Olena – Quick Reference Guide

LRDE

Copyright

Contents

Chapter 1

Installation

1.1 Requirements

1.1.1 To compile the user examples

- a POSIX shell, like Bash

- a decent C++ compiler, like GNU C++

- a 'make' utility, like GNU or BSD 'make'

1.1.2 To compile the documentation (Optional)

- a LaTeX distribution

- the 'listings' TeX package

- the utility 'convert' from ImageMagick

- GNU Autogen

- 'hevea', a TeX to HTML conversion tool

- the 'texinfo' utilities from GNU

1.1.3 To develop in Olena

- GNU Autotools (Autoconf 2.54, Automake 1.10)

1.2 Getting Olena

The latest version of Olena is available at the following addresses:

- http://www.lrde.epita.fr/dload/olena/olena.tar.gz

- http://www.lrde.epita.fr/dload/olena/olena.tar.bz2

1.3 Building Olena

First uncompress the archive. According to the downloaded archive, the options are different.

```
$ tar zxvf olena.tar.gz
```

Or

```
$ tar jxvf olena.tar.bz2
```

Then follow these steps:

```
$ mkdir olena-build
$ cd olena-build
$ ../olena-1.0/configure && make
$ sudo make install
```

Chapter 2

Foreword

2.1 Generality

The following tutorial explains the basic concepts behind Olena and how to use the most common objects and routines. This tutorial includes many code examples and figures. In order to make this tutorial easier to understand, we will mainly talk about 2D images. This is not so restrictive since 1D, 3D, nD images behave the same way.

Since examples are based on 2D images pixels are actually "points" however we will call them "sites" which is the most generic name.

Here is also a list of common variable name conventions:

Object	Variable name
Site	p
Value	v
Neighboor	n
A site close to another site p	q

Olena is organized in a namespace hierarchy. Everything is declared by Olena within the 'mln::' namespace, and possibly a sub-namespace such as '*mln::arith::*' (arithmetic operations on images), '*mln::morpho::*' (morphological operations), etc. Usually, the namespace hierarchy is mapped to the mln directory tree. For the sake of simplicity, we will neglect the '*mln::*' prefix in all the code examples.

Methods provided by objects in the library are in constant time. If you need a specific method but you cannot find it, you may find an algorithm which can compute the information you need.

2.2 Directory hierarchy

Olena's tarball is structured as follow:

- milena

  - doc
    * benchmark: set of benchmark.
    * examples: more examples.
    * oldies: partialy not updated documentation. Not recommended for new users.
    * technical: technical doc.
    * tutorial: code sample and tutorial.
  - img: a set of sample images.
  - mesh: a full example which uses Olena.
  - mln: the library. Contains only headers.
  - tests: test suite.

- swilena: Python bindings for Olena.

2.3 Writing and compiling a program with Olena

Before writing your first program, please be aware of these hints:

- By default, Olena enables a lot of internal pre and post conditions. Usually, this is a useful feature and it **should** be enabled. However, it can heavily slow down a program though so these tests can be disabled by compiling using *-DNDEBUG*.

```
$ g++ -DNDEBUG -Ipath/to/mln my_program.cc
```

- If you decide to use optimization flags to compile for debugging, prefer using *-O1*. It is much faster to compile and it gives good performance results.

Chapter 3

Site

Usually, when talking about images, we think about common images composed of a set of pixels. Since Olena is generic, we want to support many kind of images, even images which are not composed of a set of points, such as images having images as sites.

In order to express this genericity, we have the "site" concept. This concept allows us to divide a pixel into two information:

- The pixel location, e.g. its coordinates (the site itself).

- The value.

Let's say we have a 2D grid like this:
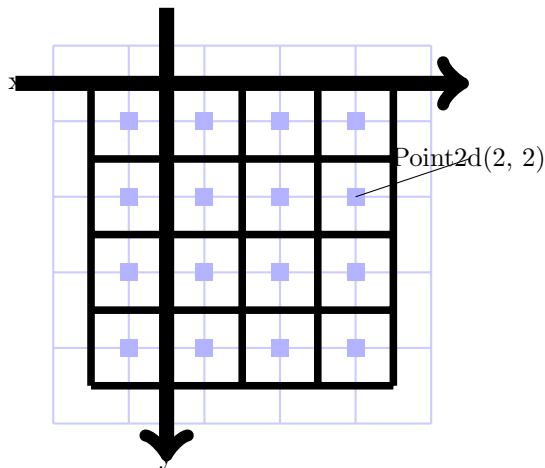
Point2d(2, 2)

On such a regular grid, in 2D, we usually use a 2D point as a site which means we have the following equivalence:

$$\text{Intersection} \equiv \text{point2d (2D site)} \equiv \text{center of a pixel}$$

The site does not store any value but refers to an area where we will be able to read its value.

Sites may have a different types, depending on the image type:

Name	Description
point2d	2D point on a regular grid
point	Generic point (nD) on a regular grid
algebra::vec	Algebraic vector
util::vertex	Graph vertex
util::edge	Graph edge

Chapter 4

Site set

Site sets are used:

1. To define an image definition domain.

2. As Site container.

They do not actually store any image value. They only store site information. Here is a list of all the site set concepts which can be found in core/site_set:

Site set	Description
p_array	site array.
p_box	compact domain defined on a regular grid (in 2D, a rectangle).
p_if	site set verifying a predicate.
p_queue	site queue.
p_run	site range.
p_runs	site range set.
p_set	mathematical site set.
p_vaccess	site set ordered by value.
p_edges	set of graph edges associated to sites.
p_vertices	set of graph vertices associated to sites.

All site sets are iterable. More detailed explanations are available in section 8.

4.1   Basic interface

Common basic interface:

Return Type	Name	Arguments	Const	Comments
bool	is_valid	-	X	Returns true if it has been initialized. The default constructor does not initialize it.
bool	has	const P& p	X	

4.2 Optional interface

Site sets may have other methods depending on their type:

Return Type	Name	Arguments	Const	Comments
size_t	nsites	-	-	Return the number of sites.
const Box&	bbox	-	X	Bounding box. Available only on grid site sets.

The previous methods are available depending on the site set. A box will have the bbox() method since it can be retrived in constant time: a box is it's own bounding box.

```
box2d b(2,3);

// The bbox can be retrived in constant time.
std::cout << b.bbox() << std::endl;

// nsites can be retrieved in constant time.
std::cout << "nsites = " << b.nsites() << std::endl;
```

Output:

```
[(0,0)..(1,2)]
nsites = 6
```

A *p_array* does not have the bbox method since its sites do not have to be adjacent. Maintaining such information, in order to keep getting the bbox in constant time, would be time and memory consuming. Instead of providing a method directly in *p_array*, an algorithm is available if this information is needed. *p_array* and *box* both have a nsites method since the internal structure allows a constant time retrieval.

```
p_array<point2d> arr;
arr.insert(point2d(1,0));
arr.insert(point2d(1,1));

// The bbox is computed thanks to bbox() algorithm.
box2d box = geom::bbox(arr);
std::cout << box << std::endl;
```

```
    // p_array provides nsites (),
    // it can be retrieved in constant time.
    std::cout << "nsites = " << arr.nsites() << std::endl;
```

Output:

```
[(1,0)..(1,1)]
nsites = 2
```

Chapter 5

Image

5.1 Definition

An image is composed both of:

- A function

$$ima : \left\{ \begin{array}{ccc} Site & \rightarrow & Value \\ p & \mapsto & ima(p) \end{array} \right.$$

- A site set, also called the "domain".

