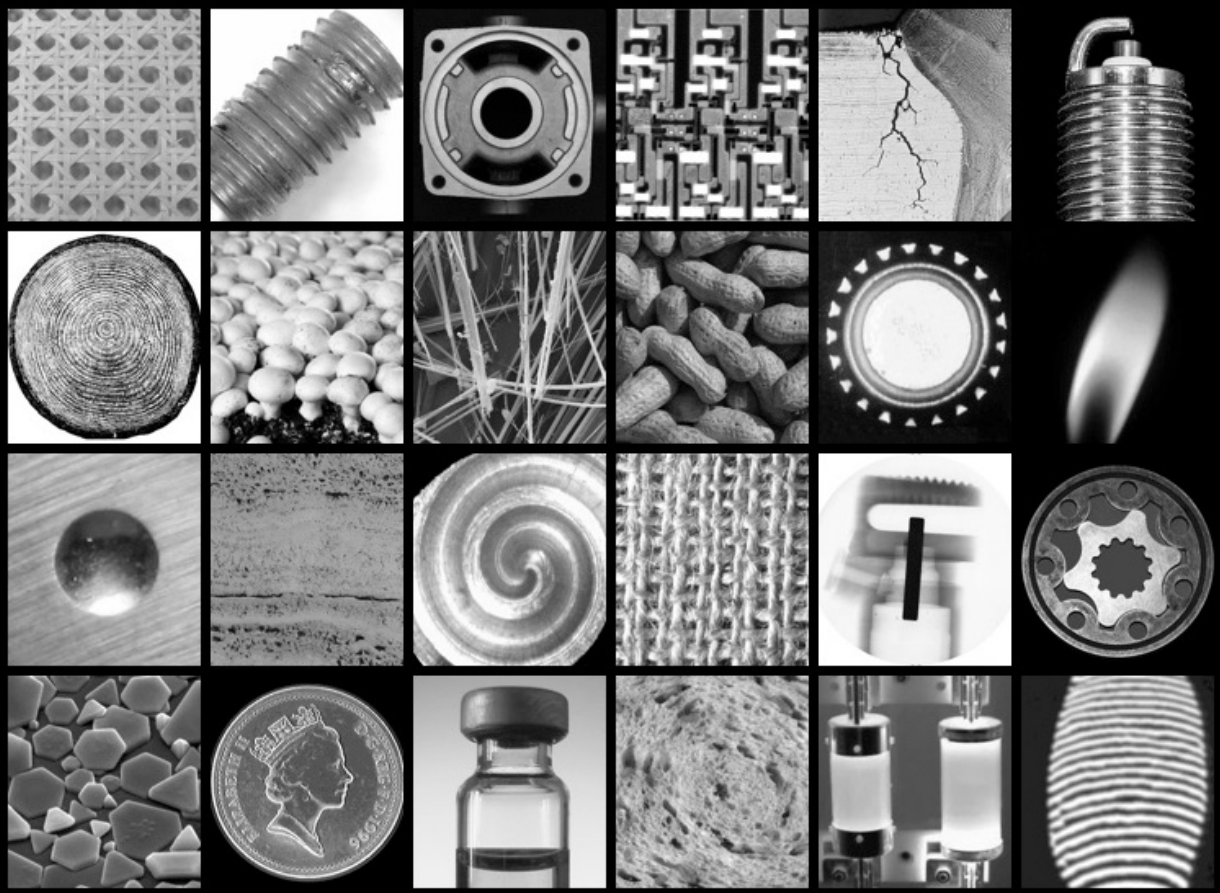
Chapter 4

256 Shades of Grey



Preamble

When studying vision in nature, we are trying to discover what data processing is taking place inside a “closed box”. There is very little opportunity for direct access into either human or animal brains. Implanted electrodes and PET scanning have provided some insight into what happens inside the brain but our knowledge is still very limited. On the other hand, to develop Machine Vision systems, we need to decide what processing capabilities to insert into “the box”. These two pursuits are quite different: studying natural vision requires analysis, while designing Machine Vision devices requires synthesis.

How do we know what to put inside “the box”? Some operations that we use to process digital images were first tried for no other reason than that they are fast and simple to perform in a computer. For example, adding (or subtracting) a fixed number to every intensity value is one obvious operation that was investigated very early. What visual effect does such an operation have? Is it useful in practice? On its own, it is not so, but when it is used in combination with other functions it is very valuable. We cannot fully appreciate its benefits until we start to build sequences of simple processing steps.

A natural first step in our journey through image processing for Machine Vision is to describe how one image can be created from another. Often, the goal is to enhance the visibility of important features, while suppressing others. An obvious example is negation, which is, of course, familiar to film photographers. Blurring, enhancing edges, pseudo-every colouring and thresholding are among the other well-known operations that we will encounter. In each of these

cases, an image is transformed into another image, with no distortion of geometry; major image features are still recognisable, sometimes made more clearly visible, sometimes less so.

Why begin with grey-scale images when colour is so much more interesting and appealing? The simple answer is that we need to take small steps first: colour image processing is builton the ideas described in this chapter. While binary images appear to be simpler than grey-scale pictures, the operations and measurements that we need to perform are conceptually very different. In practice, processing grey-scale, or colour, images almost always precedes transforming to and analysing binary images. The important features must be enhanced, identified and isolated before they can be measured.

Families of image processing operations

We can identify several families of processing operations that we can apply to grey-scale images:

1. **Monadic Operators** A single image (the *input image*) is processed to generate another image (the *output image*). Each input pixel is processed independently of all others, at the same moment. The output image is therefore the same size and shape, as the input. There is no change of size, orientation, or geometric distortion. At least some of the major image features are still identifiable. (The very point of the processing may be to suppress the visibility of other features.) For this reason, these said to be *point-by-point operators*.

2. **Dyadic Operators**. Two input images are combined to generate a third output image. Each corresponding pair of pixels is processed independently of all others, at the same moment. Again, there is no geometric distortion. These are also point-by-point operators
3. **Local operators**: Again, a single image is processed to generate another image but the value of every output pixel is derived from a (compact) group of pixels in the input image. This process is repeated for all output pixels, at the same moment. The output image is the same size and shape, as the input.
4. **Image measurements** The output is one or more numbers.
5. **Intensity histogram** (Or **Histogram**) This is a list of numbers describing the statistics of the intensity values in the image array. It can yield some very valuable measurements, which can guide further processing. The histogram is often displayed as a graph.

