LINEAR OPENINGS IN ARBITRARY ORIENTATION IN O(1) PER PIXEL

V. Morard, P. Dokladal, E. Decencière

MINES ParisTech,
CMM - Centre de Morphologie Mathématique, Mathématique et Systèmes,
35, rue St. Honoré, 77305-Fontainebleau-Cedex, France

ABSTRACT

Openings constitute one of the fundamental operators in mathematical morphology. They can be applied to a wide range of applications, including noise reduction and feature extraction and enhancement. In this paper, we introduce a new, efficient and adaptable algorithm to compute one dimensional openings along discrete lines, in arbitrary orientation. The complexity of this algorithm is linear with respect to the number of pixels of the image. More interestingly, the average complexity per pixel is constant, with respect to the size of the opening.

Index Terms— Algorithms, Mathematical Morphology, Openings, Feature Extraction, Filtering

1. INTRODUCTION

Openings and more generally, mathematical morphology (MM), are concepts introduced by Serra and Matheron in the late sixties ([1] and [2]). Any increasing, anti-extensive and idempotent operator is an opening. It can be applied to many applications for filtering and features enhancement. Computing openings with segments filters out the noise and unwanted structures. It can also be used to extract texture descriptors, the local orientation or the size distributions. This is a powerful operator, however, a naive implementation has a considerable complexity of $O(\lambda)$ per pixel. We show that it can be done in $O(1)$, independently of λ, the size of the linear segment.

Over the last years, some authors have focused their work on the speed and they have greatly improved the complexity of these operators. Van Herk proposed a one-dimensional (1D) algorithm to deal with erosions and dilations with no more than three comparisons per pixel [3]. This is a constant time algorithm with respect to λ, which has been studied and improved by many authors, including Soille et al, in [4]. Later, Van Droogenbroeck and Buckley [5], designed a very fast algorithm for erosions and openings, based on anchors. They managed to reduce the number of redundant comparison by introducing the notion of anchors (pixels which are not affected by an operator). Vincent, in [6], has worked on granulometries and opening trees and Menotti et al, in [7], have constructed the 1D component tree in linear time, with respect to the number of pixels of the image.

Here, we start from Dokladal's ([8]) and Soille's work to define a novel algorithm to build 1D openings at arbitrary orientations. The section 2 will introduce the basic notions whereas sections 3 and 4 will respectively describe this algorithm, the timings and the comparison with others efficient methods.

2. BASIC NOTIONS

MM is based on the notion of structuring elements (SE). Hereafter, we will used a linear SE (having a form of a line), of size λ and oriented in the angle α. With this SE, written B^α_λ, we define two basic operators for a grey level image f, the erosion and the dilation.

$$\epsilon^\alpha_\lambda(f)(x) = \wedge\{f(x+y), y \in B^\alpha_\lambda\} \tag{1}$$

$$\delta^\alpha_\lambda(f)(x) = \vee\{f(x-y), y \in B^\alpha_\lambda\} \tag{2}$$

If we apply an erosion followed by a dilation on f, we get a filter called an opening.

$$\gamma^\alpha_\lambda(f) = \delta^\alpha_\lambda(\epsilon^\alpha_\lambda(f)) \tag{3}$$

Further operators can be built with these linear openings. The first one is $\vee\gamma_\lambda(f)$. It is computed by taking the supremum of the openings by segments in all orientations.

$$\vee\gamma_\lambda(f) = \bigvee_{\alpha\in[0,180[} \gamma^\alpha_\lambda(f) \tag{4}$$

The second one, $\zeta_\lambda(f)$, can extract the local orientation of the structures.

$$\zeta_\lambda(f) = \operatorname*{argsup}_{\alpha\in[0,180[} \gamma^\alpha_\lambda(f) \tag{5}$$

Finally, the size distributions (granulometries) can also be deduced from a family of openings of increasing size.

3. OPENINGS IN O(1) PER PIXEL

Openings in constant time with respect to λ, are based on adapted data structures. We use a stack, which is a lifo container ("last in, first out") associated with an object, called hereafter, a *flat zone* (FZ). A FZ is composed of 3 attributes: its starting position ($StartPos$), its grey level value (F) and a flag ($Passed$). Having the information of the starting and the reading position (rp), we are able to simply compute its current length. Note, however, if the flag $Passed$ is raised, its current length is greater than λ, no matter the value of $StartPos$ and rp.

$$\textbf{if}(Passed)\begin{cases}\textbf{then } Length \geq \lambda \\ \textbf{else } Length = rp - StartPos.\end{cases} \quad (6)$$

This flag will be set to $true$ if and only if this FZ is the only one in the stack and if we ensure it has got a length greater than λ. We introduce this important flag to remember the state of the previous popped FZ (see line 8).

