4
Enhancement

The techniques we introduced at the end of Chapter 3 considered the manipulation of the dynamic range of a given digital image to improve visualization of its contents. In this chapter we consider more general image enhancement. We introduce the concept of image filtering based on localized image subregions (pixel neighbourhoods), outline a range of noise removal filters and explain how filtering can achieve edge detection and edge sharpening effects for image enhancement.

4.1 Why perform enhancement?

The basic goal of image enhancement is to process the image so that we can view and assess the visual information it contains with greater clarity. Image enhancement, therefore, is rather subjective because it depends strongly on the specific information the user is hoping to extract from the image.

The primary condition for image enhancement is that the information that you want to extract, emphasize or restore must exist in the image. Fundamentally, 'you cannot make something out of nothing' and the desired information must not be totally swamped by noise within the image. Perhaps the most accurate and general statement we can make about the goal of image enhancement is simply that the processed image should be more suitable than the original one for the required task or purpose. This makes the evaluation of image enhancement, by its nature, rather subjective and, hence, it is difficult to quantify its performance apart from its specific domain of application.

4.1.1 Enhancement via image filtering

The main goal of image enhancement is to process an image in some way so as to render it more visually acceptable or pleasing. The removal of noise, the sharpening of image edges and the 'soft focus' (blurring) effect so often favoured in romantic photographs are all examples of popular enhancement techniques. These and other enhancement operations can be achieved through the process of spatial domain filtering. The term spatial domain is arguably somewhat spurious, but is used to distinguish this procedure from frequency domain procedures (discussed in Chapter 5). Thus, spatial domain filtering simply indicates that the filtering process takes place directly on the actual pixels of the image itself.

Fundamentals of Digital Image Processing: A Practical Approach with Examples in Matlab Chris Solomon and Toby Breckon
 ISBN: 978-0-470-84472-4

Therefore, we shall refer simply to filtering in this chapter without danger of confusion. Filters act on an image to change the values of the pixels in some specified way and are generally classified into two types: linear and nonlinear. Linear filters are more common, but we will discuss and give examples of both kinds.

Irrespective of the particular filter that is used, all approaches to spatial domain filtering operate in the same simple way. Each of the pixels in an image – the pixel under consideration at a given moment is termed the target pixel – is successively addressed. The value of the target pixel is then replaced by a new value which depends only on the value of the pixels in a specified neighbourhood around the target pixel.

4.2 Pixel neighbourhoods

An important measure in images is the concept of *connectivity*. Many operations in image processing use the concept of a local image neighbourhood to define a local area of influence, relevance or interest. Central to this theme of defining the local neighbourhood is the notion of pixel connectivity, i.e. deciding which pixels are connected to each other. When we speak of 4-connectivity, only pixels which are N, W, E, S of the given pixel are connected. However, if, in addition, the pixels on the diagonals must also be considered, then we have 8-connectivity (i.e. N, NW, W, NE, SE, E, SW, S are all connected see Figure 4.1).

In Figure 4.1 (right) we use this concept to determine whether region A and region B are connected and use a local connectivity model (here $N \times N = 3 \times 3$) to determine if these are separate or the same image feature. Operations performed locally in images, such as filtering and edge detection, all consider a given pixel location (i, j) in terms of its local pixel neighbourhood indexed as an offset $(i \pm k, j \pm k)$. The size and, hence, the scale of the neighbourhood can be controlled by varying the neighbourhood size parameter N from which an offset k (generally $\lfloor N/2 \rfloor$) is computed. In general, neighbourhoods may be $N \times M$ where $N \neq M$ for unequal influence of pixels in the horizontal and vertical directions. More frequently $N = M$ is commonplace and, hence, $N \times N$ neighbourhoods arise. The majority

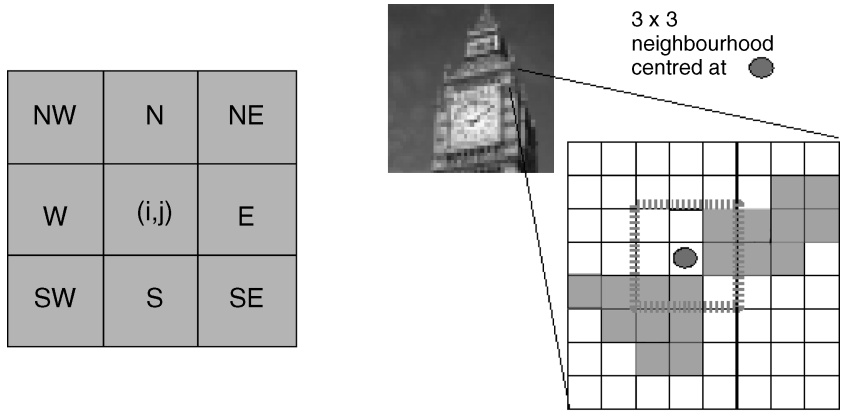


Figure 4.1 Image neighbourhood connectivity (left) and an example 3×3 neighbourhood centred at a specific image pixel location

of image processing techniques now use 8-connectivity by default, which for a reasonable size of neighbourhood is often achievable in real time on modern processors for the majority of operations. Filtering operations over a whole image are generally performed as a series of local neighbourhood operations using a sliding-window-based principle, i.e. each and every pixel in the image is processed based on an operation performed on its local $N \times N$ pixel neighbourhood (region of influence).

In Matlab® such an operation can be performed as in Example 4.1.

Example 4.1

Matlab code	**What is happening?**
A=imread('cameraman.tif');	%Read in image
subplot(1,2,1), imshow(A);	%Display image
func=@(x) max(x(:));	%Set filter to apply
B=nlfilter(A,[3 3],func);	%Apply over 3 × 3 neighbourhood
subplot(1,2,2), imshow(B);	%Display result image B

Comments

Here we specify ***func()*** as the ***max()*** filter function to apply over each and every 3×3 neighbourhood of the image. This replaces every input pixel in the output image with the maximum pixel value of the input pixel neighbourhood. You may wish to experiment with the effects of varying the neighbourhood dimensions and investigating the Matlab ***min()*** and ***mean()*** (for the latter, a type conversion will be required to display the double output type of the Matlab ***mean()*** function as an 8-bit image – specify the filter function as ***uint8(mean())***).

4.3 Filter kernels and the mechanics of linear filtering

In linear spatial filters the new or filtered value of the target pixel is determined as some linear combination of the pixel values in its neighbourhood. Any other type of filter is, by definition, a nonlinear filter. The specific linear combination of the neighbouring pixels that is taken is determined by the filter kernel (often called a mask). This is just an array/sub-image of exactly the same size as the neighbourhood[1] containing the weights that are to be assigned to each of the corresponding pixels in the neighbourhood of the target pixel. Filtering proceeds by successively positioning the kernel so that the location of its centre pixel coincides with the location of each target pixel, each time the filtered value being calculated by the chosen weighted combination of the neighbourhood pixels. This filtering procedure can thus be visualized as sliding the kernel over all locations of interest in the original image (i, j), multiplying the pixels underneath the kernel by the corresponding weights w, calculating the new values by summing the total and copying them to the same locations in a new (filtered) image f (e.g. Figure 4.2).

The mechanics of linear spatial filtering actually express in discrete form a process called convolution, an important physical and mathematical phenomenon which we will develop

[1] A point of detail for the purist. With linear filters it is really the kernel that comes first and thus *defines* the neighbourhood. For some nonlinear filters (e.g. order filters) the order is reversed because we must define the region over which we wish to do the ranking and cannot write down a kernel.

