Morphology on color images

Alexandre ABRAHAM

Technical Report n^o0835, Décembre 2008
revision 1965

This work takes place in the context of Milena, the C++ generic image processing library of the Olena platform, developed at LRDE. Morphological algorithms are one of Milena's major assets. Yet few image processing libraries implement them even though they are very useful. Those algorithms require supremum and infimum operators, which don't exist by default for composite types like red-green-blue (RGB) pixels. We therefore propose the implementation of those operators for RGB values, along with a complete toolchain allowing morphological algorithms to work on color images.

Les algorithmes morphologiques sont l'un des atouts majeurs de Milena, bibliothèque de traitement d'images générique et performante développée au LRDE. En effet, ils sont très utiles et relativement peu implémentés dans les autres bibliothèques. Ces algorithmes requièrent des opérateurs de bornes supérieure et inférieure (*supremum* et *infimum*) qui n'existent pas par défaut pour des types composites comme les couleurs encodées en rouge-vert-bleu (RVB). Nous présentons donc une implémentation de ces opérateurs pour le type RVB ainsi que toute la chaîne de traitement permettant de faire fonctionner des algorithmes morphologiques sur des images en couleurs.

Keywords
Color, Morphology, Multivaluate Ordering, Vector Median Filter, Autarkical Leveling

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
abraham@lrde.epita.fr – http://www.lrde.epita.fr/

Copying this document

Contents

Introduction

Morphological image processing is an important field of image processing domain but is not much implemented in generic image processing libraries. Its purpose is to offer operators to work on shapes and not only on pixels. This domain is widely explored at LRDE and is a strength of the Olena library.

One of the main drawback of these algorithms is that they were first conceived for binary images. Some basic requirements make them difficult to apply on complex data. Milena claims to be generic so it must provide to the user the mechanisms to apply morphology on any type of data.

Barnett (1976) indicate how to compare multivariate data like vectors (and so colors). (M. C. D'Onellas, 1998), a reference paper, put in common different ways to apply morphology on color images.

From this perspective, we first introduce basic definitions of morphological image processing and then expose the main color ordering operators. Then we present the integration in the Milena's generic environment. And then we finally conclude with some applications and algorithms integrated in Milena.

Thanks

Thanks to Thierry Géraud and Guillaume Lazzara for their supervision. Thanks to all 2009 CSI for their contribution to the laboratory in term of content and in term of good mood. Special thanks to reviewer Florian Quèze.

Chapter 1

Morphological Image Processing

1.1 Definition

Mathematical Morphology is a lattice theory. It is mainly used for image processing but can be applied on many structures like graphs, meshes... Its name comes from the Greek word *morphê* that means "shape" Wiktionary (2003). In fact, morphological image processing focuses on shape processing (size, shape, convexity, connexity...).

As an example, here is a list of basic morphological operators:

- **erosion**: erodes shapes by taking for each pixel the lowest pixel in its neighborhood
- **dilatation**: increases shape size by take the maximum of the neighborhood
- **opening**: erases little shapes by eroding and then dilating the image
- **closing**: rounds shapes by dilating image and then eroding it.

Some trickier filters can be made by composing from these operators. More elaborated operations can also be done like the watershed exposed in my previous report (Abraham, 2008).

The main shortcoming of mathematical morphology is that it has been conceived for binary images. Since most operators only use comparison and extremum operators, it has easily been extended to grayscale. However, application on complete lattices like colors is still theoretical.

1.2 Introducing colors

As said before, color morphology is still theoretical. Like all color image processing, there are many approaches. One can treat a color image in order to make it fancier or to preprocess it for

another algorithm.

There are many color spaces used for many purposes:

- **Red, Green, Blue (RGB)**: that corresponds to the human eye cone cell but is paradoxically very far form human perception
- **Y'UV**: composed of one luma (Y) and two chrominances (UV). It commonly used for data transmissions because transmission errors and compression artifacts are masked to human perception
- **Hue, Saturation, Value (HSV)**: that is the closest to human perception.

