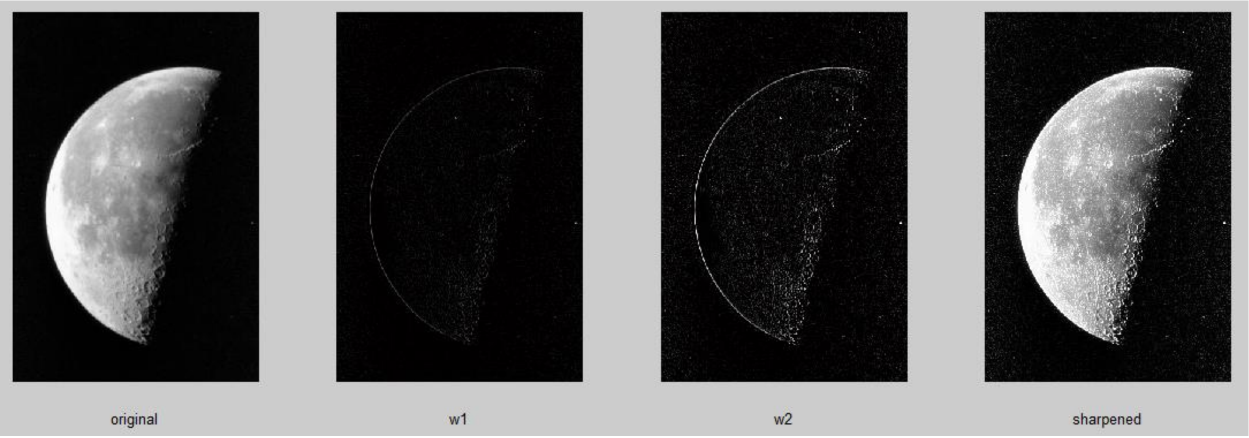
```
clc
clear
f=imread('cameraman.tif');
g=imread('moon.tif');
w1=fspecial('laplacian',0);
w2=-w1;
g1=imfilter(g,w1,'corr','replicate','same');
f1=imfilter(f,w1,'corr','replicate','same');
g2=imfilter(g,w2,'corr','replicate','same');
f2=imfilter(f,w2,'corr','replicate','same');
f_sharp=f+f2;
g_sharp=g+g2;
figure
subplot(1,4,1)
imshow(f)
xlabel('Input Image')
subplot(1,4,2)
```

```
imshow(f1)
xlabel('old mask')
subplot(1,4,3)
imshow(f2)
xlabel('new mask')
subplot(1,4,4)
imshow(f_sharp)
xlabel('sharpened')
figure
subplot(1,4,1)
imshow(g)
xlabel('original')
subplot(1,4,2)
imshow(g1)
xlabel('old mask')
subplot(1,4,3)
imshow(g2)
xlabel('new mask')
subplot(1,4,4)
imshow(g_sharp)
xlabel('sharpened')
w1 =
     0     1     0
     1    -4     1
     0     1     0
w2 =
     0    -1     0
    -1     4    -1
     0    -1     0
```

Q3 Now make two masks w1 = [0 -1 0 ; -1 4 -1 ; 0 -1 0] and w2 = [-1 -1 -1 ; -1 8 -1; -1 -1 -1]. Now apply this mask on both images and show original and sharpened images for both masks (on one figure) and for both images (on different figures). Which mask is better?

```
clc
clear
f=imread('cameraman.tif');
g=imread('moon.tif');
w1=[0 -1 0 ; -1 4 -1 ; 0 -1 0];
w2= [-1 -1 -1 ; -1 8 -1; -1 -1 -1];
g1=imfilter(g,w1,'corr','replicate','same');
f1=imfilter(f,w1,'corr','replicate','same');
g2=imfilter(g,w2,'corr','replicate','same');
f2=imfilter(f,w2,'corr','replicate','same');
f_sharp=f+f2;
g_sharp=g+g2;
figure
subplot(1,4,1)
```

```
imshow(f)
xlabel('Input Image')
subplot(1,4,2)
imshow(f1)
xlabel('w1')
subplot(1,4,3)
imshow(f2)
xlabel('w2')
subplot(1,4,4)
imshow(f_sharp)
xlabel('sharpened')
figure
subplot(1,4,1)
imshow(g)
xlabel('original')
subplot(1,4,2)
imshow(g1)
xlabel('w1')
subplot(1,4,3)
imshow(g2)
xlabel('w2')
subplot(1,4,4)
imshow(g_sharp)
xlabel('sharpened')
```



An image can also be made sharp by using unsharp filtering. In unsharp filtering we create a mask by subtracting the blurred image from original and then we add that mask to the original image to get a sharpened image.

Q4 Read an image *eight.tif* . Now create a blurred image using average filtering. Subtract the blurred image from original to get mask. Now add this mask to original image to get sharp image. Now show both original and sharp image

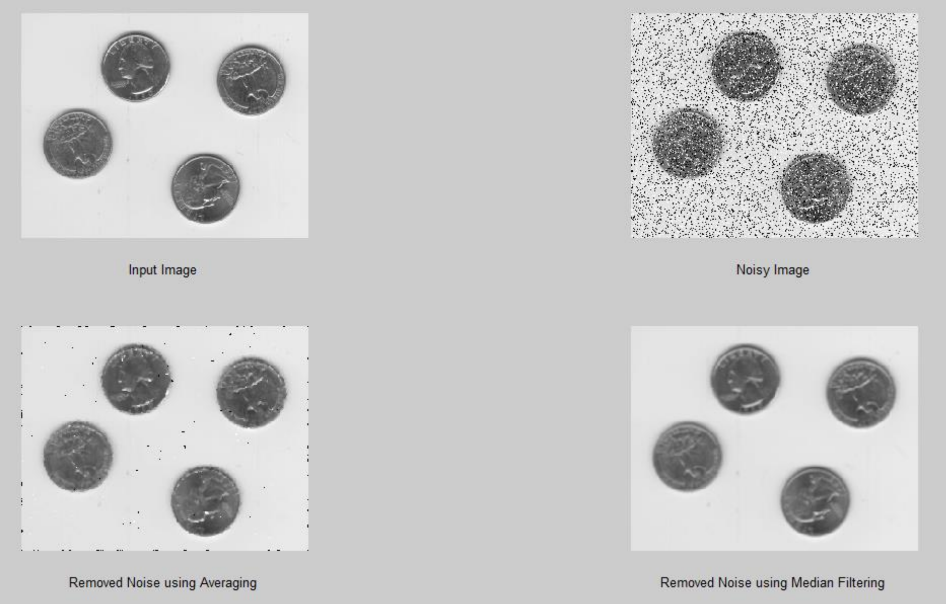
```
clc
clear
f=imread('eight.tif');
w=ones(3,3)/3^2;
g=imfilter(f,w,'corr','replicate','same');
edge=f-g;
sharp=f+edge;
subplot(2,2,1)
imshow(f)
xlabel('input Image');
subplot(2,2,2)
imshow(g)
xlabel('blur');
```

```
subplot(2,2,3)
imshow(edge)
xlabel('edge mask');
subplot(2,2,4)
imshow(sharp)
xlabel('sharpened');
```

Q5 Read an image eight.tif . Add salt & pepper noise to this image. Now apply both averaging and median filtering on the noisy image. Show original, noisy , average filtered and median filtered images on one figure. Which filter performs better. (Note: You can use the padopt as symmetric in medfilt2 to remove the noise from borders of image).

```
clc
clear
i=imread('eight.tif');
g=imnoise(i,'salt & pepper',0.2);
g1=medfilt2(g);
w=ones(3,3)/3^2;
g2=imfilter(i,w,'corr','symmetric','same')
subplot(2,2,1)
imshow(i)
xlabel('Input Image')
subplot(2,2,2)
```

```
imshow(g)
xlabel('Noisy Image')
subplot(2,2,3)
imshow(g1)
xlabel('Removed Noise using Averaging')
subplot(2,2,4)
imshow(g2)
xlabel('Removed Noise using Median Filtering')
```



Experiment#8: To get familiar with morphological operations

- **Experiment Text**
  - **i. Morphological Image Processing**
  - **ii. Dilation and Erosion**
- **Lab Exercise**
- **Exercise Questions**

EXPERIMENT TEXT:

1 Morphological Image Processing

The word morphology commonly deals with branch of biology that deals with form and structure of animals and plants. For image processing the term mathematical morphology deals with extracting image components that are useful in description and representation of a region shape, boundary, skeleton etc. Morphological processes also deal with filtering, thinning and pruning.

