CHAPTER 10

NEIGHBORHOOD PROCESSING

WHAT WILL WE LEARN?

- What is neighborhood processing and how does it differ from point processing?
- What is convolution and how is it used to process digital images?
- What is a low-pass linear filter, what is it used for, and how can it be implemented using 2D convolution?
- What is a median filter and what is it used for?
- What is a high-pass linear filter, what is it used for, and how can it be implemented using 2D convolution?

10.1 NEIGHBORHOOD PROCESSING

The underlying theme throughout this chapter is the use of neighborhood-oriented operations for image enhancement. The basics of neighborhood processing were introduced in Section 2.4.2. We call neighborhood-oriented operations those image processing techniques in which the resulting value for a pixel at coordinates (x_0, y_0)—which we shall call the *reference pixel*—is a function of the original pixel value at that point as well as the original pixel value of some of its neighbors. The way by which the neighboring values and the reference pixel value are combined to produce the result can vary significantly among different algorithms. Many algorithms work in a

Practical Image and Video Processing Using MATLAB®. By Oge Marques.

linear way and use 2D convolution (which essentially consists of sums of products, see Section 10.2), while others process the input values in a nonlinear way.

Regardless of the type (linear or nonlinear), neighborhood processing operations follow a sequence of steps [GWE04]:

1. Define a reference point in the input image, $f(x_0, y_0)$.
2. Perform an operation that involves only pixels within a neighborhood around the reference point in the input image.
3. Apply the result of that operation to the pixel of same coordinates in the output image, $g(x_0, y_0)$.
4. Repeat the process for every pixel in the input image.

In this chapter, we provide a representative collection of neighborhood-based image processing techniques for the sake of image enhancement, particularly blurring or sharpening.[1]

The techniques described in this chapter belong to one of these two categories:

- *Linear Filters*: Here the resulting output pixel is computed as a sum of products of the pixel values and mask coefficients in the pixel's neighborhood in the original image. Example: mean filter (Section 10.3.1).
- *Nonlinear Filters*: Here the resulting output pixel is selected from an ordered (ranked) sequence of pixel values in the pixel's neighborhood in the original image. Example: median filter (Section 10.3.4).

10.2 CONVOLUTION AND CORRELATION

Convolution is a widely used mathematical operator that processes an image by computing—for each pixel—a weighted sum of the values of that pixel and its neighbors (Figure 10.1). Depending on the choice of weights, a wide variety of image processing operations can be implemented. Convolution and correlation are the two fundamental mathematical operations involved in linear neighborhood-oriented image processing algorithms. The two operations differ in a very subtle way, which will be explained later in this section.

10.2.1 Convolution in the One-Dimensional Domain

The convolution between two discrete one-dimensional (1D) arrays $A(x)$ and $B(x)$, denoted by $A * B$, is mathematically described by the equation

$$A * B = \sum_{j=-\infty}^{\infty} A(j) \cdot B(x - j) \tag{10.1}$$

[1] We shall see additional examples of neighborhood operations for different purposes—for example, image restoration (Chapter 12) and edge detection (Chapter 14)—later in the book.

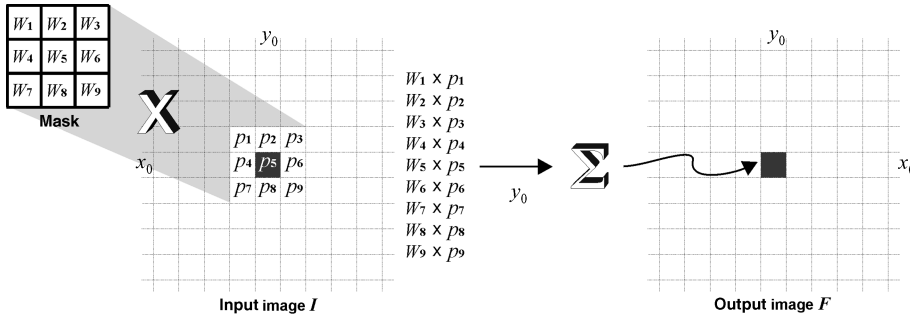


FIGURE 10.1 Neighborhood processing for the case of linear filtering.

■ EXAMPLE 10.1

In this example, we show how the result of a 1D convolution operation can be obtained step-by-step. Let $A = \{0, 1, 2, 3, 2, 1, 0\}$ and $B = \{1, 3, -1\}$. The partial results of multiplying elements in A with corresponding elements in B, as B shifts from $-\infty$ to ∞, are displayed below.

1. Initially, we mirror array B and align its center (reference) value with the first (leftmost) value of array A.[2] The partial result of the convolution calculation $(0 \times (-1)) + (0 \times 3) + (1 \times 1) = 1$ (where empty spots are assumed as zero) is stored in the resulting array $(A * B)$.

A	0	1	2	3	2	1	0
B	-1	3	1				
$A * B$	1						

2. Array B is shifted one position to the right. The partial result of the convolution calculation $(0 \times (-1)) + (1 \times 3) + (2 \times 1) = 5$ is stored in the resulting array $(A * B)$.

A	0	1	2	3	2	1	0
B	-1	3	1				
$A * B$	1	5					

3. Array B is shifted another position to the right. The partial result of the convolution calculation $(1 \times (-1)) + (2 \times 3) + (3 \times 1) = 8$ is stored in the resulting array $(A * B)$.

A	0	1	2	3	2	1	0
B		-1	3	1			
$A * B$	1	5	8				

[2]This is equivalent to saying that all the partial products from $-\infty$ until that point are equal to zero and, therefore, do not contribute to the result of the convolution operation.

4. Array B is shifted another position to the right. The partial result of the convolution calculation $(2 \times (-1)) + (3 \times 3) + (2 \times 1) = 9$ is stored in the resulting array $(A * B)$.

A	0	1	2	3	2	1	0
B			-1	3	1		
$A * B$	1	5	8	8			

5. Array B is shifted another position to the right. The partial result of the convolution calculation $(3 \times (-1)) + (2 \times 3) + (1 \times 1) = 4$ is stored in the resulting array $(A * B)$.

A	0	1	2	3	2	1	0
B				-1	3	1	
$A * B$	1	5	8	8	4		

6. Array B is shifted another position to the right. The partial result of the convolution calculation $(2 \times (-1)) + (1 \times 3) + (0 \times 1) = 1$ is stored in the resulting array $(A * B)$.

A	0	1	2	3	2	1	0
B					-1	3	1
$A * B$	1	5	8	8	4	1	

7. Array B is shifted another position to the right. The partial result of the convolution calculation $(1 \times (-1)) + (0 \times 3) + (0 \times 1) = -1$ is stored in the resulting array $(A * B)$.

A	0	1	2	3	2	1	0	
B						-1	3	1
$A * B$	1	5	8	8	4	1	-1	

The final result of the convolution operation is the array $\{1, 5, 8, 8, 4, 1, -1\}$.

10.2.2 Convolution in the Two-Dimensional Domain

The mathematical definition for 2D convolution is

$$g(x, y) = \sum_{k=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} h(j, k) \cdot f(x - j, y - k) \tag{10.2}$$

In practice, this is rewritten as

$$g(x, y) = \sum_{k=-n_2}^{n_2} \sum_{j=-m_2}^{m_2} h(j, k) \cdot f(x - j, y - k) \tag{10.3}$$

where m_2 is equal to half of the mask's width and n_2 is equal to half of the mask's height, that is,

$$m_2 = \lfloor m/2 \rfloor \tag{10.4}$$

and

$$n_2 = \lfloor n/2 \rfloor \tag{10.5}$$

where $\lfloor x \rfloor$ is the *floor* operator, which rounds a number to the nearest integer less than or equal to x.

The basic mechanism used to understand 1D convolution can be expanded to the 2D domain. In such cases, the 2D array A is usually the input image and B is a small (usually 3×3) mask. The idea of mirroring B and shifting it across A can be adapted to the 2D case as well: mirroring will now take place in both x and y dimensions, and shifting will be done starting from the top left point in the image, moving along each line, until the bottom right pixel in A has been processed.

