GENERIC IMPLEMENTATION OF MORPHOLOGICAL IMAGE OPERATORS

When Harry's C++ Code Met Sally's Algorithms

JÉRÔME DARBON*, THIERRY GÉRAUD and
ALEXANDRE DURET-LUTZ
EPITA Research and Development Laboratory
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
E-mail: {jerome.darbon, olena}@lrde.epita.fr
Phone +33 1 53 14 59 16 – Fax +33 1 53 14 59 22

Abstract.
Several libraries dedicated to mathematical morphology exist. But they lack genericity, that is to say, the ability for operators to accept input of different natures —2D binary images, graphs enclosing floating values, etc. We describe solutions which are integrated in OLENA, a library providing morphological operators. We demonstrate with some examples that translating mathematical formulas and algorithms into source code is made easy and safe with OLENA. Moreover, experimental results show that no extra costs at run-time are induced.

Key words: mathematical morphology, image processing operators, genericity, programming.

1. Introduction

Most people involved in mathematical morphology are mathematicians or image processing practitioners rather than computer scientists. Therefore, they should find it easy to use program libraries in order to avoid dealing with implementation problems and rather focus only on the methodological aspects of their work. Köthe [7] notes that the lack of algorithmic comparison in the literature is due to the difficulty of implementing computer vision algorithms. Furthermore, Mallat [10] insists on the notion of reproducible computational science; that is to say, an author of article should make source code available. Along these lines, Pitas has recently published a book [12] fully illustrated with source code.

Quite a lot of image processing libraries are available on the Internet; however, they are usually restricted to very few image structures and data types, whereas mathematical morphology applies on a wide range of data: signals, 2D and 3D images, graphs, etc., containing integers with different precisions,

* Also with: *École Nationale Supérieure des Télécommunications, Networks and Computer Science Department. 46, rue Barrault – F-75634 Paris Cedex 13 – France.*

floats, binaries, and so on. It is then very difficult for practitioners to find a library with morphological operators that meet their requirements.

When an image processing algorithm —for instance an erosion— is translated into a *single* routine in a given computer language, one says that this routine is *generic* if it accepts different input types. D'Ornellas and van den Boomgaard [4, 3] mention that generic algorithms for morphological image operators could be developed in C++ using the *generic programming* paradigm.

The aim of this paper is twofold: it presents a new library which provides mathematical morphology operators, and it shows how one can easily translate a mathematic formula into a C++ program in the context of our library.

This paper is organized as follows. In section 2, we present different paradigms to program image processing operators; we discuss their advantages and drawbacks with regard to their genericity level and their safety for the user. Then, in section 3, we study the particular cases of several morphological operators. Last, we conclude in section 4 and we give an evaluation of their performance.

2. Programming Paradigms, Types, and Safety

2.1. State of the Art

Most of image libraries are built on a C-style programming paradigm, and two families can be identified.

The first family considers that a general type, usually `float`, is enough to store data[1]. A 2D image structure is then defined as depicted below (left column). A major drawback of this approach is that there are no semantical distinctions between images with respect to their data type: procedures are therefore unable to check constraints about input images. For instance, the procedure `foo` (below, right column) expects that both input images have the same data type but cannot check it. It is then very easy for a programmer to call routines incorrectly.

```
struct image2d
{
  unsigned nrows, ncols;
  float** data;
};
```

```
void foo(image2d* ima1, image2d* ima2)
{
  /* ... */
}
```

Two other important drawbacks are that non-scalar data cannot be handled by this image structure and that images consume memory unnecessarily.

The second family of library programming style, described by Dobie and Lewis [2], addresses these problems by enforcing type control. To this end, the 2D image structure is modified, see the left column below: a new field, `type`, allows the programmer to insert assertions to check at run-time the proper nature of the input images (right column below).

[1] The general type `float` is the most convenient one for general purpose library; In mathematical morphology, a general type would rather be `short`, even if PDE-based approaches are now in fashion.

```
typedef enum { INTU8, FLOAT } data_t;

struct image2d
{
   unsigned nrows, ncols;
   data_t type;
   void** data;
};
```

```
void foo(image2d* ima1, image2d* ima2)
{
   assert(ima1->type == ima2->type);
   switch(ima1->type) {
      case INTU8:
         /* call sub-routine for INTU8 */
         foo_INTU8(ima1, ima2); break;
      /* ... */
   }
}
```

Unfortunately, if safety is enforced for the library user, the library programmer has to write as many sub-routines as there are data types. For instance, the sub-routine `foo_INTU8` contains the code dedicated to unsigned 8 bit integers. Since writing many similar routines per algorithm is long and tedious, libraries usually handle only very few data types.

