TECHNICAL REPORT

Faster spatial image processing using partial summation

Steve Hodges and Bob Richards

CUED/F-INFENG/TR.245

January 1996

Cambridge University Engineering Department
Trumpington Street
Cambridge CB2 1PZ
United Kingdom

Email: `seh,rjr@eng.cam.ac.uk`

Faster spatial image processing using partial summation

Steve Hodges and Bob Richards

Technical Report CUED/F-INFENG/TR.245

Cambridge University Engineering Department,
Trumpington Street, Cambridge, CB2 1PZ, U.K.

January 1996

Abstract

This report describes the technique of partial summation which facilitates fast access of multi-dimensional data. Treating an image as a two-dimensional array of data, partial summation can be used to implement a variety of useful processing operations simply and very efficiently.

A number of common image processing techniques are reviewed and illustrated with examples, and typical approaches to implementation are analysed. Following a detailed description of partial summation and its application to image processing, theoretical and practical speed comparisons are made between the new and conventional approaches. To demonstrate how the ideas presented may be integrated to produce a complete solution, the specific problem of feature recognition for PCB manufacture is considered in detail.

Keywords

image processing, spatial filtering, morphology, partial summation, PCB manufacture

Contents

1 Introduction **1**

2 Review of image processing techniques **1**

2.1 Thresholding 2
2.2 Lateral histograms 4
2.3 Template matching 6
2.4 Linear spatial filtering 8
2.5 Nonlinear spatial filtering 9
2.6 Binary morphology 10
2.7 Grey level morphology 12

3 Classical approaches to efficient implementation **13**

3.1 Processor architecture 13
3.2 Basic implementation 13
3.3 Accessing the image 14
3.4 Small generating kernels 14
3.5 Overlap and save 15
3.6 Binary nonlinear filters 15
3.7 Multiresolution image processing 15

4 Partial summation **16**

4.1 One-dimensional partial summation 16
4.2 Two-dimensional partial summation 17
4.3 Memory use 18

5 Using partial sums for faster spatial image processing **19**

5.1 Lateral histograms 19
5.2 Linear filtering 20
5.3 Nonlinear filtering 21
5.4 Template matching 21
5.5 Another look at mean filtering 22

6 Speed comparisons **23**

6.1 Grey level mean filtering 23

6.2 Template matching 25

6.3 A practical application 26

7 Conclusions **26**

A Algorithm analysis **27**

A.1 Running time 27

A.2 Calculating running time 28

A.3 Memory usage 29

A.4 Growth of functions 30

B Example code **31**

B.1 Standard, array-based mean filter 32

B.2 Standard, pointer-based mean filter 33

B.3 Calculating the partial sums (pointer-based) 34

B.4 Mean filter using partial sums (pointer-based) 35

B.5 Mean filter using overlap and save (pointer-based) 36

References **38**

1 Introduction

A number of different techniques are commonly used to process digitized images by computer. Arguably the most intuitive approach involves the analysis of the *intensities* or *grey levels* of pixels with respect to their position in the image. Within this broad category of *spatial image processing*, there are many different algorithms which may be appropriate for different tasks. A number of these are reviewed and illustrated with examples in Section 2. The review is by no means exhaustive, but is included to provide a consistent basis for the discussion and analysis which follow.

A common problem with spatial image processing algorithms is that they are slow when implemented on standard hardware. Typically, computation time is proportional to both the size of the image and the size of the neighbourhood of pixels used for processing. A number of techniques may be used to enhance performance; several are outlined in Section 3. Appendix A summarizes the standard notation used throughout this report for the analysis of algorithms, and Appendix B contains example C code implementations for many of them.

Section 4 presents a technique known as *partial summation*, which is derived from multi-dimensional range searching [30]. Partial summation has been cited as a useful mechanism for pre-processing data to speed up subsequent access [26]; Section 5 shows how it may be used in two dimensions to implement many of the classical spatial image processing operations of Section 2. Implementation of partial summation is straightforward and results in faster operation; qualitative and quantitative speed comparisons are made in Section 6.

Real-time image processing has been cited as a key element in the successful implementation of many industrial vision tasks, including electronics assembly [3]. The example images used throughout this report are taken from an electronics assembly application, and depict images of *printed circuit boards* (PCBs). A PCB consists of a rigid laminate with conductive *tracks* and *pads* on its surface. The size and shape of the pads vary with different PCBs, but they are generally either round with a hole in the centre or rectangular. A *solder resist mask* is often printed over the tracks and other areas of conductor (apart from the pads), changing the visual finish of the board [23].

The report ends by considering how the techniques introduced in Sections 4 and 5 can be combined to produce an elegant yet efficient solution to the image processing problems involved in integrating a vision system with a PCB assembly machine. Reference [22] provides a more detailed description of PCBs and the issues relating to their automatic manufacture.

2 Review of image processing techniques

This section reviews several common spatial image processing techniques, and presents examples of their use. Although a variety of techniques is covered, all are classed as spatial because they rely on the analysis of pixels according to their position in an image. The review is intended to provide a background for the discussion in the following sections.

2.1 Thresholding

Thresholding is one of the most straightforward image processing operations, and also one of the most important approaches to image segmentation [19]. In its simplest form it involves comparing each pixel in the original image I with a predetermined threshold t to create a new image I_{new}. The new pixel is set to one of two values, depending on whether or not the intensity of the corresponding pixel in the original image is less than the threshold [33]. For an image of size $X \times Y$ pixels[1]

$$I_{new}(x,y) = \begin{cases} 1, & I(x,y) \geq t \\ 0, & I(x,y) < t \end{cases} \qquad x \in \{0,1,\ldots,X-1\},\ y \in \{0,1,\ldots,Y-1\}. \quad (1)$$

The processed image which is created is the same size as the original; it is known as a *binary image* since each pixel has one of two values. This process, often termed *binarization*, reduces the complexity of the image by removing what is hopefully redundant information. Many image processing tasks involve *object segmentation*, i.e. differentiating the silhouettes of different objects both from each other and from a contrasting background. This can be a very difficult task [9], but thresholding provides a simple solution when the intensities of the objects and background differ substantially, and are therefore grouped into distinctive modes. The resulting binary image may subsequently be processed further.

The *grey level histogram* of an image can be useful for automatically choosing threshold values for binarization. It is simply a histogram showing the number of pixels in the image which are set to each possible grey level. Figure 1 shows an example image and its histogram. The bi-modal histogram shows that the image is largely comprised of two distinct brightnesses, therefore selecting a threshold is straightforward. Figure 1(b) shows the result of applying the threshold.

If the histogram is multi-modal, it is possible to use a series of thresholds, and assign each pixel to one of a number of different states. This can be particularly useful for segmenting a number of heterogeneous objects which are touching or overlapping, although it is generally less reliable than using a single threshold because of the difficulty of establishing multiple thresholds that isolate regions of interest effectively [19].

Even selecting a single threshold robustly can be more difficult than in the example of Figure 1 for a number of reasons: in real applications, noise is invariably present in the image, although a number of techniques may be used to overcome the problems this introduces (see Section 2.5 for example). Another problem often encountered is non-uniform illumination [41, 43], in which the brightness of the image varies due to lighting effects as well as the scene in the field of view. This means that a single threshold value cannot be used throughout the entire image; Figure 2 shows an example of this. In this case, it becomes necessary either to normalize the image before processing [27, 29], or to use an adaptive form of thresholding, where the threshold value is calculated following analysis of the intensities of nearby pixels [9, 13]. Both of these approaches tend to be computationally expensive, and invariably some information is still lost [41].

[1] In this report, the origin of an image is considered to be the top left-hand corner, and is denoted $I(0,0)$. The bottom right-hand corner of an image of size $X \times Y$ will therefore be $I(X-1, Y-1)$.

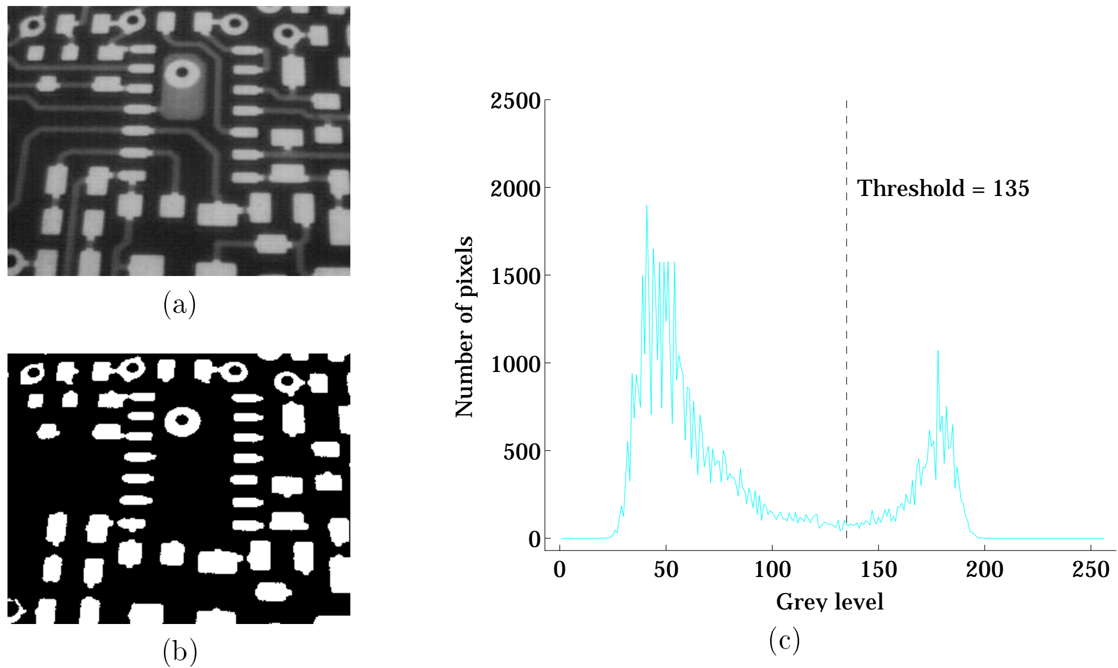


Figure 1: (a) Example image showing a section of printed circuit board (PCB). Each pixel is represented by an 8 bit number and therefore has a value between 0 (black) and 255 (white). (b) Thresholding has successfully segmented the pads from the rest of the PCB. (c) The grey level histogram of the image, showing a bi-modal distribution. The threshold grey level of 135 was selected as the local minimum of the smoothed distribution.

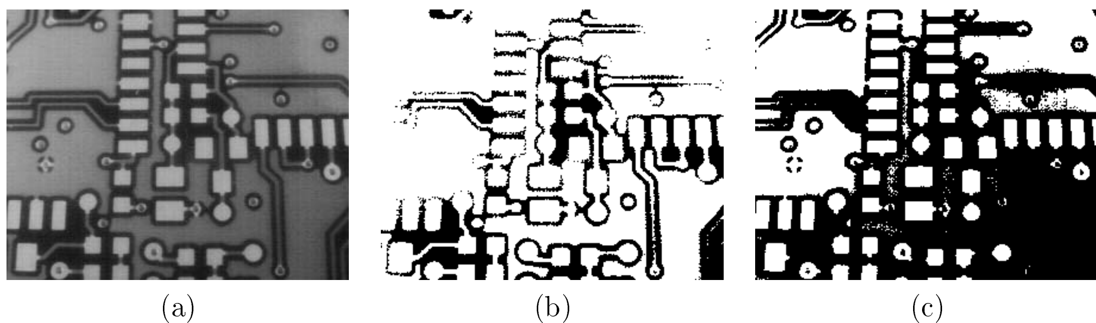


Figure 2: (a) A non-uniformly lit image. (b) & (c) The result of thresholding the image with low and high thresholds in an attempt to segment the areas of copper conductor from the bare PCB; areas are wrongly classified in each case.

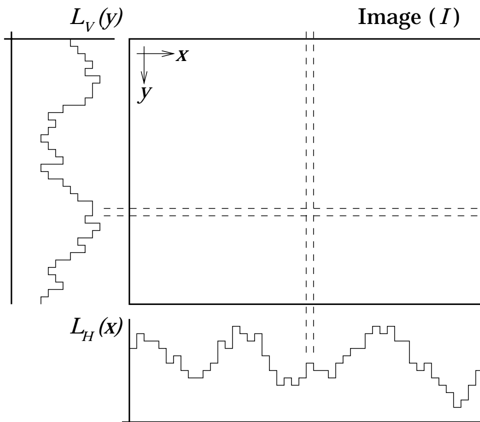


Figure 3: The lateral histograms are calculated by summing the grey levels of the pixels along the rows and columns of the image. This takes just $\Theta(XY)$ operations for an image of size $X \times Y$.

2.2 Lateral histograms

The *lateral histogram*[2] [13] is not strictly an image filtering operation, because it does not generate an image as its output. However, like the grey level histogram, it is a useful technique for analysing images prior to further processing operations. It involves projecting an image onto two (or more) axes by summing pixel intensities along lines perpendicular to each axis. The vertical and horizontal lateral histograms (L_V and L_H respectively) of an $X \times Y$ image I are therefore calculated by summing along the rows and columns respectively (see Figure 3):

$$\begin{aligned} L_V(y) &= \sum_{x=0}^{X-1} I(x,y), \quad y \in \{0,1,\ldots,Y-1\}. \\ L_H(x) &= \sum_{y=0}^{Y-1} I(x,y), \quad x \in \{0,1,\ldots,X-1\}, \end{aligned} \tag{2}$$

Analysis of the histograms allows objects in the image and their locations to be identified by their 'silhouettes'. This is usually achieved by examining the intersections of the horizontal and vertical lines corresponding to particular features in each histogram. This process is particularly attractive for two reasons: firstly, in the presence of (random) noise, the summation has the effect of reducing the variance due to noise and is therefore more robust [32, 35, 45]; secondly, it is computationally efficient—calculating the histograms only takes $\Theta(XY)$ operations[3] for an image of size $X \times Y$. If the image contains P objects, it is possible that each will generate a feature in both the vertical and horizontal lateral histograms. This means there will be at most P^2 candidate locations, each of which must be checked by some mechanism (such as template matching—see Section 2.3) to confirm or reject the presence of an object at that position. An example of this use of lateral histograms to highlight the positions of a given feature in an image is shown in Figure 4. If the image contains a large number of objects or is generally cluttered, it is often beneficial to employ a divide-and-conquer strategy by generating lateral histograms of *sub-images*, and then combining the results from these.