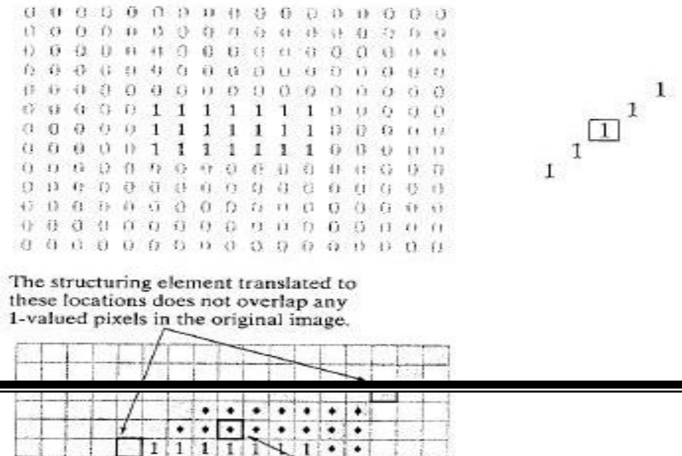
2 Dilation and Erosion

The operations dilation and erosion are fundamental to morphological image processing.

2.1 Dilation

Dilation is process that grows or thickens an object in an image. The specific manner and amount of thickness is controlled by structuring element (SE). Figure 1 shows an example of dilation. The top left part of the figure shows a simple binary image containing an object (object is represented by 1's).

The top right of figure shows a diagonal structuring element. SEs are normally represented as a combination of 1s and 0s in the form of a rectangle but sometimes we only show 1s if it is more convenient. Also the origin of the SE must also be specified and in the figure it is specified by a black outline. The middle part of the figure graphically depicts the dilation process as origin of SE moves through the image. The output image has 1 at each location of the origin such that the SE overlaps with atleast one 1-valued pixel of the input image.



The structuring element translated to these locations does not overlap any 1-valued pixels in the original image.

Mathematically dilation is defined in terms of set operations. The dilation of image A by structuring element B is written as $A \oplus B$ and is defined as

$$A \oplus B = \{z | (\hat{B})_z \cap A \neq \phi\}$$

In words, dilation of A by B is the set of all SE origin locations where the reflected and translated B overlaps atleast some portion of A. Since the SE in this case is symmetric about its origin the reflection does not change its shape.

MATLAB built in function for dilation is `imdilate` and its syntax is

$$f1 = imdilate(f, B)$$

```
f=imread('cameraman.tif');
```

```
se=[0 1 0;1 1 1;0 1 0];
d=imdilate(f,se);
subplot(1,2,1)
imshow(f);
xlabel('Input image')
subplot(1,2,2)
imshow(d)
xlabel('dilated')
```

where f, f1 are input and output binary images while B is a structuring element.

MATLAB has a built in function `strel` for creating SEs. This function has a general syntax

$$se = strel(shape, parameter);$$

where shape is a string specifying desired shape and parameters specify information about the shape. e.g. the shape could be diamond, disk, rectangle, square etc. More information about shapes and parameters can be found using help strel.

```
f=imread('cameraman.tif');
se=strel('rectangle',[3 3])
d=imdilate(f,se);
subplot(1,2,1)
```

```
imshow(f);
xlabel('Input image')
subplot(1,2,2)
imshow(d)
xlabel('dilated')
```

```
se1 = strel('square',11)      % 11-by-11 square
se2 = strel('line',10,45)     % line, length 10, angle 45 degrees
se3 = strel('disk',15)        % disk, radius 15
se4 = strel('ball',15,5)      % ball, radius 15, height 5
```

2.2 Erosion

Erosion shrinks or thins objects in an image. Figure 2 shows an example of erosion. In erosion, the output image has a value 1 if at each location of origin of structuring element, the elements of SE overlap with only 1-valued pixels of input image. The erosion of A by B denoted as $A \ominus B$ is defined as

$$A \ominus B = \{z | (B)_z \cap A^c \neq \phi\}$$

Erosion of A by B is the set of all structuring element origin locations where the translated B has no overlap with background of A i.e B completely fits within 1's of A. Erosion is performed by function imerode.

```
f=imread('cameraman.tif');
se=[0 1 0;1 1 1;0 1 0];
d=imerode(f,se);
```

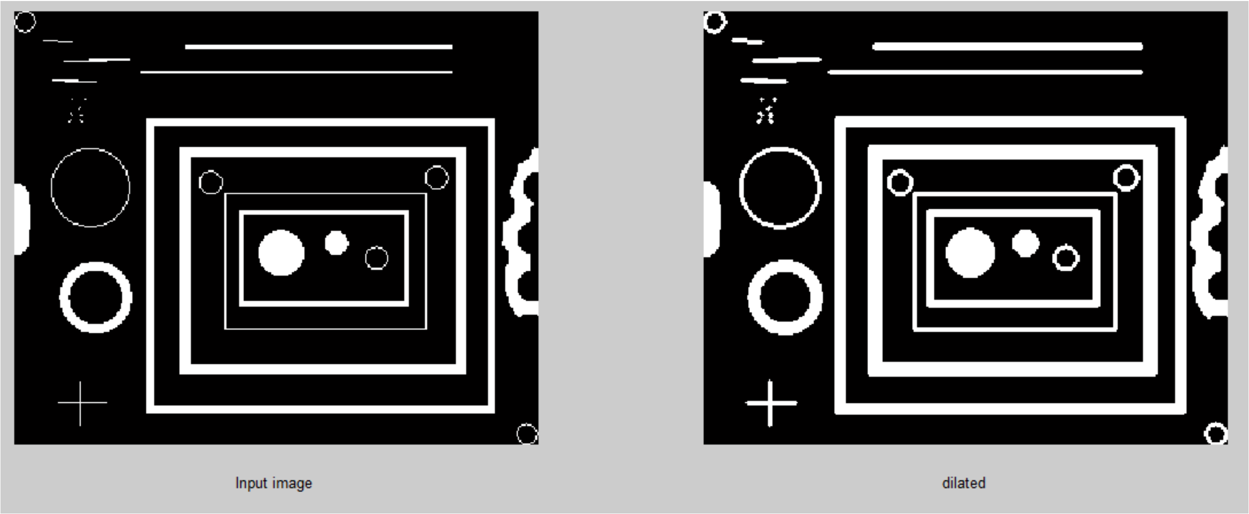
```
subplot(1,2,1)
imshow(f);
xlabel('Input image')
subplot(1,2,2)
imshow(d)
xlabel('Erosion')
```

```
f=imread('cameraman.tif');
se=strel('rectangle',[3 3])
d=imerode(f,se);
subplot(1,2,1)
imshow(f);
xlabel('Input image')
subplot(1,2,2)
imshow(d)
xlabel('Erosion')
```

