



COMPUTER VISION

LABORATORY PRACTICES

LAB 2

Basic Image Processing Techniques

Contents:

<i>INTRODUCTION.....</i>	<i>3</i>
<i>GOALS.....</i>	<i>3</i>
<i>THEORETICAL CONCEPTS.....</i>	<i>3</i>
<i>PART 1: GEOMETRIC IMAGE TRANSFORMATIONS.....</i>	<i>4</i>
<i>INTRODUCTION</i>	<i>4</i>
<i>SOME GEOMETRIC IMAGE TRANSFORMATIONS</i>	<i>5</i>
<i>PART 2: FILTERING AND IMAGE ENHANCEMENT.....</i>	<i>7</i>
<i>INTRODUCTION</i>	<i>7</i>
<i>SPATIAL IMAGE FILTERING.....</i>	<i>13</i>
<i>FILTERING IN THE FREQUENCY DOMAIN.....</i>	<i>22</i>
<i>PART 3: OTHER HELPFUL OPERATIONS WITH THE IMAGE TOOLBOX.....</i>	<i>33</i>
<i>PART 4: NON-CONTACT ACTIVITY (HOMEWORK)</i>	<i>36</i>

INTRODUCTION

This practice consists of four parts: the first three parts are proposed as basic practices and should be performed in the laboratory contact hours and the last part should be done as student homework. The first part explains how to use the image processing toolbox for some geometric operations in this toolbox such as rotations, cropping and zooming as well as more specialized operations. The second part works with examples of image enhancement, histogram equalization, etc. It is followed by an introduction to the design and implementation of filters and filtering images in the spatial domain by convolution masks.

Different supported filter types are described, and how MATLAB represents and applies them to image data. It also explains how to obtain the discrete cosine transform (DCT) and the discrete Fourier transform of an image and how to apply the latter to filters in the frequency domain. The latter part of this lab describes some non-contact activities that the student must complete. A complete report of this lab must be made and sent to the professor, containing the results of all parts of this practice (contact and non-contact).

GOALS

The main goal of this lab is that the students become familiar with the basic techniques of image processing in a guided way, using many examples with the Matlab image toolbox. In addition, it is proposed to solve several simple exercises to consolidate all the knowledge acquired in the guided part.

THEORETICAL CONCEPTS

The theoretical concepts that the student must have assimilated before this practice are included in topic 2. Some of them are the following:

- Image Processing (spatial and frequency transforms).
- Geometric transformations on images.
- Histogram.
- Image enhancement (sharpening and smoothing).

PART 1: GEOMETRIC IMAGE TRANSFORMATIONS

INTRODUCTION

Geometric functions included in this toolbox perform rotations, image cropping, zoom as well as more specialized operations. These functions support any type of image. In an RGB image each component is processed separately.

We will begin studying the concept of interpolation, a common operation to most geometric functions. Then we will discuss some geometric functions separately and show how to apply them to images.

Interpolation

Most geometric functions in this toolbox use two-dimensional interpolation as part of the operation they perform. Interpolation is the process by which the program determines values between certain pixels. For example, if an image is scaled so that it contains twice as many pixels it originally had, the program obtains the values for the new pixels through interpolation.

The Image Processing Toolbox uses three interpolation methods:

- nearest neighbor
- bilinear interpolation
- bicubic interpolation

The nearest neighbor interpolation adapts a constant surface through the intensity values. The value of an interpolated pixel is the value of its nearest pixel.

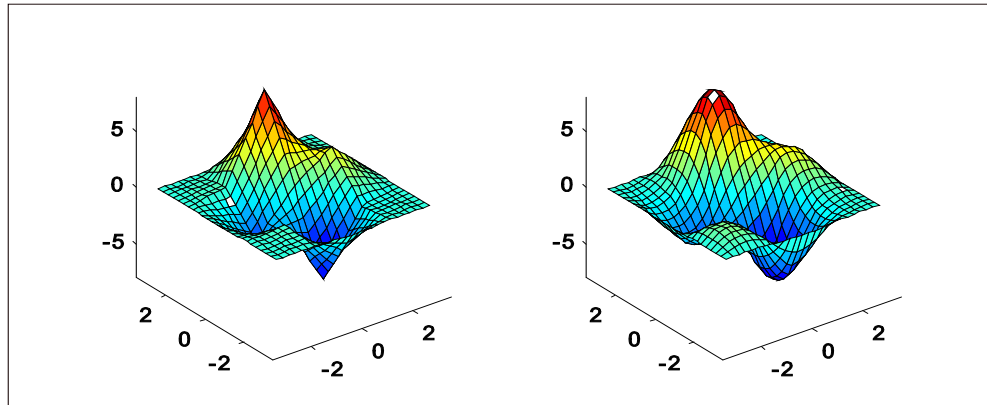
Bilinear interpolation adjusts a linear surface through the existing intensity values. The value of the interpolated pixel is a combination of the values of its 4 closest pixels. Bilinear interpolation is linear, faster and uses less memory than bicubic interpolation.

Bicubic interpolation fits a cubic surface through the existing intensity values. The value of the interpolated pixel is a combination of the values of its 16 closest pixels. This method produces a much smoother surface than bilinear interpolation. To use bicubic interpolation, the first and second derivatives of the surface must be continuous.

The commands below create a surface with two peaks and illustrate these differences.

```
[x,y,z]=peaks(5)           %Initial figure with values x,y,z
[xi,yi]=meshgrid(-3:.25:3); % Values to be interpolated
zlin=interp2(x,y,z,xi,yi,'linear'); %Linear interpolation
zcub=interp2(x,y,z,xi,yi,'cubic'); %Cubic interpolation
figure,colormap(jet);
surf(x,y,z);title('Initial figure: with few pixels');
figure,
subplot(1,2,1),           surf(xi,yi,zlin)
```

```
axis([-3.5 3.5 -3.5 3.5 -8 8]);title('Figure after linear
interpolation');
subplot(1,2,2), surf(xi,yi,zcub)
axis([-3.5 3.5 -3.5 3.5 -8 8]);title('Figure after cubic interpolation');
```



Some geometric functions allow you to specify the interpolation method as the last input argument. For these cases, it is important to consider the type of image. Duplication of pixels is usually the best method for indexed images while bilinear interpolation and bicubic are both appropriate grayscale and RGB images. Geometric functions accept the following strings to specify the type of interpolation:

```
-nearest-----> nearest neighbor interpolation.
-bilinear-----> bilinear interpolation
-bicubic-----> bicubic interpolation
```

Functions can recognize abbreviated versions of these chains with at least the first three characters.

Note: You can perform data interpolation in two dimensions, apart from the previous geometric functions, using function *interp2*. For non-uniformly spaced data *griddata* is used.

SOME GEOMETRIC IMAGE TRANSFORMATIONS

Image rotation

The *imrotate* command rotates an image using a specified interpolation method and an angle of rotation. If an interpolation method is not specified, the function determines the type of image and automatically chooses the best method.

For example, to rotate image “trees” 35°:

```
load trees
Y=imrotate(X,35);
figure, imshow(Y,map)
```

As this example does not specify the method of interpolation for *imrotate*, the function chooses the best method for this image. “trees” is an indexed image, therefore *imrotate* uses the nearest neighbor interpolation.

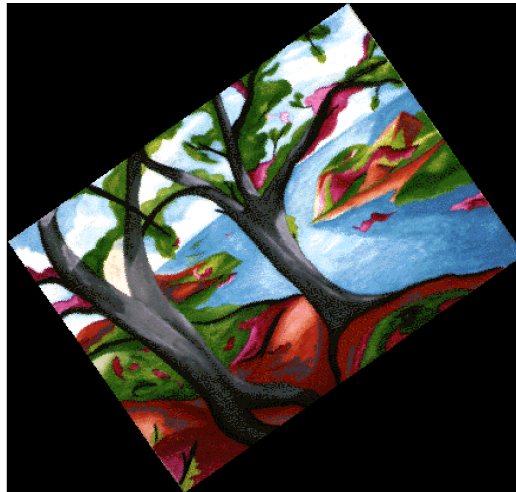


Image cropping

imcrop function extracts a rectangular portion of an image. *imcrop* defines the cutting rectangle interactively with the mouse or through a list of arguments. For example, to crop a rectangle of 92x95 pixels from image “trees” starting at the coordinate (71,107), we can use the following code:

```
load trees
figure, subplot(1,2,1), imshow(X,map)
subplot(1,2,2), imshow(imcrop(X,[71, 107, 92, 95]),map);
```



To define a crop rectangle interactively with the mouse, *imcrop* is used without input arguments. After calling *imcrop*, press the left mouse button while dragging on the displayed image. Release the mouse button when you have defined the desired area.

Resizing images

imresize function changes the size or the sampling of an image using a specified interpolation method. If an interpolation method is not specified, the function determines the type of image and automatically chooses the best method.

imresize can scale an image by a factor, or to a size specified by the number of rows and columns. The following example scales X to twice its current size:

```
Y=imresize(X,2);
```

The following example resizes image X to 100x150 pixels:

```
Y=imresize(X,[100 150]); % You can check the new size with: size(Y)
```

If the image size is reduced, *imresize* applies a low pass filter to the image before the interpolation. This reduces the Moiré effect, ripple resulting from aliasing during sampling.

Function *trueSize* can be applied to scale images. This function displays an image with a screen pixel for each pixel of the image.

Zoom

Once an image is displayed in a figure of Matlab you can see its details with *Figure* → *Tools* → *zoom in*.

Part 2: FILTERING AND IMAGE ENHANCEMENT

INTRODUCTION

This part explains how to use image processing toolbox to perform various image enhancement operations and to design and implement filters, both in space and frequency domain. Most examples shown here use grayscale images. To improve an RGB image, we work normally with arrays of red, green and blue components separately.

Different filter types are described, and how MATLAB represents and applies them to image data. Also, the discrete cosine transform (DCT) and the discrete Fourier transform of an image are explained and how to apply the latter transform to filters in the frequency domain.

Image intensity adjustment

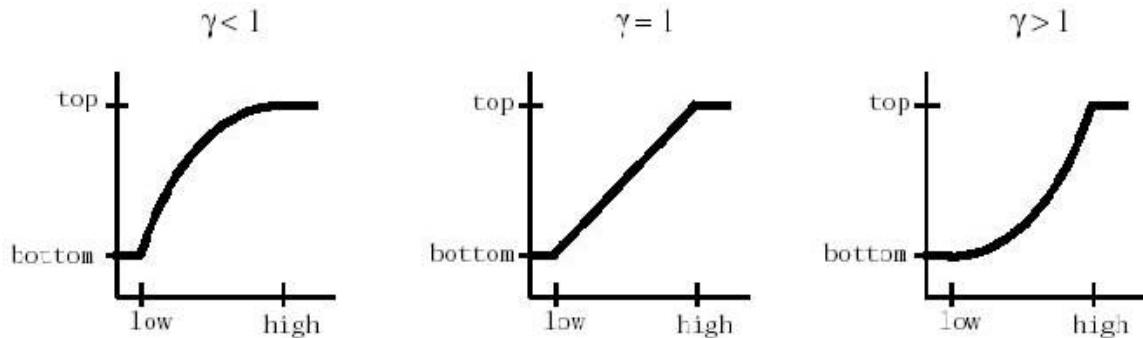
imadjust maps the intensity values of an image to a new range. There are three types of image adjustment:

- explicit ranges of intensities (input and output cropping)
- gamma correction
- image equalizing

The *imadjust* changes the intensity ranges that are passed as parameters; *imadjust* is not only used with grayscale images but also with color images operating on the red, green and blue components separately.

imadjust function can increase, decrease and change the intensity range of the input image to new ranges in the output image. In the following example *low* and *high* are the lowest and highest input intensities, and *bottom* and *top* the lowest and highest output intensities.

```
J=imadjust(I, [low high], [bot top], gamma); % with low, high, bot and top
between 0 y 1
```



