Yayi
A generic framework for morphological image processing
IPOL

Raffi Enficiaud

June 2012

(Ex.) CMM - Mines Paris

Forewords

Yayi - A generic framework for morphological image processing

http ://raffi.enficiaud.free.fr

Recent library (1st release on August 2009)
A few developers (me : Raffi Enficiaud + sometimes : Thomas Retornaz)

Licence

Released under the Boost licence (very permissive)

Why genericity is interesting ?

Multidimensional & multispectral aspects

Images domain (spanned "dimension")

- $2D$ images
- $3D$ images
- $3D + t$ images
- any dimension [a]

---

a. in which "pixel" has a sense

Pixel values

- Binary
- Scalar (integer, floats, multiprecision, ...)
- Color
- Multispectral

Functions and processings

Problem n^o1 : high functional redundancy

Example : adding a constant value "v" on all the points of the image.

$$\mathcal{F} : \forall p \in \mathbf{E}, \mathcal{J}(p) = \mathcal{I}(p) + v$$

E dimension	$\mathcal{I}$ type	v type		$\mathcal{J}$ type
2D	scalar	scalar	.....	scalar
	integer 8bits	integer 8bits		integer 16bits

Number of combinations

$$(\text{nb dimensions} \times \text{nb types})^2 \times \text{nb types}$$
$$(2 \times 4)^2 \times 4 = 256$$

Processings

Problem $n^o 2$: algorithmic redundancy

Labelling

With the same data :

- create an output image with an "id" for each cc.
- create an output image with an "id" for each cc. that is not "background"
- output the adjacency graph
- label each cc. with some measurement function (area, volume, mean, ...)

all rely on the connected component extraction

Without meta-programming

1. development of a specific function for each need
2. development of an OO architecture that may be slow at runtime

Processings

Problem n^o2 : algorithmic redundancy

Labelling

With the same data :

- create an output image with an "id" for each cc.
- create an output image with an "id" for each cc. that is not "background"
- output the adjacency graph
- label each cc. with some measurement function (area, volume, mean, ...)

all rely on the connected component extraction

With meta-programming

1. write once the cc. algorithm
2. write several delegate template methods (one for each need)
3. pass these delegates to the cc. algorithm

Genericity by meta-programming approach

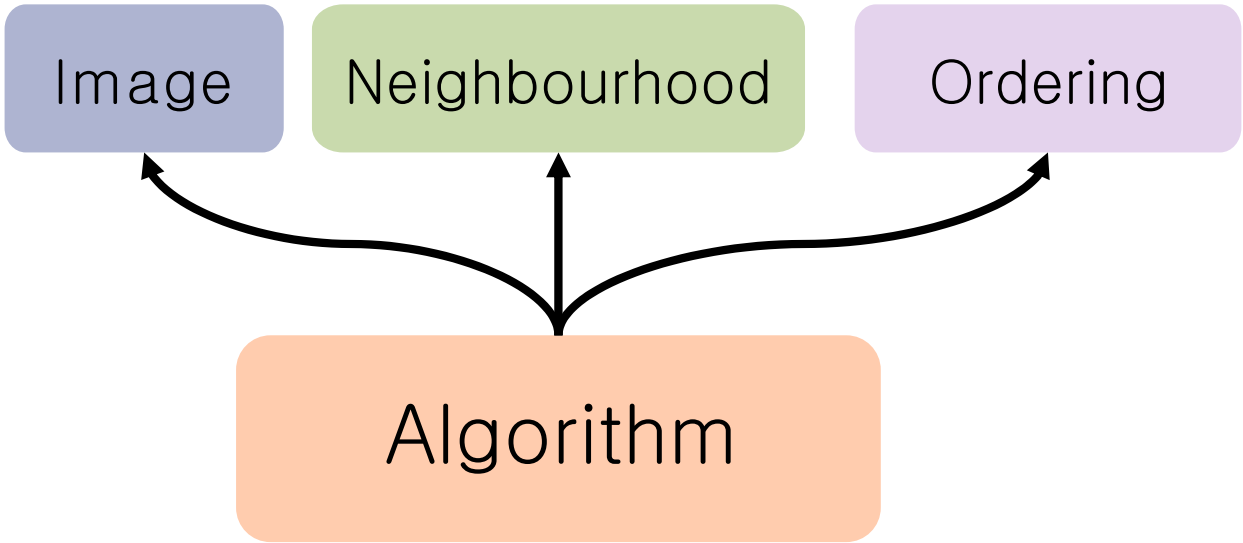
1. Focusing the efforts on the implementation of the algorithms
2. Capitalisation
3. High and efficient code reuse
4. It is "easy" to make the types abstract (and to port - a first version of - existing algorithms)

Meta-programming ?

