3.4 Group Operations

3.4.1 Template Convolution

Group operations calculate new pixel values from a pixel's neighbourhood by using a 'grouping' process. The group operation is usually expressed in terms of *template convolution* where the template is a set of weighting coefficients. The template is usually square, and its size is usually odd to ensure that the result positioned precisely on a pixel. The size is usually used to describe the template; a 3×3 template is three pixels wide by three pixels long. New pixel values are calculated by placing the template at the point of interest. Pixel values are multiplied by the corresponding weighting coefficient and added to an overall sum. The sum (usually) evaluates a new value for the centre pixel (where the template is centred) and this becomes the pixel in the output image. If the template's position has not yet reached the end of a line, the template is then moved horizontally by one pixel and the process repeats.

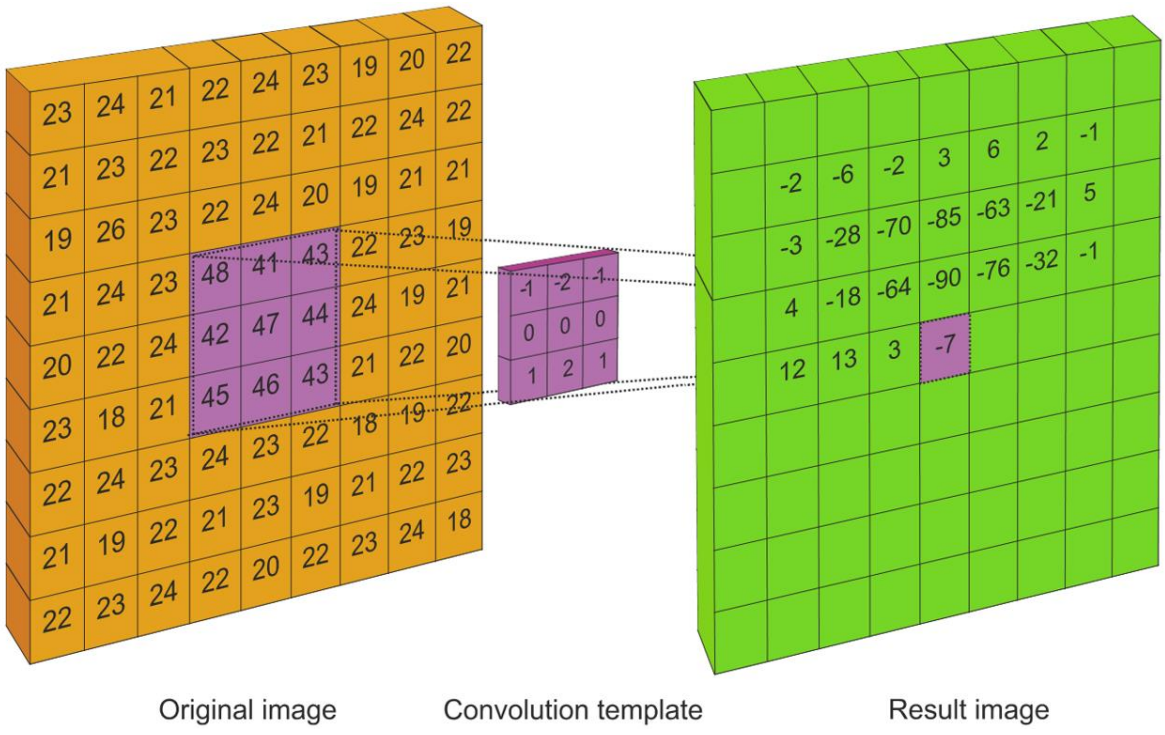


Figure 3.11 Template Convolution Process

This is illustrated in Figure 3.11 where a new image is calculated from an original one, by template convolution. The calculation obtained by template convolution at the centre pixel of the template in the original image becomes the point in the output result image. Since the template cannot extend beyond the original image, a new value cannot be computed for points at the border of the result image. When the template reaches the end of a line, it is repositioned to proceed from the start of the next line. The process is shown part way through the raster scan, the next pixel to be calculated would be derived from the nine points to the right of the current position of the centre point of the template, and stored to the right of the point containing -7. For a 3×3 neighbourhood, Figure 3.12, nine weighting coefficients w_t are applied to points in the original image to calculate a point in the new image. The position of the new point (at the centre) is shaded in the template.

