Olena Reference Manual

EPITA Research and Development Laboratory

August 23, 2003

This manual documents Olena, a generic image processing library.

Copyright 2001, 2002, 2003 Laboratoire de Recherche et Développement de l'Epita.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

Contents

Chapter 1

Introduction

This reference manual will eventually document any public class and functions available in Olena. Sadly, it only covers Intègre and the morphological processing presently.

The `demo/` directory contains a few sample programs that may be worth looking at before digging the source or sending us an email (`olena@lrde.epita.fr`).

Chapter 2

Core library

This chapter will documents the core of the Olena library.

Chapter 3

Intègre: basic data types

3.1 Global overview

3.1.1 What is Intègre ?

Intègre is a safe and efficient data types library, designed for generic algorithm writing. Even if it is now independent from Olena, it was initially developed to provide value types for images' pixels. Intègre implements basics types such as integers, floats, complexes, vectors but also more evolved types such as range (in the same spirit than Ada) or cycle.

By safe we wean that all operations (arithmetic, assignments, etc.) are checked. If there is an overflow, the user is noticed, at compile time if possible, at runtime otherwise.

By efficient we mean that using Intègre additional checks and features should not decrease too much the overall performances. This is why Intègre relies intensively on template code and meta programming.

3.1.2 Interactions with builtin types

Intègre is designed to simplify generic algorithms writing (generic in the sense of data type). But sometimes, one may want to use builtin type too in its algorithms, such as `int` for example. Intègre provides commodities to accept both Intègre's types and builtin or external types.

3.1.3 Generic algorithms support

Suppose you want to implement a sum which works on several data types:

```
template <class DataType>
ResultType Sum(DataType vals[10])
{
  ResultType s = zero_for_ResultType();
```

```
  for (unsigned i = 0; i < 10; ++i)
    s += vals[i];
  return s;
}
```

You know that the sum of ten values will certainly not fit into the same data type, so you would like to store the result into a larger data type. The same problem stands for the initial value of the result, `0` is not expressed the same way if we are dealing with integers, complexes or vectors.

Intègre provides an handy way to solve these problems:

```
template <class DataType>
ntg_cumul_type(DataType) Sum(DataType vals[10])
{
  ntg_cumul_type(DataType) s = ntg_zero_val(DataType);
  for (unsigned i = 0; i < 10; ++i)
    s += vals[i];
  return s;
}
```

This algorithm will work with almost all Intègre's data types supporting arithmetic addition, but also with c++ builtin types.

You can refer to the following chapters for more informations.

3.1.4 Available types

Here is a list of available types, you can refer to the type reference section for more details about each type.

- Reals
  - unsigned integer (`int_u`)
  - signed integer (`int_s`)
  - float with single precision (`float_s`)
  - float with double precision (`float_d`)
- Enumerated
  - binary type (`bin`)
- Vectorials
  - Static vectors (`vec`)
  - Complexes (`cplx`)
- Decorators
  - Bounding type (`range`)
  - Cycling type (`cycle`)

3.2 Using Intègre

3.2.1 Compilation

Intègre is an active library, and does not provide any object file. It only provides headers containing generic types and functions. So one just have to add a compilation flag to make the compiler find Intègre headers.

3.2.2 Namespace

All Intègre public services are in the `ntg` namespace.

3.2.3 Includes

Special include files:

- `<ntg/all.hh>`: include all Intègre features.
- `<ntg/basics.hh>`: include all Intègre mecanisms, but does not include any particular Intègre type.

Other useful include files are mentioned in the documentation of the concerned features.

3.3 Types properties

To simplify generic algorithms implementation, we need a way to get types properties by a generic way. Traits are defined to do this. As they are implemented using traits, it is possible to define properties for builtin or external types. So one can get properties for non Intègre native types.

3.3.1 Naming scheme and conventions

Properties associated to types can take two forms: types or values.

3.3.2 Properties which are types

They are suffixed by `_type`, for example: `larger_type`, `cumul_type`, `abstract_type`, etc.

Here is the list of common properties:

`abstract_type`

Empty classes representing the kind of the type. This is useful since builtin can have associated abstract type. As these classes are organized by a hierarchical way, their main interest is static concept checking.

Here are the available abstract types:

`ntg_type`

Intègre provides a good interaction with builtin types, but sometimes it is useful to get the Intègre's type associated with a builtin one (if it exists). For example, `ntg_type` for `unsigned char` is `int_u8`.

`base_type`

When using decorators types, such as `range` or `cycle`, one may needs to know the original undecorated type. `base_type` represent this type.

`storage_type`

Intègre's types generally use builtin types to store their value. For example, `int_u8` uses `unsigned char` to store its actual value. `storage_type` represent the type used to store values.

3.3.3 Properties which are values

Values properties are implemented by static functions, to ensure a compile time evaluation and to avoid any useless memory usage.

`name()`

Returns a `string` with the name of the type. This can be useful for debugging purposes.

3.3.4 Accessing types properties

To access to the properties describe above, there is two ways, using directly `type_traits` or using macros hiding the access.

`type_traits<T>`

This is the universal way to access properties. For example, use `type_traits<int_u8>::name()` or `type_traits<unsigned char>::abstract_type` to get the name of `int_u8` and the `abstract_type` representing the builtin type `unsigned char`.

Using `type_traits` can be rather fastidious (it generally requires a `typename` keyword), so macros have been defined to simplify accesses. You should use `type_traits` only when it is not possible to use macros (for example when there is a comma in the name of the type).

Macros

Macros are defined to every possible property. The naming scheme is simple: `ntg_property_type(Type)` is one wants to access to a type property, `ntg_property_val` for a value property. Here are a few examples: `ntg_max_val(int_u8)`, `ntg_abstract_type(float_s)`, etc.

3.4 Ensuring programs safety

3.4.1 Concept checking

One may want to write an algorithm working only on integers, whatever its exact type. To accept both builtin types and Intègre types, he has to write something like that:

```
template <class T>
void algorithm_on_integer(const T& my_int)
{
  // ...
}
```

There is no way to put a constraints on `T` since we want to accept builtin types. A good solution to ensure program integrity is to insert a structucal check using the `ntg_is_a(T, U)` macro. It checks whether the abstract type of `T` is a subclass of U, and return a metalic boolean (refer to metalic documentation).

