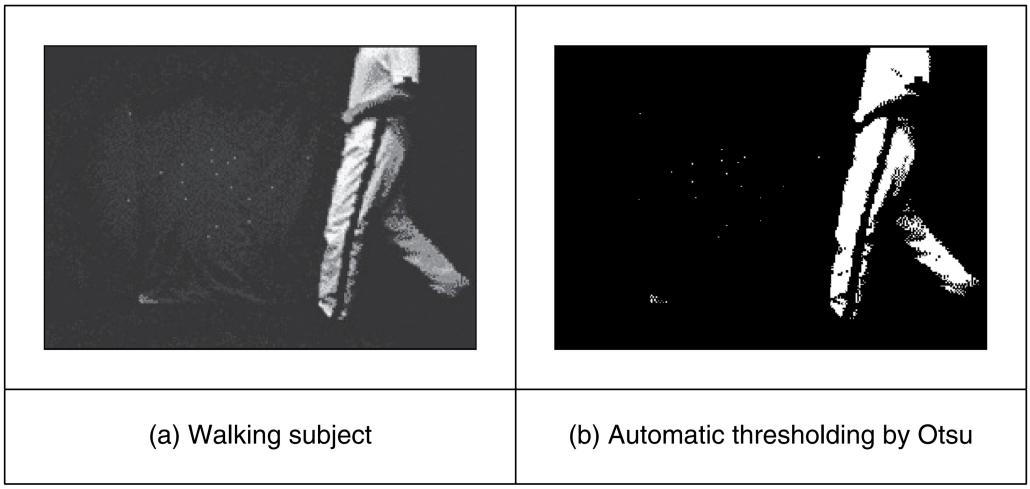

(a) Walking subject

(b) Automatic thresholding by Otsu

Figure 3.10 Thresholding an image of a walking subject

selected many objects of potential interest. As such, only uniform thresholding is used in many vision applications, since objects are often occluded (hidden), and many objects have similar ranges of pixel intensity. Accordingly, more sophisticated metrics are required to separate them, by using the uniformly thresholded image, as discussed in later chapters. Further, the operation to process the thresholded image, say to fill in the holes in the silhouette or to remove the noise on its boundary or outside, is *morphology*, which is covered in Section 3.6.

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 it can be positioned appropriately. The size is usually used to describe the template; a 3×3 template is 3 pixels wide by 3 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 a new 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.

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 for the centre pixel of the template in the original image becomes the point in the output image. Since the template cannot extend beyond the image, the new image is smaller than the original image because a new value cannot be computed for points in the border of the new image. When the template reaches the end of a line, it is repositioned at the start of the next line. For a 3×3 neighbourhood, 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.