5.2 Possible image types

Here is a short list of the main/usual image types you may want to use with Olena:

Name	Description
image1d	1D image
image2d	2D image
image3d	3D image
flat_image	Constant value image
image_if	Image defined by a function

5.3 Possible value types

Every image type **must** take its type of value as parameter. The value type can be one of the builtins one:

- bool

- char

- *unsigned*

- *int*

- *short*

- *long*

- *float*

- *double*

Other data types are also available:

Value type	underlying data type
float01_8	*unsigned long*
float01_16	*unsigned long*
float01_f	*float*
gl8	*unsigned char*
gl16	*unsigned short*
glf	*float*
hsi_d	*double*
hsi_f	*float*
int_s8	*char*
int_s16	*short*
int_s32	*int*
int_u8	*unsigned char*
int_u16	*unsigned short*
int_u32	*unsigned int*
rgb16	*mln::algebra::vec<unsigned short>*
rgb8	*mln::algebra::vec<unsigned char>*

All these types are available in mln/value and accessible in the *mln::value* namespace. Most of the time, the name of the header which **must** be included to use one of these data types is actually "type_name.hh". For instance, for *rgb8* the header will be *rgb8.hh*.

5.4   Domain

The site set contains the sites which compose the image. Sites are based on a grid so the image depends on that grid as well. It means that a 2D images can only be defined by sites based on a 2D grid. Likewise, an *image2d* will always have its bouding box defined by a *box2d*.

Being defined on a grid means that the image can be defined anywhere. For instance, defining a 2D image with a *box2d* starting from point (-20, -20) to (-3, 5) is completely valid.

The following example shows that the definition domain and the site set are exactly equivalent.

```
// Define a box2d from (-2,-3) to (3,5).
box2d b = make::box2d(-2,-3, 3,5);
// Initialize an image with b as domain.
image2d<int> ima(b);

std::cout << "b = " << b << std::endl;
std::cout << "domain = " << ima.domain() << std::endl;
```

Output:

```
b = [(-2,-3)..(3,5)]
domain = [(-2,-3)..(3,5)]
```

To know if a site belongs to an image domain or not, a method "*has()*" is available.

```
// Create an image on a 2D box
// with 10 columns and 10 rows.
image2d<bool> ima(make::box2d(10, 10));

mln_site_(image2d<bool>) p1(20, 20);
mln_site_(image2d<bool>) p2(3, 3);

std::cout << "has(p1)? "
          << (ima.has(p1) ? "true" : "false")
          << std::endl;

std::cout << "has(p2)? "
          << (ima.has(p2) ? "true" : "false")
          << std::endl;
```

Output:

```
has(p1)? false
has(p2)? true
```

Since the notion of site is independent from the image it applies on, we can form expressions where a site passed to several images:

```
point2d p(9,9);

// At (9, 9), both values change.
ima1(p) = 'M';
ima2(p) = 'W';

bool b = (ima1(p) == ima2(p));
std::cout << (b ? "True" : "False") << std::endl;
```

Output:

```
False
```

16

5.5 Border and extension

Olena provides extension mechanisms for the image domain. In the library, both the concept of border and of extension can be encountered. These concepts are useful in many algorithms and can avoid costly tests while working with sites located on image edges.

5.5.1 Image border

A border is a finite extension provided to a basic image type, such as *image2d*. By default, every image is created with a border. The default width is defined through the global variable *border :: thickness* defined in *mln/border/thickness.hh*. Since this is a variable, it can be changed as shown in the following example.

```
bool vals[3][3] = { { 0, 1, 1 },
                    { 1, 0, 0 },
                    { 1, 1, 0 } };

image2d<bool> ima_def = make::image(vals);
border::fill(ima_def, false);
debug::println_with_border(ima_def);

std::cout << "═══════════" << std::endl << std::endl;

border::thickness = 0;
image2d<bool> ima_bt0 = make::image(vals);
debug::println_with_border(ima_bt0);
```

Output:

```
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - | | - - -
- - - | - - - - -
- - - | | - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -

═══════════

- | |
| - -
| | -
```

It is important to note that to display the border in the ouput, we use a special debug function, *debug::println_with_border*. Indeed, the border and the

extension are considered as part of an image only in the algorithms. They are ignored while saving or printing an image.

Some operations can be performed on the border. The functions are located in *mln/border*.

Routine	Description
adjust	Increase the border thickness if it is inferior to a minimum.
duplicate	Assign the border with the duplicate of the edges of this image.
equalize	Equalize the border of two images so that their size is equal and is at least a minimum size.
fill	Fill the border with a given value.
find	Find the border thickness of an image.
get	Get the border thickness of an image.
mirror	Fills border using nearer pixels with a mirroring effect.
resize	Set image border to a specific size.

5.5.2   Generality on image extension

On morphed images, described in section **??**, the border concept does not exist and is generalized to the extension concept. A simple example of a morphed image is a sub-image. A sub image does not have border nor extension by default. Thanks to *mln/core/routine/extend.hh*, an extension can be defined through a function. This means that the extension can be infinite. Another point is that an image can be used as extension. For instance, in the case of a sub-image, you may be interested in extending the sub-image with the image itself.

The extension supports the following operations. These functions are located in *mln/extension*.

Routine	Description
adjust	Adjust the extension given a structural element.
adjust_duplicate	Adjust the size of the extension given a structural element and duplicate the image inner boundary.
adjust_fill	Adjust the size of the extension given a structural element and fill it with a value.
duplicate	Duplicate the values of the image inner boundary in the extension.
fill	Fill the extension with a given value.

In order to extend an image, a routine *extend* is available in *mln/core/routine/extend.hh*. The routine *extended_to* may also help during debug. It allows to extend the image domain to a larger one. The values associated to the new sites comes from the extension.

5.5.3 Different extensions

Let's say we have want to extract a sub domain from an image. In the following example, *ima_roi* holds several small rectangles from the original image.

```
image2d<rgb8> lena;
io::ppm::load(lena, MLN_IMG_DIR "/small.ppm");
box2d bbox_enlarged = lena.domain();
bbox_enlarged.enlarge(border::thickness);
mln_VAR(ima_roi, lena | fun::p2b::big_chess<box2d>(lena.domain(), 10));
```


$\longrightarrow$
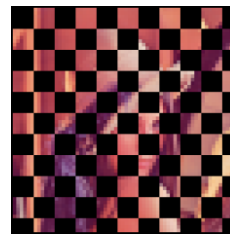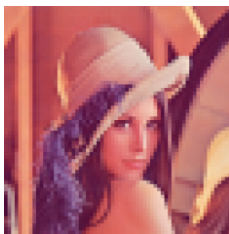
lena   *ima_roi* (black color means the sites are not included in the domain)

Then, we may extend this sub image with one of the three following extension type.

Extension with a value

Let's extend with the value *literal::blue*.

```
mln_VAR(ext_with_val, extended_to(extend(ima_roi, literal::blue), bbox_enlarged));
```
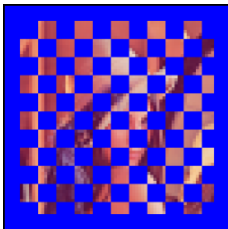

Note the use of the *extended_to()* routine. We used a larger bbox to extend the image domain. That is the reason why the image is surrounded by the extension value, blue.

Extension with a function

Let's use the following function:

```
namespace mln
{
```

19

```
    struct my_ext : public Function_v2v<my_ext>
    {

      typedef value::rgb8 result;

      value::rgb8 operator()(const point2d& p) const
      {
        if ((p.row() + p.col()) % 20)
          return literal::black;
        return literal::white;
      }

    };

} // end of namespace mln
```

```
    mln_VAR( ext_with_fun , extended_to ( extend ( ima_roi , my_ext ()) , bbox_enlarged ));
```
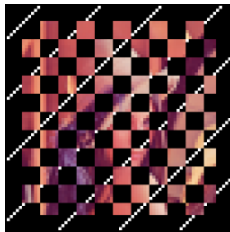

Extension with an image

Let's extend with the original image, *lena*.

```
    mln_VAR( ext_with_ima , extend ( ima_roi , lena ));
```
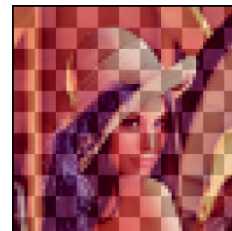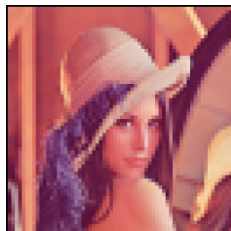

ext_with_ima, the extended image.   The actual data in the domain (light) with its extension (dark)