So here the user should write:

```
template <class T>
void algorithm_on_integer(const T& my_int)
{
  ntg_is_a(T, integer)::ensure();
  // ...
}
```

If `ntg_abstract_type(T)` is not an `integer`, compilation will fail with a clear error message.

3.4.2 Overflow checking

Intègre is designed to be a safe data types library. It tries to make programs safer checking and noticing wrong computations. We want to prevent implicit side effects, often really difficult to find out. Here is an example with builtin types:

```
int i = 256;
unsigned char foo = i; // foo == 0
```

Another one pointing out arithmetic problems:

```
unsigned int i = UINT_MAX;
unsigned int j = 5;
unsigned long long k = i + j; // k == 4
```

These behaviors can be really painful for the programmer, and mostly are overflow problems. This is why Intègre introduces various overflow checks for assignments and arithmetic operations.

Sometimes checks have to be dynamic, for example assigning an `int_u16` into an `int_u8` may be valid, depending on the value of the `int_u16`. This can only be performed at execution time, when we know the actual value.

Some checks can be avoided however, for example assigning an `int_u8` into an `int_u16` is always safe. This is the main justification of Intègre strong typed paradigm, we want to avoid a maximal quantity of checks at runtime.

Strong typing and growing types

To keep a maximal amount of static information about variables range of values, arithmetical operations make types growing. For example, adding 2 `int_u8` returns an `int_u9`. Growing rules for each type is detailed in the types reference section.

This results in the avoidance of a lot of dynamic checks.

Disabling dynamic checks

If you know that your programs works, you can disable dynamic checks defining the macro `NDEBUG` at compile time.

3.4.3 Behaviors

Intègre can detect overflow problems. But what should it do when it detects one ? Here is the reason to live of behaviors.

strict

Stops the program with an error message when a problem is detected.

saturate

Bounds toward the `max` or the `min` value of the type, depending on which is the nearest.

Example: `int_u<8, saturate> u = 350; //u == 255`.

cycle

Assign the value modulo the value range of the type.

Example: `int_u<8, saturate> u = 257; //u == 1`.

unsafe

Does nothing. Behaves almost the same way than builtin types. You should not use this behavior, but it might be useful in a few special cases.

3.5 Types reference

3.5.1 Hierarchy

Intègre types are organized hierarchically:

3.5.2 `bin`

3.5.3 `cplx`

3.5.4 `cycle`

3.5.5 `float_d`

3.5.6 `float_s`

3.5.7 `int_s`

3.5.8 `int_u`

3.5.9 `range`

3.5.10 `vec`

3.6 FAQ

Chapter 4

Processings

4.1 Morphological processings

Soille refers to *P. Soille, morphological Image Analysis – Principals and Applications*. Springer 1998.

4.1.1 morpho::area_closing

Purpose

Area closing

Prototype

```
#include <oln/morpho/attribute_closing_opening.hh>

oln_concrete_type(I)
morpho::area_closing
(const abstract::non_vectorial_image<I>& input,
const abstract::neighborhood<N>& se, unsigned int area);
```

Parameters

input	IN	input image
se	IN	neighborhood to consider
area	IN	area

Description

Compute an area closing using union/find algorithm. See A. Meijster and M. Wilkinson. A Comparison of Algorithms For Connected Set Openings and Closings. PAMI 24(2), p484–494

See also

morpho::simple_geodesic_dilation, §4.1.28, p.44.

Example

```
image2d<int_u8> im = load("lena256.pgm");
save(morpho::tarjan::area_closing(im, neighb_c4(),500), "out.pgm");
```

Figure 4.1: lena256.pgm

Figure 4.2: out.pgm

4.1.2 morpho::area_opening

Purpose

Area opening

Prototype

```
#include <oln/morpho/attribute_closing_opening.hh>

oln_concrete_type(I)
morpho::area_opening
(const abstract::non_vectorial_image<I>& input,
const abstract::neighborhood<N>& se, unsigned int area);
```

Parameters

input	IN	input image
se	IN	neighborhood to consider
area	IN	area

Description

Compute an area opening using union/find algorithm. See A. Meijster and M. Wilkinson. A Comparison of Algorithms For Connected Set Openings and Closings. PAMI 24(2), p484–494

See also

morpho::simple_geodesic_dilation, §4.1.28, p.44.

Example

```
image2d<int_u8> im = load("lena256.pgm");
save(morpho::tarjan::area_opening(im, neighb_c4(),500), "out.pgm");
```

Figure 4.3: lena256.pgm

Figure 4.4: out.pgm

4.1.3 morpho::beucher_gradient

Purpose

Morphological Beucher Gradient.

Prototype

```
#include <oln/morpho/gradient.hh>

mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::beucher_gradient
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::fast::beucher_gradient
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

oln_concrete_type(I)
morpho::beucher_gradient
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

oln_concrete_type(I)
morpho::fast::beucher_gradient
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);
```

Parameters

B>&		IN
input	IN	input image
se	IN	structural element

Description

Compute the arithmetic difference between the diltation and the erosion of *input* using *se* as structural element. Soille, p67.

See also

morpho::erosion, §4.1.7, p.23,
morpho::dilation, §4.1.6, p.22,
morpho::external_gradient, §4.1.8, p.24,
morpho::internal_gradient, §4.1.20, p.37.

Example

```
image2d<int_u8> im = load("lena256.pgm");
save(morpho::beucher_gradient(im, win_c8p()), "out.pgm");
```

Figure 4.5: lena256.pgm

Figure 4.6: out.pgm

4.1.4 morpho::black_top_hat

Purpose

Black top hat.

Prototype

```
#include <oln/morpho/top_hat.hh>

typename mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::black_top_hat
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

typename mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::fast::black_top_hat
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);
```

Parameters

B>&		IN
input	IN	input image
se	IN	structural element

Description

Compute black top hat of *input* using *se* as structural element. Soille p.105.

See also

morpho::closing, §4.1.5, p.21.

Example

```
image2d<int_u8> im = load("lena256.pgm");
save(morpho::black_top_hat(im, win_c8p()), "out.pgm");
```

Figure 4.7: lena256.pgm

Figure 4.8: out.pgm

4.1.5 morpho::closing

Purpose

Morphological closing.

