Olena Reference Manual

EPITA Research and Development Laboratory

February 12, 2003

This manual documents Olena, a generic image processing library.

Chapter 1

Introduction

This reference manual will eventually document any public class and functions available in Olena. Sadly, it only covers 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 documents the core of the Olena library.

2.1 Basic types

This section describes the basic types available in Olena. By *basic types* we roughly mean any type that can be used as pixel value in an image, such as `int`, `float`, etc. This also includes composite types like complexes, points, vectors, and colors.

For completeness we will discuss matrices too, although these are not deemed basic they are relevant to the description of vectors.

Other types not discussed here, but worth to mention, includes images and iterators. These will be described in a separate section.

2.1.1 Hierarchy of types

Here is a list of basic types supported by Olena. Although it is presented hiarchically, this does not means that these types are implemented hierarchically in the C++ code.

- Scalar types
  - Integers
    - Signed
      - C built-in types:
        - `signed long`
        - `signed int`
        - `signed short`
        - `signed char`
      - Olena types: `int_s<BITS, BEHAVIOR>`
    - Unsigned

        - C built-in types
            - `unsigned long`
            - `unsigned int`
            - `unsigned short`
            - `unsigned char`
        - Olena types: `int_u<BITS, BEHAVIOR>`
    - Floats
        - C built-in types
            - `float`
            - `double`
        - Olena types
            - `sfloat`
            - `dfloat`
- Enumerated types
    - C built-in `bool`
    - `bin`
    - `label<TYPE>`
- Vectorial types
    - FIXME: complex
        - $a + ib$
        - $r \exp i\theta$
    - points
        - `point1d`
        - `point2d`
        - `point3d`
    - `vec<N, T>`
    - FIXME: colors
- FIXME: Matrices

2.1.2 Scalar types

All types listed under *scalar types* in the above enumeration, can be used together with most common arithmetic operations ($+$, $=$, $<$, ...). However Olena's type do not behave exactly like the the built-in types. We believe it's safer To use Olena's type instead of the built-in types.

2.1.3 int_u<N, BEHAVIOR=abort>

This template can be used to define unsigned integers of various size and behaviors.

`N` is the number of bits used to store the integer. So `int_u<7>` can hold values from the range 0..127.

`BEHAVIOR` must be one of `strict`, `saturate`, or `unsafe`, this describe how the integer will behave on overflows (and underflows).

> This is the default. An overflow triggers an assertion which halts the program.
>
> Checking for overflow incurs some run-time overhead, but this is dissabled entirely when compiling with `-DNDEBUG` once the program is debugged. (`strict` and `unsafe` are identical when `-DNDEBUG` is used.)

> These are bounded types, the bounds of which cannot be escaped. Assigning a value larger (resp. smaller) than the maximum (resp. minimum) of the type, is like assigning this maximum (resp. minimum).

```
int_u<7,saturate> i = 130; // i equals 127
i += 10;                   // i still equals 127
i -= 10;                   // i equals 117
```

> This behaviour is useful for tasks like histogram construction where a value is always increased but should never overflow.
>
> Using the `saturate` behavior always incurs some overhead. (The same overhead you would get by checking the overflow manually.)

> This is the behavior of the underlying built-in type: no overflow checking. (This is mainly used internally, to declare the temporary variables used by all the other behaviors.)

Note that overflow checking matters only with operations that *modify* the current variable (assignments, construction, self operators like `+=`, `-=`, `*=`. etc.), not with operation returning new variables (like `+`, `*`, etc.).

If possible, operators that return new variables will extend the type of the operand. For instance when you add two `int_u<8>` variables, the `+` operator returns an `int_u<9>` temporary variable. This ensures that there is no overflow. If you then assign this temporary variable to a `int_u<8>` variable, overflow checking occurs.

```
int_u<8> i = 200;
int_u<8> j = 100;
int_u<9> k = i + j;  // ok, no overflow
int_u<8> l = i + j;  // assertion triggered
                     // in the constructor of l
```

