A Modern C++ Library for Generic and Efficient Image Processing

Thierry Géraud & Edwin Carlinet

firstname.lastname@lrde.epita.fr

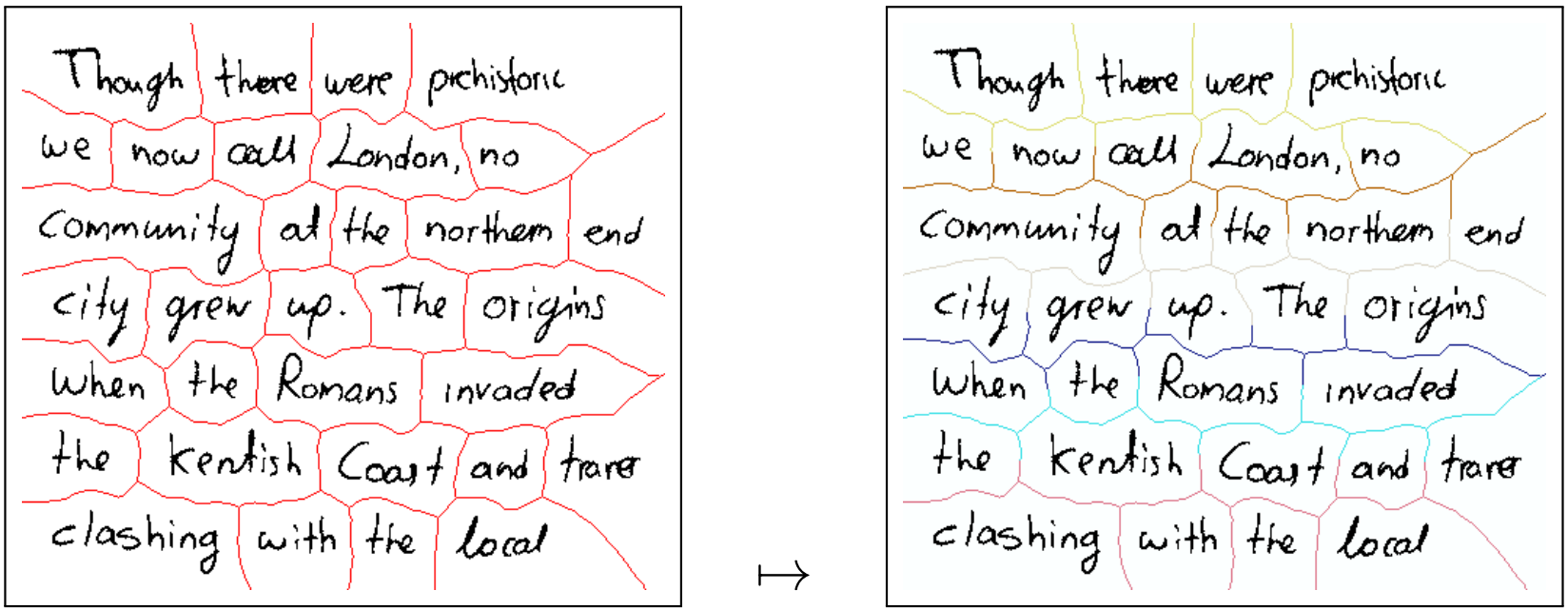
EPITA Research & Development Laboratory (LRDE)

2018/06/20 — GT GDMM — Lyon, France

Forewords

A common use case

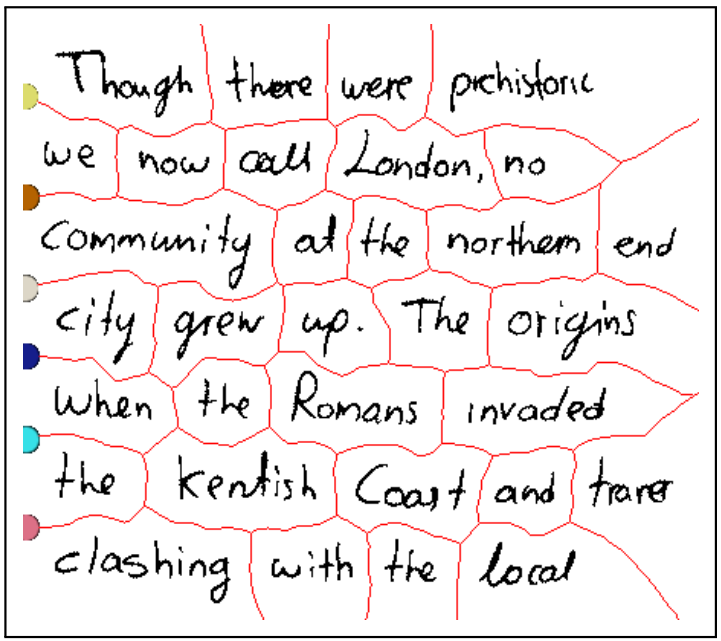
You need to do some image processing stuff, for example:



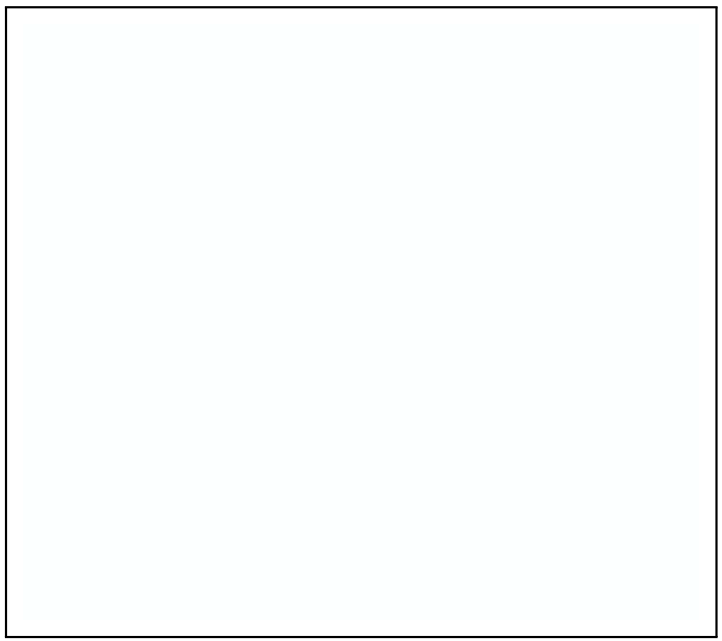
we want to identify / split the lines of text

A common use case

Idea: first label the red points of the 1st column. . .