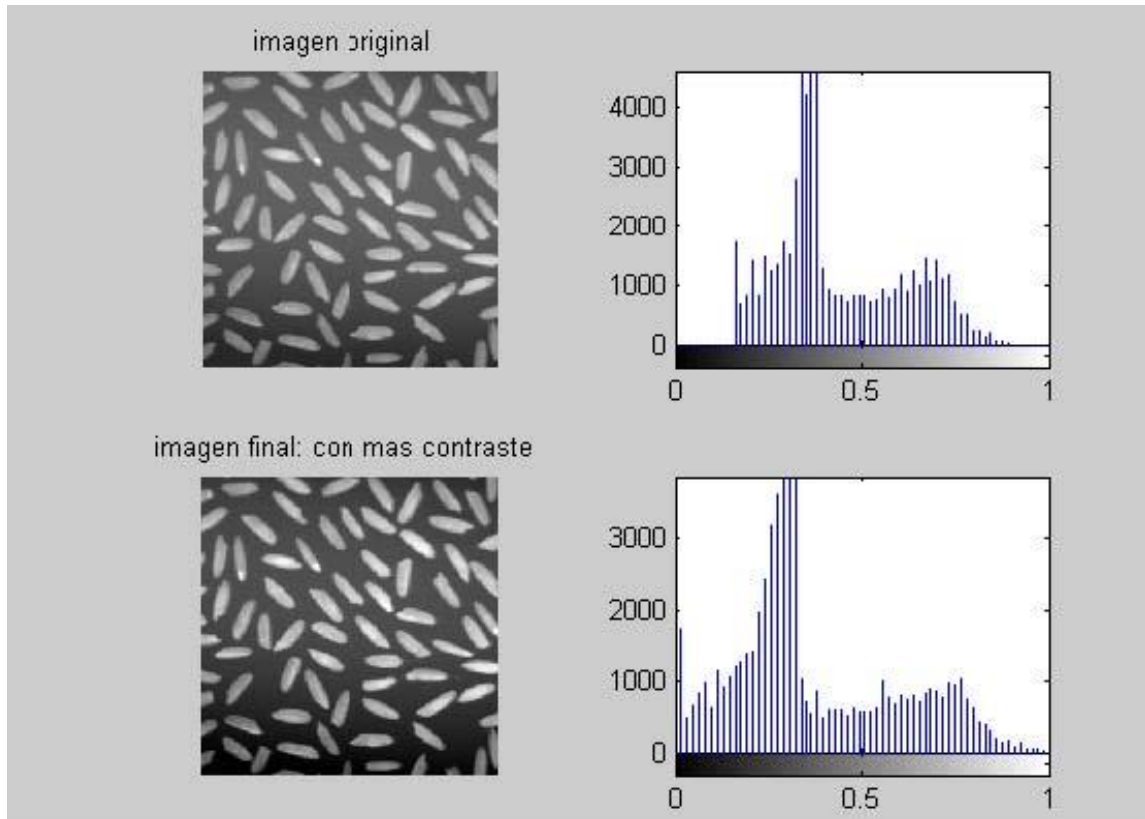
The *imadjust* function works with *low*, *high*, *bot* and *top* values between 0 and 1 (double). Therefore it is appropriate that the input image also have this type of format, as is done in the following example:

Adjust the contrast of a Grayscale image, specifying contrast limits

Test the following code:

```
I=imread('rice.tif'); % grayscale image of type uint8
I=im2double(I); % grayscale image of type double (range between 0 and 1)
J=imadjust(I,[0.15 0.9], [0 1]); %gamma, 1 by default: linear conversion
subplot(2,2,1), imshow(I); title('original image');
subplot(2,2,2), imhist(I,64);
subplot(2,2,3), imshow(J); title('resulting image more contrasted');
subplot(2,2,4), imhist(J,64);
```

As can be seen, the contrast increases (difference between the minimum and maximum gray level in the image).



Whatever the maximum and minimum values of the input image are they can be ranged between 0 and 1 in the output image by:

```
J=imadjust(I,[min(min(I)) max(max(I))], [0 1]); %Force maximum contrast
```

Similarly, the contrast can be decreased by:

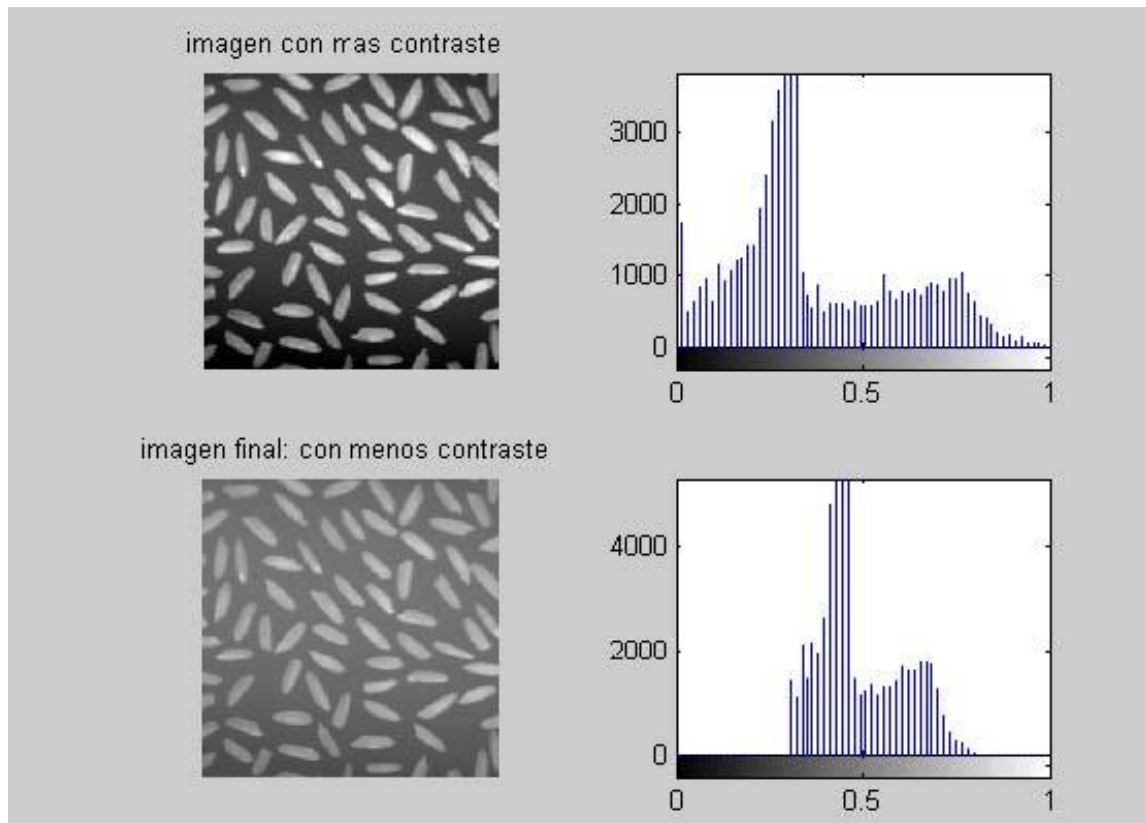
```
JJ=imadjust(J,[0 1], [0.3 0.8]); %Values between 0 and 1 are ranged between 0.3 and 0.8 to decrease the contrast
```

```
figure, subplot(2,2,1), imshow(J); title('image with high contrast');
```

```
subplot(2,2,2), imhist(J,64);
```

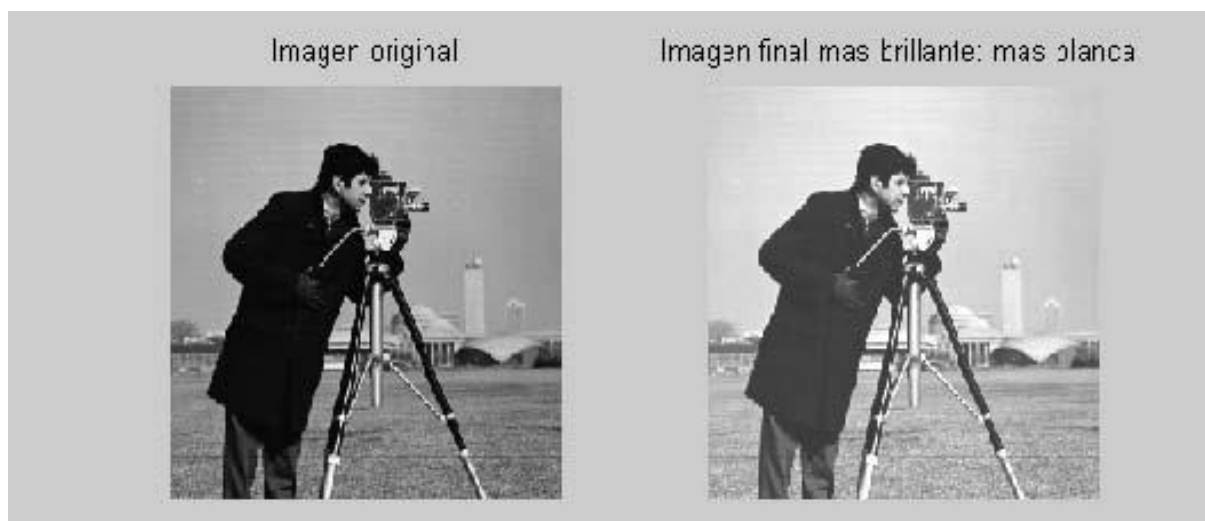
```
subplot(2,2,3), imshow(JJ); title('output image: low contrast');
```

```
subplot(2,2,4), imhist(JJ,64);
```



The brightness of an image can also be modified causing shifting of intensity values of the pixels in the histogram.

```
I=imread('cameraman.tif'); % Intensity (grayscale) image type uint8
I=im2double(I); % Intensity (grayscale) image type double (between 0 and 1)
J=imadjust(I,[0 0.8], [0.2 1]); % Increase brightness (add 0.2 to gray levels)
figure, subplot(1,2,1), imshow(I); title('Original image');
subplot(1,2,2), imshow(J); title('Output image: brighter');
```



Gamma adjust. It is an operation based on intensity association. Input intensity values

are mapped to different output values, in this case using an exponential function. If x is an input value, the output value is y such that:

$$y=x^{\gamma}$$

imadjust performs gamma correction using the following syntax:

```
J= imadjust (I, [ ], [ ], gamma);
```

where I is the image, *gamma* is the desired exponent, and the empty matrices $[]$ prevent intensities from being cropped. An example of *gamma* correction for image *forest* is shown:

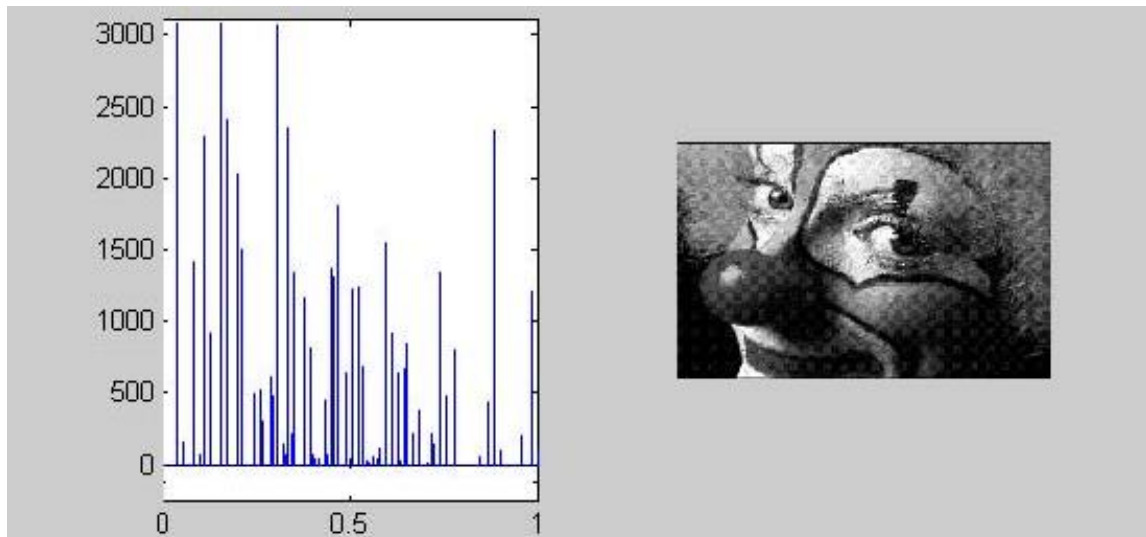
```
[X,map]=imread('forest.tif');
I=ind2gray(X,map);
J=imadjust(I,[],[],0.5);
figure, subplot(1,2,1), imshow(I); title('original image');
subplot(1,2,2), imshow(J);title('image with corrected gamma');
```



Histogram and its Equalization.