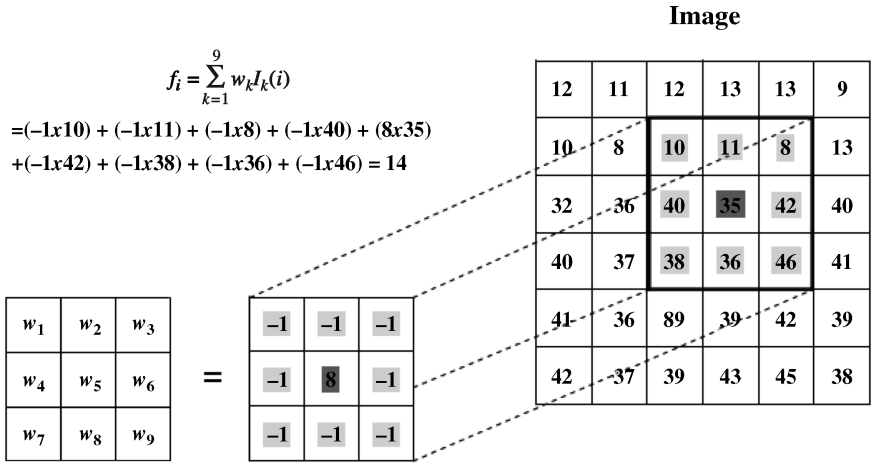


Figure 4.2 The mechanics of image filtering with an $N \times N = 3 \times 3$ kernel filter

further in Chapter 5. For this reason, many filter kernels are sometimes described as convolution kernels, it then being implicitly understood that they are applied to the image in the linear fashion described above. Formally, we can express the action of convolution between a kernel and an image in two equivalent ways. The first addresses the row and column indices of the image and the kernel:

$$f(x,y) = \sum_{i=I_{\min}}^{I_{\max}} \sum_{j=J_{\min}}^{J_{\max}} w(i,j) I(x+i, y+j) \tag{4.1}$$

Here, the indices $i = 0$, $j = 0$ correspond to the centre pixel of the kernel which is of size $(I_{\max} - I_{\min} + 1, J_{\max} - J_{\min} + 1)$. A second, equivalent approach is to use linear indices:

$$f_i = \sum_{k=1}^{N} w_k I_k(i) \tag{4.2}$$

In this case, $I_k(i)$ represents the neighbourhood pixels of the ith image pixel, where k is a linear index running over the neighbourhood region according to a row-wise (as in Figure 4.2) or column-wise convention. Here, w_k are the corresponding kernel values and f_i represents the filtered value resulting from original input value $I_k(i)$. The former notation is more explicit, whereas the latter is neater and more compact; but neither is generally recommended over the other. Figure 4.2 illustrates this basic procedure, where the centre pixel of the kernel and the target pixel in the image are indicated by the dark grey shading. The kernel is 'placed' on the image so that the centre and target pixels match. The filtered value of the target pixel f_i is then given by a linear combination of the neighbourhood pixel values, the specific weights being determined by the kernel values w_k. In this specific case the target pixel value of original value 35 is filtered to an output value of 14.

The steps in linear (convolution) filtering can be summarized as follows:

(1) Define the filter kernel.

(2) Slide the kernel over the image so that its centre pixel coincides with each (target) pixel in the image.

(3) Multiply the pixels lying beneath the kernel by the corresponding values (weights) in the kernel above them and sum the total.

(4) Copy the resulting value to the same locations in a new (filtered) image.

In certain applications, we may apply such a linear filter to a selected region rather than the entire image; we then speak of region-based filtering. We may also take a slightly more sophisticated approach in which the filter itself can change depending on the distribution of pixel values in the neighbourhood, a process termed adaptive filtering (e.g. see adaptive thresholding, discussed in Section 3.4.2).

Filtering at the boundaries of images also poses challenges. It is reasonable to ask what we should do when a target pixel lies close to the image boundary such that the convolution kernel overlaps the edge of the image. In general, there are three main approaches for dealing with this situation:

(1) Simply leave unchanged those target pixels which are located within this boundary region.

(2) Perform filtering on only those pixels which lie within the boundary (and adjust the filter operation accordingly).

(3) 'Fill in' in the missing pixels within the filter operation by mirroring values over the boundary.

Resulting undesirable edge artefacts are generally difficult to overcome and, in general, (2) or (3) is the preferred method. In certain instances, it is acceptable to 'crop' the image – meaning that we extract only a reduced-size image in which any edge pixels which have not been adequately filtered are removed entirely.

In Matlab, linear convolution filtering can be performed as in Example 4.2.

Example 4.2

Matlab code	**What is happening?**
A=imread('peppers.png');	%Read in image
subplot(1,2,1), imshow(A);	%Display image
k=fspecial('motion', 50, 54);	%Create a motion blur convolution kernel
B=imfilter(A, k, 'symmetric');	%Apply using symmetric mirroring at edges
subplot(1,2,2), imshow(B);	%Display result image B

Comments

Here we specify the ***fspecial()*** function to construct a kernel that will mimic the effect of motion blur (of specified length and angle) onto the image. Option 3) from our earlier discussion is used to deal with image edges during filtering. You may wish to investigate the use of other kernel filters that can be generated with the ***fspecial()*** function and the edge region filtering options available with the ***imfilter()*** function. Type *doc imfilter* at the Matlab prompt for details). How do they effect the image filtering result?

4.3.1 Nonlinear spatial filtering

Nonlinear spatial filters can easily be devised that operate through exactly the same basic mechanism as we described above for the linear case. The kernel mask slides over the image in the same way as the linear case, the only difference being that the filtered value will be the result of some nonlinear operation on the neighbourhood pixels. For example, employing the same notation as before, we could define a quadratic filter:

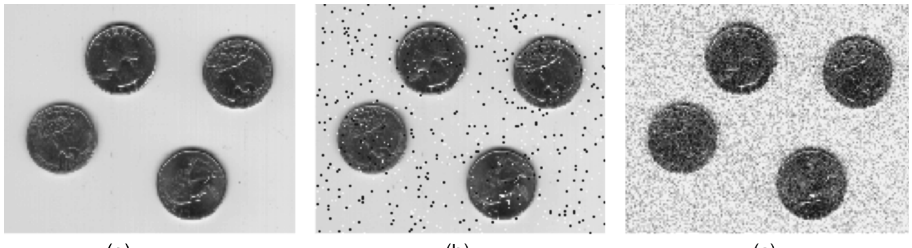
$$f_i = \sum_{k=1}^{N} w_{k1} I_k^2(i) + w_{k2} I_k(i) + w_{k3} \tag{4.3}$$

In this case, the action of the filter will be defined by the three weights which specify the contribution of the second, first- and zeroth-order terms. Nonlinear filters of this kind are not common in image processing. Much more important are order (or statistical) filters (discussed shortly), which operate by ranking the pixels in the specified neighbourhood and replacing the target pixel by the value corresponding to a chosen rank. In this case, we cannot write down a kernel and an equation of the form of our linear convolution is not applicable. In the following sections, we will present and discuss some of the more important examples of both linear and nonlinear spatial filters.

4.4 Filtering for noise removal

One of the primary uses of both linear and nonlinear filtering in image enhancement is for noise removal. We will now investigate the application of a number of different filters for removing typical noise, such as additive 'salt and pepper' and Gaussian noise (first introduced in Section 2.3.3). However, we first need to consider generation of some example images with noise added so that we can compare the effectiveness of different approaches to noise removal.

In Matlab, this can be achieved as in Example 4.3. The results of the noise addition from Example 4.3 are shown in Figure 4.3. These images will form the basis for our comparison of noise removal filters in the following sections of this chapter.



(a) (b) (c)

Figure 4.3 (a) Original image with (b) 'salt and pepper' noise and (c) Gaussian noise added

Example 4.3

Matlab code	What is happening?
I=imread('eight.tif');	%Read in image
subplot(1,3,1), imshow(I);	%Display image
Isp=imnoise(I,'salt & pepper',0.03);	%Add 3% (0.03) salt and pepper noise
subplot(1,3,2), imshow(Isp);	%Display result image Isp
Ig=imnoise(I,'gaussian',0.02);	%Add Gaussian noise (with 0.02 variance)
subplot(1,3,3), imshow(Ig);	%Display result image Ig

Comments

Here we use the ***imnoise()*** function to add both 'salt and pepper' noise and Gaussian noise to the input image. The strength is specified using the percentage density and variance (with zero mean) respectively. The reader may wish to experiment with the effects of changing these noise parameters and also explore the other noise effects available using this function (type *doc imnoise* at the Matlab prompt for details).

4.4.1 Mean filtering

The mean filter is perhaps the simplest linear filter and operates by giving equal weight w_K to all pixels in the neighbourhood. A weight of $W_K = 1/(NM)$ is used for an $N \times M$ neighbourhood and has the effect of smoothing the image, replacing every pixel in the output image with the mean value from its $N \times M$ neighbourhood. This weighting scheme guarantees that the weights in the kernel sum to one over any given neighbourhood size. Mean filters can be used as a method to suppress noise in an image (although the median filter which we will discuss shortly usually does a better job). Another common use is as a preliminary processing step to smooth the image in order that some subsequent processing operation will be more effective.

In Matlab, the mean filter can be applied as in Example 4.4 with the results as shown in Figure 4.4.