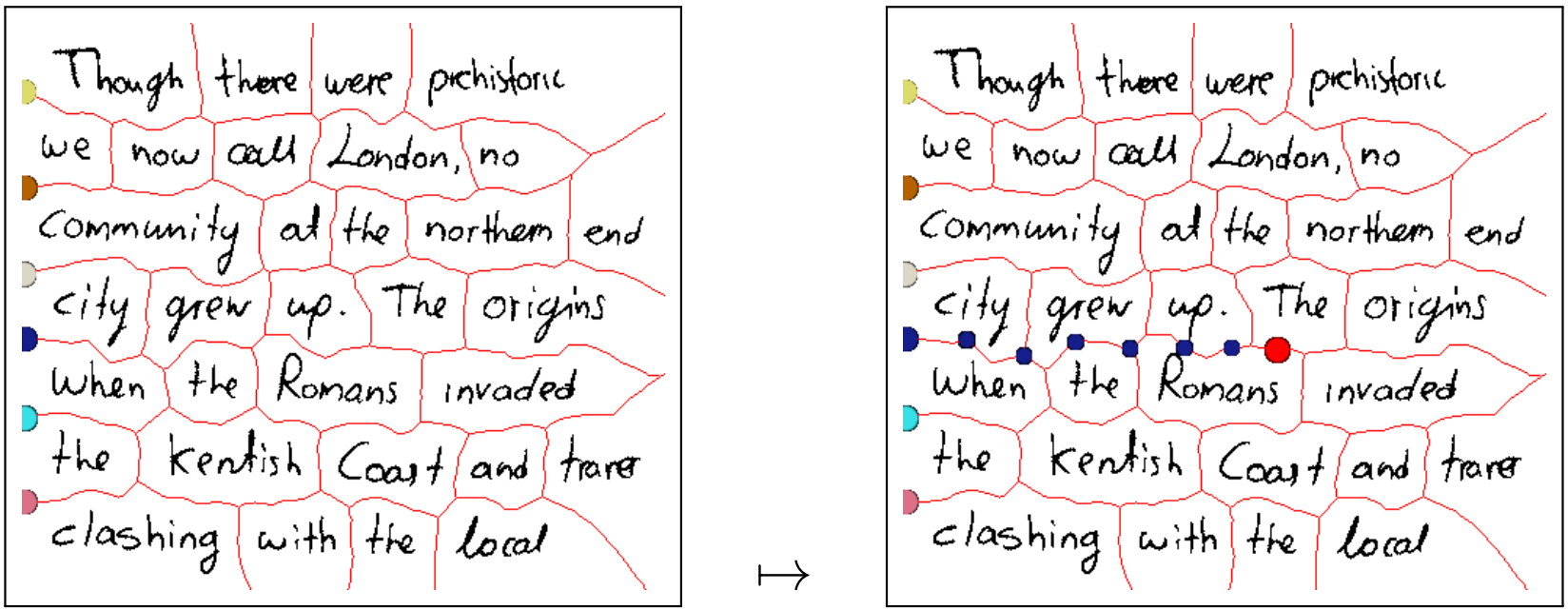


$\mapsto$



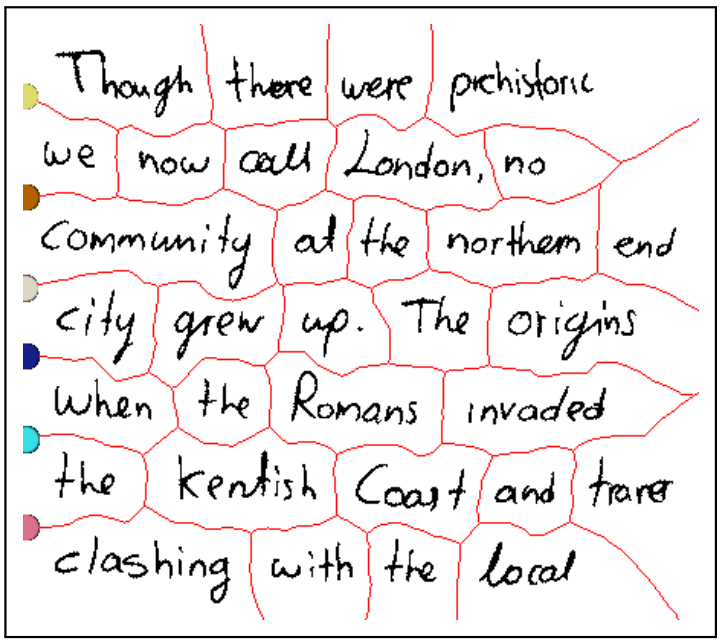
A common use case

. . . then think about computing the *influence zone* :

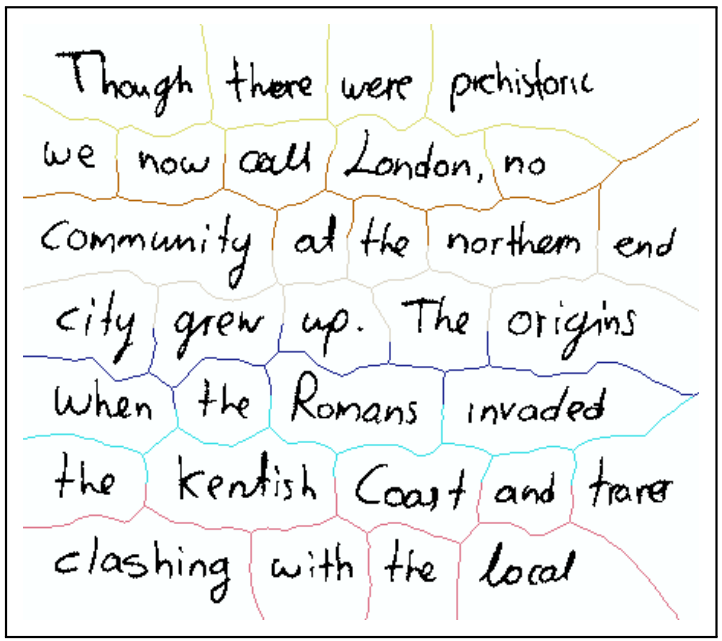


A common use case

yet, labeling the **set of red points** means. . .



$\mapsto$



. . . running an algorithm on **a non-rectangular domain**

A common use case

Can you do it?

Possible answers:

-  I just cannot do it.
-  Maybe I can do it but I'm not too sure about that...
-  I can do it but... I need to copy-paste-modify some code.
-  I can do it; yet it will take many time...

Preferred answer:

-  I can do it in a very few lines lines of code.

A common untruth

If you need to process some particular images:

- parts of images, whatever their shape; e.g. non-rectangular...
- images with uncommon value types; e.g. functions, tensors...
- images with a particular geodesy; e.g. cylinder, torus...
- 3D images
- videos; e.g. 2D+t, and why not 3D+t, or "whatever+t"
- graphs; e.g. vertex-valued, edge-valued...
- meshes
- complexes
- ...