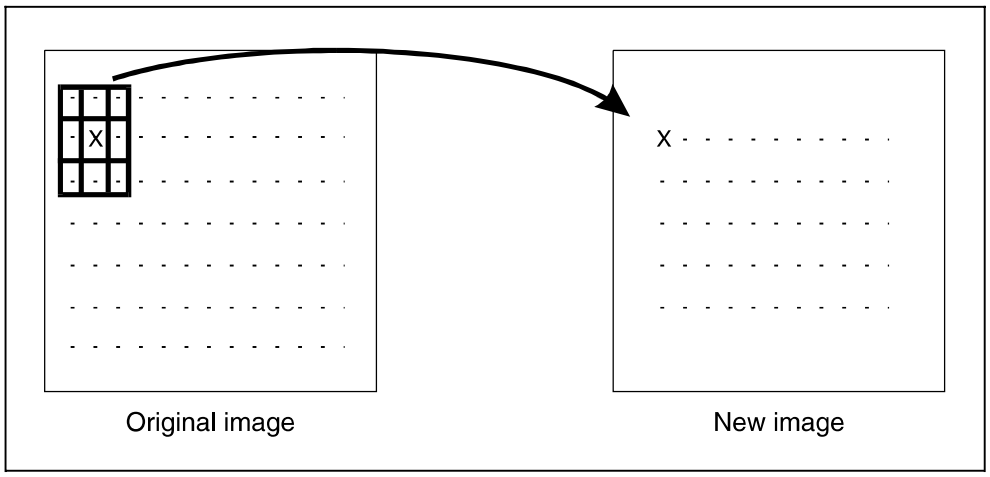

Figure 3.11 Template convolution process

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

$$\mathbf{N}_{x,y} = \begin{array}{ccccccc} w_0 \times \mathbf{O}_{x-1,y-1} & + & w_1 \times \mathbf{O}_{x,y-1} & + & w_2 \times \mathbf{O}_{x+1,y-1} & + & \\ w_3 \times \mathbf{O}_{x-1,y} & + & w_4 \times \mathbf{O}_{x,y} & + & w_5 \times \mathbf{O}_{x+1,y} & + & \forall x, y \in 2, N-1 \\ w_6 \times \mathbf{O}_{x-1,y+1} & + & w_7 \times \mathbf{O}_{x,y+1} & + & w_8 \times \mathbf{O}_{x+1,y+1} & & \end{array} \tag{3.17}$$

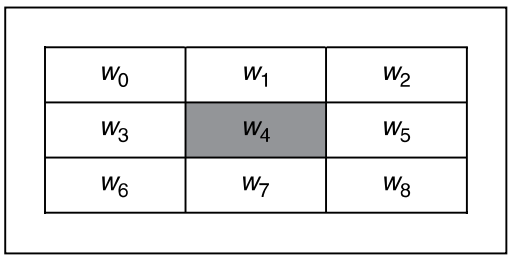

Figure 3.12 3×3 Template and weighting coefficients

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. To calculate values for the border pixels, we now have three choices:

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

None of these approaches is optimal. The results here 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

$$\mathbf{N} = \mathbf{W} * \mathbf{O} \tag{3.18}$$

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 general template convolution operator `convolve` is given in Code 3.5. This function accepts, as arguments, the picture `image` and the template to be convolved with it, `template`. The result of template convolution is a picture `convolved`. The operator first initializes the temporary image `temp` to black (zero brightness levels). Then

```
function convolved=convolve(image,template)
%New image point brightness convolution of template with image
%Usage:[new image]=convolve(image,template of point values)
%Parameters:image-array of points
% template-array of weighting coefficients
%Author: Mark S. Nixon

%get image dimensions
[irows,icols]=size(image);

%get template dimensions
[trows,tcols]=size(template);

%set a temporary image to black
temp(1:irows,1:icols)=0;

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

%then convolve the template
for x=trhalf+1:icols-trhalf %address all columns except border
 for y=tchalf+1:irows-tchalf %address all rows except border
   sum=0;
   for iwin=1:trows %address template columns
     for jwin=1:tcols %address template rows
      sum=sum+image(y+jwin-tchalf-1,x+iwin-trhalf-1)*
                        template(jwin,iwin);
     end
   end
   temp(y,x)=sum;
 end
end

%finally, normalize the image
convolved=normalize(temp);
```

Code 3.5 Template convolution operator

the size of the template is evaluated. These give the range of picture points to be processed in the outer `for` loops that give the coordinates 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. Finally, the resulting image is normalized to ensure that the brightness levels are occupied appropriately.

Template convolution is usually implemented in software. It can also 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 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 since it is presented in interlaced format.

Processing can be analogue, 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.

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 3.17, is given by the template in Figure 3.13, where

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

Figure 3.13 3×3 averaging operator template coefficients

Figure 3.14 Applying direct averaging

the location of the point of interest is again shaded. The result of averaging the eye image with a 3×3 operator is shown in Figure 3.14. This shows that much of the detail has now disappeared, revealing the broad image structure. The eyes and eyebrows are now much clearer from the background, but the fine detail in their structure has been removed.

For a general implementation, Code 3.6, we can define the width of the operator as `winsize`, the template size is `winsize` × `winsize`. We then form the average of all points within the area covered by the template. This is normalized (divided) by the number of points in the template's window. This is a direct implementation of a general averaging operator (i.e. without using the template convolution operator in Code 3.5).

```
ave(pic,winsize):= | new ← zero(pic)
                   | half ← floor(winsize/2)
                   | for x∈half..cols(pic)-half-1
                   |   for y∈half..rows(pic)-half-1
                   |     new_{y,x} ← floor( Σ_{iwin=0}^{winsize-1} Σ_{jwin=0}^{winsize-1} pic_{y+iwin-half,x+jwin-half} / (winsize·winsize) )
                   | new
```