Obviously, Milena must be able to deal with all these color types easily and the user must be able to do what he wants safely. Since color space knowledge is not required for this report, a reader interested in this subject should read a previous report entire made on this subject (Vigouroux, 2008).

1.3 Applying morphology on color images

In order to apply morphology on an image, it is necessary to have an ordering on the value (knowing a maximum value is an advantage but it is not required to apply such filters). This seems obvious for values like booleans or integers but that's another story for multivariate data like colors. For example, ordering red and blue is not easy for human mind, these are two different things like oranges and bananas. Several ways to compare are presented but more advanced methods can be found (M. C. D'Onellas, 1998).

1.3.1 Lexicographical ordering

This total ordering is inspired by the dictionary ordering, it's the most intuitive: it simply consists in comparing each component one by one. The first components are compared: if they are different, we have the result, if they are equal, we compare the next component and so on. The problem of this method is obvious: more importance is given to the first component. For example, considering a RGB value, the red color is advantaged whereas this color is not more important than others.

A way to get rid of this inconvenient is to make a linear combination of the three components.

1.3.2 Scalar valued functions

This partial ordering is based on the comparison of a linear combination of the three components. This permits to give to each component a coefficient relative to its importance. For

example, human eyes are more sensitive to the modification of the green component than red or blue one. Thus some coefficients are commonly used to reflect this behavior (equation 1.1).

$$a \preceq b \Leftrightarrow \begin{cases} v_a = 0.3 \times a_{red} + 0.6 \times a_{green} + 0.1 \times a_{blue} \\ v_b = 0.3 \times b_{red} + 0.6 \times b_{green} + 0.1 \times b_{blue} \\ v_a \leq v_b \end{cases} \tag{1.1}$$

Many other ways to compare values exist like the bit-mix that is based on the binary representation of the value. Even if it can seem awkward because it does not care of the values but of their hardware representation.

But it is legitimate to ask if color ordering in a global referential is really needed since algorithms may only need the distance between the colors.

1.3.3 Distance computation

Later in this paper (3.1) is explained a new approach of the problem which gets rid of the data ordering. In fact, it is not really necessary if one just wants a local view on the values. One can consider the distance between the color of the considered point and its neighbors. The choice of the distance computation becomes crucial and can lead to different results.

More than classical norms like Euclidean or Manhattan, some norms produce interesting results like the Mahalanobis one that takes into account the whole bunch of data and not only each component one by one.

1.3.4 Minimum or infimum?

Another question that must be asked is: when using a minimum operator, should I take a minimum that exists in the image or should I build a value based on all the values (i.e. taking the minimum of each component for example: $inf((2, 12, 7), (15, 1, 5)) = (2, 1, 5)$. The resulting color may not exist in the original image).

The choice of this behavior is done according to the purpose. If the filter is done to make the image prettier (with smooth shaded tones), then creating new colors is not a problem. If it is a preprocessing for another filter, then one should consider to keep only the image values.

Chapter 2

Integration in Milena

2.1 Morphology in Milena

Looking deeper into the algorithms, one can see that they all follow the same canvas. This canvas has been of course implemented in Milena, making implementation of these operations very easy (listing 1).

Now to define a basic operator like erosion, simply declare an operator which neutral is the supremum of your image and accumulator is the minimum. Pass it to our canvas: you are now applying an erosion. Plus, thanks to all Milena's genericity features, this filter can be applied of images of any type, any dimension and any data type.

2.2 Function Images

The purpose here is to take a RGB image and to create three "views" of this image, each view accessing directly a component (red, green or blue). This can be done by accessing, for example, the red member of each RGB pixel. This can be done easily thanks to a morpher.

