# Yayi
## A generic framework for morphological image processing
## Séminaire LRDE

Raffi Enficiaud
*INRIA / Ex. CMM Mines Paris*

3 mars 2010

Inria / (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 + new recruit : Thomas Retornaz ! ! !)

### Boost licence

Very permissive !

# Genericity

## Genericty benefits in image processing

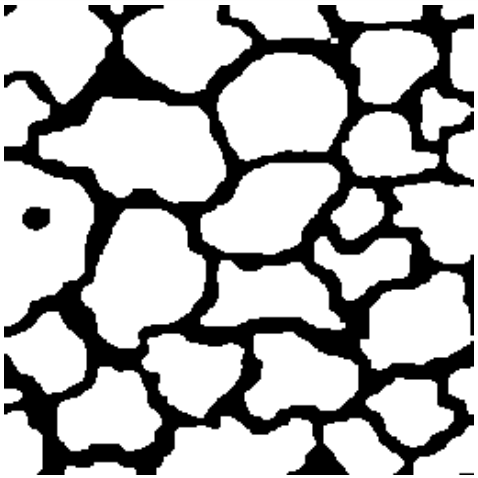
- Why genericity ?
- What kind of problem it addresses
- What it is not that good at
- What it is not able to solve

# Why genericity becomes interesting ?

## Multidimensional aspects

### Images domain ("dimension")

- $2D$ images

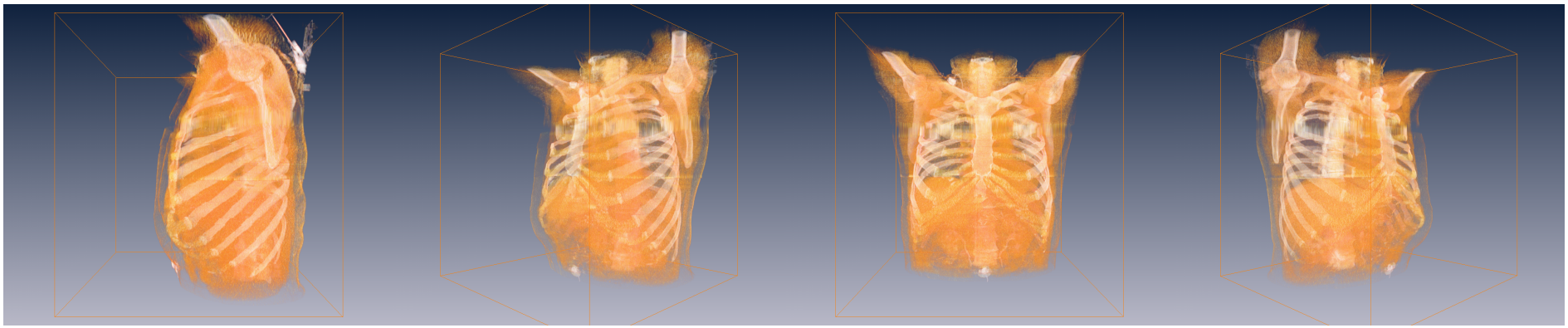


# Why genericity becomes interesting ?

## Multidimensional aspects

### Images domain ("dimension")

- $2D$ images
- $3D$ images



# Why genericity becomes interesting ?

## Multidimensional aspects

### Images domain ("dimension")

- $2D$ images
- $3D$ images

Why restricting the structures to the range of sensors capabilities ?

### Higher dimensional domains

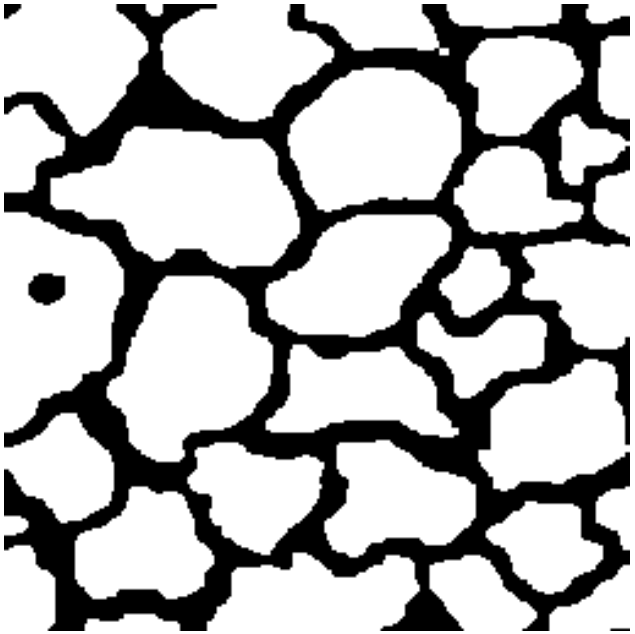
- $3D + t$ images
- Other kind of spaces (feature spaces, ...)

# Why genericity becomes interesting ?

## Multidimensional & multispectral aspects

### Types d'image

- Binary



# Why genericity becomes interesting ?

## Multidimensional & multispectral aspects

### Types d'image

- Binary
- Scalar

# Why genericity becomes interesting ?

## Multidimensional & multispectral aspects

### Types d'image

- Binary
- Scalar
- Colour

# Why genericity becomes interesting ?

## Multidimensional & multispectral aspects

### Types d'image

- Binary
- Scalar
- Colour
- Multi-spectral

# Functions and processings

## Problem $n^o1$ : high functional redundancy

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

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

**Number of combinations**

# 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** | | | |

**Number of combinations**

# 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** | | | |

**Number of combinations**

# 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<br>integer 8bits | scalar<br>integer 8bits | |

**Number of combinations**

# 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 | ..... | |
| | integer 8bits | integer 8bits | | |

**Number of combinations**

# 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**

# 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**

| nb dimensions |
|---|
| 2 |

# 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**

nb dimensions $\times$ nb types

$2 \times 4$

# 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**

$$\text{nb dimensions} \times \text{nb types} \times \text{nb types}$$
$$2 \times 4 \times 4$$

# 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 $n^o2$ : algorithmic redundancy

### Labelling

Mesures du nombre
de composantes

# Processings

## Problem $n^o2$ : algorithmic redundancy

### Labelling

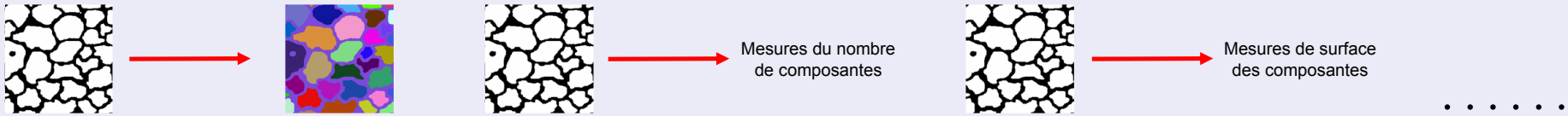
Mesures du nombre de composantes

Implementation too specific !