[2]Lateral histograms are also known as *density histograms* [36], and *projections* [45].

[3]The use of Θ- and O-notation is discussed in Appendix A.

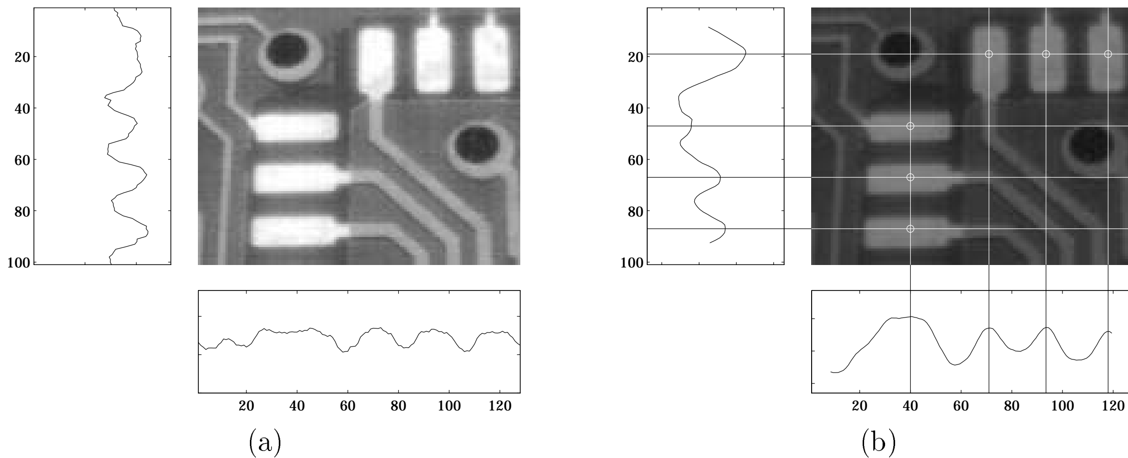


Figure 4: (a) The example image shows part of a PCB including six rectangular pads. The two lateral histograms have been calculated using Equation 2. (b) The histograms have been smoothed, and the local maxima projected onto the image (which has been darkened for clarity). The intersections indicate candidate pad locations, and the circles show successful matches.

In addition to locating entire objects, lateral histograms have been used to detect object features. For example, discontinuities in the histograms can be used to detect object corners. Once again, it is computationally much simpler to do this and then check candidate locations for the presence of actual corners, than to perform conventional corner detection on the entire image [45].

As mentioned in Section 2.1, non-uniform lighting is a common problem in the application of industrial machine vision. One solution to this problem is to use the lateral histograms to estimate local average intensities throughout the image and use these to normalize the grey levels to a certain extent. Figure 5 shows the PCB of Figure 2(a) along with the result of normalizing the image in this manner, and the resulting improvement in subsequent thresholding.

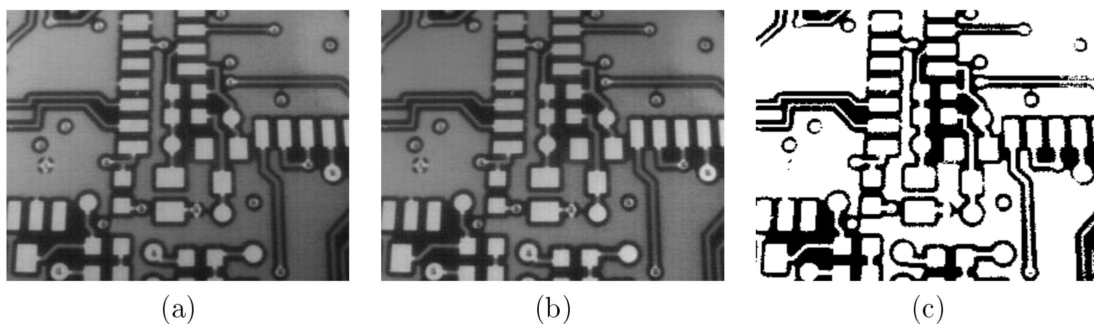
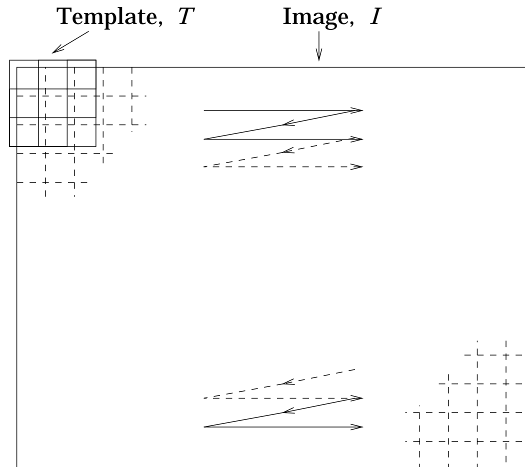


Figure 5: (a) A non-uniformly lit image. The PCB appears slightly brighter towards the top left- and top right-hand corners (b) Image normalized with respect to lighting by lateral histogram analysis. (c) Subsequent thresholding segments copper from bare PCB much more effectively over the whole image (compare with Figure 2 on page 3).

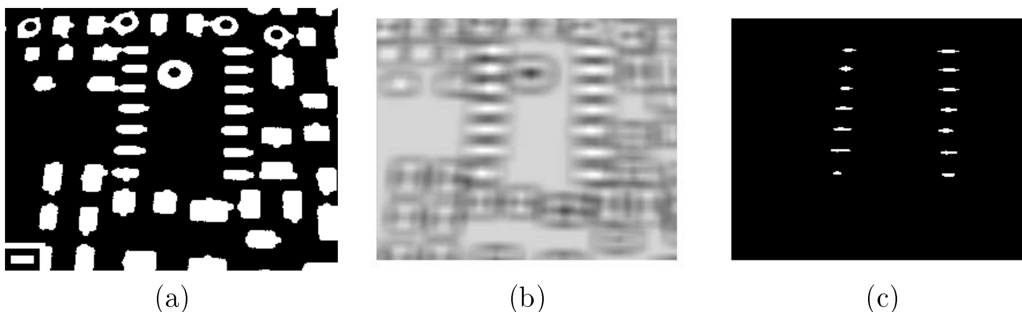
Figure 6: Template matching involves passing a template (T) over each image pixel in turn. At each position, a similarity measure is calculated based on the values in the template and the corresponding image pixel values. In this example T is a 3×3 template.



2.3 Template matching

Template matching or *prototype matching* is an intuitive method which can be used to detect a particular object or feature in an image. The template is effectively a sub-image which contains only the object to be detected, exactly as it should appear in the image. The template is moved over the image, usually in a raster fashion, so that it aligns with each pixel in the image in turn (see Figure 6). At each location a similarity (or difference) measure is computed to determine how well the image data matches the template. The array of similarity values which is built up as the template is passed over the image may itself be displayed as an image—the best matches result in high similarity values and bright areas in the image. Template matches are rarely exact because of noise, quantization effects and variations in the exact shape and structure of the object to be detected [29]. However, if the match is sufficiently close, it is assumed that the object in question is present at that location.

The simplest form of template matching uses a *binary template*, which represents the shape of the object against a constant background (see Figure 7(a)). The similarity between the



(a) (b) (c)

Figure 7: (a) An example binary image (from Figure 1), with template inset in the bottom left-hand corner. (b) The result of binary template matching the image with the template, using the number of matching pixels as the similarity measure. (c) Thresholding highlights the positions of the features which match the template.

template and the corresponding part of the image at each location is simply defined as the number of pixels which match[4] [16]. The background which is included in the template is important for successful object location—without it, matches would be made with any large bright region of the image. Figure 7 gives an example of binary template matching.

Binary template matching may be readily extended to grey level images [16]. Assuming a rectangular (to keep the analysis straightforward) template T of size $R \times S$, matching may be performed by using the *Euclidean distance* E as a difference measure:

$$E(x,y) = \sqrt{\sum_{s=0}^{S-1} \sum_{r=0}^{R-1} \Big(I(x+r\,,\, y+s) - T(r,s) \Big)^2}, \qquad \begin{array}{l} x \in \{0,1,\ldots,X-R\}, \\ y \in \{0,1,\ldots,Y-S\}. \end{array} \tag{3}$$

It is often convenient to remove the square root and consider the measure of distance to be $E^2(x,y)$. Expanding the quadratic term gives

$$E^2(x,y) = \sum_{s=0}^{S-1} \sum_{r=0}^{R-1} \Big(I^2(x+r\,,\, y+s) \; - \; 2\,I(x+r\,,\, y+s)\,T(r,s) \; + \; T^2(r,s) \Big). \tag{4}$$

The $\sum\sum T^2(r,s)$ term is the sum-squared intensity or *picture energy* of the template. This does not vary over the image and can therefore be ignored. A further simplification occurs if the $I^2(x+r, y+s)$ term, representing the image picture energy over the area of the template, is ignored. This leaves a more approximate similarity measure, the *cross-correlation* between the template and the image, which will then be maximized[5] when the portion of the image 'under' the template is identical to the template up to a constant factor[6]:

$$C(x,y) = \sum_{s=0}^{S-1} \sum_{r=0}^{R-1} I(x+r, y+s)\,T(r,s), \qquad \begin{array}{l} x \in \{0,1,\ldots,X-R\}, \\ y \in \{0,1,\ldots,Y-S\}. \end{array} \tag{5}$$

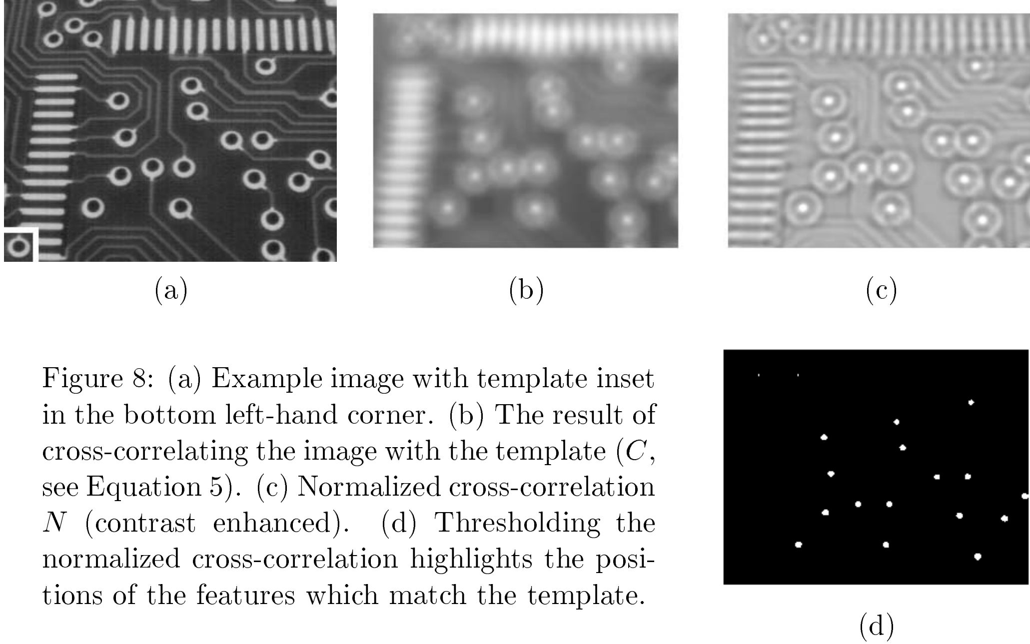
The assumption that the image energy within the template area is roughly constant throughout the image and can therefore be ignored is often not valid—a bright area in the image will heavily influence the calculation and produce false matches. One solution to this problem is to normalize the correlation in some way; several heuristics can be used to calculate the *normalized cross-correlation* N [16, 19, 29], for example

$$N(x,y) = \frac{\sum_{s=0}^{S-1} \sum_{r=0}^{R-1} I(x+r\,,\, y+s)\,T(r,s)}{\sum_{s=0}^{S-1} \sum_{r=0}^{R-1} I^2(x+r\,,\, y+s)}, \qquad \begin{array}{l} x \in \{0,1,\ldots,X-R\}, \\ y \in \{0,1,\ldots,Y-S\}. \end{array} \tag{6}$$

[4] Alternatively, matching may be performed by calculating the difference between the template and the corresponding area of the image. This results in an array of differences, where the minima represent good matches. In the case of binary template matching, Hamming distance provides a suitable difference metric.

[5] Maximizing the cross-correlation (a similarity measure) is equivalent to minimizing the distance (a difference measure). This change occurs because the -2 coefficient is dropped from the $I(x+r, y+s)T(r,s)$ term in Equation 4.

[6] Template matching can therefore be considered as cross-correlation between the image and the template, which is itself an image of the object or feature to be matched. Since correlation is essentially convolution with a reflected template [4], it is possible to perform the template matching in the frequency domain instead of the spatial domain, capitalizing on the convolution theorem and fast Fourier transformation. As a rough guide, it has been shown that for templates bigger than around 13×13 pixels, this is a more efficient approach [19].



(a) (b) (c)

Figure 8: (a) Example image with template inset in the bottom left-hand corner. (b) The result of cross-correlating the image with the template (C, see Equation 5). (c) Normalized cross-correlation N (contrast enhanced). (d) Thresholding the normalized cross-correlation highlights the positions of the features which match the template.