1. Types resolution
2. Specialising

Algorithms & Images

In order to have generic morphological algorithms, the following structures should be defined :



1. Image : generic image structure
2. Neighbourhood : generic way to encode the topology
3. Order : generic way to encode the lattice algebraic properties

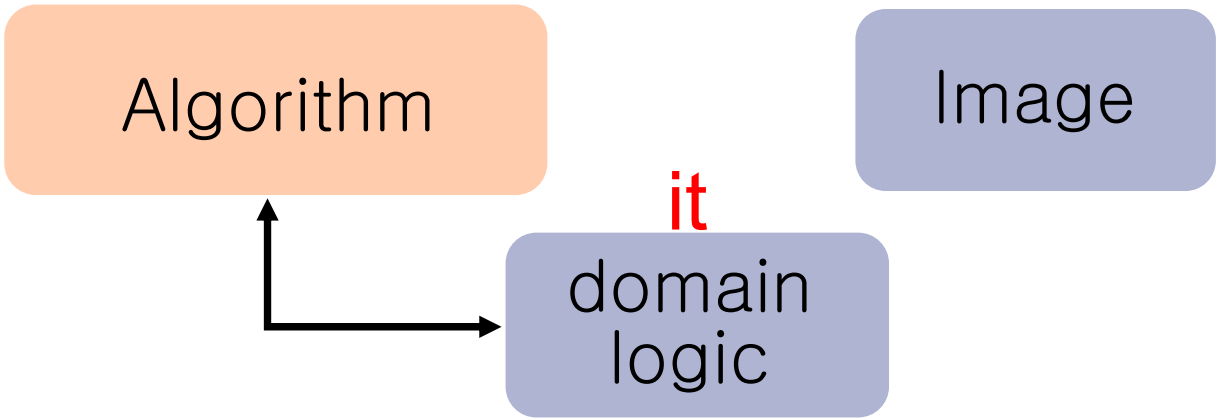
How to discover the image domain ?

Multidimensional aspects

Write « $\forall p \in \mathbf{E} \ldots$ »

Iterator approach

Query an domain iteration object *it* to the image
While *it* has not reached the end of the domain {
    Process point *p* returned by *it*
}



Domain processing through iteration

Multidimensional aspects

Iterators

Universal method for discrete domain (sequence of points)

Pros

1. The structures act as "containers" and provide an object allowing to scan their domain
2. Algorithms become independent of
   - the intrinsic coordinate system of the images ($2D$, $3D$, $4D$, ...).
   - the geometry of their domain (size, borders type, windows, masks,...)

Cons

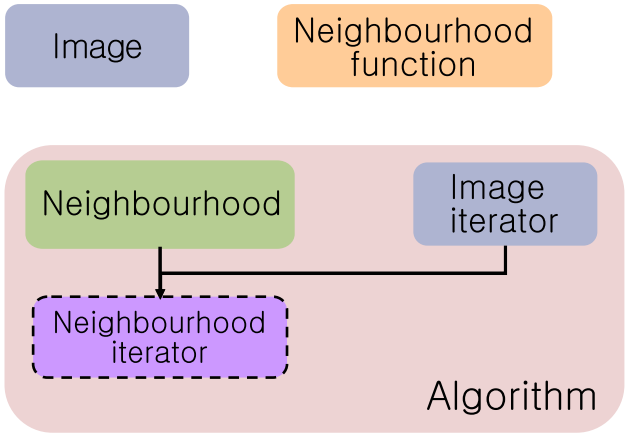
1. Less efficient than pure "C" or specific approaches
2. "Discrete" domain (points can be sequenced)

Neighbourhoods

Structuring functions

Use case

1. Neighbourhood initialization (image, neighbouring function)
2. Centring of the neighbourhood
3. Iteration over the neighbour elements
4. Loop back to 2 until the end of the domain



Pros of iterator approach

1. The topology is managed inside the type of the neighbourhood
2. The algorithms are independent from the type of the neighbourhood