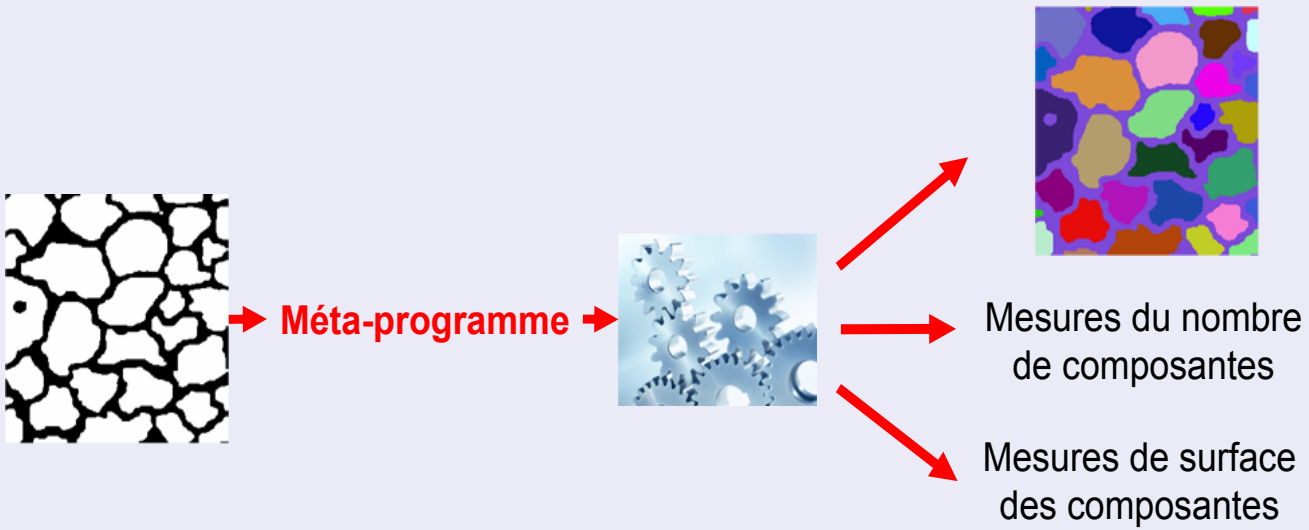
# Processings

## Problem $n^o 2$ : algorithmic redundancy

### Without meta-programming




### With meta-programming



# Genericity by meta-programming approach

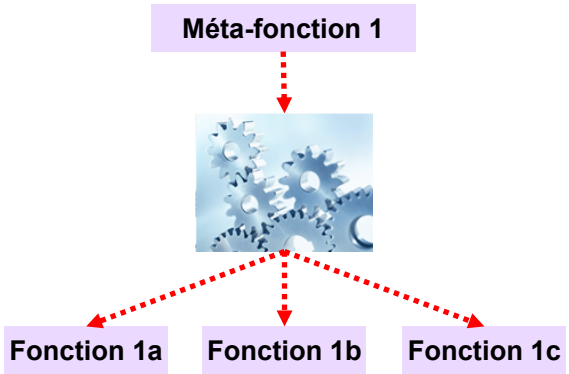
1. Focusing the efforts on the implementation of the algorithms
2. Capitalisation
3. High and efficient code reuse
4. It is "easy" to make the types abstract (and to port - a first version of - existing algorithms)