The histogram of an image is a discrete function representing the number of pixels in an image for each intensity value. Function *imhist* shows the histogram with n range of values of intensity equally distributed and for each color component. Then it calculates the number of pixels within each range. For example, the following commands display a histogram of image *clown* based on 128 intensity values.

```
load clown % Indexed image from Matlab is stored in variables X and map
I=ind2gray(X,map); % I an intensity (grayscale) image of type double ranging 0 and 1
figure, subplot(1,2,1), imhist(I,128)
subplot(1,2,2), imshow(I)
```



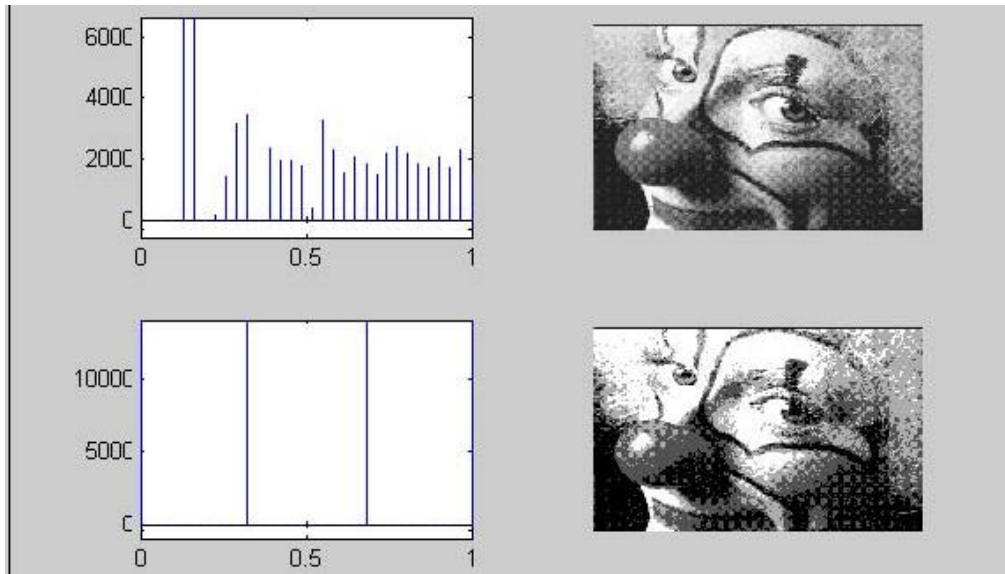
¹ Remember, with the *imfinfo* command you can obtain information about an image saved in a file. Also remember that you can see the variables used (or loaded with the *load* command) by watching the *workspace* or using the command *who* or *whos* in Matlab.

If the histogram of an image is concentrated in certain ranges of values, histogram equalization is used to redistribute the intensities, making simpler the image analysis. Histogram equalization redistributes the intensity values so that the image cumulative histogram is approximately linear. This can sometimes make the image look unnatural, but details can easily be distinguished. The *histeq* function performs histogram equalization.

Applying histogram equalization (with *histeq*) to an intensity image another intensity image is created with an approximately flat histogram. The fewer levels of output intensity we use, the flatter is the output histogram.

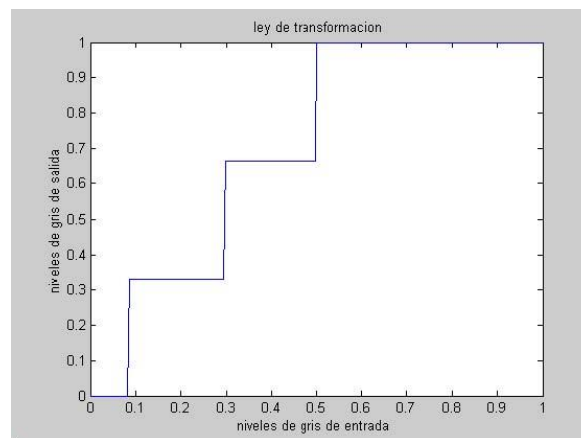
The commands below create two equalized images, one with 32 intensity values, J, and another with 4, K. Comparing the two figures we can see that the histogram of the equalized image with four intensity values is flatter:

```
load clown          %indexed image from matlab stored in X and map
I=ind2gray(X,map); %intensity image type double ranging 0-1
J=histeq(I,32);
K=histeq(I,4);
figure, subplot(2,2,1), imhist(J,32)
subplot(2,2,2), imshow(J)
subplot(2,2,3), imhist(K,32)
subplot(2,2,4), imshow(K)
```



The transfer function, i.e. output gray level versus input gray level, can also be obtained:

```
[K,T]=histeq(I,4);
% 256 input gray levels between 0 and 1 are transformed into T output levels
figure, plot((0:255)/255,T); title('transfer function');
xlabel('input gray level'); ylabel('output gray level');
```



SPATIAL IMAGE FILTERING

Filtering techniques remove image noise by filtering. The best method for a given situation depends on the image and the type of noise or degradation. Thus, for example, we have:

- Linear filters provide simplicity and speed, and are most useful when the noise is limited to a known frequency band.
- The *median* filters are very effective to remove '*salt & pepper*' noise, i.e. pixels with values 1 or 0 that originated during the image acquisition stage.

-*Wiener* filters are adaptive linear filters based on the characteristics of local variance of an image. *Wiener* filters soften an image gradually, changing areas where noise is very apparent, but keeping areas where details are present and noise is less apparent.

All functions for image restoration work with intensity images. To apply these algorithms to color images, we should process the red, green and blue components separately.

The Image Processing Toolbox supports the two-dimensional spatial filtering of finite impulse response (FIR). FIR filters have several characteristics that make them ideal for image processing in MATLAB environment:

- FIR filters are easily represented as coefficient matrices.
- Two-dimensional FIR filters are natural extensions of one-dimensional FIR filters.
- There are several well-known methods and viable for designing filters.
- FIR filters are easy to implement.
- It is possible to design *zero-phase* FIR filters. This *zero-phase* property maintains data integrity in the image, helping to prevent distortion.

A key feature for image processing is the *zero-phase* property. FIR filters are the *zero-phase* if their frequency response is a real function.

$$H(w_1, w_2) = H^*(w_1, w_2)$$

Zero-phase property also means that the impulse response is symmetric around the origin.

$$h(n_1, n_2) = h^*(-n_1, -n_2)$$

An odd-length symmetric filter with real coefficients has a real frequency response. If it is an even-length filter, it has a linear phase response.

The infinite impulse response filter (IIR) is not as suitable for image processing applications. It lacks the inherent stability and ease of design and implementation of FIR filters. Also, IIR filters are not simple two-dimensional extensions of one-dimensional filters and do not share their natural order (that is, time), which characterizes one-dimensional IIR filters. Therefore this toolbox does not provide support for IIR filters.

Understanding spatial filters in MATLAB.

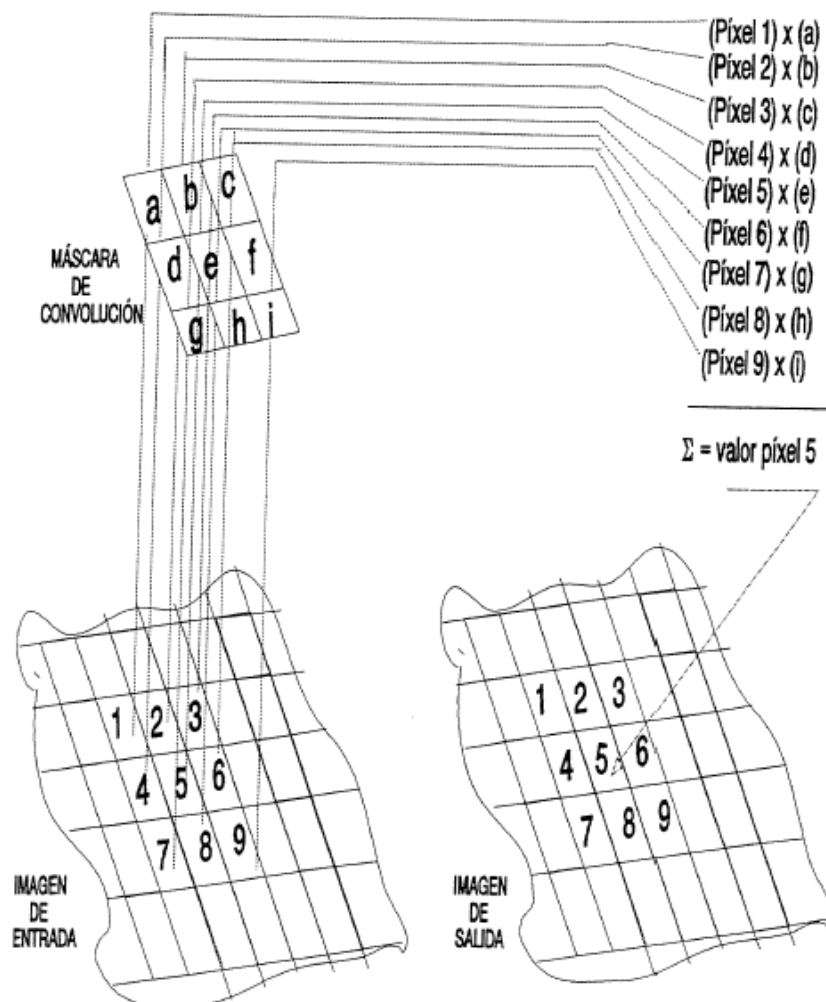
In MATLAB, a two-dimensional FIR filter is an ordered array of coefficients, $h(n_1, n_2)$, which can be introduced into a MATLAB matrix:

```
h = [ 9   4  -3  -5   6
      6  -7  12   7   0
      2   4  -1   4   1
      5   0  -2  -4   5];
```

This representation of the filter is known as filter mask. The function *filter2* from the toolbox implements a two-dimensional FIR filtering of an image using a mask. The operation that *filter2* performs is easy to picture. The mask is placed on the image overlapping some pixels of the image array. The resulting pixel of this operation is the sum of the multiplication, element by element, of each mask coefficient by its overlapped pixel in the image.

The location of the processed pixel in the output image corresponds to the location of the central pixel in the mask and depends on the shape of the mask:

- For masks with odd numbers of columns and rows, the central pixel is at the center of the mask.
- For masks with odd number of columns or rows, the central pixel is the closest to the center of the mask and in its upper left corner.