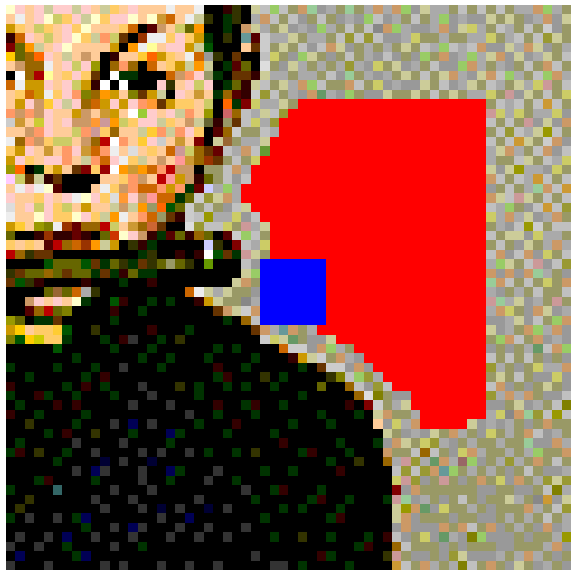
Neighbourhoods

Structuring function

Image topology management delegated to a structuring function

Morphological Amoebas

Romain Lerallut, Etienne Decencière & Fernand Meyer.
Image filtering using Morphological Amoebas.
Proceedings of the 7th ISMM, 2005.



Yayi overview

Yayi

Main design objectives

- Specialized for Mathematical Morphology
- Open source under (very) permissive Boost licence
- Reference algorithms
- Highly generic, "easy" to use with different type of usage
- Addresses in a generic manner the slowness problem of generic approaches