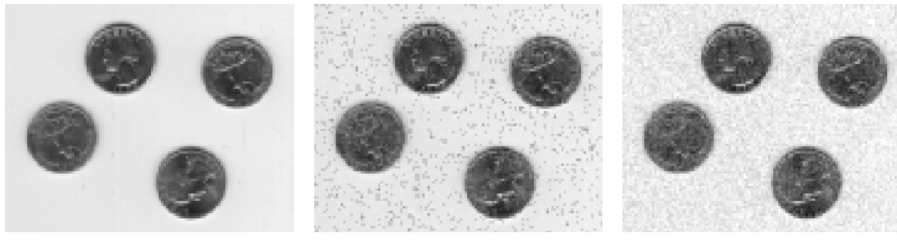


Figure 4.4 Mean filter (3×3) applied to the (a) original, (b) 'salt and pepper' noise and (c) Gaussian noise images of Figure 4.3

Example 4.4

Matlab code	What is happening?
k=ones(3,3)/9;	%Define mean filter
I_m=imfilter(I,k);	%Apply to original image
Isp_m=imfilter(Isp,k);	%Apply to salt and pepper image
Ig_m=imfilter(Ig,k);	%Apply to Gaussian image
subplot(1,3,1), imshow(I_m);	%Display result image
subplot(1,3,2), imshow(Isp_m);	%Display result image
subplot(1,3,3), imshow(Ig_m);	%Display result image

Comments

Here we define a 3×3 mean filter and apply it to the three images generated in Example 4.3 and shown in Figure 4.3. Experiment with using larger neighbourhood sizes for this filter. What effect does it have on the resulting image?

We can see that the mean filtering is reasonably effective at removing the Gaussian noise (Figure 4.4c), but at the expense of a loss of high-frequency image detail (i.e. edges). Although a significant portion of the Gaussian noise has been removed (compared with Figure 4.3c), it is still visible within the image. Larger kernel sizes will further suppress the Gaussian noise but will result in further degradation of image quality. It is also apparent that mean filtering is not effective for the removal of 'salt and pepper' noise (Figure 4.4b). In this case, the large deviation of the noise values from typical values in the neighbourhood means that they perturb the average value significantly and noise is still very apparent in the filtered result. In the case of 'salt and pepper' noise, the noisy high/low pixel values thus act as outliers in the distribution. For this reason, 'salt and pepper' noise is best dealt with using a measure that is robust to statistical outliers (e.g. a median filter, Section 4.4.2).

In summary, the main drawbacks of mean filtering are (a) it is not robust to large noise deviations in the image (outliers) and (b) when the mean filter straddles an edge in the image it will cause blurring. For this latter reason, the mean filter can also be used as a general low-pass filter. A common variation on the filter, which can be partially effective in preserving edge details, is to introduce a threshold and only replace the current pixel value with the mean of its neighbourhood if the magnitude of the change in pixel value lies below this threshold.

4.4.2 Median filtering

Another commonly used filter is the median filter. Median filtering overcomes the main limitations of the mean filter, albeit at the expense of greater computational cost. As each pixel is addressed, it is replaced by the statistical median of its $N \times M$ neighbourhood rather than the mean. The median filter is superior to the mean filter in that it is better at preserving

sharp high-frequency detail (i.e. edges) whilst also eliminating noise, especially isolated noise spikes (such as 'salt and pepper' noise).

The median m of a set of numbers is that number for which half of the numbers are less than m and half are greater; it is the midpoint of the sorted distribution of values. As the median is a pixel value drawn from the pixel neighbourhood itself, it is more robust to outliers and does not create a new unrealistic pixel value. This helps in preventing edge blurring and loss of image detail.

By definition, the median operator requires an ordering of the values in the pixel neighbourhood at every pixel location. This increases the computational requirement of the median operator.

Median filtering can be carried out in Matlab as in Example 4.5. The results of Example 4.5 are shown in Figure 4.5, where we can now see the effectiveness of median filtering on the two types of noise ('salt and pepper' and Gaussian) introduced to the images in Example 4.3/Figure 4.3.

Example 4.5

Matlab code	**What is happening?**
I_m=medfilt2(I,[3 3]);	%Apply to original image
Isp_m=medfilt2(Isp,[3 3]);	%Apply to salt and pepper image
Ig_m=medfilt2(Ig,[3 3]);	%Apply to Gaussian image
subplot(1,3,1), imshow(I_m);	%Display result image
subplot(1,3,2), imshow(Isp_m);	%Display result image
subplot(1,3,3), imshow(Ig_m);	%Display result image

Comments

Here we define a 3×3 median filter ***medfilt2()*** and apply it to the three images generated in Example 4.3 and shown in Figure 4.3. Experiment with using larger neighbourhood sizes for this filter and the effect it has on the resulting image.

(a) (b) (c)

Figure 4.5 Median filter (3×3) applied to the (a) original, (b) 'salt and pepper' noise and (c) Gaussian noise images of Figure 4.3

In this result (Figure 4.5c), we again see the removal of some Gaussian noise at the expense of a slight degradation in image quality. By contrast, the median filter is very good at removing 'salt and pepper'-type noise (Figure 4.5b), where we see the removal of this high/low impulse-type noise with minimal degradation or loss of detail in the image. This is a key advantage of median filtering.

4.4.3 Rank filtering

The median filter is really just a special case of a generalized order (or rank) filter. The general order filter is a nonlinear filter comprising the following common steps:

(1) Define the neighbourhood of the target pixel ($N \times N$).

(2) Rank them in ascending order (first is lowest value, $(N \times N)^{\text{th}}$ is highest value).

(3) Choose the order of the filter (from 1 to N).

(4) Set the filtered value to be equal to the value of the chosen rank pixel.

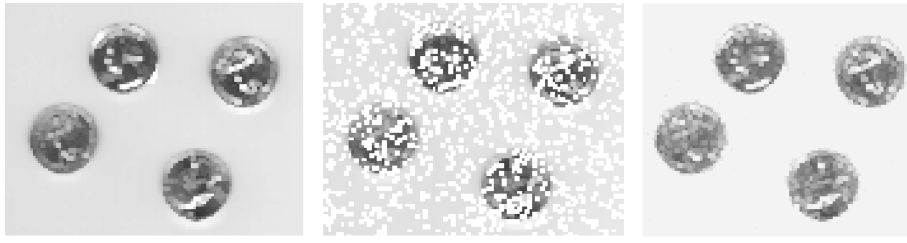
Order filters which select the maximum and minimum values in the defined neighbourhood are (unsurprisingly) called maximum and minimum filters. We can use the Matlab order-filter function as shown in Example 4.6. The results of Example 4.6 are shown in Figure 4.6, where we can now see the result of maximum filtering on the two types of noise ('salt and pepper' and Gaussian) introduced to the images in Figure 4.3. Notably, the Gaussian noise has been largely removed (Figure 4.6c), but at the expense of image detail quality (notably the lightening of the image background). The nature of the 'salt and pepper'-type noise causes its high values to be amplified by the use of a maximum filter.

Example 4.6

Matlab code	**What is happening?**
I_m=ordfilt2(I,25,ones(5,5));	%Apply to original image
Isp_m=ordfilt2(Isp,25,ones(5,5));	%Apply to salt and pepper image
Ig_m=ordfilt2(Ig,25,ones(5,5));	%Apply to Gaussian image
subplot(1,3,1), imshow(I_m); %	%Display result image
subplot(1,3,2), imshow(Isp_m);	%Display result image
subplot(1,3,3), imshow(Ig_m);	%Display result image

Comments

Here we define a 5×5 max filter and apply it to the three images generated in Example 4.3 and shown in Figure 4.3. Experiment with using larger neighbourhood sizes for this filter, varying the rank of the filter (second parameter of ***ordfilt2()*** function) and the effect it has on the resulting image.



(a) (b) (c)

Figure 4.6 Order filtering (max, order = 25, 5 × 5) applied to the (a) original, (b) 'salt and pepper' noise and (c) Gaussian noise images of Figure 4.3

A variation on simple order filtering is *conservative smoothing*, in which a given pixel is compared with the maximum and minimum values (excluding itself) in the surrounding $N \times N$ neighbourhood and is replaced only if it lies outside of that range. If the current pixel value is greater than the maximum of its neighbours, then it is replaced by the maximum. Similarly, if it is less than the minimum, then it is replaced by the minimum.