Point-by-point Operators

Monadic, dyadic and local operators are normally used as part of a complex processing sequence; they are rarely used on their own. It is therefore difficult, at this early stage, to justify them fully. Hence, for the moment, we will simply demonstrate the effects they produce. Sometimes, they do improve the visual appearance of an image, but they can also do the opposite. Even so, they are often be very effective tools and, for this reason, are essential components within nearly all artificial vision systems.

Monadic Operators

The operation of a monadic operator is explained in Figure 4.1. A typical pixel, such as that in the i^{th} column and j^{th} row, is transformed to produce a new value that is then stored in the same position, *[i,j]*, in the output image.The transformation by the so-called **Mapping Function**, may be achieved by calculation. For example, we may simply add, or multiply, the intensity of each pixel *[i,j]* by a constant (a fixed number). Alternatively, the mapping function may be implemented using a **Look-up Table** (**LUT**, Figure 4.2). Entries in the table need only be calculated once, which can save a lot of computation time. It also allows a general class of dynamic image-to-image transformations, based on prior measurements, to be performed. Monadic operations usually leave the image features clearly identifiable. For example, a transformed image will usually leave facial features, such as the eyes and mouth, clearly visible and the person easily recognisable.

Using monadic point-by-point operators, it is possible to enhance the visual appearance of an image, so that its brightness and/or contrast are improved. It is even possible to enhance the visibility of dark, or bright, features separately. To a limited extent, point-by-point operators are able to compensate for variable levels of lighting.

Adding, subtracting & multiplying by a constant

In our first and simplest example of image processing, we add the same number to each value in the intensity array. To illustrate this, we will transform the following array, which represents just a small part of a very much larger digital image:

122	124	125	127	128
131	133	134	135	158
140	151	156	187	189
201	210	215	225	228
231	233	236	237	239

After adding 10 to each element, the result is

Add 10

132	134	135	137	138
141	143	144	145	168
150	161	166	197	199
211	220	225	235	238
241	243	246	247	249

As a result of performing this operation, the whole image becomes brighter. Notice that black becomes dark grey. Now, suppose that we had tried to add 20 to the entries in our original array, instead. Some of the values will become greater than 255, which is not allowed. To avoid such non-sensical intensity values, we place limits on all elements in the array. Any entry greater than 255 is replaced by 255. The revised table entries that have been "**hard limited**" in this way are printed in red in the array below.

142	144	145	147	148
151	153	154	155	178
160	171	176	207	209
221	230	235	245	248
251	253	255	255	255

Add 20

Subtracting a constant value from each entry makes the whole picture darker. However, this can cause problems at the lower end of the intensity scale, in which case, we impose a lower hard limit of 0 (zero). In the following example we subtract 150, so there is considerable loss of information due to limiting.

Subtract 150

0	0	0	0	0
0	0	0	0	8
0	1	6	37	39
51	60	65	75	78
81	83	86	87	89

Hard limiting the minimum or maximum intensity values in an image array destroys detail. Notice that adding a constant and then subtracting the same value does not necessarily restore the original unchanged. Hard limiting occurs again, but this time at level 0 (zero).

Add 150 then subtract 150

150	150	150	150	150
150	150	150	150	150
150	151	156	187	189
201	210	215	225	228
231	233	236	237	23

The effects of image addition and subtraction are illustrated in Figures 4.3 and .4.44.

Multiplying the intensity values in the array by a fixed number is another obvious possibility. If the scaling factor is less than 1.00, the image contrast is reduced, while it is increased if the scaling factor is greater than 1.00. Fractional values in the revised image array are rounded to the nearest whole number. Multiplying intensities by a number greater than 1.00 can produce values that exceed 255. For this reason, we must again impose a hard limit. (Figure 4.5) Multiplying our little array by 1.5 we obtain

Multiply by 1.5

183	186	187	190	192
196	199	201	202	237
210	226	234	255	255
255	255	255	255	255
255	255	255	255	255

Hard limiting the values in the image array to a finite range, [0,255], may cause some detail to be lost. There is another effect that can lead to loss of information. Multiplying each intensity value by 0.8 could seemingly be reversed by multiplying the result by 1.25 (i.e. 1/0.8). However, the fact that we are only allowing whole number entries in the image array means that, in some cases, the final intensity values will differ slightly from those in the original image. This *Quantisation Effect* is not normally significant for Machine Vision. Our little image array is not altered significantly

Quantisation effects

123	124	125	128	128
131	133	134	135	158
140	151	156	188	189
201	210	215	225	228
231	233	236	238	239

Almost all readers will be familiar with image *negation*. (Figure 4.6) In this process, black is replaced by white; dark grey by light grey; light grey by dark grey; white by black. It will be convenient to express this mathematically. Consider a pixel whose intensity is *X*. For the sake of practical convenience, we will assume that *X* lies in the range 0 to 255, which can be represented in 8 bits. This is usually written in short-hand form as [0,255]. Then, the new intensity value for that pixel is *Y* where

$$Y = (255-X).$$

Notice that whatever the value of *X*, *Y* also lies in the range [0,255]. It is obvious that negating an image twice results in an image that is exactly the same as the original. Negation is therefore said to be *reversible*.

Many other monadic operations are reversible. Addition, subtraction and multiplication are reversible but only if there has been no hard limiting. Squaring the image intensities can be reversed by applying the square-root operation. (Figure 4.7) The order in which these operations are performed can be reversed. There are a few very minor differences between the restored and original images.

These are due to *quantisation effects*. Figure 4.7 also shows the results of applying reversible monadic operations, based on *anti-logarithm* (*exponential*) and *logarithm* intensity mapping functions. The fact that are reversible is clear from the fact that Figure 4.7[TL] and [TR] are almost identical.

Any one of the six images in Figure 4.7 can be transformed into any other, with only slight loss of detail due to quantisation effects. So, no information is removed from, or added, to an image by performing square, square-root, logarithm or anti-logarithm operations, even though the images in Figure 4.7 *look* very different. Within the limits imposed by quantisation effects, a machine can produce exactly the same result from processing any of these images, irrespective of their visual appearance. This is further evidence that machines and people do not see the world in the same way. Another important lesson: we must not rely solely on our eyes to judge the success of a vision system.

