**Chapter 4**

**Image Enhancement**

**using Neighborhood Operations**

* For more useful and interesting operations, we must consider not only the pixel, but also the neighboring pixels.
* The output image ***g(x,y)*** must depend on the input ***f(x,y)*** at ***(x,y)***, and values in an area surrounding (**x,y).**

**4.1 Convolution**

* The convolution operation is the basis of various neighborhood processing techniques.

M(-1,-1) M(-1,0) M(-1,1)

M(0,-1) M(0,0) M(0,1)

M(1,-1) M(1,0) M(1,1) M(1,1)

M(-1,1) M(0,1) M(1,1)

* Convolution requires defining an ***mm*** matrix,***M[j][k]*** (or ***M(j,k)***), ***j, k*** = ***– m/2*** to ***+m/2***, whose entries determine the type of neighborhood operation. This matrix is called by various names such as filter, window, mask or kernel.

Fig. 4.1 A 33 mask

* Convolution involves laying the mask over the image, multiplying each entry (coefficients) of the matrix by the pixel value under the mask, and then adding the resulting values for all entries. For an input image ***f(x,y)***, and a 33 mask ***M*** centered at ***(x,y)***, the pixel value of the output image at location ***(x,y)*** becomes

***g(x,y) = M(-1,-1) f(x-1,y-1) + M (0,-1) f(x,y-1) + … + M (0,0) f(x,y) + …***

***+ M (1,1) f(x+1,y+1)***  (4.1)

An example of the mask is neighborhood averaging where  in which the output value is the average of values for the pixel at ***(x,y)*** and its eight neighbors.

For an ***mm*** mask, the above equation becomes

 (4.2)

**Note**: Strictly speaking, the above operation is correlation, and for convolution we must flip the mask and then perform the multiplication, i.e. the product inside the summation must be ***M (j, k) f( x –j ,y –k ).*** However, since most neighborhood masks are symmetrical, the convolution and correlation give the same results.

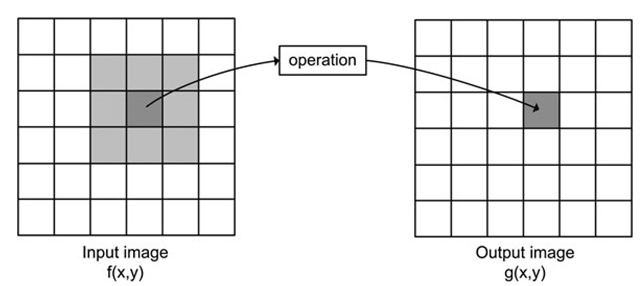


Fig. 1 Neighborhood pixels

The convolution algorithm for a single pixel is

Create an array M indexed from 0 to m –1, horizontally and vertically;

Fill M with the mask coefficients;

halfm = floor (m/2);

sum = 0;

for (k= - halfm to halfm)

for (j = - halfm to halfm)

sum = sum + M(j+ halfm, k+halfm)\*f(x+j, y+k);

g(x,y) = sum;

* Since convolution is such a basic operation in image processing, and in order to simplify the notation, we denote the convolution operation by ***©*** and write (4.2) in the compact form

***g(x,y) = M (j, k) © f(x+j, y+k)*** (4.3)

**4.1.1 Computational Considerations**

1. **Range of values**: The summation in (4.2) can produce values that are outside the range, e.g. for an 8-bit image ***g(x,y***) the summation can become more than 255. It can even produce a negative value if mask coefficients are negative. In order to resolve this, we can scan ***g(x,y)*** find the minimum and maximum values, and scale the values (essentially perform a linear mapping) so that the values are within the acceptable range, e.g. 0 to 255.
2. **Border pixels:** For border pixels some the mask elements will lie outside the image, as shown in Fig. 4.1

Image

3x3 mask

3x3 mask3x3 mask

Fig. 2 Filter outside

the image

**Pad with Zeros**: Do not perform convolution on the border pixels, and set the output values for these border pixels to zero (black).

**Replicate**: Copy the input pixel values of the border pixels to the output border pixels without applying the convolution.

**Ignore borders**: Reduce the size of the output image from hw, by half the size of the mask, to (h-m/2 -1)(w-m/2 -1).

**Ignore outside pixels**: Ignore the part of the mask outside the image, and perform convolution based only on the overlap area.