EXERCISE QUESTIONS

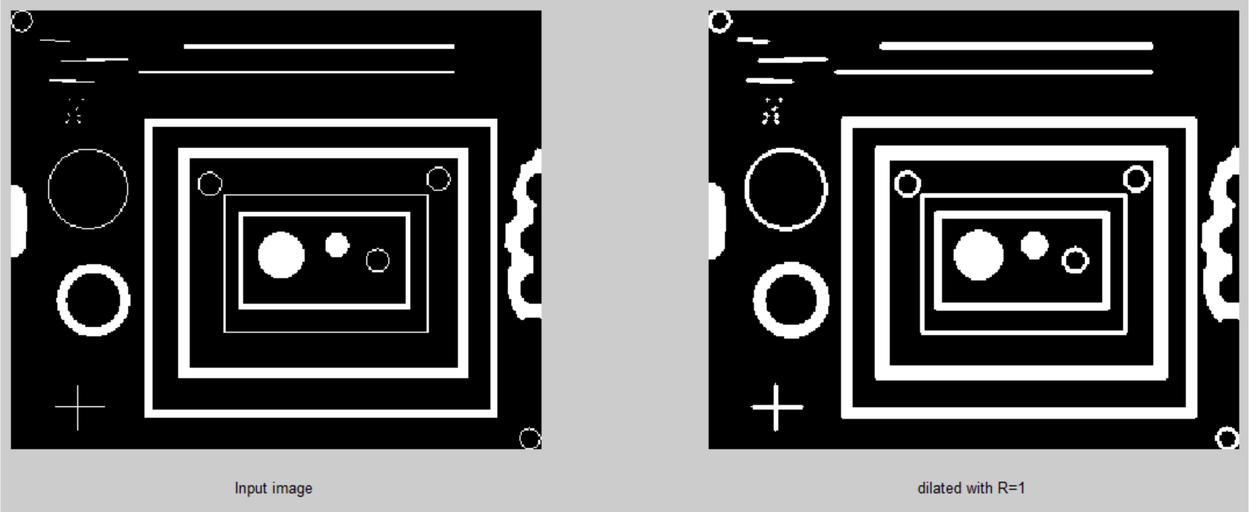
Q1 Read an image blobs.png and apply dilation using structuring element B = [0 1 0 ; 1 1 1 ; 0 1 0]. Now show both original and dilated image on one figure. Do you see thickening in the new image?

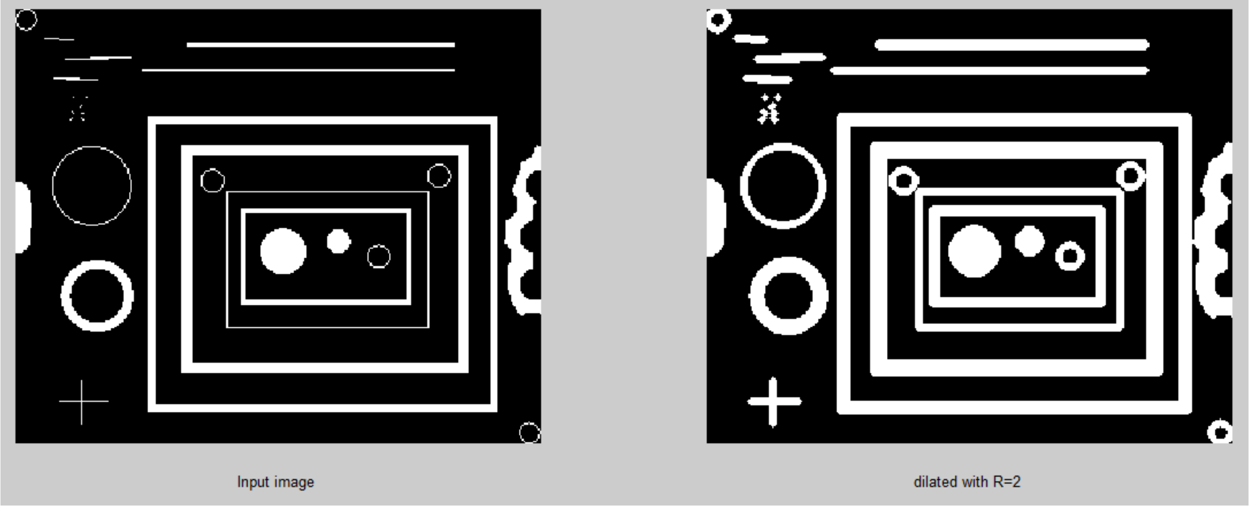
```
clc
clear
f=imread('blobs.png');
se=[0 1 0 ; 1 1 1 ; 0 1 0];
d=imdilate(f,se);
subplot(1,2,1)
imshow(f);
xlabel('Input image')
subplot(1,2,2)
imshow(d)
xlabel('dilated')
```

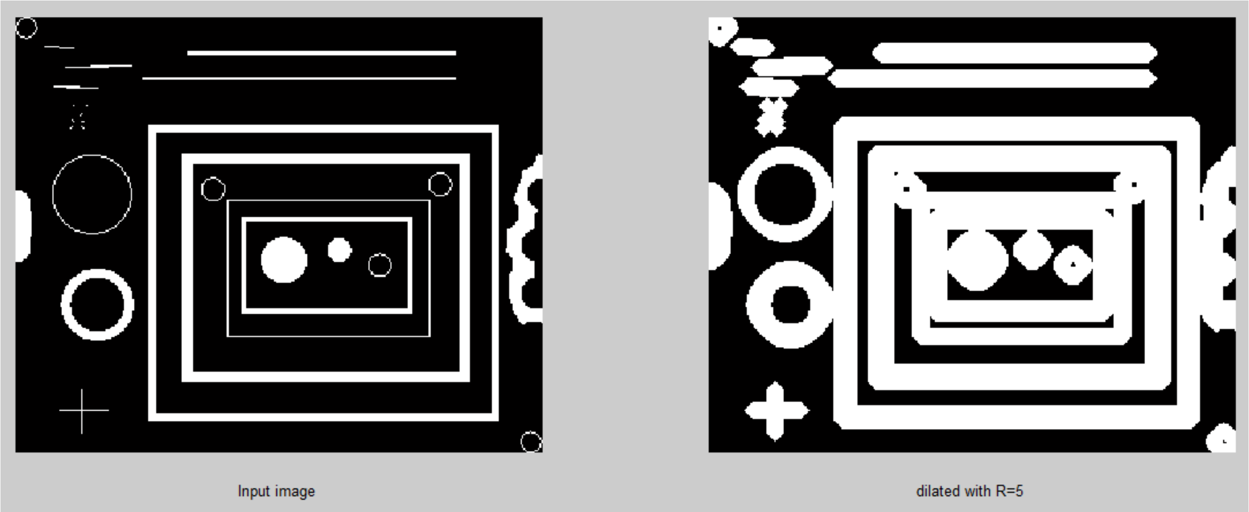


Q2 Read an image blobs.png and apply dilation using same structuring element as did in the last question but now SE should be defined using strel. (hint: use diamond shape). Now show both original and dilated image on one figure. You can change size of the structuring element to see the change in thickening.

```
clc
clear
f=imread('blobs.png');
se=strel('diamond',1)
d=imdilate(f,se);
subplot(1,2,1)
imshow(f);
xlabel('Input image')
subplot(1,2,2)
imshow(d)
xlabel('dilated with R=1')
```