Code 3.6 Direct averaging

To implement averaging by using the template convolution operator, we need to define a template. This is illustrated for direct averaging in Code 3.7, even though the simplicity of the direct averaging template usually precludes such implementation. The application of this template is also shown in Code 3.7. (Note that there are averaging operators in Mathcad and Matlab that can also be used for this purpose.)

```
averaging_template(winsize):= | sum ← winsize.winsize
                              | for y∈0..winsize-1
                              |   for x∈0..winsize-1
                              |     template_{y,x} ← 1
                              | template/sum
smoothed:=tm_conv(p,averaging_template(3))
```

Code 3.7 Direct averaging by template convolution

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 3×3 or 5×5, 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 that it implements

Smoothing was earlier achieved by low-pass filtering via the Fourier transform (Section 2.8). The Fourier transform gives an alternative method to implement template convolution and to speed it up, for larger templates. 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 $\Im(\mathbf{T})$ by the Fourier transform of the picture, $\Im(\mathbf{P})$, to which the template is to be applied. The result needs to be inverse transformed to return to the picture domain.

$$\mathbf{P} * \mathbf{T} = \Im^{-1}\left(\Im\left(\mathbf{P}\right) \times \Im\left(\mathbf{T}\right)\right) \tag{3.19}$$

The transform of the template and the picture need to be the same size before we can perform the point-by-point multiplication. Accordingly, the image containing the template is *zero padded* before its transform, which simply means that zeros are added to the template in positions which lead to a template of the same size as the image. The process is illustrated in Code 3.8 and starts by calculation of the transform of the zero-padded template. The convolution routine then multiplies the transform of the template by the transform of the picture point by point (using the vectorize operator, symbolized by the arrow above the operation). When the routine is invoked, it is supplied with a transformed picture. The resulting transform is reordered before inverse transformation, to ensure that the image is presented correctly. (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.)

```
conv(pic,temp):= |pic_spectrum←Fourier(pic)
                 |temp_spectrum←Fourier(temp)
                                  ⟶
                 |convolved_spectrum←(pic_spectrum.temp_spectrum)
                 |result←inv_Fourier(rearrange(convolved_spectrum))
                 |result

new_smooth:=conv(p,square)
```

Code 3.8 Template convolution by the Fourier transform

Code 3.8 is simply a different implementation of direct averaging. It achieves the same result, but by transform domain calculus. It can be faster to use the transform rather than the direct implementation. The computational cost of a 2D fast Fourier transform (FFT) is of the order of $N^2 \log(N)$. If the transform of the template is precomputed, there are two transforms required and there is one multiplication for each of the N^2 transformed points. The total cost of the Fourier implementation of template convolution is then of the order of

$$C_{\text{FFT}} = 4N^2 \log(N) + N^2 \tag{3.20}$$

The cost of the direct implementation for an $m \times m$ template is then m^2 multiplications for each image point, so the cost of the direct implementation is of the order of

$$C_{\text{dir}} = N^2 m^2 \tag{3.21}$$

For $C_{dir} < C_{FFT}$, we require:

$$N^2 m^2 < 4N^2 \log(N) + N^2 \tag{3.22}$$

If the direct implementation of template matching is faster than its Fourier implementation, we need to choose m so that

$$m^2 < 4\log(N) + 1 \tag{3.23}$$

This implies that, for a 256×256 image, a direct implementation is fastest for 3×3 and 5×5 templates, whereas a transform calculation is faster for larger ones. An alternative analysis (Campbell, 1969) has suggested that (Gonzalez and Wintz, 1987) 'if the number of non-zero terms in (the template) is less than 132 then a direct implementation . . . is more efficient than using the FFT approach'. This implies a considerably larger template than our analysis suggests. This is in part due to higher considerations of complexity than our analysis has included. There are further considerations in the use of transform calculus, the most important being the use of windowing (such as Hamming or Hanning) operators to reduce variance in high-order spectral estimates. This implies that template convolution by transform calculus should perhaps be used when large templates are involved, and then only when speed is critical. If speed is indeed critical, it might be better to implement the operator in dedicated hardware, as described earlier.