Algorithm for implementing the first option (i.e. ignore borders)

Create the desired mm mask

Compute halfm = m/2

for (all pixels coordinates x and y) // this is in fact two for loops

g(x,y) = 0; // initialize the output image

for (y = halfm; y < h – halfm-1; ++y)

for (x = halfm; x < w – halfm-1; ++x)

compute g(x,y) using the convolution algorithm given above;

1. **Complexity considerations**: Convolution is computationally costly. For a mask of size mm and an image of size nn, it requires

 multiplication +  additions giving a complexity of  for the image or per pixel (why?).

* The following discussion shows that in some cases it is possible to reduce this complexity to per pixel.
* Convolution is a linear operation. This property means that for an image ***f*** , amask ***M*** and a constant scalar ***k***, we have

***(k f ) © M = k (f © M)*** (4.4)

and

***(f1 + f2) © M = f1© M + f2 © M*** (4.5)

This assumes that no border or saturation effects are present (why?).

* In some cases, the mm mask matrix ***M*** can be decomposed into the product of two vectors, an m1 vector and a 1m vector . For example the neighborhood averaging mask can be decomposed (separated) as follows:



Now using the linearity property of the convolution, we can deduce the following (show it)

***f ©*** ***©*** ***©***  (4.6)

Each of the two convolutions requires ***m*** multiplication and (***m-1)*** addition, per pixel; giving a total of ***2m*** multiplication and ***2(m-1)*** additions per pixel (why?). This proves that in the case of a separable mask, the complexity has been reduced from to per pixel, and to for the image.

The algorithm for performing convolution with a separable mask is (why?)

Decompose the mask matrix into vectors Mx and My

Compute halfm = m/2

Initialize the output image g(x,y)

Initialize a temporary image temp(x,y)

for (y = halfm; y < height – halfm; ++y) {

for(x = 0; x < width – 1; ++x) {

sum = 0;

for( i = – halfm; i <= halfm; ++i)

sum = sum + My (i + halfm)\*f(x, y +i )

temp(x,y) = sum;

} }

for (y = 0; y < height – 1; ++y) {

for(x = halfm; x < width – halfm; ++x) {

sum = 0;

for( i = – halfm; i <= halfm; i++)

temp = sum + Mx (i + halfm)\*temp(x +i , y);

g(x,y) = sum; } }

* + 1. **Filtering in Matlab**

The *imfilter* function does the linear filtering with the specified kernel (window) K as

>>imfilter(image, K, ‘option’);

Where option is can be:

‘replication’ that is borders with repeating the border pixels

‘full’ that is padding with zeros and then applying the filter.

We can also use imfilter(image, K) and it will chose ‘full’. Instead of specifying our own kernel (filter) we can use the special filters provided by Matlab, for example

>> *fspecial(‘average’, [5,7])*

will return an averaging filter of size 5x7.

*>> fspecial(‘average’, 7)*

will return an averaging filter of size 7x7. Without any number it will return a 3x3 averaging filter. We will discuss other special filters later.

Example:

*>> hm=imread('home.jpg');*

*>> filt=fspecial('average');*

*>> home.filt=imfilter(home,filt);*

*>>imshow(home.filt)*

The above applies a 3x3 averaging filter to the image below and produces a burred image.

1. (b)

Fig. 3 Average filtering, (a) Original image (b) image after filtering

* 1. **Edge Detection**
* Edge detection is the most common approach for detecting meaningful discontinuity in gray level, and is used frequently for recognizing regions in an image.
* An edge is a boundary between two regions with relatively distinct gray levels, or colors.
* The idea behind most edge detection techniques is gradient, which is a derivative operator and detects changes.
* In the areas where gray level does not vary, there is no change and the resulting derivative is zero. These areas appear as black in the output image.
* At the boundaries the change is abrupt and significant, and the derivative is maximum, producing high gradient which corresponds to white boundaries.

(a) (b)

Fig. 4 The image (a) and its derivative (b).

The gradient of a continuous function ***f(x, y)*** is defined as

 (4.7)

* Note that for an input image ***f(x,y)***, the gradient operator produces two images, and .
* The gradient operator ***g*** can also be represented by gradient magnitude and gradient direction as (see Fig 4.3)

 (4.8)

Fig. 5 Definition of various gradients

 ***g***

The square root in (4.8) is relatively expensive computationally, and it is sometime replace by



If ***f(x, y)*** is an image function, then the smallest increments  or  is 1 pixel, and the gradient in ***x*** and ***y*** direction (4.7) become

 (4.9)

To find the gradients using (4.9) we must convolve the image with the masks  and  (why?), i.e.