In former versions of Olena, we used to disallow any assignment from `int_u<N>` to `int_u<M>` where `N>M`. The `l = i + j;` line in the previous example would simply not compile.
This restriction has been removed because it turned out to be very annoying. The programmer had to write `l = cast::force< int_u<8> >(i + j);` or `l = cast::bound< int_u<8> >(i + j);` whenever he knew there would be no overflow. One other work around was to bypass the temporary variable creation with `l = i; l += j;`.
There are two common places where this rule was found really annoying. One is the definition of constants.

```
const T init    = cast::force<T>(optraits<T>::min() + 1);
const T inqueue = cast::force<T>(optraits<T>::min() + 2);
```

The second is the computation of an average. Althought the sum is made in a larger type, we *know* that the final result will fit in the orgiginal type.

```
template <class T>
T average(const list<T>& l)
{
   optraits<T>::larger_type sum = optraits<T>::larger_type::zero();
   for (list<T>::iterator i = l.begin(); i != l.end(); ++i)
      sum += *i;
   return cast::force<T>(sun / l.size());
}
```

Saying that overflow checking occurs only during assignments because overflow during operations is impossible (since the type is extended as appropriate), would not the exact truth.

The type cannot always be extended during operations. Summing two `int_u<31>` variables will return a `int_u<32>` temporary variable, extending the type; however summing two `int_u<32>` variables will yield another `int_u<32>` variable (on 32bits hosts) because no larger type is available.

Because of this, overflow checking has to occur during operations too. This means that the behavior of the temporary variables is imporant. The choice is based on the behavior of the operands.

operand	operand	result
strict	strict	strict
saturate	saturate	saturate
unsafe	unsafe	unsafe
strict	unsafe	strict
strict	saturate	strict
unsafe	saturate	saturate

Another way to present this is to say that behaviors have been ordered as follow `unsafe < saturate < strict`, and that when there is a choice between two behaviors, we always choose the greater.

Please bear in mind that the behavior chosen for temporary variable only rarely matters because the type is usually extended so that overflow do not occur.

The following frequently used types are given shortands.

```
typedef int_u<8,  strict>       int_u8;
typedef int_u<16, strict>       int_u16;
typedef int_u<32, strict>       int_u32;
typedef int_u<8,  saturate>     int_u8s;
typedef int_u<16, saturate>     int_u16s;
typedef int_u<32, saturate>     int_u32s;
```

2.1.4 `int_s<N, BEHAVIOR=strict>`

This template can be used to define signed integers of various size and behaviors. This is can be used exactly like `int_u<N,BEHAVIOR>`.

Extending the type will also works during operations between unsigned and signed integers.

The following `typedef`s are defined.

```
typedef int_s<8,  strict>       int_s8;
typedef int_s<16, strict>       int_s16;
typedef int_s<32, strict>       int_s32;
typedef int_s<8,  saturate>     int_s8s;
typedef int_s<16, saturate>     int_s16s;
typedef int_s<32, saturate>     int_s32s;
```

2.1.5 `sfloat` and `dfloat`

These are the Olena equivalents of `float` and `double`, Contrary to signed and unsigned integer types, there is not *extension* during operations. Multiplying two `sfloat` yields a `sfloat`.

FIXME: so why do we use `float` and `double`?

2.1.6 Interactions between scalar types

The following unary operations are defined for any scalar type, with their common semantic: `+x`, `-x`.

These binary operations can be used with all combinasion of (Olena) scalar type, with their common semantic: `x+y`, `x-y`, `x*y`, `x/y`, `x+=y`, `x-=y`, `x*=y`, `x/=y`, `x == y`, `x < y`, `x > y`, `x <= y`, `x >= y`, `x != y`.

Especially, this means that `int_u32` is comparable to `int_s32`. For instance in

```
int_u32 i = ...;
int_s32 j = ...;
if (i < j)
```

```
// ...
```

the code generated for `i<j` should be close to `j >= 0 && i < cast::force<int_u32>(j)`.

2.1.7 Decorators for scalar types

You can decorate a type to add some constraint or change it's behavior.

`range<T, INTERVAL, BEHAVIOR=strict>`

This class, like Ada's `range` keyword, adds a constrains on type `T`. The constraint is that values should be bounded by the interval `INTERVAL`. `BEHAVIOR` specifies how overflow checking should behave.