(d)

Figure 8 gives an example of grey level template matching. Note that the pads in the top left-hand corner of the image are smaller than that in the template, generating a poor match. Coping with the changes in size and orientation which can occur in industrial environments is difficult, limiting the applications in which template matching of this form is useful. When a crude estimate of the orientation and size of the object is known, perhaps from an earlier stage in an industrial process, this technique can be useful [34]; it is seldom used when arbitrary or unconstrained rotation is present [19].

2.4 Linear spatial filtering

The problems with template matching outlined at the end of the previous section can be alleviated to some extent if the nature of the template is modified. Instead of trying to match an entire object, localized templates can be used to locate lower-level features such as edges [16]. A number of templates will be needed to deal with the different edge orientations, but since each template can be much smaller (e.g. 3×3, 5×5), the computational requirements are often reduced. Mathematically, the template is applied using a cross-correlation process (Equation 5), although this is often simply referred to as *linear spatial filtering*. Similarly, the template is often referred to as a *spatial filter*, *mask* or *kernel*. Once again it may be any size or shape, but is frequently square. The new, filtered image is defined as

$$I_{new}(x,y) = \sum_{s=0}^{S-1} \sum_{r=0}^{R-1} I(x+r\,,\; y+s)\, K(r,s), \qquad \begin{array}{l} x \in \{0,1,\ldots,X-R\}, \\ y \in \{0,1,\ldots,Y-S\}, \end{array} \tag{7}$$

where K is an $R\times S$ spatial kernel which is effectively used to form a weighted sum of image pixels. Refer back to Figure 6 on page 6 to visualize how a 3×3 kernel might be applied. Although linear spatial filtering is technically a cross-correlation process, it is equivalent to convolution, a term often used in the literature.

(a)

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

(b)

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

(c)

1	2	1
0	0	0
-1	-2	-1

(d)

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

(e)

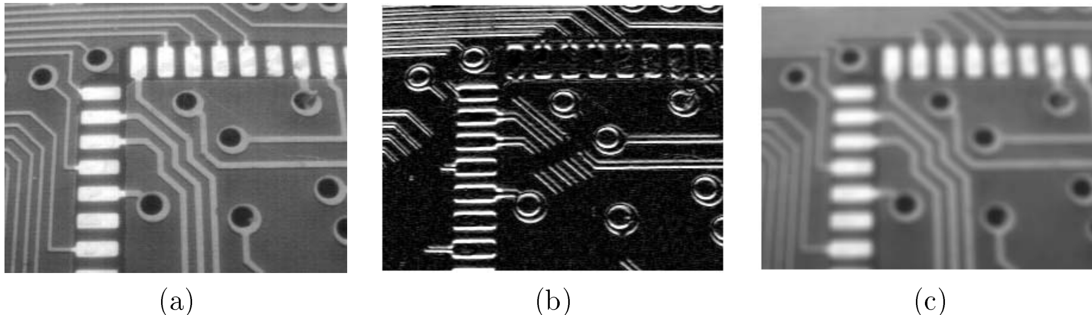
$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$
$\frac{2}{16}$	$\frac{4}{16}$	$\frac{2}{16}$
$\frac{1}{16}$	$\frac{2}{16}$	$\frac{1}{16}$

Figure 9: Example 3×3 filters. (a) & (b) Simple horizontal and vertical edge detectors. (c) Horizontal Sobel edge detector. (d) Mean filter. (e) Gaussian smoothing filter.

Figures 9(a) and (b) show an example pair of edge detecting filters—one kernel is used to detect horizontal edges, and a second to detect vertical edges. The results of the two filtering operations (which must be applied separately to the original image) can be combined to form a single image by calculating the edge magnitude at each location. Figure 9(c) shows a horizontal *Sobel* edge detector. This modified form is slightly better at detecting edges under most circumstances; Figure 10 gives an example of its use. The edge detector is a *high pass* linear filter; that is, high frequencies in the image such as edges are passed through the filter whilst low frequencies (such as areas of uniform brightness) are filtered out. By using a different kernel, the converse operation of *low pass* filtering is also possible. The most obvious example is the classical mean filter [14]—this is a $K\times K$ kernel where every element is $1/K^2$ (Figure 9(d)). This filter calculates the arithmetic mean of the grey levels under the kernel, and has the effect of smoothing the image—high frequency components of the image are removed. Figure 10(c) shows the result of applying a 7×7 mean filter to an image. Another smoothing filter which is often more effective, uses a Gaussian kernel such as that in Figure 9(e). As the size of the kernel is increased, the filter cut-off frequency is reduced and the effect is more marked.

2.5 Nonlinear spatial filtering

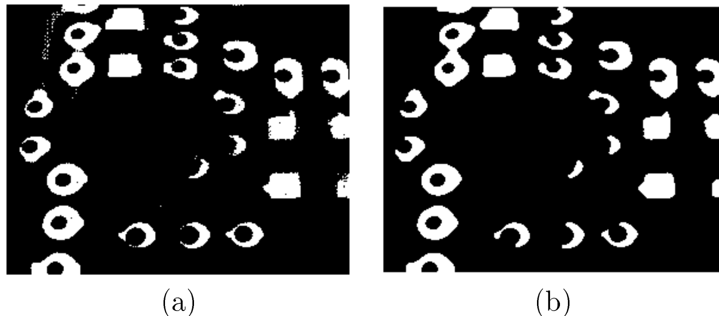
Like the linear operators discussed above, nonlinear spatial filters also operate on neighbourhoods. However, the response of a general spatial filter is defined by a nonlinear



(a) (b) (c)

Figure 10: (a) An example image. (b) The effect of a 3×3 horizontal Sobel edge detecting filter on the example image. (c) The result of applying a 7×7 mean filter to the same image.

Figure 11: (a) Noisy binary image. (b) The effect of applying a 5×5 median filter.



(a) (b)

function f of the values in the neighbourhood concerned:

$$I_{new}(x,y) = f(I(x,y), I(x+1,y), \ldots, I(x+R-1, y+S-1)), \qquad \begin{array}{l} x \in \{0,1,\ldots,X-R\}, \\ y \in \{0,1,\ldots,Y-S\}. \end{array} \tag{8}$$

A common example is the *median* filter, the response of which is defined to be the arithmetic median of the grey levels in the neighbourhood. This is useful for removing noise from an image, since noisy pixels are generally towards one extreme of the distribution of grey levels within the neighbourhood. Median filtering is often used on binary images; Figure 11 gives an example of such use. Unlike mean filtering, median filtering does not have a blurring effect.

A useful subset of nonlinear functions can be modelled using a framework known as *mathematical morphology* [6], by considering the interaction between the pixels in the neighbourhood and the values in a kernel. Mathematical morphology is based on geometry and shape, rather than a linear transformation such as convolution. The next two sections describe morphological operations in detail.

2.6 Binary morphology

The basis of mathematical morphology is set theory [19]. In binary morphology, a (binary) image is represented as the *point set* (say P) of all the pixels in the image which are 'on'—each member of the point set is a coordinate pair which represents the position in the image of the corresponding pixel.[7] This representation of an image facilitates morphological image processing. P may be transformed by applying a *structuring element* S, which is itself a small point set, expressed with respect to a local origin (called the *representative point*).

The two most common morphological operations are that of *dilation* and *erosion*. Dilation (sometimes called *fill*, *grow* or *expand* [37]) combines the two sets (P and S) using vector addition[8]—the dilation $P \oplus S$ is the set of all possible vector combinations of pairs of members, one from each of the sets P and S:

$$P \oplus S = \{y \mid \exists\, p \in P,\ s \in S : y = p + s\}. \tag{9}$$

[7]The origin $(0,0)$ is usually at the top left-hand corner of the image.

[8]Each member of P and S, which is a coordinate pair representing the position of the corresponding pixel in the image, can also be thought of as a vector. Vector addition (sometimes called *Minkowski set addition*) is performed by summing the x coordinates and the y coordinates separately to give a new coordinate pair.

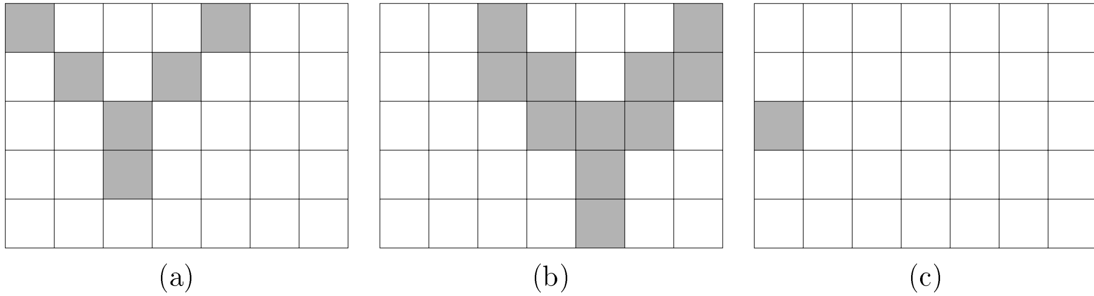


Figure 12: (a) A simple example image. (b) The result of dilating the image in (a) with the structuring element of Figure 13(a). (c) The result of eroding the image in (a) with the structuring element of Figure 13(a). The origin $(0,0)$ is the top left-hand corner of the image in each case.

For example, if the original image is that shown in Figure 12(a) and the structuring element in Figure 13(a) is used, then

$$\begin{aligned} P &= \{(0,0),(1,1),(2,2),(2,3),(3,1),(4,0)\} \qquad \text{and} \\ S &= \{(2,0),(2,1)\}. \end{aligned}$$

The result of dilating P by S is

$$P \oplus S = \{(2,0),(2,1),(3,1),(3,2),(4,2),(4,3),(4,4),(5,1),(5,2),(6,0),(6,1)\}.$$

which is illustrated in Figure 12(b). In practice, a square structuring element is commonly used; see Figure 13(c) for example. This has the effect of expanding objects in the original image by augmenting each pixel with a number of pixels in its neighbourhood, thereby filling holes in objects and smoothing edges. The dual of dilation is *erosion* (also referred to as *shrink* or *reduce*) [39, 37]. The result of the erosion of a set P (the original image) by S is given by those points y for which all possible $y + s$ are in P [37]; an example of erosion is given in Figure 12(c). Formally, the erosion $P \ominus S$ is defined as

$$P \ominus S = \{y \mid y + s \in P, \quad \forall\, s \in S\}. \tag{10}$$

The two basic morphological operators, dilation and erosion, can be combined in a number of useful ways. Dilation followed by erosion has the effect of *closing* the image (denoted by $\bullet$):

$$P \bullet S = (P \oplus S) \ominus S. \tag{11}$$

Closing connects objects that are close to each other, fills up small holes, and smooths object edges (by filling up narrow gulfs).[9] However, unlike dilation or erosion used in isolation, closing does not significantly change object sizes [37].

[9]Note that dilation and erosion are not in general the inverse of each other, i.e. $P \bullet S \neq P$.

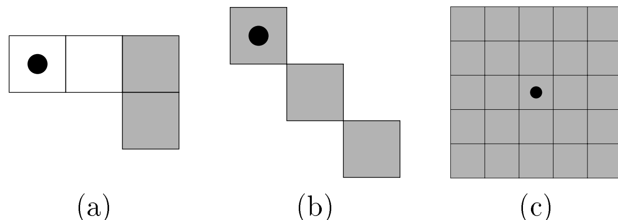


Figure 13: Example structuring elements. The black circle indicates the origin, whilst the shaded squares indicate members of the point set; for example (a) would be represented by $\{(2,0),(2,1)\}$.

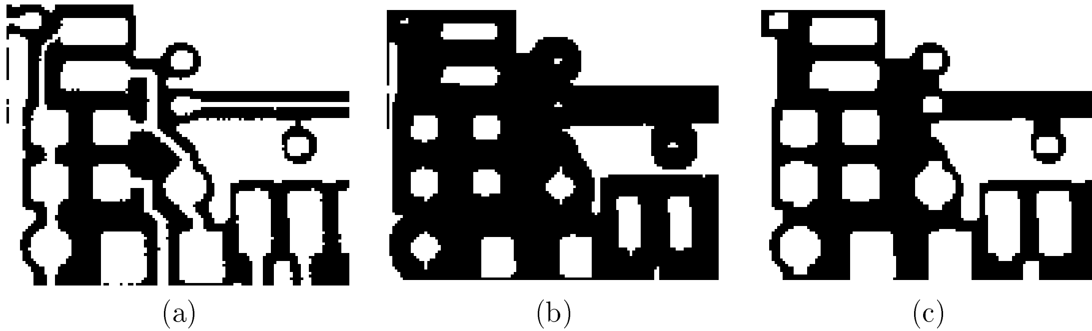


Figure 14: An example of morphological opening. (a) A binary image of part of a PCB (from Figure 5 on page 5). (b) The result of eroding the image in (a) with the structuring element of Figure 13(c): finer details, such as the tracks, are removed. (c) The eroded image in (b) is dilated by the same structuring element. The complete process from (a) to (c) is opening; note that the pads remain roughly their original size, but the tracks are removed.

Opening an image P with a structuring element S is denoted by $P \circ S$ and is defined as

$$P \circ S = (P \ominus S) \oplus S. \tag{12}$$

This has the effect of removing small objects and fine details from an image. An example of its use is given in Figure 14, where the 5×5 structuring element of Figure 13(c) is applied to the binary image of a PCB. The 'coarseness' of the operation can be varied by changing the size of the structuring element, or alternatively by eroding the image n times before dilating it a further n times.

If used, morphological transformations usually constitute an intermediate part of the image processing sequence. First the image is pre-processed and segmented to obtain a binary image with objects separated from the background (see Section 2.1 on page 2 for example). Binary morphological operators may then be applied to the shape of these objects, before a final step is used to evaluate the results of the processing [37].

