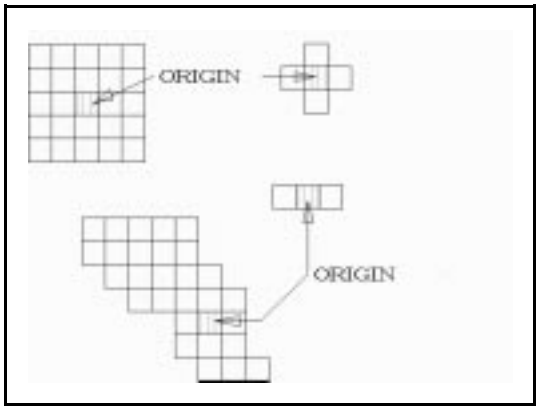
Unit 3. Area Processes

3.1 Pixel Neighborhoods for Area Processes

Pixel Neighborhoods. Let (m_0,n_0) be a pixel position in an image $\{f(m,n)\}$. A *neighborhood* of (m_0,n_0), designated as *nbhd* herafter, is a set of *connected* (continguous) pixels that contains (m_0,n_0), which is called the *origin* of the neighborhood. The most common neighborhoods are rectangles of pxq pixels that contain the pixel (m_0,n_0) at the center, where p and q are odd numbers so that an exact center exists. Another type of neighborhood is the 3x3 *star*, which can be considered to be a rectangular neighborhood with the corner points removed. Figure 3.1 shows some neighborhoods, but the definition above is rather general and includes a great variety of connected blobs of pixels with one pixel in the aggregate being distinguished as the origin, usually called the *center* pixel. An *area process* is a transformation that maps a nbhd of each center pixel f_{cntr} into a new value g_{new} for the pixel.

Figure 3.1 Neighborhoods of pixels



Neighborhoods play a central role in image processing. For example, we can process an image $\{f(m,n)\}$ by taking a 3x3 neighborhood of each pixel, in turn, with value f_{cntr} and compute the new pixel value g_{new} to be the average of all the pixel values in that neighborhood. The set of all such new pixels form a new processed image $\{g(m,n)\}$ that would have less variation and smeared edges. As another example, we could order the pixel values in each neighborhood of f_{cntr} by magnitude and then pick the middle value as the new value g_{new} for the origin pixel. This would clear the image of small dots of differing shades of gray (outliers) from the typical ones in the neighborhood. In general, for each original pixel we consider a neighborhood of it and map the pixel values in that neighborhood into a new grayscale value.

Averaging on a Pixel Neighborhood. Let $p_5 = f_{cntr}$ be a pixel in the center of its 3x3 neighborhood. Let us average the 9 pixels in the neighborhood and replace p_5 by this average, and then do this for each pixel in the image. This *smooths* the image in the sense that all pixel values that are very different from their neighborhood averages will be replaced by the average and so the differences between the pixels will be reduced. Figure 3.2 shows a 3x3 nbhd with the pixels numbered in the way that we will use them henceforth. The process of averaging replaces the pixel p_5 by the new P_5 that is computed as

$$P_5 = [p_1 + p_2 + ... + p_9] / 9 = (1/9)\sum_{k=1,9} p_k = \sum_{k=1,9} (1/9)p_k \qquad (3.1)$$

Figure 3.2. A 3x3 nbhd of a pixel.

p1	p2	p3
p4	p5	p6
p7	p8	p9

The Weighted Average on a Pixel Nbhd. In the averaging of Equation (3.1) the pixels were all weighted the same by the value $w_k = 1/9$ and all 9 of these weights added up to unity. Here the weights 1/9 are all equal and we say that the average has *equal weighting*. A weighted averaging of the pixels can use unequal weights to put more emphasis on certain pixel values and less on others. Consider the following weighting.

$$P_5 = \sum_{k=1,9} w_k p_k \qquad (3.2)$$

where $w_k = 1/10$ for $k \neq 5$ and $w_5 = 2/10 = 1/5$. This would count the middle pixel twice as much as any other pixel and so it would have more influence in the weighting. All weights sum to unity as required.

It is convenient to put the weights in an array called a *mask*, of the same size as the nbhd, and to define an operation for the weighted average. Consider the operation of a mask and nbhd shown below.

$$p_{new} = \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix} * \begin{bmatrix} p_1 & p_2 & p_3 \\ p_4 & p_5 & p_6 \\ p_7 & p_8 & p_9 \end{bmatrix} = \sum_{k=1,9} w_k\, p_k \qquad (3.3)$$

We call this operation *convolution*. It leads to the terminology of *convolution mask* for the mask and *mask convolution* for the operation. This operation is designated by the symbol *. In general, the mask and neighborhood size are fixed at some pxq for any particular convolution.

Mask Convolution on Images. To apply a weighted averaging over an entire image, the process starts at the upper lefthand corner of the image and processes one pixel at a time via the mask convolution on the pixel's neighborhood. After the top left (first) pixel is processed to obtain a new output pixel value p_{new}, the process then moves the nbhd to the right by one pixel and convolves the mask with that pixel's neighborhood to obtain an output pixel p_{new}. When the last pixel in a row of pixels has been processed, the mask is moved back to the left side of the image down one pixel. Then the second row of image pixels is processed similarly. This is repeated until all rows of pixels in the image {p(m,n)} have been processed.

The pixels on the outer boundaries of the image need not be processed (the neighborhoods of these pixel positions do not fit inside the image). Another method is to process all pixels, but on the boundary pixels to use the convolution with only the mask and nbhd entries that intersect the image. Another technique is to repeat the boundary pixels outside the boundaries sufficiently many times to complete the pxq neighborhood of the mask size.

The most basic convolution mask is the *identity* operator, which yields the pixel value $p_{new} = f_{cntr}$ on each neighborhood. The pxq identity mask contains all entries of zero except the center, which is unity. For example, the 3x3 identity mask operates on a 3x3 neighborhood via

$$p_{new} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} p_1 & p_2 & p_3 \\ p_4 & p_5 & p_6 \\ p_7 & p_8 & p_9 \end{bmatrix} = \{\sum_{(j\neq 5)} (0)p_j\} + (1)p_5 = p_5 \qquad (3.4)$$

The central pixel is often weighted more than the others, and the others may be weighted unequally for different degrees of smoothing. An example of weighted averaging convolution mask and its *mask factor* (1/16) that forces the weights to sum to unity is

$$(1/16) \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} * \begin{bmatrix} p_1 & p_2 & p_3 \\ p_4 & p_5 & p_6 \\ p_7 & p_8 & p_9 \end{bmatrix} = \{p_1+p_3+p_7+p_9 + 2[p_2+p_4+p_6+p_8] + 4p5\}/16 \qquad (3.5)$$

Averaging masks that smooth lightly are

$$(1/5) \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad (1/6) \begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad (1/10) \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 3.3. Original *Lena*.

Figure 3.4. Smoothed *Lena*.

Figure 3.5. Blurred *Lena*.