it is **not** normal that your tool **cannot** do it

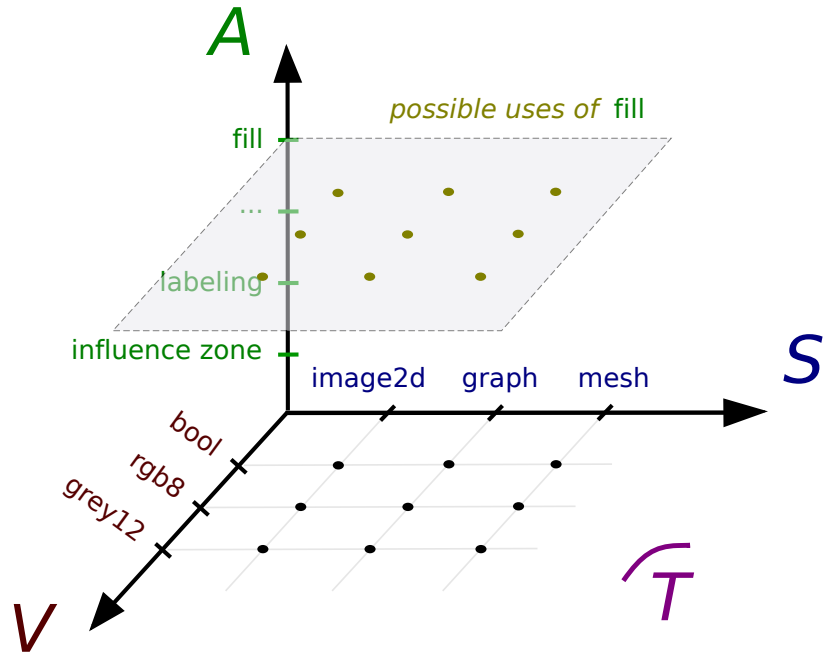
From Milena to Pylena

Key ideas of the Milena library

- Context: MM then DT (. . . DG) + software engineering & C++
- Dedicated to image processing, including many MM tools
- An image is a *function*: $p \in \mathcal{D} \longmapsto f(p) \in \mathcal{V}$
- You can browse the domain $\mathcal{D}$
  . . . and access to the pixel values $f(p)$
- Dummy (non-generic) sample uses:

```
for (row = 0; row < ima.nrows(); row += 1)
  for (col = 0; col < ima.ncols(); col += 1)
    sum += ima.at(row, col);

point2d p(16, 64);
int_u8 v;
if (ima.domain().has(p))
  v = ima(p);
```

Genericity: key feature of Milena

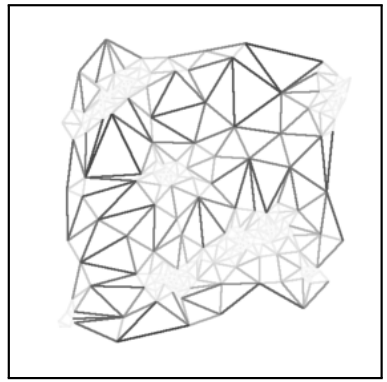
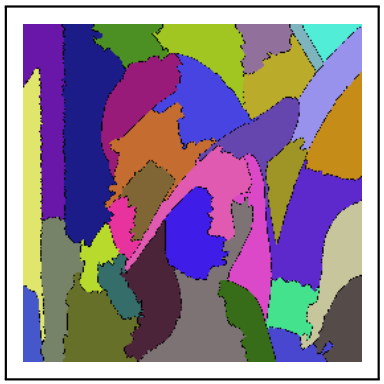
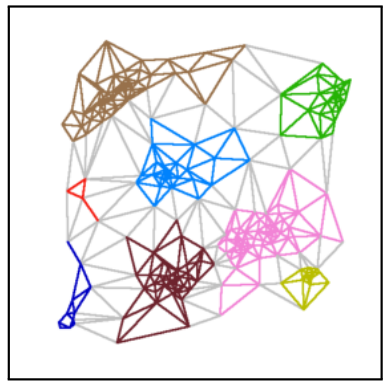
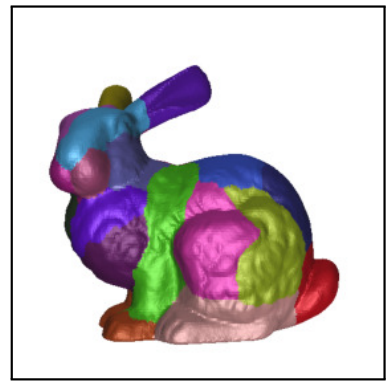


Being generic means:

- Code once an algorithm
- Run on many input types

  Not to be limited = covering *as much as possible* the space of possibilities.
- Specialize this algorithm
  when a better version exists for some particular types

Genericity: an example

	image 2D	graph	mesh
input:			
output:			

The same code run on all these input.

Genericity: How-to (tour of programming paradigms)

Acron	*Meaning*
TC	Type checking
CS	Code simplicity
E	Efficiency
1IA	One implementation per algorithm
EA	Explicit abstractions / Constrained genericity

Paradigms	TC	CS	E	1IA	EA
Code Duplication	✓	✗	✓	✗	✗
Generalization	✗	≈	≈	✓	✗
Object-Orientation	≈	✓	✗	✓	✓
Generic Programming:					
with C++11	✓	≈	✓	✓	≈
with C++17	✓	✓	✓	✓	≈
with C++20	✓	✓	✓	✓	✓

Genericity: a comparison

Morphological dilation:

$$\forall p,\ \delta(f)(p) = \vee_{q \in B_p} f(q)$$

Many libraries implement this operator:

OpenCV	Gimp	Matlab	ITK	Milena	Pylena ← new!
477	457	?	92	38	13 lines
dup	dup	gen°	GP	GP	GP

which is *only* a double loop...

History

2000	2010	2011	2018	...	
Milena...		Pylena...			
C++98	C++03	pre-C++11	C++17	and ready for	C++20

Benefits from Milena

We have:

- Gain experience on generic programming
- Solved a lot of open questions about design
- Identified some awkward choices
- A large catalog of generic algorithms

old C++ (up to C++03) $\neq$ *modern* C++ (from C++11)

Objectives of Pylena

For the simple user:

- Python bindings with *numpy* (transparent, and both ways)

For the IP practitioner:

- A generic simplified dev framework
- A greater flexibility

For the engineer:

- The best performances (0-cost abstraction)

vs

Issues of Milena we do **not** have anymore *thanks to modern C++*:

- some loss of efficiency
- complex internals (even for the expert)
- user limited by the C++ language complexity
- ugly error messages at compile-time

Advertisement

...

Advertisement (*not in the presentation*!)

Interested to:

- share information
- discuss
- follow what's happening in the community
- not to do the same thing than your colleague

about when *mathematical morphology* and *neural networks* meet?

Subscribe to:
`https://lists.lrde.epita.fr/listinfo/morphonet`

. . .

Edwin's part

A solution to Simplicity, Genericity, and Efficiency

Through an example

Alpha-blending between two images:

$0.2 \times$  $+ \ 0.8 \times$  $\rightarrow$ 

ima1 ima2 ima2 after

In C++:

```
blend_inplace(ima1, ima2, 0.2);  // ima2 is modified
```

- An old-style C/C++ efficient library (e.g. Intel IPP) would write:

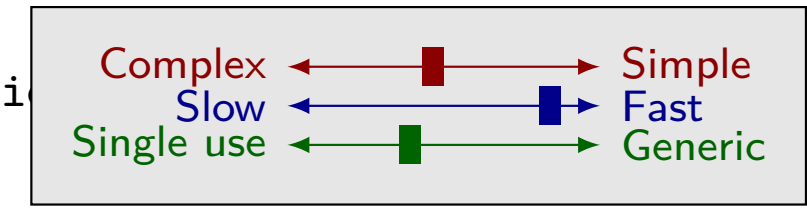
```
void blend_inplace(const uint8_t* ima1, uint8_t* ima2, float alpha,
                   int width, int height, int stride1, int stride2)
{
  for (int y = 0; y < height; ++y)
  {
    const uint8_t* iptr = ima1 + y * stride1;
    uint8_t* optr = ima2 + y * stride2;
    for (int x = 0; x < width; ++x)
      optr[x] = iptr[x] * alpha + optr[x] * (1-alpha);
  }
}
```