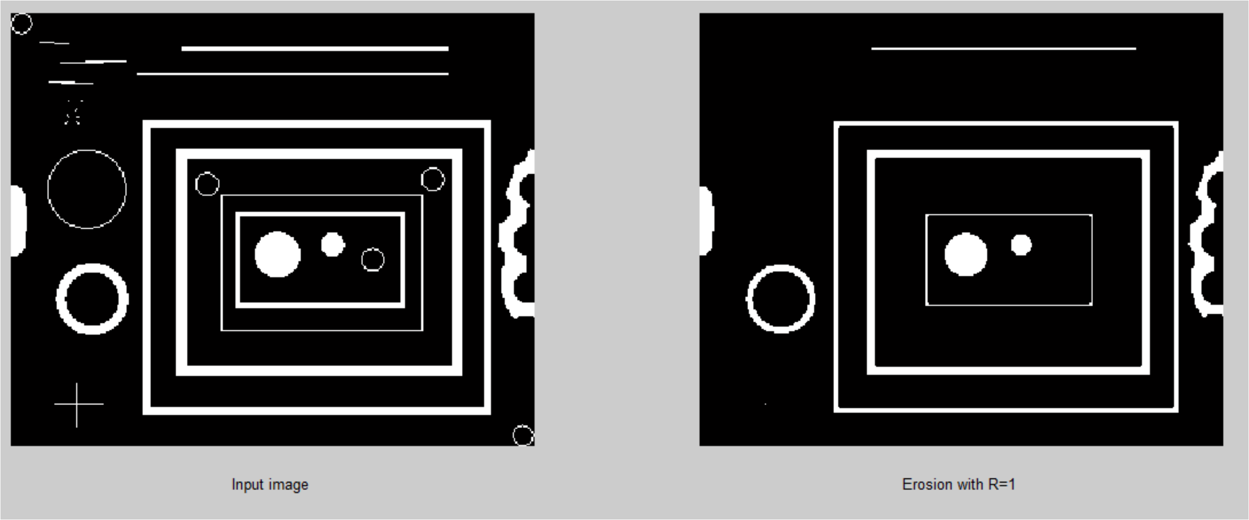


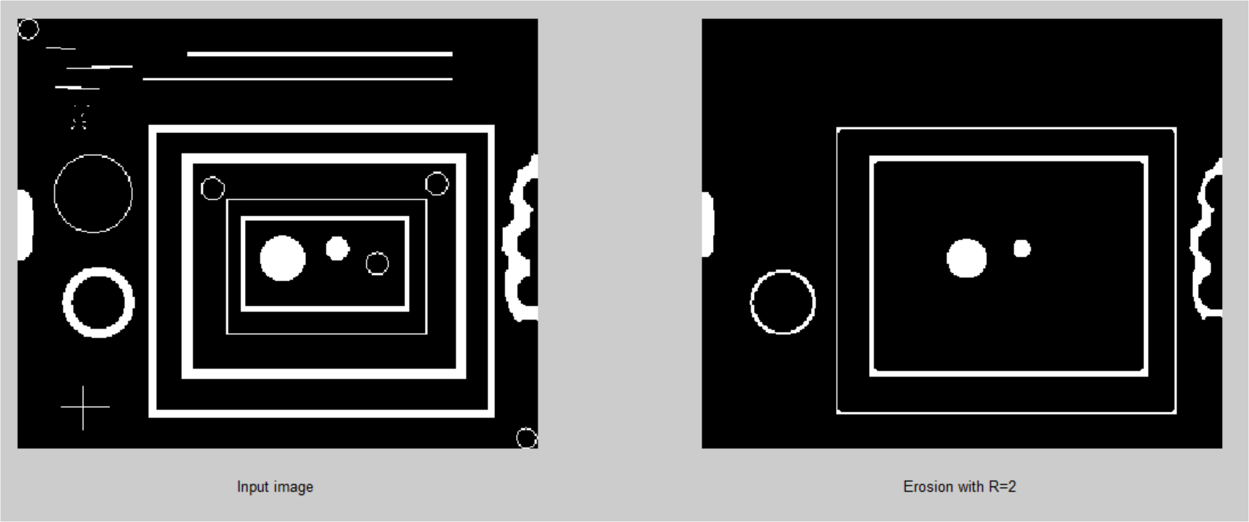


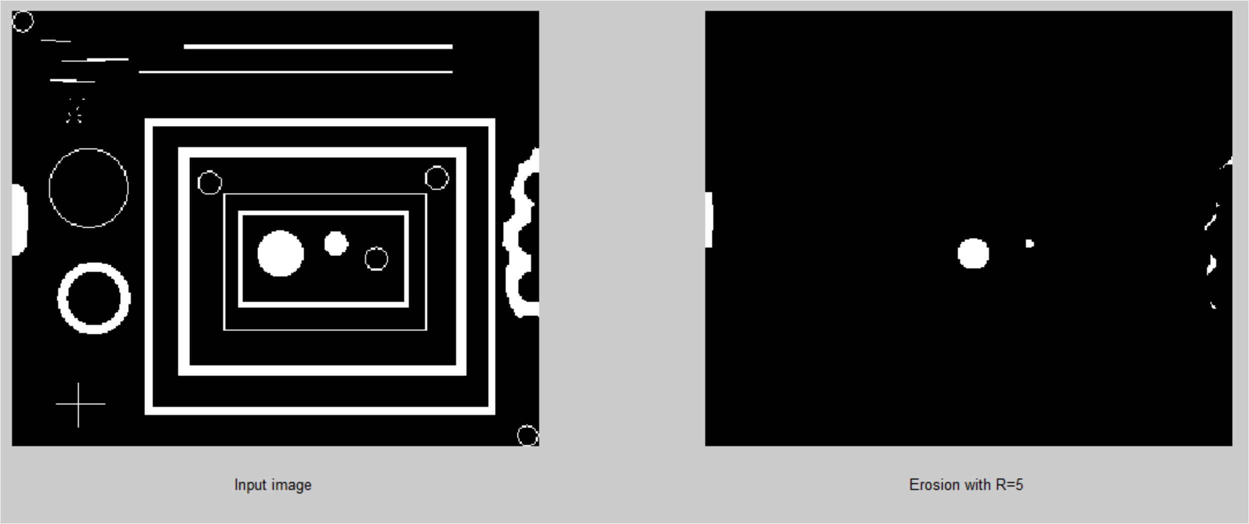


Q3 Read the image blobs.png and apply erosion using same structuring element as did in question 2. Now show both original and eroded image on one figure. What difference do you see? Now apply another SE on the same image but with a bigger size. Does increasing the size make a difference?

```
clc
clear
f=imread('blobs.png');
se=strel('diamond',1)
d=imerode(f,se);
subplot(1,2,1)
imshow(f);
xlabel('Input image')
subplot(1,2,2)
imshow(d)
xlabel('Erosion with R=1')
```