Interval should be expressed using the `bounded_s<MIN, MAX>` and `bounded_u<MIN, MAX>` templates, where `MIN` and `MAX` are signed integer in `bounded_s` and unsigned integer in `bounded_u<MIN, MAX>`.[1]

`range<int_u<8>, bounded_u<10, 25> >` is thus an unsigned integer, stored on 8 bits, but restricted to the [10...25] interval. Because `strict` is the default behavior, any attempt to escape this interval will abort the program.

`cycle<T, INTERVAL>`

This class defines a type, the values of which can be built from `T`, and that cycle over `INTERVAL`, where `INTERVAL` is defined using `bounded_u<MIN, MAX>` or `bounded_s<MIN,MAX>` as in `range`.

For instance `typedef cycle<sfloat, bounded_u<0, 360> > deg` defines a type which is useful to represent angles in degrees. `deg d = 400.0` will set `d` to `40.0`.

For the purpose of cycling, the upper bound of the range, `MAX`, is excluded. `deg d = 360.0` will set `d` to `0.0`.

Any value implicitely convertible to `T` can be involved in an arithmetic operation with `cycle<T, INTERVAL>`.

Any conversion from `cycle<T, bounded_x<A, B> >` to `cycle<T, bounded_y<C, D> >` must be explicit and should respect $B - A = D - C$ (i.e., the two cycles have equal length).

2.2 Enumerated types

2.2.1 `bin`

`bin` is a binary type, with two values: `0` and `1`. This is unlike the built-in `bool`, whose values are `false` and `true`.

[1] `MIN` and `MAX` are always integers (the C++ language does not allow floats as template arguments), so to define a `sfloat` between 0 and 1, use `range<sfloat,bounded_u<0,1> >`, not `range<sfloat,bounded_u<0.0,1.0> >`.

2.2.2 label<T, BEHAVIOR=cycle>

FIXME: not sure if we really need this.

FIXME: This documentation is out of date. We've decided to split label into ordered labels and unordered labels.

label<T, BEHAVIOR=cycle> is used to label images.

Although it is not stricly correct to define an order on labels, we do. This is useful (FIXME: Ask Jerome for an example).

2.2.3 The typetraits<T> trait

For each type T described in the previous section, Olena records additional relations in the typetraits<T> trait.

All the following members of typetraits<T> are typedefs.

The base type for T, i.e., T without any decoration. For instance typetraits< range<int_u8, bounded_u<10,2 is int_u8.

Ada programmers will probably recognize the T'Base attribute.

For a undecorated type T, typetraits<T>::base_type is T itself.

The C type used internally for storing the value. E.g., typetraits< int_u<9> >::storage_type is unsigned short.

The smallest signed type that can countain the values of T, if possible. For instance typetraits< int_u8 >::signed_type is int_s<10>. (int_s<9> is not large enough to hold the value 256.)

However typetraits< int_u32 >::signed_type is int_s<32>.

The smalled unsigned type that can countain the non negative values of T. For instance, typetraits< int_s<9> >::unsigned_type is int_u8.

A type that can be used to sum up a moderate number of values of type T. (FIXME: a moderate number? Come on... I'm sure we can put it in a way which is even more fuzzy.) For instance typetraits< int_u8 >::cumul_type is int_u16.

The largest type in the familly of T. For instance typetraits< int_u8 >::largest_type is int_u32.

These are short-hands for some combinations of signed_type, unsigned_type, cumul_type and largest_type.

An int type of the same sign as T. For instance typetraits< int_u8 >::interger_type is unsigned int; typetraits< sfloat >::interger_type is signed int.

2.2.4 The `optraits<T>` trait

Olena offers some additionnal supporting function or operators in the `optraits<T>` trait. For example, sometimes while developping a generic algorithm on type `T` we need to need the maximum value that this type can take. The `optraits<T>` trait provides support to anser this sort of question. For instance `optraits<T>::max()` returns the maximum values for type `T`.

> The reason why we have to use a separate class and write `optraits<T>::max()` instead of `T::max()` is that the latter case cannot work with built-in types.
> Besides, using a class distinct from the type itself to hold operators and various support functions allows to reuse the support from another type by inheritance on the `optraits` hierarchy, without implying any inheritance on the type hierarchy.
> For instance `optraits<range<int_u8, 10, 25> >` inherits from `optraits<int_u8>` and simply redefine a few members like the `max()` and `min()` functions. However `range<int_u8, 10, 25>` is not a subclass of `int_u8`.