The following masks show their central pixel in bold:

```

h3x3=[1    2    3
      2    3    5
      0   -1    6];

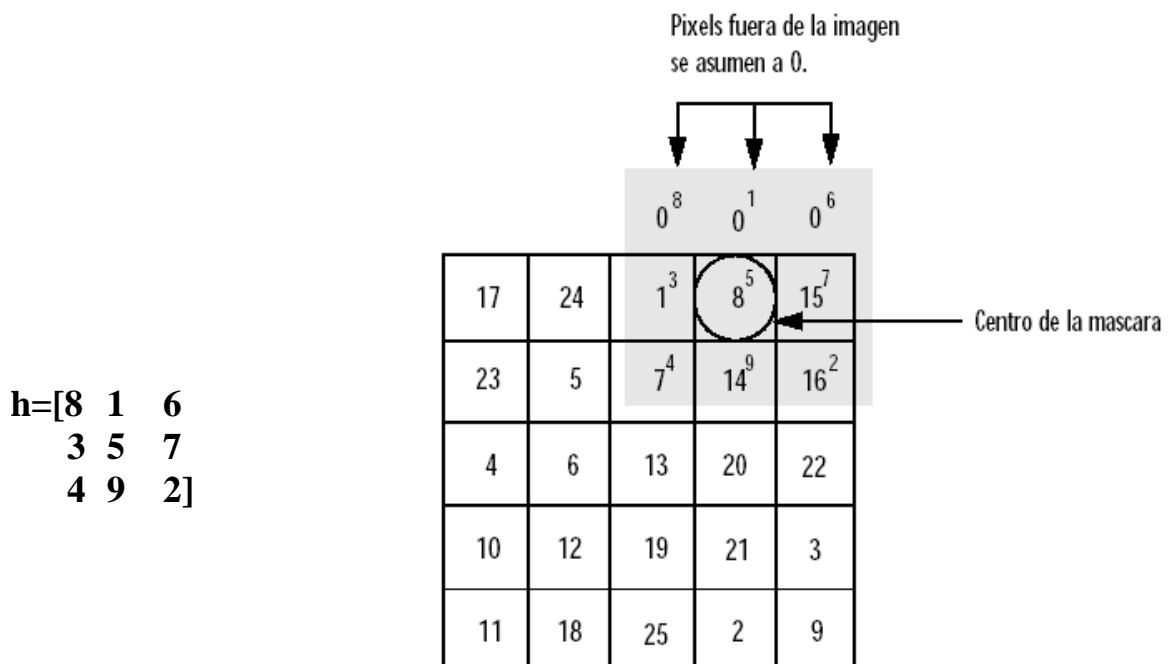
h4x3=[-1    2    5
      1    2    3
      0   -2    1
      7    6    3];

h3x4=[3    0    2    4
      8   -1    4    1
      3    2    3    1];

```

After filtering or convolving an input image of size $M \times N$ with a mask of size $P \times Q$, the output image will have dimensions of $(M+P-1) \times (N+Q-1)$, although, in many cases, only the central part of the filtering process is displayed, which has the same size of the input image.

In any case, *filter2*, in the process of overlapping of the mask, assumes zero for the pixels outside the input image (see figure below where the overlapping area of the h mask and the input image, where the multiplications element by element, is shown in gray).



Designing spatial filters

We will proceed to filter an image in the spatial domain with several masks and observe the effects they produce. First, an averaging filter is used in which the value of a pixel is calculated as the average of its neighbors (including the pixel). To do this, write a program with the following commands:


```
% spatial filtering of an image with a mask
load trees;
I=ind2gray(X,map); % It works with a gray level image type double [0 1]
h=1/9*[1 1 1; 1 1 1; 1 1 1]; % Definition of the mask. Lowpass filter
Y=filter2(h,I);
% Y=filter2(h,I,'full'); % Gets the full result of the filtering. More information: help filter2
figure, subplot(1,2,1),imshow(I),title('Original');
subplot(1,2,2),imshow(Y),title('Lowpass filtered');
```

The result¹ is shown here:



You can get the frequency response of the filter using the following function:

```
figure, freqz2(h) % type help freqz2 for more information on this function
```

Repeat the same process using the masks shown on the next page (SMOOTH, SHARPEN ...) and draw conclusions.

Keep in mind that after applying some masks it is possible that the output image (of type double), called Y in the above example, have gray levels outside the range [0—1] or [0—255] (depending on the coding for the input image). This is because some of these filters give positive or negative values whose modulus is greater than the maximum gray level when detecting an isolated point, line, etc. It is easy to see if the range of the output image is outside the range [0—1] or [0—255] by checking the minimum value in the image with `min(min(Y))` and the maximum with `max(max(Y))`.

As a workaround:

1) You can see the new image with `imagesc` which scales data, but only internally and for displaying purposes, to use all the colormap.

```
figure, colormap(gray(256)), imagesc(Y)
```

2) You can normalize the output image to the range [0-1] by subtracting the minimum value of the entire image from each pixel and dividing by the range or difference between the maximum and minimum value, and then display the image as usually with `imshow`.

¹Check the result and size of `Y=filter2(h,I,'full')` compared with `Y=filter2(h,I)`

```
n_Y=(Y-min(min(Y)))/(max(max(Y))-min(min(Y)));
figure, imshow(n_Y)
```

SMOOTH $\frac{1}{12} \begin{bmatrix} 1 & 2 & 1 \\ 1 & 2 & 1 \\ 1 & 2 & 1 \end{bmatrix}$	SHARPEN $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	HORIZ_EDGE $\begin{bmatrix} -2 & -2 & -2 \\ 0 & 0 & 0 \\ 2 & 2 & 2 \end{bmatrix}$
VERT_EDGE $\begin{bmatrix} -2 & 0 & 2 \\ -2 & 0 & 2 \\ -2 & 0 & 2 \end{bmatrix}$	LAPLACIANA $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	EDGE $\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$

In the latter case (EDGE), we have to filter the image separately with each filter and then add the results to detect horizontal and vertical edges:

```
close all
load gatin; im=ind2gray(X,map);figure,imshow(im)
h=[ 1 1 1; 0 0 0; -1 -1 -1]; % filter for detecting horizontal edges
Y=filter2(h,im); %image with horizontal edges
edges_hori=(Y-min(min(Y)))/(max(max(Y))-min(min(Y))); %normalize
figure, imshow(edges_hori);
Y=filter2(h',im); %image with vertical edges, because it was filtered with h' = vertical-edge
filter
edges_verti=(Y-min(min(Y)))/(max(max(Y))-min(min(Y))); %normalize
figure, imshow(edges_verti)
Y=edges_hori+edges_verti; %image with edges
edges=(Y-min(min(Y)))/(max(max(Y))-min(min(Y))); %normalize
figure,imshow(edges)
```

Color image filtering

To filter a color image, each of its components R, G, B must be filtered separately with the same filter (or with a different one).

```
rgb=imread('flowers.tif'); %R,G,B values in this image range between 0 and 255 (uint8)
[filas, columnas, profundidad_color]=size(rgb) %Size of the image
h=ones(5,5)/25; %Mask. Lowpass filter
r=rgb(:, :, 1); % values of r are between 0 and 255 (uint8)
g=rgb(:, :, 2); % values of g are between 0 and 255 (uint8)
b=rgb(:, :, 3); % values of b are between 0 and 255 (uint8)
n_r=filter2(h,r); % values of n_r are between 0.0 and 255.0 (double)
n_g=filter2(h,g); % values of n_g are between 0.0 and 255.0 (double)
n_b=filter2(h,b); % values of n_b are between 0.0 and 255.0 (double)
rgb1=zeros(filas,columnas,profundidad_color);
% rgb1 = image initialized to zero to store filtered components
rgb1(:, :, 1)=n_r;
rgb1(:, :, 2)=n_g;
rgb1(:, :, 3)=n_b;
```

```

rgb1=uint8(rgb1); % Convert the image to values between 0 and 255 uint8 type (instead of
double)
figure,imshow(rgb),title('Original image');
figure,imshow(rgb1),title('Filtered image');

```

You can also apply the same filter to the three components with a single function as is done in the following example, with *imfilter* function (Matlab version 6.1 or higher):

```

rgb=imread('flowers.tif'); % R,G,B values in this image range between 0 and 255 (uint8)
h=ones(5,5)/25; % Mask. Lowpass filter
rgb2=imfilter(rgb,h); % R,G,B values in this image range between 0 and 255 (uint8)
% imfilter can be used also with gray-level images instead of filter2. See: help imfilter
figure,imshow(rgb),title('Original image');
figure,imshow(rgb2),title('Filtered image');

```

Other special linear filters.

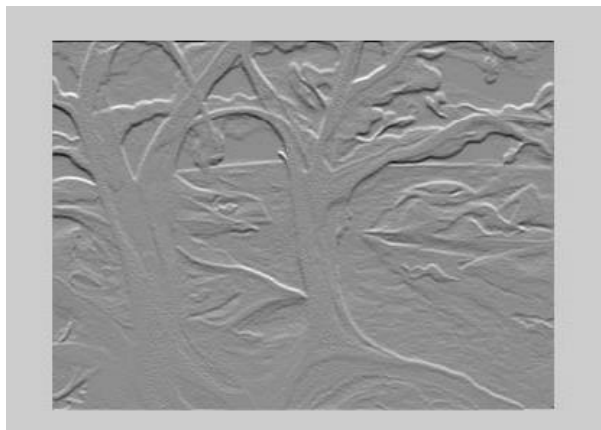
The linear filtering is often adequate to remove additive noise of limited bandwidth. We should design a specific linear filter for each situation based on the understanding of the characteristics of the existing noise. Linear filters can be designed using any of the functions for filter design previously studied, and then applying the filter with the *filter2* command.

Some filters enhance images in a special way. For example, applying the Sobel filter in a single direction, an enhancement effect can be achieved. The *fspecial* function creates various types of predefined filters. After creating a filter with *fspecial*, it can be applied to an image by using *filter2*. The following example applies a *Sobel* filter to the image *trees*.

```

load trees
I=ind2gray(X,map); % Intensity image type double (ranges 0-1)
h=fspecial('sobel') % Notice that it is a filter for horizontal edges detection
J=filter2(h,I);
figure, colormap(gray(128))
imagesc(J)
axis off

```



The image obtained after the *Sobel* filter contains values outside the range 0.0 to 1.0; *imagesc* is the best option to display it, since it matches the intensity values to the full output range. Another option is to normalize the image obtained to a range 0.0 to 1.0 (double) and then display it with *imshow*, as seen previously.

The various filters that can be obtained with the function *fspecial* are:

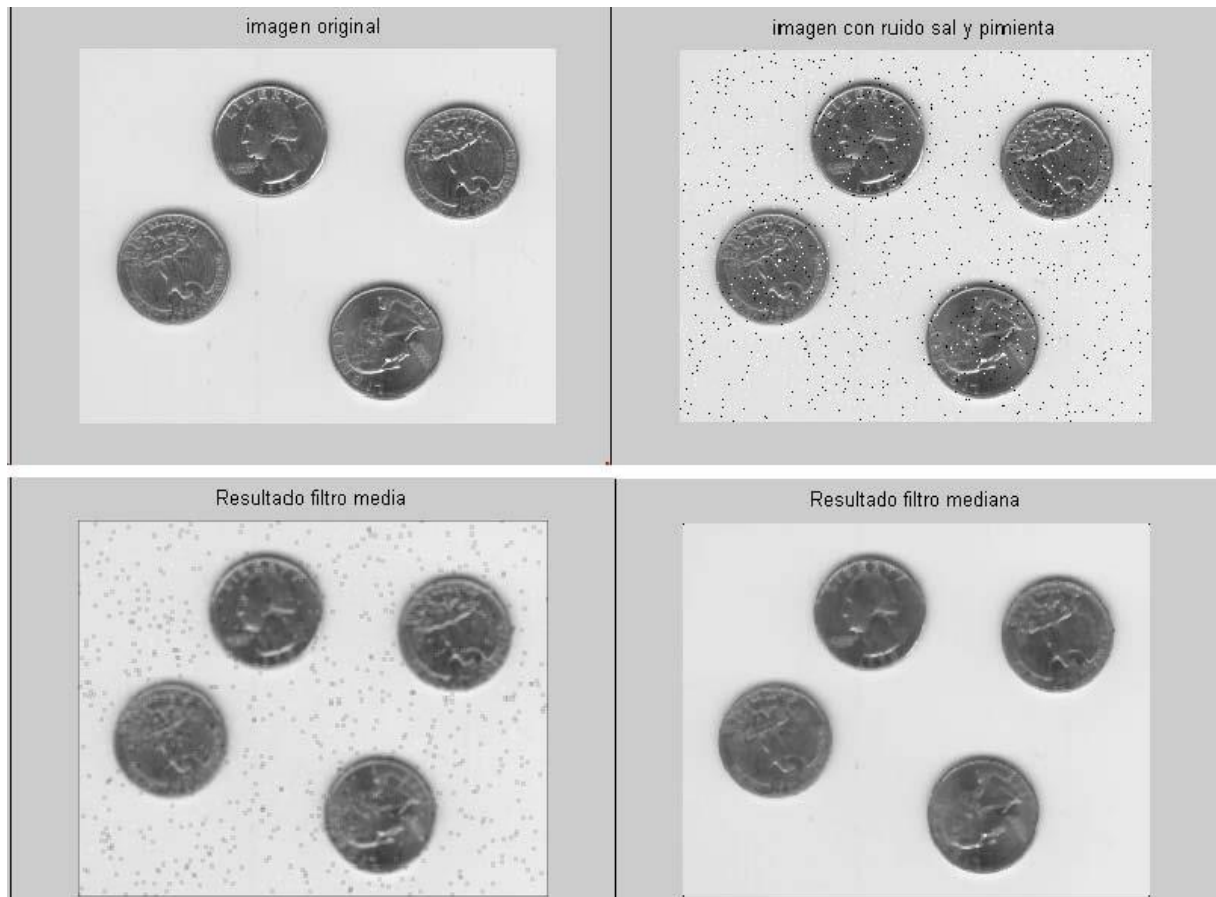
- 'gaussian' for a Gaussian lowpass filter
- 'sobel' for a Sobel horizontal edge-emphasizing filter
- 'prewitt' for a Prewitt horizontal edge-emphasizing filter
- 'laplacian' for a filter approximating the two-dimensional Laplacian operator
- 'log' for a Laplacian of Gaussian filter
- 'average' for an averaging filter
- 'unsharp' for an unsharp contrast enhancement filter