# Meta-programming

## What is it ?

### Meta-programming ?

1. Types resolution

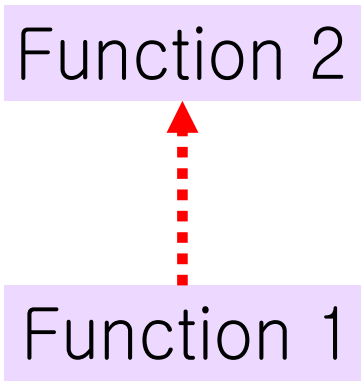


# Meta-programming

## What is it ?

### Meta-programming ?

1. Types resolution
2. Specialising

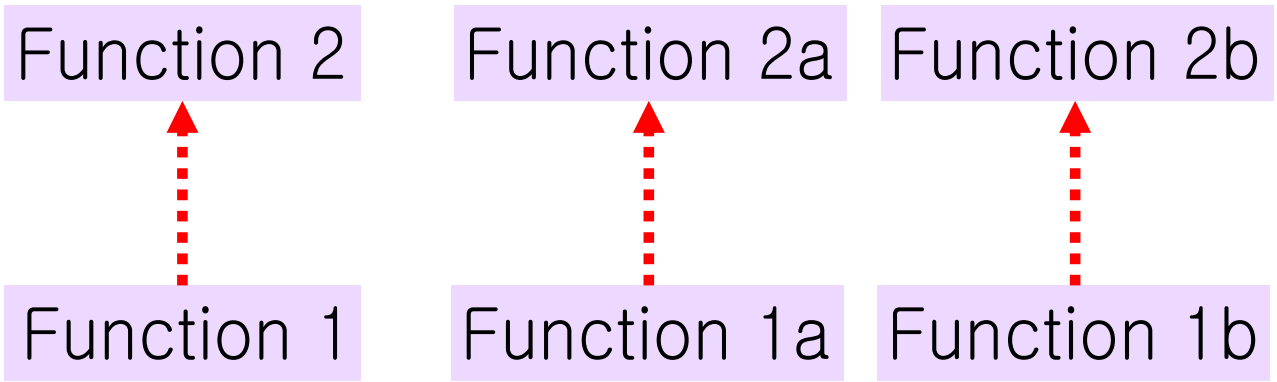


# Meta-programming

## What is it ?

### Meta-programming ?

1. Types resolution
2. Specialising

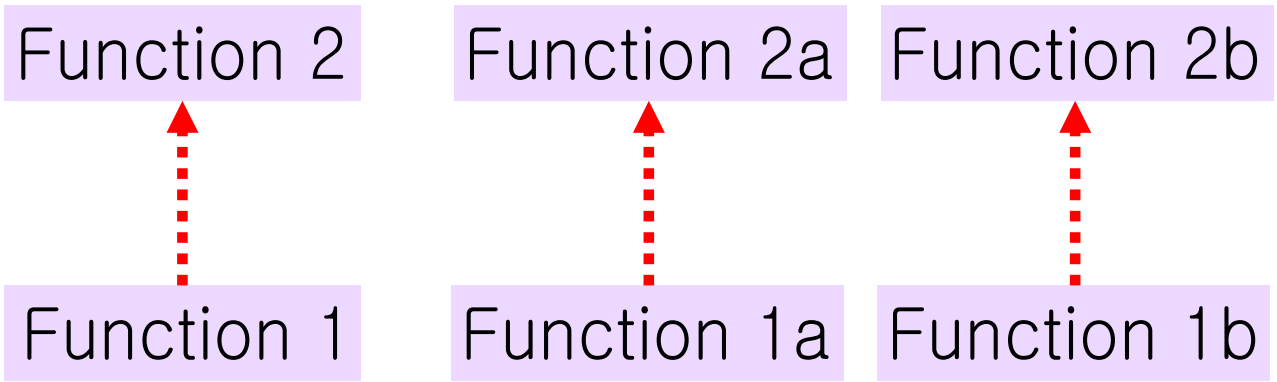


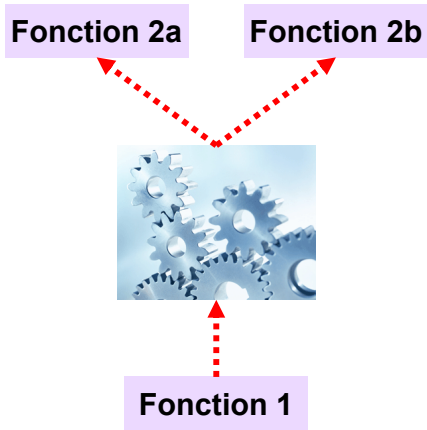
# Meta-programming

## What is it ?

### Meta-programming ?

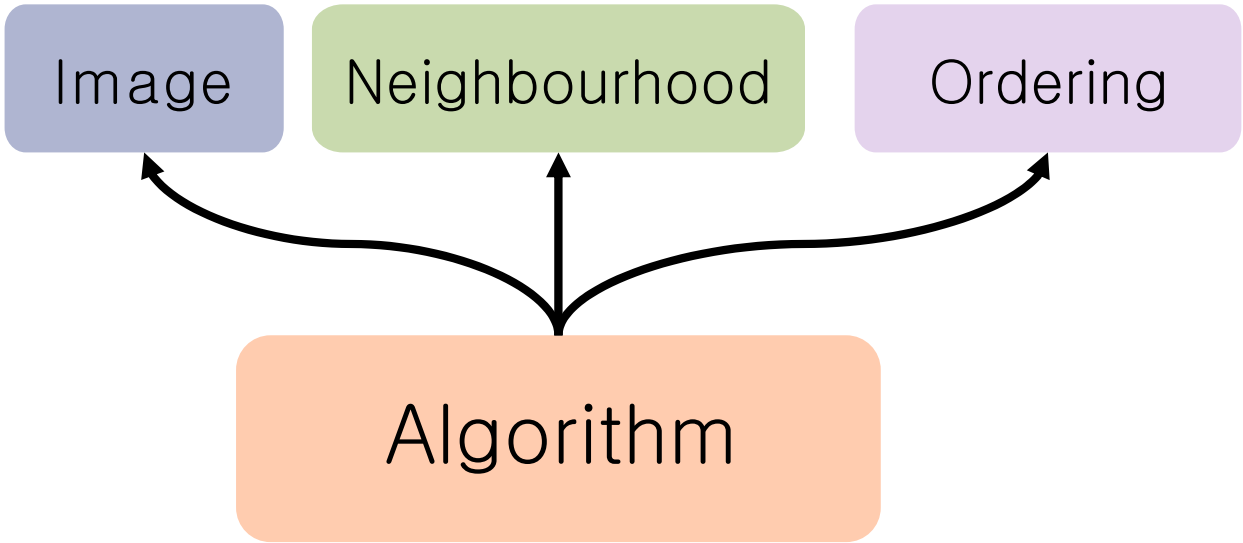
1. Types resolution
2. Specialising



# 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

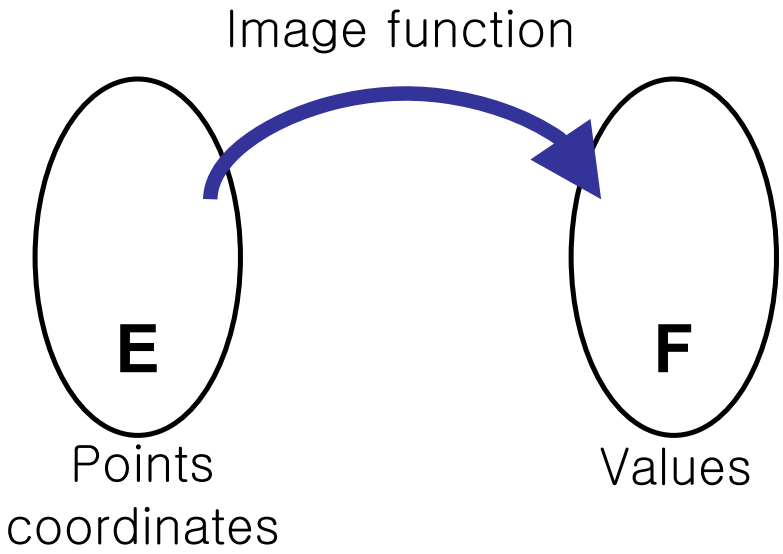
# Images

## Definition

### Definition

An image $\mathcal{I}$ is (no more, no less) an application $\mathcal{I} : \mathbf{E} \rightarrow \mathbf{F}$ associating to each coordinate $p$ of its domain a value $v$

- **E** : coordinate space
- **F** : value space



# How to discover the image domain ?

## Multidimensional aspects

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

### "C" like approach

Query every axis range

# How to discover the image domain ?

## Multidimensional aspects

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

### "C" like approach

Query every axis range

For $z \in [Z_{min}, Z_{max}]$ {
    For $y \in [Y_{min}, Y_{max}]$ {
        For $x \in [X_{min}, X_{max}]$ {

}}}

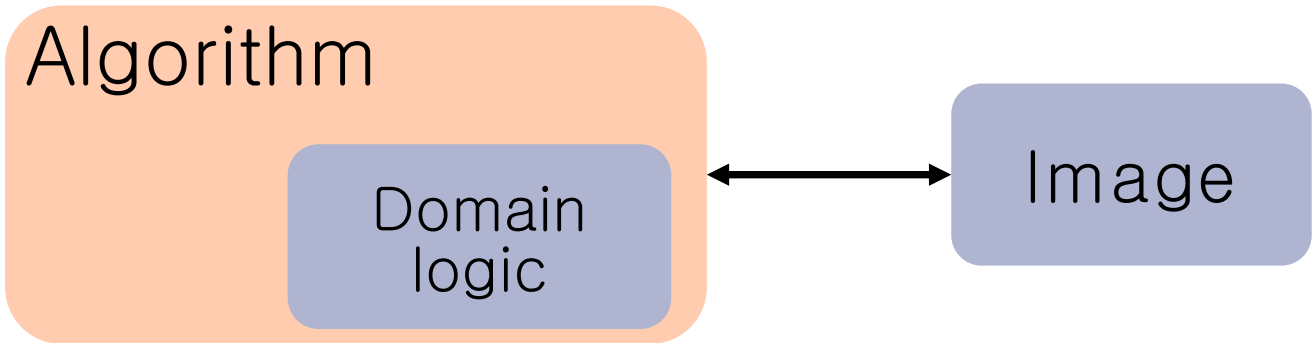
# How to discover the image domain ?