For the stack consistency, we have to introduce a key feature as follows:

Proposition 1. *A stack of flat zones will always be ordered by increasing order with respect to F.*

This proposition introduces the necessary hierarchy between flat zones and this is ensured by the code (lines 1, 4, 5). Algorithm 1 presents the pseudo code which will be executed for each line of an image and for each pixel of this line.

3.1. Description of the 1D algorithm

Let $Stack$ be our stack of flat zones with the following functions: $push()$, $pop()$, $top()$ and $empty()$. Therefore, reading an attribute of a flat zone will be written $Stack.top().x$ with x referring to F, $StartPos$ or $Passed$. Inserting a new flat zone into the stack will be written: $Stack.push(F, StartPos, Passed)$ while removing a FZ: $fz = MyStack.pop()$. The reading position rp will be set to 0 at the beginning of each line and its value will be incremented by one when the function $Process\ a\ pixel()$ ends. F is the grey level of the current pixel at position rp.

When we process a new pixel, 3 cases exist:

- The current pixel is a part of the last flat zone pushed: ($F = Stack.top().F$). We do not do anything since this pixel does not bring new information.

- The current pixel has a higher value, line 1. We store this new flat zone by pushing it, into the stack (line 2).

- The third case is more complex since the proposition 1 should always be applied. $F < Stack.top().F$ means that this is the end of, at least, one FZ. Line 5 suppresses from the stack, the topmost flat zone. We compute its current length and we compare it to the threshold value λ (line 6). By construction, if the popped flat zone is larger than λ, this guarantees that all the FZ within the stack, will also be larger than λ. Hence, we necessarily know their final output values and we write them with the function $WriteFlatZones()$ (A FZ ends when begin the next one). To avoid writing twice the value of a pixel, the stack is emptied while we write each FZ in the output image. We leave this function just after we have pushed the current flat zone into the stack, with the flag $Passed$, set to $true$ (line 8).

When the length of the popped FZ (fz) is smaller than λ, we have to check whether the stack is empty. If so, we push the current flat zone with the correct starting position and we leave this function (lines 11). Otherwise, we have to compare the current pixel to the new top most FZ. Again, the same three cases could appear: we do not change anything if $F = Stack.top().F$, we create a new flat zone if $F > Stack.top().F$ and we iterate from line 4 to 14 while $F < Stack.top().F$.

At the end of the 1D signal, some flat zones can still be stored within the stack. Hence, we pop these flat zones and, depending of their length, we write their output values.

Algorithm 1 $Process\ a\ pixel(F, rp, Stack, Out)$

```
1:  if Stack.empty() or F > Stack.top().F then
2:     Stack.push(F, rp, false)
3:  else
4:     while F < Stack.top().F do
5:        fz = Stack.pop()
6:        if fz.Passed or rp − fz.StartPos ≥ λ then
7:           WriteFlatZones(F, rp, Stack, Out, fz)
8:           Stack.push(F, rp, true)
9:           break
10:       else if Stack.empty() or F > Stack.top().F then
11:          Stack.push(F, fz.StartPos, false)
12:          break
13:       end if
14:    end while
15: end if
```

3.2. Extraction of discrete lines in arbitrary angle

The function described above, takes one pixel as input. This algorithm is clearly independent of the orientation of the line. Soille and al, in [4], described a way to go through all pixels of an image at a given orientation. They used Bresenham's lines to construct the best 1D signal at orientation α. More interestingly, the logic behind the construction of lines ensures that each pixel will be processed only once. Thus, this is very important since it allows to run in place. Few minor modifications have to be applied in the algorithm 1: we introduce a line buffer to store the index position of the pixels as soon as they are processed. Hence, we could easily write the result of the filter in the output image with no other extra computation.

Figure 1 presents the results of the operators γ_λ^α, $\vee\gamma_\lambda$ and ζ_λ on a fingerprint image.