■ EXAMPLE 10.2

Let

$$A = \begin{bmatrix} 5 & 8 & 3 & 4 & 6 & 2 & 3 & 7 \\ 3 & 2 & 1 & 1 & 9 & 5 & 1 & 0 \\ 0 & 9 & 5 & 3 & 0 & 4 & 8 & 3 \\ 4 & 2 & 7 & 2 & 1 & 9 & 0 & 6 \\ 9 & 7 & 9 & 8 & 0 & 4 & 2 & 4 \\ 5 & 2 & 1 & 8 & 4 & 1 & 0 & 9 \\ 1 & 8 & 5 & 4 & 9 & 2 & 3 & 8 \\ 3 & 7 & 1 & 2 & 3 & 4 & 4 & 6 \end{bmatrix}$$

and

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

The result of the convolution $A * B$ will be

$$A * B = \begin{bmatrix} 20 & 10 & 2 & 26 & 23 & 6 & 9 & 4 \\ 18 & 1 & -8 & 2 & 7 & 3 & 3 & -11 \\ 14 & 22 & 5 & -1 & 9 & -2 & 8 & -1 \\ 29 & 21 & 9 & -9 & 10 & 12 & -9 & -9 \\ 21 & 1 & 16 & -1 & -3 & -4 & 2 & 5 \\ 15 & -9 & -3 & 7 & -6 & 1 & 17 & 9 \\ 21 & 9 & 1 & 6 & -2 & -1 & 23 & 2 \\ 9 & -5 & -25 & -10 & -12 & -15 & -1 & -12 \end{bmatrix}$$

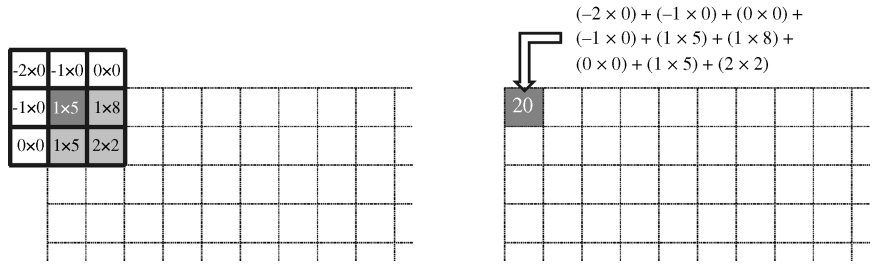


FIGURE 10.2 Two-dimensional convolution example.

Figure 10.2 shows the calculation of the top left pixel in the resulting image in detail. Note how B has been flipped in both dimensions *before* the sum of products was calculated.

Convolution with masks is a very versatile image processing method. Depending on the choice of mask coefficients, entirely different results can be obtained, for example, image blurring, image sharpening, or edge detection.

■ EXAMPLE 10.3

Suppose we apply the three convolution masks from Table 10.1 (one at a time, independently) to the same input image (Figure 10.3a). The resulting images will be a blurred version of the original (part (b)), a sharpened version of the original (part (c)), and an image indicating the presence of horizontal edges in the input image (part (d)).

10.2.3 Correlation

In the context of this chapter, 1D correlation[3] can be mathematically expressed as

$$A \odot B = \sum_{j=-\infty}^{\infty} A(j) \cdot B(x + j) \tag{10.6}$$

whereas the 2D equivalent is given by

$$g(x, y) = \sum_{k=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} h(j, k) \cdot f(x + j, y + k) \tag{10.7}$$

[3]This definition of correlation should not be confused with the most common use of the word, usually to express the degree to which two or more quantities are linearly associated (by means of a *correlation coefficient*).

TABLE 10.1 Examples of Convolution Masks

Low-Pass Filter	High-Pass Filter	Horizontal Edge Detection
$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$

In practice, this is rewritten as

$$g(x, y) = \sum_{k=-n_2}^{n_2} \sum_{j=-m_2}^{m_2} h(j, k) \cdot f(x + j, y + k) \tag{10.8}$$

where m_2 and n_2 are as defined earlier.

Simply put, correlation is the same as convolution *without* the mirroring (flipping) of the mask before the sums of products are computed. The difference between using correlation and convolution in 2D neighborhood processing operations is often irrelevant because many popular masks used in image processing are symmetrical around the origin. Consequently, many texts omit this distinction and

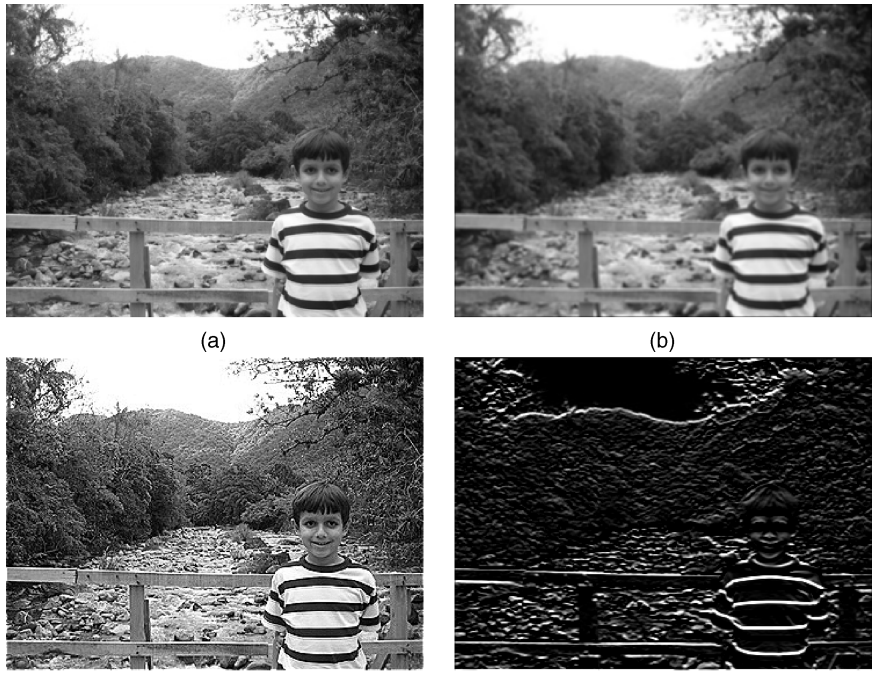


FIGURE 10.3 Applying different convolution masks to the same input image: (a) original image; (b–d) result of 2D convolution using the masks in Table 10.1.

refer to *convolution* or *spatial filtering* when referring to what technically should be called *correlation*.

In MATLAB

MATLAB's Image Processing Toolbox (IPT) has two built-in functions that can be used to implement 2D convolution:

- `conv2`: It computes the 2D convolution between two matrices. In addition to the two matrices, it takes a third parameter that specifies the size of the output:
  - `full`: Returns the full 2D convolution (default).
  - `same`: Returns the central part of the convolution of the same size as A.
  - `valid`: Returns only those parts of the convolution that are computed without the zero-padded edges.
- `filter2`: It rotates the convolution mask (which is treated as a 2D FIR filter) 180° in each direction to create a convolution kernel and then calls `conv2` to perform the convolution operation.

10.2.4 Dealing with Image Borders

Our discussion of convolution and correlation so far has overlooked the need to deal with image borders, that is, those points in the input image for which part of the mask falls outside the image borders (Figure 10.4). There are several ways of handling this:

1. Ignore the borders, that is, apply the mask only to the pixels in the input image for which the mask falls entirely within the image. There are two variants of this approach:
   (a) Keep the pixel values that cannot be reached by the overlapping mask untouched. This will introduce artifacts due to the difference between the processed and unprocessed pixels in the output image.
   (b) Replace the pixel values that cannot be reached by the overlapping mask with a constant fixed value, usually zero (black). If you use this approach, the resulting image will be smaller than the original image. This is unacceptable because the image size will be further decreased by every subsequent filtering operation. Moreover, it will make it harder to combine the input and output images using an arithmetic or logic operation (see Chapter 6) or even to compare them on a pixel-by-pixel basis.
2. Pad the input image with zeros, that is, assume that all values outside the image are equal to zero. If you use this approach, the resulting image will show unwanted artifacts, in this case artificial dark borders, whose width is proportional to the size of the convolution mask. This is implemented in MATLAB by choosing the option `X` (with `X` = 0) for the `boundary_options` parameter for function `imfilter`.