2.7 Grey level morphology

It is possible to extend the ideas of binary morphology outlined above to grey level images. This extension effectively involves the addition of a third dimension to the data—the image can be thought of as a surface in 3D space, where the height of the surface above the x, y plane represents the grey level intensity of the image at that location [38]. The set representation of an image now contains all the points in the *umbra* of the function, that is, all the points which lie on or below the surface [39]. The structuring element may be binary or it may also be a grey level sub-image. In either case, it too is represented by the set of points in its umbra.

Morphological operations such as dilation and erosion may be applied as before, by vector addition and subtraction of the set elements. Opening and closing a grey scale image can be achieved by sliding a three-dimensional structuring element over the surface representing the image brightness [38].

Figure 15: Factoring out a common divisor reduces computation time considerably. In this case, a single shift right (by four bits) achieves the required division.

$$\begin{array}{|c|c|c|}\hline \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \hline \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \hline \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \hline \end{array} = \frac{1}{16} \cdot \begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$$

3 Classical approaches to efficient implementation

Section 2 reviewed a number of common image processing techniques, explaining the theory behind them and giving examples of their use. When implementing these algorithms in practical applications, it is often sensible to optimize for fast execution. This section briefly covers approaches commonly used for efficient implementation and analyses the resources they use. Example code for many of the algorithms is presented in Appendix B.

3.1 Processor architecture

As Section 2 has demonstrated, a large number of image processing operations are based on some form of spatial filtering. Since spatial filters operate on a restricted neighbourhood of pixels, they are particularly suited to implementation on parallel processors [17]. However, in the majority of cases image processing algorithms are implemented on standard hardware, using a sequential programming language. The analysis which follows assumes that a single processor, Von Neumann-type machine is to be used.

Some hardware platforms perform integer calculations more quickly than floating point operations[10] so it makes sense to choose filter elements accordingly. In any case, calculations can be saved by factoring out, as shown in Figure 15. Indeed, if the multiplicands or divisors are powers of two it is possible to avoid multiplication or division operations entirely by using the appropriate binary shift left or right respectively [10].

3.2 Basic implementation

The simplest approach to implementation is to use a nested loop to pass the filter over the entire image (Figure 6). The new pixel value at each position ($I_{new}(x, y)$) is usually calculated using a further nested loop which passes along and down the elements of the kernel. Processing time is therefore proportional to both the size of the image and the size of the kernel, i.e. $\Theta(XYK^2)$.[11]

The size of the filtered image is $(X-(K-1)) \times (Y-(K-1))$, where $X \times Y$ is the size of the original image, and a $K \times K$ sized kernel is used.[12] Usual practice is to create a new image in a separate area of memory as the filtering is carried out, otherwise the original pixel values from one line will not be available for filtering subsequent lines. This means that $M(X,Y) = XY + (X-K+1)(Y-K+1) \approx 2XY$ (since $X, Y \gg K > 1$).

[10]This is not always the case, especially with more modern processors. The DEC Alpha has a floating point unit which can execute one multiply per clock cycle, faster than its integer multiplication.

[11]Θ, O, T and M notation are defined in Appendix A.

[12]The kernel is usually square, with odd dimensions, so that a single central pixel exists.

Figure 16: Decomposition of a 2D Gaussian filter into two 1D filters.

$$\begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array} = \begin{array}{|c|}\hline 1 \\ \hline 2 \\ \hline 1 \\ \hline \end{array} * \begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline \end{array}$$

3.3 Accessing the image

The most obvious mechanism in most programming languages for accessing elements of both the image and the filter is the array. This allows very straightforward implementation (see Section B.1). However, it also entails repeated re-calculation of memory addresses to access each element. Since access to the various elements is sequential, a more efficient (albeit less elegant) solution, is to use a mechanism such as pointers (provided in many languages, including C). Consecutive memory accesses may then be made by simply incrementing the pointer (just one machine instruction on most processors), instead of the more complex array offset calculation. Appendix B contains examples of both the array and pointer versions of the basic mean filter.

3.4 Small generating kernels

For a linear operator (Equation 7, page 8) the execution time is $\Theta(XYK^2)$ where a $K\times K$ sized kernel is applied to an $X\times Y$ image. Nonlinear operations may be slower, depending on their nature—for example median filtering, which relies on sorting values to find the median, will be $\Theta(XYK^2 \log^2 K)$. The size of the kernel is therefore in general a critical factor in the speed of execution of a filtering operation. If the size of the kernel could be reduced, the filtering operation would be much faster.

Consider the example of smoothing an image with a 3×3 Gaussian kernel. As Figure 16 shows, it is possible to express this two-dimensional Gaussian as a convolution of two one-dimensional Gaussians. Since convolution is associative, the required smoothing can be achieved by applying the two 1D filters in turn. This results in faster execution—the process is now linear in K, i.e. $\Theta(XYK)$. For a 3×3 kernel such as in this example, the speed-up is slight. However, as the kernel size increases, the reduction in execution time is increasingly significant.

This process is called *small generating kernel (SGK) convolution* [1, 29]. The decomposition of a two-dimensional Gaussian into two one-dimensional SGKs is a special case, which is particularly straightforward. In general, it is much more difficult (in some cases impossible) to break down an arbitrary spatial filter F into a number of smaller filters. However, techniques have been developed for choosing SGKs to approximate F as closely as possible in a least squares sense [29]. Thus a $K\times K$ filter F can be approximated by several[13] 3×3 kernels to provide a useful speed-up. If these kernels are denoted F_n and convolution by $*$, then

$$\hat{F} = F_1 * F_2 * \ldots * F_N. \tag{13}$$

[13] In fact, there will be N kernels, where $N = (K-1)/2$ [29].

A similar technique may be used for some nonlinear filters [13]. In this case, the filters required to emulate the desired operation will generally need to be calculated heuristically, and their effect will only be an approximation. Examples of this approach to median filtering are given in [13] and by the pseudo-median filter of [29]. Morphological structuring elements can be decomposed in a way similar to that of small generating kernels [46]. Since dilation is associative and commutative, if a structuring element S can be expressed as

$$S = S_1 \oplus S_2 \oplus \cdots \oplus S_N, \tag{14}$$

then it holds that

$$(((P \oplus S_1) \oplus S_2) \cdots) \oplus S_N = P \oplus (S_1 \oplus S_2 \oplus \cdots \oplus S_N) = P \oplus S. \tag{15}$$

This also holds for erosion. The problem of decomposing a given structuring element S into its constituents, $S_1, S_2, \cdots, S_N$ is addressed in [46].

3.5 Overlap and save

In practice, many spatial filters are symmetrical in one or both dimensions which means that much computation is repeated. It is possible to speed up the operation of certain filters by eliminating such repetition. The technique of *overlap and save* (after [1]), is implemented by storing intermediate results for re-use later. The most obvious example is perhaps a $K \times K$ mean filter (such as that in Figure 9(b) on page 9). Rather than completely recalculating the response of the filter at each location in the image, it is possible to alter the response from the previous location,[14] hence saving repeated calculation.

This has little effect on $M(X, Y)$, but effectively turns $T(X, Y, K)$ into $\Theta(XYK)$, i.e. linear with respect to the size of the kernel, K. Example code to implement this is given in Appendix B. If the filter is less symmetrical, this optimization is less appropriate—in the case of template matching for example, no saving can be made.

3.6 Binary nonlinear filters

Many nonlinear filtering operations are intrinsically very computationally intensive. For example, median filtering an $X \times Y$ grey level image using a $K \times K$ kernel is $\Theta(XYK^2 \log^2 K)$ since the image elements must be sorted. However, if a binary image is being processed, simplification is often possible. The median of a set of binary pixels can be calculated by simply counting the number of pixels which are set. If this is more than one half of the total number of pixels, the median must also be a set pixel. Similarly, erosion and dilation can be implemented by counting the number of pixels set at each structuring element location.

3.7 Multiresolution image processing

The basis of *multiresolution* or *hierarchical* image processing is to perform the processing operation (whatever it may be) at a number of resolutions (or *scales*). Usually a coarse

[14] This is done by subtracting elements from the left of the filter and adding elements to the right.

scale is used initially, and the resolution is increased on subsequent passes. For example, if a feature detector is applied at a number of scales, then increasingly more detailed information relating to the locations of such features in the image will be generated. The advantage of such a scheme is two-fold: not only is it possible to detect such features at different scales in the image, but the processing required at lower resolutions is generally less demanding whilst being more robust to changes in orientation and size of the feature to be matched.

Sometimes a coarse match is all that is required. However, a common practice is to use a low resolution first pass through the image to highlight areas of interest, followed by a high resolution phase to analyse these areas in more detail [15, 18, 28, 44]. This eliminates much of the redundant processing involved in detailed analysis of the entire image. Sometimes it is more appropriate to use different image processing techniques at the various resolutions. For example, a number of candidate locations can be highlighted first (perhaps by analysing the lateral histograms), and then each candidate can be accepted or rejected by performing template matching in its vicinity.

4 Partial summation

The basis of the approach to image analysis proposed in this report is a technique known as partial summation. Access to the image data is speeded up at the expense of a pre-processing stage, which generates a table of partial sums. This table may then be used to calculate the sum of grey levels in any arbitrary rectangular region of the image, which is useful for many image processing algorithms.

4.1 One-dimensional partial summation

Consider a single row of X pixels of image data $I(x)$ where $x \in \{0, 1, \ldots, X-1\}$. A second data structure P_1 can be generated, in which each entry is set to the cumulative sum of the pixel values in I:

$$P_1(x) = \begin{cases} 0, & x = 0 \\ \sum_{r=0}^{x-1} I(r), & x \in \{1, 2, \ldots, X\}. \end{cases} \tag{16}$$

P_1 contains the one-dimensional partial sums[15] of I and can be used to calculate the sum of grey levels for any arbitrary run of consecutive pixels in I (from $I(a)$ to $I(b)$ inclusively) [20]:

$$\sum_{r=a}^{b} I(r) = P_1(b+1) - P_1(a). \tag{17}$$

For example, the sum of the grey levels of pixels 2 through 5 inclusive is given by $P_1(6) - P_1(2)$. In the example of Figure 17, this gives a value of 11, which can be verified readily from I.

[15] This process is known as partial summation after Omohundro [26]. It is similar to the use of vector dominance to perform range searching—a technique used in computational geometry. A clear introduction to this topic is given by Preparata and Shamos [30].

Figure 17: Example row of image data (I) and corresponding one-dimensional partial sums (P_1).

I =

2	4	1	3	6	1	1	2	4	3
I(0)	I(1)	I(2)	...	...	I(5)	...		...	I(9)

P_I =

0	2	6	7	10	16	17	18	20	24	27
P_I(0)	P_I(1)	P_I(2)	...		...	P_I(6)	...		...	P_I(10)

4.2 Two-dimensional partial summation

One-dimensional partial sums are useful for certain image processing operations (see Section 5). However, the real power of partial summation is its extension to any number of dimensions. The two-dimensional case is of greatest use in image processing applications [23, 26]. In this case, two pre-processing steps are necessary. The first computes partial sums for each row of the image, as outlined above:

$$P_1(x,y) = \begin{cases} 0, & x = 0 \\ \sum_{r=0}^{x-1} I(r,y), & x \in \{1,2,\ldots,X\}. \end{cases} \tag{18}$$

These are then used in a second, vertical pass which effectively computes partial sums of partial sums[16]:

$$P_2(x,y) = \begin{cases} 0, & x = 0 \text{ or } y = 0 \\ \sum_{s=0}^{y-1} P_1(x,s), & x \in \{1,2,\ldots,X\},\ y \in \{1,2,\ldots,Y\}, \end{cases} \tag{19}$$

where $x \in \{0,1,\ldots,X\},\ y \in \{0,1,\ldots,Y\}$. Each entry $P_2(x{+}1,y{+}1)$ contains the sum of the grey levels of all pixels to the left of, above, and including the pixel at position (x,y) in I. It is now possible to sum the grey levels within any rectangular area of the image I enclosed by $I(a,b)$ and $I(c,d)$ (see Figure 18(a)). $P_2(c{+}1,d{+}1)$ is the sum of grey levels of all pixels to the left of, above, and including $I(c,d)$. Subtracting $P_2(a,d{+}1)$ from this removes the unwanted pixels to the left of the specified rectangle, and similarly those above it can be subtracted using $P_2(c{+}1,b)$. The pixels which are both to the left of and above $I(a,b)$ have now been discounted twice, so $P_2(a,b)$ must be added back in (Figure 18(b)). Thus

$$\sum_{x=a}^{c}\sum_{y=b}^{d} I(x,y) = P_2(c{+}1,d{+}1) - P_2(a,d{+}1) - P_2(c{+}1,b) + P_2(a,b). \tag{20}$$

Having performed the pre-computation, partial summation therefore enables the sum of grey levels within any arbitrary rectangular region of an image to be calculated from just four values, which is a $\Theta(1)$ process.[17] Equation 19 can be re-written as a recursive definition:

$$P_2(x,y) = \begin{cases} 0, & x = 0 \text{ or } y = 0 \\ \sum_{r=0}^{x-1} I(r,y) + P_2(x{-}1,y{-}1), & x \in \{1,2,\ldots,X\},\ y \in \{1,2,\ldots,Y\}, \end{cases} \tag{21}$$

[16] It makes no difference if the table of two-dimensional partial sums is generated by first summing columns and then summing rows with a horizontal pass.

[17] In this report, the term "partial summation" is used to refer to both the process of calculating the table of partial sums as well as the subsequent access to four entries in this table to calculate Equation 20.