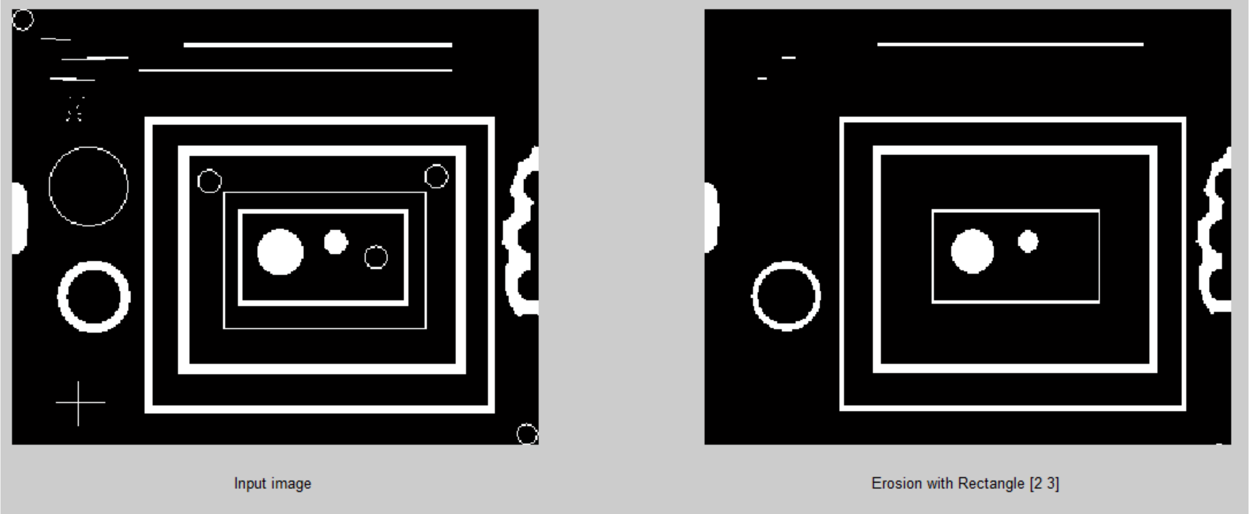






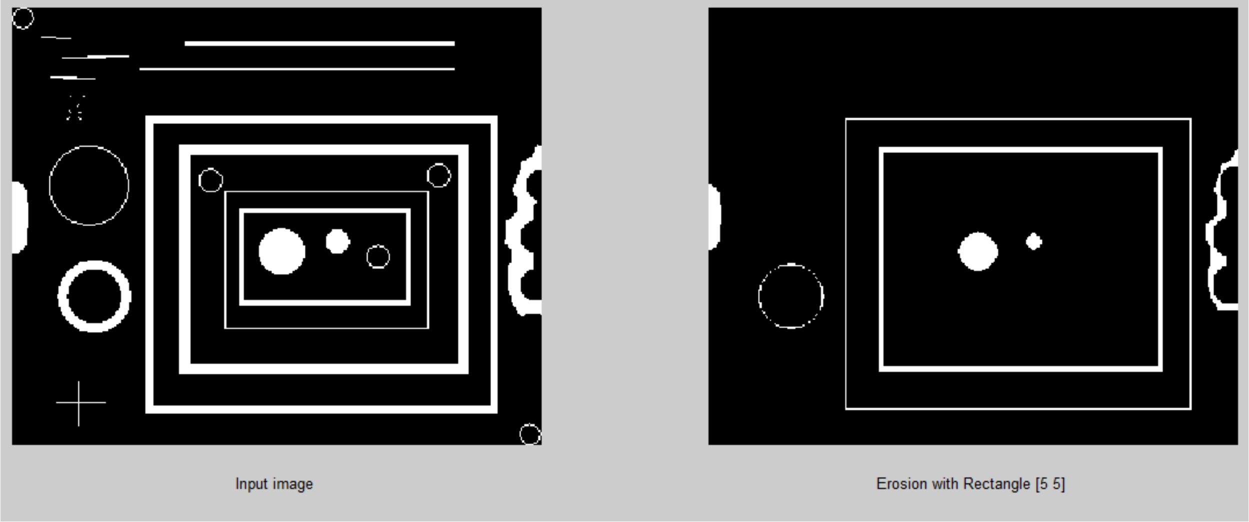
Changing SE

```
clc
clear
f=imread('blobs.png');
se=strel('rectangle',[2 3])
d=imerode(f,se);
subplot(1,2,1)
imshow(f);
xlabel('Input image')
subplot(1,2,2)
imshow(d)
xlabel('Erosion with Rectangle [2 3]')
```



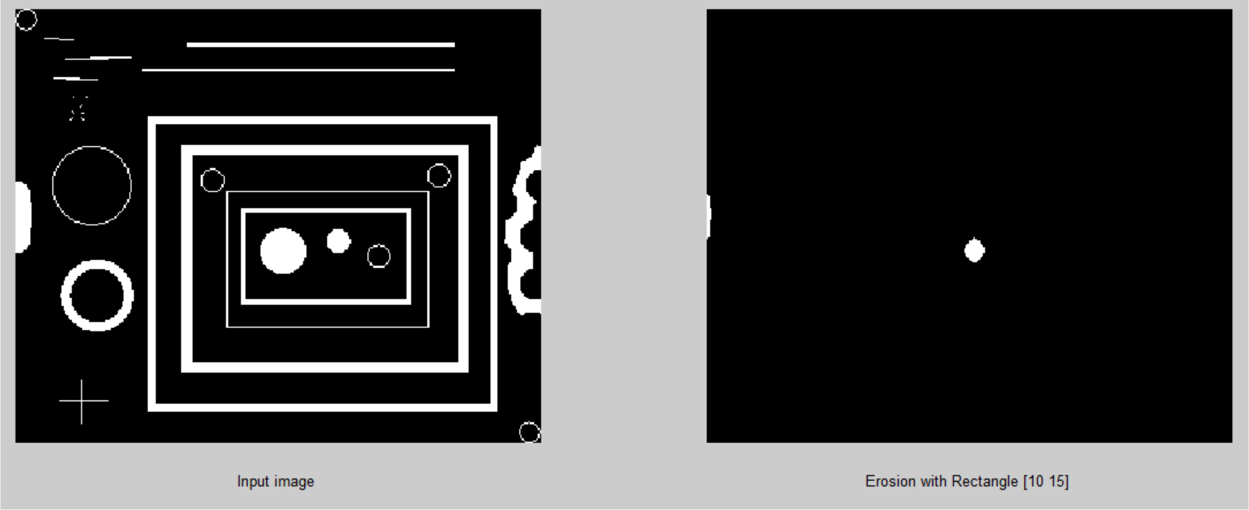
Input image

Erosion with Rectangle [2 3]



Input image

Erosion with Rectangle [5 5]

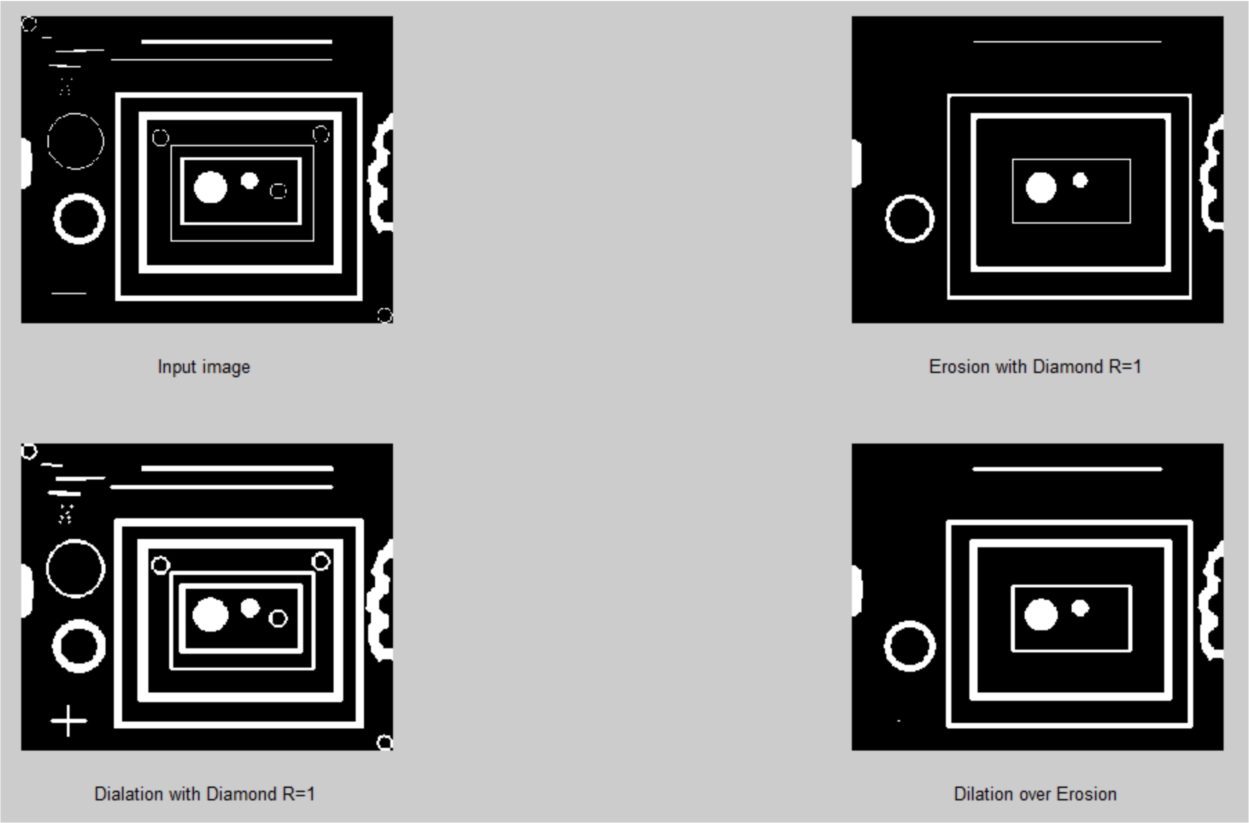


Input image

Erosion with Rectangle [10 15]