## Multidimensional aspects

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

### "C" like approach

Query every axis range
For $z \in [Z_{min}, Z_{max}]$ {
    For $y \in [Y_{min}, Y_{max}]$ {
        For $x \in [X_{min}, X_{max}]$ {
            Processing at point $(x, y, z)$
}}}



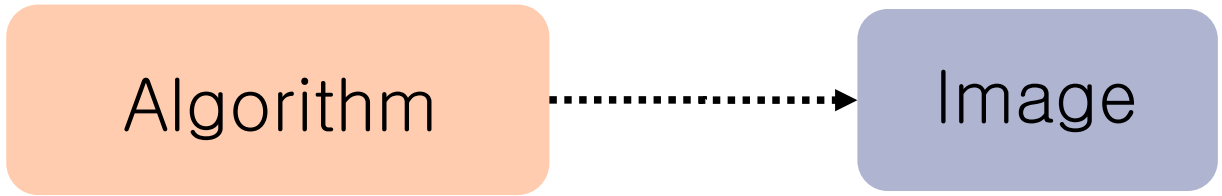
# How to discover the image domain ?

## Multidimensional aspects

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

### Iterator approach

Query an domain iteration object *it* to the image



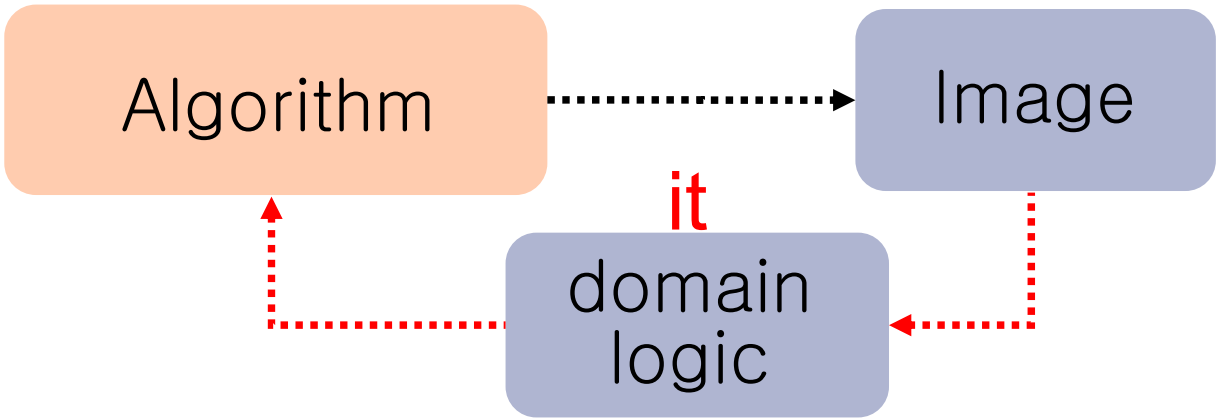
# How to discover the image domain ?

## Multidimensional aspects

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

### Iterator approach

Query an domain iteration object *it* to the image



# How to discover the image domain ?

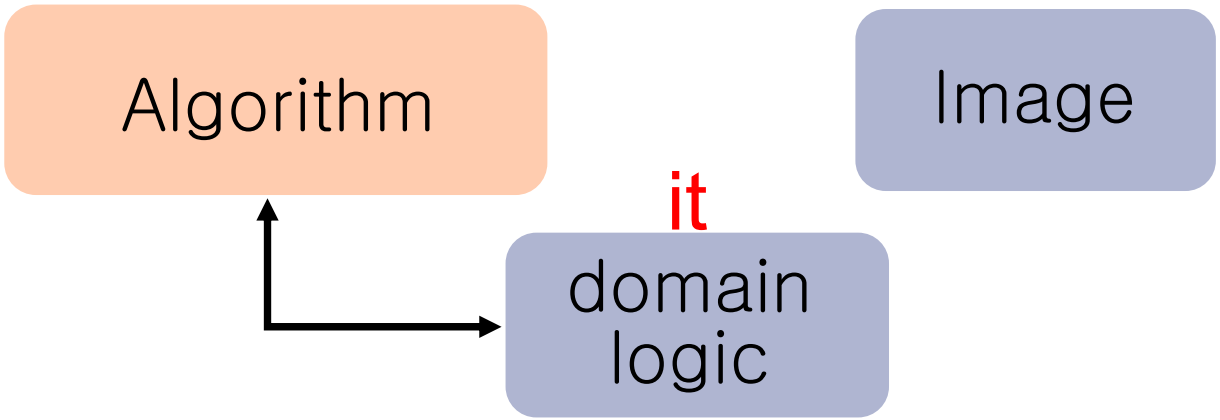
## Multidimensional aspects

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 {

}



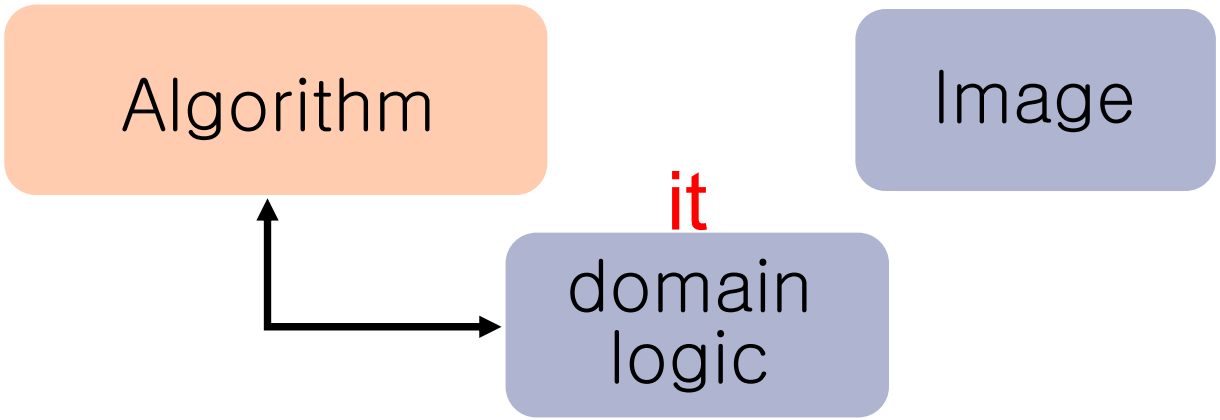
# How to discover the image domain ?