***©***  (4.10)

***©*** 

**4.2.1 Prewitt and Sobel Edge Operato**rs

* An alternative to derivative approximation

 (4.11)

The operations in (4.11) are equivalent to convolving the image ***f(x,y)*** with the masks (why?)

**M*x* = [-1 0 1]** and  (4.12)

* The gradient is calculated using values of two pixels, one just before the current pixel ***(x,y),*** and one immediately after the current pixel.
* If one of these pixels has an error due to noise, then the gradient will be in error. To avoid this problem, Prewitt suggested that the gradient be calculated based on six surrounding pixel values, with the corresponding masks as

 and  (4.13)

* A similar set of masks giving more emphasis on the center pixel is proposed by Sobel as follows

 and  (4.14)

(a) (b) (c)

Fig. 6 (a) image, (b) gradient magnitude , and

(c) gradient magnitude  both using Sobel masks



(a)

1. (c)

Fig. 7 (a) original image, (b) Gradient , (c) Gradient 

* Other gradient operators can be devised for specific applications. For example, Robert proposes the following gradient definitions

 (4.15)

The corresponding Robert’s masks are  and . These masks are sensitive to slanted edges (why?). The gradient magnitude is obtained as

 (4.16)

Matlab functions for Prewitt and Sobel horizontal filters are *imfilter(‘prewitt’*) and *imfilter(‘sobel’)*:

*>> sobl=fspecial('sobel');*

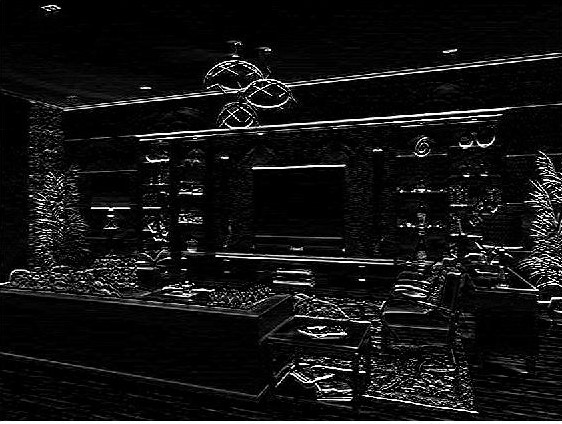
*>> fur=imread('Furniture1.jpg');*

*>> fur.sobel.Hor= imfilter(fur,sobl)*

For vertical filter use transpose of filter: >> *fur.sobel.Ver= imfilter(fur,sobl’)*

****

(a)

** **

(b) (c)

Fig. 8 (a) Furniture1 (b) edged detected by Sobel Horizontal filter, (b)Home edged detected by Sobel vertical mask

**Scaling Transformation**: After applying a filtering the resulting filtered image me have either negative values and values out of range of 0 to 255.

In order to bring the values within the range we apply the following transformation. Let the minimum values of the pixels be Lo, and the highest value be Hi, then the following will bring the values to the range.

* + 1. **The Laplacian Operator**
* The edge detectors discussed in the previous section are based on the first order derivative of the image function. Such edge detectors are useful for finding sudden changes in the gray level.
* If changes in gray levels are not strong, the second order derivatives can be used for edge localization.

The Laplacian of an image function ***f(x,y)*** is defined as

 (4.17)

In digital imaging the derivatives in (4.17) are approximated as

 (4.18)

After finding the second derivative using (4.18), the mask for the Laplacian becomes (why?)

 (4.19)

* Laplacian is rarely used on its own for edge detection since it is prone to corruption by noise due to the second derivative. It is used in conjunction with a Gaussian filter to first reduce the noise (and blur the image), followed by the application of the Laplacian.
  + 1. **Edge Thresholding**
* Once a method for approximating the gradient is selected,, then further processing must be applied to obtained clear edges.
* To remove weak edges (those values that have low gradient),

 (4.20)

This will provide a black background, and the gradient is shown as a varying gray level depending on the edge strength.

* Other thresholding schemes can also be applied. For example, the image in (4.20) can be inverted (negatized) to show gradient on a white background. Alternatively, the gradient image can be binarized as

 (4.21)

* Finally, edges can be placed on the image so that the image appears as the background for places where there is no strong edge, i.e.