- A modern, user-friendly, Python library would write:

```
def blend(ima1, ima2, alpha):
    return alpha * ima1 +  (1-alpha) * ima2;
```

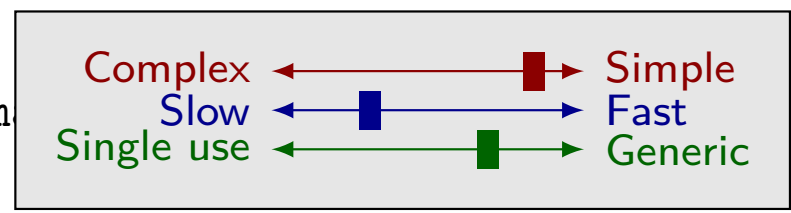
- An old-style C/C++ efficient library (e.g. Intel IPP) would write:

```
void blend_inplace(const uint8_t* ima1, uint8_t* ima2, float alpha,
                   int width, int height, int stride1, int stride2)
{
  for (int y = 0; y < height; ++y)
  {
    const uint8_t* iptr = ima1 + y * stri
    uint8_t* optr = ima2 + y * stride2;
    for (int x = 0; x < width; ++x)
      optr[x] = iptr[x] * alpha + optr[x] * (1-alpha);
  }
}
```



- A modern, user-friendly, Python library would write:

```
def blend(ima1, ima2, alpha):
    return alpha * ima1 +  (1-alpha) * im
```



Handling differents input types

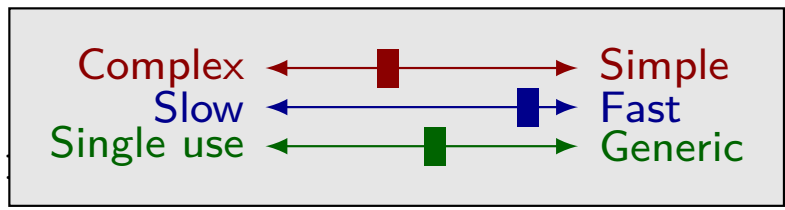
What about an HDR image (16-bit or float)?

Handling differents input types

What about an HDR image (16-bit or float)?

One just have to "template" the type of pixel values

```
template <class V> // whatever the type V of pixel values
void blend_inplace(const V* ima1, V* ima2, float alpha /*...*/)
{
  for (int y = 0; y < height; ++y)
  {
    const auto* iptr = ima1 + y * stride1;
    auto* optr = ima2 + y * stride2;
    for (int x = 0; x < width; ++x)
      optr[x] = iptr[x] * alpha + optr[x] * (1-alpha);
  }
}
```



In Python, the function remains the same

Handling Regions of Interest

What if we want to deal with a sub-part of the image?

Handling Regions of Interest

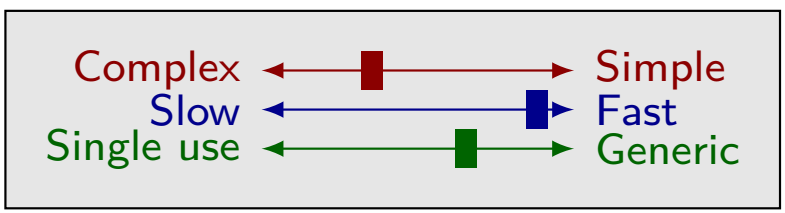
What if we want to deal with a sub-part of the image?

- Pass a "ROI" object as a **new** argument
- Pass a "mask" object as a **new** argument

Handling ROIs

- With a `Rect2d` object

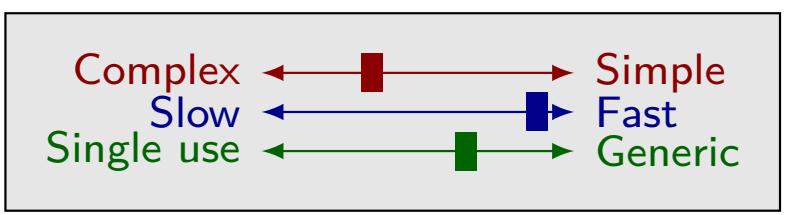
```
template <class V>
void blend_inplace(const V* ima1, V* ima2, float alpha,
                   Rect2d roi /*...*/)
{
  for (int y = roi.y; y < roi.yend; ++y) // NOW: use 'roi'
  {
    const auto* iptr = ima1 + y * stride1;
    auto* optr = ima2 + y * stride2;
    for (int x = roi.x; x < roi.xend; ++x) // NOW: use 'roi'
      optr[x] = iptr[x] * alpha + optr[x] * (1-alpha);
  }
}
```



Handling ROIs

- With a mask object

```
template <class V>
void blend_inplace(const V* ima1, V* ima2, float alpha,
                   const bool* mask /*...*/)
{
  // ...
  for (int y = 0; y < height; ++y)
  {
    const auto* iptr = ima1 + y * stride1;
    auto* optr = ima2 + y * stride2;
    const bool* mmptr = mask + y * stride3;
    for (int x = 0; x < width; ++x)
      if (mmptr[x]) // NEW: test on the mask
        optr[x] = iptr[x] * alpha + optr[x] * (1-alpha);
  }
}
```



Handling ROIs in Python

In Python, the implementation remains the same:

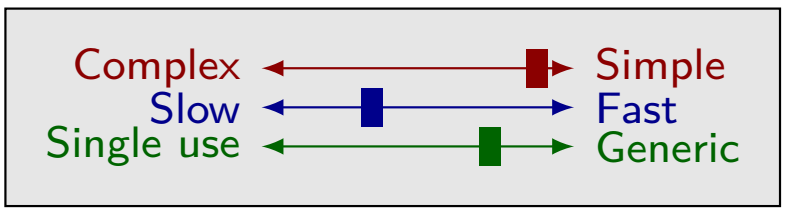
- With a ROI

```
res = blend(ima1[y:yend,x:xend], ima2[y:yend,x:xend], alpha)
```

- With a Mask

```
res = blend(ima1[M], ima2[M], alpha)
```

In Python, only the calls change.



Handling channels

What if we want to process:

1. only the red channel of an RGB image?
2. an image encoded by plane (RRR...GGG...BBB...)?

Again, duplicate the implementation (*that is pure evil!*)

Handling channels

What if we want to process:

1. only the red channel of an RGB image?
2. an image encoded by plane (RRR...GGG...BBB...)?

Again, duplicate the implementation (*that is pure evil!*)

In C, new versions (with vectorial data handling + channel + layout + ...)

In python, we would write:

```
ima2[:,:,0] = blend(ima1[:,:,0], ima2[:,:,0], alpha)  # 1
ima2[0,:,:] = blend(ima1[0,:,:], ima2[0,:,:], alpha)  # 2
```

Chaining processings

If I want a gamma correction before blending?

