appeared in
10. Workshop Farbbildverarbeitung 2004, Koblenz, Online-Proceedings
http://www.uni-koblenz.de/icv/fws2004/

An Efficient Vector Median Filter Computation

V. Hong[1], L. Csink[2], S. Bouattour[1], D. Paulus[1]

[1] Institut für Computervisualistik, Universität Koblenz-Landau,
Universitätsstr. 1, 56070 Koblenz – Germany,
{hong,bouattour,paulus}@uni-koblenz.de
[2] John von Neumann Faculty of Informatics, Budapest Polytechnic,
H1037 Budapest, Nagyszombat 19 – Hungary,
csink.laszlo@nik.bmf.hu

Abstract. The vector median filter(VMF) is a useful and common tool for noise removal in color image processing. The drawback of the approach in its general form is the high computational cost if implemented naively. Indeed it takes $O(n \times m \times m_{\mathrm{s}}^4)$ distance evaluations for computing the filter response, where m_{s} ist the mask size and $n \times m$ the dimension of the image. In this contribution we propose a general algorithmic optimization framework for reducing the complexity to $O(n \times m \times m_{\mathrm{s}}^3)$. We discuss the dependency of the VMF on the distance measure and demonstrate that for the square Euclidian distance, the VMF becomes a vector *near-mean* filter with just $O(n \times m \times m_{\mathrm{s}}^2)$ distance evaluations.

1 Introduction

By definition the VMF as used in [2, 3, 1] of an $n \times m$ RGB image $\boldsymbol{f}$ is computed as follows: Go through the image with a square mask W of size $m_{\mathrm{s}} \times m_{\mathrm{s}}$. The VMF $\boldsymbol{g}$ of $\boldsymbol{f}$ at the center of the mask (i, j) is

$$\boldsymbol{g}_{i,j} = \boldsymbol{f}(i_{Min}, j_{Min}) \tag{1}$$

where (i_{Min}, j_{Min}) is the position of the pixel with the minimal distance sum to all other pixels within the mask:

$$R_{i_{Min},j_{Min}} = \min_{(\mu,\nu)\in W} R_{\mu,\nu} \tag{2}$$

where

$$R_{\mu,\nu} = \sum_{(k,l)\in W} \text{Distance}(\boldsymbol{f}_{\mu,\nu}, \boldsymbol{f}_{k,l}) \; . \tag{3}$$

and $Distance$ is the distance function between two pixels.

According to this definition, a mask of size $m_{\mathrm{s}} \times m_{\mathrm{s}}$ is applied to each pixel in the image of size $n \times m$. For each pixel within the mask the distance between this pixel to all other pixels in the mask have to be computed. This costs $O(m_{\mathrm{s}} \times m_{\mathrm{s}})$ evaluations. This has to be repeated m_{s}^2 times within the mask and over the whole image. VMF's computation needs thus $O(n \times m \times m_{\mathrm{s}}^4)$ distance evaluations. This high computational cost is the main drawback of the VMF, especially when it is applied to real time applications.

In the following we discuss two approaches to reduce the computational complexity. In section 2 we propose a general algorithmic framework for reducing the number of evaluations by preserving the computations performed in one position and using it in the following one. This leads to a reduction of the order of $O(n \times m \times m_s^3)$. In section 3 we discuss the dependency of VMF on a special distance measure: the square Euclidian distance, and demonstrate that the minimium of Eq. 2 can be computed analytically. This yields a smoothing filter which is actually the best approximation of the mean filter. We call this the vector near-mean filter. Section 4 summarizes the paper and describes future work.

2 Algorithmic Optimization

The most expensive part of the original VMF is the computation of the pixel distances. For each mask applying the distances between each pixel in the mask and its neighbours are computed completely new, although most of the distances are already computed before. In our approach we reduce the number of pixel distance computations by reusing computed distances between pixels and those neighbours. We introduce a distance matrix and a distance cube.

While applying masks, we go through the image columnwise from left to right. Furthermore we ignore border cases.