We have shown that genericity wrt data types can be handled either by the use of a general type (`float`) or in a tedious fashion by code replication (the different cases of `switch`). However, image structures are still not generic, since we can only handle 2D images. Some C libraries use macros (keyword `#define`) to emulate C++ templates. However, macros cannot handle all features that we enjoy with templates (e.g. stronger typing, recursivity, meta-programming).

2.2. C++ and Genericity

An interesting feature of the C++ language [13] is *genericity* using the `template` keyword. In the left column sample code below, image2d is a meta-structure parameterized by an unknown data type `T` and the procedure foo is similarly parameterized. Its input must share the same data type as made explicit by the procedure signature: this constraint is now checked *at compile-time*.

```
template<class T>
struct image2d
{
   unsigned nrows, ncols;
   T** data;
};

template<class T>
void foo(image2d<T>& ima1,
         image2d<T>& ima2)
{
   //...
}
```

```
int main()
{
   image2d<float> ima1, ima2;
   //...
   foo(ima1, ima2); // first call

   image2d<int> ima3, ima4;
   //...
   foo(ima3, ima4); // second call
}
```

In the right column above, the client instantiates different kinds of image (`ima1` contains floats whereas `ima3` contains integers) and calls `foo` twice. At each call, the compiler automatically deduces the type T from the input types and creates a specialized version of the meta-procedure `foo`. In our example, `T` is set to `float` at the first call and a version of foo dedicated to process 2D floats images is generated. That means that the compiler creates specific sub-routines, and therefore saves the programmer from performing this tedious task (see section 2.1).

2.3. Full Genericity and STL Style

Now we turn to *full genericity* as we want a procedure to accept different image types. The key idea is given by the style of the Standard Template Library (STL for short) now part of the C++ Standard Library [13]: procedures should be parameterized by their input types. This paradigm is called *generic programming* by the object-oriented scientific computing community [11].

For instance in order to browse the contents of images with different structures, obviously one cannot keep two loops when input is 2D or three when it is 3D. These image structure implementation details must be hidden. The solution that early appears in imaging software [9] is the use of *iterator* objects. Consider the code below. A new procedure, `bar`, sets every pixel to 0. It is now parameterized by `InputIter` which represents an iterator type. The object `p` iterates from the first point of the targeted image to the last, these iteration boundaries being given by the methods `begin()` and `end()` of the image class. When `bar` is called, the particular procedure instantiated by the compiler uses an iterator `i` of type `image2d_iterator<float>`; the single loop is thus able browse image with different structures.

```
template<class T>
struct image2d
{
   typedef image2d_iterator<T> iterator;
   unsigned nrows, ncols;
   T** data;
   iterator begin();
   iterator end();
   //...
};
```

```
template<class InputIter>
void bar(InputIter _first, InputIter _last)
{
   for (InputIter i = _first; i != _last; ++i)
      *i = 0;
}

int main()
{
   image2d<float> ima;
   //...
   foo(ima.begin(), ima.end());
}
```

Finally, the procedure `bar` can accept various kind of input. It is fully generic, type-safe and, moreover, as fast as dedicated C. Indeed, the use of parameterization (templates) along with type deductions (typedefs) is handled by the compiler, that is, statically. In a classical object-oriented way of programming, lot of work is performed at run-time (e.g, method dispatch through a hierarchy) and the resulting programs are not as efficient as the one presented in this section. As far as we know, the only other image processing library based on this programming paradigm is VIGRA by Köthe [8].

Please note that the contents of sections 2.1 to 2.3 is discussed in more detail in [5], [4], [3], [6] and [8].