Blurring Masks. It is a sometimes useful effect to *blur* an image by oversmoothing it by averaging the neighborhood pixel values not adjacent to the center pixel. Figure 3.3 shows the original *lena256.tif* while Figure 3.4 shows the smoothed result obtained with the lefthand 3x3 mask of the three masks given above. Figure 3.5 displays the image blurred with the mask given below as Equation (3.6a).

$$M = (1/16)\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad M = (1/36)\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{3.6a.b}$$

The sum of all mask entries multiplied by the mask factor (1/36) in Equation (3.6b) should equal unity to keep the output average brightness the same as in the original image. If the sum of the weights is greater than 1 (respectively, less than 1) then the resultant image will be brighter (respectively, darker) than the original image. Blurring can also be done with larger smoothing masks.

3.2 Smoothing with Medians and Trimmed Means

Medians and Alpha-trimmed Means. We have already seen convolution masks that average, which is one form of smoothing. This is useful for eliminating low level noise or giving skin a smooth look in a photographic portrait. However, averaging is strongly affected by outlier pixel values, which is the "snake swallowing the egg" (SSTE) syndrome. An outlier causes the average to be in error, and as the mask moves to new centers nearby, the outlier remains in the new neighborhoods so that the outlier strongly influences (erroneously) a block section of the image larger than the mask. A pxq mask maps an outlier into a (2p-1)x(2q-1) block of erroneous values.

A well known method for avoiding the SSTE syndrome is to use the median (order statistics). For each pixel and neighborhood centered on it, we order the neighborhood pixel values p_k from smallest to largest. The indexing by order is designated by

$$f_{j(0)} \rfloor\ f_{j(1)} \rfloor\ \ldots \rfloor\ f_{j(pq-1)} \tag{3.7}$$

There are two ways to proceed here. The first is to take the middle pixel value, $f_{j(c)}$, where j(c) is the index of the pixel with central value, and select the new output value as

$$g_{new} = f_{j(c)} \tag{3.8}$$

This is the middle of all of the actual nbhd gray level values (the median). This is often a good way to remove *speckle* (also called *salt and pepper*) noise, which appears as pixel size dots of dark or bright.

The median of a neighborhood is not always the value that best portrays the image information. For this reason, better results are often obtained by using the *α-trimmed mean*. To obtain this value, we set α to be 1 or 2 or so, throw out the lowest α and highest α pixel values and average the remaining pq - 2α pixel values to obtain g_{new}.

Median processing does not blur the image. It preserves edges rather than smearing them as does averaging. The α-trimmed median also has less blurring and spreading of edges. Thus, these are powerful processes for smoothing noise from the image while preserving its features.

Smoothing Images. Smoothing is done after an examination of the image reveals: i) salt and pepper noise (*speckle*); ii) short line noise such as small scratches; iii) any type of nonnegligible variations on areas that should be smooth; iv) rough, blocky or zig-zaggy edges; or v) other types of noise that smoothing can ameliorate.

Smoothing is usually done before edge detection or sharpening to attenuate the amplification of variation done by those processes. Mild smoothing can often enhance an image. Median smoothing, also called *despeckling*, changes the image the least (except for removing speckle) and will not damage the image. Tools for processing images have smoothing capability, as do *Matlab*, *XView* and *LView Pro*.

Using the Tools for Smoothing. Here we actually show how to do mask convolution with the tools, but we use the smoothing process as the example. Smoothing is the most fundamental type of image processing, but we must be careful not to save a smoothed image over the original because it loses some of the information in the image that can not be recovered by sharpening or any kind of processing of the smoothed image. Sharpening, edge detection and other types of mask convolution can be done in the same way as smoothing by designing an appropriate mask and then applying it to an image.

1. Smoothing with *Matlab*: To apply a convolution mask with *Matlab* it sufficies to use the special function *imfilter()*, which lets the user designate the image and specify the mask values. After running *Matlab* and clicking in the Command Window to bring it into focus, the following commands smooth the image with a mildly smoothing mask (also called a *filter*) that uses equal weighting (see Fig. 3.6 for the original and Fig. 3.7 for the results of this smoothing).

```
> Im1 = imread('lena256.tif');          //read image into memory at Im1
> imshow(Im1);                          //show image at Im1 on the screen
> h1 = [1 1 1; 1 1 1; 1 1 1] / 9;       //define convolution mask h1
> Im2 = imfilter(Im1, h1);               //filter image with mask h1
> figure, imshow(Im2);                  //show filtered image as new figure
```

Figure 3.6. The original *lena256.tif*.

Figure 3.7. The *Matlab* smoothed *Lena*.

2. Smoothing with *XView*: *XView* does not have a mechanism for entering in the entries in a convolution mask, but it does have a smoothing algorithm where the nbhd size is selected. Upon applying the smoothing process consecutively, the degree of smoothing can be somewhat controlled. First run *XView* and load an image, then right click inside the image to bring up the control window. Select *Algorithms* on the top right and come down to *Blur* and then enter in the nbhd size (3 is the default, 5 increases the smoothing, etc.). Click *OK* to smooth. Larger nbhds or repeated smoothing will blur the image.

3. Smoothing with *LView Pro*: Run *LView Pro*. When the main window comes up select *File* and then *Open*. Next, select the image file and click the *Open* button to load and display the image. Select *Color* on the top menu bar and then come down in the pop up menu to *User Defined* (under the *Histogram* item), and then select the option *Filters*. The *User-defined Image Filters* window comes up next (see Fig. 3.9a). Click *New* to bring up the *Filter Specification* window. A large mask of text entries comes up (but we can use only the central 3x3 or 5x5 for our purposes). Enter the desired smoothing mask and then enter the appropriate integer in the *Divisor* text entry slot at the bottom (this is the denominator of the scaling factor to assure that the mask entries sum to unity). Click OK to return to the *User-defined Image Filters* window and then click the *Apply* button to run the mask convolution on the image. Click the *Close* button to exit this window and return the focus to the main *LView Pro* window.

Figure 3.8a. The *LView Pro User-defined Image Filters* window.