Prototype

```
#include <oln/morpho/closing.hh>

oln_concrete_type(I)
morpho::closing
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

oln_concrete_type(I)
morpho::fast::closing
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);
```

Parameters

input	IN	input image
se	IN	structural element

Description

Compute the morphological closing of *input* using *se* as structural element.

See also

morpho::erosion, §4.1.7, p.23,
morpho::dilation, §4.1.6, p.22,
morpho::closing, §4.1.5, p.21.

Example

```
image2d<ntg::bin> im = load("object.pbm");
save(morpho::dilation(im, win_c8p()), "out.pbm");
```

Figure 4.9: object.pbm

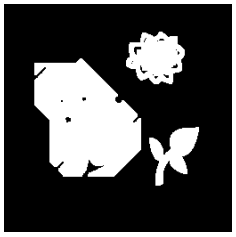

Figure 4.10: out.pbm

4.1.6 morpho::dilation

Purpose

Morphological dilation.

Prototype

```
#include <oln/morpho/dilation.hh>

oln_concrete_type(I)
morpho::dilation
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

oln_concrete_type(I)
morpho::fast::dilation
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);
```

Parameters

input	IN	input image
se	IN	structural element

Description

Compute the morphological dilation of *input* using *se* as structural element.

On grey-scale images, each point is replaced by the maximum value of its neighbors, as indicated by *se*. On binary images, a logical `or` is performed between neighbors.

The `morpho::fast` version of this function use a different

See also

morpho::n_dilation, §4.1.22, p.39,
morpho::erosion, §4.1.7, p.23.

Example

```
image2d<ntg::bin> im = load("object.pbm");
save(morpho::dilation(im, win_c8p()), "out.pbm");
```

Figure 4.11: object.pbm

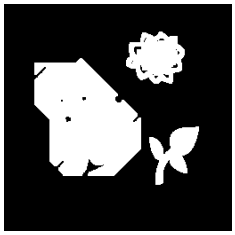

Figure 4.12: out.pbm

4.1.7 morpho::erosion

Purpose

Morphological erosion.

Prototype

`#include <oln/`*/morpho/erosion.hh*`>`

```
oln_concrete_type(I)
morpho::erosion
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

oln_concrete_type(I)
morpho::fast::erosion
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);
```

Parameters

input	IN	input image
se	IN	structural element

Description

Compute the morphological erosion of *input* using *se* as structural element.

On grey-scale images, each point is replaced by the minimum value of its neighbors, as indicated by *se*. On binary images, a logical `and` is performed between neighbors. The `morpho::fast` version of this function use a different

See also

morpho::n_erosion, §4.1.23, p.40,
morpho::dilation, §4.1.6, p.22.

Example

```
image2d<ntg::bin> im = load("object.pbm");
save(morpho::erosion(im, win_c8p()), "out.pbm");
```

Figure 4.13: object.pbm

Figure 4.14: out.pbm

4.1.8 morpho::external_gradient

Purpose

Morphological External Gradient.

Prototype

```
#include <oln/morpho/gradient.hh>

mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::external_gradient
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
```

morpho::fast::external_gradient
`(const convert::abstract::conversion<C B>&,`
`const abstract::non_vectorial_image<I>& input,`
`const abstract::struct_elt<E>& se);`

`oln_concrete_type(I)`
morpho::external_gradient
`(const abstract::non_vectorial_image<I>& input,`
`const abstract::struct_elt<E>& se);`

`oln_concrete_type(I)`
morpho::fast::external_gradient
`(const abstract::non_vectorial_image<I>& input,`
`const abstract::struct_elt<E>& se);`

Parameters

B>&		IN
input	IN	input image
se	IN	structural element

Description

Compute the arithmetic difference between and the dilatation of *input* using *se* as structural element, and the original image *input*. Soille, p67.

See also

morpho::beucher_gradient, §4.1.3, p.19,
morpho::internal_gradient, §4.1.20, p.37,
morpho::dilation, §4.1.6, p.22.

Example

```
image2d<int_u8> im = load("lena256.pgm");
save(morpho::external_gradient(im, win_c8p()), "out.pgm");
```

Figure 4.15: lena256.pgm

Figure 4.16: out.pgm

4.1.9 morpho::fast_maxima_killer

Purpose

Maxima killer.

Prototype

```
#include <oln/morpho/extrema_killer.hh>

oln_concrete_type(I1)
morpho::fast_maxima_killer
(const abstract::non_vectorial_image<I1>& marker,
const unsigned int area area,
const abstract::neighborhood<N>& Ng);
```

Parameters

marker	IN	marker image
area	IN	area
Ng	IN	neighboorhood

Description

It removes the small (in area) connected components of the upper level sets of *input* using *Ng* as neighboorhood. The implementation is based on stak. Guichard and Morel, Image iterative smoothing and PDE's. Book in preparation. p 265.

See also

morpho::sure_maxima_killer, §4.1.32, p.47.

Example

```
image2d<int_u8> light = load("light.pgm");
save(morpho::fast_maxima_killer(light, 20, win_c8p()), "out.pgm");
```

4.1.10 morpho::fast_minima_killer

Purpose

Minima killer.

Prototype

```
#include <oln/morpho/extrema_killer.hh>

oln_concrete_type(I1)
morpho::fast_minima_killer
(const abstract::non_vectorial_image<I1>& marker,
const unsigned int area area,
const abstract::neighborhood<N>& Ng);
```

Parameters

marker	IN	marker image
area	IN	area
Ng	IN	neighboorhood

Description

It removes the small (in area) connected components of the lower level sets of *input* using *Ng* as neighboorhood. The implementation is based on stak. Guichard and Morel, Image iterative smoothing and PDE's. Book in preparation. p 265.

See also

morpho::sure_minima_killer, §4.1.33, p.47.

Example

```
image2d<int_u8> light = load("light.pgm");
save(morpho::fast_minima_killer(light, 20, win_c8p()), "out.pgm");
```

4.1.11 morpho::geodesic_dilation

Purpose

Geodesic dilation.