Yayi

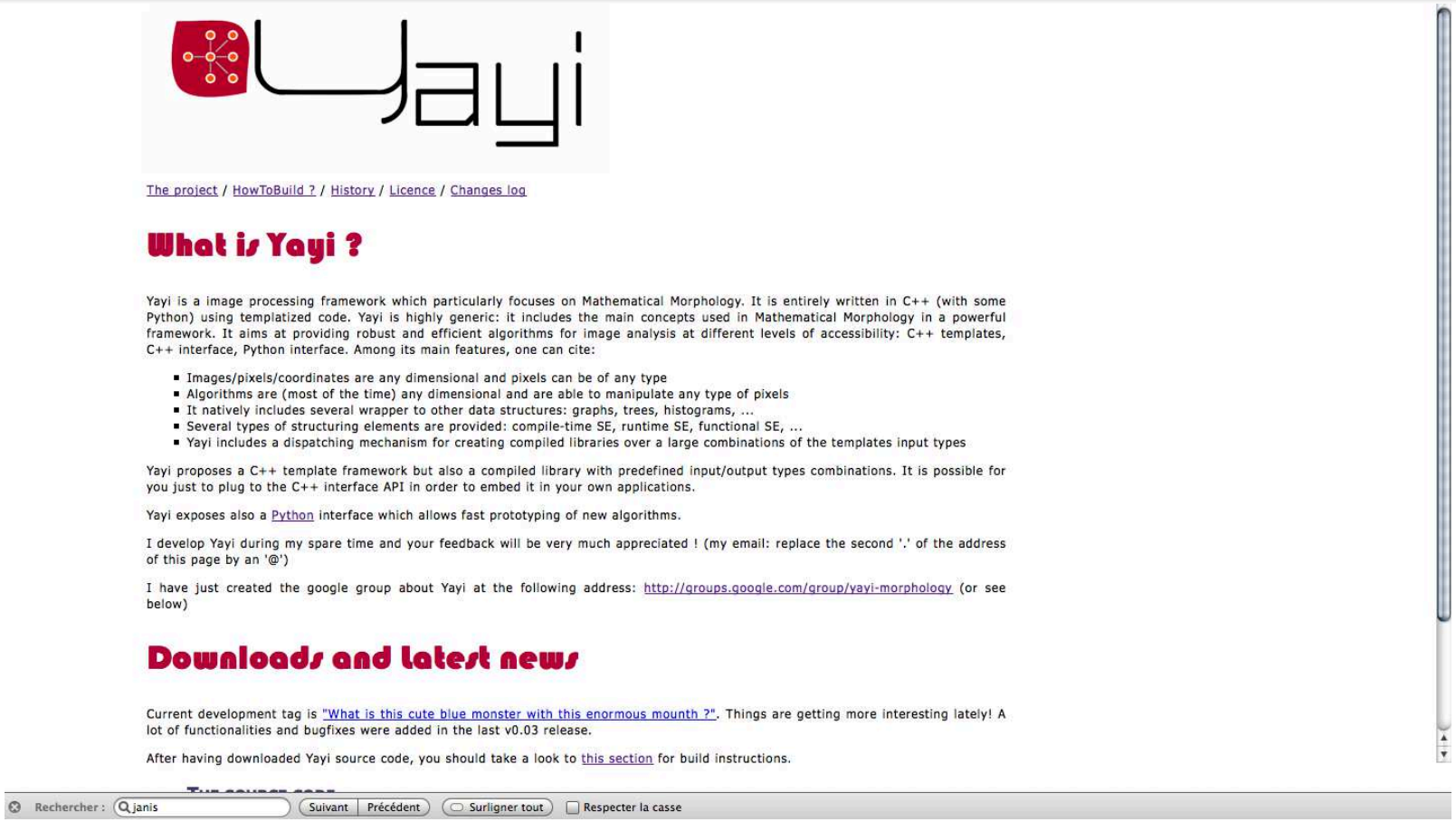
Design & licence

1. Open source, Boost licence
2. C++ / Python : cross-platform source code, standard "compliant" (tested platforms : Ubuntu, MacOSX, Win, x86/x64)
3. Few dependencies (Boost, Jpeg, PNG, HDF5 (optional)), all under permissive licence
4. Several "modules"
5. No patented code
6. Generic and easy to extend [1]

1. once the doc is available

Web site

http ://raffi.enficiaud.free.fr



"Google" group for discussions, news and distribution

No online SVN repository (code is released on archives)

Constituting modules of the library

- YayiCommon : structures communes à toutes les autres librairies (types, variants, graphs, main interfaces, colours, coordinate, errors management...)
- YayiImageCore : Image and iterators interfaces and implementation, image factory and utilities. Pixels transformation processors.
- YayiIO : images input/output function (PNG, JPG, RAW, HDF5, Tiff (next release or so))
- YayiPixelProcessing : pixel level functions (arithmetic, logical, colour)
- YayiStructuringElements : structuring elements and neighbourhood classes, predefined SE
- YayiLowLevelMorphology : neighbourhood processors and basic morphological functions
- YayiLabel : labelling algorithms
- YayiMeasurements : measurements on images
- YayiReconstruction : morphological reconstruction algorithms
- YayiNeighborhoodProcessing : local transformations
- YayiDistances : distance transform algorithms
- YayiSegmentation : segmentation algorithms

Constituting layers

Interface layer

Interfacing Yayi with minimal "intrusion"

Aim

Manipulating the objects without caring about the exact type.

Pros

Suitable for algorithmic developments
Fast compilation
Small overhead for switching on the appropriate template instance

Cons

Slower when executing algorithms working pixel level (variant transformations)

Python layer - example

Using Boost.Python.

Example of use

```
1  import YayiCommonPython as YACOM
2  import YayiImageCorePython as YACORE
3  import YayiIOPython as YAIO
4  import YayiStructuringElementPython as YASE
5  import YayiLowLevelMorphologyPython as YALLM
6
7  c3_f = YACOM.type(YACOM.c_3, YACOM.s_float)
8  sc_f = YACOM.type(YACOM.c_scalar, YACOM.s_float)
9
10 im = YAIO.readJPG(os.path.join(path_data, "release-grosse bouche.jpg"))
11
12 im_hls = YACORE.GetSameImageOf(im, c3_f)
13 YAPIX.RGB_to_HLS_l1(im, im_hls)
14
15 im_grey = YACORE.GetSameImageOf(im, sc_f)
16 YAPIX.CopyOneChannel(im_hls, 2, im_grey)
17
18 im_grey2 = YACORE.GetSameImage(im_grey)
19 YALLM.Dilation(im_grey, YASE.SESquare2D(), im_grey2)
```

Template layer

Where algorithms are designed

Aim

Generic implementation of algorithms

Pros

- Suitable for algorithmic developments, at every level (pixel, neighbourhood, etc.)
- No overhead, easy to use (includes)

Cons

- Intrusive for the external client
- Slow to compile
- Errors hard to understand :)

Template layer

Main Image Processing components

What do we have ?