4.4.4 Gaussian filtering

The Gaussian filter is a very important one both for theoretical and practical reasons. Here, we filter the image using a discrete kernel derived from a radially symmetric form of the continuous 2-D Gaussian function defined as follows:

$$f(x,y) = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) \tag{4.4}$$

Discrete approximations to this continuous function are specified using two free parameters:

(1) the desired size of the kernel (as an $N \times N$ filter mask);

(2) the value of σ, the standard deviation of the Gaussian function.

As is always the case with linear convolution filters (Section 4.3), there is a trade-off between the accurate sampling of the function and the computational time required to implement it. Some examples of discrete Gaussian filters, with varying kernel and standard deviation sizes, are shown in Figure 4.7.

Applying the Gaussian filter has the effect of smoothing the image, but it is used in a way that is somewhat different to the mean filter (Section 4.4.1). First, the degree of smoothing is controlled by the choice of the standard deviation parameter σ, not by the absolute value of the kernel size (which is the case with the mean filter). Second, the Gaussian function has a rather special property, namely that its Fourier transform is also a Gaussian function, which makes it very convenient for the frequency-domain analysis of filters (Chapter 5).

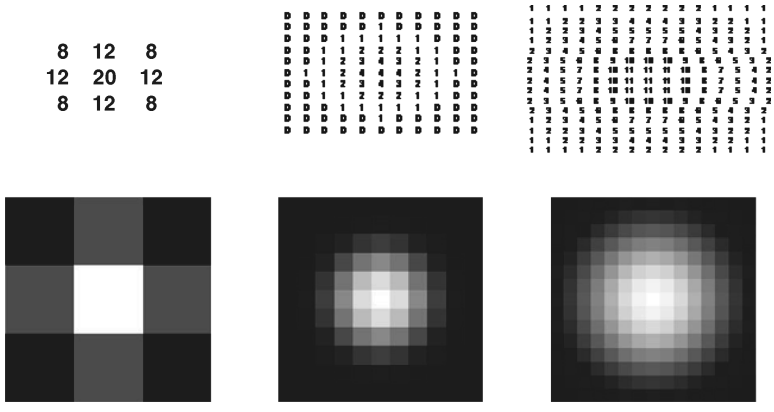


Figure 4.7 Gaussian filter kernels $3 \times 3\sigma = 1$, 11×11 $\sigma = 2$ and 21×21 $\sigma = 4$ (The numerical values shown are unnormalised)

A Gaussian function with a large value of σ is an example of a so-called low-pass filter in which the high spatial frequency content (i.e. sharp edge features) of an image is suppressed. To understand this properly requires a background in the Fourier transform and frequency-domain analysis, subjects that are developed in Chapter 5.

We can apply the Gaussian filter in Matlab as in Example 4.7. The results of Example 4.7 are shown in Figure 4.8. In all cases, the smoothing effect of the filter degrades high-frequency (edge) detail as expected (e.g. Figure 4.8a), but it also removes to some degree the noise present in both Figure 4.8b and c.

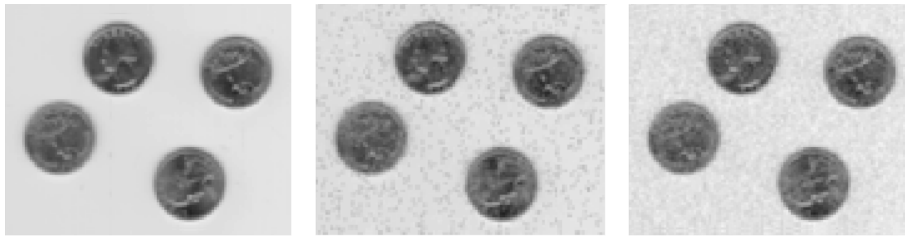
Example 4.7

Matlab code	**What is happening?**
k=fspecial('gaussian', [5 5], 2);	%Define Gaussian filter
I_g=imfilter(I,k);	%Apply to original image
Isp_g=imfilter(Isp,k);	%Apply to salt and pepper image
Ig_g=imfilter(Ig,k);	%Apply to Gaussian image
subplot(1,3,1), imshow(I_g);	%Display result image
subplot(1,3,2), imshow(Isp_g);	%Display result image
subplot(1,3,3), imshow(Ig_g);	%Display result image

Comments

Here we define a 5×5 Gaussian filter kernel with $\sigma = 2$ using the Matlab ***fspecial()*** function and apply it to the three images generated in Example 4.3 and shown in Figure 4.3. The reader can experiment by trying different kernel sizes and different σ values to understand the effect it has on the resulting image.

Gaussian smoothing (or filtering) commonly forms the first stage of an edge-detection algorithm (e.g. the Canny edge detector, discussed in Chapter 10), where it is used as a means of noise suppression.



(a) (b) (c)

Figure 4.8 Gaussian filtering (5×5 with $\sigma = 2$) applied to the (a) original, (b) 'salt and pepper' noise and (c) Gaussian noise images of Figure 4.3

4.5 Filtering for edge detection

In addition to noise removal, the other two main uses of image filtering are for (a) feature extraction and (b) feature enhancement. We will next look at the use of image filtering in both of these areas through the detection of edges within images. An edge can be considered as a discontinuity or gradient within the image. As a result, the consideration of derivative filters is central to edge detection in image processing.

4.5.1 Derivative filters for discontinuities

The mean or averaging filter sums the pixels over the specified neighbourhood and, as we have seen, this has the effect of smoothing or blurring the image. In effect, this is just integration in discrete form. By contrast, derivative filters can be used for the detection of discontinuities in an image and they play a central role in sharpening an image (i.e. enhancing fine detail). As their name implies, derivative filters are designed to respond (i.e. return significant values) at points of discontinuity in the image and to give no response in perfectly smooth regions of the image, i.e. they detect *edges*.

One of the most important aspects of the human visual system is the way in which it appears to make use of the outlines or edges of objects for recognition and the perception of distance and orientation. This feature has led to one theory for the human visual system based on the idea that the visual cortex contains a complex of feature detectors that are tuned to the edges and segments of various widths and orientations. Edge features, therefore, can play an important role in the analysis of the image.

Edge detection is basically a method of segmenting an image into regions based on discontinuity, i.e. it allows the user to observe those features of an image where there is a more or less abrupt change in grey level or texture, indicating the end of one region in the image and the beginning of another. Enhancing (or amplifying) the presence of these discontinuities in the image allows us to improve the perceived image quality under certain conditions. However, like other methods of image analysis, edge detection is sensitive to noise.

Edge detection makes use of differential operators to detect changes in the gradients of the grey or colour levels in the image. Edge detection is divided into two main categories:

Table 4.1 Derivative operators: their formal (continuous) definitions and corresponding discrete approximations

2-D derivative measure	Continuous case	Discrete case
$\frac{\partial f}{\partial x}$	$\lim_{\Delta x \to 0} \frac{f(x+\Delta x, y) - f(x,y)}{\Delta x}$	$f(x+1,y) - f(x,y)$
$\frac{\partial f}{\partial y}$	$\lim_{\Delta y \to 0} \frac{f(x, y+\Delta y) - f(x,y)}{\Delta y}$	$f(x,y+1) - f(x,y)$
$\nabla f(x,y)$	$\left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$	$[f(x+1,y) - f(x,y), f(x,y+1) - f(x,y)]$
$\frac{\partial^2 f}{\partial x^2}$	$\lim_{\Delta x \to 0} \frac{(\partial f/\partial x)(x+\Delta x, y) - (\partial f/\partial x) f(x,y)}{\Delta x}$	$f(x+1,y)$-$2f(x,y) + f(x$-$1,y)$
$\frac{\partial^2 f}{\partial y^2}$	$\lim_{\Delta y \to 0} \frac{(\partial f/\partial x)(x, y+\Delta y) - (\partial f/\partial x)(x,y)}{\Delta y}$	$f(x,y+1) - 2f(x,y) + f(x,y-1)$
$\nabla^2 f(x,y)$	$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$	$f(x+1,y) + f(x-1,y) - 4f(x,y) + f(x,y+1) + f(x,y-1)$

first-order edge detection and second-order edge detection. As their names suggest, first-order edge detection is based on the use of first-order image derivatives, whereas second-order edge detection is based on the use of second-order image derivatives (in particular the Laplacian). Table 4.1 gives the formal definitions of these derivative quantities in both continuous and corresponding discrete forms for a 2-D image $f(x,y)$.

Before we discuss the implementation of these operators, we note two things:

(1) Differentiation is a linear operation and a discrete approximation of a derivative filter can thus be implemented by the kernel method described in Section 4.3. From the discrete approximations given in Table 4.1, we must, therefore, devise appropriate filter kernels to represent each of the derivative operators (see Section 4.5.2).