Here are the public `optraits<T>` members:

The minimum *reachable* value for type `T`, if such minimum value exists (this is not the case for float value, for instance). `optraits<T>::min()` is the equivalent of `T'First` in Ada.

The maximum *reachable* value for type `T`, if such maximum value exists (this is not the case for float value, for instance). `optraits<T>::max()` is the equivalent of `T'Last` in Ada.

The lower bound for type `T`. For discrete types this is the same as `min()`, for other types such as floats this can be $-\infty$.

The upper bound for type `T`. For discrete types this is the same as `max()`, for other types such as floats this can be $+\infty$.

The unit value for type `T`, when it makes sense (it doesn't make any sense for color types such as `rgb` for instance).

The unit value for type `T`, when it makes sense.

The default value for type `T`. This is the value that a default-constructed `T` variable should have. This should be the same as `zero()` whenever possible.

`optraits<T>` also contains a number of internal members, used during operations.

FIXME: List them.

Chapter 3

Processings

3.1 Morphological processings

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

3.1.1 morpho::beucher_gradient

[illegible]
Morphological Beucher Gradient.

Prototype

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

mute<I, typename convoutput<C, Value(I)>::ret>::ret
morpho::beucher_gradient (const conversion<C>& c,
const image<I>& input, const struct_elt<E>& se);

mute<I, typename convoutput<C, Value(I)>::ret>::ret
morpho::fast::beucher_gradient (const conversion<C>& c,
const image<I>& input, const struct_elt<E>& se);

Concrete(I) morpho::beucher_gradient
(const image<I>& input, const struct_elt<E>& se);

Concrete(I) morpho::fast::beucher_gradient
(const image<I>& input, const struct_elt<E>& se);
```

Parameters

c	IN	conversion object
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, §3.1.5, p.17,
morpho::dilation, §3.1.4, p.16,
morpho::external_gradient, §3.1.6, p.18,
morpho::internal_gradient, §3.1.18, p.29.

Example

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

Figure 3.1: lena256.pgm

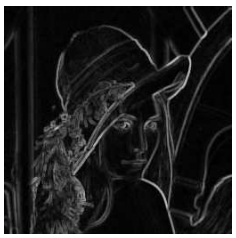

Figure 3.2: out.pgm

3.1.2 morpho::black_top_hat

Purpose

Black top hat.

Prototype

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

typename mute<I, typename convoutput<C,
Value(I)>::ret>::ret
morpho::black_top_hat (const conversion<C>& c,
const image<I>& input, const struct_elt<E>& se);

typename mute<I, typename convoutput<C,
Value(I)>::ret>::ret
morpho::fast::black_top_hat (const conversion<C>& c,
const image<I>& input, const struct_elt<E>& se);
```

Parameters

c	IN	conversion object
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, §3.1.3, p.15.

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

Figure 3.3: lena256.pgm

Figure 3.4: out.pgm

3.1.3 morpho::closing

Purpose
Morphological closing.

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

Concrete(I) morpho::closing (const image<I>& input,
const struct_elt<E>& se);

Concrete(I) morpho::fast::closing (const image<I>& input,
const 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, §3.1.5, p.17,
morpho::dilation, §3.1.4, p.16,
morpho::closing, §3.1.3, p.15.

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

Figure 3.5: object.pbm

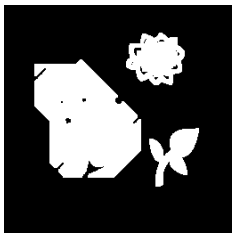

Figure 3.6: out.pbm

3.1.4 morpho::dilation

Purpose

Morphological dilation.

Prototype

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

Concrete(I) morpho::dilation (const image<I>& input ,
const struct_elt<E>& se );

Concrete(I) morpho::fast::dilation (const image<I>& input ,
const 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, §3.1.20, p.31,
morpho::erosion, §3.1.5, p.17.

Example

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

Figure 3.7: object.pbm

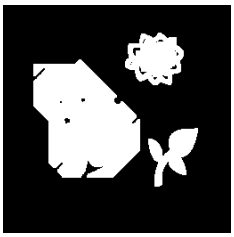

Figure 3.8: out.pbm

3.1.5 morpho::erosion

Purpose

Morphological erosion.