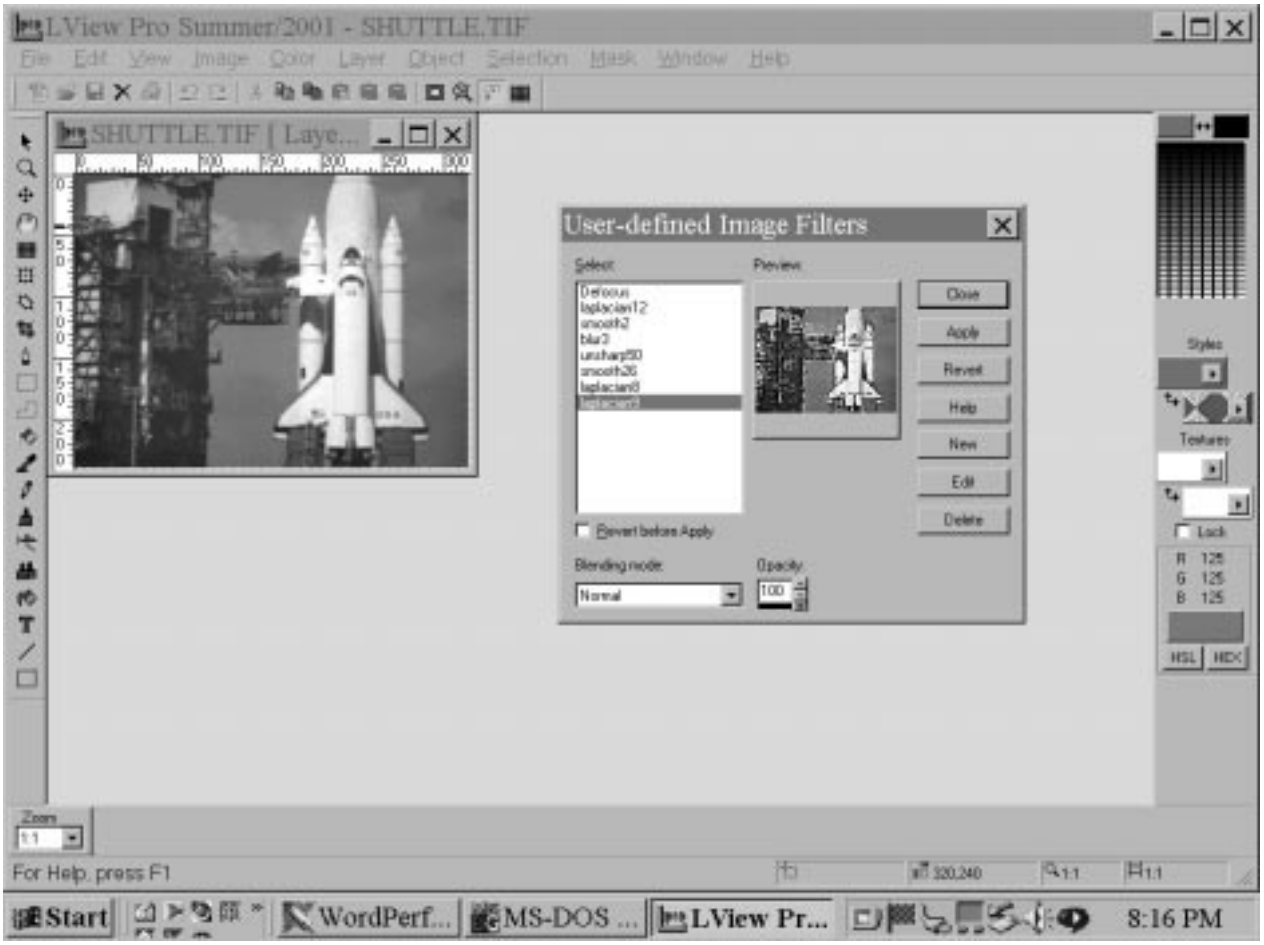
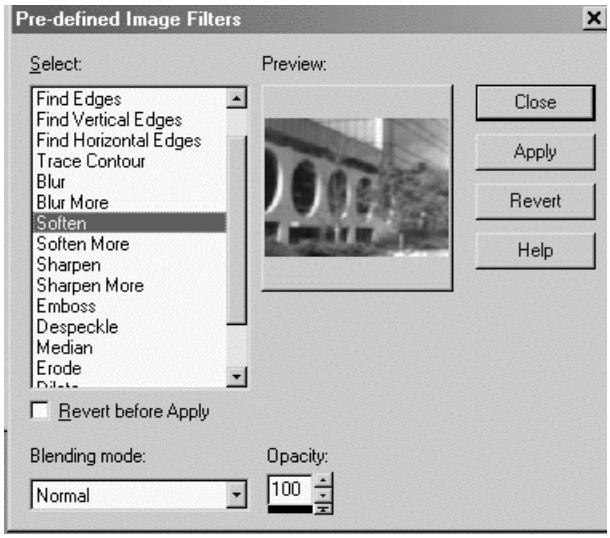


Figure 3.8b. The *Pre-defined Image Filters* window.



Another method is to click on *Color* and then on *Filters* to bring up the *Pre-defined Image Filters* window. On the left of this window there is an option box as shown in Figure 3.8b. Select the *Soften* or *Soften More* options and then click on *Apply* on the right. To undo the filtering, click on *Revert* on the right.

3.3 Sharpening Images

Unsharp Masking. A process known as *unsharp masking* (taken from the photographic processing industry) first smooths an image and then the subtracts the smoothed image from the original image. The smoothing yields an image whose variation (rates of change) in gray levels across the spatial dimensions is lowered. Smoothing actually spreads out the changes

from pixel to pixel at all rates into more gradual changes in grayscale across the dimensions m and n (x and y). When the smoothed image is subtracted, it leaves the more rapid variations (changes at a higher rate). The resultant image appears *sharp*, that is, the edges are narrower and have greater contrast on each side. The average gray level is reduced by subtraction, so we need to multiply the original image by a factor ß > 1.0 to increase its intensity, or brightness power. We first increase the brightness power by multiplying the identity convolution mask by the *gain* ß > 1.0, which is called *boosting* the original. Boosting is shown in Equation 3.9.

$$(ß)\quad \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & ß & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.9}$$

Then we subtract an averaging mask that smooths/blurs the image. Using a 5x5 mask example, the resulting unsharp masking convolution mask is obtained via

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & ß & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} - (1/26) \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = (1/26) \begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & 26ß-2 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix} \tag{3.10}$$

Fig. 3.9. Orginal shuttle. **Fig. 3.10. Smoothed shuttle.** **Fig. 3.11. Unsharp masked shuttle.**



A gain ß of approximately 2.0 is often satisfactory, but the best value depends upon the average brightness power. We used $\alpha = 2$ in Equation 3.10. Figure 3.9 shows the original *shuttle* image, while Figure 3.10 shows the smoothed version that was subtracted from the boosted original (ß = 2). Figure 3.11 presents the unsharp-masked result. While many sharpening methods have been developed, this one remains one of the very best.

Directional Derivatives. We have seen that unsharp masking by subtracting a smoothed image is an effective way to sharpen an image. The underlying process of sharpening must make edges more exaggerated, that is, thinner with a greater difference between the darker and lighter pixels along the edges. To detect the edges, it is necessary to obtain the differences in each neighborhood along the horizontal (y), vertical (x) and two diagonal (y = -x and y = x) directions. Let a neighborhood of (m,n) have the values

$$\begin{bmatrix} f(m-1,n-1) & f(m-1,n) & f(m-1,n+1) \\ f(m,n-1) & f(m,n) & f(m,n+1) \\ f(m+1,n-1) & f(m+1,n) & f(m+1,n+1) \end{bmatrix}$$

Figure 3.12 displays a sharp edge, while Figure 3.13 presents a smooth edge. The first can be considered a sharpening of the second and likewise the second can be considered to be a smoothing of the first. Edges in real world images are usually wider than the one shown in Figure 3.12.

Figure 3.12. A sharp edge.