Any sequence of monadic operations can be replaced by a single monadic operation. Some examples are shown in Figure 4.8, which also defines the operators just described in mathematical notation

Contrast Enhancement

Figure 4.7[TL] is an unprocessed image derived directly from a digital camera. Squaring the intensities (Figure 4.7[CL]) improves the image contrast. However, in Figure 4.9 the situation is different. Squaring the intensities improves the contrast of the background (top-right corner) but spoils our view of the "foreground" detail. Square-root improves that (Figure 4.9[CL]), as does logarithm (Figure

4.9[BL]) and two composite operations. (Figure 4.9[CR] and [BR]). Which on of these we consider to be "best" depends on what we hope to achieve. What enhances the visibility of one part of an image may not be appropriate for another. A Machine Vision system is able to analyse the various parts of an image in quite different ways and, afterwards combine the results, if necessary. Humans and animals cannot do this, since they process the whole image at the same time. A machine can do this because it can store images. A brain cannot! We will consider this again later but let us return now to discuss contrast enhancement in more detail. What we have described so far is naive and very limited.

A simple but very effective contrast enhancement procedure can be expressed in terms of two functions that we have already met: subtracting a constant and multiplying by a constant. Here is the definition of our new operation:

Find the minimum intensity in the image. Call this I_{min}.

Subtract I_{min} from the intensity value of each pixel in the given image.

Find the maximum intensity in the new image resulting from Step 2. Call this I_{max}.

Rescale the intensity value of each pixel in the new image by multiplying by $255/I_{max}$.

Applying this to the tiny image array that we used earlier, we get

Stretching intensity

I_{min} = 122

Rescaling: 2.18 [= 255/(239-122)]

(In this case, it is assumed that the minimum and maximum intensities have been calculated over this 5x5 pixel array,

not over a larger image.) Notice that the minimum intensity is now 0 (zero) and the maximum 255. This process is called *stretching the intensity scale*. Figure 4.10 demonstrates this on two low-quality images, prepared specially for this illustration. Such an impressive improvement in image quality is not always achieved and alternative techniques, such as those discussed later, might do even better.

Mapping functions

Monadic functions can be expressed in another more general way, without the explicit use of mathematical notation. They can be defined instead, using a table of stored numbers. Figure 4.2 explains how a monadic operation can be performed using a *Look-up Table (LUT)*. We will discuss later how suitable entries in such a table might be obtained. The LUT has the same number of entries as there are possible intensity levels (256) and defines how pixel intensities in the output image are derived from the those in the input image. A LUT is a very convenient way to implement a monadic function with an arbitrary Mapping Function. This might be pre-calculated in several ways:
(1) Applying a mathematical formula on a once-and-for-all basis
(2) Sketching a graph
(3) Performing some calculation on the image to fill the entries in the LUT.

A LUT is well suited to implementation in either a computer or specialised electronic image processing hardware. Let us see how it works.

The equation defining the mapping function that performs the square-root operation is given in Figure 4.8. (This equation includes rescaling, which ensures that the result is always in the range [0,255].) In this form, this computational process requires that the square-root be evaluated afresh for each pixel. This inevitably involves a great deal of unnecessary repeated calculation, since most intensity levels occur many times in an image of reasonable resolution. The LUT provides an alternative and much faster way to obtain the same result. We only need calculate the mapping function values once, for each number in the range [0,255], and save the results in the LUT. (Method 1) The mapping function is then implemented by referring to the table for each pixel. This allows the calculation to be performed much faster and allows more general (i.e non-mathematical) forms of monadic operator to be implemented. (Method 2) We can put any numbers, in the range [0,255] in the LUT. It can implement every monadic functions defined so far and many more beside. Later, we shall see how the contents of the look-up table can be calculated to obtain a monadic function that almost always achieves a significant improvement in image contrast. (Method 3) That is quite an exciting prospect!

The mapping function can conveniently be visualised by drawing the LUT contents as a graph (Figures 4.11 and 4.12). Graphs like these enable us to understand what the mapping function stored in a LUT will do. For example, any upward-turning curve, like that in Figure 4.11[CC], is associated with a mapping function that suppresses detail in the dark parts of an image and accentuates differences between areas of light grey. On the other hand, a curve like that in Figure 14.11[CR] does the reverse.

Thresholding

Thresholding is a very useful image processing function but not in the obvious way that is usually anticipated by newcomers to our subject. The basic concept of thresholding is straightforward: each output pixel is set to

White, if its intensity is greater than or equal to some predefined number, called the *threshold parameter*.
Black, if it is not.

Notice that the threshold parameter is held constant over the whole picture and that the result is a binary image.

The effect of thresholding with different parameter values is illustrated in Figure 4.13. Intuitively, thresholding seems to be ideal for processing silhouettes of back-lit opaque objects. This typically produces an image in which there is a dark "blob", against a bright back-ground, while there are very few pixels with mid-grey values. Figure 4.14 shows one satisfactory result for a back-lit gear. The bar chart in Figure 14[B] is called the *intensity histogram*. This is a statistical summary of the distribution of intensities in an image. Its derivation and uses will be discussed in more detail later. For the moment, let it suffice to say that, if the the histogram has two well-defined peaks, the bottom/centre of the "valley" indicates a good value for the threshold parameter.

However, dangers are lurking! Figure 4.15 demonstrates a frequently encountered situation in which thresholding does not work well. (I can supply a short video demonstrating this) To the human eye, the outline of the bottle appears clearly defined but did you notice how dark the four corners of the image are? In this case, it is impossible to find a values for the threshold parameter that will simultaneously

ensure that the bottle outline and background are intact. We will see later that it is possible to obtain a good outline of the bottle by other techniques, even though simple thresholding is unable to do so. Thresholding is just part of that more complicated procedure.

Intuitively, thresholding is an obvious way to reduce a grey-scale image to binary form. Figure 4.16 provides further evidence that it does not always work well. Several times, I have argued about its efficacy with newcomers to Machine Vision, who find it difficult to believe that such an "obvious" technique will not reliably produce the good results they expect. The principal reasons for this is that the human visual system automatically compensates for slow variations in brightness, both in time and across space. As dusk approaches, the variation in ambient light is far from obvious to the eye, until the sun is close to the horizon. However, camera exposure settings must be adjusted over a wide range during this time, indicating that the light level does change significantly. Smooth spatial brightness variations are also accommodated by the eye, as is evident in Figure 4.15. Any sharp intensity "steps" are noted immediately, whereas smooth changes may not be. To a machine using thresholding, gradual non-obvious background intensity variations may be critical.