IMPORTANT NOTE

Many times, you may want to check if a site is part of the image before applying a treatment. All images provide a method "*has(Site)*" which can return this information. **Be careful though, calling *has()* on the image returns "true" if**

the given site is part of the domain OR the the extension/border. All algorithms in Olena call that method which means that all the algorithms take in consideration the extension/border if it exists. The default border thickness is set to 3 as shown by the following example.

```
// Default border size is set to 0.

// Image defined on a box2d from
// (0, 0) to (2, 2)
image2d<int> ima1(2, 3);

std::cout << "ima1.has(0, 0) : "
          << ima1.has(point2d(0, 0)) << std::endl;

std::cout << "ima1.has(-3, 0) : "
          << ima1.has(point2d(-3, 0)) << std::endl;

std::cout << "ima1.has(2, 5) : "
          << ima1.has(point2d(2, 5)) << std::endl;

std::cout << "═══════════" << std::endl;

// Set default border size to 0.
border::thickness = 0;

// Image defined on a box2d from
// (0, 0) to (2, 2)
image2d<int> ima2(2, 3);

std::cout << "ima2.has(0, 0) : "
          << ima2.has(point2d(0, 0)) << std::endl;

std::cout << "ima2.has(-3, 0) : "
          << ima2.has(point2d(-3, 0)) << std::endl;

std::cout << "ima2.has(2, 5) : "
          << ima2.has(point2d(2, 5)) << std::endl;
```

Output:

```
ima1.has(0, 0) : 1
ima1.has(-3, 0) : 1
ima1.has(2, 5) : 1
═══════════
ima2.has(0, 0) : 1
ima2.has(-3, 0) : 0
ima2.has(2, 5) : 0
```

Most of the time, this is the good behavior. For instance, if a rotation is applied to an image, sites which were not previously in the domain will be part

of it. Thanks to the extension/border, these sites will be associated to the value of the extension/border.

In the following example, the extension is set to a constant color *yellow*. It means that whatever the new sites computed through the rotation, it will be part of the image and a value will be available. Site which were previously in the extension/border, will be associated to *yellow* in the output image.

```
border :: thickness = 30;

// Declare the image to be rotated.
image2d<value :: rgb8> ima1_(220, 220);
data :: fill (ima1_, literal :: cyan );
border :: fill (ima1_, literal :: yellow );
// Set an infinite extension.
mln_VAR(ima1, extend (ima1_, pw:: cst (literal :: yellow )));

// Declare the output image.
image2d<value :: rgb8> ima2(220, 220);
data :: fill (ima2, literal :: cyan );
border :: fill (ima2, literal :: yellow );

box2d extended_domain= ima1.domain ();
extended_domain. enlarge (border :: thickness );

// Draw the domain bounding box
draw :: box (ima1, geom :: bbox (ima1_), literal :: red );
// Save the image, including its border.
doc :: ppmsave (ima1 | extended_domain , "ima2d-rot" );

// Define and apply a point-wise rotation
fun :: x2x :: rotation <2, float > rot1 (0.5, literal :: zero );
image2d<value :: rgb8 >:: fwd_piter p (ima1.domain ());
for_all (p)
{
   algebra :: vec <2, float > pv = p.to_site ().to_vec ();
   algebra :: vec <2, float > v = rot1.inv ()(pv );
   ima2 (p) = ima1 (v);
}

draw :: box (ima2, ima2.bbox (), literal :: red );
doc :: ppmsave (extended_to (ima2, extended_domain ), "ima2d-rot" );
```

Output:
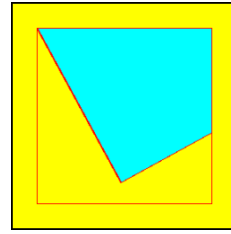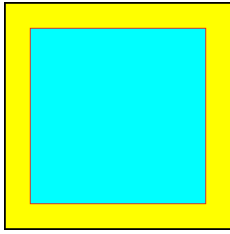
*ima*1 and its border before rotation (left) and *ima*2 and its border after rotation (right).

Sometimes taking the domain in consideration may not be the expected behavior. If you do not want to use the extension/border for a specific routine, simply restrict the image to its domain.

```
my_routine(ima | ima.domain());
```

Note that:

$$ima.domain().has() \equiv (ima - ima.domain()).has()$$

5.6   Interface

Return Type	Name	Arguments	Const	Comments
I::pvset	domain	-	X	
const Value&	operator()	const point& p	X	Used for reading.
Value&	operator()	const point& p	-	Used for writing.
bool	has	const Point& p	X	
bool	has_data	-	X	Returns true if the domain is defined.
site_id	id	-	X	Return the Id of the underlying shared data.
I::vset	destination	-	X	Value set of all the possible site values in this Image.
site_set	bbox	-	-	Returns the bounding box of the domain.
site_set	bbox_large	-	-	Returns the bouding box of the domain and the extended domain.

5.7   Load and save images

Currently, Olena supports the following input image formats:

- PBM

- PFM

- PGM

- PNM

- PPM

This support is provided through two headers for each type, *save.hh* and *load.hh*. They are located in *mln/io/<image-format>/*.

Once the right header is included, the image can be loaded:

```
image2d<bool> ima;
io::pbm::load(ima, MLN_DOC_DIR "/img/small.pbm");
```

Note that each format is associated to specific image value types:

hline Format	Value type
PBM	bool
PFM	float, double, float01_*
PGM	unsigned, long, int, int_u*, gl*
PNM	See PGM, PBM and PPM
PPM	rgb*

```
io::pbm::save(ima, MLN_DOC_DIR "/figures/ima_save.pbm");
```

5.8   Create an image

Loading an image is not mandatory, an image can be created from scratch. There are two possibilites to do so:

```
// Build an empty image;
image2d<value::int_u8> img1a;

// Build an image with 2 rows
// and 3 columns sites
image2d<value::int_u8> img1b(box2d(2, 3));
image2d<value::int_u8> img1c(2, 3);
```

*img*1*a* has no data and its definition domain is still unset. We do not know yet the number of sites it contains. However, it is really useful to have such an "empty image" because it is a placeholder for the result of some processing, or another image. Trying to access the site value from an empty image leads to an error at run-time. *img*1*b* is defined on a domain but does not have data yet.

An image can also be created and initialized at the same time:

```
bool vals[6][5] = {
    {0, 1, 1, 0, 0},
    {0, 1, 1, 0, 0},
    {0, 0, 0, 0, 0},
```

```
    {1, 1, 0, 1, 0},
    {1, 0, 1, 1, 1},
    {1, 0, 0, 0, 0}
};
image2d<bool> ima = make::image(vals);
```

It constructs the following image:
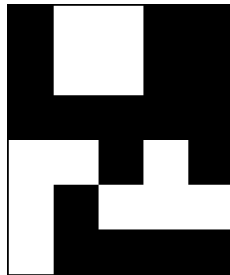
Sometimes, you may want to initialize an image from another one:

```
image2d<value::int_u8> img2a(2, 3);
image2d<value::int_u8> img2b;

initialize(img2b, img2a);
data::fill(img2b, img2a);
```

img2b is declared without specifying a domain. Its border size is set to the default one, e.g 0. By using *initialize()*, *img2b* is initialized with the same domain and border/extension as *img2a*. The data is not copied though. Other routines like *data::fill()* can be called in order to do so (See also 10.1).