w_0	w_1	w_2
w_3	w_4	w_5
w_6	w_7	w_8

Figure 3.12 3×3 Template and Weighting Coefficients

To calculate the value in new image, **N**, at point with co-ordinates *x*,*y*, the template in Figure 3.12 operates on an original image **O** according to:

$$\mathbf{N}_{x,y} = \sum_{i\in\text{template}} \sum_{j\in\text{template}} w_{i,j} \times \mathbf{O}_{x(i),y(j)} \tag{3.1}$$

where the coordinates of the image point $x(i), y(j)$ denote the position of the point that matches the weighting coefficient position. Note that we cannot ascribe values to the picture's *borders*. This is because when we place the template at the border, parts of the template fall outside the image and have no information from which to calculate the new pixel value. The width of the border equals half the size of the template. In Figure 3.11 the single pixel border points have been left blank. To calculate values for the border pixels, we have three choices:

1. set the border to black (or deliver a smaller picture);
2. assume (as in Fourier) that the image replicates to infinity along both dimensions and calculate new values by cyclic shift from the far border; or
3. calculate the border pixel value from a smaller area.

None of these approaches is optimal. The results in this book use the first option and set border pixels to black. Note that in many applications the object of interest is imaged centrally or, at least, imaged within the picture. As such, the border information is of little consequence to the remainder of the process. Here, the border points are set to black, by starting functions with a zero function which sets all the points in the picture initially to black (0).

An alternative representation for this process is given by using the convolution notation as

$$\mathbf{N} = \mathbf{W} * \mathbf{O} \tag{3.2}$$

where **N** is the new image which results from convolving the template **W** (of weighting coefficients) with the image **O**.

The Matlab implementation of a template convolution operator `template_convolve` is given in Code 3.1. This function accepts, as arguments, the picture `image` and the template to be convolved with it, `template`. The result of template convolution is an image `convolved`. The operator first sets the resulting image to black (zero brightness levels). The widths `tc` and `tr` give the range of picture points to be processed in the outer `for` loops that give the co-ordinates of all points resulting from template convolution. The template is convolved at each picture point by generating a running summation of the pixel values within the template's window multiplied by the respective template weighting coefficient.

Note that according to Eqn. 2.10, for convolution one of the signals is inverted along its principal axis. For images, convolution requires inversion along both axes which is why the template's arguments are inverted in Code 3.1. We shall consider convolution again in the next Section, via the frequency domain, and in Section 5.3.2.

```
function convolved = template_convolve(image,template)
%get image dimensions
[rows,cols]=size(image);
%get template dimensions
[trows,tcols]=size(template);

%half of template rows is
tr=floor(trows/2);
%half of template cols is
tc=floor(tcols/2);

%set an output as black
convolved(1:rows,1:cols)=0;

%then convolve the template
for x = tc+1:cols-tc %address all columns except border
    for y = tr+1:rows-tr %address all rows except border
        sum=0; %initialise the sum
        for iwin=1:tcols %address all points in the template
            for jwin=1:trows
                sum=sum+image(y+jwin-tr-1,x+iwin-tc-1)*... % sum, Eq. 3.18
                          template(trows-jwin+1,tcols-iwin+1);
            end
        end
        convolved(y,x)=sum; %store as new point
    end
end
```

Code 3.1 Template Convolution Operator

Template convolution can of course be implemented in hardware and requires a two-line store, together with some further latches, for the (input) video data. The output is the result of template convolution, summing the result of multiplying weighting coefficients by pixel values. This is called pipelining, since the pixels are essentially move along a pipeline of information. Note that two line-stores can be used if the video fields only are processed. To process a full frame, one of the fields must be stored if it is presented in interlaced format. Processing can be analog, using operational amplifier circuits and Charge Coupled Device (CCD) for storage along bucket brigade delay lines. Finally, an alternative implementation is to use a parallel architecture: for Multiple Instruction Multiple Data (MIMD) architectures, the picture can be split into blocks (spatial partitioning); Single Instruction Multiple Data (SIMD) architectures can implement template convolution as a combination of shift and add instructions.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Figure 3.12 3×3 Averaging Operator Template Coefficients