Despite these misgivings, thresholding is a very useful image analysis tool. It is widely used in simple industrial vision systems but care must always be exercised when designing the lighting, to ensure that thresholding will be both effective and reliable.

Pseudo-colour

Pseudo-colouring is a convenient way to improve feature visibility and, as, a result, is in widespread use, including areas such as thermal imaging, astronomy, scientific and medical microscopy and airport baggage x-ray inspection. Pseudo-colour exploits the eye's greater sensitivity to variations of colour compared to changes of brightness. When I first started studying image processing (mid-1970s), I worked on a project studying thermal signatures of ships. (Since then, I have avoided working on military applications of Machine Vision.) My students and I viewed and processed one particular monochrome thermal image, derived from a maritime scene, many times for two years. One day, we displayed it in pseudo-colour and immediately we saw a ship lurking on the horizon that we had never spotted before. Imagine what would have happened had that been a hostile vessel! The lesson is that, during visual examination of a scene, pseudo-colour is often able to alert us to things that we might never spot otherwise.

Pseudo-colour will be useful to us throughout this book. That is why it is introduced so early, alongside other image-enhancement techniques. The process of pseudo-colouring a monochrome image requires nothing more that three monadic operators. (Figure 4.17)

> Apply three separate monadic mapping functions to the grey-scale image, to produce three new images that we will call J_R, J_G and J_B.
>
> "Assemble" J_R, J_G, J_B as the RGB components of a single colour image.

Designing the mapping functions for a pseudo-colour displays is probably best approached experimentally,

although an experienced vision engineer can anticipate what will be "good" mapping functions.

Figure 4.18 shows one popular pseudo-color mapping pattern: one component rises as the intensity increases; another falls, while the third increases to a maximum (at mid-grey) and then falls. Furthermore, we can combine pseudo-colouring with one or more monadic functions, such as negate, square, square-root, logarithm and anti-logarithm (exponential) (Figure 4.19) There is no single "best choice"; we use whatever is helpful. Some machines, designed for applications, such as x-ray baggage inspection and detecting tumours in body-scan images, allow the user to switch rapidly and easily between different pseudo-colour mappings. Some even update the mapping functions automatically, to produce a dynamic pseudo-colouring effect.

One particular application that uses pseudo-colouring deserves special note. Thermal imaging is frequently used by fire-fighters and disaster-rescue teams. It is also used to detect hot-spots in electrical, electronic and mechanical systems. (Figure 4.20) Another important application area for thermal imaging is in energy conservation, to detect places where heat is being lost rapidly from buildings. In all of these, it is customary to relate the intensity in the thermal image to the temperature of surfaces in the scene being viewed: hot spots are displayed as white or red, while cold areas are shaded blue. (Of course, this is contrary to physical reality, where hot bodies emit light with a high content of blue light and cooler ones glow red.) Pseudo-colouring is often designed to preserve this convention. Sometimes, the pseudo-colour mapping is designed to

retain the intensity of the original monochrome image. In this case, cool areas are mapped to dark blue, hot ones to bright red and very hot parts to white. (Several example are show in Chapter 7: Applications.)

We will encounter many more examples of pseudo-colouring without further comment, as we progress through this book. The important things to remember are that pseudo-colouring is completely arbitrary and that its sole justification lies in being able to assist and augment human visual inspection.

Dyadic operators

Dyadic operators act on two pictures at once, to generate a third. (Figure 4.21) The intensity for each pixel in the output image is some arithmetic combination of the intensities of those pixels at the corresponding positions in the two input images. Let us define some notation, so that we can understand these operators easily.

A The intensity of the pixel in position *[i,j]* of the *first* input image (Blue)
B The intensity of the pixel in position *[i,j]* of the *second* input image (Green)
C The intensity of the pixel in position [i,j] of the *output* image (Red)

In what follows, the operation is performed for all pixels (i.e. for all *i* and *j)* simultaneously.

Adding two images

C = (A+B)/2

(See Figure 4.22) This produces a result resembling a photographic double exposure. It is not used on its own very often. Multi-image addition is used sometimes to reduce the noise from cameras.

Subtracting two images

C = (A-B+255)/2

This is far more useful for reasons that will be discussed in a little while. It produces an identical result to negating one of the input image and adding the result to the other input image. (See Figure 4.22)

Multiplying two images

C = A.B/255

This also produces a result resembling a photographic double exposure. It is not used very often.

Maximum of two images

C = MAXIMUM(A,B)

This operator, called the *Dyadic Maximum*, is able to superimpose white pixels, perhaps forming text, a line drawing (e.g. annotation arrows), or a graph, on dark parts of an image. Elsewhere, the grey-scale image is unchanged. (Figures 4.23 - 4.24)

Minimum of two images

C = MINIMUM(A,B)

This operator, called the *Dyadic Minimum*, is able to superimpose black pixels, perhaps forming text, a line drawing (e.g. annotation arrows), or a graph, on bright parts of an image. (Figures 4.25 & 4.26) Elsewhere, the grey-scale image is unchanged. It is often used to mask unwanted

parts of an image. Dyadic Minimum can be used to mask a colour image by applying it to the RGB components separately. (Figure 4.27)

Reducing Noise by Combining Many images

Video cameras inevitably generate a certain amount of "noise", which is visible as a dynamic speckle-like pattern superimposed on the ideal picture. Capturing two video frames, even if they are acquired consecutively, are slightly different, due to noise. As a result, even a smooth, uniform surface that is viewed under constant lighting conditions, has a slight speckle effect. Noise is a fundamental and unavoidable feature of all devices that convert light into an electrical signal. Thermal (infrared imagers and cameras operating under low-light conditions tend to generate particularly noisy pictures. For an image processing machine, noise is a nuisance: edges become jagged and plain areas appear mottled. We will describe some effective procedures for reducing the effects of noise in an image later but here we will demonstrate how the noise level noise can be by combining several images.