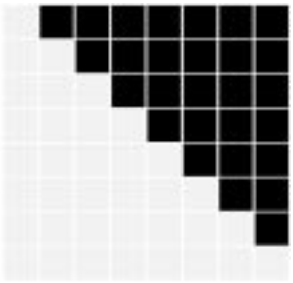
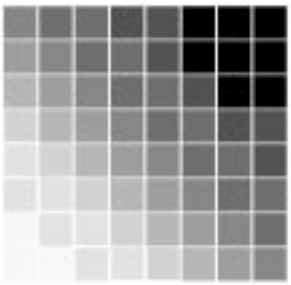


Figure 3.13. A smooth edge.



On this fixed 3x3 (or any fixed pxq) neighborhood centered on any pixel position (m,n), the differences in the various directions are

$$\partial f(x,y)/\partial y = [f(m,n+1) - f(m,n-1)]/2 \tag{3.11a}$$

$$\partial f(x,y)/\partial x = [f(m+1,n) - f(m-1,n)]/2 \tag{3.11b}$$

$$\partial f(x,y)/\partial u = [f(m+1,n+1) - f(m-1,n-1)]/2 \tag{3.11c}$$

$$\partial f(x,y)/\partial v = [f(m-1,n+1) - f(m+1,n-1)]/2 \tag{3.11d}$$

where u is in the direction along the line y = x and v is the direction along the line y = -x.

Masks that produce these differences under convolution (ignoring the 1/2 factor) are

$$\begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

More effective differencing masks average more than a single difference of pixels to mitigate errors due to a noisy pixel. The *Prewitt* operators are

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{bmatrix}$$

More effective yet are the *Sobel* operators that weight the key differencing pixels more. They are

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}$$

After convolution with all four difference operators, the edges will be light lines on a dark background in four different images. These can be thresholded to bi-level images of black and white and then combined via use of the maximum at each pixel.

Laplacian Masks. The *Laplacian* sums the squares of the second derivatives via

$$\partial^2 f(x,y) = \partial^2 f(x,y)/\partial x^2 + \partial^2 f(x,y)/\partial y^2 \tag{3.12}$$

Recalling that the second derivative can be approximated by the difference of two derivatives at two adjacent points, the second derivatives become

$$\partial^2 f(x,y)/\partial x^2 = \{[f(m+1,n) - f(m,n)]/\alpha - [f(m,n) - f(m-1,n)]/\alpha\} / \alpha =$$

$$\{[f(m+1,n) - 2f(m,n)] + f(m-1,n)\} / \alpha^2 \qquad (3.13)$$

$$\partial^2 f(x,y)/\partial y^2 = \{[f(m,n+1) - f(m,n)]/\alpha - [f(m,n) - f(m,n-1)]/\alpha\} / \alpha =$$

$$\{[f(m,n+1) - 2f(m,n)] + f(m,n-1)\} / \alpha^2 \qquad (3.14)$$

The 3x3 Laplacian mask is therefore the sum of the two second differencing masks, which is

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad (3.15)$$

when α is ignored ($\alpha = 1 = \alpha^2$). Strangely, the literature calls the Laplacian the negative Laplacian.

$$(-1)\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}, \qquad (-1)\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \qquad (3.16a,b)$$

Note, for example, that both the negative and positive Laplacian processes produce an output g_{new} near zero (very dark) when all pixels on the matching "plus-shaped" neighborhood are approximately the same (the mask entries sum to zero). On the other hand, the negative Laplacian produces a positive output only when the central pixel value is greater than the average of the others. If the central pixel value is less than the average of the others, then the negative result is truncated to zero (black). Equation (3.16a) uses only the vertical and horizontal differences while Equation (3.16b) uses differences in all four directions.

Hedged Laplacians. Figure 3.14 uses the negative 3x3 Laplacian given in Equation (3.17b) on the original "Shuttle", while Figure 3.15 shows the effect of using the *hedged* Laplacian mask of Equation (3.17).

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \qquad (3.17)$$

Figure 3.14. Laplacian on *shuttle.tif*.

Figure 3.15. Hedged Laplacian on *shuttle.tif*.

Hedged Laplacians can be considered to add the negative Laplacian processed image back onto the original image. This is actually the subtraction of the Laplacian from the original image, which is a powerful technique as shown in Figure 3.15. Note that the sum of the weights in the hedged Laplacian is equal to unity, so the average gray level will remain the same as in the original image. While the image in Figure 3.15 is sharp, the small differences due to random effects (noise) are exaggerated (a disadvantage of Laplacian and other differencing methods). A milder hedged Laplacian mask is

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad (3.18a)$$

Other interesting hedged Laplacians are

$$\begin{bmatrix} -1 & -2 & -1 \\ -2 & 13 & -2 \\ -1 & -2 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} -1 & -2 & -1 \\ -2 & 12 & -2 \\ -1 & -2 & -1 \end{bmatrix} \quad (3.18b)$$

$$\begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -2 & -1 & -1 \\ -1 & -2 & 29 & -2 & -1 \\ -1 & -1 & -2 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix} = ? \quad (3.18c)$$

Edge Enhancement. A cross-section of an image shows edges as rises in Figure 3.16 (here the dark side is on the left and the light side is on the right). Figure 3.17 shows the first and second derivatives of the moderate edge and the effect of adding the original to the (negative) Laplacian, which is actually subtracting the Laplacian from the original (see the bottom graph). This is analogous to unsharp masking, where the smoothed edges (see the center graph of Figure 3.16) are subtracted from the original edges to sharpen them, which yields a result similar to the bottom graph of Figure 3.17.

Edges are enhanced by unsharp masking and hedged Laplacians, but can also be done by thresholded Laplacians and rule-based processing. Laplacians perform edge detection, which is not the same as edge enhancement.

Figure 3.16. Edge cross-sections.

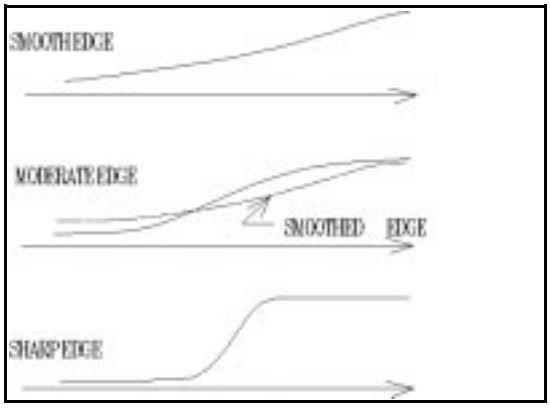
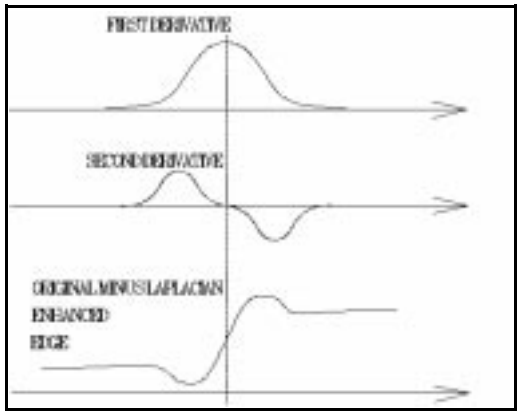


Figure 3.17. Subtracting the Laplacian.