 (4.22)

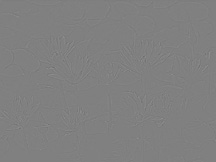
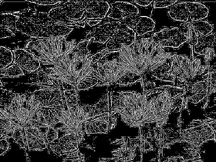
1.  (b)





(c) (d)

Fig. 9 The original image of Lilies (a), gradient magnitude image (b), gradient magnitude thresholded with T = 30 (c), and then negatized (d).



(a) (b)

Fig. 10 (a) Application of the Laplacian to blurred lilies 9(a), and (b) after thresholding with T = 128.

Matlab has a function *edge* that takes an intensity (grayscale) or a binary image Im as its input, and returns a binary image of the same size as Im, with 1's are where the function finds edges in I and 0's elsewhere. *edge* supports several different edge-finding methods:

The Sobel method *edge(Im,'sobel')* finds edges using the Sobel approximation to the derivative. It returns edges at those points where the gradient of Im is maximum. *edge(Im,'sobel', Thresh)* find the edges with values greater than *Thresh.* The corresponding methods for Prewitt and Roberts are *edge(Im,'prewitt',Thresh)* and *edge(Im,'roberts',Thresh)* .

* 1. **Linear Filtering For Noise Reduction**
* Filtering generally refers to reducing or removing image noise and enhancing the quality of the image.
* Images are corrupted with noise due to poor sampling, image acquisition and transmission. In this section we study spatial domain filters.
* These filters are described based on the spatial character of the image, i.e. gray level or color values are described as a function of (x,y) location.

**4.3.1 Types of noise**

We characterize image noise by its histogram, which is essentially the probability distribution.

**Gaussian Noise**: This noise is very common and models natural noise processes such as electronic noise. Its normalized histogram is described by the Gaussian distribution

 (4.23)

where ***l*** is the gray level (usually ***0<= l <= 255*** for an 8-bit gray level image),  is the average input image gray level, and  is the standard deviation of input gray levels. The histogram is bell shaped.

***h***

Fig. 12 Histogram of the

Gaussian distribution noise

0 ***lave*** ***l***

**Uniform Distribution**: This type of noise is the most unbiased natural noise model. The noise histogram is evenly distributed (has a flat shape),

 (4.24)

where ***a*** andb are some constants between 0 and 255 for an 8-bit image. The mean value is (a+b)/2, and the standard deviation is .

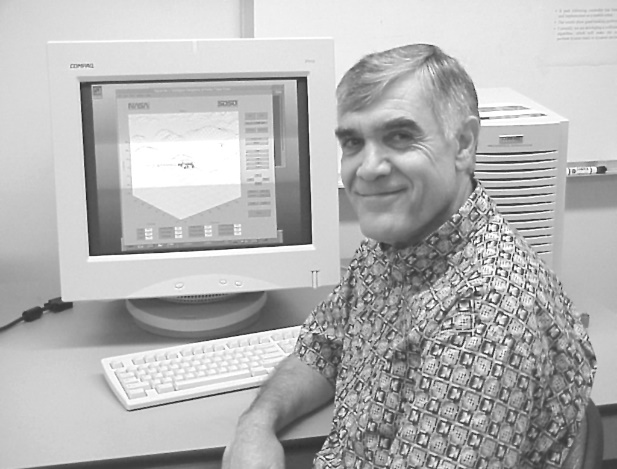
Uniform distribution can be used to generate other types of noise.

***h***

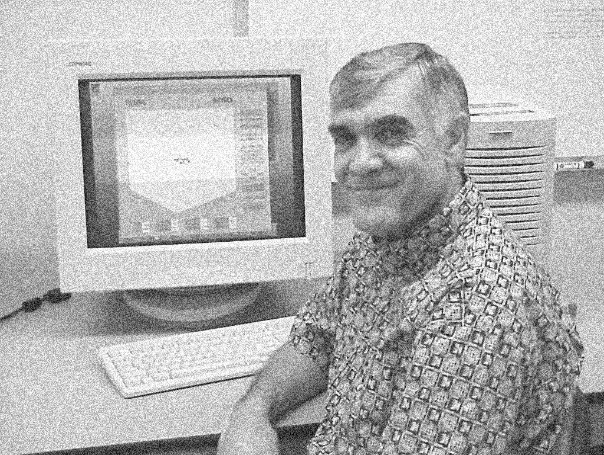
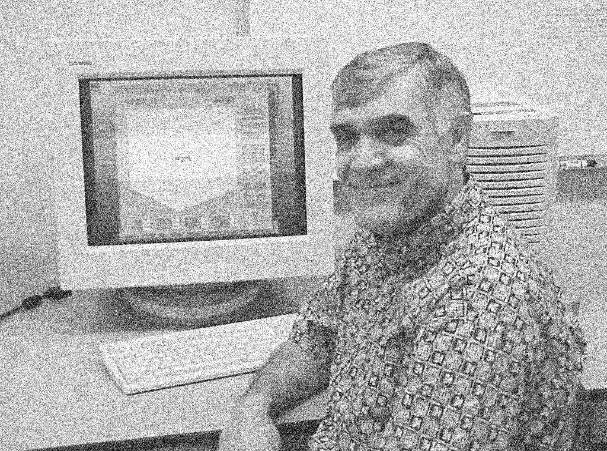
Fig. 13 Histogram of the

