Available online at www.sciencedirect.com
SCIENCE DIRECT®

Pattern Recognition Letters 27 (2006) 556–566

Pattern Recognition Letters

www.elsevier.com/locate/patrec

Moving average algorithms for diamond, hexagon, and general polygonal shaped window operations

Changming Sun [*]

CSIRO Mathematical and Information Sciences, Locked Bag 17, North Ryde, NSW 1670, Australia

Received 29 November 2004; received in revised form 9 June 2005
Available online 26 October 2005

Communicated by Prof. L. Younes

Abstract

This paper presents fast moving window algorithms for calculating local statistics in a diamond, hexagon, and general polygonal shaped windows of an image which is important for real-time applications. The algorithms for a diamond shaped window requires only seven or eight additions and subtractions per pixel. A fast sparse algorithm only needs four additions and subtractions for a sparse diamond shaped window. A number of other shapes of diamond windows such as skewed or parallelogram shaped diamond, long diamond, and lozenged diamond shaped, are also investigated. Similar algorithms are also developed for hexagon shaped windows. The computation for a hexagon window only needs eight additions and subtractions for each pixel. Fast algorithms for general polygonal shaped windows are also developed. The computation cost of all these algorithms is independent of the window size. A variety of synthetic and real images have been tested.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Moving average algorithm; Diamond shaped windows; Hexagon shaped windows; Polygonal shaped windows; Local statistics

1. Introduction

In most of the image analysis and computer vision applications, the local processing windows are usually square or rectangular shaped. The edges of these windows are aligned with the image rows and columns. Because of the use of such simple shapes, efficient processing of images can be achieved. McDonnell (1981) described a box-filtering procedure for local mean calculation where the window is rectangular shaped. The main advantage of box-filtering is its speed, which approaches four operations for each output pixel and is independent of the box size. The filtering operation is also separable: two-dimensional filtering can be implemented as two 1D filtering.

Other shapes of windows are also used. A circular shaped window gives good isotropic property, but its computational cost is linearly proportional to the radius of the circular window. Glasbey and Jones (1997) presented fast algorithms for moving average and related filters in regular octagonal windows as approximations to circular windows. The algorithm requires twelve additions and subtractions per pixel irrespective of the window size. Ferrari and Sklansky (1984) proposed a two step method for obtaining the mean of an arbitrary shaped window. The number of operations is equal to the total number of concave and convex vertices of the window boundary. Because of the sampling effect, the boundaries of diamond and hexagon windows have many vertices, and the number of vertices also depends on the size of the window. Therefore Ferrari and Sklansky's method will not be very efficient for diamond and hexagon shaped windows. Verbeek et al. (1988) presented min or max filters for low-level image processing. They gave six shapes for the min or max filter, including a full square, a full diamond, a sampled diamond, a discrete approximation of a full circle, the rim and the center,

[*] Tel.: +61 2 9325 3207; fax: +61 2 9325 3200.
E-mail address: changming.sun@csiro.au

0167-8655/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.patrec.2005.09.020

and eight contour points and the center. The computation cost of the full diamond shaped min or max filter is proportional to the size of the window. Soille and Talbot (2001) presented a decomposition method of morphological operations for diamond shaped and rotated rectangles. van Herk (1992) also developed a fast algorithm for local min or max filters on rectangular and octagonal kernels. van Droogenbroeck and Talbot (1996) presented a general algorithm that performs basic mathematical morphology operations with any arbitrary shaped structuring element in an efficient way.

In some applications such as image processing and stereo matching, the processing window can be diamond or hexagon or general polygon shaped. In stereo matching applications, different shaped windows can be used for calculating correlation coefficients. The shapes of these windows can be adaptive to the orientation of the object boundaries. Diamond shaped window could be used when object boundary are roughly in the diagonal direction as shown in Fig. 1. Square and diamond shaped windows are shown at object boundary in Fig. 1(a) and (b), respectively. The center of the diamond window are closer to the object boundary than that of the square window without intersecting with the boundary. Baaziz and Dubois (1993) used separable diamond shaped filtering for hybrid HDTV image sequence coding. Diamond shaped window can also be imagined as a rotated version of a square window as shown in Fig. 2, although in discrete space the sides of the diamond window have zig zag shapes.

In this paper we will present fast moving windows algorithms for the calculation of local statistics such as mean, variance, skew, and correlation using a diamond or hexagon or general polygonal shaped window. The topic is important for real-time applications. Our algorithms require only seven or eight additions and subtractions per pixel, while a sparse algorithm requires only four additions and subtractions per pixel for local sums calculation for a sparse diamond shaped window. Other variations of diamond windows will also be investigated. Our algorithm for hexagon shaped window requires only eight additions and subtractions. The computational cost for general polygonal shaped window is also given as a simple formula.

The rest of the paper is organised as follows: Section 2 describes three algorithms for local mean calculation in diamond shaped windows. Section 3 gives two algorithms for sparse and multiple-shift diamond windows. Section 4 presents fast algorithms for other variations of diamond window shapes, including skewed or parallelogram shaped window, long diamond shapes, lozenge shapes, and rotated diamonds. Section 5 shows algorithms for hexagon shaped windows. Section 6 gives algorithm for general polygonal shaped windows. Section 7 describes methods for extending the local mean calculation to variance, skew, and correlation calculations. Section 8 shows the experimental results obtained using our fast algorithms applied to a variety of images. Section 9 gives concluding remarks.

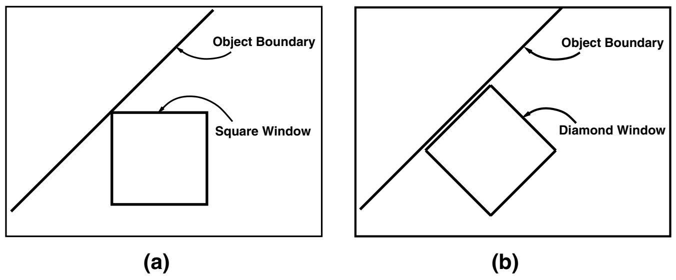

Fig. 1. Different windows at an object boundary. (a) Square window at object boundary and (b) diamond window at object boundary.