Thresholded Laplacians. A variation on the Laplacian processing is to check the value of each Laplacian against a fixed threshold T. If the Laplacian is too large (larger than the threshold), then it is likely to be erroneous due to noise and should be suppressed. In this case the neighborhood average is taken to be the new pixel value g_{new}. On the other hand, if the Laplacian is low (lower than the threshold), then the new pixel value g_{new} is enhanced by adding a proportion α of the Laplacian to it. The algorithm is

$$\text{if } \lfloor^2 f(m,n) > T \text{ then} \qquad g_c = (1/r)\Sigma_{(k=1,r)}\, f_k \quad (3.19a)$$

$$\text{else} \qquad g_c = f(m,n) + ß \lfloor^2 f(m,n) \quad (3.19b)$$

The disadvantages of Laplacian operators are that they give no direction for the edge, they produce double edges and they are extra sensitive to noise (they amplify noise). Derivatives and differences are

sensitive to noise and second differences are even more sensitive in that small errors in the input data can cause large errors in the output data. Noise should be reduced before edge detection is performed.

The gradient can be used in situations where it is important to know the direction of the edge or line segment. The gradient and direction α can be found per

$$(\partial f(x,y)/\partial x, \partial f(x,y)/\partial y) = (f(m+1,n) - f(m-1,n),\ f(m,n+1) - f(m,n-1)) \tag{3.20}$$

$$\alpha = \text{atan}(\partial f(x,y)/\partial y \ / \ \partial f(x,y)/\partial x) \tag{3.21}$$

Rule-based Simultaneous Sharpening and Smoothing. On a 3x3 nbhd of pixel p_5, we first compute the 9 differences

$$d(1) = p_1 - p_5, \ldots, \quad d(9) = p_9 - p_5 \tag{3.22}$$

To obtain a representative value for these 9 differences, we take their *1-trimmed mean* by throwing out a single maximum and a single minimum values and averaging those remaining. The resulting α is resistant to outliers. We use two thresholds T_1 and T_2 where the latter one is the greatest. The rules are

$$\text{if } (\alpha < T_1) \qquad \text{then } p_{new} = p_5 + \alpha \ ; \tag{3.23a}$$

$$\text{else if } (\alpha > T_2) \qquad \text{then } p_{new} = p_5 + \alpha \ ; \tag{3.23b}$$

Equation (3.23a) smooths while Equation (3.23b) exaggerates the difference between a center pixel and its neighbors. Otherwise there is no change. The result is a smoothing of relatively smooth pixels and a sharpening of those that differ sufficiently from the nbhd trimmed mean.

Sharpening with the Tools. We have already seen how to use *Matlab*, *Xview* and *Lview Pro* to perform smoothing with mask convolution. We do the same type of mask operations (filtering) but use different masks as described above for unsharp masking, laplacian and hedged laplacian processes. We present examples below.

1. *Matlab* Sharpening: To apply a sharpening convolution mask with *Matlab* it sufficies to use the special function *imfilter()* again as we did for smoothing. After running *Matlab* and clicking in the Command Window to bring it into focus, the following commands smooth the image with a mildly smoothing mask (also called a *filter*).

```
> Im1 = imread('lena256.tif');        //read image into memory at Im1
> imshow(Im1);                        //show image at Im1 on the screen
> h1 = [-1  -1 -1
        -1 12 -1
        -1 -1 -1] / 4;                //define sharpening convolution mask h1
> Im2 = imfilter(Im1, h1);            //filter image with mask, store at Im2
> figure, imshow(Im2);                //show filtered image at Im2 as new figure
```

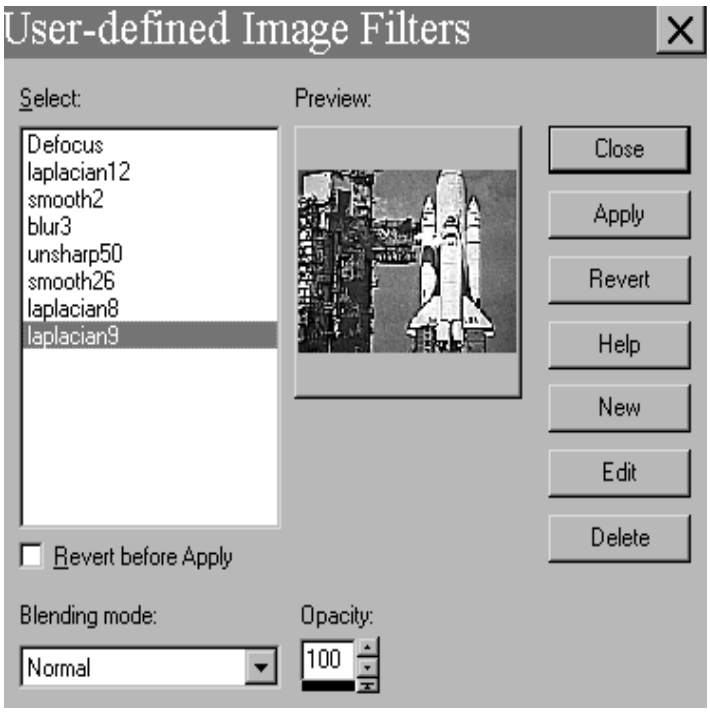
Figure 3.18. The original *lena256.tif*.

Figure 3.19. The Matlab sharpened *lena256*.

2. *XView* Sharpening: To sharpen with this tool we run *XView* and load an image. Then we right click inside the image to get the control window. We click on the *Algorithms* button at the top right and come down to *Sharpen*. Then enter a percentage for sharpening (75% is the default and is a good value). The displayed image is then sharpened on the screen. The process can be repeated for extra sharpening. This tool does not let the user enter a convolution mask but uses its own standard masks. Therefore this is a weak tool for mask convolution. To overcome this deficiency, we include a C program in Appendix 3.B that allows the user to enter a mask of the desired size, process an image and then call *XView* to display the *before* and *after* images.

3. *LView Pro* Sharpening: Run the program, select *File*, then *Open* and select a file and click the *Open* button to load and display the image. Select *Color* on the top menu bar and then come down in the pop up menu to *User Defined* (under the *Histogram* item), and then select the option *Filters*. The *User-defined Image Filters* window as shown in Figure 3.20 comes up next. Click *New* to bring up the *Filter Specification* window. A large mask of text entries comes up (but we can use only the central 3x3 or 5x5 for our purposes). Enter the hedged Laplacian mask from the left side of Equation (3.17) and enter *1* in the *Divisor* text entry slot at the bottom. Click OK to return to the *User-defined Image Filters* window and then click the *Apply* button to run the mask convolution on the image. Click the *Close* button to exit this window and return the focus to the main *LView Pro* window.

Figure 3.20. The User-defined Image Filter window.



3.4 Detecting Edges and Lines

Line Detection. Lines are extended edges. A line drawing can be developed from an image {f(m,n)} by convolving it with an edge detector such as the Laplacian or the four Sobel operators. These yield light lines on a black background, so we invert by putting g(m,n) = 255 - f(m,n) to achieve black lines on a white background. We can also use a threshold to transform the image to black and white only.