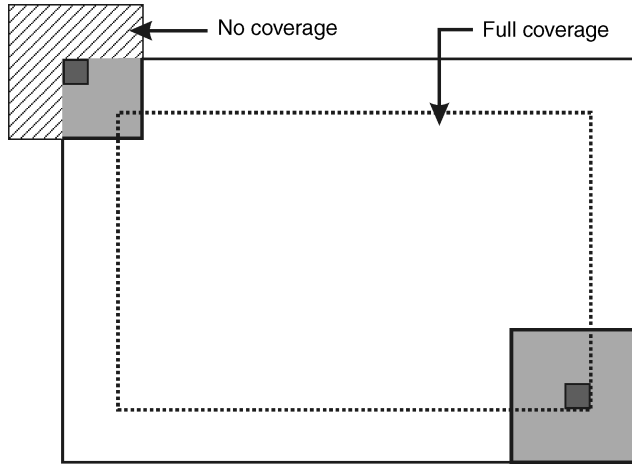


FIGURE 10.4 Border geometry. Redrawn from [BB08].

3. Pad with extended values, that is, assume that the pixel values in the input image extend beyond the image borders. This is equivalent to assuming that the input image has extra rows and columns whose pixel values are identical to the ones of the row/column closest to the border. This is the preferred method because of its simplicity and the relatively low impact of the resulting artifacts on the quality of the output image. This is implemented in MATLAB by choosing the option `'replicate'` for the `boundary_options` parameter for function `imfilter`.
4. Pad with mirrored values, that is, assume that the pixel values in the input image extend beyond the image borders in such a way as to mirror the pixel values in the rows/columns closest to the border. For small mask sizes, the result is comparable to using the *padding with extended values* approach.
5. Treat the input image as a 2D periodic function whose values repeat themselves in both horizontal and vertical directions. This is implemented in MATLAB by choosing the option `'circular'` for the `boundary_options` parameter for function `imfilter`.

10.3 IMAGE SMOOTHING (LOW-PASS FILTERS)

We have seen that different convolution masks can dramatically produce different results when applied to the same input image (Figure 10.3). It is common to refer to these operations as *filtering* operations and to the masks themselves as *spatial filters*. Spatial filters are often named based on their behavior in the *spatial frequency*

domain[4]: we call *low-pass* filters (LPFs) those spatial filters whose effect on the output image is equivalent to attenuating high-frequency components (i.e., fine details in the image) and preserving low-frequency components (i.e., coarser details and homogeneous areas in the image). Linear LPFs can be implemented using 2D convolution masks with nonnegative coefficients. Linear LPFs are typically used to either blur an image or reduce the amount of noise present in the image. In this chapter we refer to both uses, but defer a more detailed discussion of noise reduction techniques until Chapter 12.

High-pass filters (HPFs) work in a complementary way to LPFs, that is, they preserve or enhance high-frequency components (with the possible side effect of enhancing noisy pixels as well). HPFs will be discussed in Section 10.4.

In MATLAB

Linear filters are implemented in MATLAB using two functions: `imfilter` and—optionally—`fspecial`.

The syntax for `imfilter` is

```
g = imfilter(f, h, mode, boundary_options, size_options);
```

where

- `f` is the input image.
- `h` is the filter mask.
- `mode` can be either `'conv'` or `'corr'`, indicating, respectively, whether filtering will be done using convolution or correlation (which is the default);
- `boundary_options` refer to how the filtering algorithm should treat border values. There are four possibilities:
  1. `X`: The boundaries of the input array (image) are extended by padding with a value `X`. This is the default option (with `X` = 0).
  2. `'symmetric'`: The boundaries of the input array (image) are extended by mirror-reflecting the image across its border.
  3. `'replicate'`: The boundaries of the input array (image) are extended by replicating the values nearest to the image border.
  4. `'circular'`: The boundaries of the input array (image) are extended by implicitly assuming the input array is periodic, that is, treating the image as one period of a 2D periodic function.
- `size_options`: There are two options for the size of the resulting image: `'full'` (output image is the full filtered result, that is, the size of the extended/padded image) or `'same'` (output image is of the same size as input image), which is the default.
- `g` is the output image.

[4]Frequency-domain image processing techniques will be discussed in Chapter 11.

`fspecial` is an IPT function designed to simplify the creation of common 2D image filters. Its syntax is `h = fspecial(type, parameters)`, where

- `h` is the filter mask.
- `type` is one of the following:
  - `'average'`: Averaging filter
  - `'disk'`: Circular averaging filter
  - `'gaussian'`: Gaussian low-pass filter
  - `'laplacian'`: 2D Laplacian operator
  - `'log'`: Laplacian of Gaussian (LoG) filter
  - `'motion'`: Approximates the linear motion of a camera
  - `'prewitt'` and `'sobel'`: horizontal edge-emphasizing filters
  - `'unsharp'`: unsharp contrast enhancement filter
- `parameters` are optional parameters that vary depending on the `type` of filter, for example, mask size, standard deviation (for `'gaussian'` filter), and so on. See the IPT documentation for full details.

10.3.1 Mean Filter

The *mean* (also known as *neighborhood averaging*) filter is perhaps the simplest and most widely known spatial smoothing filter. It uses convolution with a (usually 3×3) mask whose coefficients have a value of 1 and divides the result by a scaling factor (the total number of elements in the mask). A neighborhood averaging filter in which all coefficients are equal is also referred to as a *box filter*.

The convolution mask for a 3×3 mean filter is given by

$$h(x, y) = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{10.9}$$

Figure 10.3b shows the result of applying the mask in equation (10.9) to the image in Figure 10.3a.

The same concept can be applied to larger neighborhoods with a proportional increase in the degree of blurriness of the resulting image. This is illustrated in Figure 10.5 for masks of size 7×7 (part (b)), 15×15 (part (c)), and 31×31 (part (d)).

10.3.2 Variations

Many variations on the basic neighborhood averaging filter have been proposed in the literature. In this section we summarize some of them.

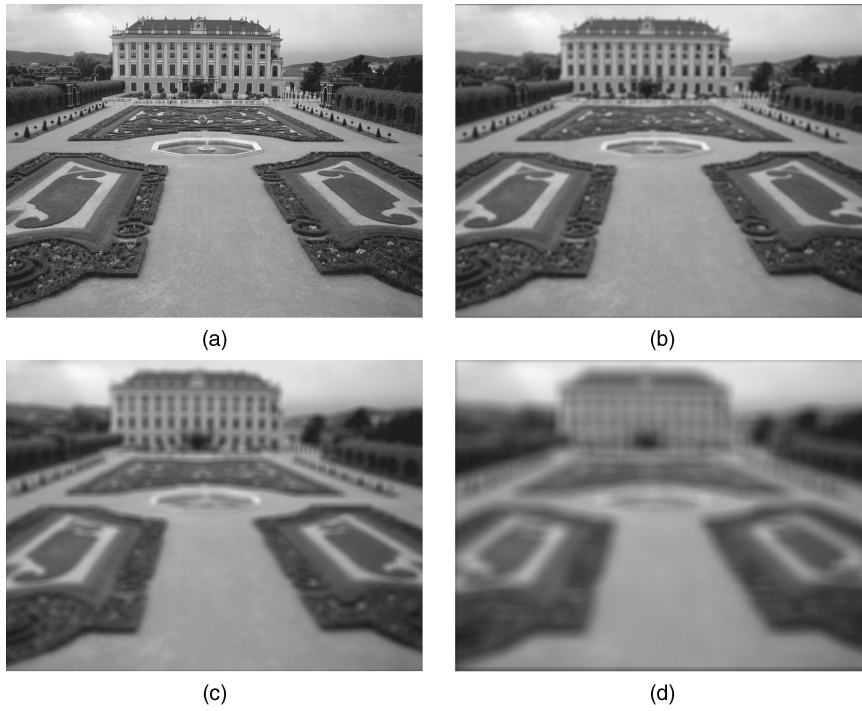


FIGURE 10.5 Examples of applying the averaging filter with different mask sizes: (a) input image (899 × 675 pixels); (b–d) output images corresponding to averaging masks of size 7 × 7, 15 × 15, and 31 × 31.