## 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,...)

### Pros

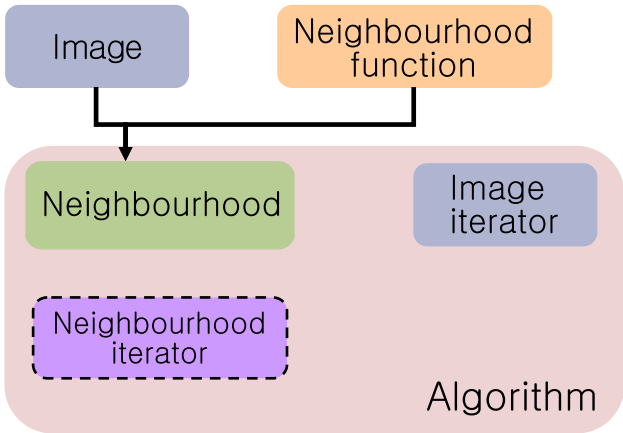
1. Less efficient than pure "C" like approaches
2. "Discrete" domain

# Neighbourhoods

## Structuring functions

### Use case

1. Neighbourhood initialization (image, neighbouring function)

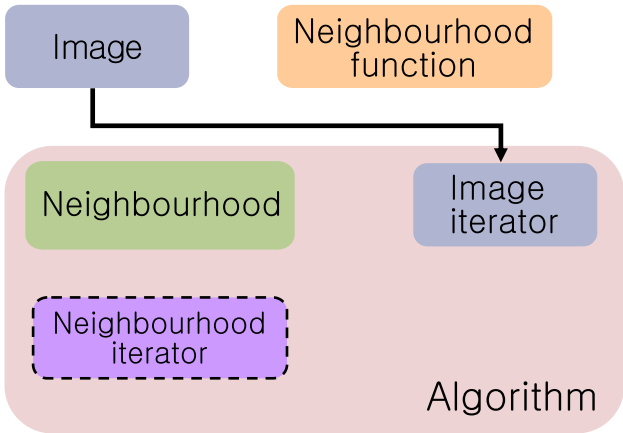


# Neighbourhoods

## Structuring functions

### Use case

1. Neighbourhood initialization (image, neighbouring function)

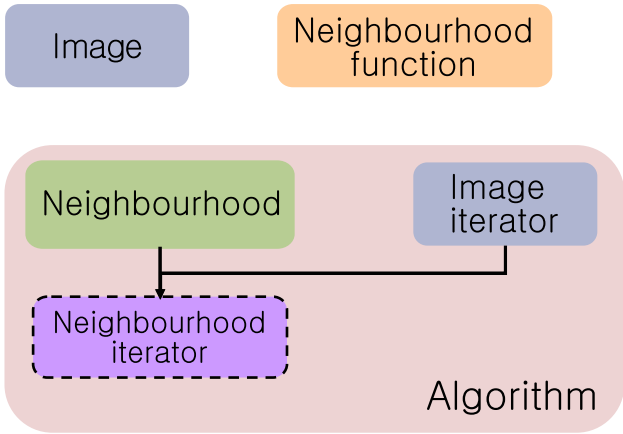


# Neighbourhoods

## Structuring functions

### Use case

1. Neighbourhood initialization (image, neighbouring function)
2. Centring of the neighbourhood

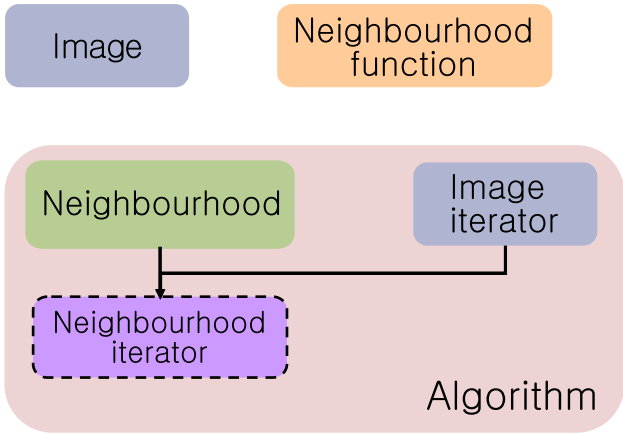


# Neighbourhoods

## 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

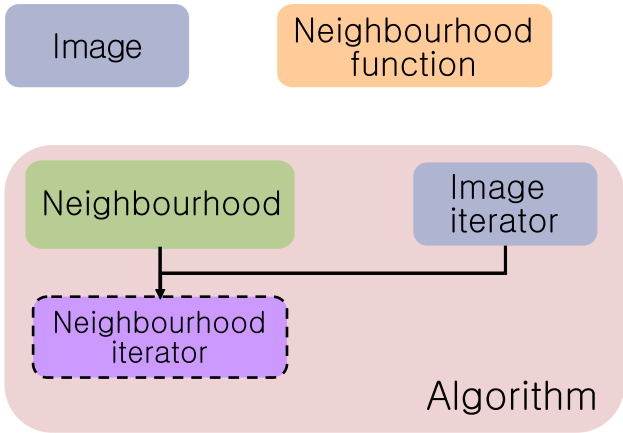


# Neighbourhoods

## 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



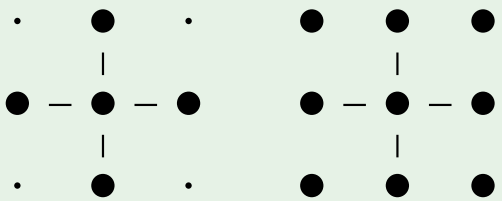

### Pros of iterator approach

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

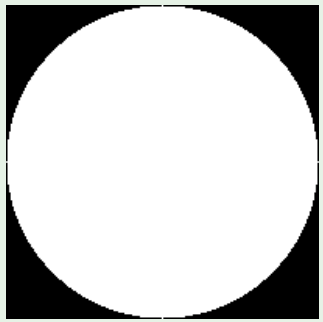
# Neighbourhoods

## Structuring functions

### Examples of neighbouring "functions"



4 & 8 connexion on $2D$ plane



More complex functions . . .

# Neighbourhoods

## Structuring function

Image topology management delegated to a structuring function

Structuring Elements following the optical flow

Nicolas Laveau & Christophe Bernard.
*Structuring elements following the optical flow.*
Proceedings of the 7th ISMM, 2005.