3.4.2 Averaging Operator

For an *averaging operator*, the template weighting functions are unity (or 1/9 to ensure that the result of averaging nine white pixels is white, not more than white!). The template for a 3×3

averaging operator, implementing Equation 3.1, is given by the template in Figure 3.13 where the location of the point of interest is again shaded. The averaging operator is then

$$\mathbf{N}_{x,y} = \frac{1}{MN} \sum_{i \in M} \sum_{j \in N} \mathbf{O}_{x(i),y(j)} \tag{3.3}$$

where $x(i), y(j)$ are the coordinates of image points within the template and M, N are the numbers of columns and rows in the template. The result of averaging an image with a 9×9 operator is shown in Figure 3.14. This shows that much of the detail has now disappeared revealing the broad image structure. In order to implement averaging by using the template convolution operator, we need to define a template and then convolve it with the image (note also that there is an averaging operator `mean` in Matlab that can be used for this purpose).

(a) original (b) after averaging

Figure 3.14 Applying Direct Averaging

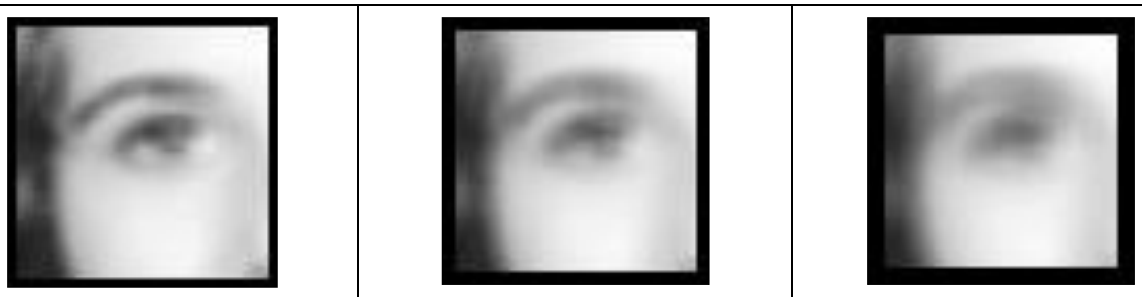
The effect of averaging is to reduce noise, this is its advantage. An associated disadvantage is that averaging causes blurring which reduces detail in an image. It is also a low pass filter since its effect is to allow low spatial frequencies to be retained, and to suppress high frequency components. A larger template, say 9×9 or 15×15, will remove more noise (high frequencies) but reduce the level of detail. The size of an averaging operator is then equivalent to the reciprocal of the bandwidth of a low-pass filter it implements

3.4.3 On Different Template Size

Templates can be larger than 3×3. Since they are usually centred on a point of interest, to produce a new output value at that point, they are usually of odd dimension. For reasons of speed, the most common sizes are 3×3, 5×5 and 7×7. Beyond this, say 9×9, many template points are used to calculate a single value for a new point, and this imposes high computational cost, especially for large images. (For example, a 9×9 operator covers 9 times more points than a 3×3 operator.) Square templates have the same properties along both image axes. Some implementations use vector templates (a line), either because their properties are desirable in a particular application, or for reasons of speed.

The effect of larger averaging operators is to smooth the image more, to remove more detail whilst giving greater emphasis to the large structures. This is illustrated in Figure 3.15. A 5×5 operator, Figure 3.15 (a), retains more detail than a 7×7 operator, Figure 3.15 (b), and much more than a 9×9 operator, Figure 3.15 (c). Conversely, the 9×9 operator retains only the largest structures such as the eye region (and virtually removing the iris) whereas this is retained more by the operators of smaller size. Note that the larger operators leave a larger border (since new values

cannot be computed in that region) and this can be seen in the increase in border size for the larger operators, in Figures 3.15 (b) and (c).



(a) 5×5 **(b)** 7×7 **(c)** 9×9

Figure 3.15 Illustrating the Effect of Window Size

```
image_eye=imread('eye_orig.jpg');
image_eye=double(image_eye(:,:,1));
image_transform=fft2(image_eye);
template_transform=fft2(pad(image_eye, ave_template(7)));
inverted_transform=ifft2(rearrange(image_transform.*template_transform));
```

The transform based implementation of direct averaging can be combined as

```
averaged_image=ifft2(rearrange(fft2(eye).*fft2(pad(eye,ave_template(7)))));
```