Modified Mask Coefficients The mask coefficients from equation (10.9) can be modified, for example, to give more importance to the center pixel and its 4-connected neighbors:

$$h(x, y) = \begin{bmatrix} 0.075 & 0.125 & 0.075 \\ 0.125 & 0.2 & 0.125 \\ 0.075 & 0.125 & 0.075 \end{bmatrix} \tag{10.10}$$

Directional Averaging The square mask can be replaced by a rectangular equivalent to emphasize that the blurring is done in a specific direction.

Selective Application of Averaging Calculation Results Another variation of the basic neighborhood averaging filter consists in applying a decision step between calculating of the neighborhood average for a certain reference pixel and applying the result to the output image. A simple decision step would compare the difference between the original and processed values against a predefined threshold (T): if the

difference is less than T, the calculated value is applied; otherwise, the original value is kept. This is usually done to minimize the blurring of important edges in the image.

Removal of Outliers Before Calculating the Average This is the underlying idea of the *average of k nearest-neighbors* technique [DR78], which is a variation of the average filter, whose basic procedure consists of four steps:

1. Sort all pixel values in the neighborhood.
2. Select k values around the median value.
3. Calculate the average gray-level value of the k values selected in step 2.
4. Replace reference (central) pixel in the destination image with the value calculated in step 3.

It was conceived to allow the exclusion of high-contrast or edge pixels from the average calculations and, therefore, reduce the degree of edge blurring in the resulting image. For larger values of k, this filter's performance will approach the conventional average filter.

10.3.3 Gaussian Blur Filter

The Gaussian blur filter is the best-known example of a LPF implemented with a nonuniform kernel. The mask coefficients for the Gaussian blur filter are samples from a 2D Gaussian function (plotted in Figure 10.6):

$$h(x, y) = \exp\left[\frac{-(x^2 + y^2)}{2\sigma^2}\right] \tag{10.11}$$

The parameter σ controls the overall shape of the curve: the larger the value of σ, the flatter the resulting curve.

■ **EXAMPLE 10.4**

Figure 10.7 shows an example (using the `imfilter` and `fspecial` functions in MATLAB) of applying a Gaussian blur filter to a monochrome image using different kernel sizes and values of σ. You should be able to notice that the Gaussian blur produces a more natural blurring effect than the averaging filter. Moreover, the impact of increasing mask size is less dramatic on the Gaussian blur filter than on the averaging filter.

Some of the most notable properties of the Gaussian blur filter are as follows:

- The kernel is symmetric with respect to rotation; therefore, there is no directional bias in the result.
- The kernel is separable, which can lead to fast computational implementations.
- The kernel's coefficients fall off to (almost) zero at the kernel's edges.

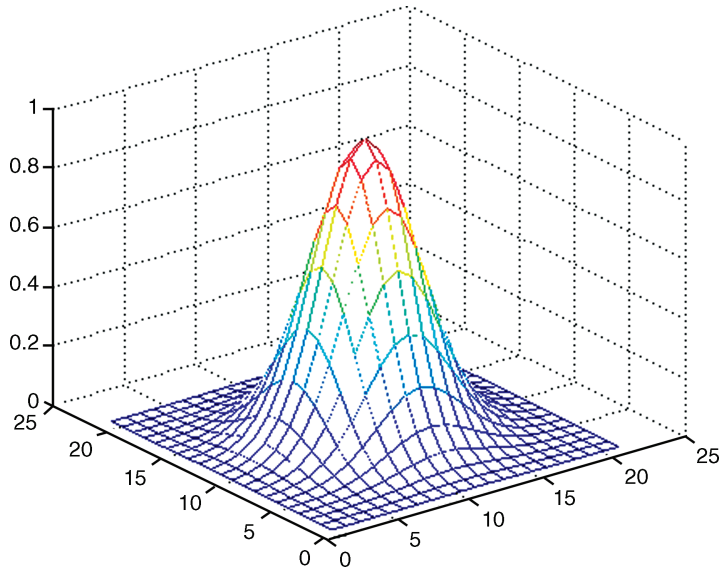


FIGURE 10.6 A 2D Gaussian function (with $\sigma = 3$).

- The Fourier transform (FT) of a Gaussian filter is another Gaussian (this will be explained in Chapter 11).
- The convolution of two Gaussians is another Gaussian.
- The output image obtained after applying the Gaussian blur filter is more pleasing to the eye than the one obtained using other low-pass filters.

10.3.4 Median and Other Nonlinear Filters

As stated earlier in this chapter, nonlinear filters also work at a neighborhood level, but do not process the pixel values using the convolution operator. Instead, they usually apply a ranking (sorting) function to the pixel values within the neighborhood and select a value from the sorted list. For this reason, these are sometimes called *rank filters*. Examples of nonlinear filters include the median filter (described in this section) and the *max* and *min* filters (which will be described in Chapter 12).

The median filter is a popular nonlinear filter used in image processing. It works by sorting the pixel values within a neighborhood, finding the median value, and replacing the original pixel value with the median of that neighborhood (Figure 10.8).

The median filter works very well (and significantly better than an averaging filter with comparable neighborhood size) in reducing "salt and pepper" noise (a type of noise that causes very bright—salt— and very dark—pepper—isolated spots to appear in an image) from images. Figure 10.9 compares the results obtained using median filtering and the averaging filter for the case of an image contaminated with salt and pepper noise.

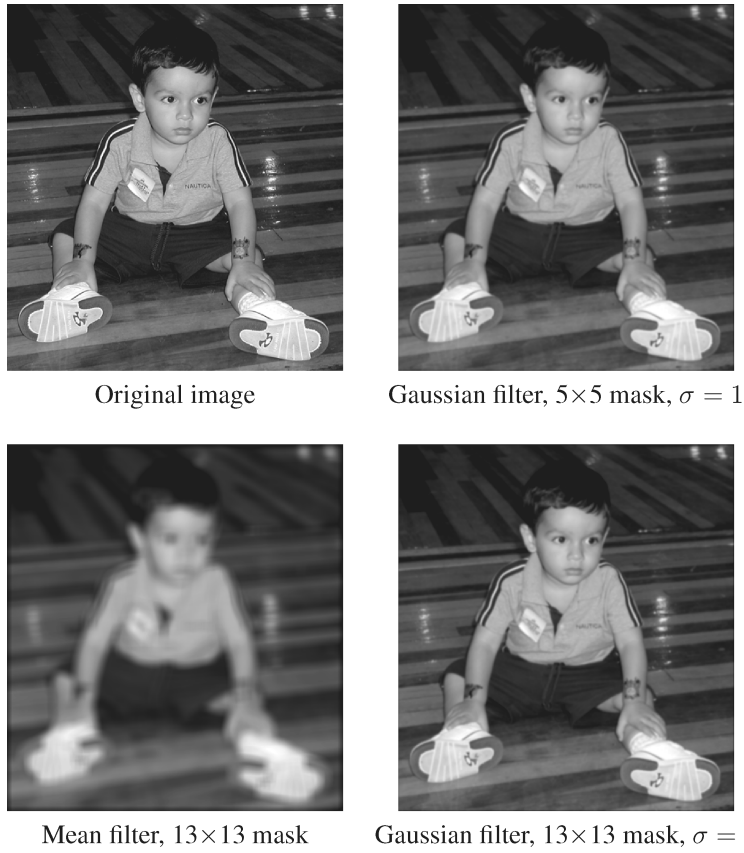


FIGURE 10.7 Example of using Gaussian blur filters.

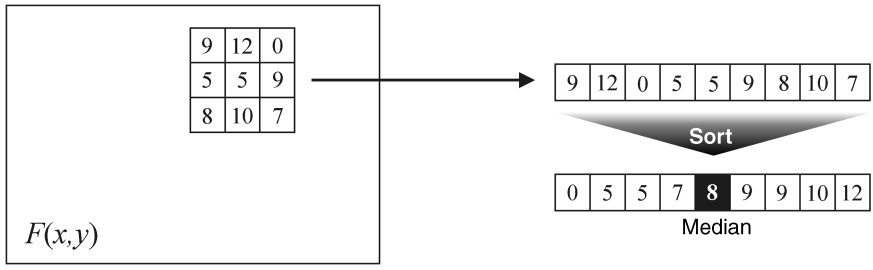


FIGURE 10.8 Median filter. Redrawn from [BB08].