# 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.

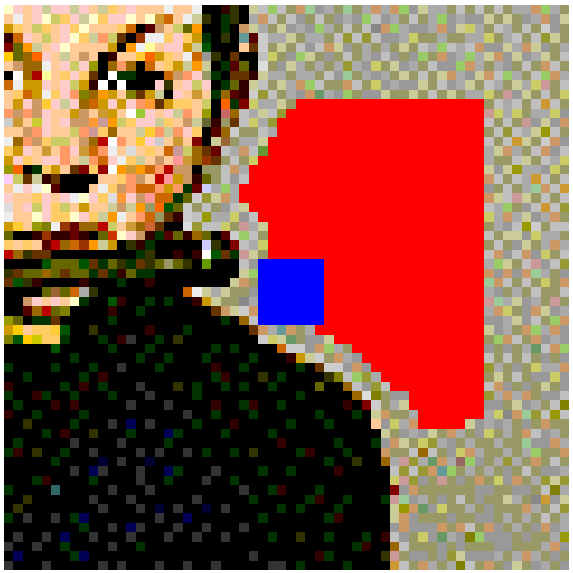
# Neighbourhoods

## 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.



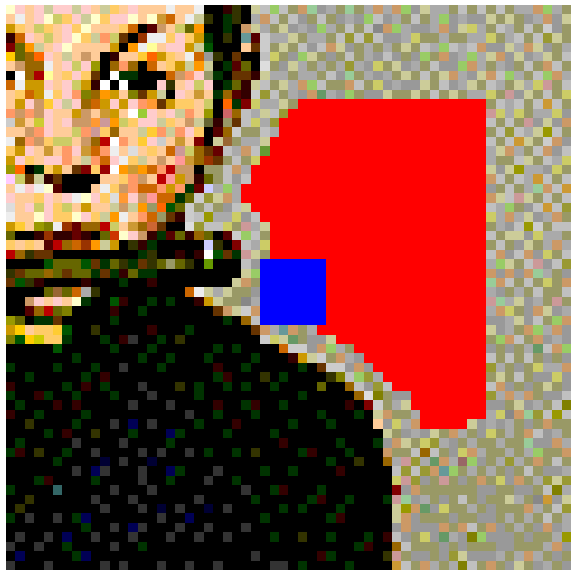
# Neighbourhoods

## 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.



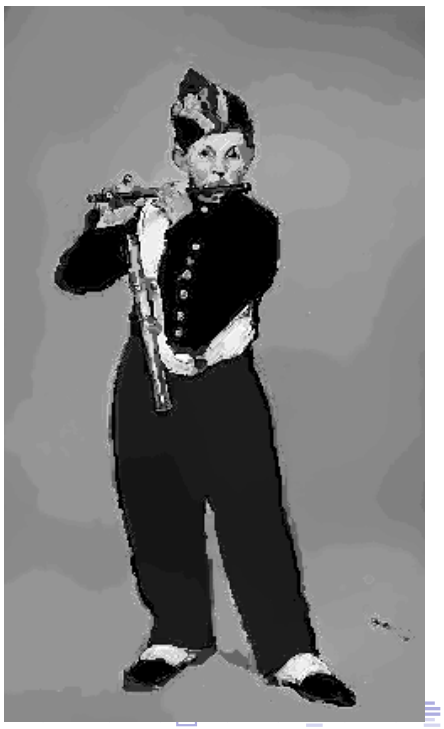
# Neighbourhoods

## 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

# Existing image processing libraries

1. Morph-M : Mathematical Morphology Centre : big amount of function, under active maintenance and development, reference implementation, closed source
2. Olena : LRDE
3. Fulguro : C.Clienti : very fast, maintained, not that generic
4. Mamba : son of Micromorph : fast, not that generic
5. Vigra : generic approach of image processing (one of the first !), not that much MM oriented
6. ....

# Yayi

## Main design objectives

- Mainly dedicated to Mathematical Morphology
- Same concepts as presented
- Open source under (very) permissive Boost licence
- Reference algorithms
- Highly generic, "easy" to use with different type of usage
- Tentative to overcome in a generic manner the slowness problem of generic approaches

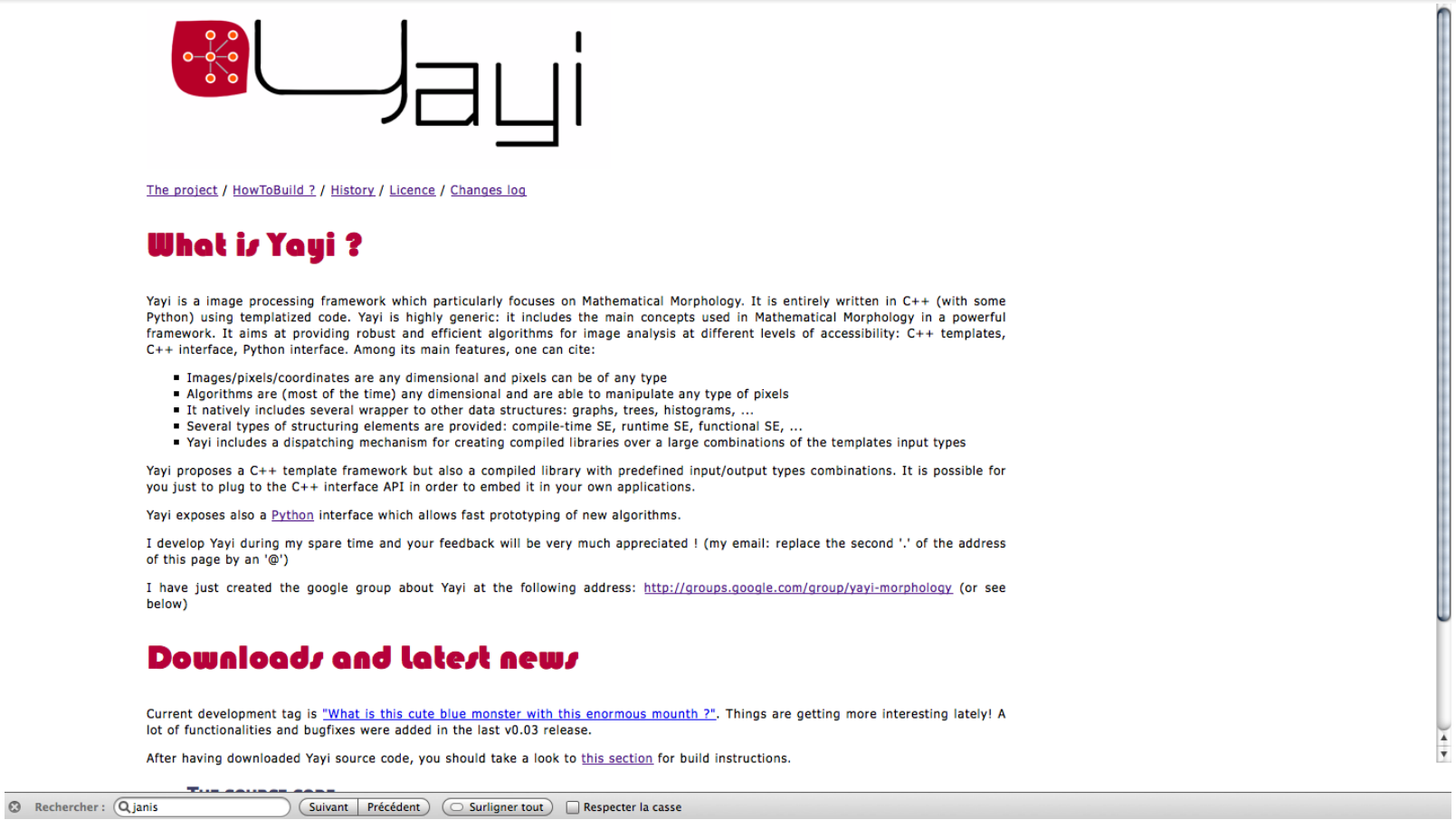
# Yayi

## Design & licence

1. Open source, Boost licence
2. C++ / Python : cross-platform source code, standard "compliant" (current targets : Ubuntu 32/64, MacOSX Leopard, Win32(/64 ?))
3. Low dependencies (Boost, Jpeg, PNG, HDF5 (optional)), all under permissive licence
4. Split of the library into functional blocks (each of which having its own exports and tests)
5. No patented code
6. Generic and easy to extend