(2) A very important condition we must impose on such a filter kernel is that its response be zero in completely smooth regions. This condition can be enforced by ensuring that the weights in the kernel mask sum to zero.

Although they are relatively trivial to implement, the discrete representations given in Table 4.1 in kernel form are not generally the filter kernels of choice in practice. This is because the detection of edges (which is the main application of derivative filters) is generally assisted by an initial stage of (most often Gaussian) smoothing to suppress noise. Such noise might otherwise elicit a large response from the edge-detector kernel and dominate the true edges in the image. The smoothing operation and the edge-response operator can actually be combined into a single

kernel, an example of which we will see in Section 4.6). More generally, noise suppression is a key part of more advanced edge-detection approaches, such as the Canny method (see Chapter 10).

Although a conceptually simple task, effective and robust edge detection is crucial in so many applications that it continues to be a subject of considerable research activity. Here, we will examine some basic edge-detection filters derived directly from the discrete derivatives.

4.5.2 First-order edge detection

A number of filter kernels have been proposed which approximate the first derivative of the image gradient. Three of the most common (their names being taken from their original authors/designers in the early image-processing literature) are shown in Figure 4.9, where we see the Roberts, Prewitt and Sobel edge-detector filter kernels. All three are implemented as a combination of two kernels: one for the x-derivative and one for the y-derivative (Figure 4.9).

The simple 2×2 Roberts operators (commonly known as the Roberts cross) were one of the earliest methods employed to detect edges. The Roberts cross calculates a simple, efficient, 2-D spatial gradient measurement on an image highlighting regions corresponding to edges. The Roberts operator is implemented using two convolution masks/kernels, each designed to respond maximally to edges running at $\pm 45^\circ$ to the pixel grid, (Figure 4.9 (left)) which return the image x-derivative and y-derivative, G_x and G_y respectively. The magnitude $|G|$ and orientation θ of the image gradient are thus given by:

$$\begin{aligned} |G| &= \sqrt{G_x^2 + G_y^2} \\ \theta &= \tan^{-1}\left(\frac{G_y}{G_x}\right) + \frac{1}{4}\pi \end{aligned} \tag{4.5}$$

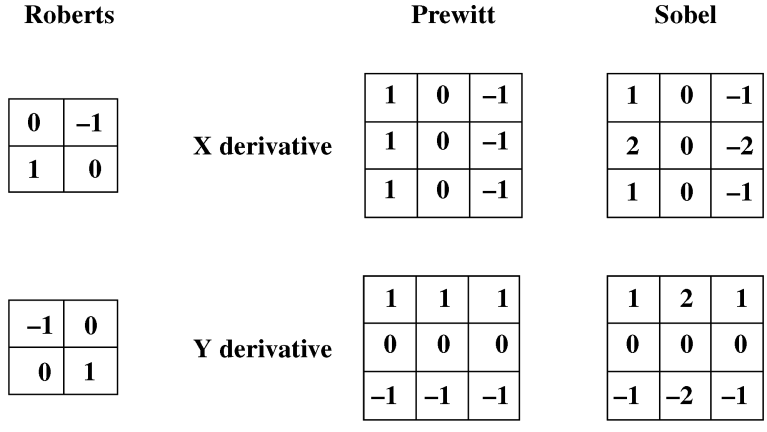


Figure 4.9 First-order edge-detection filters

This gives an orientation $\theta = 0$ for a vertical edge which is darker on the left side in the image. For speed of computation, however, $|G|$ is often approximated as just the sum of the magnitudes of the x-derivative and y-derivative, G_x and G_y.

The Roberts cross operator is fast to compute (due to the minimal size of the kernels), but it is very sensitive to noise. The Prewitt and Sobel edge detectors overcome many of its limitations but use slightly more complex convolution masks (Figure 4.9 (centre and right)).

The Prewitt/Sobel kernels are generally preferred to the Roberts approach because the gradient is not shifted by half a pixel in both directions and extension to larger sizes (for filter neighbourhoods greater than 3×3) is not readily possible with the Roberts operators. The key difference between the Sobel and Prewitt operators is that the Sobel kernel implements differentiation in one direction and (approximate) Gaussian averaging in the other (see Gaussian kernels, Figure 4.7). The advantage of this is that it smoothes the edge region, reducing the likelihood that noisy or isolated pixels will dominate the filter response.

Filtering using these kernels can be achieved in Matlab as shown in Example 4.8. The edge filter output of Example 4.8 is shown in Figure 4.10, where we can see the different responses of the three Roberts, Prewitt and Sobel filters to a given sample image. As expected, stronger and highly similar edge responses are obtained from the more sophisticated Prewitt and Sobel approaches (Figure 4.10 (bottom)). The Roberts operator is notably susceptible to image noise, resulting in a noisy and less distinct edge magnitude response (Figure 4.10 (top right)).

Example 4.8

Matlab code	**What is happening?**
I=imread('circuit.tif');	%Read in image
IEr = edge(I,'roberts');	%Roberts edges
IEp = edge(I,'prewitt');	%Prewitt edges
IEs = edge(I,'sobel');	%Sobel edges
subplot(2,2,1), imshow(I);	%Display image
subplot(2,2,2), imshow(IEr);	%Display image
subplot(2,2,3), imshow(IEp);	%Display image
subplot(2,2,4), imshow(IEs);	%Display image

Comments

Here we use the Matlab ***edge()*** function to apply the Roberts, Prewitt and Sobel edge detectors. As an extension, this function also facilitates the use of an additional third, threshold parameter in the form ***edge(Image, 'filter name', threshold)***. This exploits the concept of thresholding to select a subset of edge filter responses based on the magnitude of the filter response. The ***edge()*** function can also be used to return the individual G_x and G_y components of a given filter mask and automatically select a magnitude threshold (in a similar manner to Example 3.15; see *doc edge* in Matlab command prompt). Experiment with these parameters and the effects that can be achieved on the example images. You may also wish to investigate the use of the Matlab ***tic()***/***toc()*** functions for timing different edge-filtering operations.

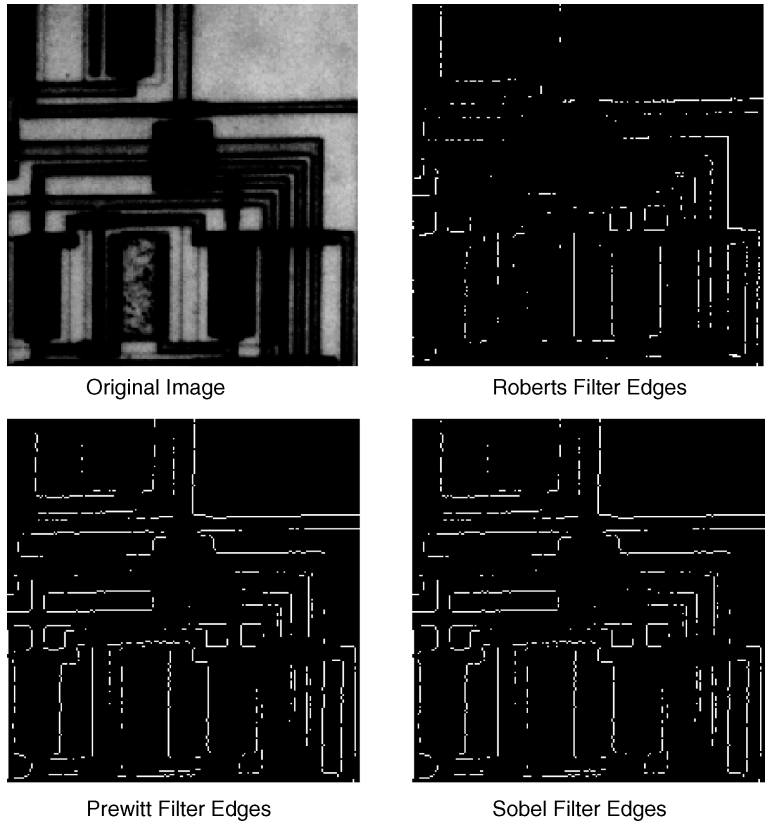


Figure 4.10 Roberts, Prewitt and Sobel edge-magnitude responses

4.5.2.1 Linearly separable filtering

The Sobel and Prewitt filters are examples of linearly separable filters. This means that the filter kernel can be expressed as the matrix product of a column vector with a row vector. Thus, the filter kernels shown in Figure 4.9 can be expressed as follows:

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \tag{4.6}$$

An important consequence of this is that the 2-D filtering process can actually be carried out by two sequential 1-D filtering operations. Thus, the rows of the image are first filtered with the 1-D row filter and the resulting filtered image is then filtered column-wise by the 1-D column filter. This effects a computational saving in terms of reducing the amount of arithmetic operations required for a given convolution with a filter kernel. The saving is modest in the 3×3 case (a reduction to six multiplications/additions compared with nine for the 2-D version), but is considerably greater if we are considering larger kernel sizes. In general, linear separable filters result in a saving of order $2N$ operations as opposed to order N^2 for nonseparable, 2-D convolution.