uniform distribution noise

0 a b ***l***



(a)

(b) (c)

Fig. 14 Original image (a), uniform noise added (b), and Gaussian noise added (c). Percentage of noise is the same for both cases.

**Salt and Pepper (impulse) noise**: This noise is produced by malfunctioning pixel elements in the camera sensors, faulty memory locations or timing errors in the digitization process.

 (4.25)

where  are some constants, typically equal to 0.1, and a or b are between 0 and 255 assuming an 8-bit image.

***h***

Fig. 15 Salt and pepper noise

0 a b ***l***

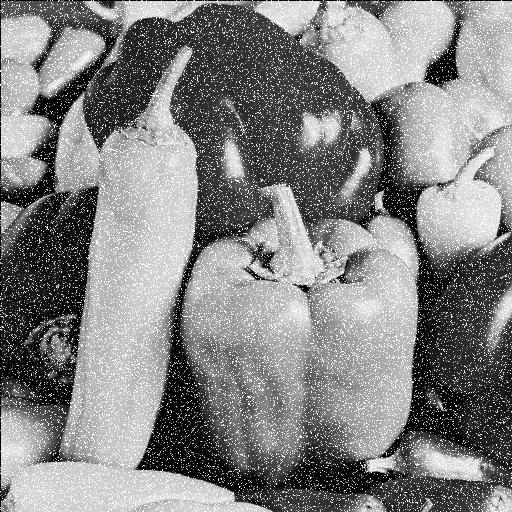


Fig. 16 Salt noise on peppers

**Sinusoidal Noise**: The intensity (gray level) of this noise can be described in a number of different forms such as

 or



where A is the amplitude, and u and v are some constants.



Fig. 17 sinusoidal noise

**Exponential Distribution**: Radar range images typically contain noise that can be modeled by Rayleigh, negative exponential, or gamma functions. The Rayleigh model is

 (4.26)

where ***a*** is a constant. This noise has a mean = and a variance of

.

**4.3.2**  **Averaging (Mean) Filter**

* This filter smooths the image and tends to blend or average out the noise with the image. The mask for this filter is typically of the form

*m=3* (4.27)

where ***m*** is the size of the mask, and all entries of the mask are 1.

* The convolution of this mask with the image results in an output image ***g(x,y)*** :

***g(x,y) ={ f(x –m/2, y –m/2) + …+ f(x –1, y) + f(x, y –1) + f(x,y) + f(x +1,y)***

+ ***f(x, y+1) + …+ f(x+m/2, y+m/2)}/***  (4.28)

Since the above operation is equivalent to finding the mean gray level values over all, the mask in (4.27 ) is also called a mean mask or mean filter.

* Note that we can evaluate (4.28) instead of convolving (4.27) with the image, and in fact this would require less computation for this special case of mean filter (why?). However, if hardware support is available for the convolution, then it is best to use the hardware (why?).
* Mean filter is also called “low pass” filter. This filter tends reduce the noise by averaging it out, but at the same time blurs the image. These two effects increase with the size of the mask. Fig. 4.15(a) shows the result of applying a 3 by 3 averaging mask, or low pass filter to the salt-type noise image 4.13. It is seen that the sharpness of the noise is reduced, but the image is blurred.