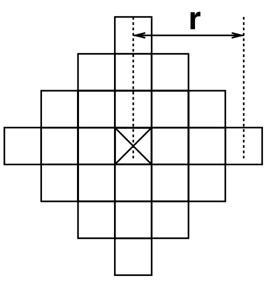

Fig. 2. Diamond shaped window. The cross is the window center, and r is the radius of the window. The value of r in this figure is 3.

2. Diamond shaped local sum calculation

In this section we propose three algorithms for obtaining the local sums in a diamond shaped window of an image. Fig. 2 shows the shape of a diamond window, and the cross in the figure indicates the window center. The size of the window is defined by its radius r. The size, or area, of the diamond shaped window with radius r is then $2r^2 + 2r + 1$. There will be one division for each pixel on the images for obtaining the mean value from the local sums. Because division operation is usually expensive, one may wish to use just the local sums. In the rest of this paper, we will concentrate on the calculation of local sums.

2.1. Edge-Updating algorithm

Assuming the local sum of a diamond window has been obtained at a particular position, when we slide the window horizontally to the right by one pixel to find the new sum, we only need to add in the pixel values from the leading edge with black circles and triangles and subtract out the pixel values from the trailing edge with white circles and triangles as illustrated in Fig. 3. For those pixels marked with circles, they lie on the lines with 45° angles from the horizontal direction. For those pixels marked with triangles, they lie on the lines with −45° angles from the horizontal direction.

Note that the sums of those pixel values on diagonal lines can be obtained using the moving window idea as shown in Fig. 4. The computational cost is only two additions and subtractions for each point on a particular line

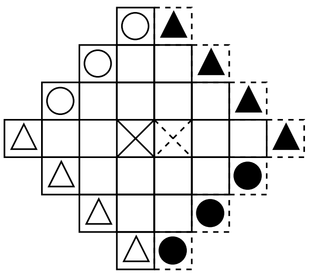

Fig. 3. Diamond shaped window updating process using 45° and −45° lines. The black circles and triangles are the image pixels coming into the diamond window and the white circles and triangles are the pixels leaving the window as the diamond window moves one pixel to the right. The cross with the dash-line indicates the new diamond window center, while the cross indicates the previous window center.

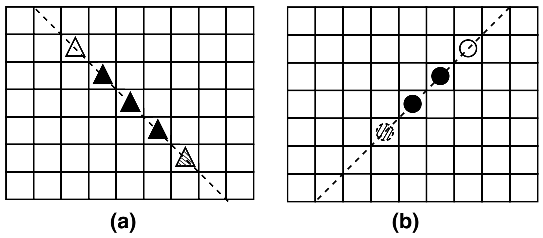

Fig. 4. The updating process for diagonal lines. Only two additions and subtractions are needed for each linear window position. (a) The −45° line, where the length of window along this line is $r+1$ and (b) the 45° line, where the length of window along this line is r.

direction irrespective of the window length. The shaded triangle as shown in Fig. 4(a) indicates the pixel value to be added and the white triangle indicates the pixel value to be subtracted from the running sum. The length of the running window for the −45° line is $r+1$. Fig. 4(b) shows the case for the 45° lines. The length of the running window is r, i.e. one pixel less than that of the −45° line.

After obtaining the running sum on the two diagonal line directions, the updating process for the local sum of a diamond shaped window can be carried out. There are four additions and subtractions for updating the diamond window sum based on the running sums of the diagonal pixels. Because there are two additions and subtractions when carrying out each of the four diagonal lines, the number of operations for diamond shaped window will be eight additions and subtractions. Note that the local sums for linear windows at the 45° and −45° at each point of the images are obtained and stored before updating the local sums for a diamond window. So the calculation of local sums for "leading" edges do not need to be carried out again when these edges become "trailing" edges. We call this algorithm for obtaining the local window sum the Edge-Updating (EU) algorithm. The algorithm steps are:

(1) Carry out the local sum for a linear window at −45° with window length $r+1$ for each pixel of the image and store the local sum values into a temporary image;
(2) Carry out the local sum for a linear window at 45° with window length r for each pixel of the image and store the local sum values into another temporary image;
(3) Obtain the pixel sum for the first position (r,r) of the diamond window on the top left of the input image by direct pixel value summation;
(4) For the rest of any particular scanline of the image, the local sum of a diamond window is obtained by adding in the local sums of the leading edges and subtract out the local sums of the trailing edges obtained in Steps 1 and 2. The window positions on the left side of the image (except the top one) can be obtained by updating the diamond window from top and bottom of the window.

2.2. Two-Grids algorithm

As can be seen from Fig. 5, the pixels in the diamond window consist of the $(r+1)\times(r+1)$ black dots and the $r\times r$ white dots (or circles) interleaved together. The centers of these two different sized grids coincide with each other. We can therefore obtain the pixel sums for the diamond window by adding together the black pixels sums and the white pixels sums, i.e. adding together the pixel values on the two different sized grids. We name this algorithm the Two-Grids (TG) algorithm.

For obtaining the sums of the white dots, we can adopt the separable principle by first adding the pixels along the −45° lines, and then along the 45° lines. Each pass only requires two additions and subtractions for each position. Similarly, the sums of the black dots can be obtained. Although the black and the white grids are interleaved as shown in Fig. 5, we need to obtain the local operations at each pixel position for both the black and the white pixels. The first pass for obtaining the black pixels sums along the −45° lines can be combined with the first pass for obtaining the white pixels sums of the same direction. That is, after obtaining the local pixel sums for the white pixels, the local sums for black pixels have already been obtained. As illustrated in Fig. 6, "Sum-r1" is the local sum from pixel values at positions "A", "B", and "C". "Sum-r2" is the local sum from pixel values at positions "B", "C", and "D" with the window shifted. "Sum-r2" is obtained

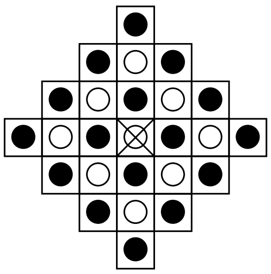

Fig. 5. Diamond composed of two grids: the black grid with $(r+1)^2$ pixels, and the white grid with r^2 pixels.