Median filtering.

The median filter (*medfilt2* function) is useful for eliminating extreme pixel values. It is a useful nonlinear filter to remove '*salt&pepper*' noise. The median filtering uses sliding neighbors to process an image, that is, it determines the value of each output pixel based on an *m*×*n* neighborhood around the input pixel. The median filter sorts the pixel values in the neighborhood and chooses the median value as a result.

As an example we will add some "*salt&pepper*" noise to an image and then filter it with an average filter and a median filter to observe the differences.

```
I=im2double(imread('eight.tif'));
J=imnoise(I,'salt & pepper',0.02); %Add 'salt&pepper' noise to image I with density 0.02
figure, imshow(I); title('original image');
figure, imshow(J);title('image with salt&peper noise');
K=filter2(fspecial('average',3),J); %Filter J with average filter using a 3x3 neighborhood
L=medfilt2(J,[3 3]); % Filter J with median filter using a 3x3 neighborhood
figure, imshow(K); title('Average filter');
figure, imshow(L); title('Median filter');
```



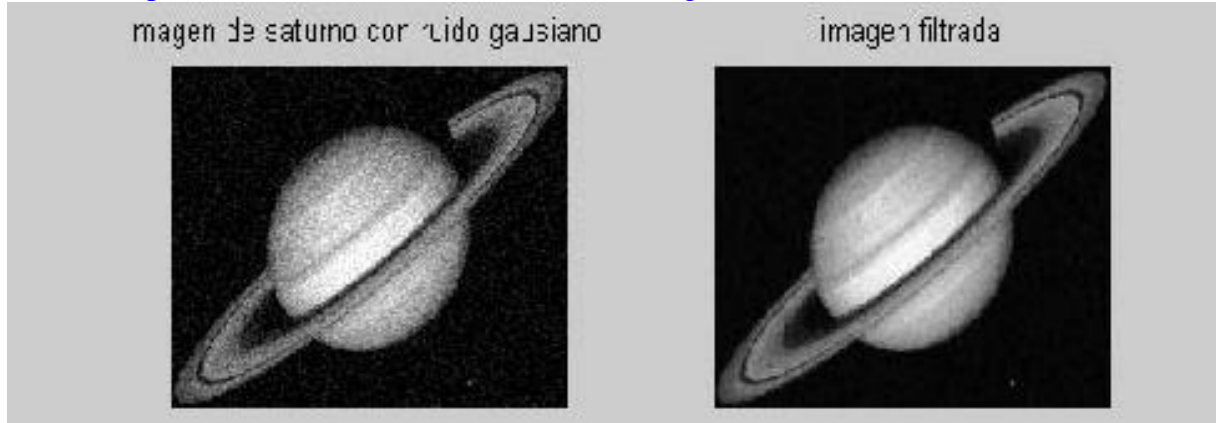
Wiener adaptive filtering.

wiener2 lowpass filters an intensity image that has been degraded by constant power additive noise. The Wiener filter often produces better results than the linear filter. Adaptive Wiener filter is more selective than the linear filter, keeping edges and other parts of high frequency of an image. Furthermore, there are no design tasks; the *wiener2* function handles all preliminary calculations, and implements the filter to an input image.

The Wiener filter gets adapted to local image variance. Where it is higher, the Wiener filter performs less smoothing. Where the variance is smaller, it performs more smoothing. Wiener filters work best when the additive noise has constant power. This example applies the Wiener filtering to a degraded image of Saturn.

```
I=imread('saturn.tif'); % Intensity image, type uint8
% Use the imnoise function to create a new image after adding noise to the
original
J=imnoise(I,'gaussian',0,0.005); % Add Gaussian noise with mean 0 and variance 0.005 to
image I
K=wiener2(J,[5 5]); % Filter noisy image J with Wiener filter using a 5x5 neighborhood
figure, subplot(1,2,1),imshow(J), title('Saturn image with Gaussian noise');
```

```
subplot(1,2,2), imshow(K); title('Filtered image');
```



FILTERING IN THE FREQUENCY DOMAIN

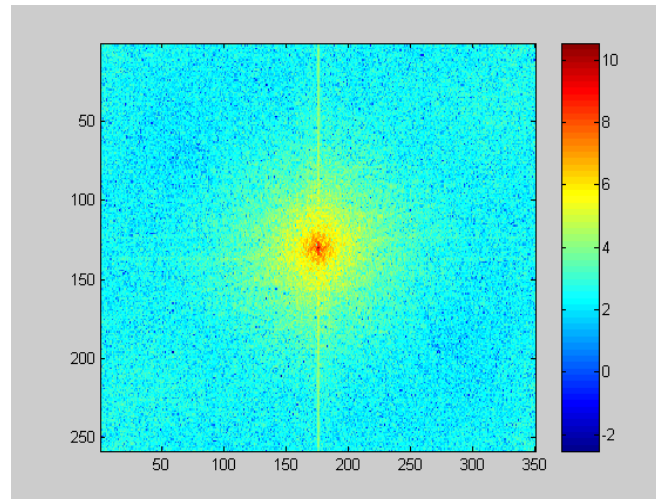
In this case the Fast Fourier Transform (FFT) and the Discrete Cosine Transform (DCT) provided by MATLAB are studied. We will see how to retrieve an image from the first coefficients of its discrete cosine transform. Then, the techniques that allow filtering operations in the frequency domain are described, explaining how a filter can be defined in that domain.

Fast Fourier Transform (FFT).

fft2 function calculates the two-dimensional fast Fourier transform (FFT). The FFT is central to many common operations in image processing. For example, the frequency response functions (*freqz2*) and transformed discrete (*dct2*) are based on the *fft2* and they use it internally.

This example obtains the two-dimensional FFT of image "trees" and displays the magnitude of the main result.

```
load trees  
I=ind2gray(X,map);  
F=fftshift(fft2(I)); %fftshift is used to center the result around F(0,0)  
figure,colormap(jet(64)), imagesc(log(abs(F))); colorbar
```

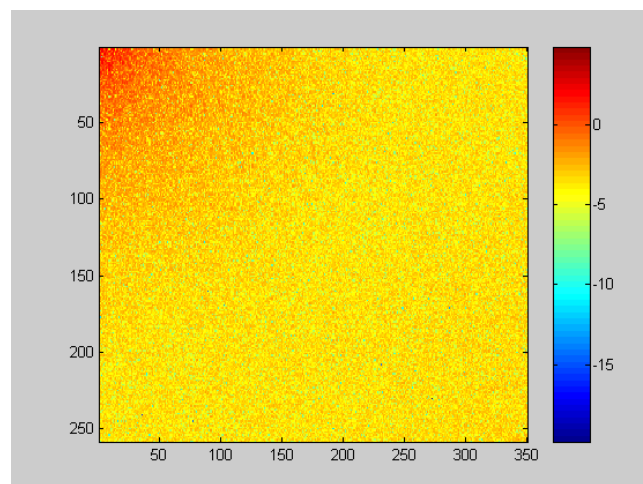
fft2 returns the frequency components for frequencies within the 0 to 2π range, with the origin (zero frequency component) in the upper left corner of the resulting matrix. *fftshift* function shifts the output of *fft2*, moving the origin to the center ($-\pi$ to π). As can be seen the transform has the same size as the input image (258x350) but now the axes correspond to frequencies. Note that most of the significant components are near the origin.

Discrete cosine transform.

The discrete cosine transform is based on the FFT, but has better energy compaction properties, making it useful for image coding. *dct2* function implements the two-dimensional discrete cosine transform.

The following example calculates the discrete cosine transform for the image "trees". Note that most of the energy is in the upper left corner. Compare the results of this transform with those of the FFT.

```
load trees
I=ind2gray(X,map);
J=dct2(I);
figure, colormap(jet(64)), imagesc(log(abs(J))),colorbar
```



The following commands set values less than 10 in the DCT matrix to zero and display the resulting image using the inverse DCT, *idct2*.

```

RGB = imread('autumn.tif');    %R,G,B values are between 0 and 255 (uint8)
I = rgb2gray(RGB);           %Convert the image to gray level, uint8 (0 black, 255 white)
J = dct2(I);                  %J is the DCT of the image (type double)
figure, imagesc(log(abs(J))), colormap(jet(64)), colorbar
J(abs(J)<10) = 0;              %Set to 0 the DCT values that are less than 10
K = idct2(J); %Calculate the inverse DCT from J. K: type double between [0 255]
figure, subplot(1,2,1), imshow(I), title('Original Image');
subplot(1,2,2), imshow(K,[0 255]), title('Image after eliminating some components
from the DCT ');
% We use imshow to display the image K of type double and range between [0 255] and not the default
[0 1]

```

Original image

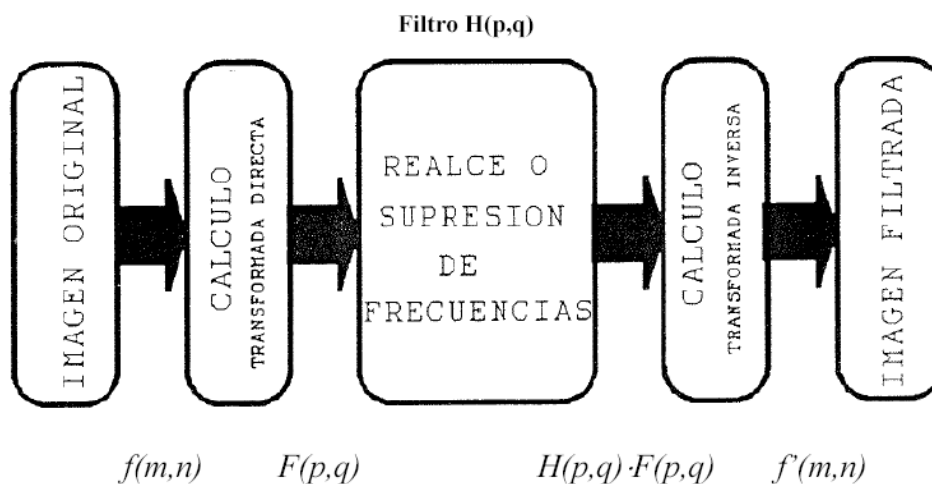
Image after eliminating some components from the DCT



In this case, we can check that 76.6% of the matrix elements from the transform are zeroed. Due to the properties of the DCT compaction, however, most of the image information is still present. Precisely for this reason, the DCT is so widely used in compression applications, such as in the compression algorithm for JPEG images.