2.1 Distance Matrix

To reduce the frequency of the expensive distance computation between pixels we introduce a symmetric matrix $D = [D(i,j,k,l)]_{i,j,k,l=0,\ldots,m_s-1}$. In this matrix the distances between pixels in a mask are stored. The pixel distances computed in this matrix can be reused for several mask operations. A matrix element $D(i,j,k,l)$ represents the distance between the pixels at mask position (i,j) and (k,l). The distance between two pixels is direction independent, that means $D(i,j,k,l) = D(k,l,i,j)$ is valid for all pixels in the mask. To increase the reuseability of this matrix we make a difference between horizontal and vertical pixelwise moving of the mask.

Initialization When the mask is applied on the image for the first time the distance matrix has to be completely computed. The complete computation of the mask is done only once. This process is divided up into two parts, determine the distances between pixels being in the same and different rows. In Fig. 1 the calculation of the distances between pixels at the same row in the mask are presented. The computational time of this step is $O(m_s^3)$.

In Fig. 2 the calculation of the distances between pixels at different rows in the mask are shown. We iterate through each row in the mask from top to bottom and compute foreach pixel the distance to each neighbours on the lower rows. The computational time of this step is $O(m_s^4)$.

```
∀ i = 1..m_s  // Iterate through each row
    ∀ j = 1..m_s − 1  // Iterate through columns
        ∀ l = j + 1..m_s  // Iterate through columns
            D(i, j, i, l) = Distance(f(i, j), f(i, l));
            D(i, l, i, j) = D(i, j, i, l);  // Symmetrical case
```

Fig. 1. Computation of distances between pixels at the same row

```
∀ i = 1..m_s − 1  // Iterate through rows
    ∀ j = 1..m_s − 1  // Iterate through columns
        ∀ k = i + 1..m_s  // Iterate through rows
            ∀ l = 1..m_s  // Iterate through columns
                D(i, j, k, l) = Distance(f(i, j), f(k, l));
                D(k, l, i, j) = D(i, j, k, l);  // Symmetrical case
```

Fig. 2. Computation of distances between pixels at the different rows

Horizontal moving After scanning an image along a column j we continue the scanning with the next right column $j + 1$ from the top. Originally we have to compute the whole distance matrix at the beginning of a new column. But luckily, we can reduce the costs of building this matrix. Before scanning an image vertical we compute recursively the distance matrices for each starting column from left to right and store these results. If we move the mask from a starting column to the next, most of the already determined distances in the previous distance matrix can be reused. We cannot reuse the distance for the pixels at the left mask side of the previous mask, because these pixels are no more in the current mask. The other distances can be reused in the way that the column index of the elements in previous matrix are increased by 1(see Fig. 3). Finally we have to compute for each pixel at the right mask side the distances to its neighbours(see Fig. 4).

The computation cost of Fig. 3 is $O(m_s^4)$.

```
∀ i = 1..m_s  // Iterate through each row
    ∀ j = 1..m_s − 1  // Iterate through columns
        ∀ k = 1..m_s  // Iterate through rows
            ∀ l = 1..m_s − 1  // Iterate through columns
                DNew(i, j, k, l) = D(i, j + 1, k, l + 1)
```

Fig. 3. Reusing the distance information of the previous distance matrix – horizontal moving

The computation cost of Fig. 4 is $O(m_s^4)$. Here sM_{Col} is an auxiliary variable keeping track of the column position to start from.

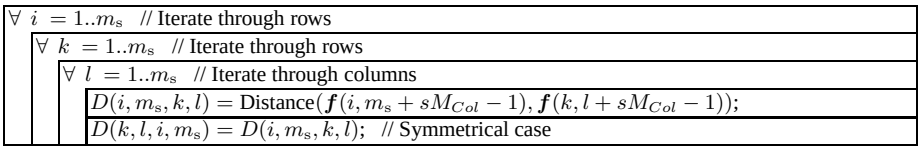

Fig. 4. Computation of the distances between pixels in the last row mask and its neighbours – horizontal moving

Vertical Moving For the vertical moving of the mask the approach is very similar to the horizontal moving. We can keep most of the distance information of the previous computed matrix (see Fig. 5). Only for the pixels in the last row of the mask the distances to its neighbours must be computed(see Fig. 6).