Prototype

```
#include <oln/morpho/geodesic_dilation.hh>

oln_concrete_type(I1)
morpho::geodesic_dilation
(const abstract::non_vectorial_image<I1>& marker,
const abstract::non_vectorial_image<I2>& mask,
const abstract::struct_elt<E>& se);
```

Parameters

marker	IN	marker image
mask	IN	mask image
se	IN	structural element

Description

Compute the geodesic dilation of *marker* with respect to the mask *mask* image using *se* as structural element. Soille p.156. Note mask must be greater or equal than marker.

See also

morpho::simple_geodesic_dilation, §4.1.28, p.44.

Example

```
image2d<int_u8> light = load("light.pgm");
image2d<int_u8> dark = load("dark.pgm");
save(morpho::geodesic_dilation(dark, light, win_c8p()), "out.pgm");
```

4.1.12 morpho::geodesic_erosion

Purpose

Geodesic erosion.

Prototype

```
#include <oln/morpho/geodesic_erosion.hh>

oln_concrete_type(I1)
morpho::geodesic_erosion
(const abstract::non_vectorial_image<I1>& marker,
const abstract::non_vectorial_image<I2>& mask,
const abstract::struct_elt<E>& se);
```

Parameters

marker	IN	marker image
mask	IN	mask image
se	IN	structural element

Description

Compute the geodesic erosion of *marker* with respect to the mask *mask* image using *se* as structural element. Soille p.158. Note marker must be greater or equal than mask.

See also

morpho::simple_geodesic_dilation, §4.1.28, p.44.

Example

```
image2d<int_u8> light = load("light.pgm");
image2d<int_u8> dark = load("dark.pgm");
save(morpho::geodesic_erosion(light, dark, win_c8p()), "out.pgm");
```

4.1.13 morpho::hit_or_miss

Purpose

Hit_or_Miss Transform.

Prototype

```
#include <oln/morpho/hit_or_miss.hh>

typename mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::hit_or_miss
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se1,
const abstract::struct_elt<E>& se2);
```

```
typename mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::fast::hit_or_miss
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se1,
const abstract::struct_elt<E>& se2);

oln_concrete_type(I)
morpho::hit_or_miss
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se1,
const abstract::struct_elt<E>& se2);

oln_concrete_type(I)
morpho::fast::hit_or_miss
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se1,
const abstract::struct_elt<E>& se2);
```

Parameters

B>&		IN
input	IN	input image
se1	IN	structural element
se2	IN	structural element

Description

Compute the hit_or_miss transform of *input* by the composite structural element (*se1*, *se2*). Soille p.131.

By definition *se1* and *se2* must have the same origin, and need to be disjoint. This algorithm has been extended to every data types (althought it is not increasing). Beware the result depends upon the image data type if it is not `bin`.

Example

```
image2d<ntg::bin> im = load("object.pbm");
window2d mywin;
mywin
 .add(-3,-2).add(-3,-1).add(-3,0).add(-3,1).add(-3,2)
 .add(-2,-1).add(-2,0).add(-2,1)
 .add(-1,0);
window2d mywin2 = - mywin;
save(morpho::fast::hit_or_miss(convert::bound<int_u8>(),
                               im, mywin, mywin2), "out.pgm");
```

Figure 4.17: object.pbm

Figure 4.18: out.pgm

4.1.14 morpho::hit_or_miss_closing

Purpose

Hit_or_Miss closing.

Prototype

```
#include <oln/morpho/hit_or_miss.hh>

oln_concrete_type(I)
morpho::hit_or_miss_closing
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se1,
const abstract::struct_elt<E>& se2);

oln_concrete_type(I)
morpho::fast::hit_or_miss_closing
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se1,
const abstract::struct_elt<E>& se2);
```

Parameters

input	IN	input image
se1	IN	structural element
se2	IN	structural element

Description

Compute the hit_or_miss closing of *input* by the composite structural element (*se1*, *se2*). This is the dual transformation of hit-or-miss opening with respect to set complementation. Soille p.135.

By definition *se1* and *se2* must have the same origin, and need to be disjoint. This algorithm has been extended to every data types (althought it is not increasing). Beware the result depends upon the image data type if it is not `bin`.

See also

morpho::hit_or_miss, §4.1.13, p.28,
morpho::hit_or_miss_closing_bg, §4.1.15, p.31,

morpho::hit_or_miss_opening, §4.1.16, p.32,
morpho::hit_or_miss_opening_bg, §4.1.17, p.33.

Example

```
 image2d<ntg::bin> im = load("object.pbm");
window2d mywin;
mywin
.add(-3,-2).add(-3,-1).add(-3,0).add(-3,1).add(-3,2)
.add(-2,-1).add(-2,0).add(-2,1)
.add(-1,0);
window2d mywin2 = - mywin;
 save(morpho::hit_or_miss_closing(im, mywin, mywin2), "out.pbm");
```

Figure 4.19: object.pbm

Figure 4.20: out.pbm

4.1.15 morpho::hit_or_miss_closing_bg

Purpose

Hit_or_Miss closing of background.

Prototype

```
#include <oln/morpho/hit_or_miss.hh>

oln_concrete_type(I)
morpho::hit_or_miss_closing_bg
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se1,
const abstract::struct_elt<E>& se2);

oln_concrete_type(I)
morpho::fast::hit_or_miss_closing_bg
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se1,
const abstract::struct_elt<E>& se2);
```

Parameters

input	IN	input image
se1	IN	structural element
se2	IN	structural element

Description

Compute the hit_or_miss closing of the background of *input* by the composite structural element (*se1*, *se2*). This is the dual transformation of hit-or-miss opening with respect to set complementation. Soille p.135.

By definition *se1* and *se2* must have the same origin, and need to be disjoint. This algorithm has been extended to every data types (althought it is not increasing). Beware the result depends upon the image data type if it is not `bin`.

See also

morpho::hit_or_miss, §4.1.13, p.28,
morpho::hit_or_miss_closing, §4.1.14, p.30,
morpho::hit_or_miss_opening, §4.1.16, p.32,
morpho::hit_or_miss_opening_bg, §4.1.17, p.33.

Example

```
 image2d<ntg::bin> im = load("object.pbm");
window2d mywin;
mywin
.add(-3,-2).add(-3,-1).add(-3,0).add(-3,1).add(-3,2)
.add(-2,-1).add(-2,0).add(-2,1)
.add(-1,0);
window2d mywin2 = - mywin;
 save(morpho::hit_or_miss_closing_bg(im, mywin, mywin2), "out.pbm");
```

Figure 4.21: object.pbm

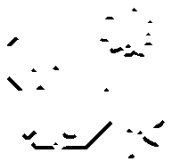

Figure 4.22: out.pbm

4.1.16 morpho::hit_or_miss_opening

Purpose

Hit_or_Miss opening.