Noise is usually additive. That is, a "noise image", resembling a snap-shot of television "snow", is unavoidably added by to the signal that represents the scene of interest. The physics of photo-electrical energy conversion shows noise is unavoidable; it is not done deliberately. All the camera designer can do is reduce noise to the lowest possible level. An effective way to reduce the effects of additive noise is to add together several digital images captured from the camera's video stream. (Figure 4.28) For this to be effective, the scene being viewed must not change during the averaging process. Objects in front of the camera

must not move, or change colour and the illumination must not be altered

Local Operators

Local operators are simple image filters that combine the intensities of several pixels in order to calculate each new intensity value in the output image. They operate on a single input image. They can perform some very useful functions and are easy to implement.

Consider a group of 9 pixels centred on the *[i,j]* pixel. (That is, the pixel in column *i* and row *j*.) The intensity for pixel *[i,j]* in the output image is found by combining the intensities of the corresponding pixel and its 8 immediate neighbours in the input image. For convenience, we will use the following notation to represent these intensities:

	Column i-1	Column i	Column i+1
Row j-1	A	B	C
Row j	D	E	F
Row j+1	G	H	I

Notice that *I* (upper case) is an intensity value and *i* (lower case) is a column number. *E* is the intensity of the *[i,j]* pixel. *A* is the intensity of the *[i-1,j-1]* pixel, displaced one row up

and one column left from *[i,j]*. *B* is the intensity of the [*i,j-1]* pixel immediately above *[i,j]* and so on.

We are now able to illustrate specific examples of local operators. Remember that the operations that we are about to describe are performed simultaneously for every pixel in the output image. The edges of the image require special consideration. For the moment, we will ignore this topic but will return to it later.

Local Averaging

See Figure 4.29. *Local averaging* is the process of adding the intensities of a small group of close pixels and produces a slight blurring effect. For example. the intensity at pixel *[i,j,]* in the output image might be calculated as follows:

(A+B+C+D+E+F+G+H+I)

This generates in image in which the intensity values do not necessarily lie inside the range [0,255]. To ensure that they do, the formula should be modified to

(A+B+C+D+E+F+G+H+I)/9

This is an example of *normalisation*, which we have already encountered, when discussing monadic and dyadic operations. It will be ignored for the moment because it will distract us from other important issues.

To emphasise the fact that the *[i,j]* pixel is being replaced, we can write this as an assignment:

E' ⬅ *(A+B+C+D+E+F+G+H+I)/9*

It must be understood that *E* to the right of the arrow refers to an intensity value in the input image and that *E'* to its left

is in the output image. (All pixel values in the output image are effectively calculated at the same time.)

Blurring

The blurring effect produced by local averaging can be increased in one of two ways:

Repeating the process several times.

Using a larger processing neighbourhood, for example 5x5, 7x7, 9x9 and 15x15 pixels.

Blurring generated in this way is very much like that produced by mis-focussing a camera, or projector, or by squinting. Looking through smoke or thin fog, cloudy water and a translucent film produces a similar blurring effect, as do cataracts. These all lead to image degradation, so why are we interested in doing the same thing inside a computer? There are two reasons:

(a) By subtracting the blurred image from the original, we can eliminate variations in background intensity, thereby making thresholding very much more reliable. We will encounter this combination of blurring, subtraction and thresholding many times in the following pages

(b) Consider an image that it totally black, except for a single white point. By applying a blurring operator, the point is spread out to form a fuzzy blob. (Figure 4.29[BL] and [BR] Notice that repeated application of a blurring filter, makes the image even more blurred. More about this later.) The diameter and shape of this fuzzy blob can be measured, alloying us to quantify and thereby control the blurring process. We refer this fuzzy blob as the *Point-spread Function* of the blurring operation. (Figure 4.29[BL] and [BR])

Edge Effects

All local operators produce anomalies around the edge of an image. *Edge Effects* become larger as the processing neighbourhood (the *kernel*) is increased in size. (Figure 4.30) A local operator based on a kernel of size *(2m+1)x(2n+1)* produces edge effects *m* pixels wide on the left and right of the filtered image and *n* pixels wide at its top and bottom.

The only safe way to accommodate edge effects is to ignore the edges of the image. (Blue area in Figure 4.30.)

Low-pass & High-pass Filtering

Blurring is an inherent property of *low-pass filtering*: narrow stripes merge, whereas wider ones do not. Small, well-separated spots tend to "melt" into the background; dark spots become brighter and versa versa. If the spots are close together they blend into one larger fuzzy cluster. Large areas with no sharp intensity steps are almost unchanged, except that their edges become blurred.

Subtracting a blurred image from the original implements a *high-pass* filter. (Figures 4.31-4.33) This retains only small, spots and accentuates sharp dark-bright intensity transitions.

A 3x3 high-pass filter can be implemented by the following formula which calculates a new value (*E'*) for a single point in the output image).

$$E' \leftarrow 8.E - (A+B+C+D+F+G+H+I) \qquad \text{(no normalisation)}$$

or with normalisation

E' ⬅ *(2040 + 8.E - (A+B+C+D+F+G+H+I))/4080*

A range of high-pass filters can be implemented, by subtracting the original image and the result of blurring using a large-kernel local-averaging filter.

Directional filtering

It is easy to modify the formula to blur in one direction only. For example, a 1-dimensional blur, in the horizontal direction, can be achieved by setting the intensity to

E' ⬅ (D+E+F).

To blur in the vertical direction, use

E' ⬅ (B+E+F).

These have kernel sizes of 3x1 and 1x3 respectively.

Using the same terminology, consider the following filter

E' ⬅ *(A+B+C) - (G+H+I).*

If used on its own, the first part of the formula defining the filter [i.e. *(A+B+C)*], blurs *horizontally* and shifts the image *down* by one row. On its own, the second part, *(G+H+I)*, blurs horizontally and shifts the image *up* by one row. The minus sign between these two parts indicates that we subtract these two partial results. The overall effect is to highlight *horizontal edges*; the filter is sensitive to steep intensity gradients along the vertical axis. (Figure 4.34)

Another, similar filter, based on the formula

E' ⬅*(A+D+G) - (C+F+I)*

detects *vertical edges*.

Adding the monadic *Fold operator* (Figure 4.11[BR]), it is possible to produce an edge detector that is sensitive to both increasing and decreasing intensity changes. Here is the calculation for detecting sharp horizontal intensity gradients:

E' ⬅ **|** *(A+D+G) - (C+F+I)* **|**

[Note: **|**..**|** is the modulus function. *I* is a number.]