2.4. Image Processing Style

We find the previous proposal unsatisfactory: programs should be closer to algorithm descriptions in mathematical language rather than in a computer scientist language. Our proposal is not to design a new language dedicated to image processing such as in [1] but to provide *tools* that make easier programming for practitioners: we want something like:

```
template<class _I>
void bar(image<_I>& _ima)
{
  Exact_ref(I, ima);
  Iter(I) p(ima);
  for_all(p) ima[p] = 0;
}
```

```
int main()
{
  image2D<float> ima;
  //...
  bar(ima);
}
```

And that's indeed what we have in our library.

Please note that, for some mathematical morphology algorithms, there are few approaches to design them (parallel, in-place, based on priority queue and so on). Although we cannot provide a single version for all these flavors, we are still able to preserve the other levels of genericity.

3. Case Studies of Mathematical Morphology Operators

In this section, we study several morphological operators and we show that mathematical formulas and algorithms can easily be written using our tools[2]. In particular, the watershed operator is described as suggested by d'Ornellas and van den Boomgaard [4].

TABLE I
Some Simple Morphological Operators.

Operator	Formula	Code
dilation	$\forall x,\ [\delta_B(f)](x) = \max_{b \in B} f(x+b)$	for_all(x) df[x] = max(f, x, B);
closing	$\phi_B(f) = \varepsilon_{\check{B}}[\delta_B(f)]$	erosion(dilation(f, B), -B);
black top-hat	$BTH_B(f) = \phi_B(f) - f$	minus(closing(f, B), f);
TH contrast op.	$\kappa^{TH} = Id + WTH_B - BTH_B$	plus(f, minus(white_top_hat(f, B), black_top_hat(f, B)));

3.1. Simple Operators

Table I presents how four morphological operators are translated in C++ code.

Dilation. In our library, the body of the `dilation` procedure (left column below) performs the following operations. Line 2 first defines the output image, `fd`, whose type is the procedure parameter `I`. To this aim, `fd` needs some structural information from the input image `f` (for instance, its size). At line 3, an iterator `x` is declared whose type is deduced from `I`. Then, the iteration is performed. To remain close to the mathematical formula, a particular function, `max`, is specialized according to the nature (type) of the structuring element. That is, whether `B` is flat or not, calling `max` does not run the same code.

[2] In future versions, C++ operator overloading capabilities will be used in order to get a more natural way of expressing formulas. For instance, `minus(closing(f,B), f)` will be replaced by `closing(f,B) - f`.

```
1  border::adapt_copy(f, B.delta());
2  I  fd(f.info());
3  Iter(I)  x(f.info());
4  for_all(x)
5      fd[x] = max(f, x, B);
6  return fd;
```

```
7  Iter(SE)  b(B);
8  b = begin;
9  Value(f)  val = f[x + b];
10 for_all_remaining(b)
11     if (val < f[x + b])
12         val = f[x + b];
13 return val;
```

The body of procedure `max` when `B` is a flat structuring element is presented in the right column above. Note that in order to keep this procedure generic, all implementation details are hidden. An important feature of our library is that we do not have to care too much about accessing data out of image support. For instance, in the case of 2D images, `x+b` (e.g., line 11) may fall outside the image support if `x` is near the image boundary. In order to save the programmer from writing extra code to test if `x+b` is valid, some image type, such as 2D image, have an outside border and we can assign values to these particular points. In the case of dilation, we call `border::adapt_copy` (line 1) which first adapts the border size of the image to that of the structuring element and then copies values of the image inner boundary to the border[3]. Finally, dilating induces no side effect. Image processing routines relying on masks, windows, neighborhoods or structuring elements, are simpler to implement.

Closing. In our library, we do not want to annoy the user with memory management of the data structures such as images or graphs. In the case of the closing operator, a temporary image is first created resulting from a dilation process, and then, erosion is applied to obtain the final result. In classical libraries, the programmer should delete the temporary image to recover its memory. There is a risk of forgetting this deletion and to get some memory leaks at run-time. Another immediate consequence is that operators cannot be chained such as in:

```
erosion(dilation(f, B), -B)
```

because no variable holds the temporary image which thus is responsible for a memory loss.

For user convenience, we have implemented a transparent memory management: when a data structure is no longer referenced, it is automatically destroyed. Combining operators is thus made as simple as possible.