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 nine 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 while giving greater emphasis to the large structures. This is illustrated in Figure 3.15. A 5×5 operator (Figure 3.15a) retains more detail than a 7×7 operator (Figure 3.15b), and much more than a 9×9 operator (Figure 3.15c). 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

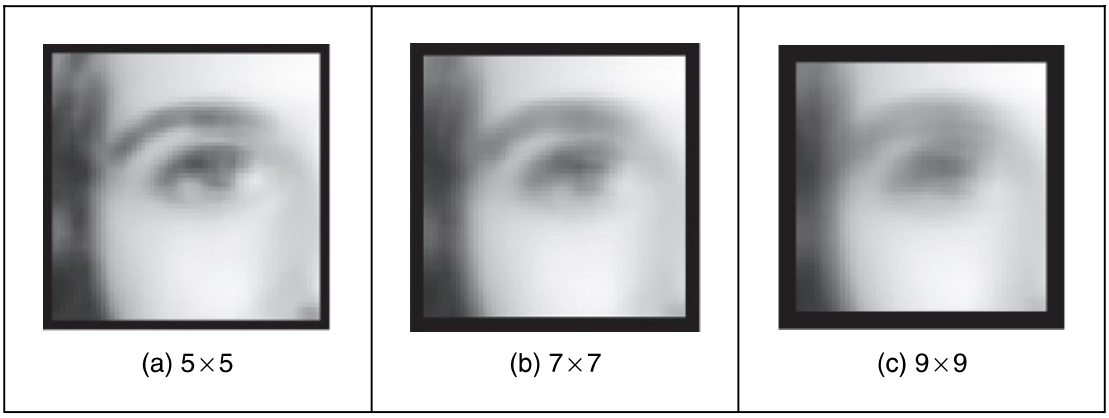

Figure 3.15 Illustrating the effect of window size

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 Figure 3.15(b) and (c).

3.4.4 Gaussian averaging operator

The *Gaussian averaging operator* has been considered to be optimal for image smoothing. The template for the Gaussian operator has values set by the Gaussian relationship. The Gaussian *function* g at coordinates x, y is controlled by the *variance* σ^2 according to:

$$g(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\left(\frac{x^2+y^2}{2\sigma^2}\right)} \tag{3.24}$$

Equation 3.24 gives a way to calculate coefficients for a Gaussian template which is then convolved with an image. The effects of selection of Gaussian templates of differing size are shown in Figure 3.16. The Gaussian function essentially removes the influence of points greater than 3σ in (radial) distance from the centre of the template. The 3×3 operator (Figure 3.16a) retains many more of the features than those retained by direct averaging (Figure 3.14). The effect of larger size is to remove more detail (and noise) at the expense of losing features. This is reflected in the loss of internal eye component by the 5×5 and the 7×7 operators in Figure 3.16(b) and (c), respectively.

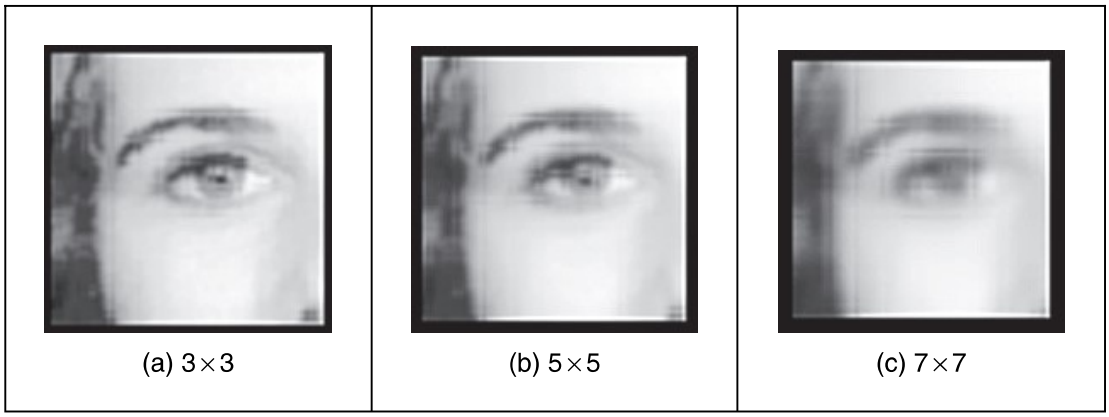

Figure 3.16 Applying Gaussian averaging

A surface plot of the 2D Gaussian function of Equation 3.24 has the famous bell shape, as shown in Figure 3.17. The values of the function at discrete points are the values of a Gaussian template. Convolving this template with an image gives Gaussian averaging: the point in the averaged picture is calculated from the sum of a region where the central parts of the picture are weighted to contribute more than the peripheral points. The size of the template essentially dictates appropriate choice of the variance. The variance is chosen to ensure that template coefficients drop to near zero at the template's edge. A common choice for the template size is 5×5 with variance unity, giving the template shown in Figure 3.18.