Of course,

E' ⬅ **|** *(A+B+C) - (G+H+I)* **|**

detects sharp vertical intensity gradients

These two steps can be combined, using dyadic addition to create an *edge detector* that highlights all sharp intensity gradients, in *any direction*. Here is the formula

E' ⬅ **|***(A+D+G) - (C+F+I)* **|** + [**|***(A+B+C) - (G+H+I)* **|**

(No attempt is made here to normalise the intensity scale.)

This is just one of the numerous of edge detectors that have been devised but it is not the best! Here the definition of another, the popular *Sobel Edge Detector*:

E' ⬅ *[* **|***(A+2D+G)-(C+2F+I)* **|** *+ [* **|***(A+2B+C)-(G+2H+I)* **|** *]/6*

(Division by 6 normalises the output intensity to lie in the range [0,255].)

Weighted Local Operators

The component

(A+2D+G)

of the Sobel Edge Detector blurs the image but not as much as

(A+D+G)

does. Sometimes, it is desirable to give greater emphasis to pixels near the centre of the processing neighbourhood. This is possible if we multiply each pixel within it by a different amount. For example we might use the formula

$$E' \leftarrow (A+2B+C+2D+3E+2F+G+2H+I)$$

to produce a mild blurring effect. Let us extend this idea to the a more general case. To do so, it is convenient to hold a set of multipliers (called *Weights*) in an array (called the *Weight Array*) thus:

Wa	Wb	Wc
Wd	We	Wf
Wg	Wh	Wi

The weights are combined with the intensity values within the 3x3 pixel processing neighbourhood as follows:

$$E' = A.Wa + B.Wb + C.Wc + D.Wd + E.We + F.Wf + G.Wg + H.Wh + I.Wi$$

So, the weight array for the previous example is as follows:

1	2	1
2	3	2
1	2	1

Weight values are shown in blue. To appreciate the local operators better, let us consider some more examples. In each case, the input image consists of only 0s and 1s. (These are numbers, not logical values.) To make things as

simple as possible, normalisation will be ignored, except where stated explicitly.

Upright Cross

-1	+1	-1
+1	+1	+1
-1	+1	-1

A filter using the weight array
produces a maximum output value (5) when the image is

0	1	0
1	1	1
0	1	0

Notice the use of both positive and negative weights.

Isolated White Point

A filter using the weight array

-1	-1	-1
-1	+9	-1
-1	-1	-1

produces a maximum output value (9) when the image i

0	0	0
1	+1	0
0	0	0

Counting White 8-Neighbours

We simply use the following weight array to count pixels with intensity 1 and lying inside each3 X3 window.

+1	+1	+1
+1	+1	+1
+1	+1	+1

We are using the local-averaging operator on a grey-scale image. That is, intensities [0,1] represent arithmetic values. Now, for a moment, think of [0,1] as logical entities. Blurring of the distinction between grey-scale and binary operations is very useful and we will return to this in the following chapter.

Detecting Specified Binary Patterns

A filter using the weight array

1	2	4
8	16	32
64	128	256

produces a unique value for every possible pattern of 0s and 1s in the 3x3 neighbourhood. The output is in the range [0,511]. By thresholding it is possible to detect any specified pattern of 0s and 1s in a 3x3 neighbourhood. Very useful! Here is an example: the image

1	0	1
0	1	0
0	1	0

produces a value of 149 (= 1+4+16+128), so we can use this to detect the simple pattern 'Y'.

Horizontal Smoothing & Vertical Differencing

Here is a simple 3x3 weight array that performs horizontal smoothing & vertical differencing.

1	1	1
0	0	1
-1	-1	-1

The idea can be extended to larger processing windows and weight arrays. For example, here is a 9x5 weight array that provides greater smoothing along rows and detects slower intensity changes along columns.

1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1

This filter produces exactly the same results as this 9x1 filter

1	1	1	1	1	1	1

followed by this 1x5 filter

+1
0
0
0
-1

Applying them in the reverse order also produces identical results.

Repeating Low-pass Filtering

Applying a 3x3 local-averaging filter to an image with a single white pixel (intensity 1), while all other pixels are black (0) produces the following result.

1	1	1
1	1	1
1	1	1

Applying the same filter to this result yields the following image

1	2	3	2	1
2	4	6	4	2
3	6	9	6	3
2	4	6	4	2
1	2	3	2	1

Now, applying the same filter for a third time produces the image

1	3	6	7	6	3	1
3	9	18	21	18	9	3
6	18	36	42	36	18	6
7	21	42	49	42	21	7
6	18	36	42	36	18	6
3	9	18	21	18	9	3
1	3	6	7	6	3	1

Figure 4.29 shows that repeating a low-pass filter produces even greater blurring. The process can be repeated as many times as we wish but note that the image border covered by edge effects becomes progressively wider.

The array just given (red) is an image array but suppose we use the same numbers to define a 7x7 weight array. This filter produces an identical result to that obtained by applying the 3x3 local-averaging filter three times.

Another possibility: exactly the same result can be obtained in a sequence of six steps

- 1x3 local-averaging filter, applied times, followed by
- 3x1 local-averaging filter, applied three times.

There are even more ways to achieve identical results, so we are free to choose one that makes the computation cheapest/fastest.

Non-linear Local Operators

So far, we have discussed so-called *Linear Local Operators* in which intensity values are added or subtracted. Now, we

turn our attention to local non-linear operators that combine intensities using the Maximum (*MAX*) and Minimum (*MIN*) operations. Here are two examples, based on a 3x3 processing window

Largest Neighbour (LNB)

E' ⬅ *MAX(A,B,C,D,E,F,G,H,I)*

Smallest Neighbour (SNB)

E' ⬅ *MIN(A,B,C,D,E,F,G,H,I)*

Larger processing windows are, of course, possible but these can be replicated by applying LNB/SNB several times.

The effect of LNB is to make bright regions slightly bigger and dark regions correspondingly smaller. Of course, SNB does the reverse. For this reason, LNB is often called *Dilation* and SNB *Erosion*. However, the latter are general terms and are often used when referring to filters with non-square kernels.

Two particularly effective edge detectors can be implemented by subtracting the result of LNB/SNB from the original image. (Figure 4.35)

Applying LNB then SNB does not reconstruct the original image. In fact, it tends to eliminate small dark spots and narrow dark streaks. If the result is subtracted from the original image, these features are clearly visible, while the background is "flattened". The result is a very useful filter, called a *Crack Detector*. (Figure 4.36 and 4.37). Sometimes, this process is termed *Closing*. If we use the operator sequence *[SNB, LNB]* instead, the filter is said to perform *Opening*.)