Filtering in the frequency domain

This filtering technique is based on the calculation of the FFT of the input image. Once the frequency spectrum of the image is obtained (frequency domain), a transfer function of a filter in the frequency domain is chosen, then it is multiplied by the spectrum of the image, and the inverse FFT is applied to that result in order to obtain the output image (space domain). This process can be seen in the following figure.



To check this process we will do the equivalent to the spatial filtering we did at the beginning of this practice, but now in the frequency:

```
load trees;
im=ind2gray(X,map); % We will work with a gray-level image, double [0 1]
h=1/9*[1 1 1;1 1 1; 1 1 1]; % Setting the lowpass filter mask in the spatial domain
imagen_tras_filtro_espacial=filter2(h,im,'full'); % Space-domain filter

% Equivalent filter in the frequency domain
[M,N]=size(im);
[P,Q]=size(h);
tamano_filas_fft=M+P-1;
% This is the number of rows of the image after convolution and is the same obtained with the FFT
tamano_columnas_fft=N+Q-1;
% This is the number of columns of the image after convolution and is the same obtained with the FFT
TF_imagen=fft2(im,tamano_filas_fft,tamano_columnas_fft); %FFT of the image
TF_mascara=fft2(h,tamano_filas_fft,tamano_columnas_fft); %FFT of the mask
TF_imagen_filtrada=TF_imagen.*TF_mascara; % Multiply the FFTs obtained in the frequency domain
imagen_tras_filtro_frecuencial=real(ifft2(TF_imagen_filtrada)); %inverse FFT to obtain filtered image
figure, subplot(1,2,1), imshow(imagen_tras_filtro_espacial), title('Image after special filter');
subplot(1,2,2), imshow(imagen_tras_filtro_frecuencial), title('Image after frequency-domain filter');
```

Imagen tras filtro espacial



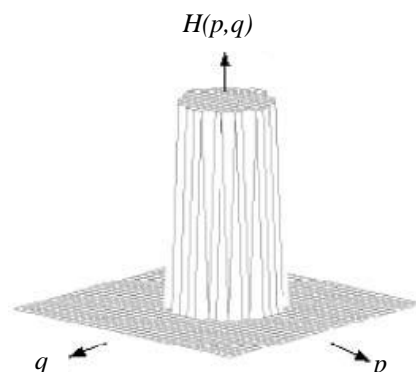
Imagen tras filtro dominio frecuencial



In the above example, we start with a filter (mask) defined in the space domain, and then its FFT was obtained to find the filter response in the frequency domain. We can also set the filter directly in the frequency domain. The fft of the input image and the filter will have both the same size, $(M+M-1) \times (N+N-1)$, so that the multiplication operation can be performed.

Ideal lowpass filter (in frequency domain):

At frequencies (p,q) within a radius around the lowest frequency $(0,0)$ the filter outputs 1, and elsewhere it outputs 0.



$$H(p, q) = \begin{cases} 1 & \sqrt{p^2 + q^2} \leq H_0 \\ 0 & \text{resto} \end{cases}$$

In the above equation the frequency point (0,0) is assumed in the position (0,0) of the

image, so that the term $\sqrt{(p^2 + q^2)} = \sqrt{(p-0)^2 + (q-0)^2}$ represents² the distance from point (p,q) to point (0,0). In the program the frequency point (0,0) is assumed to be in the center of the image and its position within the array is calculated as $[\text{floor}((M/2)+1), \text{floor}((N/2)+1)]$, and then the distance from a pixel to that central pixel is calculated.

Run the following code, headlining the images you obtain, and using different cutoff frequencies, besides that in the example (0.2*M):

```
load trees; im=ind2gray(X,map);figure,imshow(im), title('Original image');
[filas_im,columnas_im]=size(im);
N=(2*columnas_im)-1; %Number of columns of the FFT
M=(2*filas_im)-1; % Number of rows of the FFT
pp=1:N; qq=1:M;[p,q]=meshgrid(pp,qq);
% p, q are all possible combinations of pp and qq, defining all MxN possible frequencies of the filter
%Frequency 0,0 is assumed to be in the center of the array, at column floor((N/2)+1) and row
floor((M/2)+1)
D=sqrt((p-floor((N/2)+1)).^2+(q-floor((M/2)+1)).^2)<=(0.2*M);%Filter radius: 0.2*M. Sets cutt-off
frequency
% Take central frequencies around the central point of the array located at: floor((N/2)+1),floor((M/2)+1)
% To do this, we find the distance between the central point in the array (frequency 0,0) and the other
points, keeping the points whose positions are within a radius around the central point
H=zeros(M,N);
H(D)=1; %Set to 1 the central frequencies of the filter, within the filter radius around the central point
figure, mesh(p,q,H);
NIM=fft2(im,M,N).*fftshift(H);
% Multiply the Fourier transform of the image (F(0,0) in upper left corner ...
%...by the filter (shifted so that the frequency 0,0 of the filter be in the upper left corner)
figure,imagesc(log(1+abs(fftshift(fft2(im,M,N))))),colorbar %Display the FFT of the image
figure,imagesc(log(1+abs(fftshift(NIM))))),colorbar % Display the FFT of the filtered image
nim=real(ifft2(NIM)); %Perform the inverse Transform to retrieve the filtered image (space domain)
nim=nim(1:filas_im,1:1:columnas_im);figure,imshow(nim) %crop filtered image with size = input image
```

² The Euclidean distance between two points (defined in an M-dimensional space) $\mathbf{x} = [x_1, x_2, \dots, x_M]$ and $\mathbf{y} = [y_1, y_2, \dots, y_M]$, which determines the length of the straight line connecting the two points, it is defined as:

$$\text{distance} = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_M - y_M)^2}$$

Although, in our case, we work in a two-dimensional space and we find the distance between the point in the x1-y1 location (within a matrix of MxN size, which defines the MxN possible frequencies of the filter) and the point in the central position $[\text{floor}((M/2)+1), \text{floor}((N/2)+1)]$ of an MxN matrix, where the central frequency of the filter is. In short, we calculate the distance between a point location x1-y1 and the center location (frequency 0,0), to keep only those points within a circle whose radius defines the lowest frequencies of the image to be preserved.

The following line:

```
D=sqrt((p-floor((N/2)+1)).^2+(q-floor((M/2)+1)).^2)<=(0.2*M);%Filter radius: 0.2*M. Sets cutt-off frequency
```

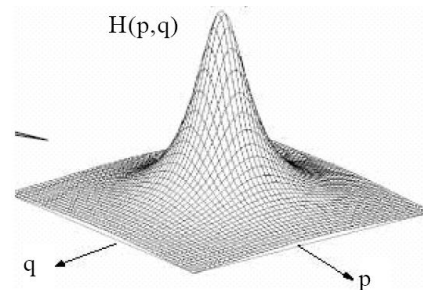
sets to 1 those points whose distance from the center point (frequency (0,0)) is less than a certain value (frequency in this example is 0.2*M). This selection defines a circle around the central point.

Finally, it is noteworthy that there are functions (like *meshgrid*) working with Cartesian coordinates (first component *x* increases rightwards and *y* downwards) and other functions (like *fft2*) work with coordinates (*row*, *column*) where the first component (rows) increases downwards and the second (column) rightwards.

Gaussian lowpass filter

In this case, the equation that defines the frequency response of the filter is:

$$H(p, q) = e^{-k(p^2 + q^2)}$$



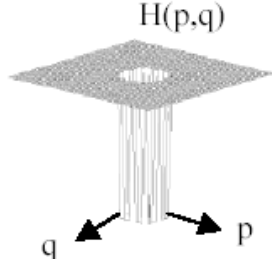
As shown in the figure, the output filter for frequency $(p, q) = (0, 0)$ is 1 and decreases towards 0, gradually, following a bell-shaped Gaussian function.

Execute the following code and title the displayed graphs. Comment the following code using different cutoff frequencies related to the width of the Gaussian bell (shown in bold in the example with a value of 0.2*M):

```
load trees; im=ind2gray(X,map);figure,imshow(im)
[filas_im,columnas_im]=size(im);
N=(2*columnas_im)-1;
M=(2*filas_im)-1;
pp=1:N; qq=1:M;[p,q]=meshgrid(pp,qq);
D=sqrt((p-floor((N/2)+1)).^2+(q-floor((M/2)+1)).^2); %Distance from each point to the central frequency (0,0)
k=1/(2*((0.2*M)^2)); % Parameter related to the width of the bell curve
H=exp(-k.*(D.^2)); %Implement the Gaussian function centered at the central point of the graph
figure, mesh(p,q,H);
NIM=fft2(im,M,N).*fftshift(H);
figure, imagesc(log(1+abs(fftshift(fft2(im,M,N))))),colorbar
figure, imagesc(log(1+abs(fftshift(NIM))))),colorbar
nim=real(ifft2(NIM));
nim=nim(1:filas_im,1:columnas_im);figure,imshow(nim)
```

Ideal highpass filter

At frequencies (p, q) within a radius around the lowest frequency (0,0) the filter outputs 0; elsewhere it outputs 1. Run the following code using different cutoff frequencies besides that used in the example of $0.02 * M$. Title every graph you display.

$$H(p,q) = \begin{cases} 1 & \sqrt{p^2 + q^2} \geq H_0 \\ 0 & \text{resto} \end{cases}$$


```
load trees; im=ind2gray(X,map);figure,imshow(im)
[filas_im,columnas_im]=size(im);
N=(2*columnas_im)-1; %Number of columns of the FFT
M=(2*filas_im)-1; % Number of rows of the FFT
pp=1:N; qq=1:M;[p,q]=meshgrid(pp,qq);
%p,q define all possible MxN frequencies of the filter
D=sqrt((p-floor((N/2)+1)).^2+(q-floor((M/2)+1)).^2)<=(0.02*M);%Filter radius. Sets cut-off frequency
% Take central frequencies around the central point of the array located at: floor((N/2)+1),floor((M/2)+1)
H=ones(M,N);
H(D)=0; %Set to 0 the central frequencies of the filter. The rest are set to 1
figure, mesh(p,q,H);
NIM=fft2(im,M,N).*fftshift(H);
% Multiply the Fourier transform of the image (F(0,0) in upper left corner ...
%...by the filter (shifted so that the frequency 0,0 of the filter be in the upper left corner)
figure,imagesc(log(1+abs(fftshift(fft2(im,M,N))))),colorbar %Display the FFT of the image
figure,imagesc(log(1+abs(fftshift(NIM))))),colorbar % Display the FFT of the filtered image
nim=real(ifft2(NIM)); % Perform the inverse Transform to retrieve the filtered image (space domain)
nim=nim(1:filas_im,1:1:columnas_im);figure,imshow(nim)
```

Methods of filter design based on frequency domain (optional for the student)

Finally, we will describe some techniques to define an H filter in the frequency domain and then calculate the two-dimensional matrix h corresponding to its impulse response, which allows filtering the image in the space domain.