5.9   Access and modify values

There are several ways to access/modify an image "*ima*":

- *opt::at(ima, x, y, ...)*

- *ima(Site)*

Most of the time, images can be modified and these two methods can be used both to read a value and modify it. Both methods are equivalent.

```
box2d b(2,3);
image2d<value::int_u8> ima(b);

// On image2d, Site <=> point2d
point2d p(1, 2);

// Associate '9' as value for the site/point2d (1,2).
```

```
// The value is returned by reference and can be changed.
opt::at(ima, 1,2) = 9;
std::cout << "opt::at(ima, 1,2) = " << opt::at(ima, 1,2)
          << std::endl;
std::cout << "ima(p) = " << ima(p) << std::endl;

std::cout << "———" << std::endl;


// Associate '2' as value for the site/point2d (1,2).
// The value is returned by reference
// and can be changed as well.
ima(p) = 2;
std::cout << "opt::at(ima, 1,2) = " << opt::at(ima, 1,2)
          << std::endl;
std::cout << "ima(p) = " << ima(p) << std::endl;
```

Output:

```
opt::at(ima, 1,2) = 9
ima(p) = 9
———
opt::at(ima, 1,2) = 2
ima(p) = 2
```

Usually, you will want to use the functional way, "*ima(Site)*", more particularly while iterating over all the sites through an iterator. This use case will be detailed further in section 8.

5.10 Image size

Most typical image types owns special methods to retrieve the image size.

Image type	Methods
image1d	length()
image2d	ncols(), nrows()
image3d	ncols(), nrows(), nslis()

If you need a more generic way to get the size, you can use the routines provided in *mln/geom* in the following files:

- *ncols.hh*

- *nrows.hh*

- *nslis.hh*

```
image2d<int> ima(make::box2d(0,0, 10,12));

std::cout << "nrows = " << ima.nrows()
          << " - "
```

```
            << "ncols = " << ima.ncols()
            << std::endl;
```

Output:

```
nrows = 11 − ncols = 13
```

Chapter 6

Structural elements: Window and neighborhood

In Olena, there are both the window and neighborhood concept. A window can be defined on any site around a central site which may also be included. A neighborhood is more restrictive and **must** not include the central site. Therefore these two concepts are really similar and are detailed together in this section.

By default, structural elements are centered. The central site is located at the origin of the grid: *"literal :: origin"*. With *image2d*, the central site is (0,0). Centered structural elements **must** have an odd size.

6.1   Define an element

6.1.1   Window

Generic Predefined windows

Name	Description	Representation
win_c4p	4-connectivity	
win_c8p	8-connectivity	

1D Predefined windows

Name	Description	Representation
segment1d	1D segment	-

2D Predefined windows

Name	Description	Representation
backdiag2d	Back diagonal	-
diag2d	2D diagonal	-
disk2d	2D disk	-
hline2d	2D horizontal line	-
octagon2d	2D octogon	-
rectangle2d	2D rectangle	-
vline2d	2D vertical line	-

3D Predefined windows

Name	Description	Representation
cube3d	3D Cube	-
cuboid3d	Cuboid	-

These predefined windows can be passed directly to a function. The headers are located in *mln/core/alias/window*.hh*.

6.1.2   Neighborhood

Predefined neighborhood:

Name	Description	Representation
c4	4-connectivity	
c8	8-connectivity	

These predefined neighborhood can be passed directly to a function. The headers are located in *mln/core/alias/neigh*.hh*.

Use case example:

```
label_8 nlabels;
image2d<label_8> lbl = labeling::blobs(ima, c4(), nlabels);
```

6.1.3   Custom structural elements

Windows

There are several ways to define a new window. The first and the most common way is to declare a window variable and insert dpoints:

```
window2d win;
win.insert(−1, −1);
win.insert(−1,  0);
win.insert(−1,  1);
```

29

This code creates the following window where "X" is the central point from which the window is computed:

```
o −
o X
o −
```

Another way to define the same window is to provide a bool array:

```
bool b[9]      =    { 1, 0, 0,
                       1, 0, 0,
                       1, 0, 0 };

bool b2[3][3] = { { 1, 0, 0 },
                  { 1, 0, 0 },
                  { 1, 0, 0 } };

window2d win = convert::to<window2d>(b);
window2d win2 = convert::to<window2d>(b2);
```

Note that despite the different ways of defining these windows, "varwin" == "*win*2". The boolean array **must** always have an odd size. While creating a windows thanks to a bool array/matrix, the window's center is the central site of the array/matrix.

6.1.4 Conversion between Neighborhoods and Windows

Windows are not convertible to a Neighborhood. Neighborhood are convertible to a window though.

A neighborhood has a method "*win()*" which returns the definition window. Be ware that this window is not centered, thus does not include the central point.

Chapter 7

Sites, psites and dpoints

7.1   Need for site

As we have seen before, an image is usually defined on a grid. It has associated
data and a site set which defines the domain of the image on that grid. Usually,
we need to access a value by its coordinates. With default images it can be done
easily, at no cost.

Example with an *image2d*:

```
  c  0  1  2  3
r
    +--+--+--+--+
0  |  |×|  |  |
    +--+--+--+--+
1  |  |  |  |  |
    +--+--+--+--+
```

The site x is the point (0, 1).   The image values are stored in a multi-
dimensional array. The point coordinates can be used directly. The site (0, 1)
is the point (0, 1) and the data is stored at row 0 and column 1.

Here we have:

I::site == I::psite == point2d

where, roughly, *point2d* = { row, column }.

7.2   Need for psite

Sometimes, accessing a value in constant-time complexity, O(1), is not possible
with a site object.

Let's have a small example. Define a function returning a value for a given
point:

```
unsigned my_values(const mln::point2d& p)
{
  if (p.row() == 0)
    return 8;
  return 9;
}
```

So, for each point having (0, x) as coordinate, this function will return 8, otherwise it will be 9.

Then, define a *p_array* with few *point2d*:

```
p_array<point2d> arr;
arr.append(point2d(3, 6));
arr.append(point2d(3, 7));
arr.append(point2d(3, 8));
arr.append(point2d(4, 8));
arr.append(point2d(4, 9));
```

Now, create a point-wise image from this function and this *p_array*:

```
mln_VAR(ima, my_values | arr);
```

ima is actually that image:

```
c   6 7 8 9
r

    +-+-+-+
3   | |×| |
    +-+-+-+-+
4       | | |
        +-+-+
```

However, in memory, since it is based on a *p_array*, sites are stored in a vector.

The site x is the point (3, 7) which corresponds to the cell 1 in the *p_array*.

```
arr[] = 0 1 2 3 4
        +-+-+-+-+-+
        | |×| | | |
        +-+-+-+-+-+
```

Obviously, we cannot check in constant time whether the site x, *point2d* here, is part of that image or not: knowing the point coordinates is not enough. That's why we need a different mechanism in order to access this information: the psites.

Here we have:

I::site == point2d but *I::psite == pseudo_site<point2d>*

where, roughly, *pseudo_site<point2d>* = { i_in_p_array, p_array_ptr }.

Psites contains all the needed information to access the values in constant-time.

7.3 From psite to site

In the last example there was an image of type *I* such as *I::site* != *I::psite*. In that case, an object of type *I::psite* is actually convertible towards an object of type *I::site*. Furthermore, a *psite* shall behave as if it was a *site*.

Design note: it seems impossible to offer through the interface of some *psite* what is expected from its corresponding *site*. For instance, when a site has a given feature, say a method "*m*", then this method has to appear in the *psite* interface. However, thanks to inheritance, we fetch an interface and an implementation that delegates to the site.

For instance, in the last example, *I::psite* has a method *row()* because *I::site*, *point2d*, provides such a method.

How it works: a *psite* inherits from *internal::site_impl<site>* which is specialized for every site type; for instance, *internal::site_impl<point2d>* owns the method "*coord row() const*" which is defined as "*return exact(this)->to_site().row()*".

7.4 Dpoint

Dpoints are relative sites. They are usually used in window and neighborhood definitions. Since the central site is not constant, windows and neighborhoods **must** be recomputed and dpoints help in doing that.

```
dpoint2d dp(−1,0);
point2d p(1,1);

std::cout << p + dp << std::endl;
```

Output:

```
(0,1)
```

Chapter 8

Iterators

Each container object in Olena like site sets or images have iterators. The iteration mechanism for images is directly derived from the mechanism for site sets.

There are usually three kinds:

- **fwd_iter**, depends on the container,

- **bkd_iter**, iterates like forward but to the opposite way,

- **iter**, usually the same as fwd_iter. It is guaranteed to iterate all over the elements.

