Touching new heights of excellence!

Department of Computer Science & Engineering

LAB MANUAL-DIP

Program: Computer Science & Engineering

Index

LAB 1: Write a program for image enhancement

Adjusting Intensity Values to a Specified Range

You can adjust the intensity values in an image using the imadjust function, where you specify the range of intensity values in the output image.

For example, this code increases the contrast in a low-contrast grayscale image by remapping the data values to fill the entire intensity range [0, 255].

```
I                    =
imread('pout.tif');
J   =   imadjust(I);
imshow(J)
figure, imhist(J,64)
```

This figure displays the adjusted image and its histogram. Notice the increased contrast in the image, and that the histogram now fills the entire range.

Specifying the Adjustment Limits

You can optionally specify the range of the input values and the output values using imadjust. You specify these ranges in two vectors that you pass to imadjust as arguments. The first vector specifies the low- and high-intensity values that you want to map. The second vector specifies the scale over which you want to map them.

For example, you can decrease the contrast of an image by narrowing the range of the data. In the example below, the man's coat is too dark to reveal any detail. imadjust maps the range [0,51] in the uint8 input image to [128,255] in the output image. This brightens the image considerably, and also widens the dynamic range of the dark portions of the original image, making it much easier to see the details in the coat. Note, however, that because all values above 51 in the original image are mapped to 255 (white) in the adjusted image, the adjusted image appears washed out.

```
I = imread('cameraman.tif');
J      =      imadjust(I,[0
0.2],[0.5               1]);
imshow(I)
figure, imshow(J)
```

Setting the Adjustment Limits Automatically

To use imadjust, you must typically perform two steps: View the histogram of the image to determine the intensity value limits. Specify these limits as a fraction between 0.0 and 1.0 so that you can pass them to imadjust in the [low_in high_in] vector.

For a more convenient way to specify these limits, use the stretchlim function. (The imadjust function uses stretchlim for its simplest syntax, imadjust(I).)

This function calculates the histogram of the image and determines the adjustment limits automatically. The stretchlim function returns these values as fractions in a vector that you can pass as the [low_in high_in] argument to imadjust; for example:

```
I = imread('rice.png');
J = imadjust(I,stretchlim(I),[0 1]);
```

By default, stretchlim uses the intensity values that represent the bottom 1% (0.01) and the top 1% (0.99) of the range as the adjustment limits. By trimming the extremes at both ends of the intensity range, stretchlim makes more room in the adjusted dynamic range for the remaining intensities. But you can specify other range limits as an argument to stretchlim. See the stretchlim reference page for more information.

Gamma Correction

imadjust maps low to bottom, and high to top. By default, the values between low and high are mapped linearly to values between bottom and top. For example, the value halfway between low and high corresponds to the value halfway between bottom and top.

imadjust can accept an additional argument that specifies the gamma correction factor. Depending on the value of gamma, the mapping between values in the input and output images might be nonlinear. For example, the value halfway between low and high might map to a value either greater than or less than the value halfway between bottom and top.

Gamma can be any value between 0 and infinity. If gamma is 1 (the default), the mapping is linear. If gamma is less than 1, the mapping is weighted toward higher (brighter) output values. If gamma is greater than 1, the mapping is weighted toward lower (darker) output values.

The figure below illustrates this relationship. The three transformation curves show how values are mapped when gamma is less than, equal to, and greater than 1. (In each graph, the x-axis represents the intensity values in the input image, and the y-axis represents the intensity values in the output image.)

The example below illustrates gamma correction. Notice that in the call to imadjust, the data ranges of the input and output images are specified as empty matrices. When you specify an empty matrix, imadjust uses the default range of [0,1]. In the example, both ranges are left empty; this means that gamma correction is applied without any other adjustment of the data.

```
[X,map]                  =
imread('forest.tif')   I   =
ind2gray(X,map);
J                        =
imadjust(I,[],[],0.5);
imshow(I)
figure, imshow(J)
```

Lab 2: Write a program for image compression

Aa standard matlab wavelet code package, WAVELAB 802, to perform the transforms. The wavelets we chose to use were the Deslauriers wavelets of polynomial size 3.