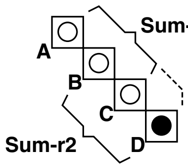

Fig. 6. Obtaining the running sum of linear window sized $r + 1$ while obtaining the running sum of linear window sized r.

by adding in "D" to "Sum-r1" to obtain a temporal value, say "Sum", and subtracting out "A" from "Sum". This temporal value "Sum" is the local running sum for the black pixels. Therefore there is no extra cost for obtaining the black pixel sums along the $-45°$ lines except an assignment operation to store the needed values. The steps of the TG algorithm can be summarised as the following:

(1) Along the $-45°$ line, update the white pixels sums with window length r (two additions and subtraction for each pixel) and obtaining the black pixels sums from the white sums (no additional operation is needed except storing the needed sum) for each point of the image; store the local sums into two different temporary images;
(2) Along the 45° lines, obtain the window sums (length r) for the white pixels (two additions and subtractions for each pixel) based on the local sums for the white pixels obtained from the previous step;
(3) Along the 45° lines, obtain the window sums (length $r + 1$) for the black pixels (two additions and subtraction for each pixel) based on the local sums for the black pixels obtained from Step 1;
(4) Finally add together the white pixel sums and the black pixel sums obtained from Steps 2 and 3 within the diamond window (one addition for each pixel).

So the total addition and subtractions for each pixel on the image is seven for obtaining the local sum within a diamond shaped window.

2.3. Grid-and-Edges algorithm

As in Fig. 7, the diamond window comprises of the grid with the black dots, the grid with the white dots (which has one pixel offset upward with respect to the black grid), the black triangles, and the white triangles. The grid size used in this case is equal to r^2 for both the black and the white grids. We can use the following steps to obtain the grid sums:

(1) Carry out the running sums along the $-45°$ lines with window length r, and store the values for each pixel. This includes the running sums for the black dots, white dots, and the black triangles;
(2) Carry out the running sums along the 45° lines with window length $r + 1$ for the white triangles, and store the values for each pixel;
(3) Obtain the small grid sums by running the linear window sum along the 45° lines based on the temporary result in Step 1;
(4) Add the small black and the white grids together;
(5) Add in the running sum for the black triangles;
(6) Add in the running sum for the white triangles.

Therefore, the total computation cost will be nine additions and subtractions per pixel on the image: 4 for obtaining the grid sums, 1 for adding the sum of the offset grid, 1 for adding the sum of the black triangles, 2 for obtaining the running sums along the 45° line, and 1 for adding the sum of the white triangles. We name this algorithm the Grid-and-Edges (GnE) algorithm.

The computational cost for each of the algorithms described in this section is invariant to the size of the window. If the window size r is very small, the direct implementation approach can be computationally cheaper than the moving average methods.

3. Sparse and multiple-shift diamonds

Our algorithms described in the previous section require seven or eight or nine additions and subtractions per pixel. In this section we propose two algorithms for efficiently obtaining the sum or mean in a sparse or a multiple-shift diamond window.

We can use the sparse black pixels shown in Fig. 8(a) to obtain the mean value. The number of black pixels in the diamond window is $(r + 1)^2$. The summation of the black pixels can be achieved by using the separable two pass process along the 45° and the $-45°$ lines. So the computation requirement is only four additions and subtractions. We call this the *sparse* algorithm. This algorithm clearly does not work with the special chess-board image because direct application of this algorithm on this image will produce complementary result. For such kind of special images, the algorithms described in Section 2 should be used. This sparse algorithm also produces spatially varying error in mean estimate compared to the algorithms in Section 2. This spatial variation is caused by different local frequency content of the image.

We can also use a window shape which contains a multiple-shift of the sparse diamond window as shown in

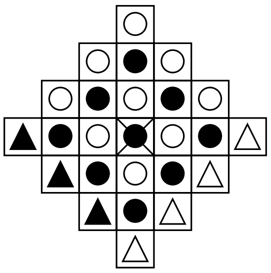

Fig. 7. Diamond sum by grid and edges. The sum of the pixels in the diamond window equals the sums of the black and white circle grids, black triangles, and white triangles.

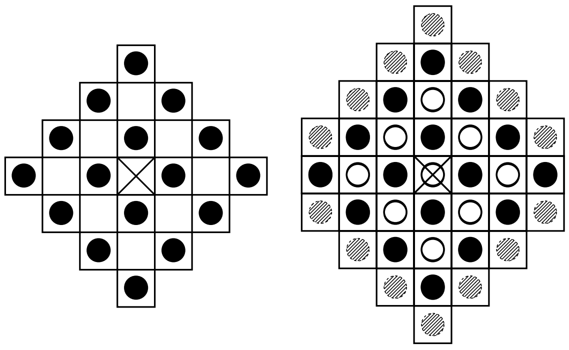

Fig. 8. (a) Sparse diamond shape for obtaining mean values and (b) multiple-shift window shape for obtaining mean values.

Fig. 8(b) for mean calculation. The drawing contains the black dots, the white dots, and the dashed dots. The white dots plus the dashed dots in the bottom half of the drawing (WD_{down}) is the same shape as the black dots (BD), except shifted one pixel down. The white dots plus the dashed dots in the top half of the drawing (WD_{up}) is the same shape as the BD, except shifted one pixel up. The mean may be obtained by averaging the sums of the pixel values in the three grids WD_{down}, BD, and WD_{up}. We call this the Multiple-Shift (MS) algorithm. The computation involves six additions and subtractions and one division for each window position in the image.

4. Variations of diamond shaped window

The edges of the diamond windows described earlier all have 45° or −45° directions. In this section we will describe some variants of the standard diamond window and propose algorithms for obtaining the local sum or mean for each of these windows.