Prototype

```
#include <oln/morpho/hit_or_miss.hh>

oln_concrete_type(I)
```

morpho::hit_or_miss_opening
(const abstract::non_vectorial_image<I>& *input*,
const abstract::struct_elt<E>& *se1*,
const abstract::struct_elt<E>& *se2*);

oln_concrete_type(I)
morpho::fast::hit_or_miss_opening
(const abstract::non_vectorial_image<I>& *input*,
const abstract::struct_elt<E>& *se2*,
const abstract::struct_elt<E>& *se2*);

Parameters

input	IN	input image
se1	IN	structural element
se2	IN	structural element

Description

Compute the hit_or_miss opening of *input* by the composite structural element (*se1*, *se2*). Soille p.134.

By definition *se1* and *se2* must have the same origin, and need to be disjoint. This algorithm has been extended to every data types (althought it is not increasing). Beware the result depends upon the image data type if it is not `bin`.

See also

morpho::hit_or_miss, §4.1.13, p.28,
morpho::hit_or_miss_closing, §4.1.14, p.30,
morpho::hit_or_miss_closing_bg, §4.1.15, p.31,
morpho::hit_or_miss_opening_bg, §4.1.17, p.33.

Example

```
 image2d<ntg::bin> im = load("object.pbm");
window2d mywin;
mywin
.add(-3,-2).add(-3,-1).add(-3,0).add(-3,1).add(-3,2)
.add(-2,-1).add(-2,0).add(-2,1)
.add(-1,0);
window2d mywin2 = - mywin;
 save(morpho::hit_or_miss_opening(im, mywin, mywin2), "out.pbm");
```

4.1.17 morpho::hit_or_miss_opening_bg

Purpose

Hit_or_Miss opening of background.

Figure 4.23: object.pbm

Figure 4.24: out.pbm

Prototype

```
#include <oln/morpho/hit_or_miss.hh>

oln_concrete_type(I)
morpho::hit_or_miss_opening_bg
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se1,
const abstract::struct_elt<E>& se2);

oln_concrete_type(I)
morpho::fast::hit_or_miss_opening_bg
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se1,
const abstract::struct_elt<E>& se2);
```

Parameters

input	IN	input image
se1	IN	structural element
se2	IN	structural element

Description

Compute the hit_or_miss opening of the background of *input* by the composite structural element (*se1*, *se2*). Soille p.135.

By definition *se1* and *se2* must have the same origin, and need to be disjoint. This algorithm has been extended to every data types (althought it is not increasing). Beware the result depends upon the image data type if it is not `bin`.

See also

morpho::hit_or_miss, §4.1.13, p.28,
morpho::hit_or_miss_closing, §4.1.14, p.30,
morpho::hit_or_miss_closing_bg, §4.1.15, p.31,
morpho::hit_or_miss_opening, §4.1.16, p.32.

Example

```
image2d<ntg::bin> im = load("object.pbm");
```

```
window2d mywin;
mywin
.add(-3,-2).add(-3,-1).add(-3,0).add(-3,1).add(-3,2)
.add(-2,-1).add(-2,0).add(-2,1)
.add(-1,0);
window2d mywin2 = - mywin;
 save(morpho::hit_or_miss_opening_bg(im, mywin, mywin2), "out.pbm");
```

Figure 4.25: object.pbm

Figure 4.26: out.pbm

4.1.18 morpho::hybrid_geodesic_reconstruction_dilation

Purpose

Geodesic reconstruction by dilation.

Prototype

```
#include <oln/morpho/reconstruction.hh>

oln_concrete_type(I1)
morpho::hybrid_geodesic_reconstruction_dilation
(const abstract::non_vectorial_image<I1>& marker,
const abstract::non_vectorial_image<I2>& mask,
const abstract::struct_elt<E>& se);
```

Parameters

marker	IN	marker image
mask	IN	mask image
se	IN	structural element

Description

Compute the reconstruction by dilation of *marker* with respect to the mask *mask* image using *se* as structural element. Soille p.160. The algorithm used is the one defined as hybrid in Vincent(1993), Morphological grayscale reconstruction in image analysis: applications and efficient algorithms, itip, 2(2), 176–201.

See also
morpho::simple_geodesic_dilation, §4.1.28, p.44.

Example

```
image2d<int_u8> light = load("light.pgm");
image2d<int_u8> dark = load("dark.pgm");
save(morpho::hybrid_geodesic_reconstruction_dilation(light, dark, win_c8p()), "ou
```

4.1.19 morpho::hybrid_geodesic_reconstruction_erosion

Purpose
Geodesic reconstruction by erosion.

Prototype
`#include <oln/`*/morpho/reconstruction.hh*`>`

```
oln_concrete_type(I1)
morpho::hybrid_geodesic_reconstruction_erosion
(const abstract::non_vectorial_image<I1>& marker,
const abstract::non_vectorial_image<I2>& mask,
const abstract::struct_elt<E>& se);
```

Parameters

marker	IN	marker image
mask	IN	mask image
se	IN	structural element

Description
Compute the reconstruction by erosion of *marker* with respect to the mask *mask* image using *se* as structural element. Soille p.160. The algorithm used is the one defined as hybrid in Vincent(1993), Morphological grayscale reconstruction in image analysis: applications and efficient algorithms, itip, 2(2), 176–201.

See also
morpho::simple_geodesic_erosion, §4.1.29, p.45.

Example

```
image2d<int_u8> light = load("light.pgm");
image2d<int_u8> dark = load("dark.pgm");
save(morpho::sequential_geodesic_reconstruction_erosion(light, dark, win_c8p()),
```

4.1.20 morpho::internal_gradient

Purpose

Morphological Internal Gradient.

Prototype

```
#include <oln/morpho/gradient.hh>

mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::internal_gradient
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::fast::internal_gradient
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

oln_concrete_type(I)
morpho::internal_gradient
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

oln_concrete_type(I)
morpho::fast::internal_gradient
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);
```

Parameters

B>&		IN
input	IN	input image
se	IN	structural element

Description

Compute the arithmetic difference between the original image *input* and the erosion of *input* using *se* as structural element. Soille, p67.