Do it in two steps through an intermediate image.
(hum... who cares about performances and memory anyway...)

Chaining processings

If I want a gamma correction before blending?

Do it in two steps through an intermediate image.
(hum... who cares about performances and memory anyway...)

What if the image is 3D?

That's a new algorithm! (Code, code again, and again...)

Chaining processings

If I want a gamma correction before blending?

Do it in two steps through an intermediate image.
(hum... who cares about performances and memory anyway...)

What if the image is 3D?

That's a new algorithm! (Code, code again, and again...)

What if the image is a graph, a mesh, a complex?

Stop!

Genericity = Versatility = Simplicity

What if the simplicity of Python was possible in C++?

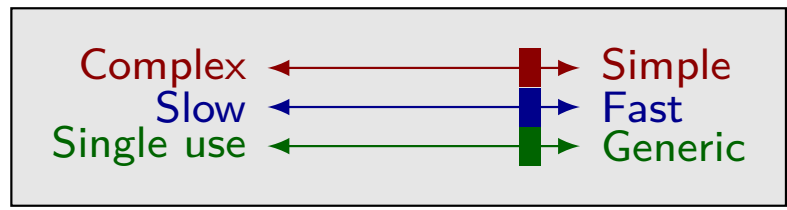
```
template <class I1, class I2> // whatever the input types
void blend_inplace(const I1& ima1, I2&& ima2, float alpha)
{
    auto zz = imzip(ima1, ima2);
    for (auto&& [v1, v2] : zz.values())
        v2 = v1 * alpha + v2 * (1 - alpha);
}
```

- `imzip`:
  Create an image which maps $p \mapsto (ima1(p), ima2(p))$
- `for`:
  Iterate on all values `(v1, v2)` of the pixels of `(ima1, ima2)`

Genericity = Versatility = Simplicity

What if the simplicity of Python was possible in C++?

```
template <class I1, class I2> // whatever the input types
void blend_inplace(const I1& ima1, I2&& ima2, float alpha)
{
    auto zz = imzip(ima1, ima2);
    for (auto&& [v1, v2] : zz.values())
        v2 = v1 * alpha + v2 * (1 - alpha);
}
```



- The pixel type does not appear in the code
- The layout neither (*no need to care about impl. details*)
- High-level access to the image contents (*abstraction with Ranges & Iterators*)

Genericity = Versatility = Simplicity

What if the image

- is HDR (16-bit or float)?
- is encoded by plane?
- is 3D?

Does not change anything, it simply works :)

Genericity = Versatility = Simplicity

What if the image
- is HDR (16-bit or float)?
- is encoded by plane?
- is 3D?

Does not change anything, it simply works :)

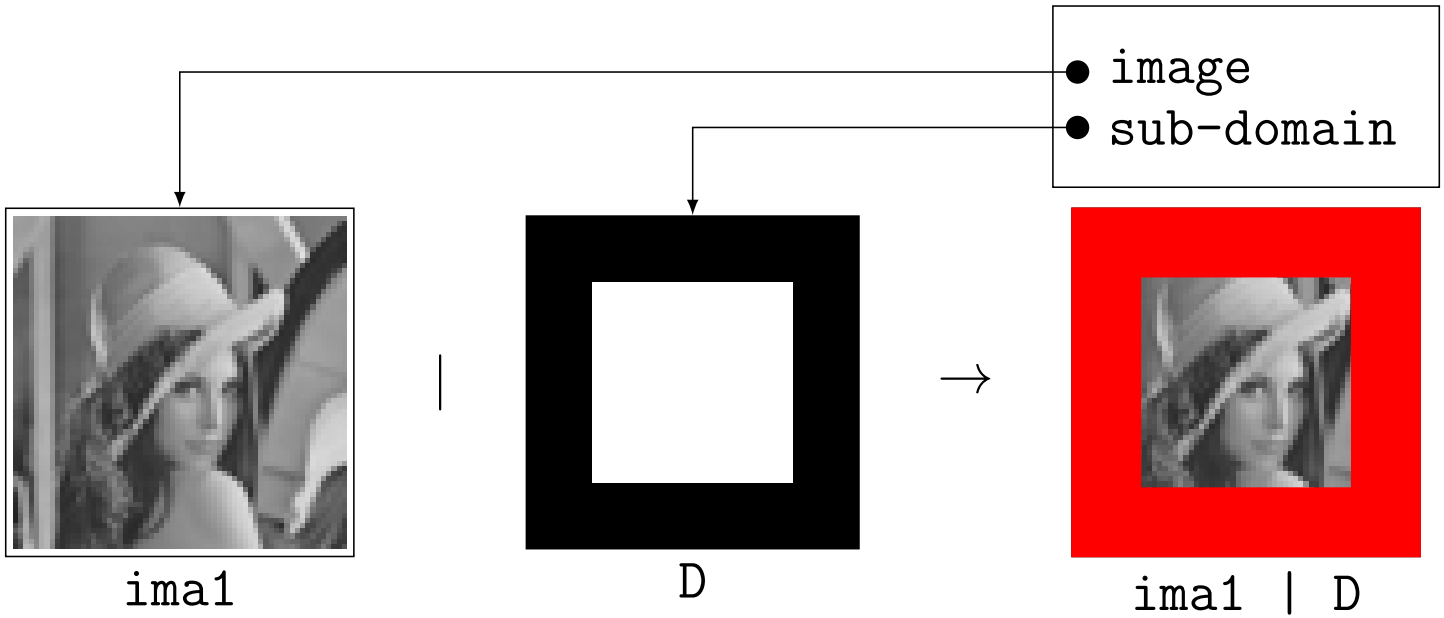
What if we want to process
- only a ROI?
- only the red channel?

Do not change the algorithm,
change the inputs (like in Python)

Pylena uses **views** introduced in Milena, allowing *lazy-evaluation*, *efficiency*, and *higher genericity* :-)