Cracks and scratches are common manufacturing faults and the Crack Detector has proved to be very useful in many practical applications, such as detecting thin fragments of bone in X-rays of fish and meat, thin wires, scratches, and of course, cracks. Since it highlights thing dark streaks, we must apply negation first, if we want to find bright lines.

Adding the images resulting from LNB and SNB applied successively is an efficient way to reduce "noise" in an image.

Another way to do this is to use the *Median Filter*, defined thus for a 3x3 kernel:

$$E' \leftarrow FIFTH\_LARGEST(A,B,C,D,E,F,G,H,I)$$

(Figure 4.38.)

There are many more interesting possibilities. For example, an edge detector that is relatively insensitive to noise can be created by computing the following quantities:

$$E_3 \leftarrow SEVENTH\_LARGEST(A,B,C,D,E,F,G,H,I)$$

$$E_7 \leftarrow THIRD\_LARGEST(A,B,C,D,E,F,G,H,I)$$

$$E' \leftarrow |E_3 - E_7|$$

Notice that we are beginning to define image processing operators as sequences of simpler "building blocks". It is often very much easier to understand and design algorithms in this way, rather than trying to express the calculation in one large mathematical equation.

Intensity Histograms

We encountered histograms informally earlier, in Figures 4.14 & 4.28. Now, we need to explain how they can help us to analyse an image in quantitative terms.

A *Histogram*, properly called an *Intensity Histogram*, is a table of numbers indicating how many pixels have each possible intensity value. (Figure 4.39)

Figure 4.40 shows three sample histograms. Notice that Figure 4.40[TR] has two distinct well-separated peaks. (It is therefore said to be *Bi-modal*.) Back-lighting opaque objects often generates histograms like this. Figures 4.14, 4.41[T], 4.42[T] & 4.43 show further examples of bi-modal histograms. The valley between these peaks corresponds to the few pixels that are on the edge of the object silhouette. Placing an intensity threshold at the bottom/centre of such a valley is often an effective way to separate the object from its background. (Figure 4.42) However, back-illuminating a transparent object, such as a glass bottle, may produce a histogram in which there is no obvious valley. (Figure 4.40[CR]).

Before we move on, refer back to Figure 4.41[BR]. This strange form of histogram is sometimes found when an image has been JPEG coded.

Figure 4.44 shows the histogram derived from the monochrome image of a quiche and the histograms of its parts. Notice that the peaks they generate overlap, showing that they are not perfectly separable by thresholding. Figure 4.44 also shows the results of thresholding at the intensity levels corresponding to the valleys in the histogram.

The histogram can be used as an analytical tool, for example, to study information loss caused by image coding. In Figure 4.45, the histograms of an image and its JPEG-coded version are quite different, even though the images themselves are visually very similar. When these images are subtracted, we obtain what seems to be a meaningless noisy mess. (Figure 4.45[BL]) However, the histogram of this difference image is revealing: it consists of a single narrow spike (Figure 4.45[BR]) whose width is a measure of the accuracy of the JPEG coding: a narrow spike indicates that little information has been lost.

Cumulative Histogram

The *Cumulative Histogram* is derived from the intensity histogram by a simple recursive calculation, involving only addition. Let *h(i)* be the intensity histogram value for intensity level *i* and *H(i)* the Cumulative Histogram value. Then, we calculate *H(i)* as follows

$$H(i) = H(i-1) + h(i)$$

where

$$H(0) = h(0).$$

This is illustrated in Figure 4.46. Also see Figure 4.47.

The cumulative histogram is often more useful than the intensity histogram when we want to measure a grey-scale image. For example, we can use the cumulative histogram to calculate the intensity threshold that segments the image in given proportions. For example, the 5% and 95% centiles (called *C(5)* and *C(95)*) might be used to stretch the intensity scale, to improve contrast. The calculation is as follows:

$$X' = 255*(X - C(5))/(C(95) - C(5))$$

(This inevitably requires hard limiting, to ensure that the result lies within the range [0,255].) *X′* is the pixel intensity after rescaling and *X* is the intensity before. Of course, other pairs of centile values might be used instead. (e.g. 2% and 97%) (Figure 4.48) Rescaling intensity like this ignores both extremely bright and extremely dark pixels and makes the process less sensitive to noise than the simpler alternative:

$$X' = 255*(X - Imin)/(Imax - Imin)$$

where *Imax* and *Imin* are the maximum and minimum intensity values, calculated over the whole image.

Histogram Equalisation

However, the greatest benefit provided by the cumulative histogram is that it forms the basis for a very effective *contrast enhancement* procedure, called *Histogram Equalisation.* (Figure 4.48)

Let us assume that the image has a total of *N* pixels and that, as usual, we want to normalise the calculated intensity values to stay within the range *[0,255]*. As before, let us represent the cumulative histogram by *[H(0), H(1), H(2), H(3), ..., ,H(255)]*. First, we rescale the cumulative histogram thus

$$R(i) = 255*H(i)/N, \text{ for } i = 0, 1, 2, 3, ..., 255.$$

Then, we use the rescaled values, *[R(0), R(1), R(2), R(3), ..., ,R(255)],* to fill the look-up table for a monadic operator (Figure 4.2) and apply this to the original image. The histogram of this modified image is (nearly) flat, whatever the initial shape of the histogram.

Let us summarise this process:

- Calculate the intensity histogram of the original image.

  [h(0), h(1), h(2), h(3), ..., ,h(255)].
- Compute the cumulative histogram

  [H(0), H(1), H(2), H(3), ..., ,H(255)].
- Rescale the cumulative histogram and load these values into the look-up table for a monadic operator.

  [R(0), R(1), R(2), R(3), ..., ,R(255)]
- Apply that monadic operator to the original image.

Histogram Equalisation can be applied to just part of an image. For example, black pixels might be ignored when calculating the mapping function.

Histogram Equalisation is just one of a range of powerful *Histogram Modification* procedures. In Figure 4.49, histogram equalisation has been applied in association with pseudo-colouring, to enhance the differences existing in a low-contrast image. It can also be used in combination with other monadic operators, such as *negate*, *square, square-root, logarithm, exponential,* etc. Choosing the "best" combination, for a given application is most effectively achieved interactively, by experimentation.