The computation cost of Fig. 5 is $O(m_s^4)$.

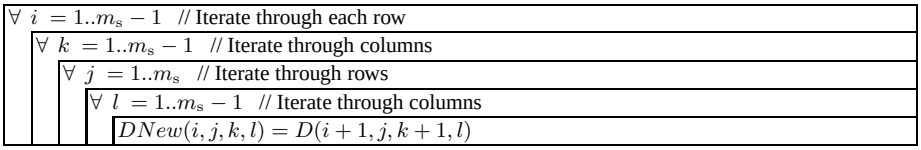

Fig. 5. Reusing the distance information of the previous distance matrix – vertical moving

The computation cost of Fig. 6 is $O(m_s^2)$. Here sM_{Row} and sM_{Col} are auxiliary variables keeping track of the row and column positions, respectively, to start from.

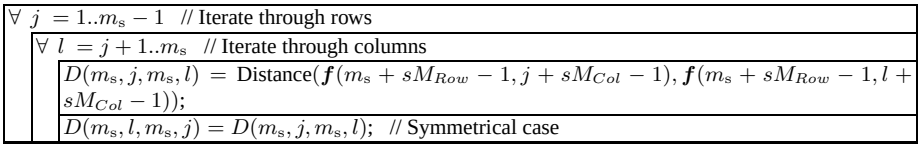

Fig. 6. Computation of the distances between pixels in the last row mask and its neighbours – vertical moving

2.2 E-Cube

In the standard VMFD we have to compute for each pixel in the mask the sum of the row distances between this pixel and its neighbours in the corresponding mask row. In

general, the computed row distances can be used many times again. For this purpose we introduce the E-Cubethat is a matrix of the dimension m_s^3. Each element $E(i, j, s)$ represents the distance between the element (i, j) in the mask to all its neighbours in the mask row s.

Initialization For each element in the row i and column j the distance $E(i, j, k)$ to all its neighbours in the row k is computed by using the distances matrix (see Fig. 7).

```
∀ s = 1..m_s  // Iterate through each slice
    ∀ i = 1..m_s  // Iterate through rows
        ∀ j = 1..m_s  // Iterate through columns
            ∀ l = 1..m_s  // Iterate through columns
                E(i,j,s) = E(i,j,s) + D(s,j,i,l)
```

Fig. 7. Initialization of the E-Cube

Horizontal Moving If the mask is shifted to the next vertical position, then most of the computed distances in the E-Cube can be reused in the following way. All elements in the E-Cube are moved in the way that the column index is decreased by one. The value of each element $ENew(i, j, s)$ must be updated in two steps. First, the distance between the pixel (s, j) and $(i, 1)$ is removed, because the pixel $(i, 1)$ is no longer valid. Second, the distance between the pixel (s, j) and (i, m_s) is added, because the pixel (i, m_s) is new in the mask.

Additionally, for each pixel in the right column the distances pixels have to be computed new.

In Fig. 8 is the algorithm of the horizontal shifting described.

```
∀ s = 1..m_s  // Iterate through each slice
    ∀ i = 1..m_s  // Iterate through each row
        ∀ j = 1..m_s − 1  // Iterate through columns
            ENew(i,j,s) = E(i,j+1,s) − DOld(s,j+1,i,1) + D(s,j,i,m_s)
            // Remove old distance between current pixel and left pixel in previous (left) mask
            // Add distance between current pixel and right pixel in current mask
        ∀ l = 1..m_s  // Iterate through columns
            ENew(i,m_s,s) = E(i,j+1,s) + D(s,m_s,i,l)
             // Compute the right pixel in current row new
```

Fig. 8. Horizontal Shifting of the E-Cube

Vertical Moving The vertical shifting of the E-Cube is analogous to the horizontal shifting in 2.2.