1. template structures (graphs, priority queues, histograms, variants, pixels, coordinates, images, SE...)
2. pixel wise image processors
3. neighbourhood image processors

These things are not new...

Template layer

Example of use

```
1  #include <Yayi/core/yayiImageCore/include/yayiImageCore_Impl.hpp>
2  #include <Yayi/core/yayiImageCore/include/yayiImageUtilities_T.hpp>
3  #include <Yayi/core/yayiPixelProcessing/include/image_copy_T.hpp>
4
5  // simple constant generator
6  struct dummy_generator {
7    int operator()() const {
8      return 0;
9  }};
10
11 // ...
12
13 // 3D unsigned char image type
14 typedef Image<yaUINT8, s_coordinate<3> > image_type3D;
15
16 // instance
17 image_type3D im3D;
18
19 // settings \& allocation
20 im3D.SetSize(c3D(10, 15, 20));
21 im3D.AllocateImage();
22
23 // "block" iterators
24 for(image_type3D::iterator it(im3D.begin_block()), ite(im3D.end_block());
25       it != ite;
26      ++it) {
27   *it = 0;
28 }
29
30 // equivalent to
31 std::generate(im3D.begin_block(), im3D.end_block(), dummy_generator());
```

Template layer

Example of use (continued)

Using “window” iterators

```
1  // "windowed" iterators
2  s_hyper_rectangle<3> h(c3D(2,2,1), c3D(5,5,1));
3
4  image_type3D::window_iterator it(im3D.begin_window(h)), ite(im3D.end_window(h));
5  std::generate(it, ite, dummy_generator());
6
7  // 3D float image type
8  Image<yaF_simple, s_coordinate<3> > im3Dtemp;
9
10 // copy properties
11 im3Dtemp.set_same(im3D);
12
13 // copy content (and cast)
14 copy_image_t(im3D, im3Dtemp);
```

Template layer

Summary

Pros

- "Header only"[a] : no particular library to link, code directly generated inside the target library/binary
- Types resolved automatically by the compiler
- Only needed functions/structures are generated
- Very simple to use

a. almost

Cons

- Compilation time increases
- Generated binary size increases
- Sometimes violates licences (not for Yayi of course)

Pixelwise operators

Simple example of multiplication with a constant

Definition of a pixel functor

Operation at pixel level

```
1  template <class in_t, class val_t, class out_t>
2  struct s_mult_constant :
3    std::unary_function< typename boost::call_traits<in_t>::param_type, out_t>
4  {
5    // the constant value stored in the functor
6    typename boost::add_const<val_t>::type value;
7
8    s_mult_constant(typename boost::call_traits<val_t>::param_type p)
9      : value(p)
10   {}
11
12   out_t operator()(typename boost::call_traits<in_t>::param_type v1) const throw()
13   {
14     return static_cast<out_t>(v1 * value);
15   }
16 };
```

Pixelwise operators

Simple example of multiplication with a constant

Definition of the image transform using the previous pixel functor

Operation at image level

```
1  template <class image_in_t, class val_t, class image_out_t>
2  yaRC multiply_images_constant_t(
3    const image_in_t& imin,
4    typename boost::call_traits<val_t>::param_type val,
5    image_out_t& imo) {
6
7    // the type of the functor
8    typedef s_mult_constant<
9      typename image_in_t::pixel_type,
10     val_t,
11     typename image_out_t::pixel_type>  operator_type;
12
13   // the pixel processor instance
14   s_apply_unary_operator op_processor;
15
16   // the functor instance
17   operator_type op(val);
18
19   // the processing
20   return op_processor(imin, imo, op);
21 }
```

Pixelwise operators

What is important here ?

The s_apply_unary_operator (ie. pixel processor) contains all the logics for :

1. extract the appropriate iterators from the images (windowed, non-windowed)
2. call the functor at each pixel
3. in a central way for the library (improving the processor improves everything...)

Things are getting more complicated :

1. when several images are involved (binary, ternary, n-ary pixel functors) : dependant on the geometry and the type of the images
2. when one would like to dispatch the processing onto several threads

Pixelwise operators

Example 1 - binary operators (without return)

Generic processing

```
1  template <class it_strategy /* = iterators_independant_tag*/>
2  struct s_apply_op_range<it_strategy, operator_type_binary_no_return> {
3    template <class op_, class iter1, class iter2, class image1, class image2>
4    yaRC operator()(op_& op, iter1 it1, const iter1 it1e, iter2 it2, const iter2 it2e,
           image1&, image2&) {
5      for(; it1 != it1e && it2 != it2e; ++it1, ++it2) {
6        op(*it1, *it2);
7      }
8      return yaRC_ok;
9    }
10 };
```

Pros

Very generic (any kind of iterators, any kind of image type, any domain, etc).