4.1. Skewed diamond or parallelogram shape

Fig. 9 shows two diamond shaped windows which may be taken as the skewed rectangular windows. Fig. 9(a) shows a window skewed horizontally, while Fig. 9(b) shows a window skewed vertically. These shapes are actually parallelograms with one pair of edges aligned with image rows or columns. Two of the parameters are the lengths of the window edges, $k1$ and $k2$. The other two parameters, $\mathrm{d}x$ and $\mathrm{d}y$, define the skewness of the window shape, where $\mathrm{d}x$ and $\mathrm{d}y$ are integer numbers which have no common divisors. The line representation of these two sides can be carried out using Bresenham lines (Bresenham, 1965) or periodic lines (Jones and Soille, 1996). An example of a digital line segment with $\mathrm{d}x = 3$ and $\mathrm{d}y = -1$ is shown in Fig. 10(a). Fig. 10(b) shows a skewed diamond window by putting together five of the line segments shown in Fig. 10(a).

We adopt the separable principle for obtaining the pixel sums within the skewed diamond window. The first pass is for obtaining the running sums along the digital lines represented by $\mathrm{d}x$ and $\mathrm{d}y$. The second pass will carry out the running sums horizontally (for shapes similar to Fig. 9(a)) or vertically (for shapes similar to Fig. 9(b)) based on the values obtained from the previous pass. The algorithm only needs four addition and subtractions per pixel in the image. General parallelogram shapes where none of the sides is aligned with the image row or column can also be used. In this case, another pair of parameters describing the slope of the sides is needed.

4.2. Long diamond, lozenge, diamond with angle

The local window can also be a rotated rectangle, or a long diamond shape as shown in Fig. 11(a). We need four parameters to define this window: $\mathrm{d}x$, $\mathrm{d}y$, $k1$, and $k2$. $\mathrm{d}x$

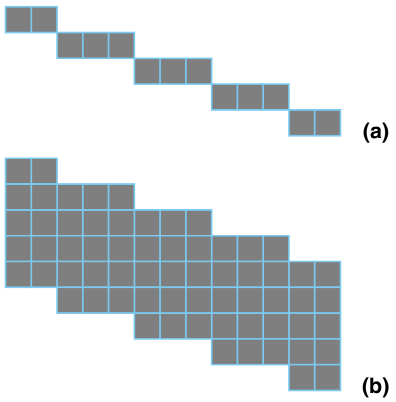

Fig. 10. (a) Digital line segment with $\mathrm{d}x = 3$ and $\mathrm{d}y = -1$ and (b) skewed window by stacking five line segments shown in (a) together.

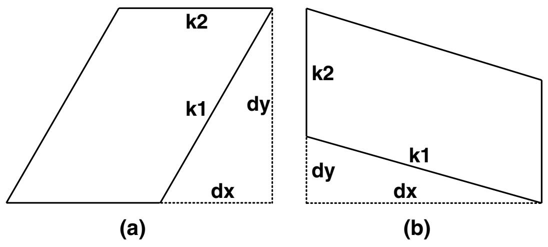

Fig. 9. Skewed diamond shapes. (a) Window skewed in the horizontal direction and (b) window skewed in the vertical direction.

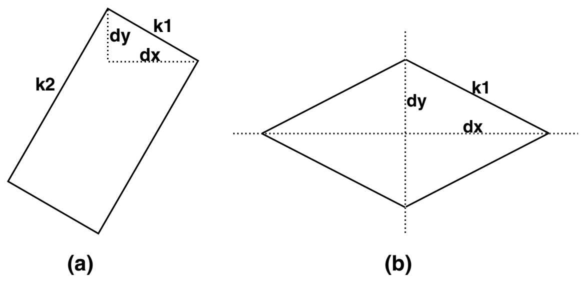

Fig. 11. (a) Elongated diamond shapes and (b) lozenged diamond shapes.

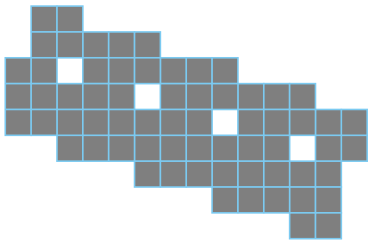
Fig. 12. Long window containing gap pixels. Digital lines along the long edge of the window have the same slope as in Fig. 10(a). The slope on the orthogonal direction is defined by d$x = 1$ and d$y = 3$.

and dy give the slope for the line with length $k1$. The lines in the orthogonal direction will have slope given by dy and $-$dx, and have length $k2$. We will then use the separable principle by carrying out the running sum along one direction first and then along the other direction. The computation cost will be four additions and subtractions per pixel in the image.

Depending on the parameters used for the diamond window, there will be situations when the window does not include all the points in the window as shown in Fig. 12. For those pixels not included in the computation, we call them the *gap* pixels. A special case is when d$x = 1$, d$y = 1$, as in the case of sparse diamond as shown in Fig. 8(a). The positions of these gap pixels may be worked out based on the shape of the digital lines used for the window; but it is not an easy task. We will leave this for future work. For the lozenge and rotated diamond shapes to be discussed later, there may also be gap pixels in the window. However, the skewed diamond shape does not have gap pixels in the window as illustrated in Fig. 10.

Another variant of the diamond window is called lozenge shaped, or rhombus shaped window, as shown in Fig. 11(b). Three parameters can define the window shape: dx, dy and $k1$. dx and dy gives the slope for the line on the top right side of the window. The slope for the line on the top left side of the window will be given by $-$dx and dy. Each side of the window has the same length $k1$. We can use the separable principle to obtain the window sums. The computation cost for this case will be four additions and subtractions per pixel position in the image.

The standard diamond can be rotated by a certain angle. However, this is a special case for the long diamond shape, as shown in Fig. 11. In this special case $k1 = k2$. The rotation angle defines the dx and dy values. The computation cost for this case will be four additions and subtractions per pixel position in the image if using the separable principal to obtain the window sums. This is similar to the case of sparse diamond shape as shown in Fig. 8(a).