But, thinking about genericity, one can want to see his image through the function he wants (cosinus, norm, ...). This function can be a simple function, it can be bijective so that a read / write access on the image is possible or even more complicated.

```
I morphological_algorithm (Image<I> input,
                           Window<W> output,
                           Operator<O> op)
{
  // Output image
  I output;

  // Accumulator that take value of neighbours and
  // output the result of an operation
  A accu = op.accu(ima);

  // Fill the border with a neutral value not to
  // have to test each time if we are on a border
  border::fill(input, op.neutral());

  // Psite iterator
  mln_piter(I) p(input.domain);

  // Psite neighbor iterator
  mln_qiter(W) q(win, p);

  for_all(p)
  {
    accu.init();
    for_all(q) if (input.has(q))
      accu.take(input(q));
    output(p) = accu.to_result();
  }

  return output;
}
```

Listing 1: Morphological algorithm canvas in Milena. This code is very simplified.

Thus we define three main types of functions:

- **v2v function**: a one way function (modulus, ...)
- **v2w2v function**: or bijective function, a function that has an inverse (cos/arccos, ...)
- **v2w_w2v function**: or two-way function, a function that can be inversed knowing the previous state of the value (norm, ...).

The last point can seem pretty obscure but can be explained easily thanks to an example. Let us take an image of vectors. One can easily see this image through a norm function using Euclidean norm for example. If one wants to set the norm of a vector v to 1, it can be done easily by making $v = v/norm(v)$ but the knowledge of the value of the vector is needed, not only its norm. This behavior is very powerful since it becomes a one-line call to normalize all the vectors of an image: `level::fill(fun_image(input, norm), 1);`.

Accessing to the red component of a RGB pixel and changing its value is a v2w_w2v function. In fact, accessing the value is easy but changing the value requires to know the value of the other components. Going further, one can imagine that, for some purpose, the user wants to use his own RGB value type.

Some functions like `red` may be declared as meta-functions since their behaviour depends on the type of their arguments. It is implemented in Milena as template specialisations. For example, if one wants to write his own `red` function, he has just to write a specialisation for his data type (listing 2).

```
1  struct custom_rgb
2  {
3    typedef int value;
4    value red;
5    value green;
6    value blue;
7  };
8
9  struct function< meta::red< custom_rgb >
10 : public Function_v2w_w2v<function< meta::red < custom_rgb > > >
11 {
12   typedef typename custom_rgb::value result;
13   result read(const custom_rgb& c)
14   { return c.red; }
15
16   typedef typename custom_rgb::value& lresult;
17   lresult write(custom_rgb& c)
18   { return c.red; }
19 };
```

Listing 2: Example of custom red function

It is now easy to apply a morphological operator on each color. Unfortunately, most color image users may want to use all the components at the same time in order to take advantage of these data. More operations are needed to make it work in Milena.

2.3 Mixin values

Adding a comparison operator to a data type must be simple, easy, local and type safe. The best way to add a member function to a data type is a cast hack (listing 3). As we can see, a "violent cast" (cast by going through `void*`) is performed to add the member function. Because this behavior is not very safe, a security has been set so that one can only cast a type to another type if the two structures have the same size.

The violent cast used here can be extended for some other usage. For example, to save an image with Milena, this image must be RGB because most of the formats impose it. Considering that a RGB value is just a vector of 3 values, one can want to save a HSV image in order to avoid conversion between color spaces. This should be possible thanks to the `violent_cast_image`.