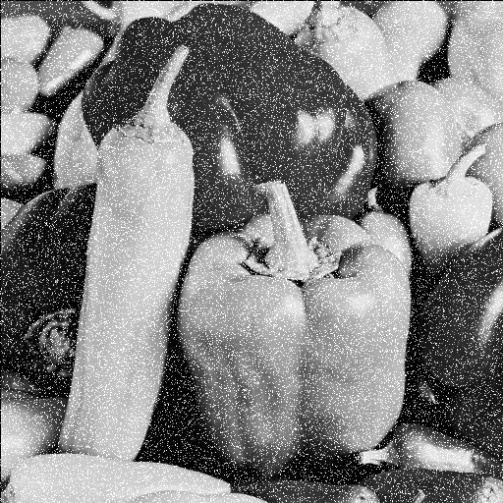
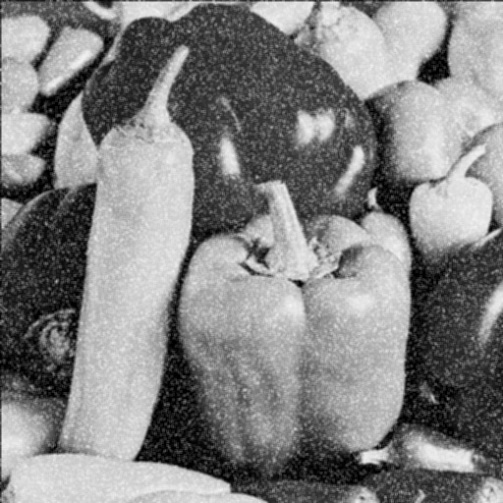
 

Fig. 18 Effect of mean filter on salt noise

1. (b)

Fig. 19 Effect of mean filter on sinusoidal noise; (a) original image, (b)image after filtering .

**Gaussian Filter**

Low pass (mean or smoothing) filtering can also be achieved by non-equal mask coefficients. The most common example of the latter is the Gaussian filter in which the mask coefficients are obtained from a discrete two-dimensional bell-shaped Gaussian function

  ***j, k = -m/2, …, m/2*** (4.29)

The Matlab function is implemented as *fspecial(‘gaussian’[s s], sigma)* where [s s] is the size of the Gaussian filter, usually 5x5, and sigma is the standard deviation. Below we generate a large filter to better see the shape:

>>s=50; sigma=3;

>> gauss=fspecial('gaussian', [s s],sigma);

>> surf(1:s,1:s,gauss)

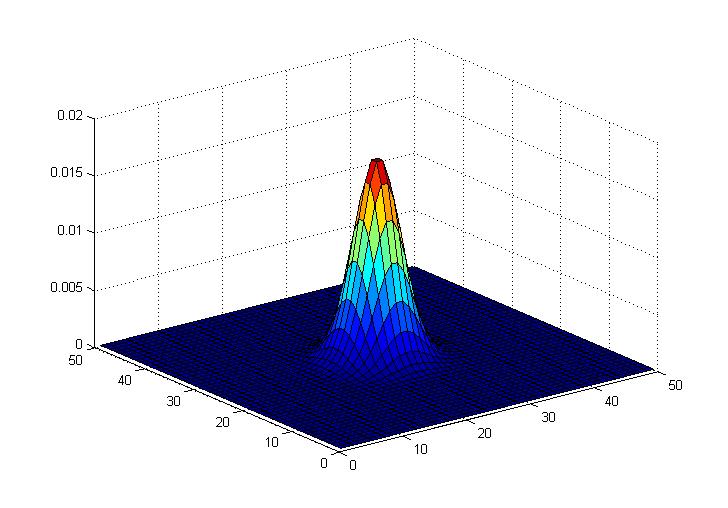


Fig. 20 A 5 by 5 mask with standard deviation , the Gaussian mask



* The mask coefficients decrease with increasing distance from the mask center, and therefore more weight is given to central pixels than to those in the periphery of the mask. As the dimension of the mask ***m*** increases, the standard deviation  must also increase.
* The Gaussian filter has several advantages. One is that it is rotationally symmetric so that there is no bias in the amount of smoothing. Another is that the Gaussian mask is separable, which allows fast computation as discussed before.
  + 1. **Unsharp and High Boost (High Pass) Filters**

Unsharp masking (edge crisping) is used to make edges in the image sharper which makes the image more pleasing to the eye. Unsharp making performs the following task:

Original image 🡺 blur with low pass filter 🡺scale with k<1 🡺 subtract from original image 🡺 scale for display.

We start with the filter . Applying this filter leaves the image unchanged. The unsharp filter is obtained from

where A is an averaging (low pass) filter, k>1 and s is a scaling factor to ensure the sum of all elements of U is 1. So and therefore . Suppose we choose k =1.5, then s=3

U=3

Note that this filter emphasizes on the center pixel. The Matlab code below applies the above unsharp filter to the bird image to get bird.unsharp and then again unsharp filter is applied to the acquired image. Enlarge the image to see the details.