See also

morpho::beucher_gradient, §4.1.3, p.19,
morpho::external_gradient, §4.1.8, p.24,
morpho::erosion, §4.1.7, p.23.

Example

```
image2d<int_u8> im = load("lena256.pgm");
save(morpho::internal_gradient(im, win_c8p()), "out.pgm");
```

Figure 4.27: lena256.pgm

Figure 4.28: out.pgm

4.1.21 morpho::laplacian

Purpose

Laplacian.

Prototype

```
#include <oln/morpho/laplacian.hh>

typename mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::laplacian
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

typename mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::fast::laplacian
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

typename mute<I,oln_value_type(I)::slarger_t>::ret
morpho::laplacian
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

typename mute<I,oln_value_type(I)::slarger_t>::ret
morpho::fast::laplacian
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);
```

Parameters

B>&		IN
input	IN	input image
se	IN	structural element

Description

Compute the laplacian of *input* using *se* as structural element.

See also

morpho::dilation, §4.1.6, p.22,
morpho::erosion, §4.1.7, p.23.

Example

```
image2d<int_u8> im = load("lena256.pgm");
save(morpho::laplacian(convert::bound<int_u8>(), im, win_c8p()), "out.pgm");
```

Figure 4.29: lena256.pgm

Figure 4.30: out.pgm

4.1.22 morpho::n_dilation

Purpose

Morphological dilation itered n times.

Prototype

```
#include <oln/morpho/dilation.hh>

oln_concrete_type(I)
morpho::n_dilation
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se, unsigned n);
```

Parameters

input	IN	input image
se	IN	structural element
n	IN	number of iterations

Description

Apply `morpho::dilation` n times.

See also

morpho::dilation, §4.1.6, p.22,
morpho::n_erosion, §4.1.23, p.40.

4.1.23 morpho::n_erosion

Purpose

Morphological erosion itered n times.

Prototype

```
#include <oln/morpho/erosion.hh>

oln_concrete_type(I)
morpho::n_erosion
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se, unsigned n);
```

Parameters

input	IN	input image
se	IN	structural element
n	IN	number of iterations

Description

Apply `morpho::erosion` n times.

See also

morpho::erosion, §4.1.7, p.23,
morpho::n_dilation, §4.1.22, p.39.

4.1.24 morpho::opening

Purpose

Morphological opening.

Prototype

```
#include <oln/morpho/opening.hh>

oln_concrete_type(I)
morpho::opening
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);
oln_concrete_type(I)
morpho::fast::opening
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);
```

Parameters

input	IN	input image
se	IN	structural element

Description

Compute the morphological opening of *input* using *se* as structural element.

See also

morpho::erosion, §4.1.7, p.23,
morpho::dilation, §4.1.6, p.22,
morpho::closing, §4.1.5, p.21.

Example

```
image2d<ntg::bin> im = load("object.pbm");
save(morpho::opening(im, win_c8p()), "out.pbm");
```

Figure 4.31: object.pbm

Figure 4.32: out.pbm

4.1.25 morpho::self_complementary_top_hat

Purpose

Self complementary top hat.

Prototype

```
#include <oln/morpho/top_hat.hh>

typename mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::self_complementary_top_hat
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

typename mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::fast::self_complementary_top_hat
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

typename mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::self_complementary_top_hat
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);
```

```
typename mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::fast::self_complementary_top_hat
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);
```

Parameters

B>&		IN
input	IN	input image
se	IN	structural element

Description

Compute self complementary top hat of *input* using *se* as structural element. Soille p.106.

See also

morpho::closing, §4.1.5, p.21,
morpho::opening, §4.1.24, p.40.

Example

```
image2d<int_u8> im = load("lena256.pgm");
save(morpho::self_complementary_top_hat(im, win_c8p()), "out.pgm");
```

Figure 4.33: lena256.pgm

Figure 4.34: out.pgm

4.1.26 morpho::sequential_geodesic_reconstruction_dilation

Purpose

Geodesic reconstruction by dilation.

Prototype

```
#include <oln/morpho/reconstruction.hh>

oln_concrete_type(I1)
morpho::sequential_geodesic_reconstruction_dilation
(const abstract::non_vectorial_image<I1>& marker,
const abstract::non_vectorial_image<I2>& mask,
const abstract::struct_elt<E>& se);
```

Parameters

marker	IN	marker image
mask	IN	mask image
se	IN	structural element

Description

Compute the reconstruction by dilation of *marker* with respect to the mask *mask* image using *se* as structural element. Soille p.160. The algorithm used is the one defined as sequential in Vincent(1993), Morphological grayscale reconstruction in image analysis: applications and efficient algorithms, itip, 2(2), 176–201.

See also

morpho::simple_geodesic_dilation, §4.1.28, p.44.

Example

```
image2d<int_u8> light = load("light.pgm");
image2d<int_u8> dark = load("dark.pgm");
save(morpho::sequential_geodesic_reconstruction_dilation(light, dark, win_c8p()), "out.pgm"
```

4.1.27 morpho::sequential_geodesic_reconstruction_erosion

Purpose

Geodesic reconstruction by erosion.

Prototype

```
#include <oln/morpho/reconstruction.hh>

oln_concrete_type(I1)
morpho::sequential_geodesic_reconstruction_erosion
(const abstract::non_vectorial_image<I1>& marker,
const abstract::non_vectorial_image<I2>& mask,
const abstract::struct_elt<E>& se);
```

Parameters

marker	IN	marker image
mask	IN	mask image
se	IN	structural element

Description

Compute the reconstruction by erosion of *marker* with respect to the mask *mask* image using *se* as structural element. Soille p.160. The algorithm used is the one defined as sequential in Vincent(1993), Morphological grayscale reconstruction in image analysis: applications and efficient algorithms, itip, 2(2), 176–201.

See also

morpho::simple_geodesic_erosion, §4.1.29, p.45.

Example

```
image2d<int_u8> light = load("light.pgm");
image2d<int_u8> dark = load("dark.pgm");
save(morpho::sequential_geodesic_reconstruction_erosion(light, dark, win_c8p()),
```

4.1.28 morpho::simple_geodesic_dilation

Purpose

Geodesic dilation.