Prototype

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

Concrete(I) morpho::erosion (const image<I>& input,
const struct_elt<E>& se);

Concrete(I) morpho::fast::erosion (const image<I>& input,
const 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, §3.1.21, p.31,
morpho::dilation, §3.1.4, p.16.

Example

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

Figure 3.9: object.pbm

Figure 3.10: out.pbm

3.1.6 morpho::external_gradient

Purpose

Morphological External Gradient.

Prototype

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

mute<I, typename convoutput<C, Value(I)>::ret>::ret
morpho::external_gradient (const conversion<C>& c,
const image<I>& input, const struct_elt<E>& se);

mute<I, typename convoutput<C, Value(I)>::ret>::ret
morpho::fast::external_gradient (const conversion<C>& c,
const image<I>& input, const struct_elt<E>& se);

Concrete(I) morpho::external_gradient
(const image<I>& input, const struct_elt<E>& se);

Concrete(I) morpho::fast::external_gradient
(const image<I>& input, const struct_elt<E>& se);
```

Parameters

c	IN	conversion object
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, §3.1.1, p.13,
morpho::internal_gradient, §3.1.18, p.29,
morpho::dilation, §3.1.4, p.16.

Example

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

Figure 3.11: lena256.pgm

Figure 3.12: out.pgm

3.1.7 morpho::fast_maxima_killer

Purpose

Maxima killer.

Prototype

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

Concrete(I1_) morpho::fast_maxima_killer
(const image<I1>& marker, const unsigned int area area,
const 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, §3.1.30, p.38.

Example

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

3.1.8 morpho::fast_minima_killer

Purpose

Minima killer.

Prototype

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

```
Concrete(I1_) morpho::fast_minima_killer
(const image<I1>& marker, const unsigned int area area,
const 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, §3.1.31, p.39.

Example

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

3.1.9 morpho::geodesic_dilation

Purpose

Geodesic dilation.

Prototype

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

Concrete(I1) morpho::geodesic_dilation
(const image<I1>& marker, const image<I2>& mask,
const 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, §3.1.26, p.35.

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");
```

3.1.10 morpho::geodesic_erosion

Purpose

Geodesic erosion.

Prototype

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

Concrete(I1) morpho::geodesic_erosion
(const image<I1>& marker, const image<I2>& mask,
const 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, §3.1.26, p.35.

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");
```

3.1.11 morpho::hit_or_miss

Purpose

Hit_or_Miss Transform.

Prototype

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

typename mute<I_, typename convoutput<C,
Value(I_)>::ret>::ret
```

morpho::hit_or_miss (const conversion<C>& *c*,
const image<I>& *input*, const struct_elt<E>& *se1*,
const struct_elt<E>& *se2*);

typename mute<I_, typename convoutput<C,
Value(I_)>::ret>::ret
morpho::fast::hit_or_miss (const conversion<C>& *c*,
const image<I>& *input*, const struct_elt<E>& *se1*,
const struct_elt<E>& *se2*);

Concrete(I) **morpho::hit_or_miss** (const image<I>& *input*,
const struct_elt<E>& *se1*, const struct_elt<E>& *se2*);

Concrete(I) **morpho::fast::hit_or_miss** (const image<I>& *input*,
const struct_elt<E>& *se1*, const struct_elt<E>& *se2*);

Parameters

c	IN	conversion object
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<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");
```

3.1.12 morpho::hit_or_miss_closing

Purpose

Hit_or_Miss closing.

Figure 3.13: object.pbm

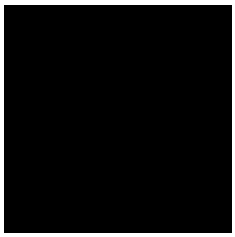

Figure 3.14: out.pgm

Prototype

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

Concrete(I) morpho::hit_or_miss_closing
(const image<I>& input, const struct_elt<E>& se1,
const struct_elt<E>& se2);

Concrete(I) morpho::fast::hit_or_miss_closing
(const image<I>& input, const struct_elt<E>& se1,
const 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, §3.1.11, p.21,
morpho::hit_or_miss_closing_bg, §3.1.13, p.24,
morpho::hit_or_miss_opening, §3.1.14, p.25,
morpho::hit_or_miss_opening_bg, §3.1.15, p.26.