*>> e=[0 0 0;0 1 0;0 0 0];*

*>> f=fspecial('average');*

*>> u=3\*e-2\*f;*

*>> bird.unsharp=imfilter(brd,u);*

*>> imshow(bird.unsharp)*

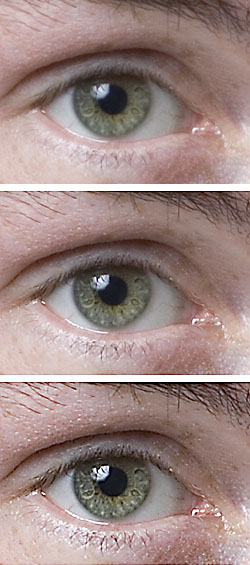
*>> bird.unsharp2=imfilter(bird.unsharp,u);*

*>> imshow(bird.unsharp2)*

(a) (b) (c)

Fig. 21 (a) Original image, (b)After one pass of unsharp filter, (c) (a) After second unsharp filter application.





**Example of unsharp filter**

**High Boost Filter (or sharpening)** is obtained by the following operation

(High boost) = a (original) – (low pass)

where a is the amplification factor.

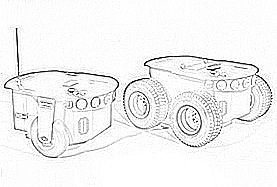
* High boost is achieved by emphasizing the difference in a pixel value relative to its neighbors. The sharpening mask has both positive and negative coefficients.
* Sharpening has a similarity with edge detection, but these two operations are somewhat different.
* The high boost (sharpening) mask has positive coefficient in the center and negative coefficients at the periphery. The well-known 3 by 3 sharpening mask is obtained using a=1.

(4.30)

where for the above mask, and in general so that. This implies that when the mask is applied to slowly varying gray level area, the result of the convolution is zero. However, when the gray level changes rapidly, the output image emphasizes these changes.

* Note that sharpening is equivalent to subtracting the image from a blurred version of that image (why?). In other words

 (4.31)

1. (b)

Fig. 22 The original image (a), sharpened by 3 by 3 mask and inverted image (b). Note that the slowly varying sections of the image have disappeared.

In order to keep the image details and but sharpening it, the following operation is performed

 (4.31-1)

where . This operation will only partially remove the slowly varying parts.

* The constant in (4.30) can be in which case more emphasis is placed on the center pixel relative to its neighbors. In the extreme case when ***c*** is very large, the convolution will have almost no effect on the image (why?). By adjusting ***c***, we can obtain a desired sharpening effect.
  1. **Ordered Filters**
* Nonlinear filtering methods have a number of useful characteristics that cannot be achieved by linear filtering methods of the last section. Some of these nonlinear filtering methods fall under the category of ordered or rank filtering.
* In order filters, we define a mask or window but do not convolve the mask with the image, rather we order the pixel values under the mask in increasing order. Suppose that the  pixel values under the mask, i.e. ***f(x, y), f(x-1, y)***, etc. are sorted in increasing order and denoted as

 (4.32)

For example if the nine pixels under the mask are 23 36 54

as shown on the right, then the ordered values 11 98 71

would be 11, 23, 29, 36, 54, 71, 88, 98, 120. 88 120 29

In Matlab the syntax for ordered filters is ordfilt2(image, k, ones(3,3)) where image is the image file name, k is the k-th largest elements.

* + 1. **Median Filter**
* In median filtering, the value of the output pixel ***g(x,y)*** is the median of the values under the mask with the center of the mask placed at ***(x,y)***. Note that the median is the middle value in the ordered set of values and is the  largest value in the set.

The median filter is particularly effective at removing salt and pepper (impulse) noise when the number of noisy pixels under the mask is less than  (why?). As an example suppose consider the following sub-image

23 36 54 24 48

11 **250** 240 55 26

44 245 253 50 47

Note the pixel with 250 is replaced by 44. In Matlab either

>>*PicMed=ordfilt2(Pic,5,ones(3,3));*

where 5th largest is the median. Alternatively, median has a special function and any size window can be used

>>*PicMed=medfilt2(Pic, [7,7]);*

where [7,7] is the size of the of median mask.

(a) (b) (c)

Fig. 4.18 Image corrupted by impulse (salt) noise (a), image cleaned by median filtering (b), and image cleaned by mean filter. Note the median filter has almost perfectly cleaned the image without blurring the image.