The compression scheme we used was to set a threshold value that was some fraction of the norm of the entire wavelet transform matrix. If the magnitude of a wavelet in the representation was not larger than this value, it was not included in the compression. We then rebuilt an image which (to some degree that depended on how many bases we included) resembled the original image by running the inverse transform.

ORIGINAL IMAGE

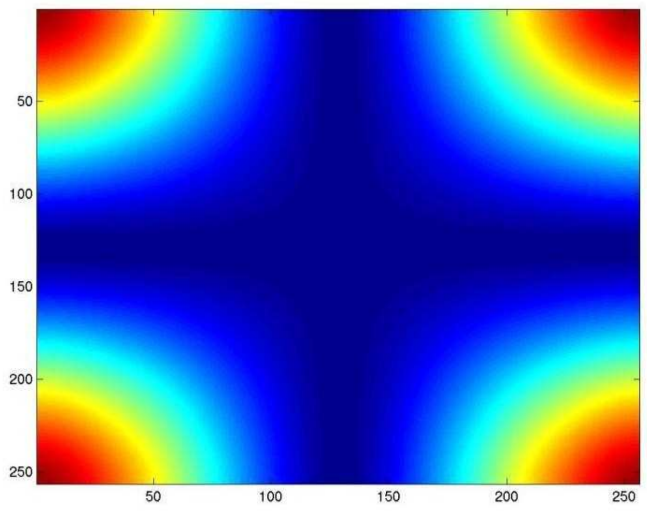
The fabulous Lena

50
100
150
200

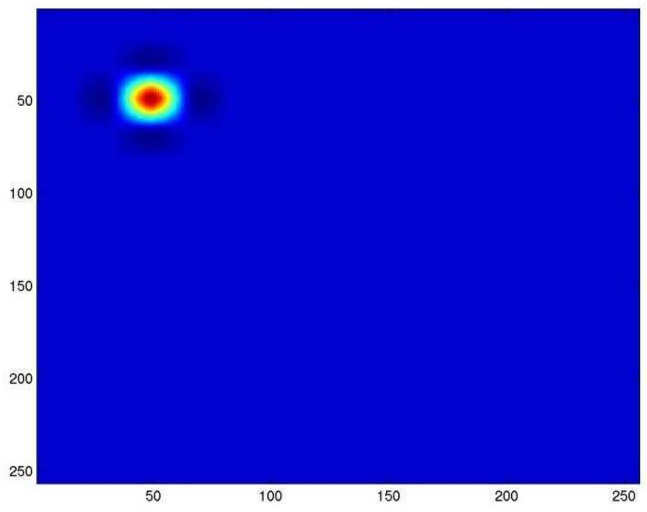
BASIS VECTORS

These are some basis vectors obtained by running the inverse wavelet transform on a matrix with a single nonzero value in it.

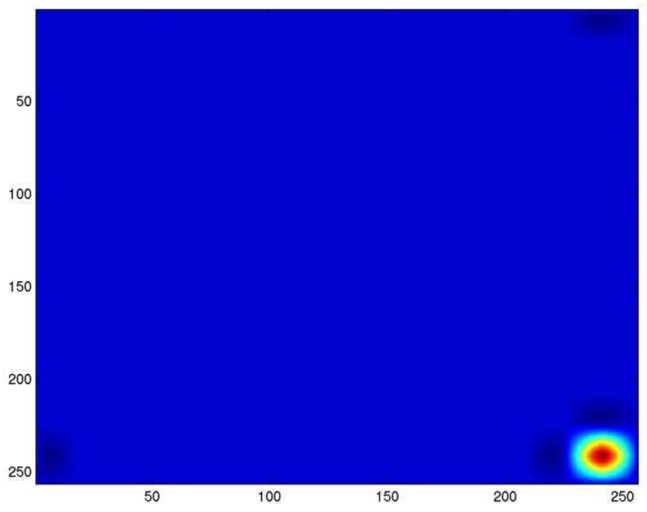
Deslaurier(1,1)