Example

```
 image2d<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 3.15: object.pbm

Figure 3.16: out.pbm

3.1.13 morpho::hit_or_miss_closing_bg

Purpose

Hit_or_Miss closing of background.

Prototype

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

Concrete(I) morpho::hit_or_miss_closing_bg
(const image<I>& input, const struct_elt<E>& se1,
const struct_elt<E>& se2);

Concrete(I) morpho::fast::hit_or_miss_closing_bg
(const image<I>& input, const struct_elt<E>& se1,
const 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, §3.1.11, p.21,
morpho::hit_or_miss_closing, §3.1.12, p.22,
morpho::hit_or_miss_opening, §3.1.14, p.25,
morpho::hit_or_miss_opening_bg, §3.1.15, p.26.

Example

```
 image2d<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 3.17: object.pbm

Figure 3.18: out.pbm

3.1.14 morpho::hit_or_miss_opening

Purpose

Hit_or_Miss opening.

Prototype

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

Concrete(I) morpho::hit_or_miss_opening
(const image<I>& input, const struct_elt<E>& se1,
const struct_elt<E>& se2);

Concrete(I) morpho::fast::hit_or_miss_opening
(const image<I>& input, const struct_elt<E>& se1,
const 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, §3.1.11, p.21,
morpho::hit_or_miss_closing, §3.1.12, p.22,
morpho::hit_or_miss_closing_bg, §3.1.13, p.24,
morpho::hit_or_miss_opening_bg, §3.1.15, p.26.

Example

```
 image2d<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");
```

Figure 3.19: object.pbm

Figure 3.20: out.pbm

3.1.15 morpho::hit_or_miss_opening_bg

Purpose

Hit_or_Miss opening of background.

Prototype

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

Concrete(I) morpho::hit_or_miss_opening_bg
(const image<I>& input, const struct_elt<E>& se1,
const struct_elt<E>& se2);
```

```
Concrete(I) morpho::fast::hit_or_miss_opening_bg
(const image<I>& input, const struct_elt<E>& se1,
const 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, §3.1.11, p.21,
morpho::hit_or_miss_closing, §3.1.12, p.22,
morpho::hit_or_miss_closing_bg, §3.1.13, p.24,
morpho::hit_or_miss_opening, §3.1.14, p.25.

Example

```
 image2d<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 3.21: object.pbm

Figure 3.22: out.pbm

3.1.16 morpho::hybrid_geodesic_reconstruction_dilation

Purpose

Geodesic reconstruction by dilation.

Prototype

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

Concrete(I1) morpho::hybrid_geodesic_reconstruction_dilation
(const image<I1>& marker, const image<I2>& mask,
const 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, §3.1.26, p.35.

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
```

3.1.17 morpho::hybrid_geodesic_reconstruction_erosion

Purpose

Geodesic reconstruction by erosion.

Prototype

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

Concrete(I1) morpho::hybrid_geodesic_reconstruction_erosion
(const image<I1>& marker, const image<I2>& mask,
const 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, §3.1.27, p.36.

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()), "out.pgm")
```

3.1.18 morpho::internal_gradient

Purpose

Morphological Internal Gradient.

Prototype

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

mute<I, typename convoutput<C, Value(I)>::ret>::ret
morpho::internal_gradient (const conversion<C>& c,
const image<I>& input, const struct_elt<E>& se);

mute<I, typename convoutput<C, Value(I)>::ret>::ret
morpho::fast::internal_gradient (const conversion<C>& c,
const image<I>& input, const struct_elt<E>& se);

Concrete(I) morpho::internal_gradient
(const image<I>& input, const struct_elt<E>& se);

Concrete(I) morpho::fast::internal_gradient
(const image<I>& input, const struct_elt<E>& se);
```

Parameters

c	IN	conversion object
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, §3.1.1, p.13,
morpho::external_gradient, §3.1.6, p.18,
morpho::erosion, §3.1.5, p.17.

Example

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

Figure 3.23: lena256.pgm

Figure 3.24: out.pgm

3.1.19 morpho::laplacian

Purpose

Laplacian.