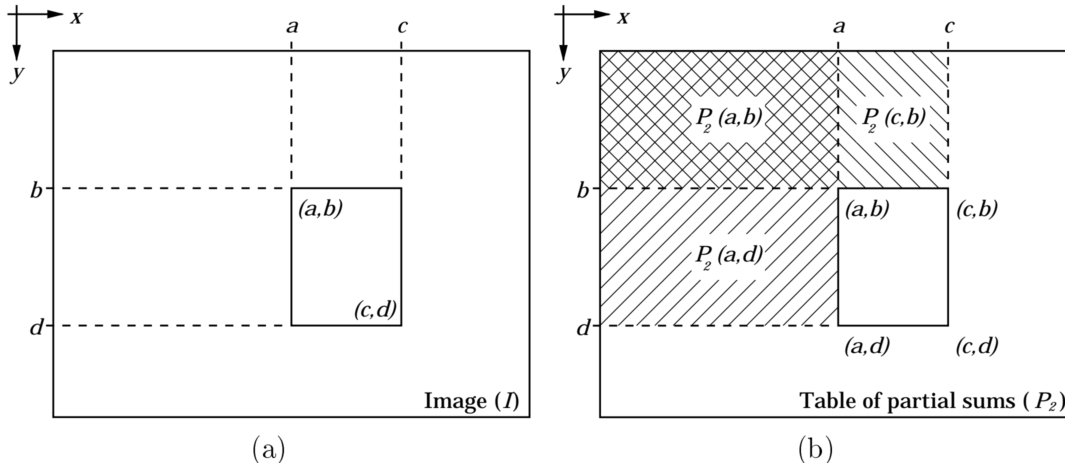


Figure 18: (a) Rectangular region in I defined by two vertices. (b) Using K to sum the grey levels in the region.

where $x \in \{0, 1, \ldots, X\}$, $y \in \{0, 1, \ldots, Y\}$. From this it can be seen that the pre-computation of P_2 can be carried out in a single pass through the image, by summing along successive lines and adding in previously computed sums.

4.3 Memory use

The increased speed of access to image data afforded by the generation of partial sums comes at the expense of increased memory use. For an image of $X \times Y$ E-bit pixels, the largest possible entry in the two-dimensional table of partial sums is given by

$$\max_{x,y,I}(P_2(x,y)) = \max_{I}(P_2(X,Y)) = X \times Y \times (2^E - 1), \tag{22}$$

thus the number of bits required to store each entry is

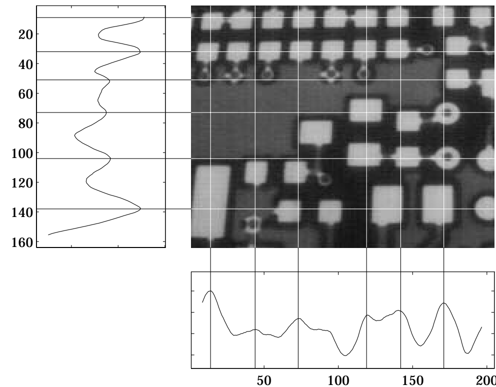
$$\left\lceil \log_2\Big((X \times Y \times (2^E - 1)) + 1\Big) \right\rceil, \tag{23}$$

where $\lceil x \rceil$ denotes the smallest integer greater than or equal to x. A typical image may well have $E = 8$ and $256 \leq X, Y \leq 1024$, and will therefore require between 24 and 28 bits for each partial sum entry. In practice this means that four (8-bit) bytes will be needed to store each entry. Due to the nature of partial sum calculation, the same area of memory may be used to store first the one-dimensional, and subsequently the full two-dimensional partial sums. If a small image is being analysed, or only one-dimensional partial sums are needed, it may still prove more efficient to use 4-byte entries due to the processor architecture.[18]

Example code to generate partial sums is given in Appendix B.

[18] In theory it is possible to use variable sized entries in the table of partial sums, as required. However, the motivation for the generation and use of partial sums is to facilitate fast image processing without the need for complex algorithms. Decreasing memory requirements is therefore not a high priority.

Figure 19: The lateral histograms of a cluttered image do not provide reliable information as to the positions of features in the image.



5 Using partial sums for faster spatial image processing

Chapter 3 discussed practical ways which have been used in the past to implement various vision algorithms. This section demonstrates how the partial summation of Section 4 may be used as an alternative basis for many image processing algorithms. Not only does it provide an efficient mechanism, but the resulting algorithms are relatively simple, making them easy to understand, code and debug.

5.1 Lateral histograms

The lateral histograms of an image, obtained by summing pixels along rows and columns, can be generated very simply from two-dimensional partial sums. Calculating the lateral histograms explicitly requires two passes through the image (see Section 4.2), which may be marginally slower than the combined time for calculating the partial sums and then using these to calculate the histograms.[19] However, avoiding partial sum calculation reduces the amount of memory required dramatically. Therefore, pre-computing the partial sums only to calculate lateral histograms seems wasteful. However, if the partial sums of the image already exist, it becomes natural to re-use them. It is often helpful to smooth the lateral histograms to reduce the effects of noise, before further processing (e.g. Figure 4(b), page 5). This can be achieved simply by summing pixels in a neighbourhood of rows (or columns),[20] which in turn can be calculated directly from the partial sums, in constant ($\Theta(1)$) time with respect to the neighbourhood size.

One major problem with the use of lateral histograms was outlined in Section 2.2. If the image to be processed is cluttered, the 'silhouettes' of the objects begin to merge, and selecting candidate locations is difficult (see Figure 19). The solution to this problem is to use a multiresolution approach. The original image is split into several (overlapping) sub-images, and these are processed separately before the results are recombined. Partial summation again provides a natural solution, because the lateral histograms of any arbitrary rectangle in the image may easily be computed. The relatively time-consuming step

[19]Although calculating both lateral histograms requires two nested loops, each of these is simpler than the single nested loop needed to calculate the two-dimensional partial sums. This means that the total computation time for both is comparable.

[20]Dividing by the number of rows (or columns) used is strictly necessary, but if it is purely the shape of the histogram which is of importance, this step is not needed.

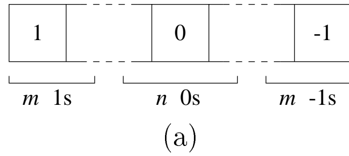


Figure 20: (a) Hager's edge-detecting filter [20]. (b) 5×5 edge-detecting filter.

2	1	0	-1	-2
2	1	0	-1	-2
2	1	0	-1	-2
2	1	0	-1	-2
2	1	0	-1	-2

(b)

of calculating the partial sums need only be done once.

If the only purpose of computing the partial sums is to aid lateral histogram generation, and smoothing is not required, it is sensible to modify the nature of the partial sums. Rather than compute the set of two-dimensional partial sums, it is more appropriate to compute both sets of one-dimensional partial sums. This enables each entry in a lateral histogram to be calculated from just two values, at the expense of slightly greater storage requirements. Once again the partial summation need only be done once.

5.2 Linear filtering

Partial sums may also be used in the implementation of certain linear spatial filters. The most obvious example is the mean smoothing filter (e.g. Figure 9(d), page 9). When applied to an area of the image, the filter effectively calculates the mean of the grey levels in that area. Traditionally this is done by moving along the rows and down the columns of the area of the image concerned, summing the pixel values. For a $K \times K$ kernel, this takes time proportional to the number of filter elements, i.e. $T(K) = \Theta(K^2)$. This calculation can be done more simply by using the partial sums of the image to calculate the sum of grey levels in the area concerned, and then dividing this by the number of pixels. The size of the filter has no effect on the time needed to perform this calculation—$T(K) = \Theta(1)$. If the partial sums need to be calculated exclusively for use in a single filtering operation, this can still be faster than the traditional approach (even for filters as small as 3×3). Section 6 presents some empirical speed comparisons confirming these ideas.

Other linear operators cannot be implemented so readily with partial summation. However, many common operators contain blocks of repeated elements, and the response of each of these may be calculated separately from the partial sums. For example, the response of the filter shown in Figure 20(b) can be computed from four partial sums (one for each of the non-zero columns). This approach has previously been successfully used by Hager [20] to speed up processing for a one-dimensional spatial filter. Hager pre-computes one-dimensional partial sums in the horizontal direction and uses the spatial filter shown in Figure 20(a) to detect vertical edges in the image. However, the extension to two dimensions as proposed in this report can generate further savings.

5.3 Nonlinear filtering

As mentioned in Section 3.6, median, shrink and expand filters are particularly simple to implement when a binary image is used. The total number of pixels within the filter area needs to be compared with a threshold. This would traditionally involve K^2 memory reads, K^2-1 additions and a comparison,[21] i.e. $T(K) = 2K^2 = \Theta(K^2)$. However, if the filter or structuring element is rectangular (which is often the case), and the partial sums of the image have been pre-computed, only four reads and three additions are required to calculate the number of pixels within the neighbourhood concerned. Including the comparison operation, $T(K) = 8 = \Theta(1)$. In this way, many common binary image filtering operations can be implemented using partial summation, giving a speed-up in operation proportional to the square of the kernel size.

Partial summation can also be useful in the more general case of binary morphological operations where the structuring element is of any size or shape. For example, binary erosion involves testing each of the pixels under the structuring element to see if they are all set, which is equivalent to summing the pixels under the structuring element. If the structuring element can be represented (or approximated) by some combination of rectangles, partial summation can be used to perform the summations over each rectangle, and hence potentially reduce computation. The efficiency of this approach depends largely on whether or not it is possible to represent the structuring element with a small number of rectangles, and hence keep the number of partial summations required to a minimum.

5.4 Template matching

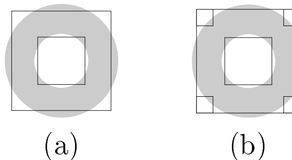
As described in Section 2.3, template matching can be made much less computationally demanding if a low resolution pass is made through the image to generate candidate matches. A coarse template is robust to small changes in orientation and scale, and may be sufficient to identify the feature in question. If necessary it is possible to analyse each candidate feature in more detail, perhaps using a higher resolution template.

For example, the different pads on the PCB shown in Figure 7(a) on page 6 can be approximated to various uniform-intensity rectangles. This means that they can be detected (in a coarse fashion) by looking for suitably sized bright rectangular regions which are set against a darker background. Partial summation can be used to calculate the mean pixel intensities within the required rectangular regions (in $\Theta(1)$ time). A match is made if the inner region is bright, but the band surrounding it (the difference between the outer and inner regions) is dark. A similar technique can also be used to speed up industrial inspection tasks involving analysis of grey level distributions in different regions [28]. This is much faster than full template matching, whilst also being robust to slight changes in orientation, scale and lighting.

Rectangular feature detection is particularly suited to implementation using partial summation. However, binary shapes in general may be represented at different resolutions by approximating them using quad trees [31] or indeed arbitrary combinations of rectangles. For example, Figure 21(a) shows how circular pads may be matched very coarsely using a square approximation; Figure 21(b) uses six squares to approximate the pad slightly

[21] Ignoring any looping and array indexing overheads.

Figure 21: Two examples of how to approximate a circular pad as a combination of rectangles.



more accurately, but still requires just twenty memory reads (one for each vertex in the approximation)—a considerable saving over full template matching.

Sometimes the approach outlined above is only sufficient to highlight candidate features; full template matching is needed to confirm the presence of the relevant feature. In this case, the partial sums can be re-used for normalizing the result of the cross-correlation, allowing the divisor of Equation 6 to be calculated in $\Theta(1)$ time.

5.5 Another look at mean filtering

Section 3 presented a number of techniques which may be used to provide a significant speed-up over the basic nested-loop implementation of a $K \times K$ mean filter. These include overlap and save techniques and small generating kernels which both produce a speed-up by reformulating the traditional $\Theta(K^2)$ process as a $\Theta(K)$ algorithm. Section 5.2 showed how two-dimensional partial summation can be used to create a further saving in execution time ($T(K) = c = \Theta(1)$), at the expense of increased memory requirements. However, simple mean filtering may be achieved even more efficiently by combining the overlap and save technique with a form of one-dimensional partial summation, to give a $\Theta(1)$ algorithm with a smaller constant c, and which uses less memory.

To recapitulate, a mean filter is conventionally applied by passing over the image in a raster fashion (Figure 6, page 6). At each location, the pixels under the filter are added up, and the sum is divided by the size of the filter to give the filter response at that position. With overlap and save, having applied the filter at one position, the result is re-used in the calculation of the response at the next position, one to the right. This involves subtracting pixel values from one side of the filter and adding in the values to the other side of the filter, effectively moving the filter along by one location. This is still a $\Theta(K)$ process, because the number of pixels which must be added and subtracted for each move is proportional to K. If just a single value were added and subtracted each time, the filtering would be $\Theta(1)$ with respect to K. This can be achieved by using one-dimensional partial sums to calculate the values concerned. In fact, slightly greater efficiency is achieved if overlap and save is first used in one dimension to pre-compute the values (which are effectively the sums of vertical runs of K pixels).

This approach, referred to as *double overlap and save* hereafter, uses less memory than full partial summation, since each entry in the table of vertical sums need only be 16 bits long (for an 8 bit grey level image and $K \leq 256$). In addition, if a single mean filtering operation is all that is required, double overlap and save is slightly faster than partial summation, since the constant term in $T(K)$ is smaller. However, the table of vertical sums generated by double overlap and save can only be used for a single, predefined filter size. If any form of multiresolution processing is needed, this approach will be less efficient. Section 6 gives a quantitative measure of the speeds of each algorithm.

Double overlap and save may also be used to implement binary median, shrink and expand filtering with rectangular templates, since these all involve summing the pixels in the area of the template.

6 Speed comparisons

This report has reviewed many common image processing techniques, discussed ways of implementing them, and presented a new approach to implementation. Theoretical analysis of the running times of the various algorithms which may be used indicated that partial summation is particularly efficient. This section gives the results of quantitative comparisons between the different approaches to mean filtering which confirm the superior performance of partial summation. In addition, the advantages of using partial summation for the task of PCB pad location are demonstrated.