Cons

Very generic : misses some possible optimizations, genericity assumption covers too many cases

Pixelwise operators

Example 1 - binary operators (without return)

Where the processing power is lost ?

```
1 | for(; it1 != it1e && it2 != it2e; ++it1, ++it2)
```

1. two possibly heavy objects
2. operator++ : $\times 2 \times N$ calls
3. operator!= : $\times 2 \times N$ calls

Remark : iterators are useful for generic domain discovery, in practice, image operands share more common properties.

Possible optimizations - geometry

- images share the same kind of geometry (domain) :

```
1 | for(; it1 != it1e; ++it1)
2 |   op(*it1, im2.pixel(it1.Offset());
```

- ...

Pixelwise operators

Example 2 - binary operators (without return)

Possible optimizations - geometry and operands

- images are the same (add, mult, ...)

```
1 for(; it1 != it1e; ++it1){
2   ref_t p = *it1;
3   op(p, p);
4 }
```

- "block" type iterators applied over the whole image domain (pointer arithmetic instead of complex iterators)
- images share the same kind of geometry (domain) and windows are of the same size (constant shift)

```
1 const offset shift = it2.Offset() - it1.Offset();
2 for(; it1 != it1e; ++it1)
3   op(*it1, im2.pixel(it1.Offset() + shift);
```

A lot of different cases should be determined at runtime !

In practice, not that much optimizations can be performed and covering more runtime configurations would only bloat the code.

Pixelwise operators

Example 3 - binary operators with simple states

Possible optimizations - threads 1

- Simple case 1 : the functor is "read only" (not mutable), "copy constructible" (one local instance per thread) & the iterators are "random access" (eg. add constant, random generator, ...)
- Simple case 2 : the functor is "stateless" & the iterators are "random access" (eg. arithmetic, logics, comparisons, ...)

Possibility to distribute the processing over several threads

Pixelwise operators

Example 4 - binary operators with functor "semantics"

Possible optimizations - threads & functor semantics

Less simple case : given a non-stateless functor f, $\exists$ a function g, such that for a partition $\{X_i\}_i$ of the image's domain we have $f(\cup X_i) = g(\cup f(X_i))$ (eg. histograms, $\int$, ...)

Contents

Algorithms & functions

Currently available

- Pixel processing
  - arithmetic, logical, combinations, comparisons
  - color processing
- Basic morphology (grey scale)
  - erosions, dilations, geodesic erosion/dilation
  - Minkowski addition, subtraction
  - hit-or-miss
  - openings, closings
- Distances
  - morphological distances
  - quasi distance
  - generic exact distance transform
- Labellings
  - connected components with adjacency predicates
  - connected components with measurements (area...)
  - local extrema
  - adjacency graph
  - extraction of the geometry of the cc.

Algorithms & functions

Currently available

- Reconstructions
  - opening, closing by reconstruction
  - levelling
- Local transformations
  - local color transform
  - quantiles, means
- Segmentation
  - isotropic watershed
  - viscous watershed

Algorithms / functions / structures

Roadmap

Priority 0

1. multithreaded pixel-wise operations
2. documentation (a nice one)

Priority 1

1. more colour and pixels transforms
2. morphological skeleton
3. reducing the complexity of neighbourhood operations
4. more “native” SE. (homothetic, line)
5. morphological operations on line SE.
6. common image transforms (interpolations in any dimension, sobel, bilateral, etc)
7. hierarchical segmentations

Algorithms / functions / structures

Roadmap

Priority 2

1. (really) binary images
2. swig & matlab interface
3. installation tool & precompiled libraries

Priority 3

1. lattice structure
2. graph interface to images

Priority 4

1. distributed image structure (for very large data)

Synthesis

Synthesis

Yayi ...

1. open source, free, under a permissive licence
2. performs a lot of generic things you do not want to know
3. features many mathematical morphology functions/methods/algorithms
4. is waiting for your feedback !

Thank you for your attention !

Questions ?