Processing a ROI

```
auto D = box2d{{10, 10}, {54, 54}};
```

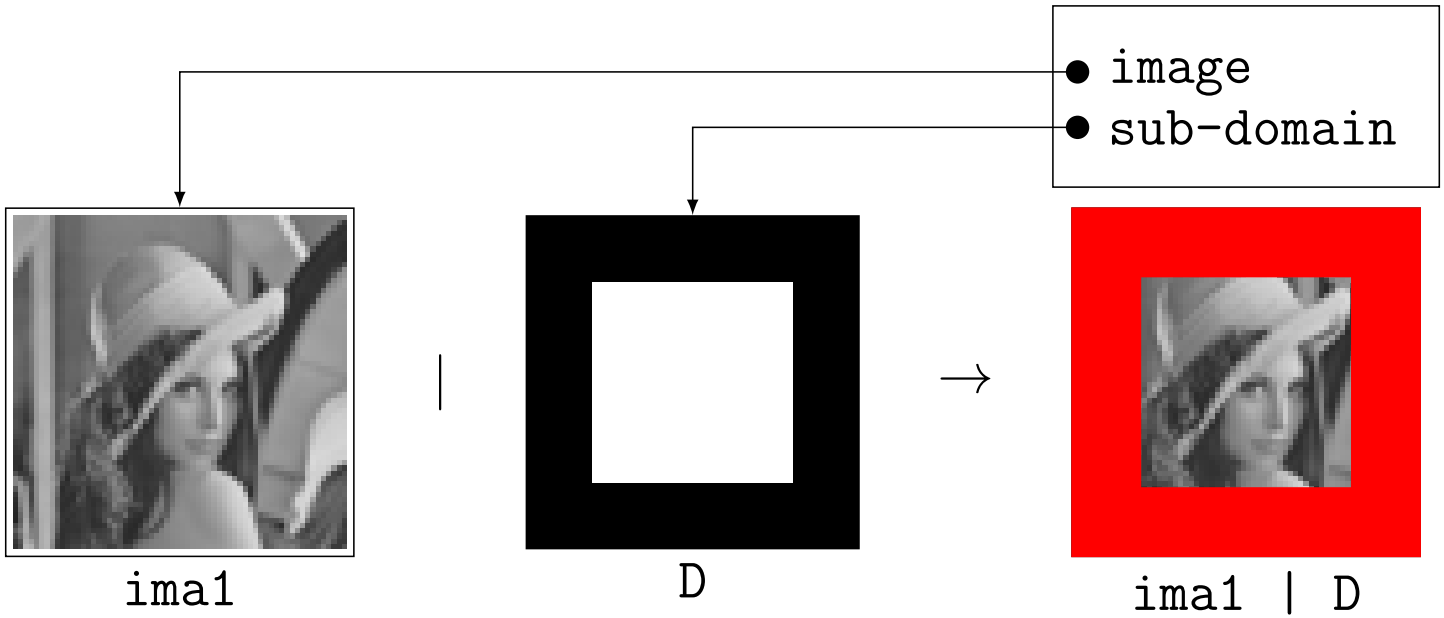


white = true, black = false, red = outside 'D'

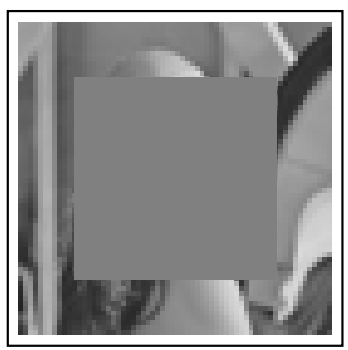
- `ima1 | D` reads "my image restricted to the sub-domain `D`"
- it is a lightweight object:
  - no new allocation
  - "points to" existing data
  - gives a particular *view* of `ima1`

Processing a ROI

```
auto D = box2d{{10, 10}, {54, 54}};
```



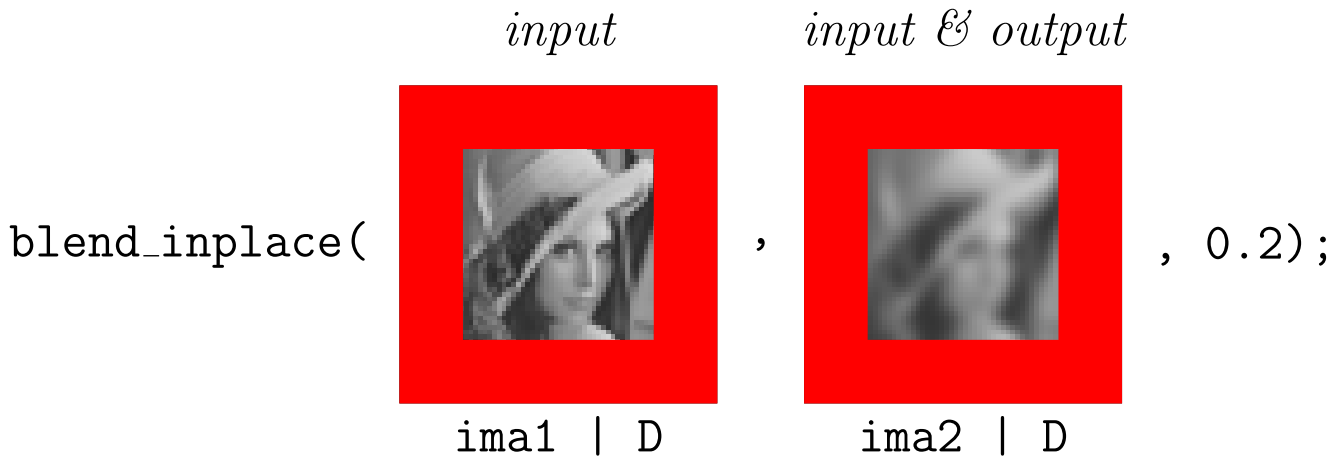

```
fill(ima1 | D, 128);
```



ima1 after

Processing a ROI defined by a sub-domain

The line:

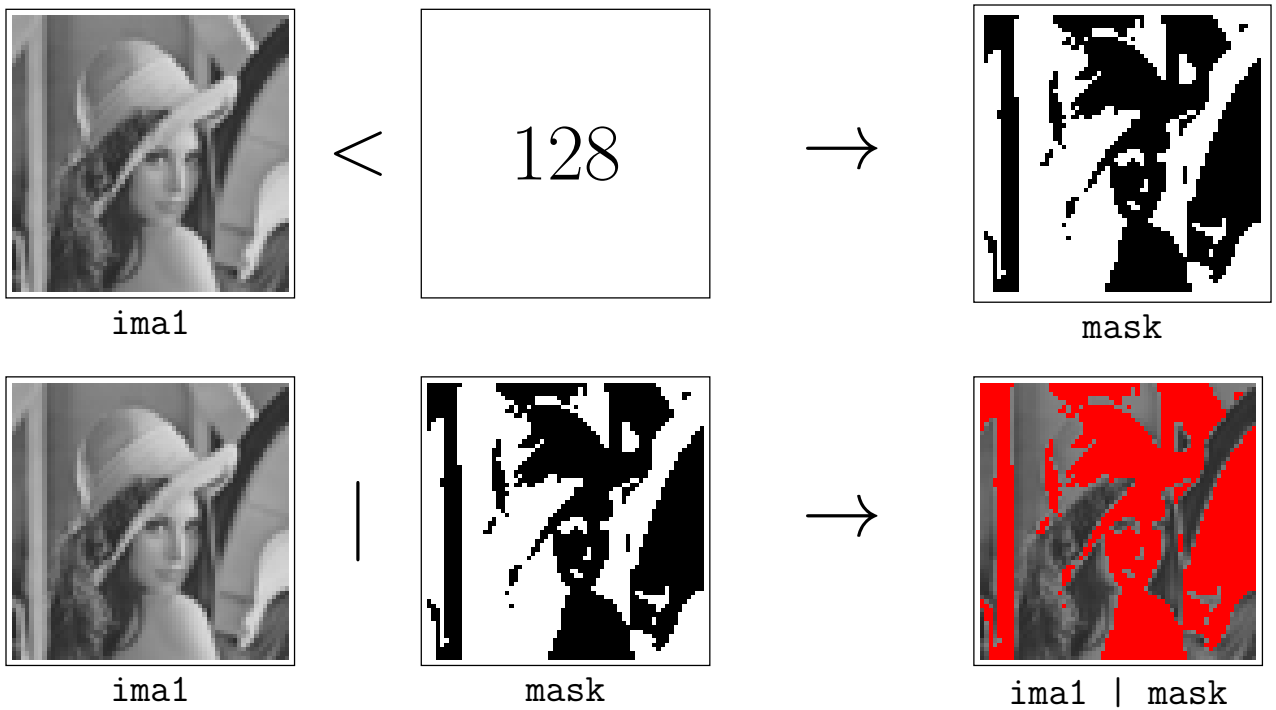


gives:

ima2

Processing a ROI (low-light regions) defined by a mask

```
auto mask = ima1 < 128; // lightweight image
blend_inplace(ima1 | mask, ima2 | mask, alpha);
```