4.5.3 Second-order edge detection

In general, first-order edge filters are not commonly used as a means of image enhancement. Rather, their main use is in the process of edge detection as a step in image segmentation procedures. Moreover, as we shall see in our discussion of the Canny method in Chapter 10, although the application of a derivative operator is a vital step, there is actually considerably more to robust edge detection than the simple application of a derivative kernel. A much more common means of image enhancement is through the use of a second-order derivative operator:- the Laplacian.

4.5.3.1 Laplacian edge detection

A very popular second-order derivative operator is the Laplacian:

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{4.7}$$

The discrete form is given from Table 4.1 as:

$$\nabla^2 f = f(x+1,y) + f(x-1,y) - 4f(x,y) + f(x,y+1) + f(x,y-1) \tag{4.8}$$

This can easily be implemented in a 3×3 kernel filter, as shown in Figure 4.11A. However, if we explore an image applying this operator locally, then we expect the response to be greatest (as the Laplacian is a second-order derivative operator) at those points in the image where the local gradient changes most rapidly. One of the potential shortcomings of applying the mask in the form given by Figure 4.11A is the relative insensitivity to features lying in approximately diagonal directions with respect to the image axes. If we imagine rotating the axes by 45° and superimposing the rotated Laplacian on the original, then we can construct a filter that is invariant under multiple rotations of 45° (Figure 4.11B).

Figure 4.12 compares the response of the first-order Sobel and second-order Laplacian derivative filters. Note how the first-order gradient operator tends to produce 'thick edges', whereas the Laplacian filter tends to produce finer edges in response to the change in gradient rather than the image gradient itself. The Laplacian operator can be applied in Matlab as in Example 4.9.

The second-order derivative property that allows the Laplacian to produce a fine edge response corresponding to a change in gradient, rather than the less isolated response of the first-order edge filters, makes it suitable as the first stage of digital edge enhancement.

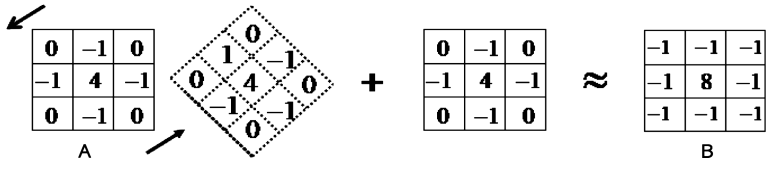


Figure 4.11 Construction of the Laplacian discrete kernel

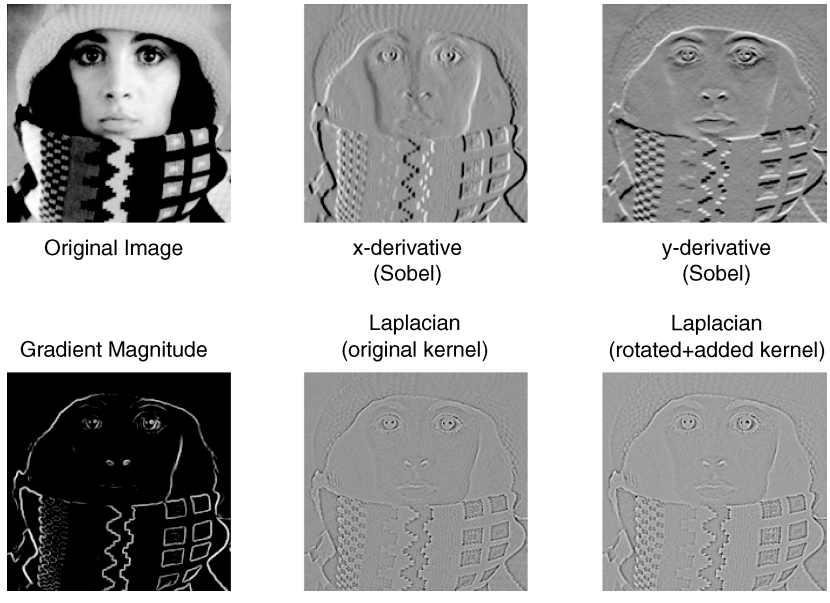


Figure 4.12 Comparison of first-order derivative (Sobel) and second-order (Laplacian) filters

However, as the Laplacian kernels approximate a second derivative over the image they are in fact very sensitive to noise.

Example 4.9

Matlab code	**What is happening?**
I=rgb2gray(imread('peppers.png'));	%Read in image (in grey scale)
k=fspecial('laplacian');	%Create Laplacian filter
IEl=imfilter(double(I),k,'symmetric');	%Laplacian edges
subplot(1,2,1), imagesc(I);	%Display image
subplot(1,2,2), imagesc(IEl);	%Display image
colormap('gray');	

Comments

Here we first construct the Laplacian filter (in a similar manner to Example 4.2) and then apply it to the image using the Matlab ***imfliter()*** function. Note the inline use of the ***rgb2gray()*** function to load the (colour) example image as grey scale. In addition, we perform the Laplacian operation on a floating-point version of the input image (function ***double()***) and, as the Laplacian operator returns both positive and negative values, use function ***imagesc()*** and ***colormap()*** to correctly scale and display the image as shown in Figure 4.12.

4.5.3.2 Laplacian of Gaussian

To counter this high noise sensitivity of the Laplacian filter, the standard Laplacian kernel (Figure 4.11) is commonly combined with the Gaussian kernel (Figure 4.7) to produce

a robust filtering method. These two kernels could be applied sequentially to the image as two separate convolution operations – first smoothing with the Gaussian kernel and then with the Laplacian. However, as convolution is associative (Section 4.3), we can combine the kernels by convolving the Gaussian smoothing operator with the Laplacian operator to produce a single kernel: the Laplacian of Gaussian (LoG) filter. This single kernel is then applied to the image in a single pass. This offers a significant computational saving by approximately halving the calculations required.

The response of the filter will be zero in areas of uniform image intensity, whilst it will be nonzero in an area of transition. At a given edge, the operator will return a positive response on the darker side and negative on the lighter side.

We can see this effect in Figure 4.12 and the results of Example 4.9. If we wish to apply the LoG operator in Matlab, then we can do so by replacing the 'laplacian' parameter of the ***fspecial()*** function with 'log'. The result should be a smoothed version of the result in Example 4.9. Additional input parameters to the ***fspecial()*** function allow the level of smoothing to be controlled by varying the width of the Gaussian.

4.5.3.3 Zero-crossing detector

The zero-crossing property of the Laplacian (and LoG) also permits another method of edge detection: the zero-crossing method. We use a zero-crossing detector to locate pixels at which the value of the Laplacian passes through zero (i.e. points where the Laplacian changes sign). This occurs at 'edges' in the image where the intensity of the image changes rapidly (or in areas of intensity change due to noise). It is best to think of the zero-crossing detector as a type of feature detector rather than as a specific edge detector. The output from the zero-crossing detector is usually a binary image with single-pixel thickness lines showing the positions of the zero-crossing points.

The starting point for the zero-crossing detector is an image which has been filtered using the LoG filter (to overcome the effects of noise). The zero crossings that result are strongly influenced by the size of the Gaussian used for the smoothing stage of this operator. As the smoothing is increased, then fewer and fewer zero-crossing contours will be found, and those that do remain will correspond to features of larger and larger scale in the image.

We can use a zero-crossing detector with the LoG filter in Matlab as in Example 4.10. Figure 4.13 shows edge transitions detected using the zero-crossings concept applied to the LoG filter (from Example 4.10). The effect of noise on the second derivative despite the use of Gaussian smoothing is evident.