A problem is that the lines are too thick (edge detection thickens the lines). Other problems include broken lines and noise. It is useful to remove noise before processing to detect lines, but this should not be done by averaging because this spreads and weakens the edges. The thinning of lines after line detection is often necessary, as is the removal of spurs sticking out from the lines at various angles. The processes of thinning lines and trimming spurs from them is covered in a later unit.

Line detection is a type of edge detection. Consider a horizontal line segment in an image. If the pxq neighborhood straddles a horizontal line, for example, that line could be detected if the pixels along the center row have a higher average than those of the other rows. Thus

$$\begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix} \qquad \begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ 2 & 2 & 2 & 2 & 2 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

are horizontal line detectors. The larger mask is more immune to noise. Similarly detectors of vertical, -45° and 45° lines are

$$\begin{bmatrix}-1 & 2 & -1\\ -1 & 2 & -1\\ -1 & 2 & -1\end{bmatrix} \quad \begin{bmatrix}-1 & 2 & -1\\ -1 & 2 & -1\\ -1 & 2 & -1\\ -1 & 2 & -1\end{bmatrix} \quad \begin{bmatrix}-1 & -1 & 2\\ -1 & 2 & -1\\ 2 & -1 & -1\end{bmatrix} \quad \begin{bmatrix}-1 & -1 & -1 & -1 & 4\\ -1 & -1 & -1 & 4 & -1\\ -1 & -1 & 4 & -1 & -1\\ -1 & 4 & -1 & -1 & -1\\ 4 & -1 & -1 & -1 & -1\end{bmatrix} \quad \begin{bmatrix}2 & -1 & -1\\ -1 & 2 & -1\\ -1 & -1 & 2\end{bmatrix} \quad \begin{bmatrix}4 & -1 & -1 & -1 & -1\\ -1 & 4 & -1 & -1 & -1\\ -1 & -1 & 4 & -1 & -1\\ -1 & -1 & -1 & 4 & -1\\ -1 & -1 & -1 & -1 & 4\end{bmatrix}$$

We can now answer an important question: *is it possible to detect lines in all directions with a single mask?* Upon adding the 3x3 horizontal and vertical line detector masks first to obtain a horizontal-vertical line detector, and then adding the 3x3 diagonal line detector masks second to obtain a diagonal line detector, we obtain the respective two sums

$$\begin{bmatrix}-2 & 1 & -2\\ 1 & 4 & 1\\ -2 & 1 & -2\end{bmatrix} \qquad \begin{bmatrix}1 & -2 & 1\\ -2 & 4 & -2\\ 1 & -2 & 1\end{bmatrix}$$

The sum of these two masks yields a mask that detects edges in the horizontal, vertical, -45° and 45° diagonal directions, which is

$$\begin{bmatrix}-2 & 1 & -2\\ 1 & 4 & 1\\ -2 & 1 & -2\end{bmatrix} + \begin{bmatrix}1 & -2 & 1\\ -2 & 4 & -2\\ 1 & -2 & 1\end{bmatrix} = \begin{bmatrix}-1 & -1 & -1\\ -1 & 8 & -1\\ -1 & -1 & -1\end{bmatrix} \tag{3.24}$$

Thus we have proved that the Laplacian mask on the righthand side of Equation (3-22) detects edges in all directions (*quod erat demonstrandum*).

Sobel Edge Detection. Certain mask operators weight key differencing pixels more and are more effective than certain other edge operators. The *Prewitt* operators were previously defined to be

$$\begin{bmatrix}-1 & 0 & 1\\ -1 & 0 & 1\\ -1 & 0 & 1\end{bmatrix} \quad \begin{bmatrix}-1 & -1 & -1\\ 0 & 0 & 0\\ 1 & 1 & 1\end{bmatrix} \quad \begin{bmatrix}-1 & -1 & 0\\ -1 & 0 & 1\\ 0 & 1 & 1\end{bmatrix} \quad \begin{bmatrix}0 & 1 & 1\\ -1 & 0 & 1\\ -1 & -1 & 0\end{bmatrix}$$

The next set of mask edge operators are the *Sobel* operators that we defined previously and that are more powerful and useful than those of Prewitt (they weight the central differences more).

$$\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix} \quad \begin{bmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ 1 & 2 & 1\end{bmatrix} \quad \begin{bmatrix}-2 & -1 & 0\\ -1 & 0 & 1\\ 0 & 1 & 2\end{bmatrix} \quad \begin{bmatrix}0 & 1 & 2\\ -1 & 0 & 1\\ -2 & -1 & 0\end{bmatrix}$$

Note that the sum of the entries in each differencing mask is zero. This means that if a neighborhood consists of pixels of constant value a, then the new pixel has value

$$g_{new} = \sum)_{(j=0,pq-1)}\, h_j f_j \;=\; \sum_{(j=0,pq-1)}\, h_j a = a\{\sum_{(j=0,pq-1)}\, h_j\} \;= a\{0\} = 0 \tag{3.25}$$

Thus pixels with near constant neighborhood values are changed to near black (near 0). If the average of the pixels in the positive direction is greater than that in the negative direction (both taken from the center), then the new pixel value g_{new} will be brighter and contrast against the darker background. If the average pixel value in the negative direction is greater than that in the positive direction, then the new pixel value computed will be negative, and will be truncated to zero and so will be black (0). For L levels of grayscale, all computed pixel values are bounded by 0 and L-1 (usually 0 and 255). Thus each difference (edge detecting) operator works in only one direction.

After convolution with all four difference operators, the edges will be light lines on a dark background in four different images. These can be thresholded to bi-level images of black and white and then combined via use of the maximum at each pixel.

Sobel* Edge Detection with *Matlab. The function *edge()* is what we use in *Matlab* to apply different edge detection methods such as, Sobel, Canny, Prewitt, Laplacian, etc. To apply Sobel edge detection, the following command line is used.

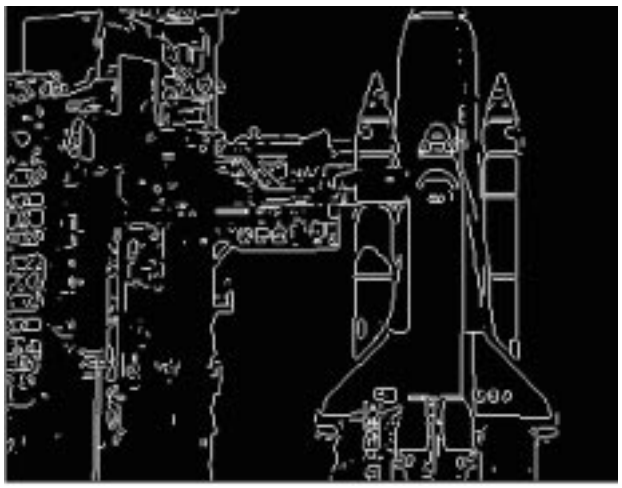
>> Im_out= *edge*(Im_in, 'sobel', threshold, direction);

The second parameter in the function is a string that specifies the *Sobel* method. The third parameter, *threshold*, is numerical to specify the sensitivity threshold for the *Sobel* method. All edges that are not stronger than *threshold* will be ignored. If *threshold* is not specified by the user when the function is called, it will be done automatically. The fourth parameter, *direction*, specifies the direction of detection for the *Sobel* method. The parameter *direction* is a string specifying whether to look for 'horizontal' or 'vertical' edges, or 'both' (the default).