(a) Input (b) γ_λ^{70} (c) $\vee\gamma_\lambda$ (d) ζ_λ

Fig. 1. Results of three operators with $\lambda = 21$ pixels. From left to right, the initial image, an oriented filtering (70 degrees), the enhancement of linear structures and the local orientation extraction.

3.3. Complexity of the 1D algorithm

Let n be the number of pixels of the image and m be the number of grey levels. We consider a rectangular image, with W and H its width and height, respectively and a linear horizontal opening. Considering a complexity analysis, we have to analyse the two main loops of algorithm 1. The first one is a loop where the function $Process\ a\ pixel()$ will be executed for each pixel of the image (n times). This function uses a stack, where we insert a new flat zone when the reading pixel is bigger than the topmost FZ. Therefore, the maximal number of flat zones that can be inserted into the stack will be $min(W, m)$. This bound will be reached with a strictly monotonic signal.

The second inner loop ($while$) is controlled by this stack. The function $WriteFlatZones()$ is called a variable number of time, but we ensure that each pixel will be written exactly once. In the worst case, algorithm 1 also ensures that each pixel will be pushed and popped once and this is strictly independent of the size of the opening.

The complexity is varying from one pixel to another. However, in average and in the worst case, we only have few comparisons, a push, a pop, and a write on the output image, per pixel. All these operations are computed in $O(1)$. Hence, this algorithm have a constant complexity with respect to λ.

We note, however, that the number of comparisons and the number of pixels inserted into the stack depend on the image's content. A constant signal will reach a theoretical lower bound with only one comparison per pixel (See section 4.1).

Therefore, the overall complexity of this algorithm for an image is in average, $O(n)$.

4. TIMINGS AND COMPARISON

4.1. Benchmark on the image content

By analysing algorithm 1, we notice that the timings will change with the image content. A constant image will define the theoretical lower bound of this algorithm, with only one comparison per pixel. Figure 2 collects the timings for a horizontal opening with respect to λ, for different images of size 512×512 pixels: goldhill, a binary version of goldhill, a random and a constant picture. As expected, the constant image defines the lower bound of this algorithm. We note that a picture with uniform noise is computed quicker than goldhill's image. A general rule for this algorithm is that, the timings are correlated to the mean number of pixel into the stack. A random signal will have, in average, fewer pixels in the stack than a natural image.

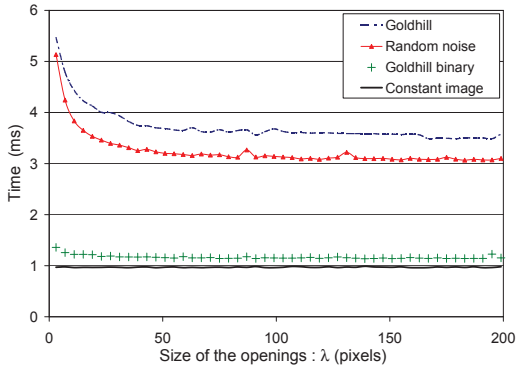

Fig. 2. Timings for horizontal openings of size λ for different images (512×512 pixels)

The fewer flat zones and the closer you get to the theoretical lower bound.

4.2. Benchmark on existing methods

For the sake of comparison, we have implemented 3 other algorithms. The first one has been published by Soille, Breen and Jones in [4]. It has also a complexity of $O(1)$ per pixel, building erosions and dilations with no more than 3 comparisons per pixel. It will be referenced as Soille's algorithm hereafter. The second method is the openings by anchors of Van Droogenbroeck and Buckley, [3], available in [9]. The third is the naive implementation of a linear opening. We note that these algorithms are written in C++ using pointer arithmetic. They have been integrated to the same platform, with exactly the same interface. All the optimization flags are set to true during the compilation. Therefore, no bias has been introduced between these methods.

We will study exclusively horizontal openings, as, to our knowledge, there is no existing implementation of Van Droogenbroeck's algorithm for openings at arbitrary angles. Furthermore, we will only use 8 bits images as Van Droogenbroeck's algorithm is based on a histogram which is built for this kind of pictures. For every λ, 100 independents realizations have been computed and averaged for each method. The timings are shown in figure 3. The image used is goldhill, having 512×512 pixels with 8 bits. We note that the results are approximately the same with other images.