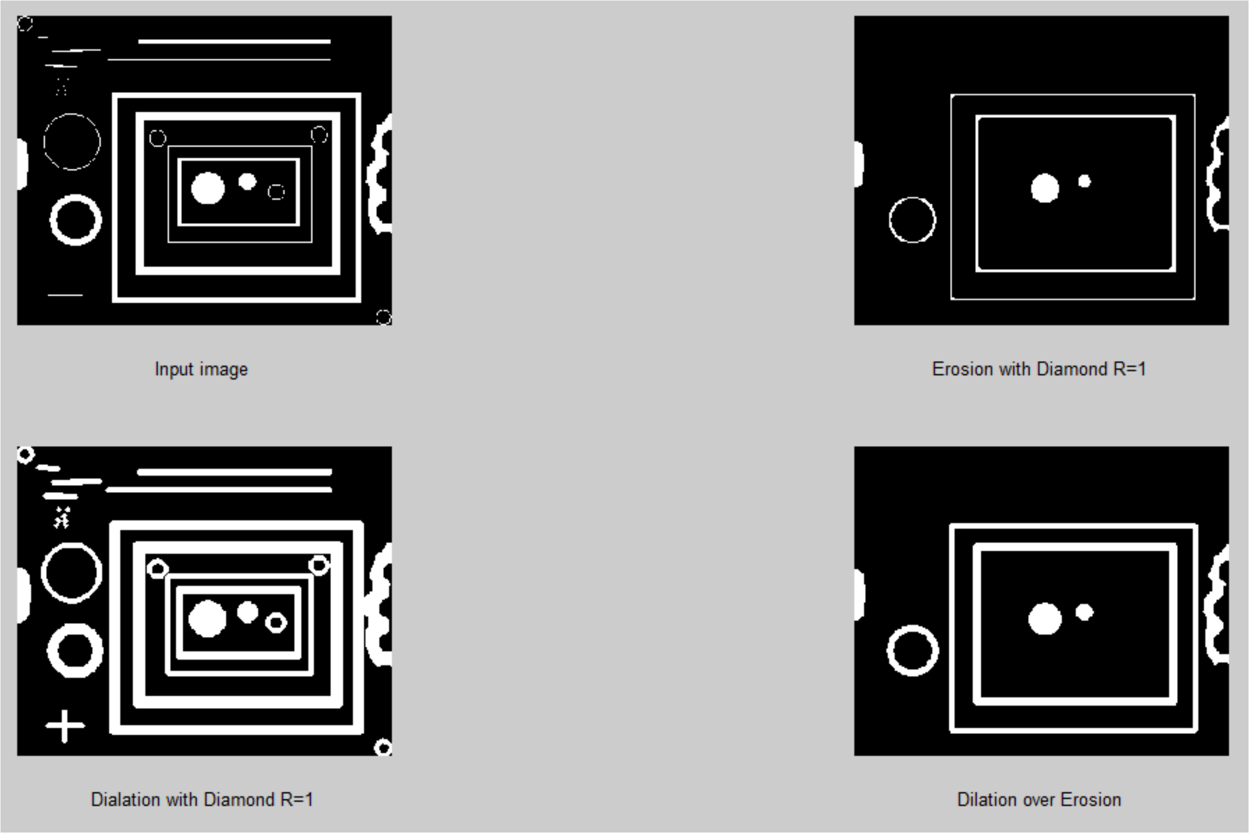
Q4 Read the image blobs.png and apply erosion and then dilation on the eroded image.Now show original, eroded and eroded+dilated image on one figure . What do you see?

```
clc
clear
f=imread('blobs.png');
se=strel('diamond',1)
d=imerode(f,se);
x=imdilate(f,se);
y=imdilate(d,se);
subplot(2,2,1)
imshow(f);
xlabel('Input image')
subplot(2,2,2)
imshow(d)
xlabel('Erosion with Diamond R=1')
subplot(2,2,3)
imshow(x)
xlabel('Dialation with Diamond R=1')
subplot(2,2,4)
imshow(y)
xlabel('Dilation over Erosion')
```



```
clc
clear
f=imread('blobs.png');
se=strel('diamond',2)
d=imerode(f,se);
x=imdilate(f,se);
y=imdilate(d,se);
subplot(2,2,1)
imshow(f);
xlabel('Input image')
subplot(2,2,2)
imshow(d)
xlabel('Erosion with Diamond R=1')
subplot(2,2,3)
imshow(x)
xlabel('Dialation with Diamond R=1')
subplot(2,2,4)
```

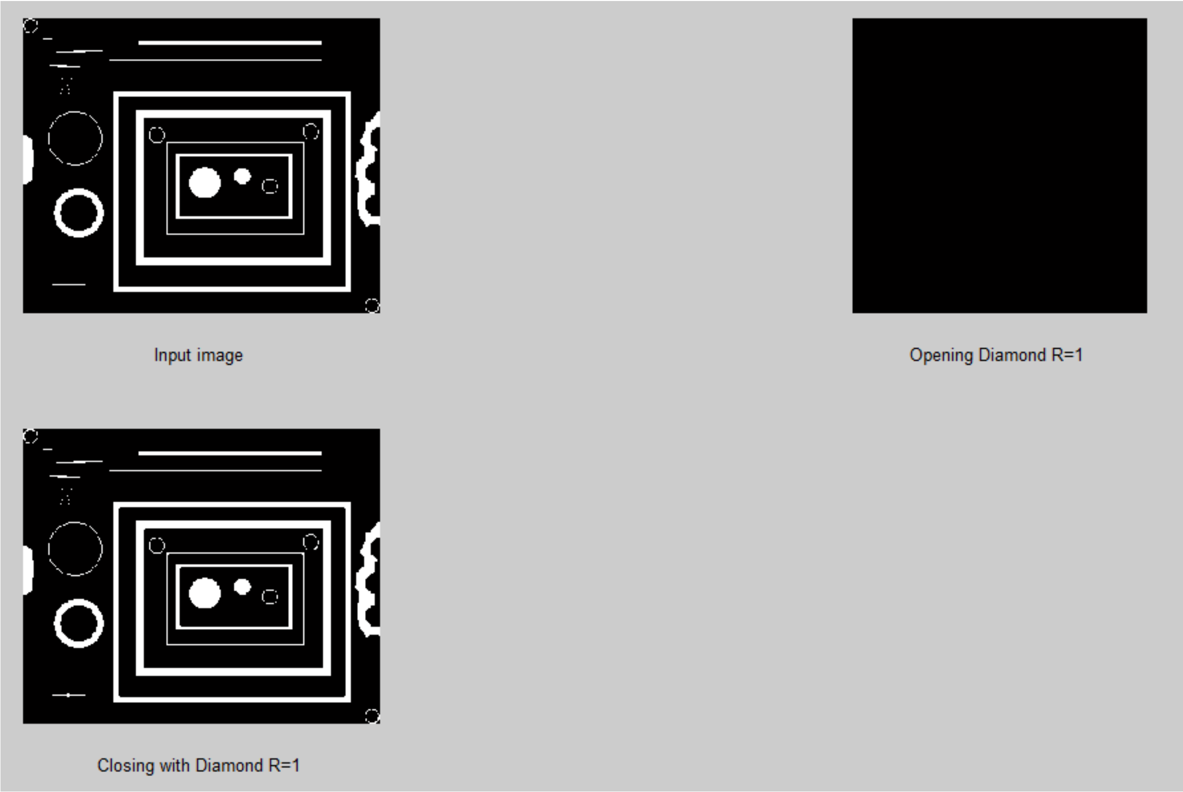
```
imshow(y)
xlabel('Dilation over Erosion')
```