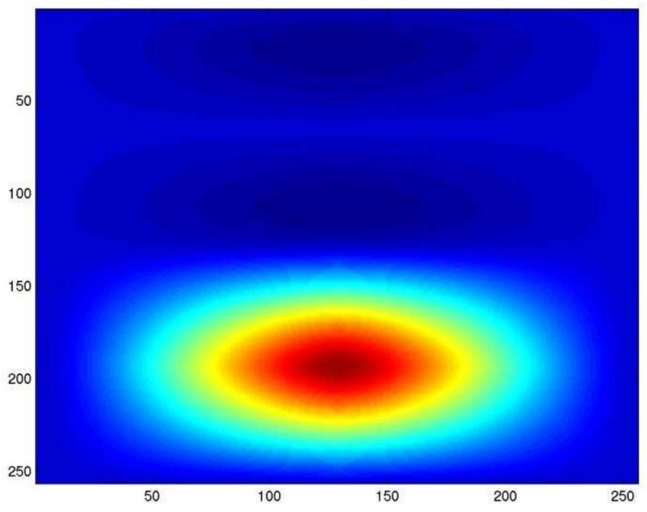
Deslaurier(10,10)



Deslaurier(16,16)

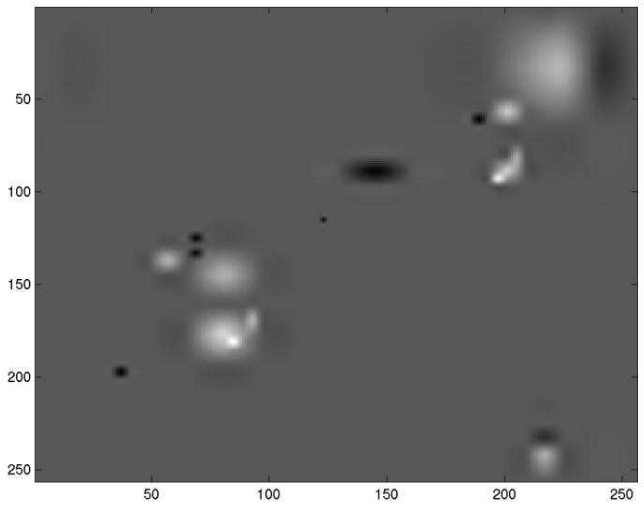


Deslaurier(4,2)