This template is then convolved with the image to give the Gaussian blurring function. It is possible to give the Gaussian blurring function antisymmetric properties by scaling the x and y

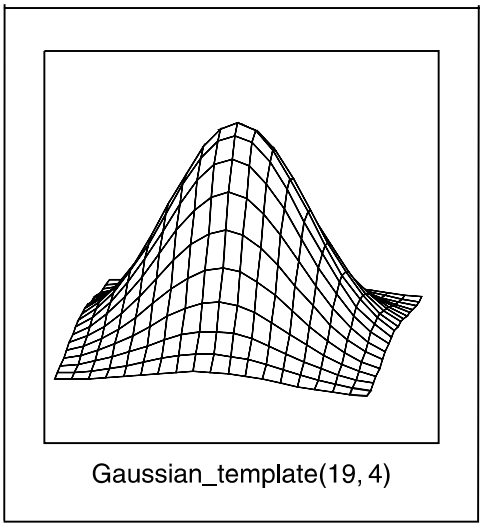

Figure 3.17 Gaussian function

0.002	0.013	0.220	0.013	0.002
0.013	0.060	0.098	0.060	0.013
0.220	0.098	0.162	0.098	0.220
0.013	0.060	0.098	0.060	0.013
0.002	0.013	0.220	0.013	0.002

Figure 3.18 Template for the 5×5 Gaussian averaging operator ($\sigma = 1.0$)

coordinates. This can find application when an object's shape, and orientation, is known before image analysis.

By reference to Figure 3.16 it is clear that the Gaussian filter can offer improved performance compared with direct averaging: more features are retained while the noise is removed. This can be understood by Fourier transform theory. In Section 2.4.2 (Chapter 2) we found that the Fourier transform of a *square* is a 2D *sinc* function. This has a non-even frequency response (the magnitude of the transform does not reduce in a smooth manner) and has regions where the transform becomes negative, called sidelobes. These can have undesirable effects since there are high frequencies that contribute *more* than some lower ones, which is a bit paradoxical in low-pass filtering to remove noise. In contrast, the Fourier transform of a Gaussian function is another Gaussian function, which decreases smoothly without these sidelobes. This can lead to better performance since the contributions of the frequency components reduce in a controlled manner.

In a software implementation of the Gaussian operator, we need a function implementing Equation 3.24, the `Gaussian_template` function in Code 3.9. This is used to calculate the coefficients of a template to be centred on an image point. The two arguments are `winsize`, the (square) operator's size, and the standard deviation σ that controls its width, as discussed earlier. The operator coefficients are normalized by the sum of template values, as before. This summation is stored in sum, which is initialized to zero. The centre of the square template is then evaluated as half the size of the operator. Then, all template coefficients are calculated by

a version of Equation 3.24 which specifies a weight relative to the centre coordinates. Finally, the normalized template coefficients are returned as the Gaussian template. The operator is used in template convolution, via `convolve`, as in direct averaging (Code 3.5).

```
function template=gaussian_template(winsize,sigma)
%Template for Gaussian averaging

%Usage:[template]=gaussian_template(number, number)

%Parameters: winsize-size of template (odd, integer)
% sigma-variance of Gaussian function
%Author: Mark S. Nixon

%centre is half of window size
centre=floor(winsize/2)+1;

%we'll normalize by the total sum
sum=0;

%so work out the coefficients and the running total
for i=1:winsize
  for j=1:winsize
    template(j,i)=exp(-(((j-centre)*(j-centre))+((i-centre)
                        *(i-centre)))/(2*sigma*sigma))
    sum=sum+template(j,i);
  end
end

%and then normalize
template=template/sum;
```

Code 3.9 Gaussian template specification

3.5 Other statistical operators

3.5.1 More on averaging

The averaging process is actually a statistical operator since it aims to estimate the mean of a local neighbourhood. The error in the process is high; for a population of N samples, the statistical error is of the order of:

$$\text{error} = \frac{\text{mean}}{\sqrt{N}} \tag{3.25}$$

Increasing the averaging operator's size improves the error in the estimate of the mean, but at the expense of fine detail in the image. The average is an estimate optimal for a signal corrupted by additive *Gaussian noise* (see Appendix 3, Section 11.1). The estimate of the mean maximizes the probability that the noise has its mean value, namely zero. According to the *central limit theorem*, the result of adding many noise sources together is a Gaussian distributed noise source. In images, noise arises in sampling, in quantization, in transmission and in processing. By the