Histogram Equalisation allows us to threshold an image so that any given percentage of the resulting binary image is white. This is particularly useful for pre-processing textured images. In Figure 4.50 three threshold values (giving 25%, 50% and 75% white) are used. None of them is totally satisfactory, due to the uneven lighting. Although, fixed-level thresholding fails, Histogram Equalisation does not, as it

clearly demonstrates that the original image is not good enough.

Local Area Histogram Equalisation

Histogram equalisation can be useful for preprocessing, as a step towards analysing image texture. However, applying the procedure to the complete image can be disappointing as Figure 47(B] and Figure 50 show. What appear to be minor changes in the intensity of the background are, in fact, critical.

One seemingly attractive way to avoid problems caused in this way is to apply Histogram Equalisation to small parts of an image. Small adjoining non-overlapping patches, covering the whole of the input image, would be processed separately and then "reassembled" to form a complete image. Unfortunately, this does not work very well, because the edges of adjacent patches show marked intensity differences. We will not pursue this approach.

A superior solution is to perform the histogram-equalisation calculation for a block of pixels covering a small compact region of the image but keep the result for only the central pixel. The processing window is scanned across the entire input image and the process repeated for each pixel. This is, of course, exactly what a non-linear local operator does. The name of this operation is fairly obvious: *Local Area Histogram Equalisation*. The same result can be obtained in a simpler way: within each processing window, count the number of pixels that are brighter/darker than the central pixel. (Of course, some rescaling may be needed.) Figure 4.51 shows Local Area Histogram Equalisation applied to a sample of plain carpet. Compare this result with Figure 4.50.

Row Integration & Row Maximum

These are two operations that seemingly do not have any real value. However, they are both invaluable components of more complex procedures. Figure 4.52 explains them both. Figure 4.53 illustrates the use of row maximum and Figure 4.54 how row/column integration can be used to detect linear features that are aligned parallel to one of the image axes. Figure 4.55 demonstrates how sensitive row and column integration are to the orientation of those features.

Image Rotation

Image rotation is not as straightforward as we might think. (Figure 4,56) Sharp edges are always degraded to some extent by rotating a digital image. The reason is explained in Figure 4.57[T]. There are two simple approaches to estimating values for those output pixels that do not coincide exactly with input pixels:

- Choose the nearest neighbour. (D in Figure 4.57[B])
- Use interpolation to estimate an appropriate value. Linear interpolation of the values of the four nearest pixels (A, B, C and D in Figure 4.57[B]) is the simplest and most obvious.

Image Warping

Rotation and warping share the same fundamental difficulty that is summarised in Figure 4.57. Why should we be interested in warping images? There are several reasons:

- A vantage point close to the subject introduces geometric distortion.
- Lenses can introduce significant geometric distortion.

- Images may be derived from non-standard cameras.
- The surface being viewed may be curved.

Industrial vision systems are sometimes required to inspect an object, rotating continuously using a line-scan camera. (Figure 4.58. Also see Chapter 2) In this situation, it is often advantageous to convert polar coordinates (circular) to Cartesian coordinates (rectangular). (Figure 4.59) The reverse axis transformation allows us to reconstruct an image that can be compared to the normal view. (Figure 4.59[TL]). While we can process images in either format, filtering results will not be the same because pixels in the polar-axis version are not evenly spaced. (Figure 4.58[BR])

Figure 4.60 demonstrates a situation where it would be easier for a program to interpret the polar-coordinate image than the Cartesian-coordinate version that a standard camera generates.

Imagine trying to inspect the mouth of a bottle, using a standard array camera (not line-scan) looking vertically downwards, onto its top. Most of the interesting detail is conveyed by pixels near the edges of the image; there are many "wasted" pixels within the mouth of the bottle. These occupy computer memory and reduce the speed of inspection (measured in bottles/minute) Figure 4.61 shows how the "hole in the middle " can be removed.

Figure 62 and Figure 63 demonstrate several more examples of image warping. Geometric distortion can often be corrected satisfactorily by software, provided the distortion mapping function can be expressed

mathematically, or derived empirically from experimental observation. Like rotation, image warping requires interpolation and it therefore inevitably associated with loss of resolution, in some locations worse than others

The following situations: can benefit.

- Close-up viewing (Figure 4.63[TC])
- Viewing a spherical, cylindrical surface
- General curved surface such as a page in a thick book (Figure 4.63[BC])
- Pin-cushion and barrel distortion (Figure 4.63[TR])
- Lens distortions (e.g. cylindrical, fish-eye and anamorphic lenses)
- Reflections on surfaces, such as cylinders, cones (Figure 63[BR])
- Images derived from flying-spot scanners.
- Variable central ("foveal") & peripheral resolution.

Detecting Straight Lines

The *Radon Transform (RT)* is used to detect linear features in images. When combined with an edge-detection operator, it can detect continuous lines, straight edges and linear arrangements of disconnected spots. It is an analytical tool and, as such, does not show visually recognisable features seen in the original image.

The Radon Transform, combines image rotation and row-integration, so let us look at these first. Refer to Figure 4.55 again. It illustrates the directional sensitivity of the row/column-integration operator. Figure 4.64 demonstrates the

ability of row-integration to detect when printed text is aligned to the horizontal image axis. Figure 4.65 illustrates the same point, with reference to multiple micro-electronic devices on a ceramic plate

The procedure for generating the RT, is defined thus:

1. *Repeat steps 2-4 for suitable values of N. (e.g. N = 0:179)*
2. *Rotate the input image by angle N.k, where k is a constant. (e.g. k = 1°)*
3. *Row-integrate the result of step 1.*
4. *Copy the right-most column of the image generated in step 3 and place it into column N of the RT output image.*

Notice that we can adjust the range of angles and the constant *k* (angular increment for image rotation) So, for example, we might generate a coarse, wide-angle RT (e.g. N = 0:19, k = 5°) or a narrow, high-resolution version (e.g N = 25:125, k = 0.1°)

Figure 4.66 shows how the RT is used in practice. Notice the bright spots in Figure 4.66[CL], each of which indicates the presence of a strong linear feature in [TR]. It is possible to relate the position of bright spots in the RT to linear features in the original image. This results in the *Inverse Radon Transform*. Figure 4.66[BL] and Figure 4.67