Example 4.10

Matlab code	**What is happening?**
I=rgb2gray(imread('peppers.png'));	%Read in image (in grey scale)
k=fspecial('log', [10 10], 3.0);	%Create LoG filter
IEzc = edge(I, 'zerocross', [], k);	%Zero-crossing edges (auto-thresholded)
subplot(1,2,1), imshow(I);	%Display image
subplot(1,2,2), imshow(IEzc);	%Display image

Comments

As is evident from Figure 4.13, the results are still quite noisy even with a broad 10×10 Gaussian ($\sigma = 3.0$) as specified in the above example. The reader may wish to experiment with different Gaussian parameters (note that the filter k can itself be visualized as an image using the ***imagesc()*** function).

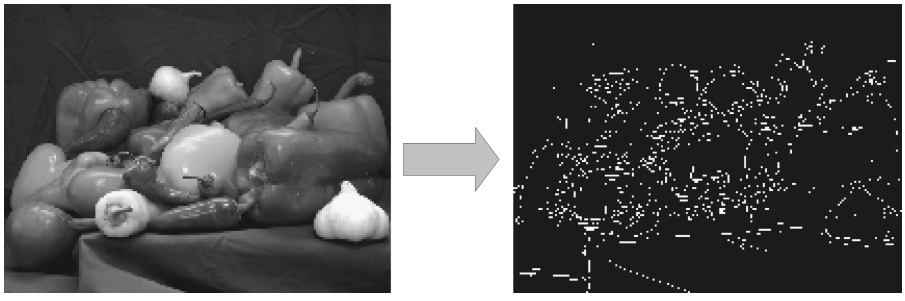


Figure 4.13 Edges detected as the zero crossings of the LOG operator

A couple of points to note with this operator are:

- In the general case, all edges detected by the zero-crossing detector are in the form of closed curves in the same way that contour lines on a map are always closed. In the Matlab implementation, if a threshold of zero is specified then this is always the case (NB: Example 4.10 uses an autoselected threshold (parameter specified as '[ ]') on which edges to keep). The only exception to this is where the curve goes off the edge of the image. This can have advantages for later processing.

- As we have seen, the LoG filter is quite susceptible to noise if the standard deviation of the smoothing Gaussian is small. One solution to this is to increase Gaussian smoothing to preserve only strong edges. An alternative is to look at the gradient of the zero crossing and only keep zero crossings where this is above a certain threshold (i.e. use the third derivative of the original image). This will tend to retain only stronger edges but as the third derivative is also highly sensitive to noise this greatly amplifies any high-frequency noise in the image.

4.6 Edge enhancement

In the final part of this chapter we look at the use of second-order edge detection (Section 4.5.3) as a method for edge enhancement (commonly known as image sharpening).

4.6.1 Laplacian edge sharpening

We have seen that the Laplacian responds only to the fine detail in the image (i.e. those image regions where the change in gradient is significant) but has a zero response to

constant regions and regions of smooth gradient in the image. If, therefore, we take the original image and add or subtract the Laplacian, then we may expect to enhance the fine detail in the image artificially. It is common practice just to subtract it from the original, truncating any values which exhibit integer overflow in the common 8-bit representation. Using the Laplacian definition of Section 4.5.3.1, we can define this as follows:

$$I_{\text{output}}(x, y) = I_{\text{in}}(x, y) - \nabla^2 I_{\text{in}}(x, y) \tag{4.9}$$

Using Matlab, Laplacian image sharpening can be achieved as in Example 4.11. The output from Example 4.11 is shown in Figure 4.14, where we can see the original image, the Laplacian 'edges' and the sharpened final output. Note that we can see the enhancement of edge contrast in this filtering result, but also an increase in image noise in the sharpened image.

Example 4.11

Matlab code	**What is happening?**
A=imread('cameraman.tif');	%Read in image
h=fspecial(laplacian, [10 0], 3.0);	%Generate 3 × 3 Laplacian filter
B=imfilter(A,h);	%Filter image with Laplacian kernel
C=imsubtract(A,B);	%Subtract Laplacian from original.
subplot(1,3,1), imshow(A);	
subplot(1,3,2), imagesc(B); axis image; axis off	%Display original, Laplacian and
subplot(1,3,3), imshow(C);	%enhanced image

Comments

In this example, because the images are not first converted to floating-point format (data type double), the Laplacian filtered image is automatically truncated into the 8-bit form. The reader may experiment by first converting both images A and B to floating point (***double()***), performing the calculation in floating-point image arithmetic and displaying such images as per Example 4.9.

Figure 4.14 Edge sharpening using the Laplacian operator

Figure 4.15 Edge sharpening using the LoG operator

In order to overcome this problem we replace the Laplacian operator in Example 4.11 with the LoG operator (Section 4.5.3.2). The reader may wish to experiment using this variation on Example 4.11, applying different-size Gaussian kernels to investigate the difference in image sharpening effects that can be achieved. The example of Figure 4.14 is again shown in Figure 4.15 using the alternative LoG operator where we can note the reduced levels of noise in the intermediate LoG edge image and the final sharpened result.

4.6.2 The unsharp mask filter

An alternative edge enhancement filter to the Laplacian-based approaches (Section 4.6.1) is the unsharp mask filter (also known as boost filtering). Unsharp filtering operates by subtracting a smoothed (or unsharp) version of an image from the original in order to emphasize or enhance the high-frequency information in the image (i.e. the edges). First of all, this operator produces an edge image from the original image using the following methodology:

$$I_{\text{edges}}(c, r) = I_{\text{original}}(c, r) - I_{\text{smoothed}}(c, r) \tag{4.10}$$

The smoothed version of the image is typically obtained by filtering the original with a mean (Section 4.4.1) or a Gaussian (Section 4.4.4) filter kernel. The resulting difference image is then added onto the original to effect some degree of sharpening:

$$I_{\text{enhanced}}(c, r) = I_{\text{original}}(c, r) + k(I_{\text{edges}}(c, r)) \tag{4.11}$$

using a given constant scaling factor k that ensures the resulting image is within the proper range and the edges are not 'oversharp' in the resulting image. Generally, $k = 0.2$–0.7 is acceptable, depending on the level of sharpening required. It is this secondary stage that gives rise to the alternative name of boost filtering.

To understand this approach to sharpening we need to consider two facts. First, the smooth, relatively unchanging regions of the original image will not be changed significantly by the smoothing filter (for example, a constant region which is already perfectly smooth will be completely unaffected by the smoothing filter, e.g. Figures 4.4 and 4.8). By contrast, edges and other regions in the image in which the intensity changes rapidly will be affected significantly. If we subtract this smoothed image I_{smoothed} from the original image I_{original},

then we get a resulting image I_{edges} with higher values (differences) in the areas affected significantly (by the smoothing) and low values in the areas where little change occurred. This broadly corresponds to a smoothed edge map of the image. The result of subtracting this smoothed image (or some multiple thereof) from the original is an image of higher value pixels in the areas of high contrast change (e.g. edges) and lower pixel values in the areas of uniformity I_{edges}. This resulting image can then be added back onto the original, using a specified scaling factor k, to enhance areas of rapid intensity change within the image whilst leaving areas of uniformity largely unchanged. It follows that the degree of enhancement will be determined by the amount of smoothing that is imposed before the subtraction takes place and the fraction of the resulting difference image which is added back to the original. Unsharp filtering is essentially a reformulation of techniques referred to as high boost – we are essentially boosting the high-frequency edge information in the image.

This filtering effect can be implemented in Matlab as shown in Example 4.12. The result of Example 4.12 is shown in Figure 4.16, where we see the original image, the edge difference image (generated by subtraction of the smoothed image from the original) and a range of enhancement results achieved with different scaling factors k. In the examples shown, improvement in edge detail (sharpness) is visible but the increasing addition of the edge image (parameter k) increases both the sharpness of strong edges and noise apparent within the image. In all cases, image pixels exceeding the 8-bit range 0–255 were truncated.

Example 4.12

Matlab code	What is happening?
A=imread('cameraman.tif');	%Read in image
Iorig=imread('cameraman.tif');	%Read in image
g=fspecial('gaussian',[5 5],1.5);	%Generate Gaussian kernel
subplot(2,3,1), imshow(Iorig);	%Display original image
Is=imfilter(Iorig,g);	%Create smoothed image by filtering
Ie=(Iorig - Is);	%Get difference image
subplot(2,3,2), imshow(Ie);	%Display unsharp difference
Iout=Iorig + (0.3).*Ie;	%Add k * difference image to original
subplot(2,3,3), imshow(Iout);	
Iout=Iorig + (0.5).*Ie;	%Add k * difference image to original
subplot(2,3,4), imshow(Iout);	
Iout=Iorig + (0.7).*Ie;	%Add k * difference image to original
subplot(2,3,5), imshow(Iout);	
Iout=Iorig + (2.0).*Ie;	%Add k * difference image to original
subplot(2,3,6), imshow(Iout);	

Comments

In this example we perform all of the operations using unsigned 8-bit images (Matlab type ***uint8***) and we compute the initial smoothed image using a 5×5 Gaussian kernel ($\sigma = 1.5$). Try changing the levels of smoothing, using an alternative filter such as the mean for the smoothing operator and varying the scaling factor k.