6.1 Grey level mean filtering

Figure 22 shows the results of applying various sizes of square mean filters to a 288×227 (i.e. 64kB) grey level image using various different algorithms. As expected, for a given kernel size (K) the different algorithms generate identical filtered images, but as the plot of execution time against kernel size shows, the algorithms exhibit varying degrees of efficiency. As predicted, the naïve approach using conventional nested loops to apply the mean filter displays $\Theta(K^2)$ performance; execution times grow very quickly. The use of small generating kernels or the overlap and save technique reduce the problem to $\Theta(K)$, as reflected in the graph. In this case, overlap and save is marginally more efficient, although this might change for different filter kernels. However, the most efficient approaches, both with $\Theta(1)$ execution time, are the partial summation and double overlap and save algorithms. In fact, as kernel size grows, they both show a small drop in execution time, due to the reduction in the size of the filtered output image as K increases (see Section 3.2).

For any given mean filter, double overlap and save is marginally faster than partial summation. If a number of mean filters are to be applied at different resolutions, however, partial summation proves more efficient, because the initial step of calculating the partial sums is only performed once. Figure 23 shows how execution time grows as the number of resolutions at which the filter is applied increases.

As explained in Section 3.6, binary median, shrink and expand filters can be implemented by simply counting pixels. This means that the implementation is almost identical to that of the grey level mean filter, and hence the execution speeds will be virtually the same as those given in Figures 22 and 23.

Pointers were used throughout the examples given above to ensure efficient implementation by the compiler (see Section 3.3). For interest, Figure 24 compares two traditional implementations of the mean algorithm, using arrays and pointers. The benchmarks were carried out on a 25MHz 486SX IBM compatible PC with 4MB of RAM. Code was written in C and compiled using Borland C++ [7] under MS-DOS [25] and the timings shown were averaged over ten trials to minimize any timer quantization effects. A 100MHz machine,

Figure 22: Comparison between execution time for a number of different approaches to mean filtering. As the filter size (K) increases, it becomes apparent that the conventional approach is $\Theta(K^2)$, the SGK and overlap and save solutions are $\Theta(K)$, whilst partial summation and the double overlap and save algorithms are $\Theta(1)$. See text for more details.

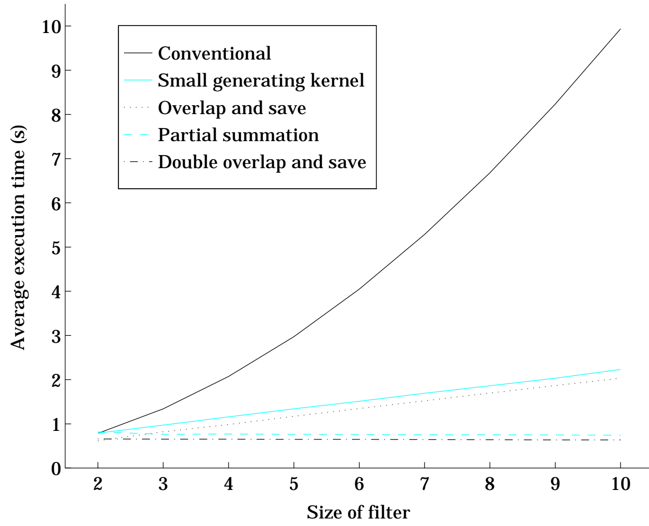


Figure 23: If any form of multi-resolution image processing is employed, the operator in question must be applied at a number of scales. This graph shows how execution time changes as the number of scales/resolutions increases for the two fastest algorithms. See text for details.

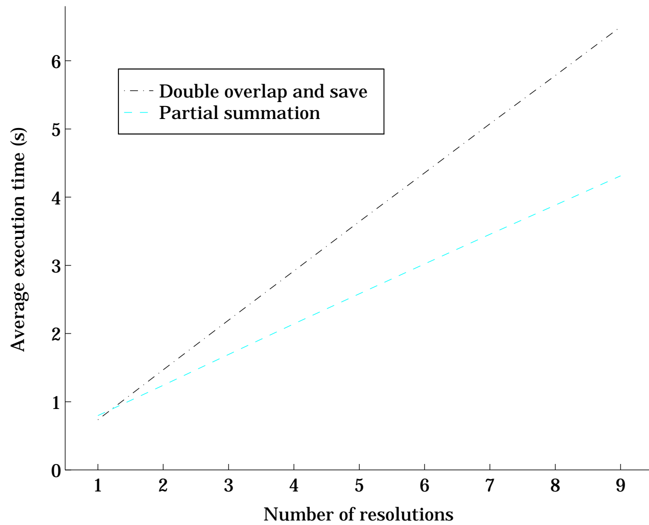


Figure 24: Comparison between execution time of mean filter using arrays and pointers. Both approaches are $\Theta(K^2)$ with respect to kernel size (K), but the constant term makes the array implementation around three times slower in practice. With larger kernels, this is quite a significant difference.

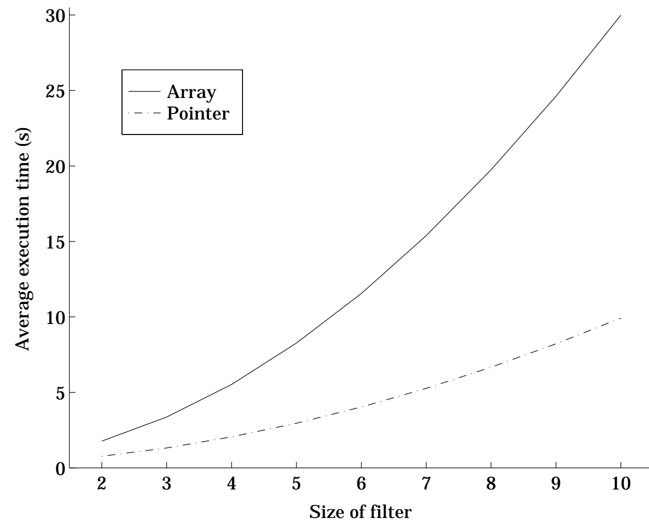
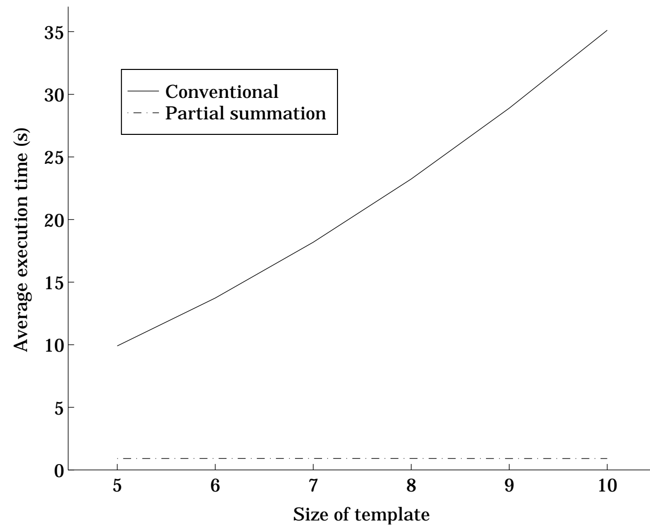


Figure 25: Comparison between the execution time of (i) normalized cross correlation template matching and (ii) partial summation. The feature to be located was represented as a square grey level template in (i), and by two square areas (a bright central square surrounded by dark band) in (ii). With larger templates (more likely in real applications), the speed-up will be even greater.



not uncommon by today's standards, would potentially speed up operation fourfold. Using a good optimizing compiler or implementing the algorithms directly in assembly language would be likely to provide a further speed-up.[22] Given the relative simplicity of the algorithms, the use of assembler is not the daunting task it might otherwise be. Additional improvements would result from running the code under 80486 protected mode (using a DOS extender), instead of real mode [24].

6.2 Template matching

Section 2.3 described and gave examples of the process of template matching. Being $\Theta(K^2)$, template matching is a fairly computationally demanding process, especially when large templates are used. Section 5.4 showed that binary template matching with a rectangular template could be implemented very efficiently by using partial summation to calculate Hamming distance. However, even for arbitrarily shaped features in grey level images, partial summation can be used to implement coarse template matching efficiently.

As Section 3.7 indicated, use of a coarse template not only speeds up the matching process during an initial first pass, but also makes it more robust to variations in orientation and scale of the feature to be matched. For example, if the feature to be matched is a rectangular pad on a grey level image of a PCB, the template can be approximated by a bright rectangle on a dark (rectangular) background (see Section 5.4). The brightness of these two regions (the central rectangle and its surrounding band) can be calculated using two partial summations, and matching carried out by comparing these averages with predetermined thresholds. This is a lot faster than template matching using normalized cross-correlation—Figure 25 compares execution times for the two processes. However, although the partial summation approach produces a much coarser match, it is still very useful, as Figure 7 on page 6 shows.

[22] Borland C++ is known to compile quickly, but it does not necessarily generate the most efficient code [24]. In addition, `huge` pointers, which are normalized after every use, were used to ensure that no overflow occurred when accessing data structures greater than 64kB in size; this normalization overhead affected performance detrimentally.

6.3 A practical application

The motivation behind the development of the techniques described in this report was the need for fast image processing algorithms for an IBM PC-based electronics assembly machine [22]. The delay in the feedback generated by the machine's vision system must be kept as small as possible in order to minimize cycle times. At the same time, it is very important that the feedback information is accurate; the system must be robust to changes in lighting, PCB characteristics, different components and so on. One of the specific problems involved is locating pads on the PCB, so that as the placement head of the machine moves, its precise position relative to the PCB can be calculated.

The solution developed is based on partial summation. Having calculated the partial sums, the second stage of the algorithm involves analysing the grey level histograms of the image to roughly determine any non-uniformities in illumination. In the prototype system, this is done by fitting a straight line to the histograms[23] and using this to normalize subsequent accesses to image data, as in Figure 5 on page 5. The rectangular pads are then located by coarse template matching as described in Section 5.4. In this application the orientation of the PCB and consequently the orientation of the pads are well-defined which aids feature detection greatly. However, the template model is only very crude, and it is possible that false matches are made; for example, the foreshortening which results from the required camera location means that the perceived pad size varies quite significantly. For this reason, each candidate match from the template matching stage is examined in more detail. This is done by constructing the lateral histograms in the neighbourhood of each candidate and examining them in a fashion similar to that of Section 2.2 to determine if any pads do exist at the given location.

7 Conclusions

This report has presented a new approach to image processing, based on a technique known as partial summation. The first step in this approach is to create a table of partial sums, which can be calculated simply from any given image. This table may subsequently be used to provide information about the image, suitable for many image processing operations, with minimal computation. Not only does partial summation provide a very efficient mechanism for tackling many problems in image processing, but it also leads to readable, understandable code which is ultimately easier to debug than the complex code arising from other efficient implementations. The overhead of using this approach is that a (relatively) large amount of memory is needed to store the table of partial sums. Typically this will be four times the memory needed for a single image (e.g. 256kB if a 64kB image is used).

Partial summation is particularly suitable if some form of multiresolution image processing is being used, since the table need only be computed once. If the newly created filtered image is itself to be filtered, the partial sums need to be recalculated; updating the original table is no more efficient than recalculating it from scratch. Similarly, the similarity between successive frames in a video sequence cannot be used sensibly to update partial

[23] If appropriate, another model could easily be substituted.

sum information. However, even if the partial sums are only used once, the approach advocated in this report is still often more efficient than other image processing techniques.

There are many other spatial image processing operations which are not covered here and for which partial summation would offer no speed-up. In addition, there are a number of techniques which operate outside the spatial domain, such as Fourier analysis, which have not been considered. Indeed, even when implementing the algorithms presented in this report, partial summation is only useful in specific circumstances. However, it is beneficial to many practical machine vision applications, where a significant speed-up in execution time can be made. As a bonus, partial summation algorithms are generally simple to understand and straightforward to implement, an attribute worth consideration. Even if an algorithm is not optimal under all conditions, there is great merit in simplicity [40]; this is perhaps particularly true for image processing algorithms [8, 34]. Simple algorithms are easier to code and debug, which is important in today's competitive environment where development times are continually reducing.

Partial summation can undoubtedly be used in ways not discussed in this report, to provide further increases in efficiency of image processing applications. For example, with certain hardware it is possible to calculate the partial sums in real time, as the rows of pixel data are generated by the A to D converter of the frame-grabber [11]. However, this report has analysed a number of spatial image processing techniques and demonstrated how these may be implemented using partial summation, and therefore provides a basis which can be built upon as necessary in individual applications.

Acknowledgements

The authors would like to thank Oliver Bühler of the Department of Applied Mathematics and Theoretical Physics for extensive, informative and insightful discussions during the course of this work. Thanks are also due to Mark Mackey and Simon Moore for enlightening descriptions of implementation issues and to Melanie Bowran, Andreas Cambitsis, Mike Majors, Ferdi Samaria, Michael Taylor and Iain Tuddenham for diligent proof reading and valuable subsequent discussions.

A Algorithm analysis

Analysing an algorithm involves predicting the resources required by that algorithm. This appendix provides a brief introduction to the techniques and terminology involved in algorithm analysis, so that the reader may follow the arguments presented in this report more easily.

A.1 Running time

When solving a problem, there may be a number of algorithms which can be used; two often contradictory goals form the grounds for choosing one algorithm over another. Firstly, it should be easy to understand, code and debug, and secondly it should be efficient in use of the computer's resources, and in particular quick to execute [2]. Frequently, the latter

of these goals becomes more important. Indeed, the motivation for the ideas presented in this report was largely the desire to improve the speed of image processing operations although it is hoped that a side-effect of partial summation is understandable code. The *running time* of a program depends on factors such as [2]:

- the quality of the code generated by the compiler;
- the nature and speed of the instructions on the machine used;
- the input to the program;
- the (time) complexity of the algorithm underlying the program.

The first two factors are usually beyond the programmer's control, and it is the last which concerns this report. However, the running time of a particular algorithm is closely related to its input (the third factor in the list). In fact, the running time does not vary solely with the exact input, but largely depends on the 'size' of the input. It is customary therefore, to talk of $T(n)$, the running time of a program given an input of size n. The units of $T(n)$ are left unspecified; to express $T(n)$ in seconds requires knowledge of the compiler, the processor and so on (i.e. (1) and (2) above). Instead $T(n)$ is considered to be the number of unit operations executed by an algorithm.