Top-Hat Contrast Operator. A lot of morphological operators rely on arithmetics, and usually, image processing libraries use *built-in* types —that is, types provided by the programming language— to express the nature of data. A resulting well-known problem is value overflow. In the sample line below positive values are encoded on 8 bit, ranging from 0 to 255:

```
unsigned char i = 255, j = 1, k = (i + j) / 2;
```

[3] In the case of the input image being a graph, since the notion of border does not exist, calling `border::adapt_copy` is still valid but does not execute any code. Another approach is to set the border to $-\infty$ ($+\infty$) in the case of dilation (erosion).

we finally have `k` set to 0. In the case of non-flat structuring elements and in the case of morphological operators involving arithmetics, e.g., the top-hat contrast operators, the programmer should not deal with from these problems. To this end, we have defined our own data types and the corresponding safe arithmetics. For instance, in the line above, one should use `int_u8` instead of `unsigned char` and the compiler raises en error at *at compile-time* because the type of expression `(i + j) / 2` is now `int_u9` and because the assignment from this type towards the "smaller" type `int_u8` is forbidden.

We also have equipped our library with conversion routines that can be used either as stand-alone functions or as first argument of other routines. In the sample code below, the user does not need to know the data type of the image returned by the contrast operator; she can guarantee that, at the very end, every pixel value falls between 0 and 255. So, she benefits simultaneously from safe arithmetics *and* convenient data type manipulations.

Last, the notion of *scoping*, whose correctness is verified by the compiler, ensures the programmer that she uses as little memory as possible; at the end of the `main` scope, only `lena` lies in memory.

```
// file inclusions:
#include "basics2d.hh"
#include "io/pnm.hh"
#include "convert/bound.hh"
#include "morpho/top_hat.hh"

// usage declarations:
using namespace oln;
using convert::bound;
using morpho::top_hat_contrast_op;
```

```
int main ()
{
  image2d<int_u8> lena;
  { // sub-scope
    read_pnm(lena, "lena.pgm");
    image2d<int_u8> output
      = top_hat_contrast_op(bound<int_u8>(),
                            lena,
                            square(5));
    save_pnm(output, "output.pgm");
  }
  // here, only lena is in the scope
}
```

3.2. Watershed

Mathematical morphology operators, such as the watershed, are often more complicated than those already discussed. In [4], d'Ornellas and van den Boomgaard argue that a *generic* implementation of the watershed is possible based on a wave-front propagation; they give the algorithm canvas that we recall in the left column below. Our library provides a *generic* implementation of this algorithm, that is, a function which works on various structures (n-D images, graphs, etc.) containing data of various types[4]. The corresponding code excerpts from our library are given in the right column below.

[4] The queue-based priority algorithm presented here is of course optimal for discrete data types. However, the user can alsa call it when data are floating values.

Initialization Step: PQ q; **For** (every p in domain D){ **If** (M(p) != 0 and (exist ($p\prime$) in $N_G(p)$: $M(p\prime) == 0$)) q.enq(p, f(p));} **Data Driven Propagation Step:** **While**(!(q.empty())){ q.deq(); **For**(every($p\prime$) in ($N_G(p) \cap D$)) **If** ($M(p\prime) == 0$) **If** (exists ($p\prime\prime$) in $N_G(p\prime)$: $M(p\prime\prime)$!= $M(p)$) $M(p\prime)$ = WSHED; **else** { $M(p\prime) = M(p)$; q.enq($p\prime, f(p\prime)$) } }	*// Initialization Step:* PQ q; Iter(I2) p(M); for_all(p) **if** (M[p] != 0 && exist_eq(M, p, Ng, 0)) q.push(queue_elt(p, f[p])); *// Data Driven Propagation Step:* **while** (!q.empty()) { Point(I1) p = q.top().first; q.pop(); Neighb(N) p_prime(Ng, p); for_all_neigh (p_prime) **if** (M.hold(p_prime)) **if** (M[p_prime] == 0) **if** (exists_neq(M, p_prime, Ng, M[p])) M[p_prime] = WSHED **else** { M[p_prime] = M[p]; q.push(queue_elt(p_prime, f[p_prime])); } }

Note that we also succeeded in providing tools making "sophisticated" morphological algorithms implementation easy.