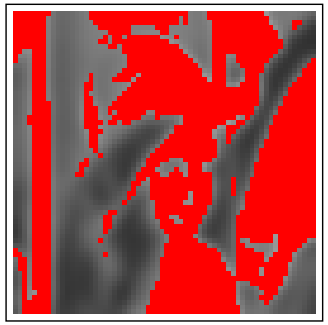


white = true, black = false, red = outside the domain

Processing a ROI defined by a mask

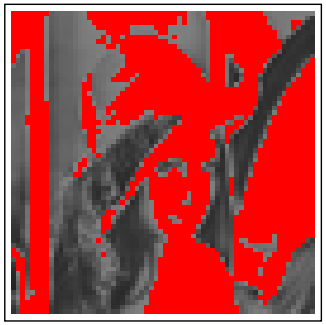
The line:

```
blend_inplace(
```



ima2 | mask

```
,
```



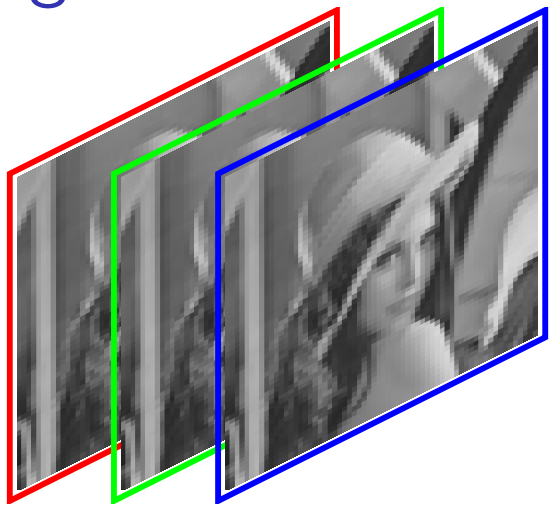
ima1 | mask

```
, 0.8);
```

gives:

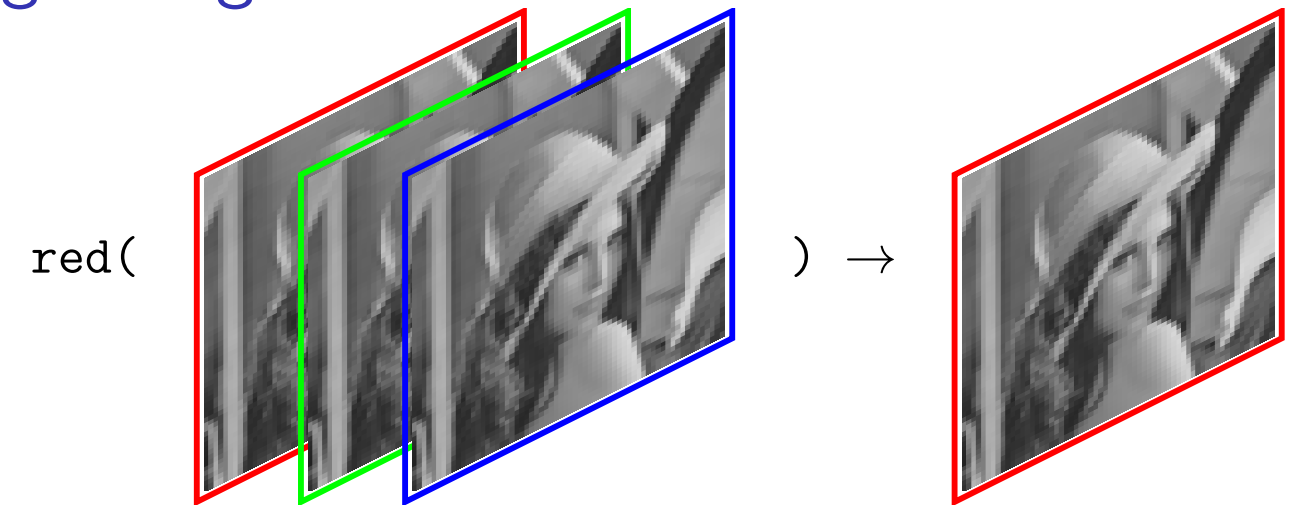
ima1

Processing a single channel

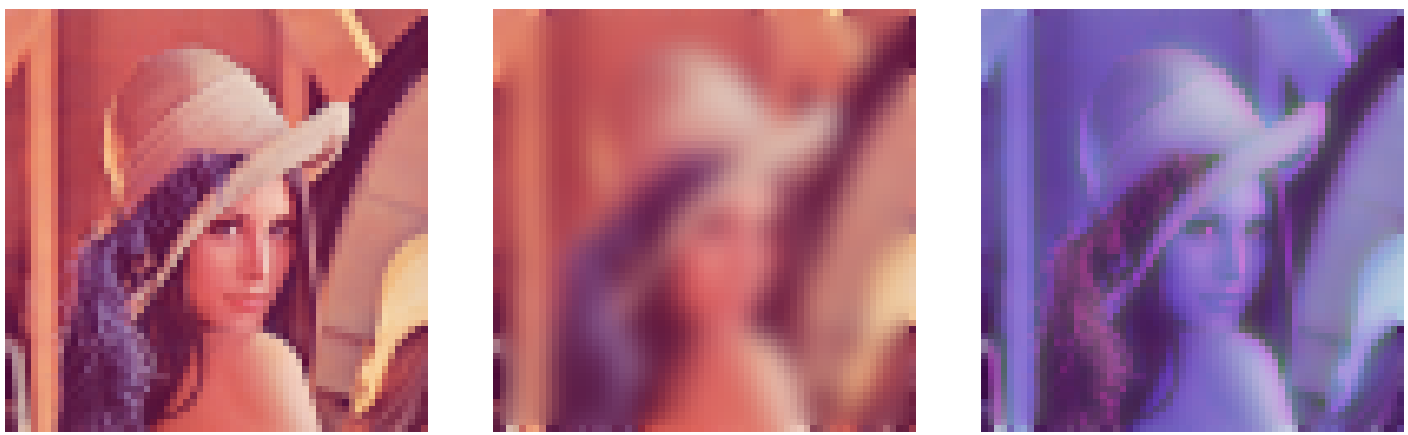
red() → 

```
blend_inplace(red(ima1), blue(ima2), 0.8);
blend_inplace(blue(ima1), red(ima2), 0.8);
```