Every iterable object have these three kinds of iterator. There are all bidirectional containers. Whatever the iterator used, the basic iterator has the only property of browsing every site once.

The iterator type name depends on the data pointed by it:

Data type	Iterator Names
Site	fwd_piter, bkd_piter, piter
Value	fwd_viter, bkd_viter, viter
Neighboors	fwd_niter, bkd_niter, niter

As you may have noticed, according to the data type, the word "iter" is prefixed by the usual name variable used for that data type. Sites variables are usually called "p" so the proper iterator is "typepiter". (See the foreword)

An iterator has the following interface:

Return Type	Name	Arguments	Const	Comments
void	start	-	-	
void	next	-	-	
bool	is_valid	-	-	Return false if created with the default constructor and not associated to a proper container.

Example of different forward iterations:

- box2d: from top to bottom then from left to right.

- p_array<point2d>: from left to right.

A *for_all()* macro is available to iterate over all the sites:

```
box2d b(3, 2);
mln_piter_(box2d) p(b);

for_all(p)
  std::cout << p; //prints every site coordinates.
```

Output:

```
(0,0)(0,1)(1,0)(1,1)(2,0)(2,1)
```

Note that when you declare an iterator, prefer using the "*mln_*iter*" macros. They resolve the iterator type automatically from the given container type passed as parameter. These macros can be used with any container like images or site sets.

Here follow an example with the implemantions of the most basic routines which use the *for_all()* loop: *data::fill()* and *data::paste()*.

```
template <typename I>
void fill(I& ima, mln_value(I) v)
{
  mln_piter(I) p(ima.domain());
  for_all(p)
    ima(p) = v;
}
```

```
template <typename I, typename J>
void paste(const I& data, J& dest)
{
  mln_piter(I) p(data.domain());
  for_all(p)
  dest(p) = data(p);
}
```

Important note: macros for iterators exist in two versions: "*mln_*iter*" and "*mln_*iter_*". The difference is that the first version **must** be used in templated function whereas the second one **must** be used in non templated functions.

If you want a list of all the macros available in Olena, please refert to section 14.

Chapter 9

Memory management

In the Olena library, all image types behave like image2d:

- An "empty" image is actually a mathematical variable.

  → just think in a mathemetical way when dealing with images;

- No dynamic memory allocation/deallocation is required. the user never has to use "new / delete" (the C++ equivalent for the C "malloc / free") so she does not have to manipulate pointers or to directly access memory.

  → Olena prevents the user from making mistakes;

- Image data/values can be shared between several variables and the memory used for image data is handled by the library.

  → Memory management is automatic.

Exemple with image2d

Images do not actually store the data in the class. Images store a pointer to an allocated space which can be shared with other objects. Once an image is assigned to another one, the two images share the same data so they have the same ID and point to the same memory space. Therefore, assigning an image to another one is NOT a costly operation. The new variable behaves like some mathematical variable. Put differently it is just a name to designate an image:

```
image2d<int> ima1(box2d(2, 3));
image2d<int> ima2;
point2d p(1,2);

ima2 = ima1; // ima1.id() == ima2.id()
// and both point to the same memory area.

ima2(p) = 2; // ima1 is modified as well.
```

```
// prints "2 − 2"
std :: cout << ima2 ( p ) << " − " << ima1 ( p ) << std :: endl ;
// prints "true"
std :: cout << ( ima2 . id_ () == ima1 . id_ () ) << std :: endl ;
```

If a deep copy of the image is needed, a *duplicate()* routine is available:

```
image2d<int> ima1 ( 5 ,  5 );
image2d<int> ima3 = duplicate ( ima1 );  // Makes a deep copy.

point2d  p ( 2 ,  2 );
ima3 ( p )  =  3;

std :: cout << ima3 ( p ) << " − " << ima1 ( p ) << std :: endl ;
std :: cout << ( ima3 . id_ () == ima1 . id_ () ) << std :: endl ;
```

Output:

```
3 − 0
0
```

Chapter 10

Basic routines

Routine name	Description
duplicate()	creates a deep copy of an object. Any shared data is duplicated.
data::fill()	fill an object with a value.
data::paste()	paste object data to another object.
labeling::blobs()	find and label the different components of an image.
logical::not_() logical::not_inplace()	Point-wise "logical not"
*::compute()	compute an accumulator on specific elements.

10.1   Fill

First, create an image:

```
image2d<char> imga(5, 5);
```

Memory has been allocated so data can be stored but site values have not been initialized yet. So we fill *imga* with the value 'a':

```
data::fill(imga, 'a');
```

The *fill()* algorithm is located in the sub-namespace "*mln::data*" since this algorithm deals with the site values.

The full name of this routine is *mln::data::fill()*. To access to a particular algorithm, the proper file shall be included. The file names of algorithms strictly map their C++ name; so *mln::data::fill* is defined in the file *mln/data/fill.hh*.

Note

Most algorithms in Olena are constructed following the classical scheme: "output algo(input)", where the input image is only read. However some few algorithms take an input image in order to modify it. To enforce this particular feature, the user shall explicitly state that the image is provided so that its data

is modified "read/write". The algorithm call shall be *data::fill(ima.rw(), val).* When forgetting the *rw()* call, it does not compile.

```
data :: fill ((imga | box2d (1 ,2)). rw (), 'a ');
```

10.2   Paste

We then define below a second image to play with. As you can see this image has data for the sites (5, 5) to (14, 14) (so it has 100 sites).

```
image2d<unsigned char> imgb(make :: box2d (5 ,5,  7 ,8));
// Initialize imga with the same domain as imgb.
image2d<unsigned char> imga(imgb.domain ());

// Initialize the image values.
data :: fill (imgb, 'b ');

// Paste the content of imgb in imga.
data :: paste (imgb, imga );

debug :: println (imga );
```

Output:

```
98  98  98  98
98  98  98  98
98  98  98  98
```

Note

With this simple example we can see that images defined on different domains (or set of sites) can interoperate. The set of sites of an image is defined and can be accessed and printed. The following code:

```
image2d<int> ima1 (5,  5);
image2d<int> ima2 (10,  10);

std :: cout << "ima1.domain () = " << ima1.domain ()
            << std :: endl ;
std :: cout << "ima2.domain () = " << ima2.domain ()
            << std :: endl ;
```

Gives:

```
image2d<int> ima1 (5,  5);
image2d<int> ima2 (10,  10);

std :: cout << "ima1.domain () = " << ima1.domain ()
```

```
                    << std::endl;
    std::cout << "ima2.domain() = " << ima2.domain()
                    << std::endl;
```

The notion of site sets plays an important role in Olena. Many tests are performed at run-time to ensure that the program is correct.

For instance, the algorithm *data::paste()* tests that the set of sites of *imgb* (whose values are to be pasted) is a subset of the destination image.

10.3   Blobs

labeling::blobs() is used to label an image. It returns a new image with the component id as value for each site. The background has 0 as id therefore the component ids start from 1.
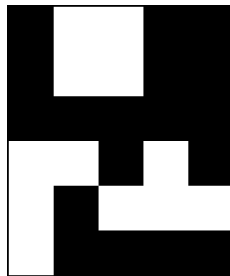
Consider the following image:

```
bool vals[6][5] = {
    {0, 1, 1, 0, 0},
    {0, 1, 1, 0, 0},
    {0, 0, 0, 0, 0},
    {1, 1, 0, 1, 0},
    {1, 0, 1, 1, 1},
    {1, 0, 0, 0, 0}
};
image2d<bool> ima = make::image(vals);
```
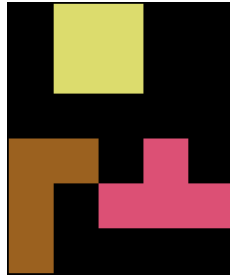
Output:

Then label this image thanks to *labeling::blobs()*:

```
label_8 nlabels;
image2d<label_8> lbl = labeling::blobs(ima, c4(), nlabels);
```

Output:

Note that this routine returns the number of components in its third parameter. This parameter **must** be of the same type as the returned image value.
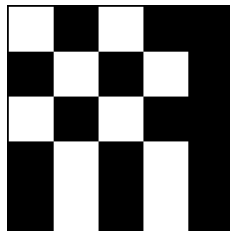
10.4 Logical not

Header	mln/logical/not.hh
Full namespace	mln::logical
Routine(s)	not_(), not_inplace()

This small routine only works on binary images. It performs a point-wise "logical not" and therefore "negates" the image. There are two versions of that algorithm: a version which returns a new image and another which works in place. Example:
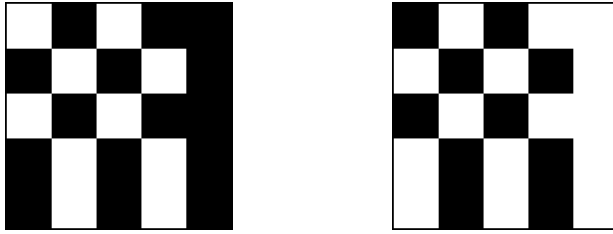
Make a binary image:

```
bool vals[5][5] = {
  {1, 0, 1, 0, 0},
  {0, 1, 0, 1, 0},
  {1, 0, 1, 0, 0},
  {0, 1, 0, 1, 0},
  {0, 1, 0, 1, 0}
};
image2d<bool> ima = make::image(vals);
```


Return the result in a new image:

```
image2d<bool> ima_neg = logical::not_(ima);
```
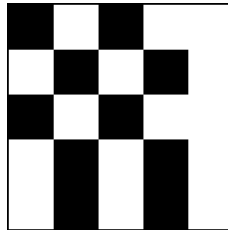
ima (left) and ima_neg (right) after having called logical::not_().

Or, work in place:

```
logical::not_inplace(ima);
```

Then, *ima* looks like:

10.5   Compute

There are several flavour of the compute routine, depending on what the kind of elements it computes.

labeling::compute()	compute an accumulator for each component in a labeled image.
data::compute()	compute an accumulator on the values of an image.

10.5.1   Accumulators

An accumulator is a special object accumulating data while iterating all over the image values or sites. Hereby follows a list of accumulators available in Olena.

Accumulators on sites

Name	Description
bbox	Bounding boxes
count_adjacent_vertices	Count adjacent vertices
count	Count the number of sites
height	
volume	

Accumulators on values

Name	Description
histo	Histogram
max	Max value
max_h	Max value (Hexa)
mean	Mean value
median_alt	Median
median_h	Median (Hexa)
min	Min value
min_h	Min value (Hexa)
min_max	Min and Max value
rank_bool	
rank	
rank_high_quant	
sum	Sum the values

Special accumulators

Name	Description
pair	Pair of accumulators
tuple	n-uplets of accumulators

Each accumulator can be used in *::compute()*. It exists two versions of each accumulator.

- *mln::accu::**, this version require the site or value type as parameter. For instance, for the bbox accumulator, the type would be *accu::bbox¡mln_psite(I)¿*, where I is the type of the image on which it will be computed.

- *mln::accu::meta::**, this is usually the easiest version to use. The type of site or value do not need to be specified and will be deduced at compile time. For the bbox accumulator, the accumulator type would be *accu::meta::bbox*.

Note that when an accumulator is passed to *::compute()*, it **must** be instanciated. You cannot write:

```
data :: compute ( accu :: meta :: stat :: max, ima );
```

Instead, you **must** write:

```
data :: compute ( accu :: meta :: stat :: max (), ima );
```

10.5.2 Example with labeling::compute()

In this example we will try to retrieve the bounding box of each component in an image.

Consider the following image:

```
bool vals[6][5] = {
    {0, 1, 1, 0, 0},
    {0, 1, 1, 0, 0},
    {0, 0, 0, 0, 0},
    {1, 1, 0, 1, 0},
    {1, 0, 1, 1, 1},
    {1, 0, 0, 0, 0}
};
image2d<bool> ima = make::image(vals);
```
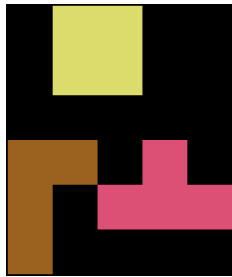
Then label this image thanks to *labeling::blobs()*:

```
label_8 nlabels;
image2d<label_8> lbl = labeling::blobs(ima, c4(), nlabels);
```


Output:

Then, use *labeling::compute()* with the bbox accumulator:

```
util::array<box2d> boxes =
    labeling::compute(accu::meta::shape::bbox(),
                      lbl,
                      nlabels);
```

labeling::compute() holds an accumulator for each component, which means it returns an array of accumulator results. In this case, it returns an array of *box2d*.

Important note: since *labeling::blobs()* labels the component from 1 and set the background to 0, we will want to iterate from 1 to nlabels included.

```
for (unsigned i = 1; i <= nlabels; ++i)
    std::cout << boxes[i] << std::endl;
```

Output:

```
[(0,1)..(1,2)]
[(3,0)..(5,1)]
[(3,2)..(4,4)]
```

10.5.3 Routines based on accumulators and *::compute()

In order to make the code cleaner, small routines are available for the most used accumulators.

Currently there are the following routines:

Name	Description
nsites	Return the number of sites of an image or a site set.
mean	Return the mean of the values of an image.
min_max	Return the min and max values of the values of an image.
sum	Return the sum of the values of an image.

These routines can be found in *mln/geom* and in *mln/estim*. For example, with *geom::nsites()* simply write:

```
unsigned nsites = geom::nsites(ima);
```

10.6   Working with parts of an image

Sometimes it may be interesting to work only on some parts of the image or to extract only a sub set of that image. Olena enables that through the operator '|'.

Three kinds of that operator exist:

Prototype	Comments
Image | Sub Domain	Create a new image.
Image | Function_p2b	Do not create a new image but create a morpher.
Function_p2v | Sub Domain	Do not create a new image but create a morpher.

A Sub Domain can be a site set, an image or any value returned by this operator. For a given site, *Function_p2v* returns a value and *Function_p2b* returns a boolean. These functions are actually a sort of predicate. A common *Function_p2v* is *pw::value(Image)*. It returns the point to value function used in the given image. C functions can also be used as predicate by passing the function pointer.

You can easily get a *Function_p2b* by comparing the value returned by a *Function_p2v* to another Value. The following sample codes illustrate this feature.

In order to use C functions as predicate, they **must** have one of the following prototype if you work on 2D images:

```
//function_p2b
bool my_function_p2b(mln::point2d p);

//function_p2v
//V is the value type used in the image.
template <typename V>
V my_function_p2v(mln::point2d p);
```

Of course, you just need to change the point type if you use another image type. For instance, you would use *point3d* with 3D images. The returned value type

V for *Function_p2v* depends on the image value type. With *image2d<int>*, *V* would be *int*.

In this section, all along the examples, the image *ima* will refer to the following declaration:

```
bool vals[6][5] = {
    {0, 1, 1, 0, 0},
    {0, 1, 1, 0, 0},
    {0, 0, 0, 0, 0},
    {1, 1, 0, 1, 0},
    {1, 0, 1, 1, 1},
    {1, 0, 0, 0, 0}
};
image2d<bool> ima = make::image(vals);
```

Output:
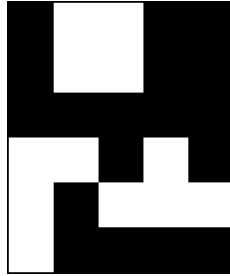
10.6.1 Restrict an image with a site set