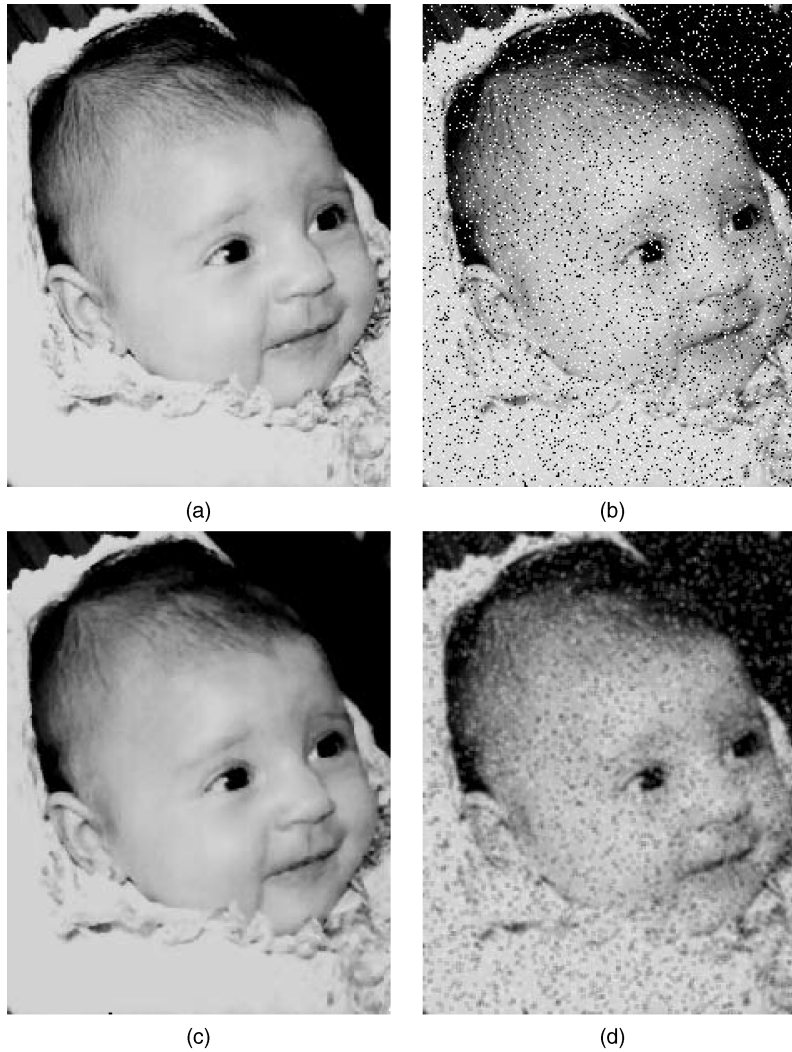


FIGURE 10.9 (a) Original image; (b) image with salt and pepper noise; (c) result of 3×3 median filtering; (d) result of 3×3 neighborhood averaging.

We shall resume our discussion of median filter, its implementation details, some of its variants, and its use in noise reduction in Chapter 12.

10.4 IMAGE SHARPENING (HIGH-PASS FILTERS)

We call *high-pass* filters those spatial filters whose effect on an image is equivalent to preserving or emphasizing its high-frequency components (i.e., fine details, points, lines, and edges), that is, to highlight transitions in intensity within the image.

Linear HPFs can be implemented using 2D convolution masks with positive and negative coefficients, which correspond to a digital approximation of the *Laplacian*, a simple, *isotropic* (i.e., *rotation invariant*) second-order derivative that is capable of responding to intensity transitions in any direction.

10.4.1 The Laplacian

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

$$\nabla^2(x, y) = \frac{\partial^2(x, y)}{\partial x^2} + \frac{\partial^2(x, y)}{\partial y^2} \tag{10.12}$$

where the second derivatives are usually approximated—for digital signals—as

$$\frac{\partial^2(x, y)}{\partial x^2} = f(x + 1, y) + f(x - 1, y) - 2f(x, y) \tag{10.13}$$

and

$$\frac{\partial^2(x, y)}{\partial y^2} = f(x, y + 1) + f(x, y - 1) - 2f(x, y) \tag{10.14}$$

which results in a convenient expression for the Laplacian expressed as a sum of products:

$$\nabla^2(x, y) = f(x + 1, y) + f(x - 1, y) + f(x, y + 1) + f(x, y - 1) - 4f(x, y) \tag{10.15}$$

This expression can be implemented by the convolution mask below:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

An alternative digital implementation of the Laplacian takes into account all eight neighbors of the reference pixel in the input image and can be implemented by the convolution mask below:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Note that it is common to find implementations in which the signs of each coefficient in the convolution masks above are reversed.

10.4.2 Composite Laplacian Mask

High-pass filters can be implemented using the Laplacian defined in equation (10.15) and combining the result with the original image as follows:

$$g(x, y) = f(x, y) + c[\nabla^2(x, y)] \tag{10.16}$$

where c is a constant used to comply with the sign convention for the particular implementation of the Laplacian mask: $c = 1$ if the center coefficient is positive, while $c = -1$ if the same is negative.

The goal of adding the original image to the results of the Laplacian is to restore the gray-level tonality that was lost in the Laplacian calculations.[5]

It is worth noting that if we want to implement the composite Laplacian mask using the `fspecial` function in MATLAB, we need to figure out the correct value of `alpha`.[6] Choosing `alpha = 0` will result in the following mask:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

It is also common to factor equation (10.16) into the design of the mask, which produces the *composite* Laplacian mask below:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

■ **EXAMPLE 10.5**

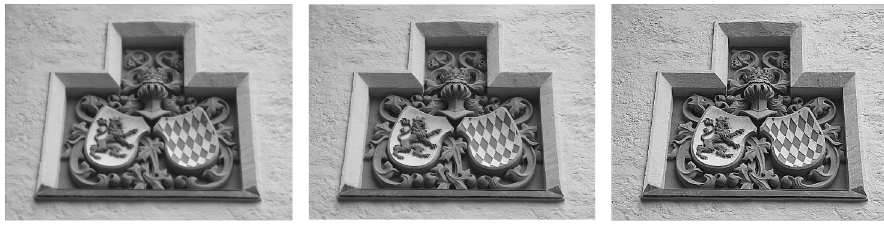
Figure 10.10 shows an example (using the `imfilter` and `fspecial` functions in MATLAB) of applying a high-pass filter to enhance (sharpen) a monochrome image. Figure 10.10a shows the original image. Figure 10.10b shows the resulting enhanced image obtained by applying equation (10.16) with $c = -1$ and Figure 10.10c shows the result of using the eight-directional Laplacian operator instead. It can be claimed that the results in part (c) are crisper than the ones obtained in part (b).

10.4.3 Directional Difference Filters

Directional difference filters are similar to the Laplacian high-frequency filter discussed earlier. The main difference is that—as their name suggests—directional

[5]The Laplacian mask, as well as any mask whose coefficients add up to zero, tends to produce results centered around zero, which correspond to very dark images, whose only bright spots are indicative of what the mask is designed to detect or emphasize, in this case, omnidirectional edges.

[6]Refer to the IPT documentation for further details.



(a) (b) (c)

FIGURE 10.10 Example of using Laplacian masks to enhance an image.

difference filters emphasize edges in a specific direction. There filters are usually called *emboss filters*. There are four representative masks that can be used to implement the emboss effect:

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

10.4.4 Unsharp Masking

The unsharp masking technique consists of computing the subtraction between the input image and a blurred (low-pass filtered) version of the input image. The rationale behind this technique is to "increase the amount of high-frequency (fine) detail by reducing the importance of its low-frequency contents." There have been many variants of this basic idea proposed in the literature. In Tutorial 10.3 (page 227), you will use MATLAB to implement unsharp masking in three different ways.

10.4.5 High-Boost Filtering

The high-boost filtering technique—sometimes referred to as *high-frequency emphasis*—emphasizes the fine details of an image by applying a convolution mask:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & c & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

where c ($c > 8$) is a coefficient—sometimes called *amplification factor*—that controls how much weight is given to the original image and the high-pass filtered version of that image. For $c = 8$, the results would be equivalent to those seen earlier for the conventional isotropic Laplacian mask (Figure 10.10c). Greater values of c will cause significantly less sharpening.