Thanks to that system, any operator can be added to any type of value. For example, eroding the red component of a RGB image just requires the addition of a "less than" comparison operator and of a "max" (which is the neutral value of the erosion operator).

```cpp
#include <iostream>

struct not_comparable
{
  int value;
};

struct make_comparable
{
  bool less(not_comparable &lhs,not_comparable &rhs)
  { return lhs.value < rhs.value; }
};

template <typename T, typename O>
struct mixin: T
{ };

template <typename T, typename O>
bool operator<(mixin<T, O> &lhs, mixin<T, O> &rhs)
{
  static O op;
  return op.less(lhs, rhs);
}

template <typename T, typename A>
T& violent_cast(A& a)
{
  return *(T*)(void*)(&a);
}

int main ()
{
  not_comparable a, b;
  a.value = 3;
  b.value = 5;

  typedef mixin<not_comparable, make_comparable> comp;

  std::cout << (violent_cast<comp>(a) < violent_cast<comp>(b));
}
```

Listing 3: Example of comparison operator addition

```
struct red_only
{
  value::rgb8 max () const
  {
    return value::rgb8(255, 0, 0);
  }

  template <unsigned n>
  bool less(const value::rgb<n>& a, const value::rgb<n>& b)
  {
    return a.red() < b.red();
  }
};
```

Listing 4: Operators needed for red erosion

Chapter 3

Algorithms, implementations and applications

3.1 A new approach

Ordering colors totally is a pain and does not always give great results. On the other hand, a human mind is perfectly able to compare colors and to score the difference: everyone would say that orange is closer to red than blue. Starting from this assumption, the Euclidean norm can be used to measure difference between colors treated as 3D points.

Plus, we want to find more accurate values, not just the min or the max in the neighborhood. For this purpose, when considering a point p, instead of only considering the relation with its neighbours, one will consider the relation between each neighbour and all the other points in the neighborhood. That leads us to create a new canvas (listing 5).

Thanks to this new approach, more complex algorithms can be developed.

3.2 Vector Median Leveling

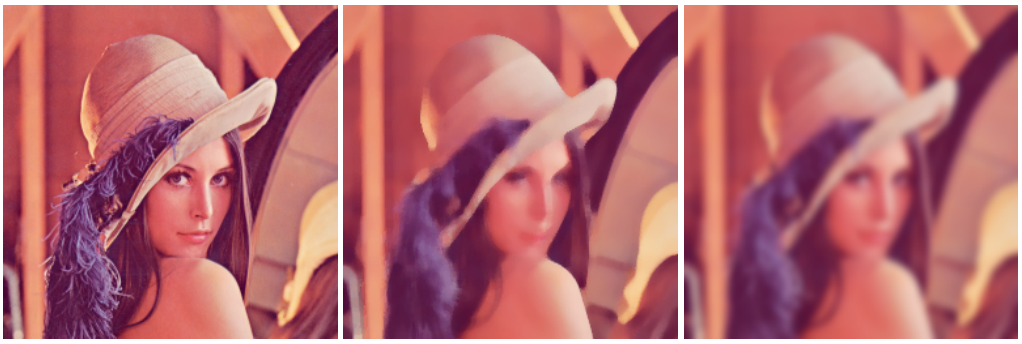
VML is a denoising filter. The main advantage of this filter is that it removes noise while keeping a maximum information on the image edges. The algorithm is pretty simple: each point p of the image is replaced by the point that minimizes the sum of the distance between him and all other points (which is a kind of median).

$$\sum_{i \in N} ||\vec{v_{vm}} - \vec{v_i}|| \leq \sum_{i,j \in N} ||\vec{v_j} - \vec{v_i}||$$

```
I morphological_algorithm (Image<I> input,
                           Window<W> output,
                           Operator<O> op)
{
  // Output image
  I output;

  // Accumulator that take value of neighbours and
  // output the result of an operation
  A accu = op.accu(ima);
  A naccu = op.accu_neighb(ima);

  // Psite iterator
  mln_piter(I) p(input.domain);

  // Psite neighbor iterators
  mln_qiter(W) q(win, p);
  mln_qiter(W) r(win, p);

  for_all(p)
  {
    accu.init();
    for_all(q) if (input.domain().has(q))
    {
      naccu.init(q);
      for_all(r) if (input.domain().has(r))
        naccu.take(input(r));
      accu.take(naccu.to_result());
    }
    output(p) = accu.to_result();
  }

  return output;
}
```

Listing 5: New canvas base on differences between values.