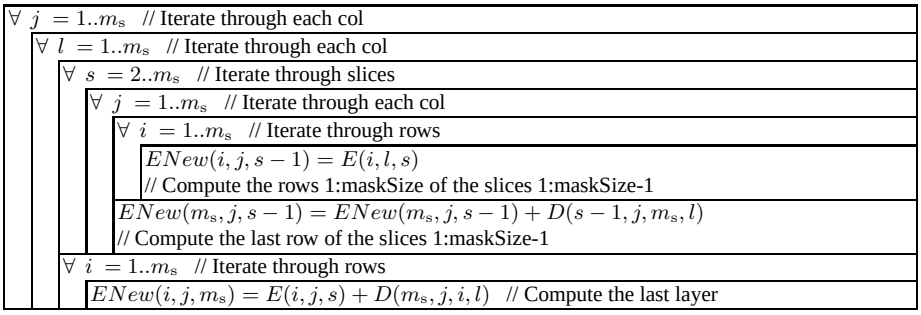

Fig. 9. Vertical Shifting of the E-Cube

2.3 Main Algorithm

Fig. 10 outlines the main algorithm of our approach. This algorithm consists of the following parts: first initialization, then columnwise scanning of the image from left to right.

In the initialization step the mask starts at the position $(startRow, startCol)$. The distance matrix and the E-Cubeare created (see sect. 2.1 and 2.2) and stored in the auxiliary lists $DInits$ resp. $CubeList$. After this step the mask is moved horizontal from left to right. For each mask moving the distance matrix and E-Cubeare updated by using the horizontal moving algorithm (see sect. 2.1 and 2.2). The results are appended to the lists $DInits$ resp. $CubeList$.

After the initialization the mask is moved to the position $(startRow, startCol)$. Then the image is scanned columnwise. For each new column the distance matrix and E-Cubeare recomputed by using the corresponding element of the lists $DInits$ resp. $CubeList$. For each column the new value of the applied pixel is computed in the $computeMinPixel$ function using current distance matrix and E-Cube. After each pixel apply the distance matrix and E-Cubeare updated by using the in sect. 2.1 and 2.2 presented algorithms.

The function $computeMinPixel$ returns the pixel in the mask with minimal distance to all its neighbours by using the e-cube.

3 Analytic Optimization

When the VMF is applied on one-dimensional pixel values, the computation of the distances over each two pixels becomes unnecessary since it can be shown that the

```
sM_Col = 1
startCol = (m_s + 1)/2
endOfColLoop = cols − (m_s − 1)/2
startRow = (m_s + 1)/2
endOfRowLoop = rows − (m_s − 1)/2
stripeSize = endOfColLoop − startCol + 1
DInits[1] = createDistMatrix(m_s, f)   // Create first distance matrix
CubeList[1] = createECube(m_s, f)   // Create first e-cube
∀ i = 2..stripeSize   // Iterate through the stripes
    // Compute recursive the other distance matrices
    DInits[i] = shiftHorDistMatrix(i + sM_Col − 1, m_s, D, f)
    // Compute recursive the other e cubes
    CubeList[i] = shiftHorECube(i + sM_Col − 1, m_s, DInits[i − 1], DInits[i], f)
∀ s = 1..m_s   // Iterate through each slice
    ∀ i = 1..m_s   // Iterate through rows
        ∀ j = 1..m_s   // Iterate through columns
            ∀ l = 1..m_s   // Iterate through columns
                E(i, j, s) = E(i, j, s) + D(s, j, i, l)
∀ j = 1..endOfColLoop   // Iterate through the stripes
    sM_Row = 1
    sM_Row = 1
    g(startRow, j) = computeMinPixel(sM_Row, sM_Col, E(, , m_s, , )f)   // Compute new pixel
    ∀ i = 1 + startRow..endOfRowLoop   // Iterate through the rows of the stripe
        sM_Row + +
        DInits[j] = updateDistMatrix(sM_Row, sM_Col, DInits[j], m_s, f)
        DInits[j] = updateECube(sM_Row, sM_Col, CubeList[j], m_s, f)
        g(startRow, j) = computeMinPixel(sM_Row, sM_Col, E(, , m_s, , )f)
          // Compute new pixel
    sM_Col + +
```