10.5 REGION OF INTEREST PROCESSING

Filtering operations are sometimes performed only in a small part of an image—known as a *region of interest* (ROI)—which can be specified by defining a (usually binary) *mask* that delimits the portion of the image in which the operation will take place. *Image masking* is the process of extracting such a subimage (or ROI) from a larger image for further processing.

In MATLAB

ROI processing can be implemented in MATLAB using a combination of two functions: `roipoly`—introduced in Tutorial 6.2—for image masking and `roifilt2` for the actual processing of the selected ROI. Selecting a polygonal ROI can be done interactively—clicking on the polygon vertices—or programmatically—specifying the coordinates of the vertices using two separate vectors (one for rows and one for columns).

■ EXAMPLE 10.6

Figure 10.11 shows an example of ROI processing using the `roifilt2` function in MATLAB.

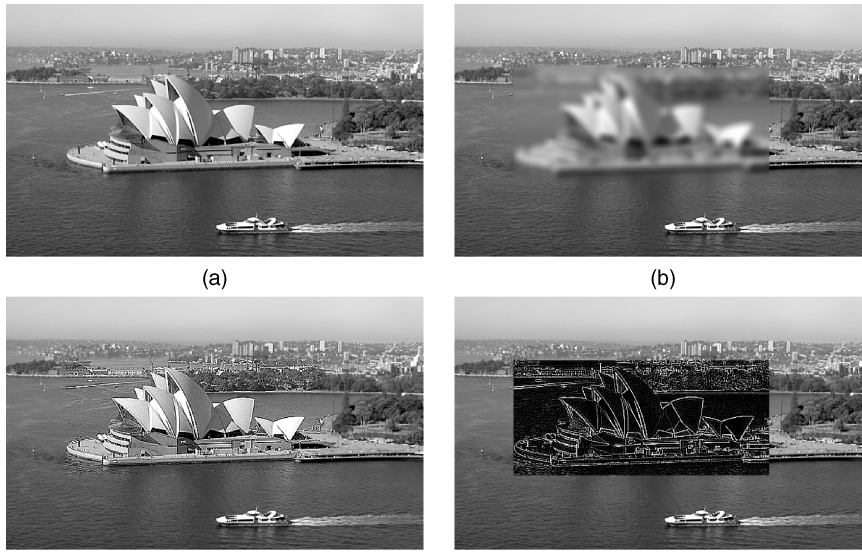


FIGURE 10.11 Example of region of interest processing: (a) original image; (b) result of applying a Gaussian blur to a selected ROI; (c) result of applying a HPF to a selected ROI; (d) result of applying a Laplacian mask to a selected ROI.

10.6 COMBINING SPATIAL ENHANCEMENT METHODS

At the end of this chapter—especially after working on its tutorials—you must have seen a significant number of useful methods and algorithms for processing monochrome images. A legitimate question to ask at this point is as follows "When faced with a practical image processing problem, which techniques should I use and in which sequence?" Naturally, there is no universal answer to this question. Most image processing solutions are problem specific and usually involve the application of several algorithms—in a meaningful sequence—to achieve the desired goal. The choice of algorithms and fine-tuning of their parameters is an almost inevitable trial-and-error process that most image processing solution designers have to go through. Using the knowledge acquired so far and a tool that allows easy experimentation—MATLAB—you should be able to implement, configure, fine-tune, and combine image processing algorithms for a wide variety of real-world problems.

10.7 TUTORIAL 10.1: CONVOLUTION AND CORRELATION

Goal

The goal of this tutorial is to learn how to perform a correlation and convolution calculations in MATLAB.

Objectives

- Learn how to perform a correlation of two (1D and 2D) matrices.
- Learn how to perform a convolution of two (1D and 2D) matrices.
- Explore the `imfilter` function to perform correlation and convolution in MATLAB.

Procedure

We shall start by exploring convolution and correlation in one dimension. This can be achieved by means of the `imfilter` function.

1. Specify the two matrices to be used.

```
a = [0 0 0 1 0 0 0];
f = [1 2 3 4 5];
```

2. Perform convolution, using `a` as the input matrix and `f` as the filter.

```
g = imfilter(a,f,'full','conv')
```

Question 1 What is the relationship between the size of the output matrix, the size of the original matrix, and the length of the filter?

Question 2 How does changing the third parameter from `'full'` to `'same'` affect the output?

3. Perform correlation on the same set of matrices.

```
h = imfilter(a,f,'full','corr')
```

The results from the previous step should confirm that convolution is related to correlation by a reflection of the filter matrix, regardless of the number of dimensions involved.

Let us see how correlation works on a small window of size 3×3. Consider the window of values extracted from a larger image in Figure 10.12.

The correlation of two matrices is a sum of products. Numerically, the calculation would be as follows:

$$(140)(-1) + (108)(0) + (94)(1) + (89)(-2) + (99)(0) + (125)(2) + (121)(-1) + (134)(0) + (221)(1) = 126$$

Here, we specify the image (which in our case will be the image region as in Figure 10.12) and the mask from Figure 10.13. We will also explicitly tell the function to use correlation, as it can perform both correlation and convolution.

4. Clear all workspace variables.
5. Use `imfilter` to perform a correlation of the two matrices.

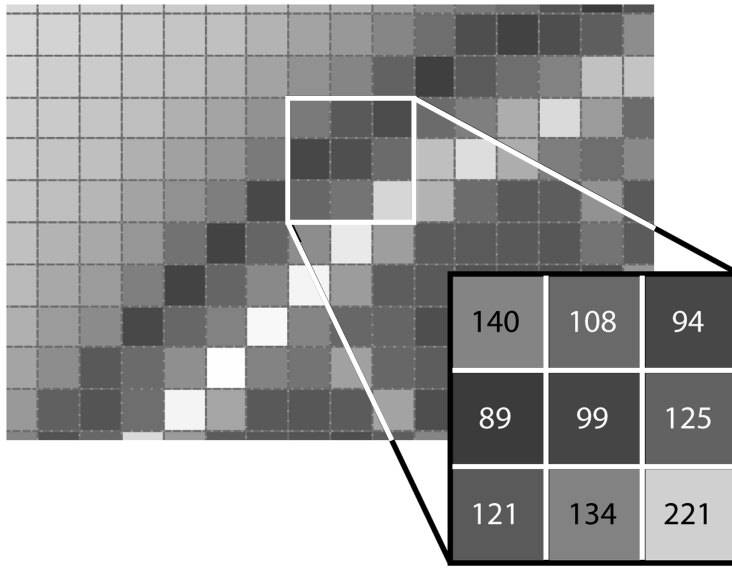


FIGURE 10.12 A 3×3 image region.

–1	0	1
–2	0	2
–1	0	1

FIGURE 10.13 A 3 × 3 mask.

```
x = [140 108 94;89 99 125;121 134 221]
y = [-1 0 1;-2 0 2;-1 0 1]
z = imfilter(x,y,'corr')
```

Question 3 In the resulting matrix (`z`), we are interested only in the center value. How does this value compare with our calculation illustrated above?

Question 4 What are the other values in the resulting matrix?

Question 5 Note in the last step we did not specify if the output should be `'full'` or `'same'`. What is the default for this setting if it is not specified?

To perform convolution, we use the same technique as in correlation. The difference here is that the filter matrix is rotated 180° before performing the sum of products. Again, the calculation of the convolution of the given image region and mask is performed as follows:

$$(140)(1) + (108)(0) + (94)(-1) + (89)(2) + (99)(0) + (125)(-2) + (121)(1) + (134)(0) + (221)(-1) = -126$$

6. Use `imfilter` to perform a convolution of the two matrices.

```
z2 = imfilter(x,y,'conv')
```

Question 6 How does the center value of the resulting matrix compare with our calculation above?

10.8 TUTORIAL 10.2: SMOOTHING FILTERS IN THE SPATIAL DOMAIN

Goal

The goal of this tutorial is to learn how to implement smoothing filters in the spatial domain.

Objectives

- Learn how to use the `fspecial` function to generate commonly used kernels.

- Explore applying smoothing filters to images using the `imfilter` function.
- Learn how to implement uniform and nonuniform averaging masks.
- Learn how to implement a Gaussian mask.