5. Hexagon shaped windows

A different window shape is the hexagon as illustrated in Fig. 13. We first need to obtain the running sums along the two line directions which are not horizontal. The exact

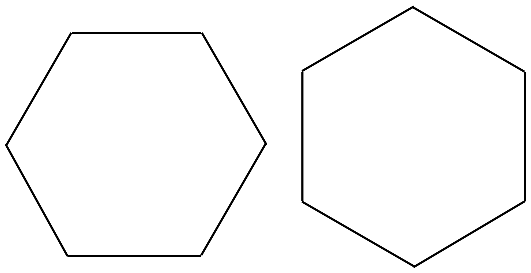
Fig. 13. Two hexagon shaped windows.

slopes for the two lines (for the left hexagon in Fig. 13) are $\tan(\pi/3)$ and $\tan(-\pi/3)$. But in a digital implementation, we need to use two integers dx and dy to represent the line slopes. The dx and dy values as illustrated in Fig. 14 can be calculated from the number of pixels of the top edge of the hexagon. If the number of pixels on the top edge is l (this number needs to be odd so that the shape of the hexagon can be symmetric), dx can take the integer value of $l/2$. dy can be calculated by using the integer part of $\sqrt{3}l/2$.

The window updating procedure in this case will be similar to the process we used for the Edge-Updating algorithm described in Section 2.1. The window length for the line direction with brick texture in Fig. 14 is dy, and the window length for the line direction with gray shade is d$y - 1$. We need four additions and subtractions for updating the running sums for the non-horizontal edges. The cost for obtaining each of the running sum for the non horizontal edge has two additions and subtractions. Therefore we have eight (2×4) additions and subtractions for each window position in the image. The local sums for linear windows along non-horizontal edges at each point of the images are obtained and stored before updating the local sums for a hexagon window. So the calculation of local sums for "leading" edges do not need to be carried out again when these edges become "trailing" edges.

Similar process can be used to obtain the local sum for the hexagon shape shown on the right of Fig. 13. In this case, we can move the hexagon window from top to bottom of the image rather than from left to right, and the slopes of the non-vertical edges are $\tan(\pi/6)$ and $\tan(-\pi/6)$.

Table 1 summarizes the number of addition and subtraction operations for each of the algorithms described

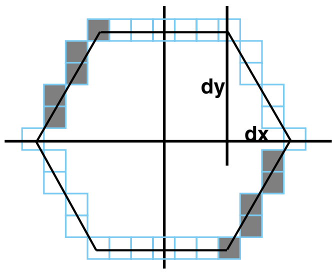

Fig. 14. Hexagon shaped window updating.

Table 1
The computation costs (numbers of additions and subtractions) for different algorithms

Radius	Diamond					Other	
	EU	TG	GnE	SD	MS	S/L/L/A	Hex
$r = n$	8	7	9	4	$6 + 1$	4	8

EU: Edge-Updating; TG: Two-Grids; GnE: Grid-and-Edges; SD: Sparse Diamond; MS: Multiple-Shift; S/L/L/A: the window shapes for skew, long, lozenge, and diamond with angle; Hex: hexagon shaped window.

in Sections 2–4. All the algorithms shown in the table have constant computation cost. That is the computation cost is invariant to window sizes. One more division operation is needed to obtain the mean from the local sum if needed (except the multiple-shift method which generates local means).

6. General polygonal shaped windows

Similar algorithms using edge updating can also be developed for general polygonal window shapes such as triangle, trapezoid, pentagon, heptagon, and decagon. Fig. 15 gives some example polygonal window shapes.

We will now derive a general expression for the computational cost for general polygonal shaped windows. Let n be the number of sides of the polygon, m be the number of sides which are aligned with the window's moving direction either horizontally or vertically, p be the number of pair of polygon sides which are parallel to each other and with the same length and not aligned with the windows moving direction. For each edge of the polygon window, there are two additions and subtractions for obtaining the running sums. Because we do not need to calculate the running sums for the m sides which are aligned with the windows moving direction. We then have $2(n - m)$ additions and subtractions for just obtaining the running sums for edges. We also need to add in or subtract out these running sums to obtain the sums for the whole window. Then we have $3(n - m)$ additions and subtractions for a general polygonal window. If there are p pairs of edges which are in the same direction and having the same length (as for the opposite edges in the shape of hexagon or octagon windows), then redundant calculation of $2p$ for the running sums can be eliminated. Then the computation cost, C, is

$$C = 3(n - m) - 2p \tag{1}$$

The computational cost is related to the number of sides, but is independent of the size of the polygonal windows. This formula holds for all the regular even sided polygons with opposite sides parallel to each other.

Some example values of n, m, p, and C for different polygonal shaped windows is given in Table 2. Take the hexagon shape for example, the number of sides, n, is 6. The number of sides that are aligned with the windows moving direction, m, is 2. The number of pairs of parallel sides/edges having the same length, p, is 2. Therefore based on Eq. (1) the computation cost is eight additions and subtractions for a hexagon shaped window.

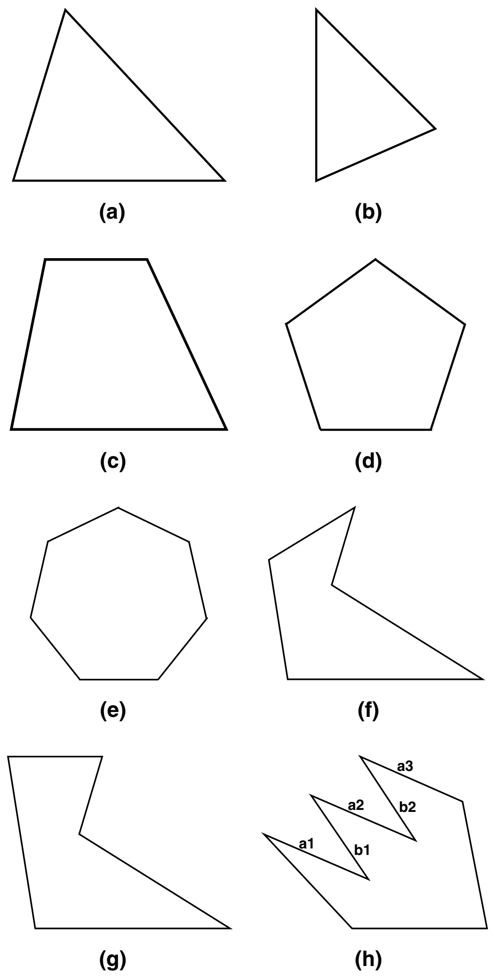

Fig. 15. Some examples of polygonal shapes. (a) Triangle with a horizontal edge; (b) triangle with a vertical edge; (c) trapezoid; (d) pentagon; (e) heptagon; (f) general polygon with one horizontal edge; (g) general polygon with two horizontal edges; (h) general polygon with two sets of parallel edges (one set with $a1$, $a2$, and $a3$; another set with $b1$ and $b2$).