In many instances the assumption that the execution time depends solely on the size of the input is invalid. In this case, $T(n)$ is defined to be the *worst case* running time, i.e. the maximum running time of all possible inputs of size n. It is possible to use an *average* or *expected* running time metric to compare different algorithms. $T_{avg}(n)$ is defined as the average time taken to execute an algorithm over all inputs of size n. However, it may be incorrect to assume that all inputs are equally likely—for some algorithms the worst case occurs frequently, and besides which, the average case is often nearly as bad as the worst case [12]. It is also tends to be more difficult to determine the average case running time analytically. For these reasons, worst case analysis is usually preferred.

A.2 Calculating running time

In order to compare different algorithms, it is useful to estimate the running time ($T(n)$) of each, given an input of size n. In order to estimate $T(n)$, it is necessary to have a model of the implementation technology which will be used [12], so that the complexity of the different steps which form the algorithm can be evaluated. In the analysis of this report it is assumed that a generic single-processor Von Neumann computer with random access memory is used. Perfect analysis requires translation of the algorithm into machine level instructions, and consideration of the number of clock cycles required for each of these. However, to make analysis easier, the steps in the algorithm can be considered to be combinations of *unit operations*[24] which will be performed as the algorithm executes. A suitable set of unit operations are [5]:

- assigning a value to a variable of simple or pointer type;
- evaluating a component variable[25] of simple or pointer type;

[24] i.e. fundamental operations which take an equal time to execute.

[25] A component variable is an indexed variable or a record field selector.

- executing an arithmetic, boolean or relational operator;
- executing an empty, input or output statement;
- initializing or terminating a procedure or function call.

These assumptions are clearly a simplification—for example different arithmetic operations may well take different amounts of time—but they provide a suitable basis for the analysis needed here. As an example, two lines of code (from Appendix B) are given below along with their breakdown into unit instructions. Analysis of the `for` loop involves calculating the number of times each part of the statement is executed in addition to the number of unit operations. These are then multiplied together to determine the total execution time.

```
sum += I[x + kx][y + ky] * T[kx][ky];
```

Breakdown: 1 ($x + k_x$), 1 ($y + k_y$); 2 ($I[\ldots][\ldots]$), 2 ($T[k_x][k_y]$); 1 (multiplication); 2 (addition and assignment).

Number of unit operations	
2	Addition within array indices
4	Calculation of array indices
1	Multiplication
2	Addition and assignment
9	In total

```
for (x = 0; x < (X - K); x++)
```

Executed: `x = 0` once (1 unit); `x < (X - K)` $(X–K+1)$ times (2 units); `x++` $(X–K)$ times (1 unit).

Number of unit operations	
$1(1)$	Initialize loop counter
$2(X–K+1)$	Subtract and test
$1(X–K)$	Increment loop counter
$3X-3K+3$	In total

A.3 Memory usage

Just as the running time of an algorithm may be determined analytically, so too may the memory required. The exact amount of memory used will depend on many factors, as with running time. Once again, however, it is convenient to express the memory used by a particular algorithm as a function of the size of the input. Thus $M(n)$ is defined as the (worst case) storage used by an algorithm given an input of size n. Frequently, $M(n) \propto n$, but this is not always true, and in any case it may be useful to determine the constant of proportionality.

A.4 Growth of functions

Section A.2 showed how to calculate a fairly precise estimate of the running time of an algorithm. However, the precision gained through this technique is often not worth the effort involved in calculating it [12]. For large enough inputs, the multiplicative constants and lower-order terms of the exact running time are dominated by the effects of the input size itself. In order to compare the relative performance of different algorithms, it is often appropriate to use a measure of *asymptotic* efficiency, which considers *rate of growth* or *order of growth* of a function. This expresses how the running time of an algorithm increases with the size of the input 'in the limit'—as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs. Thus, when considering the rate of growth, only the leading term in the expression of $T(n)$ is of significance. Indeed, the coefficient of this term may also be ignored since it is independent of the size of the input [12].

To express this notion of the growth rate of a function, a special notation is used. For a given function $g(n)$, $\Theta(g(n))$ (pronounced *big-theta* of $g(n)$) is defined as a set of functions:

$$\Theta\Big(g(n)\Big) = \Big\{f(n) \mid \exists\, c_1, c_2, n_0 \geq 0 : n \geq n_0 \Rightarrow 0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)\Big\}. \tag{24}$$

Thus a function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that $f(n)$ lies between $c_1\, g(n)$ and $c_2\, g(n)$ for sufficiently large n. If $f(n) \in \Theta(g(n))$, it is usual to actually write "$f(n) = \Theta(g(n))$" [12]. For all values of n above n_0, the value of $f(n)$ lies at or above $c_1\, g(n)$ and at or below $c_2\, g(n)$, i.e. $g(n)$ is an *asymptotically tight bound* for $f(n)$. For example, suppose that $T(n) = 3n^2 + n$. From the preceding definition, $\Theta(n^2)$ is the set of all functions for which n^2 is an asymptotically tight bound, which includes $T(n)$. Therefore, we write $T(n) = 3n^2 + n = \Theta(n^2)$. It is usually considered bad style to include constants or low-order terms in $g(n)$, although it is not technically incorrect [42].

There are several variations on the Θ-notation, but the most common is O-, or *big-oh* notation[26] which asserts only an asymptotic upper bound:

$$O\Big(g(n)\Big) = \Big\{f(n) \mid \exists\, c, n_0 \geq 0 : n \geq n_0 \Rightarrow 0 \leq f(n) \leq c\, g(n)\Big\}. \tag{25}$$

O-notation is used to express the upper bound of a function, to within a constant factor. Since Θ-notation is stronger that O-notation, $f(n) = \Theta(g(n))$ implies that $f(n) = O(g(n))$ (formally, $\Theta(g(n)) \subseteq O(g(n))$ [12]).

Since O-notation defines only an upper bound, it is valid to make potentially misleading statements, such as $n = O(n^2)$ [21]. However, this can be useful for describing algorithms whose execution time varies with different inputs of the same size. The bubble sort, for example, will take $\Theta(n^2)$ time when given an input which is sorted in reverse, but executes in $\Theta(n)$ time when given a correctly ordered input[27]; it is therefore appropriate to class this algorithm as $O(n^2)$. Since none of the algorithms presented in this report have conditional statements, execution time depends only on the size of the input and the tighter Θ-notation is more appropriate.

[26] O-notation was formalized by Hardy in 1908, and therefore pre-dates Θ-notation [21].

[27] Sort time for a pre-sorted input is $\Theta(n)$ rather than $\Theta(1)$ since it is necessary to check that each pair of elements is in the correct order.

Table 1: Common asymptotic bounds [42], in increasing order from top left to bottom right.

1	constant	n^2	quadratic
$\log n$	logarithmic	n^3	cubic
$\log^2 n$	log-squared	2^n	exponential
n	linear	$n!$	factorial

To prove that some function $T(n) = \Theta(f(n))$, it is usually easiest to apply a repertoire of known results intuitively [42]. The most significant of these are:

- if $T(n)$ is a polynomial of degree p, then $T(p) = \Theta(n^p)$;
- if $T_1(n) = \Theta(g_1(n))$ and $T_2(n) = \Theta(g_2(n))$, then:

$$T_1(n) + T_2(n) = \max(\Theta(g_1(n)), \Theta(g_2(n))) \quad \text{and}$$
$$T_1(n) \times T_2(n) = \Theta(g_1(n) \times g_2(n)).$$

Common asymptotic bounds are shown in Table 1, in order of increasing size.

B Example code

This appendix contains C code listings of several of the algorithms presented in the report. It is assumed that `I`, `I_new` and `P_2` (which contain the original image, new image and two-dimensional table of partial sums respectively) are suitably sized arrays/areas of memory; `X`, `Y` represent the size of the image and `K` the size of the kernel. Counters and so on are generally **`unsigned int`**s; the pointer declarations are also omitted for clarity. There are many places in the code where the introduction of a temporary variable would produce a further speed-up; such optimizations have, in general, not been made in order to keep the code as readable as possible.

Alongside each line of code, the number of unit operations (see Section A.2) needed to execute it is written, along with the number of times that line is executed. Section A.2 also gives examples of how these figures are calculated.[28] By multiplying these two values, and then summing over all the lines in the routine, an estimate of the total execution time can be made. If it is assumed that $X, Y \gg K > 1$, then this estimate can be simplified, and from this the asymptotic behaviour of the algorithm calculated.

[28] The content of a pointer is assumed to be accessible with no overhead. The same is true for the calculation of `&array[0][0]`, the address of the zero'th element of any array.

B.1 Standard, array-based mean filter

```
/*
** STANDARD MEAN FILTER, USING ARRAYS
*/
for (y = 0; y <= (Y - K); y++)
{
    for (x = 0; x <= (X - K); x++)
    {
        total = 0;

        /* PASS OVER THE FILTER AREA */
        for (k_y = 0;  k_y < K; k_y++)
        {
            for (k_x = 0;  k_x < K; k_x++)
            {
                /* SUM THE RELEVANT PIXELS */
                total += I[x+k_x][y+k_y];
            }
        }

        /* CALCULATE MEAN AND STORE IN I_new */
        I_new[x][y] = total / K^2;
    }
}
```

Line	Unit instructions: current line	Unit instructions: iterations
`for (y = 0; y <= (Y - K); y++)`	$3Y-3K+6$	1
`for (x = 0; x <= (X - K); x++)`	$3X-3K+6$	$Y-K+1$
`total = 0;`	1	$(X-K+1)(Y-K+1)$
`for (k_y = 0;  k_y < K; k_y++)`	$2K+2$	$(X-K+1)(Y-K+1)$
`for (k_x = 0;  k_x < K; k_x++)`	$2K+2$	$K(X-K+1)(Y-K+1)$
`total += I[x+k_x][y+k_y];`	6	$K^2(X-K+1)(Y-K+1)$
`I_new[x][y] = total / K^2;`	5	$(X-K+1)(Y-K+1)$
Total	$(8K^2+4K+11)(X-K+1)(Y-K+1) + 6(Y-K+1) + 3$	

$$
\begin{aligned}
T(X,Y,K) &= (8K^2+4K+11)(X-K+1)(Y-K+1)+6(Y-K+1)+3 \\
&\approx XY(8K^2+4K+11)+6Y \\
\therefore T(K) &= \Theta(K^2)
\end{aligned}
$$

B.2 Standard, pointer-based mean filter

```
/*
** STANDARD MEAN FILTER, USING POINTERS
*/
for (y = 0; y <= (Y - K); y++)
{
    /* MOVE TO START OF NEW ROW */
    pointer_I_new = &I_new[0][0] + (X * y);
    pointer_I = &I[0][0] + (X * y);
    for (x = 0; x <= (X - K); x++)
    {
        pointer = pointer_I;
        total = 0;
        /* PASS OVER THE FILTER AREA */
        for (k_y = 0;  k_y < K; k_y++)
        {
            for (k_x = 0;  k_x < K; k_x++)
            {
                /* SUM THE RELEVANT PIXELS */
                total += *pointer;
                /* MOVE TO NEXT PIXEL */
                pointer++;
            }
            /* MOVE TO START OF NEXT ROW */
            pointer += (X - K);
        }
        /* CALCULATE MEAN AND STORE IN I_new */
        *pointer_I_new = total / K^2;
        /* MOVE TO NEXT PIXEL */
        pointer_I_new++;
        pointer_I++;
    }
}
```

Code line	Unit instructions: current line	Unit instructions: iterations
`for (y = 0; y <= (Y - K); y++)`	$3Y{-}3K{+}6$	1
`pointer_I_new = &I_new[0][0] + (X * y);`	3	$Y{-}K{+}1$
`pointer_I = &I[0][0] + (X * y);`	3	$Y{-}K{+}1$
`for (x = 0; x <= (X - K); x++)`	$3X{-}3K{+}6$	$Y{-}K{+}1$
`pointer = pointer_I;`	1	$(X{-}K{+}1)(Y{-}K{+}1)$
`total = 0;`	1	$(X{-}K{+}1)(Y{-}K{+}1)$
`for (k_y = 0; k_y < K; k_y++)`	$2K{+}2$	$(X{-}K{+}1)(Y{-}K{+}1)$
`for (k_x = 0; k_x < K; k_x++)`	$2K{+}2$	$K(X{-}K{+}1)(Y{-}K{+}1)$
`total += *pointer;`	2	$K^2(X{-}K{+}1)(Y{-}K{+}1)$
`pointer++;`	1	$K^2(X{-}K{+}1)(Y{-}K{+}1)$
`pointer += (X - K);`	3	$K(X{-}K{+}1)(Y{-}K{+}1)$
`*pointer_I_new = total / K^2;`	3	$(X{-}K{+}1)(Y{-}K{+}1)$
`pointer_I_new++;`	1	$(X{-}K{+}1)(Y{-}K{+}1)$
`pointer_I++;`	1	$(X{-}K{+}1)(Y{-}K{+}1)$

Total: $(5K^2{+}7K{+}12)(X{-}K{+}1)(Y{-}K{+}1) + 12(Y{-}K{+}1) + 3$

$$
\begin{aligned}
T(X,Y,K) &= (5K^2 + 7K + 12)(X - K + 1)(Y - K + 1) + 12(Y - K + 1) + 3 \\
&\approx XY(5K^2 + 7K + 10) + 12Y \\
\therefore T(K) &= \Theta(K^2)
\end{aligned}
$$