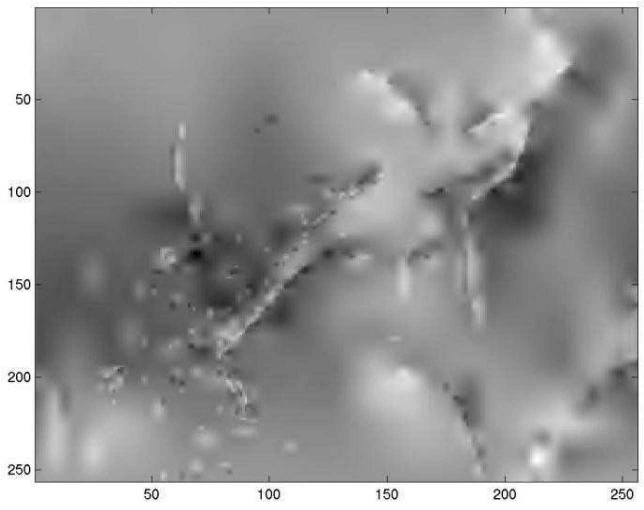


COMPRESSED IMAGES

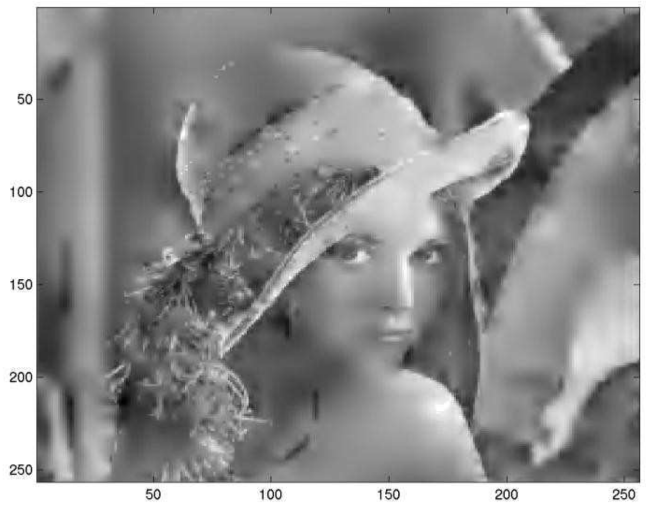
Threshold = .5, Bases included = 19, Compression ratio = 3400 : 1



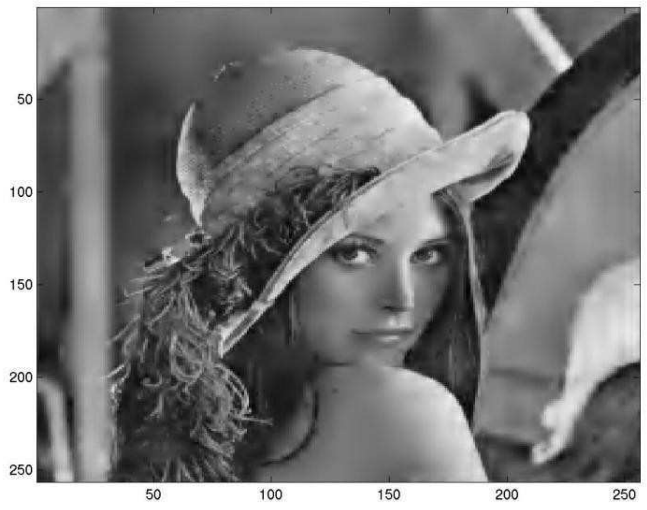
Threshold = 1, Bases included = 470, Compression ratio = 140 : 1



Threshold = 2, Bases included = 2383, Compression ratio = 27 : 1



Threshold = 4, Bases included = 6160, Compression ratio = 10 : 1



Threshold = 8, Bases included = 12378, Compression ratio = 5 : 1

MATLAB CODE

```
load
/home/nirav/elec301/lena256.mat;
imagesc(lena256);
colormap(gray(256));


[qmf, dqmf] = MakeBSFilter('Deslauriers', 3);

% The MakeBSFilter function creates biorthonormal filter pairs. The filter
% pairs that we're making is an Interpolating (Deslauriers-Dubuc) filter
% of polynomial degree 3

wc = FWT2_PB(lena256, 1, qmf, dqmf);

% wc correspond to the wavelet coefficients of the sample image
% FWT2_PB is a function that takes a 2 dimensional wavelet transform
% We specify the image matrix, the level of coarseness (1), the quadrature
% mirror filter (qmf), and the dual quadrature mirror filter (dqmf)

% we take a tolerance which is some fraction
% of the norm of the sample image

nl = norm(lena256) / (4 * norm(size(lena256)));

% if the value of the wavelet coefficient matrix at a particular
% row and column is less than the tolerance, we 'throw' it out
% and increment the zero count.

zerocount   =
 0;   for   i   =
 1:256 for j =
 1:256
   if ( abs(wc(i,j)) < nl)
```

```
        wc(i,j) = 0;
        zerocount = zerocount
    + 1; end
  en
d
end

x   =   IWT2_PB(wc,   1,   qmf,
dqmf); imagesc(x);

%  here  is  some  sample  code  to  view  how  these  deslauriers

wavelets look [qmf, dqmf] = MakeBSFilter('Deslauriers',
3); for i = 1:256
  for     j     =
    1:256
    wc(i,j)    =
    0;
  en
d
end

%   this   is   the   Deslauriers(4,2)

matrix wc(4, 2) = 1000;
x   =   IWT2_PB(wc,   1,   qmf,
dqmf); imagesc(x);
```