For polygon windows with a set of parallel and equal length edges, if this set contains more than two edges as indicated by $a1$, $a2$, and $a3$ in Fig. 15(h), further computation redundancy exists and can be eliminated. A more general formula than that expressed in Eq. (1) for C is

$$C = 3(n - m) - \sum_{i=1}^{p} 2(n_i - 1) \tag{2}$$

Table 2
Example numbers for n, m, p, and C for different polygonal shaped windows

Polygon shape	n	m	p	C
Triangle (Fig. 15(a))	3	1	0	6
Square/rectangle	4	2	1	4
Trapezoid (Fig. 15(c))	4	2	0	6
Pentagon (Fig. 15(d))	5	1	0	12
Hexagon (Fig. 13)	6	2	2	8
Heptagon (Fig. 15(e))	7	1	0	18
Octagon	8	2	3	12
Polygon (Fig. 15(f))	5	1	0	12
Polygon (Fig. 15(g))	5	2	0	9

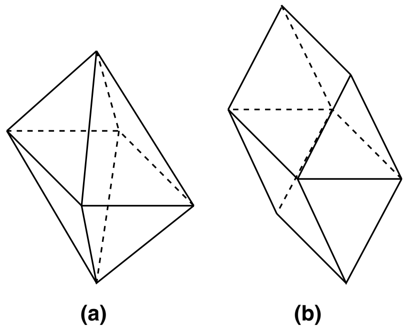

Fig. 16. 3D diamond shapes. (a) Non-separable and (b) separable.

where $n_i > 1$ is the number of parallel edges with equal length within a set, and p is the number of such sets. If all the n_i's equal 2, Eq. (2) becomes Eq. (1).

We can also consider a 3D diamond shaped window for 3D images. Fig. 16(a) shows a 3D diamond shape which is not separable, while Fig. 16(b) shows a different 3D diamond shape which is separable. In this case the 3D local sum can be obtained by first accumulating the 2D sum using the diamond shaped window, and then carrying out the sum in the orthogonal direction based on the sums already obtained.

7. Other types of operations

Local pixel variance within a diamond, hexagon or polygonal shaped window can also be obtained quickly during the same process as when calculating the local sum or mean as described in Sections 2–6. This is achieved by accumulating the square of the intensity values while accumulating original pixel values for mean calculation. f_{ij} is the image pixel value at position i, j. D is the set of pixels within the local window shape, and N is the number of pixels in D. The first term $\sum_{m,n\in D} f_{m,n}^2$ at the r.h.s. of the following equation for variance calculation can be obtained during the same process as the local sum or mean is calculated.

$$\mathrm{var}_{ij}(f) = \frac{1}{N}\sum_{m,n\in D}(f_{m,n} - \bar{f}_{ij})^2 = \frac{1}{N}\sum_{m,n\in D} f_{m,n}^2 - \bar{f}_{ij}^2$$

where $\bar{f}_{ij}$ is the local mean. High-order statistics such as local skew $1/N\sum_{m,n\in D}(f_{m,n} - \bar{f}_{ij})^3$ can also be obtained using the following equation:

$$\mathrm{skew}_{ij}(f) \quad = \quad 1/N \sum_{m,n\in D} f_{m,n}^3 - 3S\bar{f}_{ij} + 2\bar{f}_{ij}^3 \tag{3}$$

where $S = 1/N\sum_{m,n\in D} f_{m,n}^2$.

Local cross correlation within a diamond, hexagon or polygonal windows for two images can also be obtained efficiently. Cross correlation coefficients can be used as a reliable measure of similarity for matching purposes. The cross correlation can be defined as

$$\mathrm{cov}_{ij,d_x d_y}(f,g) = \sum_{m,n\in D} f_{m,n} \times g_{m+d_x,n+d_y} - N\bar{f}_{ij} \times \bar{g}_{i+d_x,j+d_y}$$

where d_x and d_y are the shifts for image g along the x and the y axes. The first term of the r.h.s. of above equation is the summation of the pixel multiplications over the correlation window with the image g shifted. Similar to the process of calculating the variance, the multiplication of $f_{m,n} \times g_{m+d_x,n+d_y}$ is used rather than $f_{m,n}^2$. The second term of the r.h.s. of the above equation is straight-forward calculation using the available mean values. The local cross correlation in the above equation can be normalised with the local variance.

The fast algorithm described for cross correlation can also be easily adopted to obtain the sum of absolute difference (SAD) (using $\mathrm{SAD}_{ij,d_x d_y}(f,g) = \sum_{m,n\in D}|f_{m,n} - g_{m+d_x,n+d_y}|$) and sum of squared differences (SSD) (using $\mathrm{SSD}_{ij,d_x d_y}(f,g) = \sum_{m,n\in D}(f_{m,n} - g_{m+d_x,n+d_y})^2$) measures efficiently. See (Sun, 2001) for fast algorithms using rectangular shaped windows for local statistics in N-dimensional images and see (Sun, 2002) for correlation calculation for stereo matching.

Soille and Talbot (2001) proposed methods for directional morphological filtering at arbitrary angles. They also proposed decomposition based methods for diamond shaped windows and rotated rectangles min or max operation. The computation cost of their algorithms is six comparisons for sparse diamond and ten comparisons for full diamond shapes. We can adopt the techniques for the rotated rectangle algorithms in (Soille and Talbot, 2001) for our skewed and lozenge window shapes min or max operations. Only six comparisons are needed for each pixel.