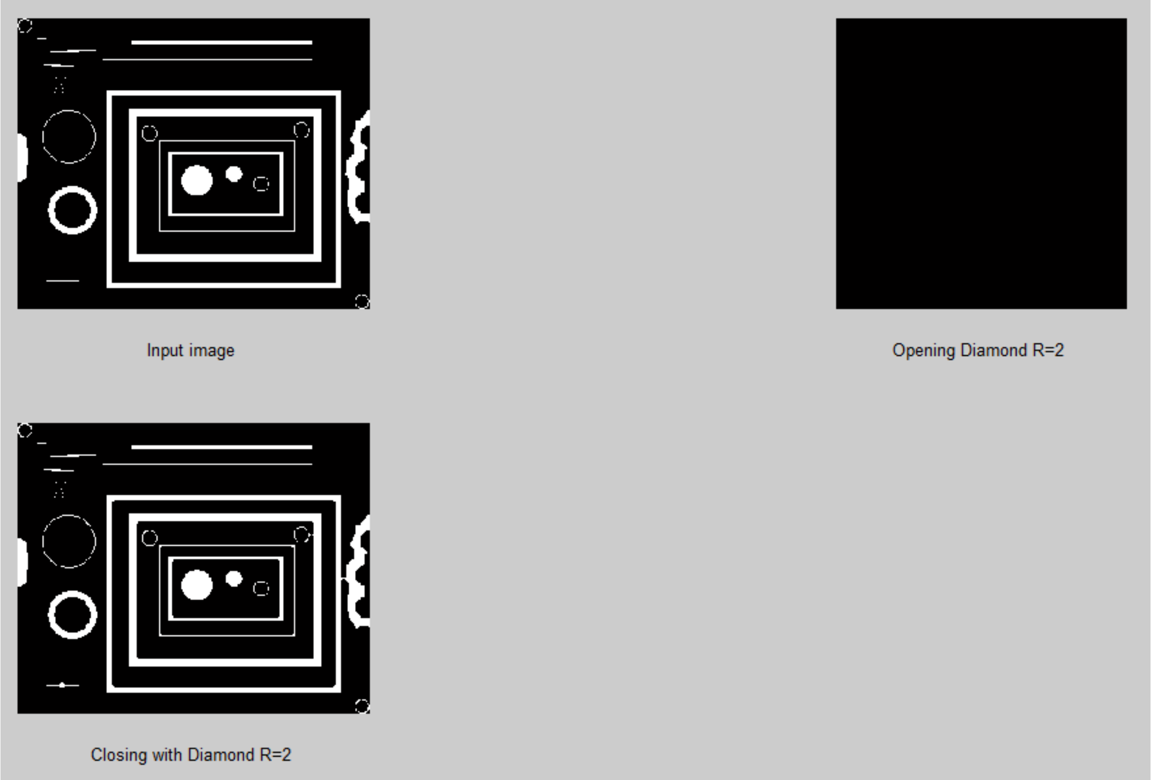


MATLAB has a built in function for erosion followed by dilation (opening) and dilation followed by erosion (closing) by same SE. The built in function are imopen(f,B) and imclose(f,B)

```
clc
clear
f=imread('blobs.png');
se=strel('diamond',1)
subplot(2,2,1)
imshow(f);
xlabel('Input image')
o=imopen(f,se)
subplot(2,2,2)
imshow(0)
xlabel('Opening Diamond R=1')
```

```
c=imclose(f,se)
subplot(2,2,3)
imshow(c)
xlabel('Closing with Diamond R=1')
```