A simple example is to fill only a part of an image with a specific value:
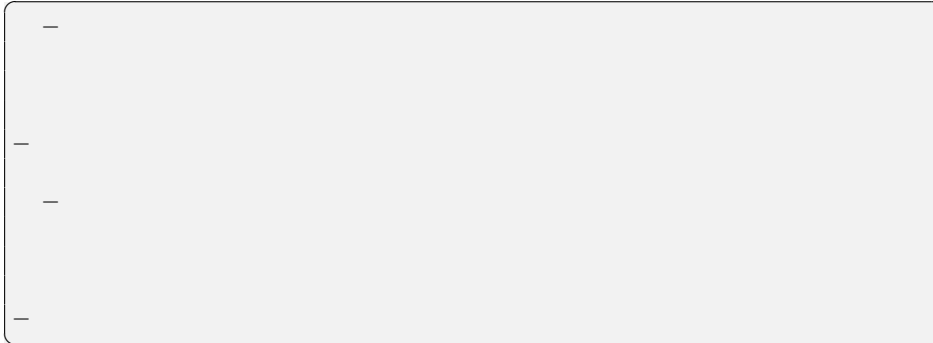
```
p_array<point2d> arr;

// We add two points in the array.
arr.append(point2d(0, 1));
arr.append(point2d(4, 0));

// We restrict the image to the sites
// contained in arr and fill these ones
// with 0.
// We must call "rw()" here.
data::fill((ima | arr).rw(), 0);

debug::println((ima | arr));

mln_VAR(ima2, ima | arr);
// We do not need to call "rw()" here.
data::fill(ima2, 0);
```

Output:
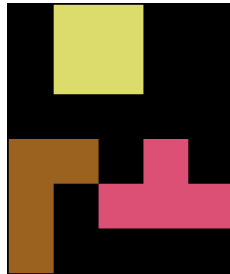
47

```
_



_


_




_
```

10.6.2 Restrict an image with a predicate

In the following example, we aim at extracting a component from an image and create a new image with it.

First, find and label the components.

```
label_8  nlabels ;
image2d<label_8> lbl = labeling :: blobs (ima , c4 () , nlabels );
```
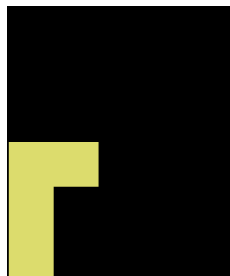
Output:

Then, restrict the image to the sites being part of component 2.

```
mln_VAR( lbl_2 , lbl  |  (pw:: value ( lbl ) == pw:: cst (2u )));
```
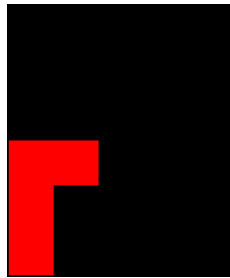
lbl_2 is a new image. *lbl_2* looks like:

Finally, create a new color image, fill it with black and fill the sites part of component 2 with red.

```
image2d<rgb8> ima2;
initialize(ima2, ima);
data::fill(ima2, literal::black);

data::fill((ima2 | lbl_2.domain()).rw(), literal::red);
```

Output:

The previous example can be written more quickly:

```
label_8 nlabels;
image2d<label_8> lab = labeling::blobs(ima, c4(), nlabels);

image2d<rgb8> ima2;
initialize(ima2, ima);
data::fill(ima2, literal::black);

data::fill((ima2 | (pw::value(lab) == pw::cst(2u))).rw(), literal::red);
```

10.6.3 Restrict an image with a C function

In this example, the image is restricted to its odd lines. A new image is created in which odd lines are in red and others in black.

Here is the simple C function used as predicate:

```
bool row_oddity(mln::point2d p)
{
  return p.row() % 2;
}
```

Restrict the image with it:

```
image2d<rgb8> ima2;
initialize(ima2, ima);
data::fill(ima2, literal::black);

data::fill((ima2 | row_oddity).rw(), literal::red);
```

Output:

Important note

When writing:

```
ima | sub_D
```

    sub_D **must** be included in *ima.domain()*.
    Let's have an image, *imab*, like this:

```
0 1 0
1 1 1
```

    Extract a sub image from *imab* with sites having their value set to 1.

```
mln_VAR(imab1, ima | (pw::value(ima) == pw::cst(1u)));
```

Then, *imab1* looks like:

```
1
1 1 1
```

    Now, if we want to extract a sub image it may fail, depending on the site set used:

```
box2d b1(1,0, 1, 2);
mln_VAR(imac, imab1 | b1);

// Print:
// 1 1 1
debug::prinln(imac);

box2d b2(0,0, 1, 1);
// Will fail at runtime.
// ima.domain().has((0,0)) is false.
mln_VAR(imad, imab1 | b2);
debug::prinln(imad);
```

    If you do not want this constraint, you may want to use an alternative operator:

```
ima / sub_D
```

Chapter 11

Input / Output

Olena offers a builtin support for PNM (PBM, PGM & PPM), PFM and dump
file formats.

You can extend the range of supported files by installing third-parties libraries such as:

- ImageMagick: support for usual images (PNG, TIFF, JPEG, ...)

- GDCM: support for DICOM medical images

11.1   ImageMagick

http://www.imagemagick.org
You have to install ImageMagick with Magick++ support. You will be able to
load every file recognized as an image by ImageMagick.
Olena only support binary and 8 bits images through ImageMagick.
During the compilation, you will have to specify the ImageMagick flags and
libraries.
To do so, just add the following line to your compilation:

```
'Magick++-config --cppflags --cxxflags --ldflags --libs'
```

Magick++-config will automatically fill the dependencies depending of your
installation.

11.2   GDCM

http://apps.sourceforge.net/mediawiki/gdcm
GDCM is a library for manipulating DICOM files. DICOM files are used in
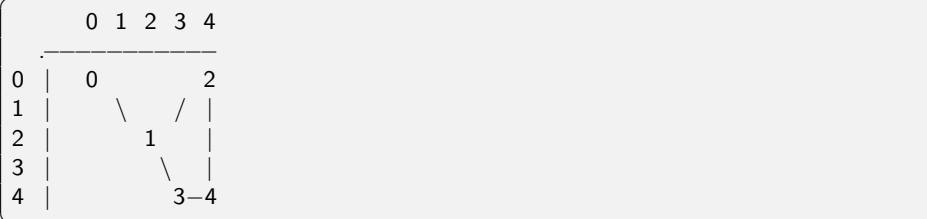medical imaging.

Chapter 12

Graphs and images

12.1 Description

Olena enables the possibility of using graphs with images. Graphs can help you to handle directly parts of an image and represent their relationship. Specific data can be associated to each vertex and/or edges.

12.2 Example

First, create a graph which looks like the following:

```
        0  1  2  3  4
      ._____
  0 |   0          2
  1 |      \    /  |
  2 |         1    |
  3 |          \   |
  4 |           3−4
```

First we need to add vertices:

```cpp
util::graph g;

for (unsigned i = 0; i < 5; ++i)
  g.add_vertex(); // Add vertex 'i';
```

Finally, populate the graph with edges:

```cpp
g.add_edge(0, 1); // Associated to edge 0.
g.add_edge(1, 2); // Associated to edge 1.
g.add_edge(1, 3); // Associated to edge 2.
g.add_edge(3, 4); // Associated to edge 3.
g.add_edge(4, 2); // Associated to edge 4.
```

Now there is a graph topology and we want to associate elements of this graph to a site in the image. The idea is to use specific site sets such as *p_vertices* and *p_edges*. Let's associate the vertices with sites. To do so we need a function which maps a vertex id to a site, e.g. a *point2d* here.

```
typedef fun::i2v::array<point2d> F;
F f(5); // We need to map 5 vertices.
f(0) = point2d(0, 0);
f(1) = point2d(2, 2);
f(2) = point2d(0, 4);
f(3) = point2d(4, 3);
f(4) = point2d(4, 4);
```

Then declare a *p_vertices*:

```
typedef p_vertices<util::graph, F> pv_t;
pv_t pv(g, f);
```

Thanks to the *p_vertices* there is now a mapping between vertices and sites. We may want to map data to it. The idea is to provide a function which returns the associated data according to the site given as parameter. Combining this function and the *p_vertices*, we get an image which can be used with algorithms and *for_all* loops.

```
template <typename S>
struct viota_t : public mln::Function_v2v< viota_t<S> >
{
  typedef unsigned result;

  viota_t(unsigned size)
  {
    v_.resize(size);
    for(unsigned i = 0; i < size; ++i)
      v_[i] = 10 + i;
  }

  unsigned
  operator()(const mln_psite(S)& p) const
  {
    return v_[p.v().id()];
  }

  protected:
    std::vector<result> v_;
};
```

```
// Constructs an image
viota_t<pv_t> viota(pv.nsites());
mln_VAR(graph_vertices_ima, viota | pv);
```

```
//Prints each vertex and its associated data.
mln_piter_(graph_vertices_ima_t) p(graph_vertices_ima.domain());
for_all(p)
  std::cout << "graph_vertices_ima(" << p << ") = "
              << graph_vertices_ima(p) << std::endl;
```

Output:

```
graph_vertices_ima((0,0)) = 10
graph_vertices_ima((2,2)) = 11
graph_vertices_ima((0,4)) = 12
graph_vertices_ima((4,3)) = 13
graph_vertices_ima((4,4)) = 14
```

Note that like any image in Olena, graph images share their data. Therefore, while constructing a graph image from a graph and a function, the graph is not copied and this is NOT a costly operation.