Prototype

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

typename mute<I, typename convoutput<C,
Value(I)>::ret>::ret
morpho::laplacian (const conversion<C>& c,
const image<I>& input, const struct_elt<E>& se);

typename mute<I, typename convoutput<C,
Value(I)>::ret>::ret
morpho::fast::laplacian (const conversion<C>& c,
const image<I>& input, const struct_elt<E>& se);

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

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

Parameters

c	IN	conversion object
input	IN	input image
se	IN	structural element

Description

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

See also

morpho::dilation, §3.1.4, p.16,
morpho::erosion, §3.1.5, p.17.

Example

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

Figure 3.25: lena256.pgm

Figure 3.26: out.pgm

3.1.20 morpho::n_dilation

Purpose

Morphological dilation itered n times.

Prototype

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

Concrete(I) morpho::n_dilation (const image<I>& input,
const 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, §3.1.4, p.16,
morpho::n_erosion, §3.1.21, p.31.

3.1.21 morpho::n_erosion

Purpose

Morphological erosion itered n times.

Prototype

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

Concrete(I) morpho::n_erosion (const image<I>& input,
const 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, §3.1.5, p.17,
morpho::n_dilation, §3.1.20, p.31.

3.1.22 morpho::opening

Purpose

Morphological opening.

Prototype

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

Concrete(I) morpho::opening (const image<I>& input,
const struct_elt<E>& se);

Concrete(I) morpho::fast::opening (const image<I>& input,
const 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, §3.1.5, p.17,
morpho::dilation, §3.1.4, p.16,
morpho::closing, §3.1.3, p.15.

Example

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

Figure 3.27: object.pbm

Figure 3.28: out.pbm

3.1.23 morpho::self_complementary_top_hat

Purpose

Self complementary top hat.

Prototype

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

typename mute<I, typename convoutput<C,
Value(I)>::ret>::ret
morpho::self_complementary_top_hat (const conversion<C>& c,
const image<I>& input, const struct_elt<E>& se);

typename mute<I, typename convoutput<C,
Value(I)>::ret>::ret
morpho::fast::self_complementary_top_hat
(const conversion<C>& c, const image<I>& input,
const struct_elt<E>& se);

typename mute<I, typename convoutput<C,
Value(I)>::ret>::ret
morpho::self_complementary_top_hat (const image<I>& input,
const struct_elt<E>& se);

typename mute<I, typename convoutput<C,
Value(I)>::ret>::ret
morpho::fast::self_complementary_top_hat
(const image<I>& input, const struct_elt<E>& se);
```

Parameters

c	IN	conversion object
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, §3.1.3, p.15,
morpho::opening, §3.1.22, p.32.

Example

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

Figure 3.29: lena256.pgm

Figure 3.30: out.pgm

3.1.24 morpho::sequential_geodesic_reconstruction_dilation

Purpose

Geodesic reconstruction by dilation.

Prototype

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

Concrete(I1)
morpho::sequential_geodesic_reconstruction_dilation
(const image<I1>& marker, const image<I2>& mask,
const 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, §3.1.26, p.35.

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"
```

3.1.25 morpho::sequential_geodesic_reconstruction_erosion

Purpose

Geodesic reconstruction by erosion.

Prototype

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

Concrete(I1)
morpho::sequential_geodesic_reconstruction_erosion
(const image<I1>& marker, const image<I2>& mask,
const 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, §3.1.27, p.36.

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()), "out.pgm")
```

3.1.26 morpho::simple_geodesic_dilation

Purpose

Geodesic dilation.

Prototype

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

```
Concrete(I1) morpho::simple_geodesic_dilation
(const image<I1>& marker, const image<I2>& mask,
const 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");
```

3.1.27 morpho::simple_geodesic_erosion

Purpose

Geodesic erosion.

Prototype

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

Concrete(I1) morpho::simple_geodesic_erosion
(const image<I1>& marker, const image<I2>& mask,
const 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");
```

3.1.28 morpho::sure_geodesic_reconstruction_dilation

Purpose
Geodesic reconstruction by dilation.

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

Concrete(I1) morpho::sure_geodesic_reconstruction_dilation
(const image<I1>& marker, const image<I2>& mask,
const 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, §3.1.26, p.35.