4. Conclusion

In this article, we have shown that computer programs can achieve both *genericity* and *user-friendliness.* Based on the conclusions of d'Ornellas and van den Boomgaard [4], we have proposed solutions —some of them being described here— and we have built an appropriate framework of object-oriented tools: OLENA. OLENA is a library dedicated to image processing practitioners, and in particular, to mathematical morphology users. The sources of OLENA are freely available on the Internet at the address:

`http://www.lrde.epita.fr/olena`

Last, getting all the benefits described in this article has not compromised efficiency. Table II gives processing time for some morphological operators. These algorithms were tested on the classical gray-level 256×256 image LENA with a 1 Ghz personal computer running GNU/LINUX; the code was compiled using the GNU C++ compiler with all optimizations enabled. The column "regular" refers to operators being implemented in a classical way; equivalent "fast" versions of morphological operators, based on [14], are also available in OLENA. We are aware of the optimal 11×11 dilation which uses a decomposition of two dilations by straight lines. The algorithm is $O(1)$ with line dilation approaches such as van Herck's [15]. But we do not use it because this is only a text-book study to evaluate and compare execution times. Finally, table III presents major functionalities in OLENA.

TABLE II
Performance Evaluation (time in seconds).

Algorithms	Structural Elt.	Regular	Fast
Dilation	3×3 Square	0,04	0.05
Dilation	11×11 Square	0,46	0,12
Dilation	Disk of radius 6	0,44	0,13
Closing	Disk of radius 6	0,85	0,31
Top-Hat Contrast Op.	Disk of radius 6	1,74	0,62
Watershed	4-Connectivity	0,17	—

TABLE III
Functionalities.

Dilation / Erosion
Closing / Opening
White Top Hat / Black Top Hat / Top Hat Contrast Operator
Hit-or-Miss
Hit-or-Miss Opening Foreground / Background
Hit-or-Miss Closing Foreground / Background
Beucher / Internal / External Gradient
Geodesic Dilation / Erosion
Geodesic Reconstruction by Dilation / Erosion (simple, sequential, hybrid)
Minima Imposition
Regional Minima
Watershed
Minima / Maxima Killer

References

1. R. Cecchini and A. Del Bimbo. A programming environement for imaging applications. *Pattern Recognition Letters*, 14:817–824, October 1993.
2. M. Dobie and P. Lewis. Data structures for image processing in C. *Pattern Recognition Letters*, 12(8):457–466, 1991.
3. M. C. d'Ornellas. *Algorithmic Patterns for Morphological Image Processing.* PhD thesis, University of Amsterdam, 2001.
4. M. C. d'Ornellas and R. van den Boomgaard. Generic algorithms for morphological image operators — a case study using watersheds. In H. Heijmans and J. Roerdink, editors, *Mathematical Morphology and its Applications to Image and Signal Processing*, pages 323–330, 1998.
5. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the computational geometry algorithms library. Technical Report 3407, INRIA, April 1998.
6. T. Géraud, Y. Fabre, A. Duret-Lutz, D. Papadopoulos-Orfanos, and J.-F. Mangin. Obtaining genericity for image processing and pattern recognition algorithms. In *Proceedings of the 15th International Conference on Pattern Recognition*, volume 4, pages 816–819. IEEE Computer Society, September 2000.
7. U. Köthe. Reusable implementations are necessary to characterize and compare vi-

sion algorithms. in DAGM-Workshop on Performance Characteristics and Quality of Computer Vision Algorithms, September 1997.
8. U. Köthe. STL-style generic programming with images. *C++ Report Magazine*, 12(1):24–30, January 2000.
9. D. Lawton and D. Mead. A modular object oriented image understanding environment. In *Proceeding of the 10th International Conference on Pattern Recognition*, volume 2, pages 611–616, Atlantic City, NJ, USA, June 1990.
10. S. Mallat. *A Wavelet Tour of Signal Processing*, chapter 1, pages 17–18. Academic Press, 1999.
11. Scientific computing in object-oriented languages. Web page. http://oonumerics.org/.
12. I. Pitas. *Digital Image Processing Algorithms and Applications*. Wiley, 2000.
13. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.
14. M. Van Droogenbroeck and H. Talbot. Fast computation of morphological operations with arbitrary structuring elements. *Pattern Recognition Letters*, 17(14):1451–1460, 1996.
15. M. Van Herk. A fast algorithm for local minimum and maximum filters on rectangular and octogonal kernels. *Pattern Recognition Letters*, 13:517–521, 1992.