Of course, creating a graph image is not necessary and you can work directly with the graph and container/function mapping sites and data.

```
// Function which maps sites to data.
viota_t viota(g.v_nmax());

// Iterator on vertices.
mln_vertex_iter_(util::graph) v(g);

// Prints each vertex and its associated value.
for_all(v)
  std::cout << v << " : " << viota(v) << std::endl;
```

Output:

```
0 : 10
1 : 11
2 : 12
3 : 13
4 : 14
```

Graphs have iterators like any other site sets and also provide specific iterators in order to iterate over graphs in a more intuitive way.

Iteration over the adjacent edges of all the vertices:

```
// Iterator on vertices.
mln_vertex_iter_(util::graph) v(g);

// Iterator on v's edges.
mln_vertex_nbh_edge_iter_(util::graph) e(v);

// Prints the graph
// List all edges for each vertex.
```

```
      for_all(v)
      {
        std::cout << v << " : ";
        for_all(e)
          std::cout << e << " ";
        std::cout << std::endl;
      }
```

Output:

```
0 : (0,1)
1 : (0,1) (1,2) (1,3)
2 : (1,2) (2,4)
3 : (1,3) (3,4)
4 : (3,4) (2,4)
```

Iteration over the adjacent edges of all the edges:

```
      // Iterator on edges.
      mln_edge_iter_(util::graph) e(g);

      // Iterator on edges adjacent to e.
      mln_edge_nbh_edge_iter_(util::graph) ne(e);

      // Prints the graph
      // List all adjacent edges for each edge.
      for_all(e)
      {
        std::cout << e << " : ";
        for_all(ne)
          std::cout << ne << " ";
        std::cout << std::endl;
      }
```

Output:

```
(0,1) : (1,2) (1,3)
(1,2) : (0,1) (1,3) (2,4)
(1,3) : (0,1) (1,2) (3,4)
(3,4) : (1,3) (2,4)
(2,4) : (1,2) (3,4)
```

Iteration over the adjacent vertices of all the vertices:

```
      // Iterator on vertices.
      mln_vertex_iter_(util::graph) v(g);

      // Iterator on vertices adjacent to v.
      mln_vertex_nbh_vertex_iter_(util::graph) nv(v);

      // Prints the graph
```

```
    // List all adjacent edges for each edge.
    for_all(v)
    {
      std::cout << v << " : ";
      for_all(nv)
        std::cout << nv << " ";
      std::cout << std::endl;
    }
```

Output:

```
0 : 1
1 : 0 2 3
2 : 1 4
3 : 1 4
4 : 3 2
```

Chapter 13

Useful global variables

Name	Description	Possible values
literal::zero	Generic zero value. Can be used with various types such as algebra::vec, dpoint...	n.a.
literal::one	Generic one value. Can be used with various types such as algebra::vec, dpoint...	n.a.
literal::origin	Generic value for the origin point on a grid.	n.a.
border::thickness	Set the default border thickness of images	$[0 - UINT\_MAX]$
trace::quiet	Enable trace printing	true/false

Chapter 14

Useful macros

14.1 Variable declaration macros

Name	Arguments	Description
mln_VAR(N, V)	N : name of the variable.	
	V : value to assign to N.	Declare a variable N of type N_t and assign the value V.
mln_const_VAR(N, V)	N : name of the variable.	
	V : value to assign to N.	Declare a const variable N of type N_t and assign the value V.

14.2 Iterator type macros

14.2.1 Default iterator types

Name	Arguments	Description
mln_eiter(T)	T : iterable container type	Type of the element iterator of T
mln_niter(T)	T : iterable container/Image type	Type of the neighborhood iterator of T
mln_piter(T)	T : iterable container/image type	Type of the site iterator
mln_qiter(T)	T : iterable container/image type	Type of the window neighbors iterator of T
mln_viter(T)	T : iterable value container type	Type of the value iterator of T
mln_pixter(I)	I : image	Type of the pixel iterator of I
mln_qixter(I, W)	I : image type,	
	W : window Type	Type of the pixel iterator of a window on an image of type I.
mln_nixter(I, N)	I : image type,	
	N : neighborhood type	Type of the pixel iterator of a neighborhood on an image of type I.

14.2.2   Forward iterator types

Name	Arguments	Description
mln_fwd_eiter(T)	T : iterable container type	Type of the element forward iterator of T
mln_fwd_niter(T)	T : iterable container/Image type	Type of the neighborhood forward iterator of T
mln_fwd_piter(T)	T : iterable container/image type	Type of the site forward iterator
mln_fwd_qiter(T)	T : iterable container/image type	Type of the window neighbors forward iterator of T
mln_fwd_viter(T)	T : iterable value container type	Type of the value forward iterator of T
mln_fwd_pixter(I)	I : image	Type of the pixel forward iterator of I
mln_fwd_qixter(I, W)	I : image type,	
	W : window Type	Type of the pixel forward iterator of a window on an image of type I.
mln_fwd_nixter(I, N)	I : image type,	
	N : neighborhood type	Type of the pixel forward iterator of a neighborhood on an image of type I.

14.2.3 Backward iterators

Name	Arguments	Description
mln_bkd_eiter(T)	T : iterable container type	Type of the element backward iterator of T
mln_bkd_niter(T)	T : iterable container/Image type	Type of the neighborhood backward iterator of T
mln_bkd_piter(T)	T : iterable container/image type	Type of the site backward iterator
mln_bkd_qiter(T)	T : iterable container/image type	Type of the window neighbors backward iterator of T
mln_bkd_viter(T)	T : iterable value container type	Type of the value backward iterator of T
mln_bkd_pixter(I)	I : image	Type of the pixel backward iterator of I
mln_bkd_qixter(I, W)	I : image type,	
	W : window Type	Type of the pixel backward iterator of a window on an image of type I.
mln_bkd_nixter(I, N)	I : image type,	
	N : neighborhood type	Type of the pixel backward iterator of a neighborhood on an image of type I.

14.2.4 Graph iterators

Name	Arguments	Description
mln_vertex_iter(G)	G : graph type	Iterator on vertices.
mln_edge_iter(G)	G : graph type	Iterator on edges.
mln_vertex_nbh_edge_iter(G)	G : graph type	Iterator on the edges adjacent to a vertex.
mln_vertex_nbh_vertex_iter(G)	G : graph type	Iterator on the vertices adjacent to a vertex.
mln_edge_nbh_edge_iter(G)	G : graph type	Iterator on the edges adjacent to an edge.

Chapter 15

Common Compilation Errors

In this section, the most common compilation errors are gathered and explained.

- **error: 'check' is not a member of 'mln::metal::not_equal<bool, bool>**
  error: 'check' is not a member of 'mln::metal::converts_to<mln::value::rgb<8u>, unsigned int>'

  The routine does not support a given image with such a value type or an automatic conversion from the image value type to the expected value type is not available.

- **error: using 'typename' outside of template**

  Macros like *mln_site* or *mln_domain* can only be used in templated functions. In order to use them in a non-templated function, a '_' must be appended to the macro name. For instance : *mln_site_* and *mln_domain_*.