This can be done using the previous canvas with an Euclidean norm accumulator as neighbor accumulator and a minimum accumulator as classical accumulator. One can see that this filter conserves edges far better than the Gaussian blur (figure 3.1). It is mainly used as a preprocessing filter.



(a) Original image (b) Vector Median Filter (c) Gaussian Blur

Figure 3.1: Comparative of denoising filters. Size of window is 9x9.

3.3 Autarkical leveling

3.3.1 Leveling

The leveling is a morphological filter which purpose is to "cut crests" and "fill gaps". Basically, it is just reducing all values of an image that is greater than a maximum value to this maximum value, creating "flat levels" (figure 3.2).

This basic leveling is not very interesting. Let us introduce a marker function g. The rule is simple: if $g \leq f$ (resp. $g \geq f$), the leveling must create a flat zone that is as low (resp. high) as possible without passing under (resp. above) the g function. An example is shown to improve comprehension (figure 3.3).

3.3.2 The algorithm

That is not obvious in the description above (because the 1D image considered is continuous) but, during a leveling, some values that are not present in the image can be created (1.3.4). Given that this behavior may not be wanted, some algorithms have been made that do not create any value but use the ones present in the image. These algorithms are called "autarkical". There are many autarkical leveling, but the one implemented is from (Gomila and Meyer, 1999).

In order to make the leveling, a marker function is needed. Here this is the Vector Median Filter that will be used. The leveling of the image I thanks to the marker M is done as follow:

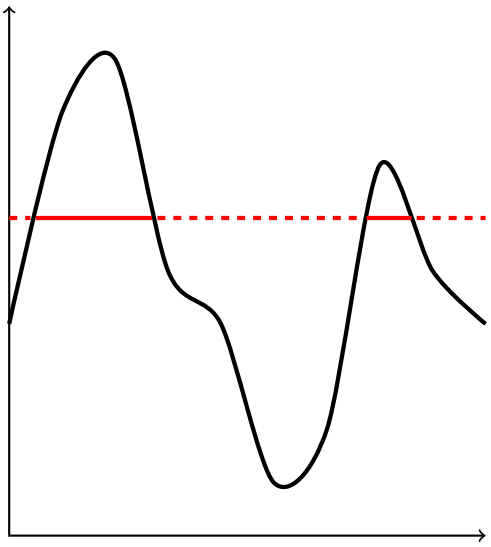


Figure 3.2: An example of leveling, the crests are cut.

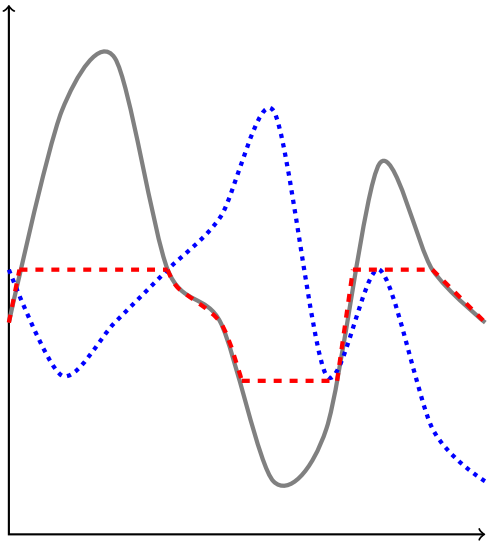


Figure 3.3: Result of the leveling according to the dotted marker function is dashed.