LAB 3: Write a program for color image processing

Color Approximation

To reduce the number of colors in an image, use the rgb2ind function. This function converts a truecolor image to an indexed image, reducing the number of colors in the process. rgb2ind provides the following methods for approximating the colors in the original image: Quantization Uniform quantization Minimum variance quantization Colormap mapping

The quality of the resulting image depends on the approximation method you use, the range of colors in the input image, and whether or not you use dithering. Note that different methods work better for different images. See Dithering for a description of dithering and how to enable or disable it.

Quantization

Reducing the number of colors in an image involves quantization. The function rgb2ind uses quantization as part of its color reduction algorithm. rgb2ind supports two quantization methods: uniform quantization and minimum variance quantization.

An important term in discussions of image quantization is RGB color cube, which is used frequently throughout this section. The RGB color cube is a three-dimensional array of all of the colors that are defined for a particular data type. Since RGB images in MATLAB can be of type uint8, uint16, or double, three possible color cube definitions exist. For example, if an RGB image is of class uint8, 256 values are defined for each color plane (red, blue, and green), and, in total, there will be 224 (or 16,777,216) colors defined by the color cube. This color cube is the same for all uint8 RGB images, regardless of which colors they actually use.

The uint8, uint16, and double color cubes all have the same range of colors. In other words, the brightest red in a uint8 RGB image appears the same as the brightest red in a double RGB image. The difference is that the double RGB color cube has many more shades of red (and many more shades of all colors). The following figure shows an RGB color cube for a uint8 image.

Quantization involves dividing the RGB color cube into a number of smaller boxes, and then mapping all colors that fall within each box to the color value at the center of that box.

Uniform quantization and minimum variance quantization differ in the approach used to divide up the RGB color cube. With uniform quantization, the color cube is cut up into equal-sized boxes (smaller cubes). With minimum variance quantization, the color cube is cut up into boxes (not necessarily cubes) of different sizes; the sizes of the boxes depend on how the colors are distributed in the image.

Uniform Quantization. To perform uniform quantization, call rgb2ind and specify a tolerance. The tolerance determines the size of the cube-shaped boxes into which the RGB color cube is divided. The allowable range for a tolerance setting is [0,1]. For

example, if you specify a tolerance of 0.1, the edges of the boxes are one-tenth the length of the RGB color cube and the maximum total number of boxes is
n = (floor(1/tol)+1)^3

The commands below perform uniform quantization with a tolerance of 0.1. RGB = imread('peppers.png');
[x,map] = rgb2ind(RGB, 0.1);

The following figure illustrates uniform quantization of a uint8 image. For clarity, the figure shows a two-dimensional slice (or color plane) from the color cube where red=0 and green and blue range from 0 to 255. The actual pixel values are denoted by the centers of the x's.

After the color cube has been divided, all empty boxes are thrown out. Therefore, only one of the boxes is used to produce a color for the colormap. As shown earlier, the maximum length of a colormap created by uniform quantization can be predicted, but the colormap can be smaller than the prediction because rgb2ind removes any colors that do not appear in the input image.

Minimum Variance Quantization. To perform minimum variance quantization, call rgb2ind and specify the maximum number of colors in the output image's colormap. The number you specify determines the number of boxes into which the RGB color cube is divided. These commands use minimum variance quantization to create an indexed image with 185 colors.

```
RGB                    =
imread('peppers.png');
[X,map]                =
rgb2ind(RGB,185);
```

Minimum variance quantization works by associating pixels into groups based on the variance between their pixel values. For example, a set of blue pixels might be grouped together because they have a small variance from the center pixel of the group.

In minimum variance quantization, the boxes that divide the color cube vary in size, and do not necessarily fill the color cube. If some areas of the color cube do not have pixels, there are no boxes in these areas.