With the default settings, the function can be simply used as Im_out = *edge* (Im_in, 'sobel', threshold), Im_out = *edge*(Im_in, 'sobel', 'direction') or Im_out = *edge*(Im_in, 'sobel'). The execution of following code gives the edges shown in Figure 3.21. A threshold of *0.05* worked best for this example.

```
>> Im1=imread('shuttle.tif ');          //load image into memory at Im1
>> Im2=edge(Im1, 'sobel', 0.05);        //Sobel edge detect and store at Im2
>> imshow (Im2);                        //show image at Im2 on screen
```

Figure 3.21. Sobel results on *shuttle.*



The *Canny* Edge Detector. The Canny method employs a Guassian low pass filter first to smooth the image before edge detection. The standard deviation α is an input parameter that determines the width of the filter and hence the amount of smoothing. Then the gradient vector (the magnitude and direction of the gray level change) at each pixel of the smoothed image is calculated. Next, non-maximal suppression and hysteresis thresholding are applied, where non-maximal suppression thins the wide ridges around local maxima in gradient magnitude down to edges that are only one pixel wide. Hysteresis thresholding uses two thresholds, T_{low} and T_{high}. The higher one is used to mark the best edge pixel candidates. It then attempts to grow these pixels into contours by searching for neighbours with gradient magnitudes higher than T_{low} for connecting together with lines.

Canny Edge Detection with *Matlab*. We still use the *edge( )* function here, but apply *Canny* edge detection by substituting *canny* in place of *sobel*. We call it with the following parameter settings.

>> Im_out = *edge*(Im_in, 'canny', threshold, sigma)

While the second parameter specifies the Canny method, *threshold* specifies sensitivity thresholds for the Canny method. The value of *threshold* itself is taken as the high threshold, and 0.4**threshold* is used for the low threshold. If you do not specify *threshold*, low and high thresholds will be chosen automatically. The parameter *sigma* is used as the standard deviation of the *Gaussian* filter. The default *sigma* is 1 and the size of the filter is chosen automatically, based on *sigma*.

Also, we can leave *threshold* or both *threshold* and *sigma* unspecified as shown below.

```
>> Im_out = edge(Im_in, 'canny', thresh);

>> Im_out = edge(Im_in, 'canny');
```

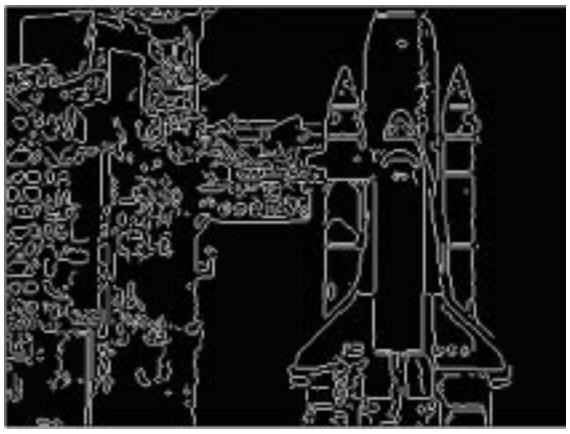
The following *Matlab* script uses threshold = 0.1 and α = 0.6 to yield the image in Figure 3.22, which is displayed below.

```
>>Im1=imread('shuttle.tif ');          //load image into memory at Im1

>>Im2=edge(Im1, 'canny', 0.1, 0.6);    //Canny edge detect, store at Im2

>>imshow (Im2);                        //show image at Im2 on screen
```

Figure 3.22. *Canny* results on *shuttle.tif*.



Edge Detection with *XView*. To run *XView* and bring up the image *shuttle.tif* we type in the command line shown below.

```
> xv shuttle.tif
```

Next, right-click inside the image to bring up the control window (see Figure 1.5). On the upper-right corner, there is the *Algorithm* item. Left click and hold down and move down to the *Edge Detect* item and release the button. Now only the detected edges are left on the original image as shown in Figure 3.23.

The *Color Editor* can be used to reverse the result to black line drawing on a white background. Chose the *Color Edit* item on the control window and adjust the *Intensity* profile as shown in Figure 3.24 to get an image shown as in Figure 3.25.

Figure 3.23. *XView* detected edges.

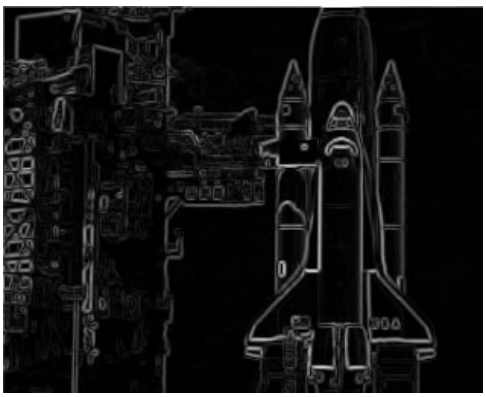


Fig. 3.24. Intensity profile of color inversion.

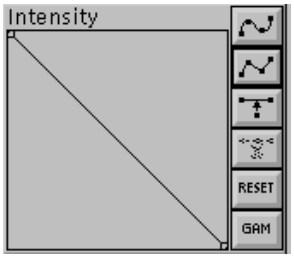
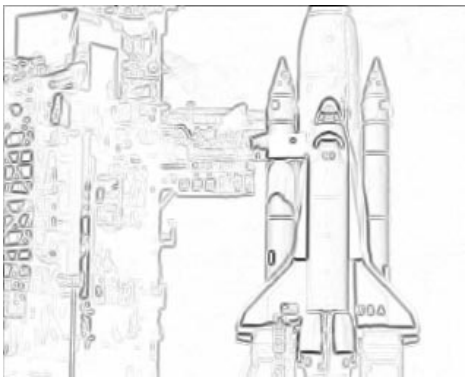


Figure 3.25. Edges after inversion.



Edge Detection with *LView Pro*. Run the program, load an image and then click on the *Color* item on the top menu bar. Select *Filters* on the menu that comes up. The *Pre-defined Image Filters* window (shown as in Figure 3.26) will pop up. Click on the *Find Edges* item on the left side and then click on *Apply* on the right side. Click *Close* to keep the modified image (shown as in Figure 3.27) or else click *Revert* to get the original image back.

To invert the color of the result image, click *Color* on the top menu and then *Adjustments* on the menu that comes up. When the *Pre-defined Color Adjustments* window (See Figure 1.14) pops up, select the *Negative* item on it.

Fig. 3.26. *LView Pro Pre-defined Image Filters* window.