Fig. 10. Main Algorithm

median of the mask is the solution of the optimization problem of equation 2. It has to be emphasized that this is fulfilled, only if the the absolute values of differences between each pixel pair is the used distance measure in equation 3.

For color images, pixel values are three-dimensional vectors. The application of the median rule is no more valid. In the following we will demonstrate that even in this case we can solve the optimization problem analytically under the assumption of a special distance measure: the *square Euclidian distances*.

The equation to be minimized in the case of square Euclidian distance is:

$$R_{\mu,\nu} = \sum_{k,l \in W} \| \boldsymbol{f}_{\mu,\nu} - \boldsymbol{f}_{k,l} \|^2. \tag{4}$$

The minimum can be computed by finding the vector that sets the first derivative with respect to $\boldsymbol{f}_{\mu,\nu}$ to zero:

$$\frac{d\boldsymbol{R}_{\mu,\nu}}{\boldsymbol{f}_{\mu,\nu}} = \sum_{k,l \in W} 2(\boldsymbol{f}_{\mu,\nu} - \boldsymbol{f}_{k,l}) \stackrel{!}{=} 0 \tag{5}$$

$$\Longrightarrow \qquad m_{\mathrm{s}}^2 \hat{\boldsymbol{f}}_{\mu,\nu} - \sum_{k,l \in W} \boldsymbol{f}_{k,l} = 0$$

$$\Longrightarrow \qquad \hat{\boldsymbol{f}}_{\mu,\nu} = \frac{1}{m_{\mathrm{s}}^2} \sum_{k,l \in W} \boldsymbol{f}_{k,l} \tag{6}$$

Eq. 6 shows that the vector that minimizes Eq. 2 for the square Euclidian distance is the *mean* vector within the mask. As the mean value can have values which are not existent in the image, the best approximation is the vector which has the smallest square Euclidian distance to the mean vector in the mask. In this case the number of evaluations of the distance measure over the whole image is reduced to $O(n \times m \times m_{\mathrm{s}}^2)$. The computation of the mean vectors grows linearly with the mask size.

Analytical solutions for other distance measures have to be considered in dependency with the properties of the measure. If it is not possible to compute the solution in a closed form, the algorithmic optimization decreases in any case the computational costs.

4 Conclusion

In our paper we propose two different ways of improving the computational time of the VMF ($O(n \times m \times m_{\mathrm{s}}^4)$). Although there are some elements of the algorithm of $O(m_{\mathrm{s}}^4)$, but these are not multiplied by $n \times m$ - this is a key point of the algorithmic improvement.

The algorithmic way is independent of the used distance function. It reduces the computational time by reusing computed pixel distances ($O(n \times m \times m_{\mathrm{s}}^3)$). The analytic way works only for the square Euclidian distance. This approach reduces the computational costs of the VMF to $O(n \times m \times m_{\mathrm{s}}^2)$.

Further work remains to be done on extending the analytic way to any distance function.

References

1. Jaakko Astola, Pekka Haavisto, and Yrjo Neuvo. Vector median filters. *Proceedings of the IEEE*, 78:678–689, 1990.
2. Vinh Hong, Henryk Palus, and Dietrich Paulus. Edge preserving filters on color images. In *ICCS2004 – International Conference on Computational Science*, volume 4, pages 35–42, Berlin, Germany, 6 2004. Academic Computer Centre CYFRONET AGH, Springer Verlag.
3. Konstantinos N. Plataniotis and Anastasios N. Venetsanopoulos. *Color Image Processing and Applications*. Springer Verlag, 2000. INF 2003/3326.