While you set the number of boxes, n, to be used by rgb2ind, the placement is determined by the algorithm as it analyzes the color data in your image. Once the image is divided into n optimally located boxes, the pixels within each box are mapped to the pixel value at the center of the box, as in uniform quantization.

The resulting colormap usually has the number of entries you specify. This is because the color cube is divided so that each region contains at least one color that appears in the input image. If the input image uses fewer colors than the number you specify, the output colormap will have fewer than n colors, and the output image will contain all of the colors of the input image.

The following figure shows the same two-dimensional slice of the color cube as shown in the preceding figure (demonstrating uniform quantization). Eleven boxes have been created using minimum variance quantization.

For a given number of colors, minimum variance quantization produces better results than uniform quantization, because it takes into account the actual data. Minimum variance quantization allocates more of the colormap entries to colors that appear frequently in the input image. It allocates fewer entries to colors that appear infrequently. As a result, the accuracy of the colors is higher than with uniform quantization. For example, if the input image has many shades of green and few shades of red, there will be more greens than reds in the output colormap. Note that the computation for minimum variance quantization takes longer than that for uniform quantization.

Colormap Mapping

If you specify an actual colormap to use, rgb2ind uses colormap mapping (instead of quantization) to find the colors in the specified colormap that best match the colors in the RGB image. This method is useful if you need to create images that use a fixed colormap. For example, if you want to display multiple indexed images on an 8-bit display, you can avoid color problems by mapping them all to the same colormap. Colormap mapping produces a good approximation if the specified colormap has similar colors to those in the RGB image. If the colormap does not have similar colors to those in the RGB image, this method produces poor results.

This example illustrates mapping two images to the same colormap. The colormap used for the two images is created on the fly using the MATLAB function colorcube, which creates an RGB colormap containing the number of colors that you specify. (colorcube always creates the same colormap for a given number of colors.) Because the colormap includes colors all throughout the RGB color cube, the output images can reasonably approximate the input images.

```
RGB1 =
imread('autumn.tif'); RGB2
= imread('peppers.png');
X1 =
rgb2ind(RGB1,colorcube(128));
X2 =
rgb2ind(RGB2,colorcube(128));
```

Lab 4: Write a program for image segmentation

demonstration of global and local thresholding for segmentation

```
% threshdemo.m
% Demonstration of global and local threshold operations of an image


clear                    all
[tmp,idx]=imread('lena.bmp');
a=ind2gray(tmp,idx); % gray scale image of lena, value between 0
and 1 clear tmp idx
figure(1),clf,colormap('gray'),imshow(a),title('original
Lena image') [m,n]=size(a); % size of image a
b=reshape(a,m*n,1);  %  into  a  column
vector figure(2),hist(b,50),title('histogram
of image')
% first do global thresholding
mu=rand(2,1); % value betwen 0 and 1, two clusters
only  [W,iter,Sw,Sb,Cova]=kmeansf(b,mu);%  W  is
the mean,
% Cova is the covariance matrices
%  member:  membership  of  each  X:  K  by  1  vector  of  elements  1  to  c
[d,member]=kmeantest(b,sort(W));
c=reshape(member-
1,m,n);  clear  d  member
b
figure(3),clf,colormap('gray'),imsh
ow(c) title('global threshold')

% next do local threshold, partition the image into 64 x 64 blocks
%  and  do  threshold  within  each
block c=zeros(512,512); trials=0;
for i=1:8,
  for       j=1:8,
    trials=trials+
    1;
    disp([int2str(trials)  '  of  64  iterations
    ...']);     tmp=a(64*(i-1)+1:64*i,64*(j-
    1)+1:64*j);
    tmpi=reshape(tmp,64*64,1);
    mu=sort(rand(2,1));  %  value  betwen  0  and  1,  two
    clusters                                          only
    [W,iter,Sw,Sb,Cova]=kmeansf(tmpi,mu);%  W  is  the
    mean,
    % Cova is the covariance matrices
    % member: membership of each X: K by 1 vector of elements
    1    to    c    [d,member]=kmeantest(tmpi,sort(W));    c(64*(i-
    1)+1:64*i,64*(j- 1)+1:64*j)=reshape(member,64,64);
  en
```

```
d
end
figure(4),clf,colormap('gray'),imsho
w(c- 1), title('local threshold, 64x64
block');
```