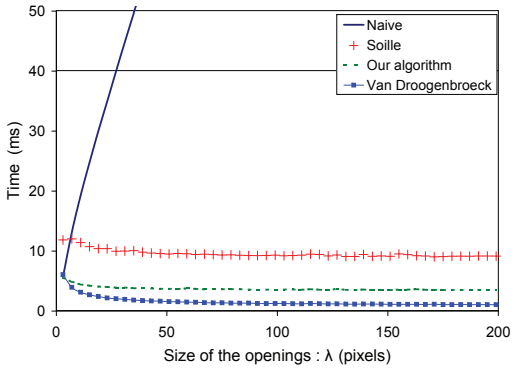

Fig. 3. Timings for an horizontal opening with regard to λ for different algorithm.

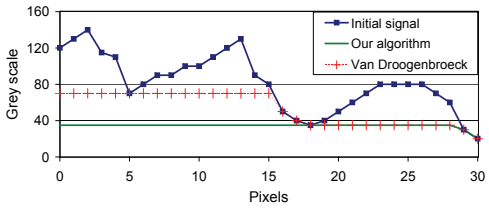

Fig. 4. Opening of size $\lambda = 21$ pixels. Van Droogenbroeck's algorithm is not correct at the beginning of the line. With an opening of size 21 pixels, the output value should be the smallest values of the first 21 pixels.

The difference between the naive implementation and others methods is huge. The naive implementation's complexity is independent on the image content but it does depend on the radius of the openings: $O(\lambda \times n)$.

Soille's algorithm is not as fast as the 2 other methods, and Van Droogenbroeck's algorithm outperforms our algorithm especially for large values of λ. Two reasons can be pointed out to explain this difference:

- For our algorithm, every pixel of the output image is written exactly once. This can slow down our algorithm compared to an algorithm that only writes the modified pixels.
- Van Droogenbroeck's algorithm considers the infinite signal domain. In practice, on bounded supports, the result is incorrect around the borders, as shown in figure 4.

5. CONCLUSION

This paper introduces a novel algorithm to build one dimensional openings with a constant complexity, with respect to λ. It can be applied on floating-point data without additional computational time, as we only perform comparison operations with no histogram. We have compared it to other methods such as the naive implementation, Soille's and Van Droogenbroeck's algorithm. Our method has very good execution times and we correctly handle the borders. Moreover, with the construction of openings at arbitrary angles, we can efficiently compute powerful operators such as the supremum of openings or the local orientation.

For future work, we will do some small modifications of this algorithm to compute granulometries in linear time with respect to the number of pixels. This is an improvement compared with Vincent's algorithm presented in [6] (Vincent defined a quasi linear algorithm). Again, with little change on this very adaptable algorithm, we can also extract the component tree in linear time. Finally, to increase the fexibility of this operator, it can be extended to path operators.

6. REFERENCES

[1] G. Matheron, *Random sets and integral geometry*, Wiley series in probability and mathematical statistics. Wiley, New York,, 1974, Bibliography: p. 254-256.

[2] J. Serra, "Image analysis and mathematical morphology," *Academic, London*, vol. 1, 1982.

[3] M. Van Herk, "A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels," *Pattern Recognition Letters*, vol. 13, no. 7, pp. 517–521, 1992.

[4] P. Soille, E.J. Breen, and R. Jones, "Recursive implementation of erosions and dilations along discrete lines at arbitrary angles," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, no. 5, pp. 562–567, 1996.

[5] M. Van Droogenbroeck and MJ Buckley, "Morphological erosions and openings: fast algorithms based on anchors," *Journal of Mathematical Imaging and Vision*, vol. 22, no. 2, pp. 121–142, 2005.

[6] L. Vincent, "Granulometries and opening trees," *Mathematical morphology*, p. 57, 2000.

[7] D. Menotti, L. Najman, and A. de Albuquerque Araújo, "1d component tree in linear time and space and its application to gray-level image multithresholding," in *8th International Symposium on Mathematical Morphology (ISMM)*, 2007, pp. 437–448.

[8] P. Dokládal and E. Dokládalová, "Grey-scale morphology with spatially-variant rectangles in linear time," in *Advanced Concepts for Intelligent Vision Systems*. Springer, 2008, pp. 674–685.

[9] R. Dardenne and M. Van Droogenbroeck, "libmorpho, http://www.ulg.ac.be/telecom/research.html," .