8. Experimental results

This section shows some of the test results obtained using our new algorithms described in this paper. A variety of images have been tested, including synthetic images, and different types of real images.

8.1. Image tests

Fig. 17 shows some obtained results using different algorithms developed in this paper. Fig. 17(a) is the input image. Fig. 17(b)–(e) gives the results obtained by using

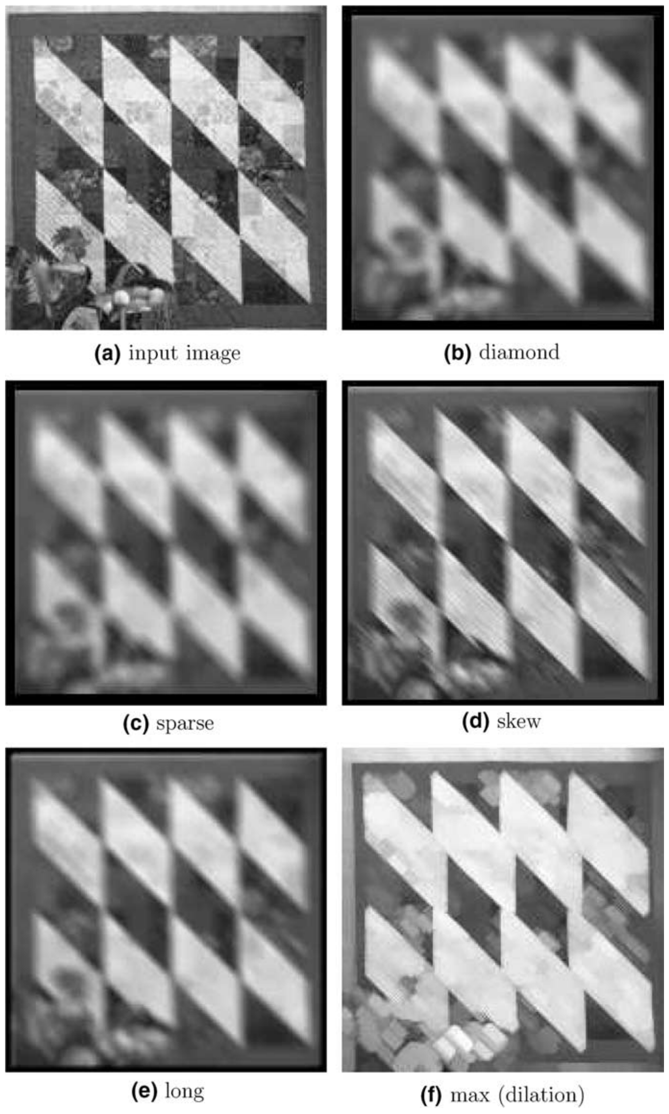

Fig. 17. The mean filtering results using different shapes of windows. (a) Input image; (b) using diamond window using the Edge-Updating algorithm ($r = 5$); (c) result using a sparse window ($r = 5$); (d) result using skew in x shaped ($dx = 1, dy = 1, k1 = 7, k2 = 1$); (e) result using long diamond ($dx = 1, dy = 1, k1 = 7, k2 = 3$); and (f) result using max filter ($dx = 1, dy = 1, k1 = 7, k2 = 3$).

diamond, sparse diamond, skew diamond, long diamond for local mean calculation. The results in Fig. 17(b) and (c) are very similar after using diamond and sparse diamond window filtering. Fig. 17(d) is essentially a linear averaging along the 45° line. Fig. 17(f) shows the result by running the max filter or dilation operation.

8.2. Different results for diamond, sparse diamond and multiple-shift algorithms

Table 3 gives the results showing the differences between the diamond and the sparse diamond and multiple-shift diamonds for different images with different window sizes. The errors are the average intensity differences across the whole image. Larger window sizes seem to give smaller errors. The errors are also depend on the contents of the image. Random image tends to generate larger errors. Smooth or low variation images usually gives smaller errors.

Table 3
Average errors for each pixel when using Sparse and Multiple-Shift algorithms

Image names	Window sizes (r)	SD error	MS error
Random1	3	8.820	4.306
	7	4.848	1.420
	11	3.338	0.755
Liz	3	2.150	2.224
	7	1.339	1.323
	11	0.882	0.862
Pentagon	3	0.922	1.037
	7	0.583	0.591
	11	0.421	0.415
Diamond	3	0.775	0.995
	7	0.567	0.642
	11	0.468	0.507

SD: Sparse Diamond algorithm; MS: Multiple-Shift algorithm.

8.3. Running times

The speed of the algorithms were tested on a 1.7 GHz Pentium 4 PC running Linux. The implementation language is C. The programs for each algorithm were run for several hundred times and the average running time is obtained. The typical running time for the fast algorithms on a 1024×1024 image is in the order of 1 s. Table 4 gives some of the typical running times of different algorithms on different size of windows. The second column gives the running times for the local mean calculation in a diamond window using direct implementation. The running time is in accordance with the computation cost of $2r^2 + 2r$, apart from that of the very small radius size due to implementation overheads. The last three columns clearly show that the computation costs of the EU, TG, and GnE algorithms are invariant to window sizes. The slight reduction of computation time with the increase of window size is due to the boundary effect. That is a smaller number of pixels need to be processed if a larger window is used. The boundary pixels where the local window does not fit into the image can be processed by enlarging or padding the input image using mirror reflection at the image boundaries. One can also calculate the local window mean values in this case using

Table 4
Running times of different algorithms with different window sizes (s)

Window sizes (r)	Drct	EU	TG	GnE
1	0.147	0.483	1.039	0.640
4	0.396	0.475	1.032	0.635
7	0.781	0.468	1.008	0.635
10	1.381	0.467	1.007	0.631
13	2.133	0.467	1.005	0.629

Image size is 1024×1024.

direct averaging of those pixels that lie within the input image. Although the computation costs for the EU algorithm is one more than that of the TG algorithm as shown in Table 1, the memory storage and the number of runs going through the image are different. Therefore we can see the different running times for the EU and TG algorithms as shown in the third and fourth columns of Table 4.