Procedure

In the first part of this procedure, we will use the `imfilter` function to implement a 3×3 mean (average) filter. We could easily generate the mask array ourselves (nine values, each equal to 1/9), but the IPT offers a function that will automatically create this and several other commonly used masks.

1. Load the `cameraman` image and prepare a subplot.

```
I = imread('cameraman.tif');
figure, subplot(1,2,1), imshow(I), title('Original Image');
```

2. Create a mean (averaging) filter automatically through the `fspecial` function.

```
fn = fspecial('average')
```

Question 1 Explain what the value of the variable `fn` represents.

Question 2 What other commonly used masks is the `fspecial` function capable of generating?

3. Filter the `cameraman` image with the generated mask.

```
I_new = imfilter(I,fn);
subplot(1,2,2), imshow(I_new), title('Filtered Image');
```

Question 3 What was the effect of the averaging filter?

The mean filter we just implemented was a uniform filter—all coefficients were equivalent. The nonuniform version of the mean filter gives the center of the mask (the pixel in question) a higher weighted value, while all other coefficients are weighted by their distance from the center. This particular mask cannot be generated by the `fspecial` function, so we must create it ourselves.

4. Create a nonuniform version of the mean filter.

```
fn2 = [1 2 1; 2 4 2; 1 2 1]
fn2 = fn2 * (1/16)
```

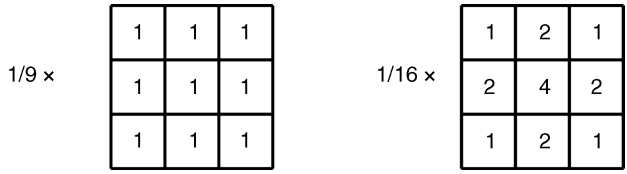


FIGURE 10.14 Uniform and nonuniform averaging masks.

Recall that the uniform mean filter could be created by generating a 3×3 matrix of 1's, and then multiplying each coefficient by a factor of 1/9. In the nonuniform mean filter implantation above, note that the sum of all the original values in the filter equals 16—this is why we divide each coefficient by 16 in the second step. Figure 10.14 illustrates the previous two masks we created.

5. Filter the original image with the new, nonuniform averaging mask.

```
I_new2 = imfilter(I,fn2);
figure, subplot(1,2,1), imshow(I_new), title('Uniform Average');
subplot(1,2,2), imshow(I_new2), title('Non-uniform Average');
```

Question 4 Comment on the subjective differences between using the uniform averaging filter and the nonuniform averaging filter.

The Gaussian filter is similar to the nonuniform averaging filter in that the coefficients are not equivalent. The coefficient values, however, are not a function of their distance from the center pixel, but instead are modeled from the Gaussian curve.

6. Create a Gaussian filter and display the kernel as a 3D plot.

```
fn_gau = fspecial('gaussian',9,1.5);
figure, bar3(fn_gau,'b'), ...
title('Gaussian filter as a 3D graph');
```

7. Filter the `cameraman` image using the Gaussian mask.

```
I_new3 = imfilter(I,fn_gau);
figure
subplot(1,3,1), imshow(I), title('Original Image');
subplot(1,3,2), imshow(I_new), title('Average Filter');
subplot(1,3,3), imshow(I_new3), title('Gaussian Filter');
```

Question 5 Experiment with the size of the Gaussian filter and the value of σ. How can you change the amount of blur that results from the filter?

10.9 TUTORIAL 10.3: SHARPENING FILTERS IN THE SPATIAL DOMAIN

Goal

The goal of this tutorial is to learn how to implement sharpening filters in the spatial domain.

Objectives

- Learn how to implement the several variations of the Laplacian mask.
- Explore different implementations of the unsharp masking technique.
- Learn how to apply a high-boost filtering mask.

Procedure

To implement the Laplacian filter, we can either create our own mask or use the `fspecial` function to generate the mask for us. In the next step, we will use `fspecial`, but keep in mind that you can just as well create the mask on your own.

1. Load the `moon` image and prepare a subplot figure.

```
I = imread('moon.tif');
Id = im2double(I);
figure, subplot(2,2,1), imshow(Id), title('Original Image');
```

We are required to convert the image to doubles because a Laplacian filtered image can result in negative values. If we were to keep the image as class `uint8`, all negative values would be truncated and, therefore, would not accurately reflect the results of having applied a Laplacian mask. By converting the image to `doubles`, all negative values will remain intact.

2. Create a Laplacian kernel and apply it to the image using the `imfilter` function.

```
f = fspecial('laplacian',0);
I_filt = imfilter(Id,f);
subplot(2,2,2), imshow(I_filt), title('Laplacian of Original');
```

Question 1 When specifying the Laplacian filter in the `fspecial` function, what is the second parameter (in the case above, 0) used for?

Question 2 What is the minimum value of the filtered image?

Question 3 Verify that a `uint8` filtered image would not reflect negative numbers. You can use the image `I` that was previously loaded.

You will notice that it is difficult to see details of the Laplacian filtered image. To get a better perspective of the detail the Laplacian mask produced, we can scale the image for display purposes so that its values span the dynamic range of the gray scale.

3. Display a scaled version of the Laplacian image for display purposes.

```
subplot(2,2,3), imshow(I_filt,[]), title('Scaled Laplacian');
```

The center coefficient of the Laplacian mask we created is negative. Recall from the chapter that if the mask center is negative, we subtract the filtered image from the original, and if it is positive, we add. In our case, we will subtract them.

4. Subtract the filtered image from the original image to create the sharpened image.

```
I_sharp = imsubtract(Id,I_filt);
subplot(2,2,4), imshow(I_sharp), title('Sharpened Image');
```

A composite version of the Laplacian mask performs the entire operation all at once. By using this composite mask, we do not need to add or subtract the filtered image—the resulting image *is* the sharpened image.

5. Use the composite Laplacian mask to perform image sharpening in one step.

```
f2 = [0 -1 0; -1 5 -1; 0 -1 0]
I_sharp2 = imfilter(Id,f2);
figure, subplot(1,2,1), imshow(Id), title('Original Image');
subplot(1,2,2), imshow(I_sharp2), title('Composite Laplacian');
```

Question 4 You may have noticed that we created the mask without using the `fspecial` function. Is the `fspecial` function capable of generating the simplified Laplacian mask?

Question 5 Both Laplacian masks used above did not take into account the four corner pixels (their coefficients are 0). Reapply the Laplacian mask, but this time use the version of the mask that accounts for the corner pixels as well. Both the standard and simplified versions of this mask are illustrated in Figure 10.15. How does accounting for corner pixels change the output?

Unsharp Masking Unsharp masking is a simple process of subtracting a blurred image from its original to generate a sharper image. Although the concept is

1	1	1
1	–8	1
1	1	1

–1	–1	–1
–1	9	–1
–1	–1	–1

FIGURE 10.15 Laplacian masks that account for corner pixels (standard and composite).

straightforward, there are three ways it can be implemented. Figures 10.16–10.18 illustrate these processes.

Let us first implement the process described in Figure 10.16.

6. Close all open figures and clear all workspace variables.
7. Load the `moon` image and generate the blurred image.

```
I = imread('moon.tif');
f_blur = fspecial('average',5);
I_blur = imfilter(I,f_blur);
figure, subplot(1,3,1), imshow(I), title('Original Image');
subplot(1,3,2), imshow(I_blur), title('Blurred Image');
```

Question 6 What does the second parameter of the `fspecial` function call mean?

We must now shrink the histogram of the blurred image. The amount by which we shrink the histogram will ultimately determine the level of enhancement in the final result. In our case, we will scale the histogram to range between 0.0 and 0.4, where the full dynamic grayscale range is [0.0 1.0].