* The sorting of the pixel values under the mask is a significant computation burden. Quicksort is known for its efficiency, and its complexity is  for sorting values. However, this complexity is for one pixel, and when applied to the whole image the complexity of median filtering will be where  is the total number of pixels in an image. In addition, Quicksort is not efficient for small size files (i.e. 33 or 55 masks) due to the overhead.
* Other algorithms exist that find the median with ***O(m)*** complexity (see e.g. Computer Algorithm by Baase and Van Gelder, pp. 234).
  + 1. **Minimum, Maximum and Mid-Point Filters**

In ordered sequence

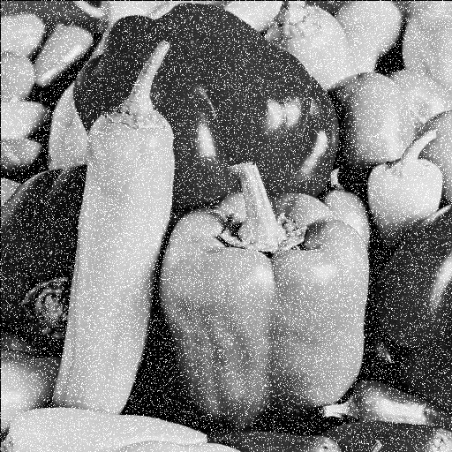


* Minimum filter selects  and is best suited to salt-type noise (why?).
* Maximum filter selects  and is suitable for pepper type noise (why?).
* Instead of the minimum and maximum filter, we can also select specific pixel rank, e.g. second largest or third smallest value, if needed. The mid-point filter selects  value, and is useful for the Guassian and uniform noise.
* Finding the minimum or maximum does not need sorting of the pixels values, rather this can be determined by a sequential search requiring only  comparisons.

The Matlab code below uses the following structure for ordered filters

>> *Pic\_k=ordfilt2(Pic,k,ones(3,3))*

where k=1 is the minimum filter, k=9 is the maximum filter and k=5 is the median filter.

Noisy image Min Filtered



Max Filtered

Fig. 23 Effect of mean, and max filter on a noisy image

**Modified Mean Filters**

* The mean filter discussed in Section 4.3.2 is a linear filter and finds the average of pixel values under the mask, i.e.

 (4.33)

Various modifications to this filter are possible and useful for reducing different types of noise. These modifications result in nonlinear filters.

**Alpha-Trimmed Mean Filter**: This is the same as the mean filter but with end-point ranked values (e.g. low and high values under the mask) excluded. The output is

 (4.34)

where  is the number of pixels excluded at each end of the ordered set, and ranges as . If , the average is found, and when , the median is evaluated. This filter is useful when image contains multiple type of noise such as both Gaussian and pepper/salt noise.

**Harmonic Mean Filte**r: The output produced by this filter is

 (4.35)

This filter is used for salt noise and Gaussian noise, and retains useful information better than the averaging (arithmetic mean) filter. However, it must not be used for pepper noise (why?).

**Contra-Harmonic Mean Filter**: This filter produces the following output

 (4.36)

where***r***is an integer and is called the filter order. When ***r*** is positive the above filter eliminates the pepper noise, and when it is negative it eliminates the salt noise (why?).

**Geometric Mean Filter**: The filter output is

 (4.37)

This filter works best with the Gaussian noise, and retains noise free pixels better than a regular (arithmetic) mean filter. It must not be used with pepper noise (why?).

* 1. **Adaptive Filters**
* Each of the spatial domain filters discussed so far performs the same calculation at almost every pixel in an image. In some cases, the properties of an image can vary spatially, and a filter that is suitable for one part of the image may not be useful for another part. For example, the image may be corrupted by Gaussian noise, and a mean filter may improve the affected parts, but it may have the adverse blurring effect on the uncontaminated edges. These problems can be reduced by an adaptive filter whose behavior changes in response to the local image properties.
* Adaptive filters base their behavior on the local gray level statistics within a neighborhood of a pixel. A classic example is the minimal mean square error filter that computes

 (4.38)

where  is an estimate of noise variance,  is the gray level variance computed for the neighborhood centered at (x,y), and 

is the mean gray level in that neighborhood.

* In the homogeneous areas, noise will be dominant, and thus , and . In the proximity of an edge, dominates local noise variance, i.e. , and the output pixel value will be the same as the input pixel value (why?). Thus the behavior of the filter changes according to the local characteristics of the image.