# Web site

http ://raffi.enficiaud.free.fr




"Google" group for discussions, news and distribution

No online SVN repository (code is released on archives)

# Constituting blocks of the library

Functional division :

**Clarity, modularity**

**Inter-library dependency sometimes hard to handle, code duplication**

- YayiCommon : structures communes à toutes les autres librairies (types, variants, graphs, interfaces de base, 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 (experimental))
- 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
- YayiDistances : distance transform algorithms
- YayiSegmentation : segmentation algorithms

# CMake

## Aim

Having a build system that is reliable, cross-platform, easy to maintain and unique

Allows also (very important) unit tests (C++ and Python) and the automatic generating of installers.

## Feature

For each library :

1. main library
2. unit and regression tests (using boost :: unit_test )
3. python layer (using boost :: python), with python tests (using python. unit_test )

# Constituting layers

# 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
6  typedef Image<yaUINT8, s_coordinate<3> > image_type3D;
7  image_type3D im3D;
8  im3D.SetSize(c3D(10, 15, 20)); // test return
9  im3D.AllocateImage(); // test return
10
11 // "block" iterators
12 for(image_type3D::iterator it(im3D.begin_block()), ite(im3D.end_block()); it != ite;
       ++it) {
13   *it = 0;
14 }
15
16 // "windowed" iterators
17 image_type3D::window_iterator it(im3D.begin_window(c3D(2,2,1), c3D(5,5,1))),
18   ite(im3D.end_window(c3D(2,2,1), c3D(5,5,1)));
19
20 Image<yaF_simple, s_coordinate<3> > im3Dtemp;
21 im3Dtemp.set_same(im3D);
22
23 copy_image_t(im3D, im3Dtemp); // test result
```

# 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

### Operation at pixel level

```
1  template <class in1_t, class val_t, class out_t>
2  struct s_mult_constant :
3    std::unary_function< typename boost::call_traits<in1_t>::param_type, out_t> {
4    typename boost::add_const<val_t>::type value;
5    s_mult_constant(typename boost::call_traits<val_t>::param_type p) : value(p) {}
6    out_t operator()(typename boost::call_traits<in1_t>::param_type v1) const throw() {
7      return static_cast<out_t>(v1 * value);
8    }
9  };
```

### Operation at image level

```
1  template <class image_in1_, class val_t, class image_out_>
2  yaRC multiply_images_constant_t(const image_in1_& imin1,
3    typename boost::call_traits<val_t>::param_type val,
4    image_out_& imo) {
5
6    typedef s_mult_constant<typename image_in1_::pixel_type, val_t,
7      typename image_out_::pixel_type>  operator_type;
8    s_apply_unary_operator op_processor; // pixel processor
9    operator_type op(val);
10   return op_processor(imin1, imo, op);
11 }
```

# 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 1 : 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 $f(\cup X_i) = g(f(X_1), g(f(X_2), \ldots))$ (eg. histograms, $\int$, $\ldots$)
- Less simple case 2 : given a non-stateless functor $f$, $\exists$ a function $g$ that puts $f$ into the same state as of calling $f$ several times, independently from the sequence on which it is applied (eg. counter)

# 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)

# Dispatching mechanism

## Aim

Find the appropriate template function instance that matches with the interface runtime arguments

## Implementation

Heavily relying on several features of Boost :

1. boost :: function_types : arguments type discovery at compilation time
2. boost :: fusion : type lists iteration and reduction
3. boost :: mpl : type manipulation

# Dispatching mechanism

## Details

Given $n$ input/output "interface" parameters, call the proper instance of the template function that matches the type of the $n$ parameter. For this, we should test the possibility to express/convert the $n$ interface parameters with the $n$ parameters of $F$.

Mechanism falling into two steps :

1. At compilation
   1. determine the exact arity $n$ and the type (function, class method) of the template function/method $F$
   2. recurse over $n$ : for each $k \in \mathbb{N}_n^*$, find a convertor between the interface type $A_k$ and the template type $T_k$ [1]
2. At runtime :
   1. recurse over $n$ : for each $k \in \mathbb{N}_n^*$, test the convertibility of the instance $a_k$ of type $A_k$ to a type $T_k$
   2. if convertible, perform the conversion
   3. call the function $F$ with the converted parameters
   4. convert back the input/output instances of the parameters

1. also prevents from compiling if no convertor was found

# Dispatching mechanism

## Details

```
1  template <
2    class return_t = boost::mpl::void_,
3    class A1 = boost::mpl::void_, class A2 = boost::mpl::void_
4    /* ... */ >
5    struct s_dispatcher
6  {
7    // ...
8    s_dispatcher(r  = R0(), a1 = A1(), a2 = A2(), ...);
9    template <class F> yayi::yaRC operator()(F function) const;
10   template <class vector_functions>
11     yayi::yaRC calls_first_suitable(vector_functions v) const;
12 };
```

During compilation, for each interface argument $a_x$, find a mapping with a concrete type of the template function. Currently, 3 types of conversion are supported :

1. compilation : implicit conversion allowed
2. dynamic_cast : mother/daughter classes (eg. image)
3. runtime : conversion dependant on the content (eg. variant, graph)

```
1 struct s_convertible_compile_time {};
2 struct s_convertible_runtime_time {};
3 struct s_convertible_dynamic_cast {};
```

# Dispatching mechanism

## Some insight on converters

```
1 template <class I, class T, bool B_WRITE_ONLY = false> struct s_conversion_policy;
```

Structure with 2/3 methods : "query", "convert" (and eventually "const convert").