8. Shrink the histogram of the blurred image.

```
I_blur_adj = imadjust(I_blur,stretchlim(I_blur),[0 0.4]);
```

9. Now subtract the blurred image from the original image.

```
I_sharp = imsubtract(I,I_blur_adj);
```

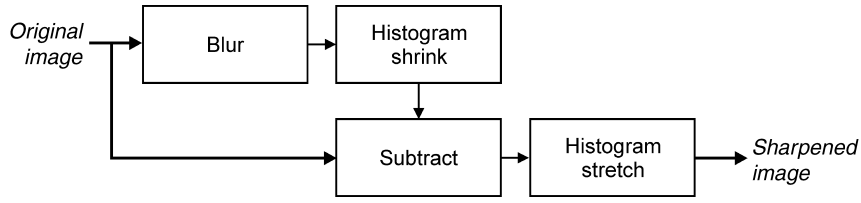


FIGURE 10.16 Unsharp masking process including histogram adjustment.

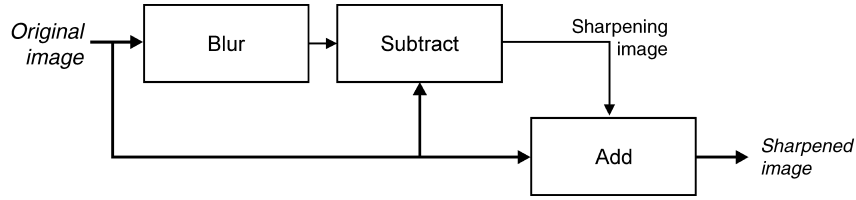


FIGURE 10.17 Unsharp masking process with sharpening image.

We must now perform a histogram stretch on the new image in order to account for previously shrinking the blurred image.

10. Stretch the sharpened image histogram to the full dynamic grayscale range and display the final result.

```
I_sharp_adj = imadjust(I_sharp);
subplot(1,3,3), imshow(I_sharp_adj), title('Sharp Image');
```

Question 7 We learned that by shrinking the blurred image's histogram, we can control the amount of sharpening in the final image by specifying the maximum range value. What other factor can alter the amount of sharpening?

We will now look at the second implementation of the unsharp masking technique, illustrated in Figure 10.17. We have already generated a blurred version of the `moon` image, so we can skip that step.

11. Subtract the blurred image from the original image to generate a sharpening image.

```
I_sharpening = imsubtract(I,I_blur);
```

12. Add sharpening image to original image to produce the final result.

```
I_sharp2 = imadd(I,I_sharpening);
figure, subplot(1,2,1), imshow(I), title('Original Image');
subplot(1,2,2), imshow(I_sharp2), title('Sharp Image');
```

Question 8 How can we adjust the amount of sharpening when using this implementation?

The third implementation uses a convolution mask, which can be generated using the `fspecial` function. This implementation is illustrated in Figure 10.18.

13. Generate unsharp masking kernel using the `fspecial` function.

```
f_unsharp = fspecial('unsharp');
```

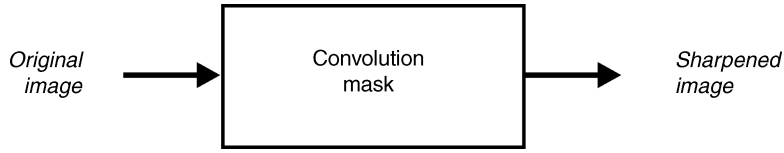


FIGURE 10.18 Unsharp masking process using convolution mask.

14. Apply the mask to the original image to create a sharper image.

```
I_sharp3 = imfilter(I,f_unsharp);
figure, subplot(1,2,1), imshow(I), title('Original Image');
subplot(1,2,2), imshow(I_sharp3), title('Sharp Image');
```

Question 9 How do we control the level of sharpening with this implementation?

High-Boost Filtering High-boost filtering is a sharpening technique that involves creating a sharpening image and adding it to the original image. The mask used to create the sharpening image is illustrated in Figure 10.19. Note that there are two versions of the mask: one that does not include the corner pixels and another that does.

15. Close any open figures.
16. Create a high-boost mask (where `A` = 1) and apply it to the `moon` image.

```
f_hb = [0 -1 0; -1 5 -1; 0 -1 0];
I_sharp4 = imfilter(I,f_hb);
figure, subplot(1,2,1), imshow(I), title('Original Image');
subplot(1,2,2), imshow(I_sharp4), title('Sharp Image');
```

Question 10 What happens to the output image when `A` is less than 1? What about when `A` is greater than 1?

You may have noticed that when `A` = 1, the high-boost filter generalizes to the composite Laplacian mask discussed in step 5. As the value of `A` increases, the output image starts to resemble an image multiplied by a constant.

0	−1	0
−1	A+4	−1
0	−1	0

−1	−1	−1
−1	A+8	−1
−1	−1	−1

FIGURE 10.19 High-boost masks with and without regard to corner pixels.

17. Show that a high-boost mask when A = 3 looks similar to the image simply multiplied by 3.

```
f_hb2 = [0 -1 0; -1 7 -1; 0 -1 0];
I_sharp5 = imfilter(I,f_hb2);
I_mult = immultiply(I,3);
figure, subplot(1,3,1), imshow(I), title('Original Image');
subplot(1,3,2), imshow(I_sharp5), title('High Boost, A = 3');
subplot(1,3,3), imshow(I_mult), title('Multiplied by 3');
```

Question 11 At what value of A does this filter stop being effective (resemble the image multiplied by a constant)?

WHAT HAVE WE LEARNED?

- Neighborhood processing is the name given to image processing techniques in which the new value of a processed image pixel is a function of its original value and the values of (some of) its neighbors. It differs from point processing (Chapters 8 and 9), where the resulting pixel value depends only on the original value and on the transformation function applied to each pixel in the original image.
- Convolution is a widely used mathematical operator that processes an image by computing—for each pixel—a weighted sum of values of that pixel and its neighbors. Depending on the choice of weights, a wide variety of image processing operations can be implemented.
- Low-pass filters are used to smooth an image or reduce the amount of noise in it. Low-pass linear filters can be implemented using 2D convolution masks with nonnegative coefficients. The mean (averaging) filter is the simplest—and most popular—low-pass linear filter. It works by averaging out the pixel values within a neighborhood.
- Low-pass nonlinear filters also work at a neighborhood level, but do not process the pixel values using the convolution operator. The median filter is one of the most popular low-pass linear filters. It works by sorting the pixel values within a neighborhood, finding the median value, and replacing the original pixel value with the median of that neighborhood. The median filter works extremely well in removing salt and pepper noise from images.
- Convolution operators can also be used to detect or emphasize the high-frequency contents of an image, such as fine details, points, lines, and edges. In such cases, the resulting filter is usually called a high-pass filter. High-pass linear filters can be implemented using 2D convolution masks with positive and negative coefficients.
- The spatial-domain image enhancement techniques discussed in Chapters 8–10 can be combined to solve specific image processing problems. The secret of

success consists in deciding which techniques to use, how to configure their parameters (e.g., window size and mask coefficients), and in which sequence they should be applied.

LEARN MORE ABOUT IT

- Section 6.5 of [BB08] and Section 7.2 of [Eff00] discuss implementation aspects associated with spatial filters.
- Chapter 7 of [Pra07] discusses convolution and correlation in greater mathematical depth.
- Chapter 4 of [Jah05] and Section 6.3 of [BB08] describe the formal properties of linear filters.
- Chapter 11 of [Jah05] is entirely devoted to an in-depth discussion of linear filters.
- Section 3.4 of [Dav04] introduces *mode filters* and compares them with mean and median filters in the context of machine vision applications.
- Section 5.3.1 of [SHB08] discusses several *edge-preserving* variants of the basic neighborhood averaging filter. MATLAB implementation for one of these methods—smoothing using a rotating mask—appears in Section 5.4 of [SKH08].
- Section 3.8 of [GW08] discusses the use of fuzzy techniques for spatial filtering.

10.10 PROBLEMS

10.1 Image processing tasks such as blurring or sharpening an image can easily be accomplished using neighborhood-oriented techniques described in this chapter. Can these tasks also be achieved using point operations (such as the ones described in Chapters 8 and 9)? Explain.

10.2 Write MATLAB code to reproduce the results from Example 10.1.

10.3 Write MATLAB code to reproduce the results from Example 10.2.

10.4 Write MATLAB code to implement a linear filter that creates a horizontal blur over a length of 9 pixels, comparable to what would happen to a image if the camera were moved during the exposure interval.

10.5 Write MATLAB code to implement the emboss effect described in Section 10.4.3 and test it with an input image of your choice. Do the results correspond to what you expected?