(a) Matlab

```
# Padding
widthPad, heightPad = width+kernelSize-1, height+kernelSize-1

templatePadFlip = createImageF(widthPad, heightPad)
for x,y in itertools.product(range(0, kernelSize), range(0, kernelSize)):
    templatePadFlip[y, x] = kernelImage[kernelSize-y-1, kernelSize-x-1]

# Compute coefficients
imageCoeff, maxFrequencyW, maxFrequencyH = computeCoefficients(inputPad)
templateCoeff, _, _ = computeCoefficients(templatePadFlip)


# Frequency domain multiplication
for kw,kh in itertools.product(range(-maxFrequencyW, maxFrequencyW + 1),     \
                               range(-maxFrequencyH, maxFrequencyH + 1)):
    w = kw + maxFrequencyW
    h = kh + maxFrequencyH

    resultCoeff[h,w][0] = (imageCoeff[h,w][0] * templateCoeff[h,w][0] -      \
                           imageCoeff[h,w][1] * templateCoeff[h,w][1])
    resultCoeff[h,w][1] = (imageCoeff[h,w][1] * templateCoeff[h,w][0] +      \
                           imageCoeff[h,w][0] * templateCoeff[h,w][1])
```

(b) Python

Code 3.2 Template Convolution via the Fourier Transform

3.4.4 Template Convolution via the Fourier Transform

The Fourier transform actually gives an alternative method to implement template convolution and to speed it up, for larger templates. The question to be answered here is 'how big?'. In Fourier transforms, the process that is dual to *convolution* is multiplication (as in Section 2.3). So template

convolution (denoted $*$) can be implemented by multiplying the Fourier transform of the template $\Im(\mathbf{T})$ with the Fourier transform of the picture, $\Im(\mathbf{P})$, to which the template is to be applied. It is perhaps a bit confusing that we appear to be multiplying matrices, but the multiplication is point-by-point in that the result at each point is that of multiplying the (single) points at the same positions in the two matrices. The result needs to be inverse transformed to return to the picture domain.

$$\mathbf{P} * \mathbf{T} = \Im^{-1}\left(\Im(\mathbf{P}).\times \Im(\mathbf{T})\right) \tag{3.4}$$

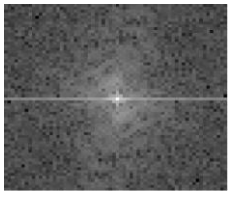
The transform of the template and the picture need to be the same size before we can perform the point by point multiplication ($.\times$). Accordingly, the image containing the template is zero-padded prior to its transform which simply means that zeroes are added to the template which lead to a template of the same size as the image. The process is illustrated in Code 3.2(a) and starts by calculation of the transforms of the image and of the zero-padded template. Then, the transform of the template is multiplied by the transform of the picture point-by-point (using the `.*` operator). (Theoretical study of this process is presented in Section 5.3.2 where we show how the same process can be used to find shapes in images.) Finally, the inverse Fourier transform is used to deliver the result. Code 3.2(b) shows an implementation in Python. This code computes the summation defining the Fourier transform by performing an iteration. First, the template is flipped and padded. Afterwards, the coefficients are obtained by performing the multiplication of the complex numbers of the image and of the template coefficients.

Code 3.2 is simply a different implementation of direct averaging. It achieves a similar result, but by transform domain calculus. The operation is shown in Figure 3.16: an image of the eye (a) is transformed to give (d); the averaging template is padded to the same size as the image (b) and transformed (e); the multiplied transforms (f) are inverse transformed to give an averaged version of the eye (c). There is one major difference between the Fourier and the direct implementations: the borders of the images differ (where the *border* is of width equal to one half of the template's width). This is because for direct averaging the border points are set to zero whereas in the Fourier implementation the image is assumed to replicate to infinity, as in Equation 2.26. (The rearrange function, Equation 2.30, is used since the padding function places the template at the centre of the image). Note that the template transform is a 2D sinc function viewed as an image and that the *logarithm* of the magnitude (Section 3.3.1) has been used to display all transforms.

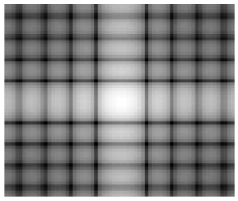
(a) image of eye

(b) padded averaging template

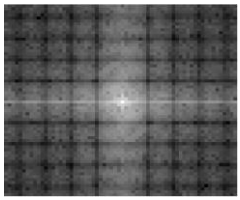
(c) resulting averaged image



(d) image transform



(e) template transform



(f) multiplied transforms

Figure 3.16 Template Convolution via Fourier Transform