```
1  template <class I, class T, bool B_WRITE_ONLY = false>
2  struct s_no_conversion
3  {
4    typedef boost::false_type type;        // compile-time check
5    typedef typename remove_const<I>::type Icv;
6    typedef typename add_reference<typename add_const<I>::type>::type I_;
7
8    static inline bool  is_convertible  (I_ )
9      {YAYI_THROW("Should not be called");}
10   static inline T     convert         (Icv )
11     {YAYI_THROW("Should not be called");}
12 };
```

# Dispatching mechanism

## Some details on the variant converter

```
1  template <class T, bool B_WRITE_ONLY>
2  struct s_conversion_policy<yayi::variant&, T, B_WRITE_ONLY > {
3    // conversion success depends on the content of the variant
4    typedef s_convertible_runtime_time type;
5  };
```

```
1  template <class T> struct s_runtime_conversion<yayi::variant&, T, false > {
2    typedef boost::true_type type;
3    typedef yayi::variant variant_t;
4    typedef boost::true_type need_holder_tag;
5
6    typedef typename remove_reference<T>::type T_noref;
7
8    static bool is_convertible(const variant_t& r_) throw() {
9      try {
10       r_.operator T_noref();
11       return true;
12     }
13     catch(errors::yaException& /*e*/) {
14       return false;
15     }
16   }
17   static const T_noref convert(const variant_t& r_) {
18     return r_.operator T_noref();
19   }
20   static T_noref convert(variant_t& r_) {
21     return r_.operator T_noref();
22   }
23 };
```

# Dispatching mechanism

## Some details on the image converter

```
1  template <class T, class coordinate_type, class allocator_type>
2  struct s_conversion_policy<
3    IImage*,
4    Image<T, coordinate_type, allocator_type>&,
5    false >
6  {
7    typedef s_convertible_dynamic_cast type;
8  };
```

```
1  yaRC CopySplitChannels(const IImage* imin, IImage* imout1, IImage* imout2, IImage*
       imout3) {
2    yaRC return_value;
3    yayi::dispatcher::s_dispatcher<yaRC, const IImage*,
4      IImage*,IImage*,IImage*>
5      dispatch_object(return_value, imin, imout1, imout2, imout3);
6
7    yaRC res = dispatch_object.calls_first_suitable(
8      fusion::vector_tie(
9        channels_split_t< Image<s_compound_pixel_t<yaUINT8, mpl::int_<3> > >,
10                         Image<yaUINT8> >,
11       channels_split_t< Image<s_compound_pixel_t<yaF_simple, mpl::int_<3> > >,
12                         Image<yaF_simple> >,
13       channels_split_t< Image<s_compound_pixel_t<yaF_double, mpl::int_<3> > >,
14                         Image<yaF_double> >
15   ));
16   if(res == yaRC_ok)
17     return return_value;
18   return res;
19 }
```

# Python layer

**Aim**

To have a simple, easy to use, powerful and pluggable usage tool for conducting experiments, which is equivalent to the interface layer.

Relying on Boost.Python.

# Python layer - example

## 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_I1(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)
```

# Contents

# Algorithms & functions

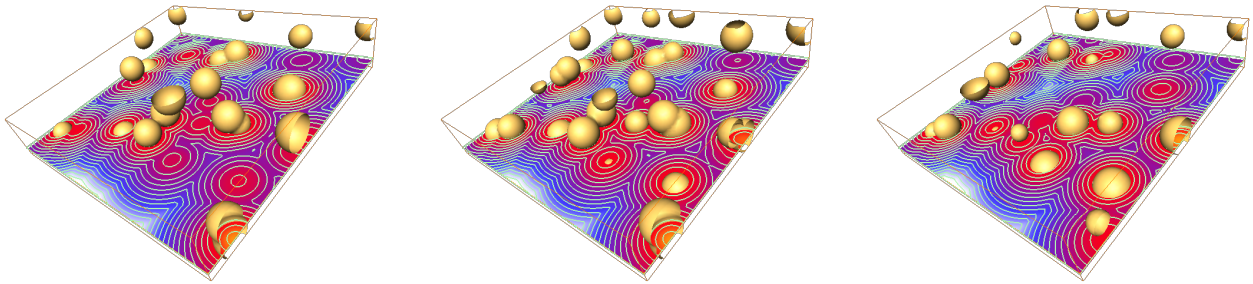
## Currently available

- Basic morphology (grey scale)
  - erosions, dilations, geodesic erosion/dilation
  - openings, closings
- Distances
  - morphological distances
  - quasi distance
- Labellings
  - connected components with adjacency predicates
  - connected components with measurements (area...)
  - local extrema
  - adjacency graph
- Segmentation
  - isotropic watershed

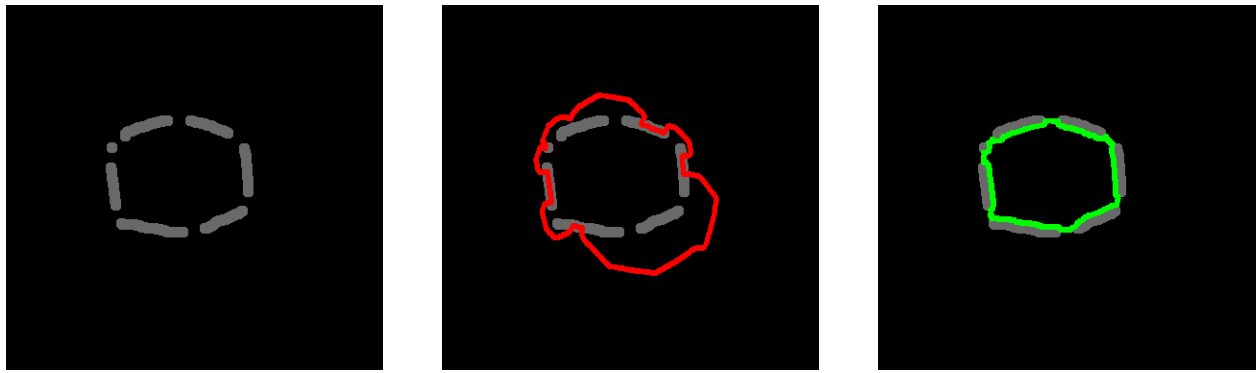
# Algorithms & functions

## Under active development - algorithms

- exact distances in any dimension



- viscous watershed



# Algorithms / functions / structures

## Roadmap

### Priority 0

1. measurements & statistics library
2. color transforms
3. operations on regions
4. homothetic structuring elements

### Priority 1

1. multithreaded pixel-wise operations + benches
2. reducing the complexity of neighbourhood operations

# Algorithms / functions / structures

## Roadmap

### Priority 2

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

### Priority 3

1. reduce compilation time (pre-compiled headers ?)
2. lattice structure

### Priority 4

1. distributed image structure (for very large data)

# Synthesis

# Synthesis

Yayi ...

1. open source, free, under a permissive licence
2. already operational over many mathematical morphology functions/methods/algorithms
3. is getting bigger
4. is waiting for your feedback !

# Thank you for your attention !

Release early, (and try to) release often ! [2]

2. Dixit Romain Lerallut, LRDE, January 2008