This toolbox provides 3 methods to design filters:

- The frequency sampling method, which creates a filter based on a desired frequency response.
 - The window method, which convolves the desired frequency response with a window function before generating the filter.
 - The frequency transformation method, which transforms a one-dimensional FIR filter into a two-dimensional one.
- The reader may get the impulse response h of each of the filters and apply it to the image "trees" observing its effect with the command:
- ```
output_image=filter2(h,input_image).
```

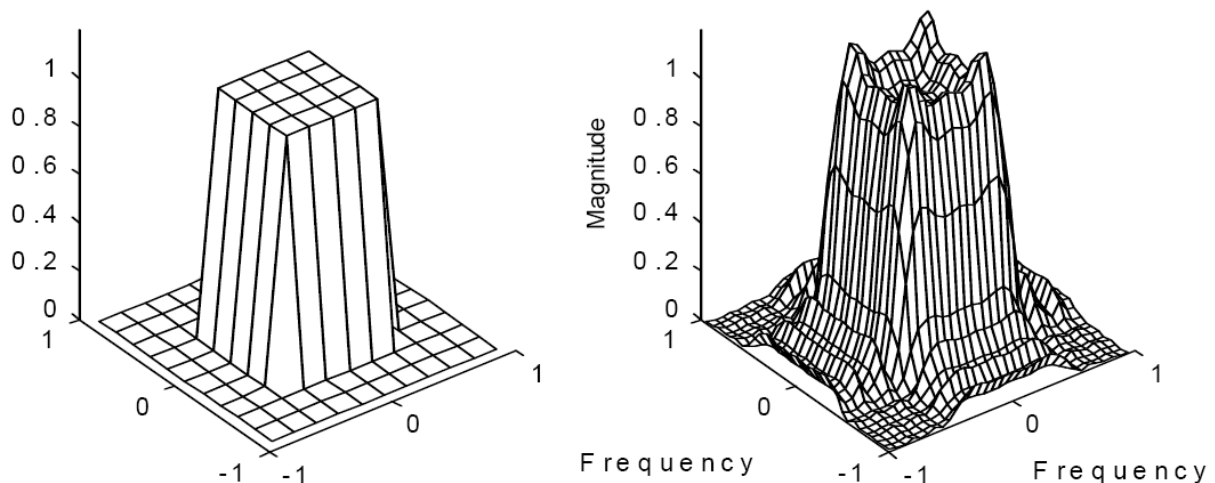
#### a) Frequency sampling method.

The frequency sampling method creates a filter based on its desired frequency response. Given an array of points that define the shape of the frequency response, this method creates a filter whose frequency response passes through those points. The frequency sampling points do not set the behavior of the frequency response between those given points; the response is usually crimped between these areas.

Function *fsamp2* from the toolbox implements a frequency sampling method to create a two-dimensional FIR filter. *fsamp2* returns an *h* filter whose frequency response passes through the points defined within the input matrix *Hd*. The following example creates an 11x11 filter using *fsamp2*, and draw the frequency response of the desired filter. Then the actual frequency response that was calculated is displayed:

```
Hd=zeros(11,11);Hd(4:8,4:8)=ones(5,5);
% The following 4 lines are to display the frequency response of the created filter
(ideal respnse)
[f1,f2]=freqspace([11 11]); % In f1 and f2 there are 11 frequencies normalized
between -1 and 1
[x,y]=meshgrid(f1,f2); % x, y are all possible combinations of f1 with f2
colormap(jet(64))
subplot(1,2,1), mesh(x,y,Hd), axis([-1 1 -1 1 0 1.2])
h=fsamp2(Hd);
subplot(1,2,2), freqz2(h,[32 32]), axis([-1 1 -1 1 0 1.2])
```

*freqz2* function, which was already discussed above, calculates and displays the two-dimensional frequency the response of a filter.



To reduce the ripple effect, the matrix defining the desired frequency response should contain more points. Filters designed using the sampling frequency method are the same size as the matrix defining the desired frequency response *Hd*.

Large filters have several disadvantages when compared with smaller filters. They require more computation time for filtering and distortion effects at the limits are higher. Smaller filters with better frequency characteristics can normally be designed using one of the other design methods. In particular, the window method obtains good results.

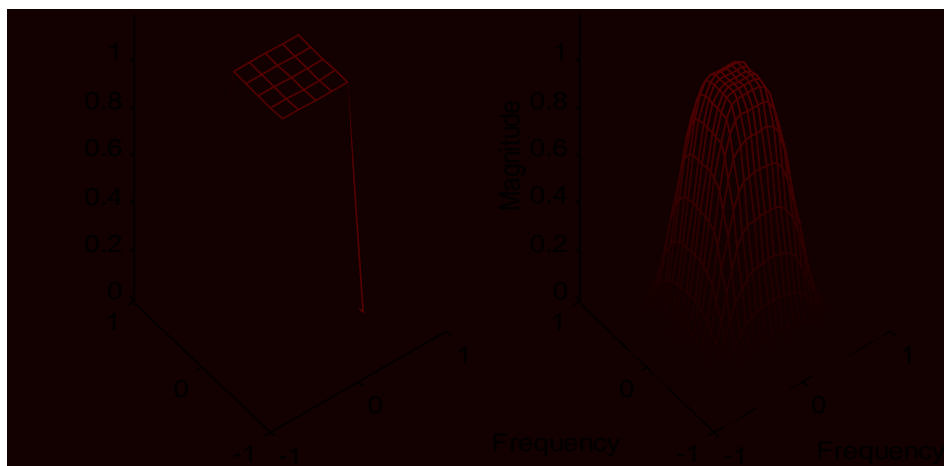
### b) Window method.

Like the sampling frequency, this method produces a window filter based on its desired frequency response. The window method, however, convolves the desired frequency response with a window function before generating the corresponding filter. Filters designed this way are softer and obtain less ripple than filters with the same size designed using the sampling frequency. Transition bands are also smoother.

The toolbox provides two functions for filter design based on the window method, *fwind1* and *fwind2*. *fwind1* creates a two-dimensional window extending a one-dimensional window. *fwind2* uses a two-dimensional window specified directly.

The following example uses *fwind1* to create an 11x11 filter of the desired frequency response Hd. Here, *fwind1* uses the *hamming* function of the signal processing toolbox to specify a circular window.

```
Hd=zeros(11,11);
Hd(4:8,4:8)=ones(5,5);
[f1,f2]=freqspace([11 11]);
[x,y]=meshgrid(f1,f2);
colormap(jet(64))
subplot(1,2,1), mesh(x,y,Hd), axis([-1 1 -1 1 0 1.2])
h=fwind1(Hd,hamming(11));
subplot(1,2,2), freqz2(h,[32 32]), axis([-1 1 -1 1 0 1.2])
```



### c) Frequency Transformation Method.

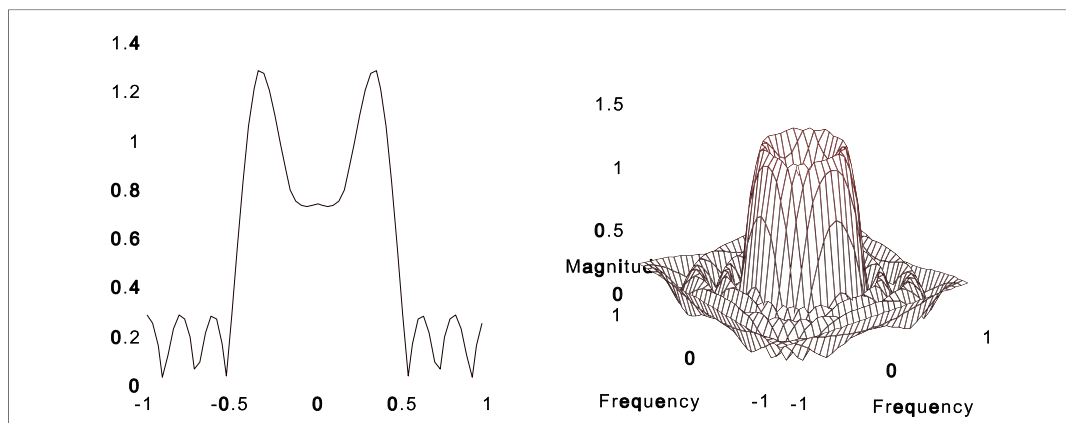
The frequency transformation method transforms a one-dimensional FIR filter into a two-dimensional FIR filter. The frequency transformation method keeps most of the one-dimensional filter characteristics, particularly the transition bandwidth and ripple characteristics. This method uses a transformation matrix to define the frequency transformation.

*ftrans2* function from the toolbox implements the frequency transformation method. The default transformation matrix of this function produces filters with nearly circular symmetry. Different symmetries can be obtained by defining the transformation matrix.

The frequency transformation method generally produces very good results, and it is easier to design a one-dimensional filter with particular characteristics than the equivalent two-dimensional filter.

The following example designs an optimal one-dimensional FIR filter with ripple equally spaced and uses it to create a two-dimensional filter with the same characteristics. The shape of the one-dimensional frequency response is clearly evident in the two-dimensional response:

```
b=remez(10,[0 .45 .5 1],[1 1 0 0]);
%Create a 10-coefficient filter. At frequency 0 it outputs 1. At frequency 0.45 it outputs 1...
h=ftrans2(b);
[H,w]=freqz(b,1,64,'whole');
colormap(jet(64))
subplot(1,2,1), plot(w/pi-1,fftshift(abs(H)))
subplot(1,2,2), freqz2(h,[32 32])
```



### Create the required frequency response matrix.

Odd-sized and real-valued arrays describing the desired frequency response can be used to ensure that the filter response is zero phase.

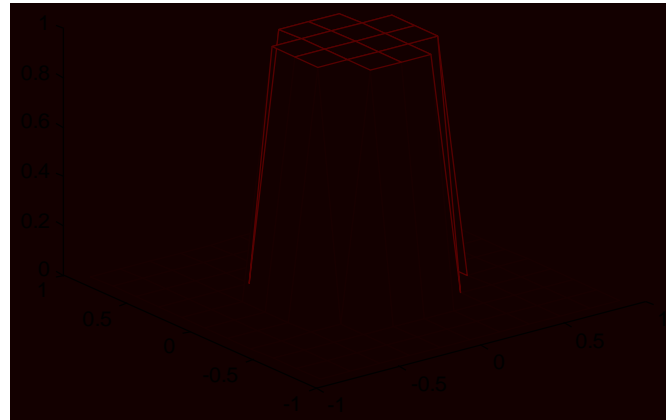
The frequency response of a symmetrical even-sized filter is linear phase, that is, the response is generally complex. Nevertheless, odd-sized filters are recommended because their symmetric and real frequency response results in a symmetric and real filter.

It is easy to create a matrix describing the desired frequency response using the *freqspace* function, which returns the correct frequency values and equally spaced for any size of response. If an array describing a desired frequency response is created by using frequency points, instead of those returned by *freqspace*, unexpected results are obtained, as a phase nonzero.



For example, to create a circular-shaped low-pass frequency response with a cutoff frequency of 0.5:

```
[f1,f2]=freqspace([11,11],'meshgrid');
Hd=zeros(11,11);
d=find(sqrt(f1.^2+f2.^2)<0.5); % Find values within the circle
Hd(d)=ones(size(d));
mesh(f1,f2,Hd)
```



### Frequency response.

The *freqz2* function calculates the frequency response for a two-dimensional filter. If no output arguments are given, *freqz2* creates a mesh grid of the frequency response. For example, consider the following FIR filter,

```
h=[0.1667 0.6667 0.1667
 0.6667 -3.3333 0.6667
 0.1667 0.6667 0.1667];
```

To calculate and display the frequency response of *h* with 64x64 points:

```
freqz2(h)
```

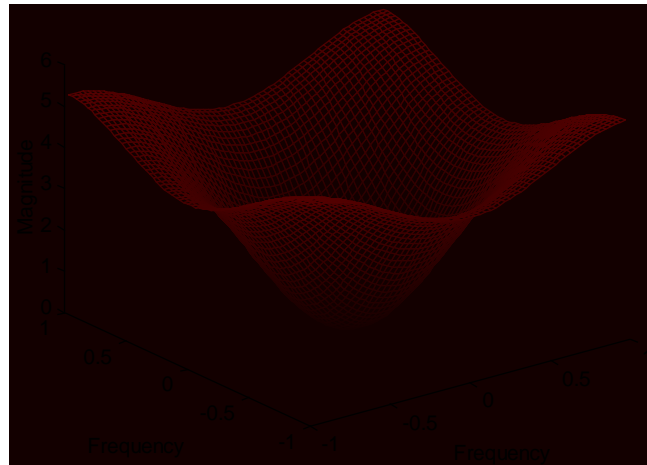
To obtain the frequency response matrix *H* and the vectors *w1*, *w2*, defining the frequency points, then the following output arguments are used:

```
[H,w1,w2]=freqz2(h);
```

*freqz2* normalizes frequencies *w2* and *w1* where the Nyquist frequency is  $\pm 0.5$  for the two-dimensional case.