B.3 Calculating the partial sums (pointer-based)

```
                                                      Unit instructions
                                                   current line   iterations
/*
** Compute the 2D table of partial sums
*/

/* Make the zero'th row of P2 zeros */
pointer_P2 = &P2[0][0];                                 1            1
for (x = 0; x < (X + 1); x++)                           3X+6         1
{
    *pointer_P2 = 0;                                    1            X+1
    pointer_P2++;                                       1            X+1
}

/* Make the zero'th column of P2 zeros */
pointer_P2 = &P2[0][0];                                 1            1
for (y = 0; y < (Y + 1); y++)                           3Y+6         1
{
    *pointer_P2 = 0;                                    1            Y+1
    pointer_P2 += (X + 1);                              3            Y+1
}

/* Calculate 2D partial sums */
pointer_I = &I[0][0];                                   1            1
/* Start on first row, zero'th column of P2 */
pointer_P2 = &P2[1][0];                                 2            1
pointer_prev_P2 = &P2[0][0];                            1            1
for (y = 0; y < Y; y++)                                 2Y+2         1
{
    total = 0;                                          1            Y

    /* Skip zero'th column */
    pointer_P2++;                                       1            Y
    pointer_prev_P2++;                                  1            Y
    for (x = 0; x < X; x++)                             2X+2         Y
    {
        /* Add in current entry and store */
        total += *pointer_I;                            2            XY
        *pointer_P2 = total + *pointer_prev_P2;         2            XY

        /* Move to next entry */
        pointer_I++;                                    1            XY
        pointer_P2++;                                   1            XY
        pointer_prev_P2++;                              1            XY
    }
}
                                                   ---------------------------
                                                      9XY + 14Y + 5X + 26
```

$$
\begin{aligned}
T(X,Y,K) &= 9XY + 14Y + 5X + 26 \\
\therefore T(K) &= \Theta(1)
\end{aligned}
$$

B.4 Mean filter using partial sums (pointer-based)

```
/*
** Apply a K×K mean filter
** using the table of partial sums (P2)
*/
for (y = 0; y <= (Y - K); y++)
{
    /* Calculate pointers for each
       corner of filter area */
    tl_P2 = &P2[0][0] + ((X + 1) * y);
    tr_P2 = tl_P2 + K;
    bl_P2 = tl_P2 + (K * (X + 1));
    br_P2 = bl_P2 + K;

    pointer_Inew = &I[0][0] + (X * y);

    for (x = 0; x <= (X - K); x++)
    {
        /* Calculate mean */
        *pointer_Inew = (*br_P2 -
            *tr_P2 - *bl_P2 + *tl_P2) / K^2;

        /* Move to next pixel */
        tl_P2++; tr_P2++; bl_P2++; br_P2++;
        *pointer_Inew++;
    }
}
```

Code line	Unit instructions: current line	Unit instructions: iterations
`for (y = 0; y <= (Y - K); y++)`	$3Y-3K+6$	1
`tl_P2 = &P2[0][0] + ((X + 1) * y);`	4	$Y-K+1$
`tr_P2 = tl_P2 + K;`	2	$Y-K+1$
`bl_P2 = tl_P2 + (K * (X + 1));`	4	$Y-K+1$
`br_P2 = bl_P2 + K;`	2	$Y-K+1$
`pointer_Inew = &I[0][0] + (X * y);`	3	$Y-K+1$
`for (x = 0; x <= (X - K); x++)`	$3X-3K+6$	$Y-K+1$
`*pointer_Inew = (*br_P2 - *tr_P2 - *bl_P2 + *tl_P2) / K^2;`	6	$(X-K+1)(Y-K+1)$
`tl_P2++; tr_P2++; bl_P2++; br_P2++;`	4	$(X-K+1)(Y-K+1)$
`*pointer_Inew++;`	1	$(X-K+1)(Y-K+1)$
Total	$14(X-K+1)(Y-K+1) + 21(Y-K+1) + 3$	

$$
\begin{aligned}
T(X,Y,K) &= 14(X-K+1)(Y-K+1) + 21(Y-K+1) + 3 \\
&\approx 14XY + 21Y \\
\therefore T(K) &= \Theta(1)
\end{aligned}
$$

B.5 Mean filter using overlap and save (pointer-based)

$$\begin{aligned} T(X,Y,K) &= (11K+11)(X-K+1)(Y-K+1)+(10K+26)(Y-K+1) \\ &\quad +5K^2+7K+9 \\ &\approx (11K+11)XY+(10K+26)Y \\ \therefore T(K) &= \Theta(K) \end{aligned}$$

Code	Unit instructions: current line	Unit instructions: iterations
`/*`		
`** Overlap and save`		
`*/`		
`/* Bootstrap algorithm by calculating first kernel position in full */`		
`total = 0;`	1	1
`pointer = &I[0][0];`	1	1
`for (y = 0; y < K; y++)`	$2K+2$	1
`{`		
`    for (x = 0; x < K; x++)`	$2K+2$	K
`        total += *pointer;`	2	K^2
`        pointer++;`	1	K^2
`    }`		
`    *pointer += (X - K);`	3	K
`}`		

continued...

```
pointer_I_new = &I_new[0][0];                            1            1
pointer_I = &I[0][0];                                    1            1
for (y = 0; y <= (Y - K); y++)                           3Y-3K+6      1
{
    /* Save row start total for use later */
    total_orig = total;                                  1            Y-K+1
    /* Move along rows */
    for (x = 0; x <= (X - K); x++)                       3X-3K+6      Y-K+1
    {
        /* Store previous result */
        *pointer_I_new = total / K^2;                    3            (X-K+1)(Y-K+1)
        pointer = pointer_I;                             1            (X-K+1)(Y-K+1)
        for (k_y = 0;  k_y < K; k_y++)                   2K+2         (X-K+1)(Y-K+1)
        {
            /* Remove values down
               left-hand column */
            total -= *pointer;                           2            K(X-K+1)(Y-K+1)
            pointer += K;                                2            K(X-K+1)(Y-K+1)
            /* Add in values down
               next right-hand column */
            total += *pointer;                           2            K(X-K+1)(Y-K+1)
            pointer += (X - K);                          3            K(X-K+1)(Y-K+1)
        }
        /* Move to next location */
        pointer_I_new++;                                 1            (X-K+1)(Y-K+1)
        pointer_I++;                                     1            (X-K+1)(Y-K+1)
    }

    /* Restore the total from the
       start of the previous row */
    total = total_orig;                                  1            Y-K+1
    /* Calculate position of new row start */
    pointer_I_new = &I_new[0][0] + (X * (y + 1));        4            Y-K+1
    pointer_I = &I[0][0] + (X * (y + 1));                4            Y-K+1
    /* Remove values along top row */
    pointer = pointer_I - X;                             2            Y-K+1
    for (k_x = 0;  k_x < K; k_x++)                       2K+2         Y-K+1
    {
        total -= *pointer;                               2            K(Y-K+1)
        pointer++;                                       1            K(Y-K+1)
    }
    /* Add in values along next row down */
    pointer = pointer_I - (X * (K - 1));                 4            Y-K+1
    for (k_x = 0;  k_x < K; k_x++)                       2K+2         Y-K+1
    {
        total += *pointer;                               2            K(Y-K+1)
        pointer++;                                       1            K(Y-K+1)
    }
}
```

$$(11K+11)(X-K+1)(Y-K+1) + (10K+26)(Y-K+1) + 5K^2 + 7K + 9$$

References

[1] Jean-François Abramatic and Oliver D. Faugeras. Sequential convolution techniques for image filtering. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-30(1):1–10, February 1982.

[2] Alfred A. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[3] Larry Anderson (editor). Achieving assembly flexibility for printed circuit boards. *Machine Vision World*, 7(6):20–21, December 1989.

[4] Dana H. Ballard and Christopher M. Brown. *Computer Vision.* Prentice-Hall, 1982.

[5] Lech Banachowski, Antoni Kreczmar, and Wojciech Rytter. *Analysis of Algorithms and Data Structures.* Addison-Wesley, 1991.

[6] H. Bässman and Ph.W. Besslich. *Image Processing Ad Oculos.* Springer-Verlag, 1993.

[7] Borland, Scotts Valley, CA. *Borland C++ 3.0*, 1991.

[8] John F. Canny and Kenneth Y. Goldberg. “RISC” for industrial robotics: Recent results and open problems. In *Proccedings of the IEEE International Conference on Robotics and Automation, San Diego*, May 1994.

[9] C.K. Chow and T. Kaneko. Automatic boundary detection of the left ventricle from cineangiograms. *Computers and Biomedical Research*, 5:388–410, 1972.

[10] W.F. Clocksin. An implementation of model-based visual feedback for robot arc welding of thin sheet steel. *The International Journal of Robotics Research*, 4(1):13–26, Spring 1985.

[11] P.I. Corke. Video-rate robot visual servoing. In K. Hashimoto, editor, *Visual Servoing: Real-Time Control of Robot Manipulation Based on Visual Sensory Feedback*, Robotics and Automated Systems, pages 257–283. World Scientific, 1993.

[12] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT press/McGraw-Hill, 1990.

[13] E.R. Davies. *Machine Vision: Theory, Algorithms, Practicalities.* Academic Press, 1990.

[14] L.S. Davis and Azriel Rosenfeld. Noise cleaning by iterated local averaging. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-8(9):705–710, September 1978.

[15] Drager, Friedrich, Klinge, Meyer, Reithinger, Schopmann, and Waterschek. *The World of Surface Mount Technology.* Siemens AG, 1993.

[16] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis.* Wiley, 1973.

[17] M.J.B. Duff. Pattern recognition and image processing. Lecture course given at University College, London, October–December 1990.

[18] Brian Frederick, R. Michael Carrell, Eugene M. Alexander, and Glen J. Van Sent. Handling of irregular-sized mailpieces by adaptive robotics. *IEEE Control Systems Magazine*, pages 3–7, February 1989.

[19] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*. Addison Wesley, second edition, 1992.

[20] Gregory D. Hager. Real-time feature tracking and projective invariance as a basis for hand-eye coordination. Technical Report YALEU/DCS/RR-998, Yale University Department of Computer Science, New Haven, CT, December 1993.

[21] G.H. Hardy. *A Course of Pure Mathematics*. Cambridge University Press, 10th edition, 1952 (first edition 1908).

[22] S.E. Hodges. *Look and learn: Reducing the cost of automated PCB manufacture*. PhD thesis, Cambridge University Engineering Department, Trumpington Street, Cambridge, U.K., in preparation.

[23] S.E. Hodges and R.J. Richards. Fast multi-resolution image processing for PCB assembly. In F. Deravi, editor, *IEE Colloquium on "Multi-resolution modelling and analysis in image processing and computer vision"*. IEE, April 1995.

[24] Mark Mackey. Personal Communication, January 1996. Cambridge University Chemistry Department.

[25] Microsoft Corporation, Seattle, WA. *MS-DOS 6.2 Operating System plus Enhanced Tools*, 1993.

[26] S.M. Omohundro. Efficient algorithms with neural network behavior. Technical Report UIUCDCS–R–87–1331, Department of Computer Science, University of Illinois at Urbana Champaign, Urbana, Illinois, April 1987.

[27] Alan V. Oppenheim, Ronald W. Schafer, and Thomas G. Stockham, Jr. Nonlinear filtering of multiplied and convolved signals. *Proceedings of the IEEE*, 56(8):1264–1291, August 1968.

[28] Jong-Seok Park and Julius T. Tou. A solder joint inspection system for automated printed circuit board manufacture. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1290–1295. IEEE, 1990.

[29] William K. Pratt. *Digital Image Processing*. Wiley, second edition, 1991.

[30] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[31] S. Ranade, A. Rosenfeld, and H. Samet. Shape approximation using quadtrees. *Pattern Recognition*, 15(1):31–40, 1982.

[32] Arturo A. Rodriguez and Jon R. Mandeville. Image registration for automated inspection of printed circuit patterns using CAD reference data. *Machine Vision and Applications*, 6:233–242, 1993.

[33] Azriel Rosenfeld and Avinash Kak. *Digital Picture Processing.* Academic Press, 1976.

[34] P. Saraga and B.M. Jones. Simple assembly under visual control. In A. Pugh, editor, *Robot Vision*, pages 209–233. Springer-Verlag, 1983.

[35] Günter Schiebel and Gabriela Reckewerth. Placing BGAs. *Electronic Production*, 23(9):17, October 1994.

[36] Andrew William Senior. *Off-line Cursive Handwriting Recognition using Recurrent Neural Networks.* PhD thesis, Cambridge University Engineering Department, Trumpington Street, Cambridge, U.K., September 1994.

[37] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image Processing, Analysis and Machine Vision.* Chapman & Hall, 1993.

[38] Stanley R. Sternberg. Grayscale morphology. *Computer Vision, Graphics, and Image Processing*, 35:333–355, 1986.

[39] F. F. Suess. Binary morpology. Tutorial given during Advanced Image Processing Summer School, University of Cambridge, April 1994.

[40] Godfried T. Toussaint and David Avis. On a convex hull algorithm for polygons and its application to triangulation problems. *Pattern Recognition*, 15(1):23–29, 1982.

[41] Li Wang and Theo Pavlidis. Direct gray-scale extraction of features for character recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(10):1053–1067, October 1993.

[42] Mark Allen Weiss. *Data Structures and Algorithm Analysis.* Benjamin/Cummings, 1992.

[43] Pierre David Wellner. *Interacting with Paper on the DigitalDesk.* PhD thesis, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, U.K., October 1993. Also available as Technical Report No. 330, March 1994.

[44] David Wolfe. *Direct Vision Feedback for Robot Control: the potential, the problems and a practical approach.* PhD thesis, Cambridge University Engineering Department, Trumpington Street, Cambridge, U.K., February 1991.

[45] Zhong-Qian Wu and Azriel Rosenfeld. Filtered projections as an aid in corner detection. *Pattern Recognition*, 16(1):31–38, 1983.

[46] Xinhua Zhuang and Robert M. Haralick. Morphological structuring element decomposition. *Computer Vision, Graphics, and Image Processing*, 35:370–382, 1986.