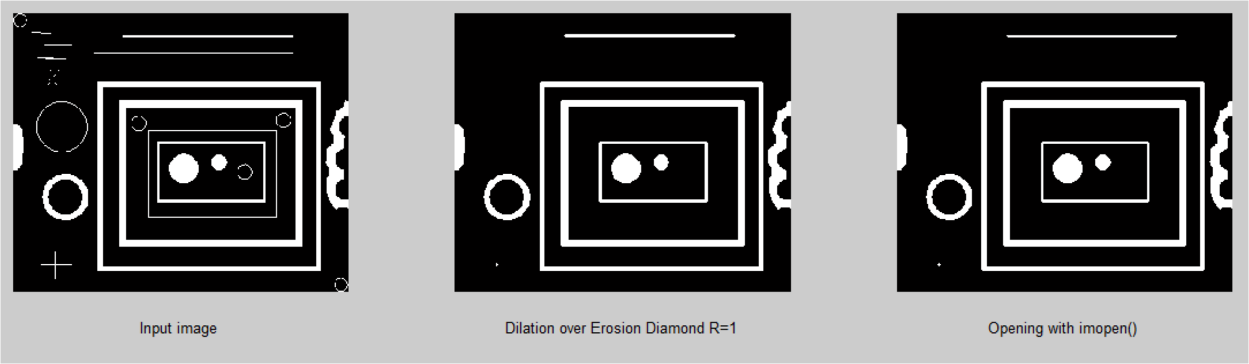


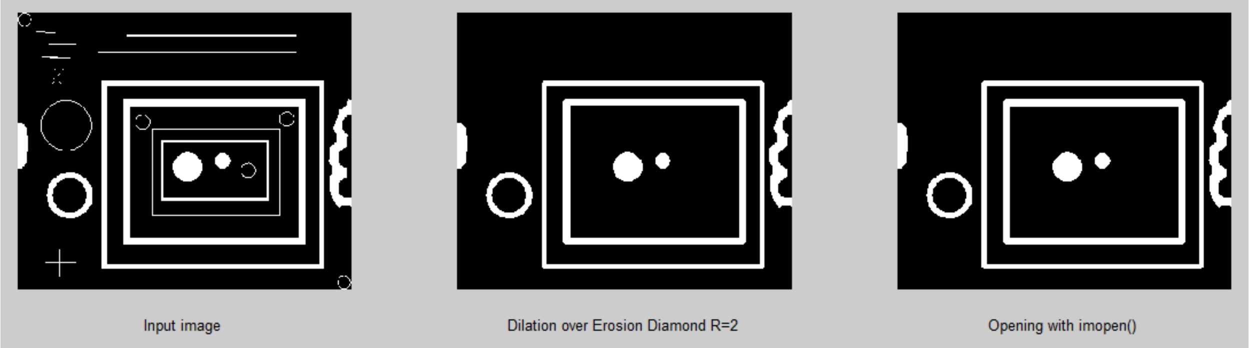


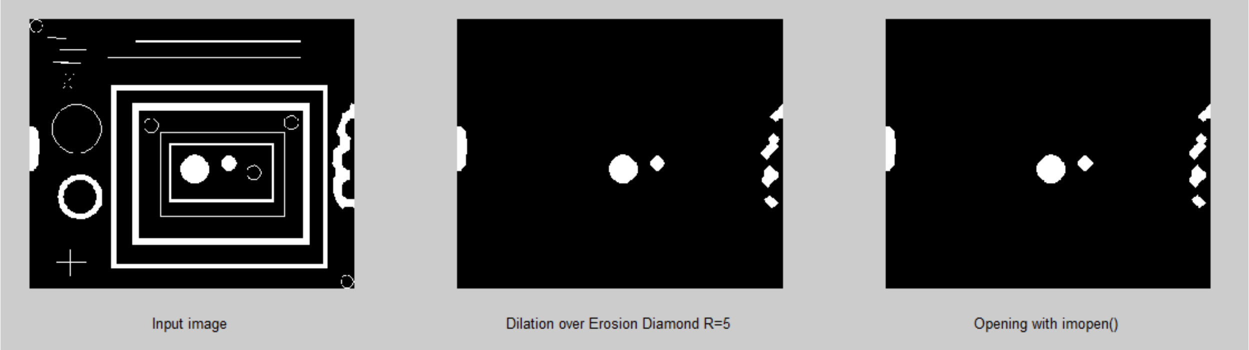
Q5 Read the image blobs.png and apply erosion and then dilation on the eroded image. Now apply opening on original image. Now show original, eroded+dilated image and opened image on one figure. Are the last two images same?

```
clc

clear

f=imread('blobs.png');

se=strel('diamond',1);

er=imerode(f,se);

di=imdilate(er,se);

op=imopen(f,se);

subplot(1,3,1)

imshow(f);

xlabel('Input image')

subplot(1,3,2)

imshow(di)

xlabel('Dilation over Erosion Diamond R=1')

subplot(1,3,3)
```

```
imshow(op)
xlabel('Opening with imopen()')
```



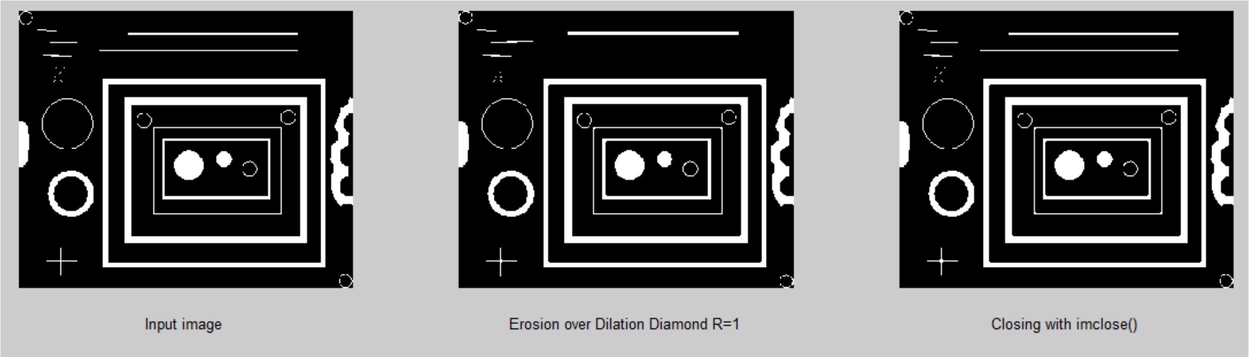


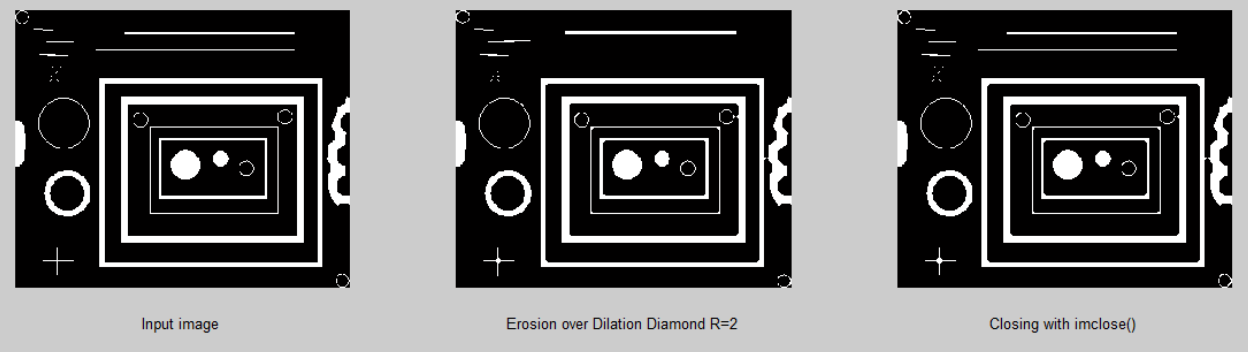


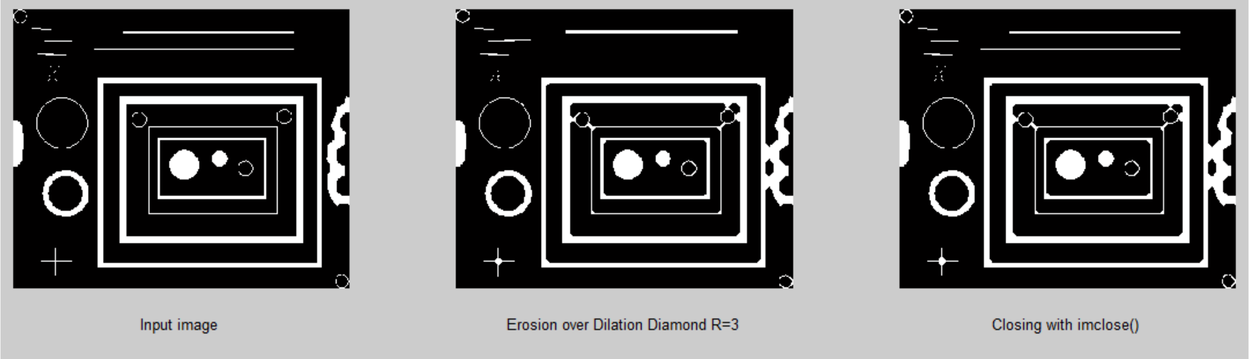
Now apply closing on original image and see what does it do. (Hint: Use a bigger SE for closing).

```
clc
clear
f=imread('blobs.png');
se=strel('diamond',1);
di=imdilate(f,se);
cl=imerode(di,se);
```

```
close=imclose(f,se);
subplot(1,3,1)
imshow(f);
xlabel('Input image')
subplot(1,3,2)
imshow(cl)
xlabel('Erosion over Dilation Diamond R=1')
subplot(1,3,3)
imshow(close)
xlabel('Closing with imclose()')
```



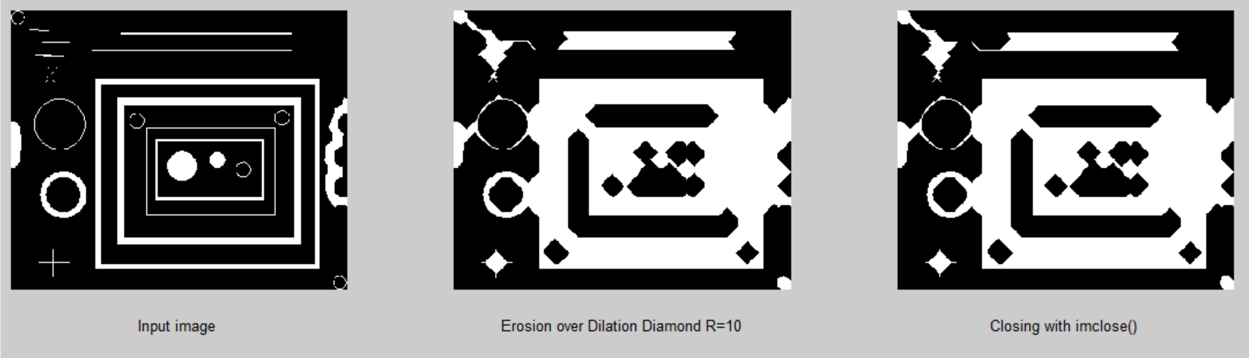




Input image

Erosion over Dilation Diamond R=3

Closing with imclose()



Input image

Erosion over Dilation Diamond R=10

Closing with imclose()