For a simple *m*×*n* response, as shown above, *freqz2* uses the two dimensional version of the fast Fourier transform, *fft2* (see figure below).





## PART 3: OTHER HELPFUL OPERATIONS WITH THE IMAGE TOOLBOX

The techniques for image analysis provide information about the data that make up an image. Here are some types of analysis operations that support this toolbox, along with a brief description of each.

### Intensity profile.

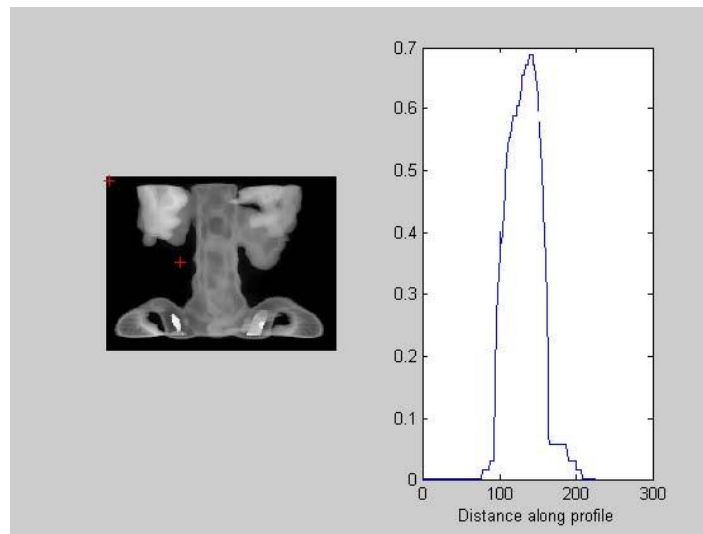
The command *improfile* calculates and traces the intensities along a line or path within an image. We can provide the coordinates of a line or lines as input arguments, or we can define the desired path with the mouse.

If we call *improfile* with no arguments, the cursor changes into a cross when it is over the image. Then we can define line segments as follows: select the ends of the line by pressing the left button. To end press the right button or press Return.

The function *improfile* may return the coordinates of the different points the lines pass through (x,y), their intensities (c), and the coordinates of the ends of the segments (xi, yi). To display the results we can use *plot3(x,y,c)*. The following example displays image *spine* and the trace of intensities along a line of the image with the *improfile* command.

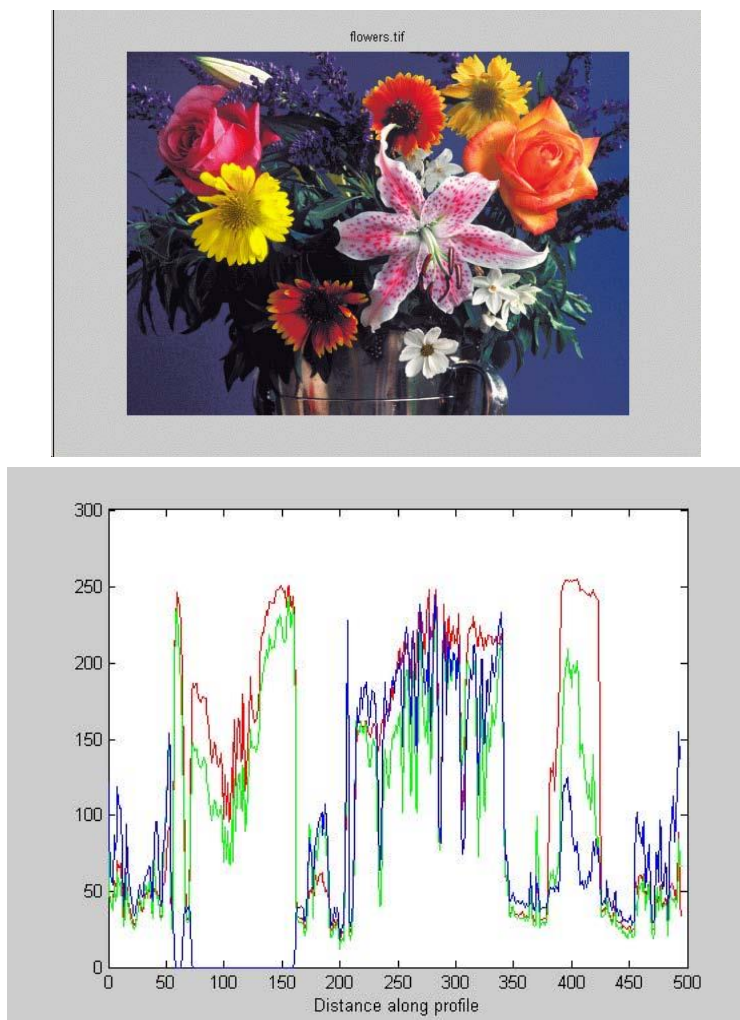
```
load spine % load indexed image which is saved in variables X and map
I=ind2gray(X,map); % I is now an intensity image type double between 0 and 1
figure, subplot(1,2,1), imshow(I);
%Paint a cross symbol on image I at coordinates (x1,y1)=(7,8) and
(x2,y2)=(156,179)
hold on, plot([7 156],[8 179],'r+');
%Draw gray levels (intensity) along the line (x1,y1)=(7,8) and (x2,y2)=(156,179)
subplot(1,2,2), improfile(I,[7 156],[8 179])
```

In the last line of code above, the first bracket contains the  $x$  coordinates of successive points that determine the lines, and the second bracket contains the  $y$  coordinates (x-axis increases rightwards and y-axis downwards).



In an RGB image intensity profile of the R, G, B components would be displayed in the same way with command *improfile*:

```
figure, imshow flowers.tif;
improfile; % Select with the mouse endpoints of the line you want to see the
intensity profile
```



### Pixel selection.

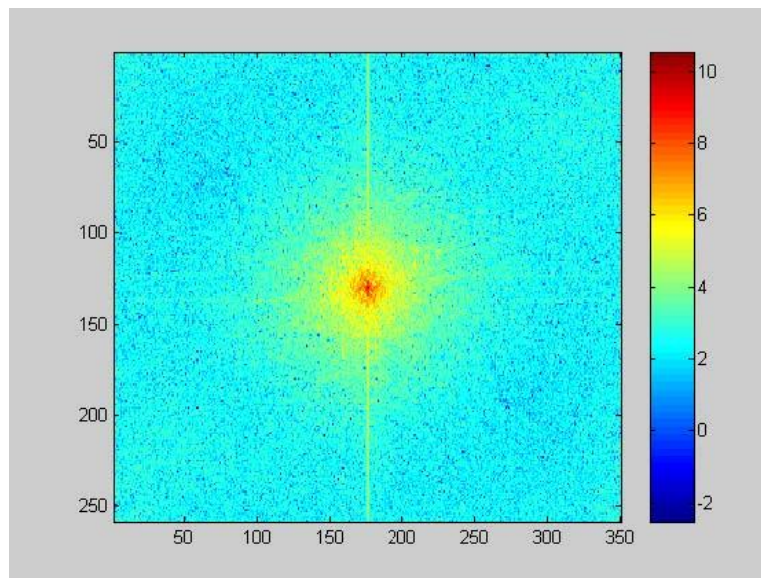
The *impixel* command gets the color value of the pixel or a set of selected pixels. If no input parameters are given, *impixel* lets you use the mouse to select points with the left button. When Return or the right button is pressed, the last pixel is added and the color values of these pixels are returned in an  $m \times 3$  array. With the coordinates of the pixels as input arguments, *impixel* returns a list of color values for the given pixels. Example:

```
load clown vals=impixel(X,map,[231 35],[122 50]) %impixel(X,map,[x1 x2 ...],[y1 y2 ...])
% It returns in vals the R,G,B values of the selected pixels
% vals = 0.1250 0 0
% 0.8672 0.4141 0
```

### Color bar.

The *colorbar* command adds a color bar to any axis. Color bars relate the colors in an image with physical values, such as temperature, etc. Here is a typical use for *colorbar*:

```
load trees
I=ind2gray(X,map);
F=fftshift(fft2(I));
figure, imagesc(log(abs(F)));colorbar,colormap(jet)
```



*colorbar* function works for any type of graph, and can be placed both horizontally or vertically. If no arguments are given, *colorbar* updates an existing color bar.

## Exercises

1. Experiment more filters with different masks defined in *fspecial* function and draw conclusions about their operation.
2. Perform the following operations related to image enhancement:
  - 2.1. Load and display image 'pout.tif'
  - 2.2. Check the image in memory (within the Matlab's work space)
  - 2.3. Obtain its histogram
  - 2.4. Equalize its histogram and display the equalized image
  - 2.5. Write the image to a jpg file
  - 2.6. Retrieve the original image
  - 2.7. Maximize its contrast
  - 2.8. Add some Gaussian noise
  - 2.9. Obtain the FFT of the noisy image
  - 2.10. Filter the image optimally (using the frequency transformation method. Whereas the optimal filter in the spatial domain corresponds to Wiener filtering)

## PART 4: NON-CONTACT ACTIVITY (Homework)

Perform all steps and present a report to the professor about the entire lab practice, including both contact- and non-contact parts. It should always be indicated both the commands used and the result. In addition, expand the report with the explanations you deem appropriate. All images we will use in this activity must first be downloaded from the Web page of the course.

- 1.- Read image *therese.gif* and indicate its format (RGB or indexed color map).
- 2.- Display the previous image.
- 3.- Convert the previous image to a grayscale image (containing only luminance information) and store it in an array called *i*. Indicate the range of values in which the luminance (black→white) of the resultant image is quantified.
- 4.- Find out the dimensions of our image *i* (rows x columns).
- 5.- Display the histogram of the gray scale image.
- 6.- Modify the brightness (adding a constant) and contrast (multiplied by a constant) of the grayscale image and display both results.
- 7.- Perform a binarization of image *i* with a threshold whose value is approximately half of its range of gray scale and display the result.
- 8.- Get the negative of the grayscale image obtained in point 3.

**9.-** Obtain three noisy versions of the image *i*: (a) with uniform distribution noise 'speckle', (b) with 'salt and pepper' noise with 5% noise density, (c) with Gaussian noise with zero mean and 0.01 variance.

**10.-** Based on the above image contaminated with Gaussian noise, filter it using the neighborhood average, first with a 3x3 neighborhood and then with a 5x5, and display the results.

**11.-** Read image *venas.tif* and rotate it a 30-degree angle.

**12.-** From the original image *venas.tif*, get 16 images in different trials with Gaussian noise of zero mean and variance 0.01. Save them in files with their respective names (*noisy1.bmp*, ..., *noisy16.bmp*).

**13.-** Perform some tests on noise filtering using the technique of image averaging on the images created in the previous point. To do this, try it first with 4 noisy images, then 8, and finally with 16. Compare results visually.

**14.-** Create an image with Paint, composed entirely of an intermediate greyish background luminance and save it to a file. Complete the following steps in Matlab:

- read the image with Matlab, convert it to grayscale and display its histogram
- contaminate the grayscale image with 'speckle' noise and store the result in another image file
- observe the histogram of the noisy image and confirm that the noise has a uniform distribution
- ask for help on the *imnoise* command to see how we can change the noise variance and contaminate the image with speckle noise, using other variances, and see what happens with its histogram.

**15.-** Create a dark, very poorly contrasted image, preferable from a photograph whose gray levels are multiplied by 0.1. Then display that image and its histogram. Finally, obtain the logarithm of the image, calculating previously the optimum value of K that maximizes its output gray level. Display the resulting image and observe the enhancement obtained.

**16.-** Read image *bazo.bmp* and display its histogram. Check that it is an image poorly contrasted.

**17.-** Convert the image above to grayscale and then equalize its histogram. Display the resulting image and its histogram, contrasting the differences with the image of the previous point.