Lab 5: Write a program for image morphology

demonstrate boundary extraction, interior filling

```
% demonstrate morphological boundary extraction
%
clear all, close all

A0=imread('myshap4.bmp');
imshow(A0);  %  a  heart  shape  hand
drawing title('original image');
pause
% A0 contains mostly 1s and the drawing contains 0s,
uint8 A1=1-double(A0); % invert black and white

B=ones(3);
A2=imopen(imclose(A1,B),B); % fill 1 pixel hole and remove

sticks A3=imfill(A2,[100 100]); % fill the interior

A=double(A2) + double(A3);

imshow(A),title('after   interior   filling
using imfill'); pause
Ab=A-double(erode(A,B));

imshow(Ab),              title('extracted

boundary');    clear    A1    A2    A3;

Ac=Ab; vidx=[[1:20:280] 280];

Ac(vidx,:)=1;

Ac(:,vidx)=1;

imshow(Ac)
```

Lab 6: Program for Image Restoration

```
% load image
X                         =
double(imread('midterm.bmp
')); X = X-mean(X(:));
[m,n] = size(X);


%  show  image
and  DFT  fX  =
fft2(X);
figure(1)
imshow(real(X),[])
;
title('origina
l      image')
figure(2)
imshow(fftshift(log(1+abs(fX))
),[])     title('log(1+|DFT|))
original image');

%             model
             blurrin
g filter s = 24; t=
0;
u = 1; v=0;
g = [ones(s,1);zeros(m-s-t,1); ones(t,1)];
%g    =    [ones(s,1);0.99;  zeros(m-s-t-
2,1);0.99; ones(t,1)]; g = g/sum(abs(g));
h    =    [ones(u,1);   zeros(n-u-v,1);
ones(v,1)];      h      =
h/sum(abs(h)); f =g*h';
ff         =
fft2(f);
figure(3)
imshow(fftshift(log(1+abs(ff)
)),[])     title('amplitude:
log(1+|OTF|)');    figure(4)
imshow(fftshift(angle(ff)),[
]) title('phase of OTF');

% get pseudo inverse filter
ff(find(abs(ff)==0))=
NaN; aff = abs(ff);
pff = ff./aff;
apiff        =
1./aff;   ppiff
= conj(pff);
ppiff(find(isnan(ppiff
))) = 0; cap = 11;
apiff(find(apiff  >  cap))
=                    cap;
apiff(find(isnan(apiff)))
= 0; piff = apiff.*ppiff;



%  deblur  and
show   frX   =
piff.*fX;
rX
=real(ifft2(frX)
); figure(5)
imshow(fftshift(log(1+abs(frX))),[])
title('log(1+|DFT|))
restored         image')
figure(6)
imshow(rX(:,5:n),[
]);
title('restored
image')
```

Lab 8: Blurring 8 bit color versus monochrome

```
% smoothing in eight bit color and monochrome
% parameter definition


% get   image   from
       MATLAB      library
load('clown');

%   construct   convolution
functions [m,n] = size(X);
gs = [0.5 0.5]; ge = [];
hs = [0.5 0.5]; he = [];
g = [gs,zeros(1,m-length(gs)-length(ge)),ge];
h = [hs,zeros(1,n-length(hs)-length(he)),he];

% construct convolution matrices and  blur
           sparse matrices Y = spcnvmat(g);
Z           =
spcnvmat(h);
W = Y*X*Z';

%  show  original  and  blurred
images figure(1);
imshow(X,[]);
figure(2);
imshow(W,[]);
figure(3);
```

```
imshow(X,[]);
colormap(map)
figure(4);
imshow(W,[]);
colormap(map)
```