It should be mentioned that the proposed algorithms have more memory requirements compared with direct implementations, as shown in Table 5. The window size does not affect the memory usage of the algorithms. Direct implementation requires the least memory usage. Among the three algorithms of EU, TG, and GnE, the EU algorithm requires the least memory usage. The EU algorithm is also the fastest among these three algorithms. The extra memory requirements are for the storage of the different line directions of local sums. For most of the diamond and hexagonal windows, two temporary images are required for storing the linear local sums. This should be an issue to consider when carrying out hardware implementation of such algorithms.

Schutte and van Kempen (1997) presented three methods based on transposing the image to improve the data cache usage for separable image processing algorithms on general purpose workstations. In their case, the processing was along the image rows or columns or slices for 3D images. In our case, almost all the processings are along lines which are not aligned with image rows or columns. This makes it difficult to use Schutte and van Kempen's method to improve cache use. For diamond window operations, it may be possible to rotate the image so that the original diamond window becomes two different sized squares which are aligned with image row and columns as illustrated in Fig. 18. The diamond window becomes two interleaving squared windows. But overhead is involved for such transformations. The size of the transformed image becomes larger and there are undefined values around the boundaries of the transformed image.

Table 5
Memory usages (heap peak in Mb) of different algorithms with different window sizes

Window sizes (r)	Drct	EU	TG	GnE
4	2.097	10.498	18.894	14.696
10	2.097	10.498	18.894	14.696

Image size is 1024 × 1024.

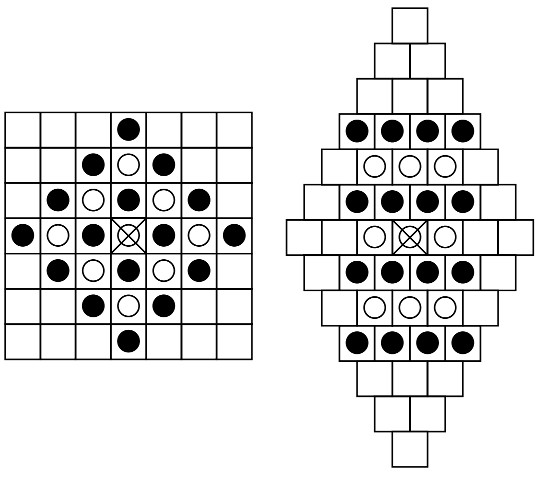

Fig. 18. Diamond shapes after rotation transformation so that the black and the white grids lie along the image row and column directions.

8.4. Translation-invariant issue

To achieve translation-invariant operation for general discrete lines, periodic lines can be used rather than Bresenham lines (Soille et al., 1996; Jones and Soille, 1996; Glasbey and Jones, 1997; Soille and Talbot, 2001). For edges of polygonal windows, one needs to make sure that the edges can be represented as periodic lines (Glasbey and Jones, 1997). But this can be restrictive for general polygonal shapes. Glasbey and Jones (1997) suggested, in this case, to use edges that are separable into cascades of two orthogonal periodic lines. Soille and Talbot (2001) presented algorithms for translation-invariant morphological filtering.

9. Conclusions

We have developed several fast moving average algorithms for several types of operations using diamond shaped window. The algorithms require seven or eight or nine additions and subtractions for each pixel on the image. A fast sparse algorithm only needs four additions and subtraction for each pixel of the image. The algorithms are also extended for a variety of other shaped diamond windows, such as long diamond, skewed diamond, and lozenge shapes. Similar algorithms are also developed for hexagon shaped windows. The computation for hexagon window only needs eight additions and subtractions for each pixel. Fast algorithms for general polygonal shaped windows operation are also developed. The computation cost of all these algorithms are independent of the window size.

Acknowledgements

The author thanks the anonymous referees and Dr. Hugues Talbot and Dr. Michael Buckley of CSIRO for their valuable comments and suggestions.

References

Baaziz, N., Dubois, E., 1993. Separable diamond-shaped filtering for hybrid HDTV image sequence coding. In: 8th Workshop on Image and Multidimensional Signal Processing. Cannes, France, pp. 162–163.

Bresenham, J.E., 1965. Algorithm for computer control of digital plotter. IBM Systems J. 4 (1), 25–30.

Ferrari, L., Sklansky, J., 1984. A fast recursive algorithm for binary-valued two dimensional filters. Computer Vision, Graphics, and Image Process. 26 (3), 292–302.

Glasbey, C., Jones, R., 1997. Fast computation of moving average and related filters in octagonal windows. Pattern Recognition Lett. 18 (6), 555–565.

Jones, R., Soille, P., 1996. Periodic lines: Definition, cascades, and application to granulometries. Pattern Recognition Lett. 17 (10), 1057–1063.

McDonnell, M.J., 1981. Box-filtering techniques. Computer Graphics and Image Process. 17 (1), 65–70.

Schutte, K., van Kempen, G., 1997. Optimal cache usage for separable image processing algorithms on general purpose workstations. Signal Process. 59 (1), 113–122.

Soille, P., Breen, E., Jones, R., 1996. Recursive implementation of erosions and dilations along discrete lines at arbitrary angles. IEEE Trans. Pattern Anal. Machine Intell. 18 (5), 562–567.

Soille, P., Talbot, H., 2001. Directional morphological filtering. IEEE Trans. Pattern Anal. Machine Intell. 23 (11), 1313–1329.

Sun, C., 2001. Fast algorithm for local statistics calculation for *N*-dimensional images. J. Real-Time Imaging 7 (6), 519–527.

Sun, C., 2002. Fast stereo matching using rectangular subregioning and 3D maximum-surface techniques. Internat. J. Computer Vision 47 (1/2/3), 99–117.

van Droogenbroeck, M., Talbot, H., 1996. Fast computation of morphological operations with arbitrary structuring elements. Pattern Recognition Lett. 17 (14), 1451–1460.

van Herk, M., 1992. A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels. Pattern Recognition Lett. 13 (7), 517–521.

Verbeek, P.W., Vrooman, H.A., van Vliet, L.J., 1988. Low-level image processing by max-min filters. Signal Process. 15 (3), 249–258.