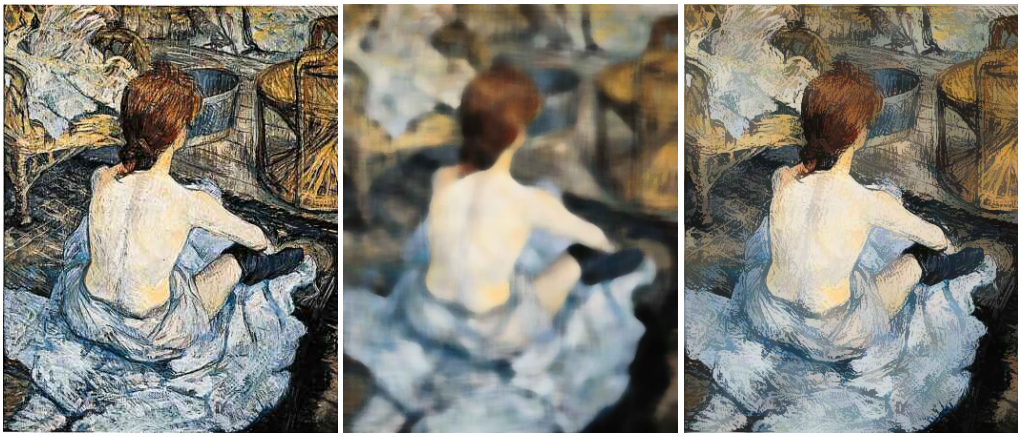
for each point $p \in I$, if all the neighbors of p in M are *on the same side* of p, then $I(p)$ is the marker vector the *closest* to p, otherwise nothing happens. The process is repeated until stability.

In order to make it more readable, $I(p)$ is noted I_p. Three points $I_p, M_q, M_{q'}$ (each component of the color are taken as coordinates) are *on the same side* if the angle $M_q\widehat{I_p}M_{q'}$ is acute. That is to say if I_p does not belong to the sphere of diameter $[M_qM_{q'}]$. Classically, the closest vector to I_p is the vector M_q that minimizes the Euclidean distance from I_p such as $M_q = min_{q \in N_p} ||\vec{I_p} - \vec{M_q}||$.

In the example, one can see that edges are preserved while tint areas are wider (figures 3.4 and 3.5).

(a) Original image (b) Vector Median Filter (c) Autarkical Leveling

Figure 3.4: Après le bain, femme s'essuyant la nuque, Degas, 1898. RMN ©



(a) Original image (b) Vector Median Filter (c) Autarkical Leveling

Figure 3.5: The toilette, Toulouse-Lautrec, 1896. Musée d'Orsay © Paris

Conclusion

After a description of morphology and of the color theory applied in mathematical morphology, one knows that there are many ways to compare colors in function of the needs. Whatever solution that one chose, it is possible to implement it in Milena thanks to the tools described here. This respects the idea of genericity of the library. As an example, some algorithms have been developed and show that Milena can handle such data.

Some work still needs to be done though. Some algorithms presented here (like the vector median filter) have a too big complexity ($O(nw^2)$ with n the size of the image and w the size of the window) and should be optimized using preprocessing. Some color spaces are also missing and should be integrated thanks to previous works. More algorithms should also be implemented in order to be able to fulfil any user requirement.

Chapter 4

Bibliography

Abraham, A. (2008). Topological watershed.

Barnett, V. (1976). The ordering of multivariate data. *Journal of Royal Statistical Society A, 3:318-355*.

Gomila, C. and Meyer, F. (1999). Levelings in vector spaces. *Institute for Signal and Information Processing*.

M. C. D'Onellas, R. Van den Boomgard, J.-M. G. (1998). Morphological algorithms for color images based on a generic-programming approach. *SIBGRAPI*.

Marcin Iwanowski, J. S. (1999). Morphological interpolation and color images. *International Conference on Image Analysis and Processing*.

Serra, J. (2003). Introduction aux représentations d'images couleur et à leur traitement.

Vigouroux, C. (2008). Color types in milena.

Weiss, B. (2006). Fast median and bilateral filtering. *Special Interest Group in GRAPHics*.

Wiktionary (2003). morphology. Website. http://en.wiktionary.org/wiki/morphology.