Prototype

```
#include <oln/morpho/geodesic_dilation.hh>

oln_concrete_type(I1)
morpho::simple_geodesic_dilation
(const abstract::non_vectorial_image<I1>& marker,
const abstract::non_vectorial_image<I2>& mask,
const abstract::struct_elt<E>& se);
```

Parameters

marker	IN	marker image
mask	IN	mask image
se	IN	structural element

Description

Compute the geodesic dilation of *marker* with respect to the mask *mask* image using *se* as structural element. Soille p.156. Computation is performed by hand (i.e without calling dilation). Note mask must be greater or equal than marker.

See also

morpho::sure_geodesic_dilation, §**??**, p.**??**.

Example

```
image2d<int_u8> light = load("light.pgm");
image2d<int_u8> dark = load("dark.pgm");
save(morpho::simple_geodesic_dilation(dark, light,
                                      win_c8p()), "out.pgm");
```

4.1.29 morpho::simple_geodesic_erosion

Purpose

Geodesic erosion.

Prototype

```
#include <oln/morpho/geodesic_erosion.hh>

oln_concrete_type(I1)
morpho::simple_geodesic_erosion
(const abstract::non_vectorial_image<I1>& marker,
const abstract::non_vectorial_image<I2>& mask,
const abstract::struct_elt<E>& se);
```

Parameters

marker	IN	marker image
mask	IN	mask image
se	IN	structural element

Description

Compute the geodesic erosion of *marker* with respect to the mask *mask* image using *se* as structural element. Soille p.156. Computation is performed by hand (i.e without calling dilation). Note marker must be greater or equal than mask.

See also

morpho::sure_geodesic_dilation, §**??**, p.**??**.

Example

```
image2d<int_u8> light = load("light.pgm");
image2d<int_u8> dark = load("dark.pgm");
save(morpho::geodesic_erosion(light, dark, win_c8p()), "out.pgm");
```

4.1.30 morpho::sure_geodesic_reconstruction_dilation

Purpose

Geodesic reconstruction by dilation.

Prototype

```
#include <oln/morpho/reconstruction.hh>

oln_concrete_type(I1)
morpho::sure_geodesic_reconstruction_dilation
(const abstract::non_vectorial_image<I1>& marker,
const abstract::non_vectorial_image<I2>& mask,
const abstract::struct_elt<E>& se);
```

Parameters

marker	IN	marker image
mask	IN	mask image
se	IN	structural element

Description

Compute the reconstruction by dilation of *marker* with respect to the mask *mask* image using *se* as structural element. Soille p.160. This is the simplest algorithm: iteration is performed until stability.

See also

morpho::simple_geodesic_dilation, §4.1.28, p.44.

Example

```
image2d<int_u8> light = load("light.pgm");
image2d<int_u8> dark = load("dark.pgm");
save(morpho::sure_geodesic_reconstruction_dilation(light, dark, win_c8p()), "out.
```

4.1.31 morpho::sure_geodesic_reconstruction_erosion

Purpose

Geodesic reconstruction by erosion.

Prototype

```
#include <oln/morpho/reconstruction.hh>

oln_concrete_type(I1)
morpho::sure_geodesic_reconstruction_erosion
(const abstract::non_vectorial_image<I1>& marker,
const abstract::non_vectorial_image<I2>& mask,
const abstract::struct_elt<E>& se);
```

Parameters

marker	IN	marker image
mask	IN	mask image
se	IN	structural element

Description

Compute the reconstruction by erosion of *marker* with respect to the mask *mask* image using *se* as structural element. Soille p.160. This is the simplest algorithm : iteration is performed until stability.

See also

morpho::simple_geodesic_erosion, §4.1.29, p.45.

Example

```
image2d<int_u8> light = load("light.pgm");
image2d<int_u8> dark = load("dark.pgm");
save(morpho::sure_geodesic_reconstruction_erosion(light, dark, win_c8p()), "out.pgm");
```

4.1.32 morpho::sure_maxima_killer

Purpose

Maxima killer.

Prototype

```
#include <oln/morpho/extrema_killer.hh>

oln_concrete_type(I1)
morpho::sure_maxima_killer
(const abstract::non_vectorial_image<I1>& marker,
const unsigned int area area,
const abstract::struct_elt<E>& se);
```

Parameters

marker	IN	marker image
area	IN	area
se	IN	structural element

Description

It removes the small (in area) connected components of the upper level sets of *input* using *se* as structual element. The implementation uses the threshold superposition principle; so it is very slow ! it works only for int_u8 images.

See also

morpho::fast_maxima_killer, §4.1.9, p.25.

Example

```
image2d<int_u8> light = load("light.pgm");
save(morpho::sure_maxima_killer(light, 20, win_c8p()), "out.pgm");
```

4.1.33 morpho::sure_minima_killer

Purpose

Minima killer.

Prototype

```
#include <oln/morpho/extrema_killer.hh>
```

```
oln_concrete_type(I1)
morpho::sure_minima_killer
(const abstract::non_vectorial_image<I1>& marker,
const unsigned int area area,
const abstract::struct_elt<E>& se);
```

Parameters

marker	IN	marker image
area	IN	area
se	IN	structural element

Description

It removes the small (in area) connected components of the lower level sets of *input* using *se* as structual element. The implementation uses the threshold superposition principle; so it is very slow ! it works only for int_u8 images.

See also

morpho::fast_maxima_killer, §4.1.9, p.25.

Example

```
image2d<int_u8> light = load("light.pgm");
save(morpho::sure_minima_killer(light, 20, win_c8p()), "out.pgm");
```

4.1.34 morpho::top_hat_contrast_op

Purpose

Top hat contrastor operator.

Prototype

```
#include <oln/morpho/top_hat.hh>

typename mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::top_hat_contrast_op
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

typename mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::fast::top_hat_contrast_op
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

typename mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
```

morpho::top_hat_contrast_op
`(const abstract::non_vectorial_image<I>& input,`
`const abstract::struct_elt<E>& se);`

`typename mute<I, typename convoutput<C,`
`B,oln_value_type(I)>::ret>::ret`
morpho::fast::top_hat_contrast_op
`(const abstract::non_vectorial_image<I>& input,`
`const abstract::struct_elt<E>& se);`

Parameters

B>&		IN
input	IN	input image
se	IN	structural element

Description

Enhance contrast *input* by adding the white top hat, then substracting the black top hat to *input*. Top hats are computed using *se* as structural element. Soille p.109.