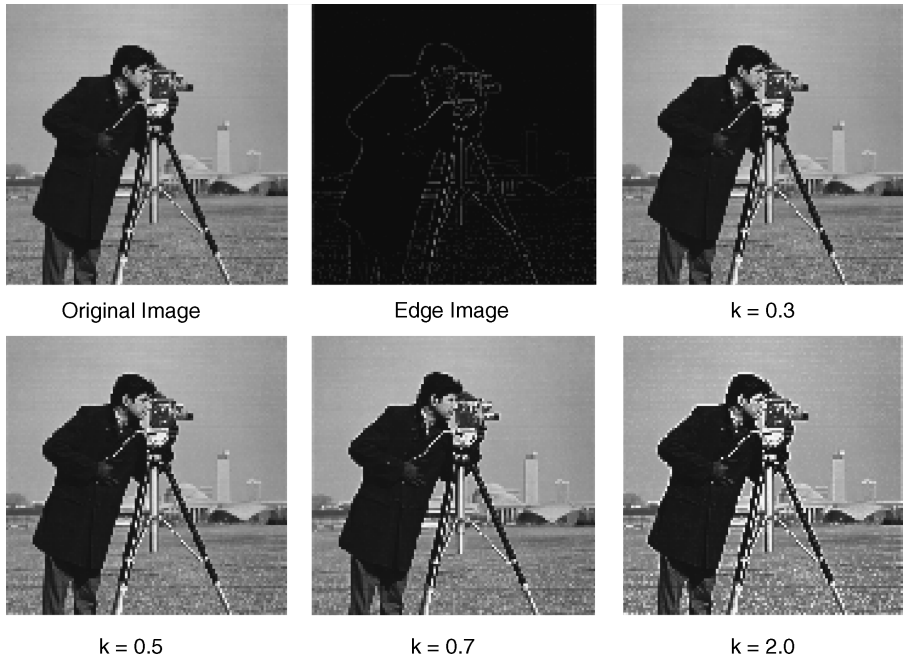


Figure 4.16 Edge sharpening using unsharp mask filter

Exercises

The following exercises are designed to reinforce and develop the concepts and Matlab examples introduced in this chapter. Additional information on all of the Matlab functions presented in this chapter and throughout these exercises is available in Matlab from the function help browser (use *doc <function name>* at the Matlab command prompt where *<function name>* is the function required).

Matlab functions: ***imnoise***, ***plot***, ***tic***, ***toc***, ***min***, ***max***, ***mean***, ***function***, ***imresize***, ***for***, ***colfilt***, ***roipoly***, ***roifilt***, ***imfilter***, ***fspecial***.

Exercise 4.1 Based on Example 4.3, experiment with the Matlab ***imnoise()*** function for adding different levels of 'salt and pepper' and Gaussian noise to images. Use the 'peppers.png' and 'eight.tif' example images as both a colour and grey-scale example to construct a series of image variables in Matlab with varying levels and types of noise (you may also wish to investigate the other noise types available from this function, type *doc imnoise* at the Matlab prompt). Based on the filtering topics presented in Section 4.4, investigate the usefulness of each of the mean, median and Gaussian filtering for removing different types and levels of image noise (see Examples 4.4, 4.5 and 4.7 for help).

Exercise 4.2 Building on Example 4.2, experiment with the use of different neighbourhood dimensions and the effect on the resulting image output. Using the plotting functions of Matlab (function ***plot()***) and the Matlab timing functions ***tic*** and ***toc***, create a plot of

neighbourhood dimension N against operation run-time for applying the ***min()***, ***max()*** and ***mean()*** functions over an example image. Does the timing increase linearly or not? Why is this?

Exercise 4.3 Building on the local pixel neighbourhood definition described Section 4.2, write a Matlab function (see Matlab help for details on functions or type *doc function* at the Matlab function) to extract the $N \times N$ pixel neighbourhood of a given specified pixel location (x, y) and copy the pixel values to a new smaller $N \times N$ image. You may want to take a look at the syntax for the Matlab for loop function (*doc for*) in the first instance.

Combine your extraction program with the Matlab ***imresize()*** function to create a region extraction and magnification program.

Exercise 4.4 The Matlab ***colfilt()*** function utilizes the highly efficient matrix operations of Matlab to perform image filtering operations on pixel neighbourhoods. Using the Matlab timing functions ***tic()***/***toc()***, time the operation performed in Example 4.2 with and without the use of this optimizing parameter. Vary the size of the neighbourhood over which the operation is performed (and the operation: ***min()***/***max()***/***mean()***). What do you notice? Is the increase in performance consistent for very small neighbourhoods and very large neighbourhoods?

Exercise 4.5 Based on Example 4.4 for performing mean filtering on a given image, record the execution time for this operation (using Matlab timing functions ***tic()***/***toc()***) over a range of different neighbourhood sizes in the range 0–25. Use Matlab's plotting facilities to present your results as a graph. What do you notice? How does the execution time scale with the increase in size of the neighbourhood? (*Hint*. To automate this task you may want to consider investigating the Matlab ***for*** loop constructor).

As an extension you could repeat the exercise for differently sized images (or a range of image sizes obtained from ***imresize()***) and plot the results. What trends do you notice?

Exercise 4.6 Repeat the first part of Exercise 4.5, but compare the differences between mean filtering (Example 4.4) and median filtering (Example 4.5). How do the trends compare? How can any differences be explained?

Exercise 4.7 A region of interest (ROI) within an image is an image sub-region (commonly rectangular in nature) over which localized image-processing operations can be performed. In Matlab an ROI can be selected interactively by first displaying an image (using ***imshow()***) and then using the ***roipoly()*** function that returns an image sub-region defined as a binary image the same size as the original (zero outside the ROI and one outside the ROI).

Investigate the use of the ***roifilt()*** function for selectively applying both Gaussian filtering (Example 4.7) and mean filtering (Example 4.4) to an isolated region of interest within one of the available example images.

You may also wish to investigate combining the ROI selection function with your answer to Exercise 4.3 to extract a given ROI for histogram equalization (Section 3.4.4) or edge-detection processing (Section 4.5) in isolation from the rest of the image.

Exercise 4.8 Several of the filtering examples in this chapter make use of the Matlab ***imfilter()*** function (Example 4.2), which itself has a parameter of how to deal with image boundaries for a given filtering operation. Select one of the filtering operations from this chapter and experiment with the effect of selecting different boundary options with the ***imfilter()*** function. Does it make a difference to the result? Does it make a difference to the amount of time the filtering operation takes to execute? (Use ***tic()***/***toc()*** for timing.)

Exercise 4.9 Implement a Matlab function (*doc function*) to perform the conservative smoothing filter operation described towards the end of Section 4.4.3. (*Hint*. You may wish to investigate the Matlab ***for*** loop constructor.) Test this filter operation on the noise examples generated in Example 4.3. How does it compare to mean or median filtering for the different types of noise? Is it slower or faster to execute? (Use ***tic()***/***toc()*** for timing.)

Exercise 4.10 Considering the Roberts and Sobel edge detectors from Example 4.8, apply edge detection to three-channel RGB images and display the results (e.g. the 'peppers.png' and 'football.jpg' images). Display the results as a three-channel colour image and as individual colour channels (one per figure).

Note how some of the edge responses relate to distinct colour channels or colours within the image. When an edge is visible in white in the edge-detected three-channel image, what does this mean? You may also wish to consider repeating this task for the HSV colour space we encountered in Chapter 1. How do the results differ in this case?

Exercise 4.11 Building upon the unsharp operator example (Example 4.12), extend the use of this operator to colour images (e.g. the 'peppers.png' and 'football.jpg' images). How do the areas of image sharpening compare with the areas of edge-detection intensity in Exercise 4.10?

The unsharp operator can also be constructed by the use of an 'unsharp' parameter to the Matlab ***fspecial()*** function for us with ***imfilter()***. Compare the use of this implementation with that of Example 4.12. Are there any differences?

For further examples and exercises see http://www.fundipbook.com