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.pgm");
```

3.1.29 morpho::sure_geodesic_reconstruction_erosion

Purpose
Geodesic reconstruction by erosion.

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

```
Concrete(I1) morpho::sure_geodesic_reconstruction_erosion
(const image<I1>& marker, const image<I2>& mask,
const 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, §3.1.27, p.36.

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.p
```

3.1.30 morpho::sure_maxima_killer

Purpose

Maxima killer.

Prototype

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

Concrete(I1) morpho::sure_maxima_killer
(const image<I1>& marker, const unsigned int area area,
const 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, §3.1.7, p.19.

Example

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

3.1.31 morpho::sure_minima_killer

Purpose

Minima killer.

Prototype

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

Concrete(I1) morpho::sure_minima_killer
(const image<I1>& marker, const unsigned int area area,
const 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, §3.1.7, p.19.

Example

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

3.1.32 morpho::top_hat_contrast_op

Purpose

Top hat contrastor operator.

Prototype

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

typename mute<I, typename convoutput<C,
Value(I)>::ret>::ret
morpho::top_hat_contrast_op (const conversion<C>& c,
const image<I>& input, const struct_elt<E>& se);
```

```
typename mute<I, typename convoutput<C,
Value(I)>::ret>::ret
morpho::fast::top_hat_contrast_op (const conversion<C>& c,
const image<I>& input, const struct_elt<E>& se);

typename mute<I, typename convoutput<C,
Value(I)>::ret>::ret
morpho::top_hat_contrast_op (const image<I>& input,
const struct_elt<E>& se);

typename mute<I, typename convoutput<C,
Value(I)>::ret>::ret
morpho::fast::top_hat_contrast_op (const image<I>& input,
const struct_elt<E>& se);
```

Parameters

c	IN	conversion object
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, §3.1.36, p.42,
morpho::black_top_hat, §3.1.2, p.14.

Example

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

Figure 3.31: lena256.pgm

Figure 3.32: out.pgm

3.1.33 morpho::watershed_con

Purpose

Connected Watershed.

Prototype

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

typename mute<I, DestValue>::ret
morpho::watershed_contexttt¡class DestValue¿
(const image<I>& im, const 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.

3.1.34 morpho::watershed_seg

Purpose

Segmented Watershed.

Prototype

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

typename mute<I, DestValue>::ret
morpho::watershed_segtexttt¡class DestValue¿
(const image<I>& im, const 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).

3.1.35 morpho::watershed_seg_or

Purpose

Segmented Watershed with user-supplied starting points.

Prototype

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

```
Concrete(I2)& morpho::watershed_seg_or
(const image<I1>& levels, image<I2>& markers,
const 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 `Value(I2)::min()` (this is the *unknown* label) and mark the starting points or regions (usually these are minima in *levels*) using a value between `Value(I2)::min()+1` and `Value(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 `Value(I2)::max()`. *markers* should not contains any `Value(I2)::min()` value on output.

3.1.36 morpho::white_top_hat

Purpose

White top hat.

Prototype

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

```
typename mute<I, typename convoutput<C,
Value(I)>::ret>::ret
morpho::white_top_hat (const conversion<C>& c,
const image<I>& input, const struct_elt<E>& se);
typename mute<I, typename convoutput<C,
Value(I)>::ret>::ret
morpho::fast::white_top_hat (const conversion<C>& c,
const image<I>& input, const struct_elt<E>& se);
Concrete(I) morpho::white_top_hat (const image<I>& input,
const struct_elt<E>& se);
Concrete(I) morpho::fast::white_top_hat
(const image<I>& input, const struct_elt<E>& se);
```

Parameters

c	IN	conversion object
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, §3.1.22, p.32.

Example

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

Figure 3.33: lena256.pgm

Figure 3.34: out.pgm

3.2 Level processings

3.2.1 level::connected_component

Purpose

Connected Component.

Prototype

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

typename mute<I_, DestType>::ret
level::connected_component (const image<I1>& marker,
const struct_elt<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, §3.2.2, p.44.

Example

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

3.2.2 level::frontp_connected_component

Purpose

Connected Component.

Prototype

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

typename mute<I, DestType>::ret
level::frontp_connected_component (const image<I1>& marker,
const 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, §3.2.1, p.43.

Example

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