See also

morpho::white_top_hat, §4.1.38, p.51,
morpho::black_top_hat, §4.1.4, p.20.

Example

```
 image2d<int_u8> im = load("lena256.pgm");
 save(morpho::top_hat_contrast_op(convert::bound<int_u8>(),
im, win_c8p()), "out.pgm");
```

Figure 4.35: lena256.pgm

Figure 4.36: out.pgm

4.1.35 morpho::watershed_con

Purpose

Connected Watershed.

Prototype

`#include <oln/`*/morpho/watershed.hh*`>`

```
typename mute<I, DestValue>::ret
morpho::watershed_contexttt¡class DestValue¿
(const abstract::non_vectorial_image<I>& im,
const abstract::neighborhood<N>& ng);
```

Parameters

DestValue		type of output labels
im	IN	image of levels
ng	IN	neighborhood to consider

Description

Compute the connected watershed for image *im* using neighborhood *ng*.

`watershed_con` creates an ouput image whose values have type *DestValue* (which should be discrete). In this output all basins are labeled using values from `DestValue::min()` to `DestValue::max() - 4` (the remaining values are used internally by the algorithm).

When there are more basins than `DestValue` can hold, wrapping occurs (i.e., the same label is used for several basin). This is potentially harmful, because if two connected basins are labeled with the same value they will appear as one basin.

4.1.36 morpho::watershed_seg

Purpose

Segmented Watershed.

Prototype

```
#include <oln//morpho/watershed.hh>

typename mute<I, DestValue>::ret
morpho::watershed_segtexttt¡class DestValue¿
(const abstract::non_vectorial_image<I>& im,
const abstract::neighborhood<N>& ng);
```

Parameters

DestValue		type of output labels
im	IN	image of levels
ng	IN	neighborhood to consider

Description

Compute the segmented watershed for image *im* using neighborhood *ng*.

`watershed_seg` creates an ouput image whose values have type *DestValue* (which should be discrete). In this output image, `DestValue::max()` indicates a watershed, and all basins are labeled using values from `DestValue::min()` to `DestValue::max() - 4` (the remaining values are used internally by the algorithm).

When there are more basins than `DestValue` can hold, wrapping occurs (i.e., the same label is used for several basin).

4.1.37 morpho::watershed_seg_or

Purpose

Segmented Watershed with user-supplied starting points.

Prototype

```
#include <oln/morpho/watershed.hh>

oln_concrete_type(I2)&
morpho::watershed_seg_or
(const abstract::non_vectorial_image<I1>& levels,
abstract::non_vectorial_image<I2>& markers,
const abstract::neighborhood<N>& ng);
```

Parameters

levels	IN		image of levels
markers	IN	OUT	image of markers
ng	IN		neighborhood to consider

Description

Compute a segmented watershed for image *levels* using neighborhood *ng*, and *markers* as starting point for the flooding algorithm.

markers is an image of the same size as *levels* and containing discrete values indicating label associated to each basin. On input, fill *markers* with `oln_value_type(I2)::min()` (this is the *unknown* label) and mark the starting points or regions (usually these are minima in *levels*) using a value between `oln_value_type(I2)::min()+1` and `oln_value_type(I2)::max()-1`.

`watershed_seg_or` will flood *levels* from these non-*unknown* starting points, labeling basins using the value you assigned to them, and markining watershed lines with `oln_value_type(I2)::max()`. *markers* should not contains any `oln_value_type(I2)::min()` value on output.

4.1.38 morpho::white_top_hat

Purpose

White top hat.

Prototype

```
#include <oln/morpho/top_hat.hh>

typename mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::white_top_hat
(const convert::abstract::conversion<C B>&,
```

```
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

typename mute<I, typename convoutput<C,
B,oln_value_type(I)>::ret>::ret
morpho::fast::white_top_hat
(const convert::abstract::conversion<C B>&,
const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

oln_concrete_type(I)
morpho::white_top_hat
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);

oln_concrete_type(I)
morpho::fast::white_top_hat
(const abstract::non_vectorial_image<I>& input,
const abstract::struct_elt<E>& se);
```

Parameters

B>&		IN
input	IN	input image
se	IN	structural element

Description

Compute white top hat of *input* using *se* as structural element. Soille p.105.

See also

morpho::opening, §4.1.24, p.40.

Example

```
image2d<int_u8> im = load("lena256.pgm");
save(morpho::white_top_hat(im, win_c8p()), "out.pgm");
```

Figure 4.37: lena256.pgm

Figure 4.38: out.pgm

4.2 Level processings

4.2.1 level::connected_component

Purpose

Connected Component.

Prototype

```
#include <oln/level/connected.hh>

typename mute<I, DestType>::ret
level::connected_component
(const abstract::image<I1>& marker,
const abstract::neighborhood<E>& se);
```

Parameters

marker	IN	marker image
se	IN	structural element

Description

It removes the small (in area) connected components of the upper level sets of *input* using *se* as structural element. The implementation comes from *Cocquerez et Philipp, Analyse d'images, filtrages et segmentations* p.62.

See also

level::frontp_connected_component, §4.2.2, p.53.

Example

```
image2d<int_u8> light = load("light.pgm");
save(level::connected_component<int_u8>(light, win_c8p()), "out.pgm");
```

4.2.2 level::frontp_connected_component

Purpose

Connected Component.

Prototype

```
#include <oln/level/cc.hh>

typename mute<I, DestType>::ret
level::frontp_connected_component
(const abstract::image<I>& marker,
const abstract::neighborhood<E>& se, numeric_value& nb);
```

Parameters

marker	IN	marker image
se	IN	neighbourhood
nb	IN	nb_label (optional)

Description

It removes the small (in area) connected components of the upper level sets of *input* using *se* as structural element. The implementation uses front propagation.

See also

level::connected_component, §4.2.1, p.53.

Example

```
image2d<int_u8> light = load("light.pgm");
save(level::frontp_connected_component<int_u16>(light, win_c8p()),
"out.pgm");
```