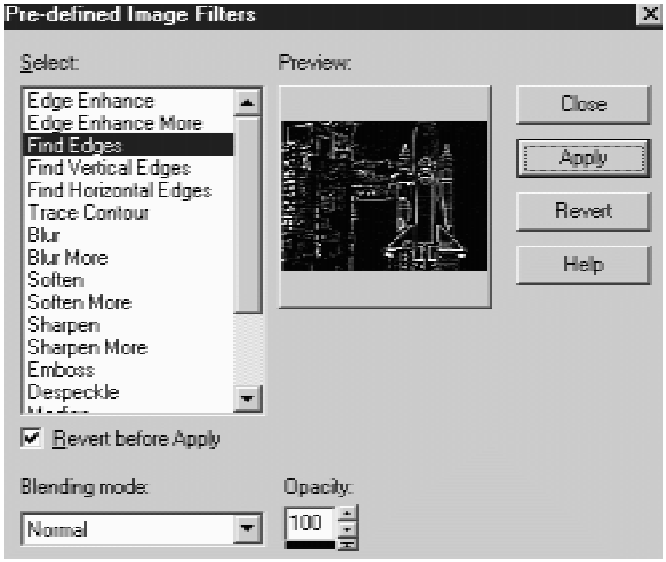
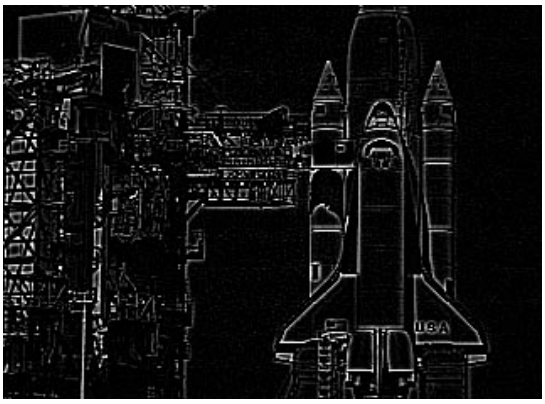


Fig. 3.27. Shuttle Edges by *LView Pro.*



Otherwise, we can choose *Color*, *User-defined*, *Filters*, *New* and enter a mask that detects edges. Then click on Apply to process with the mask.

Competitive Fuzzy Edge Detection. Fuzzy logic can also be applied on image edge detection. Here we introduce a fuzzy classifier, which first classifies image pixels as different kinds of edge pixels or non-edge pixels. It then applies competitive rules on them according to their classification so that not all the edge pixels are output as edges, which thins the edges. This fuzzy classifier, is called the *competitive fuzzy edge detector*.

The competitive fuzzy classifier is made up of the fuzzy classifier part and the rule-based competition part. In the fuzzy classifier, there are six classes: background, vertical edge, horizontal edge, two kinds of diagonal edge classes and a speckle edge classes. Either a Gaussian or an extended Epanechnikov function [Liang and Looney] is used for each class as its fuzzy set membership function. The fuzzy truth for a pixel to be a member of a class is given by the evaluation of its function on the vector of features for that pixel.

Each pixel determines a vector of differences (d(1), d(2), d(3), d(4)) in the four directions across that pixel in its 3x3 nbhd. For example, the vertical difference is

$$d(1) = |p_2 - p_5| + |p_8 - p_5| \tag{3.26}$$

A feature vector is evaluated by putting it through each fuzzy set membership function that is centered on a prototypical difference vector for that class. Thus each pixel has a fuzzy truth value for each class and the maximum fuzzy truth determines the class of that pixel.

In the competition, neighboring edge pixels in the same direction compete with each other in directional edge strength. To be a black edge pixel in the output image, a classified directional edge pixel must have larger edge strength on its edge direction in comparison to its neighbors on that direction. If an edge pixel does not win the competition, then it will be output as white (background). The result is a black line drawing on a white background.

Speckle edge pixels are mapped to black directly without competition. This may introduce isolated single/double-pixel speckles. A despeckler operates on the image after all other processes have been completed so as to remove these speckles. See [Liang and Looney, 2002] for complete details.

Fuzzy Classification

Step1: set parameters for the fuzzy set membership functions; open the image file;
Step2: for each pixel in the image
    compute graylevel change magnitudes in the 4 different directions
    construct the pixel feature vector from those magnitudes
    for each class
        compute the fuzzy truth of the feature vector
    determine maximum fuzzy truth and record pixel class for that pixel

Edge Strength Competition

Step1: for each pixel in the image
    if (edge class) then apply competitive rule and record pixel value
    if (background class) then write white pixel
    if (speckle edge class) then write black pixel

Despeckling

Step1: for each pixel in the image
    if (pixel is isolated single/double speckle) then change to white.

Fig. 3.26. Original *peppers*. **Fig. 3.27. CFED *peppers* results.** **Fig. 3.28. Canny *peppers* results.**

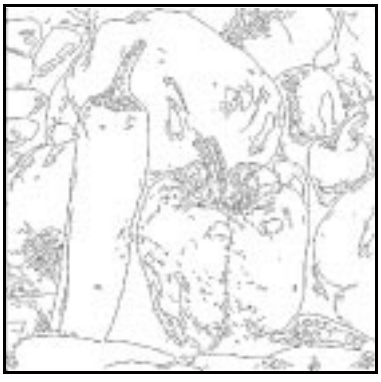


Figure 3.28 is the original peppers image, which presents a difficult edge detection problem. The results of the competitive fuzzy edge detection method are shown in Figure 3.29 and the results of the Canny method with the threshold set at 0.04 and α set at 0.5 are shown in Figure 3.30. The CFED method took about 1/10 of the time that the Canny method took.

3.5 Exercises

3.1 Is it possible to construct a 5x5 mask that detects a weak (smooth) vertical edge that is dark on the left side and light on the right side, but that doesn't use the middle column (it consists of zeros)? If so, explain how it works.

3.2 Develop an algorithm whereby a pixel is smoothed if all pixels in its neighborhood are close in graylevel to its graylevel but is sharpened if a majority of the neighborhood pixels differ significantly. In the processing, use neighborhoods of differences between the pixels and the origin (center) pixel.

3.3 Discuss the effect of unsharp masking when ß > 1 and the smoothing mask to be subtracted is multiplied by α = ß - 1.

3.4 What is the result of adding a sharpening and a smoothing mask to obtain a mask whose entries sum to unity?

3.5 Develop a 5x5 Laplacian mask by adding line detectors in the horizontal, vertical and ±45° directions.

3.6 Complete Equation (3.19c). Explain the effects of each of the two parts.

3.7 Invert *shuttle.pgm* and then smooth it lightly before detecting its edges. Show the results.

3.8 Use *XView* or *LView Pro* to threshold the image of building.tif to convert it to black and white only.

3.9 Write a simple program in C that performs mask convolution (see Appendix 3.B).

3.10 Perform unsharp masking on shuttle.tif to sharpen it by entering a single mask into Matlab.

3.11 What effect does ß have (try a higher and lower value than ß = 2) on unsharp masking? What effect does the degree of smoothing have? Experiment with these and describe the results.

3.12 Describe an overall method to process an image to remove noise, detect edges and convert it to a line drawing. Implement this method on *building.tif*.