Processing a single channel

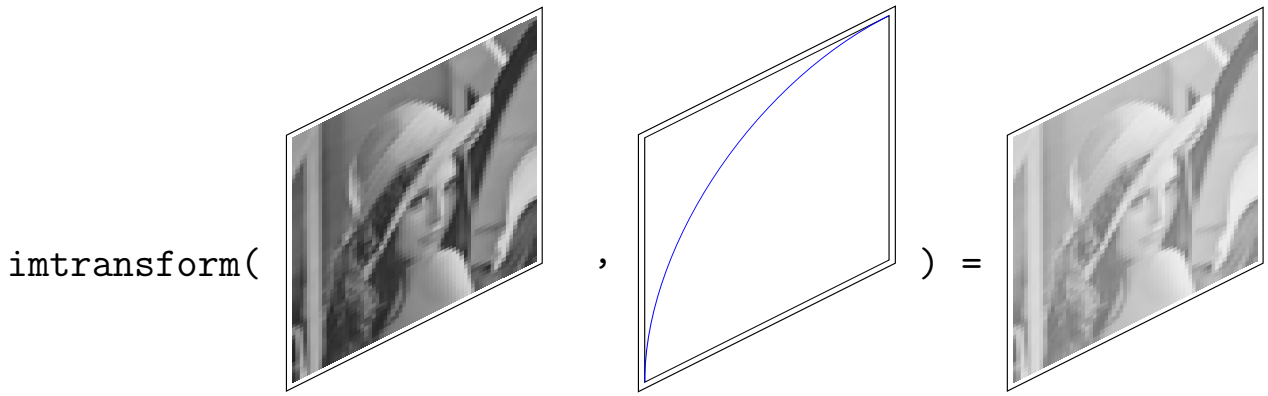



```
blend_inplace(red(ima1), blue(ima2), 0.8);
blend_inplace(blue(ima1), red(ima2), 0.8);
```



ima1	ima2 before	ima2 after

Chaining processings



Gamma correction before blending

```
constexpr float gamma = 2.5f;
auto gamma_fun = [](auto x) { return std::pow(x, 1/gamma); });
auto f = imtransform(ima1, gamma_fun);
...
```

Views = Versatility + Performance

5 fundamental image views:

View	*Function*	Domain
`g = imzip(`f_1`,`f_2`,...,`f_n`)`	$p \mapsto (f_1(p), ..., f_n(p))$	$\mathcal{D}_{f_*}$
`g = imtransform(f, F)`	$p \mapsto F(f(p))$	$\mathcal{D}_f$
`g = immorph(f, T)`	$p \mapsto f(T(p))$	$\mathrm{Im}_T(\mathcal{D})$
`g = f | D`	$p \mapsto f(p)$	$\mathcal{D} \subset \mathcal{D}_f$
`g = imfilter(f, pred)`	$p \mapsto f(p)$	$\{p \in \mathcal{D}_f \mid pred(p)\}$

Views = Versatility + Performance

5 fundamental image views:

View	*Function*	Domain
`g = imzip(`f_1`,`f_2`,...,`f_n`)`	$p \mapsto (f_1(p), ..., f_n(p))$	$\mathcal{D}_{f_*}$
`g = imtransform(f, F)`	$p \mapsto F(f(p))$	$\mathcal{D}_f$
`g = immorph(f, T)`	$p \mapsto f(T(p))$	$\mathrm{Im}_T(\mathcal{D})$
`g = f | D`	$p \mapsto f(p)$	$\mathcal{D} \subset \mathcal{D}_f$
`g = imfilter(f, pred)`	$p \mapsto f(p)$	$\{p \in \mathcal{D}_f \mid pred(p)\}$

The others are *just* "sugar":

```
auto red = [](auto f) {
  return imtransform(f, [](auto& v) { return v.red; };
};

template <class I, class J>
auto operator+(const I& f, const J& g) {
  return imtransform(imzip(f,g), std::plus<>());
}
```

Making `blend` a view

```
auto blend = [](auto f, auto g, float alpha) { // this is a lambda
  return alpha * f + (1 - alpha) * g;
};
```

Making `blend` a view

```
auto blend = [](auto f, auto g, float alpha) { // this is a lambda
  return alpha * f + (1 - alpha) * g;
};
```

Mission: after applying a gamma correction, blend `ima1` and `ima2` on low-light regions only, and save the result...

Making `blend` a view

```
auto blend = [](auto f, auto g, float alpha) { // this is a lambda
  return alpha * f + (1 - alpha) * g;
};
```

Mission: after applying a gamma correction, blend `ima1` and `ima2` on low-light regions only, and save the result...

```
auto G = [](auto x) { return std::pow(x, 1/2.5f); };
auto ima1G = imtransform(ima1, G);
auto ima2G = imtransform(ima2, G);
auto blended = blend(ima1G, ima2G, 0.3); // not inplace & lightweight
auto output = where(ima1 < 128, blended, ima1);
io::imsave(imcast<uint8>(output), "output.tif");
```

Making `blend` a view

```
auto blend = [](auto f, auto g, float alpha) { // this is a lambda
  return alpha * f + (1 - alpha) * g;
};
```

Mission: after applying a gamma correction, blend `ima1` and `ima2` on low-light regions only, and save the result...

```
auto G = [](auto x) { return std::pow(x, 1/2.5f); };
auto ima1G = imtransform(ima1, G);
auto ima2G = imtransform(ima2, G);
auto blended = blend(ima1G, ima2G, 0.3); // not inplace & lightweight
auto output = where(ima1 < 128, blended, ima1);
io::imsave(imcast<uint8>(output), "output.tif");
```

- Clean code: ✓
- Memory efficient code: ✓ (*no* allocation; lazy evaluation)
- High-performant code: ✓ (a single loop at *save* time)

Conclusion

Conclusion: Myths about genericity

- For the user

With a generic lib, one is forced to code in a generic way.

No, we have seen some examples...

If it's generic, it's much pain for the user.

No, with modern C++, it is not true anymore...

- About the library

To get a generic lib, just add the `template` keyword.

No, that's just a good start...

A library is either generic or not.

No, there's a degree of genericity from poor to high...

Conclusion: About C++ evolution

The differences between *old C++* and *modern C++*:

- for the user:
  - `templated` code is simple to use and write
  - way much better (readable) error messages
  - a high degree of genericity
  - type safety
- for the programmers of the library:
  - no more hardcore `template` metaprogramming
  - the compiler now does the painful work required to get the *highest possible genericity*

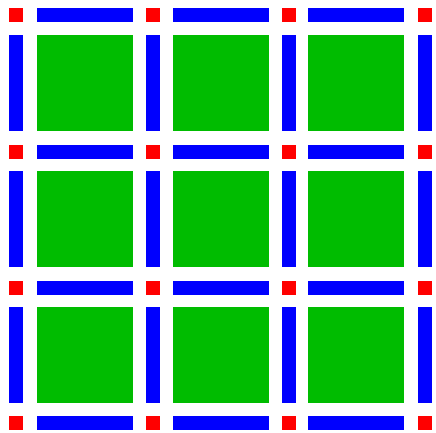
Conclusion: About Pylena

Pylena at a glance:

- featuring:
  - open source / free software
  - modern C++ (that rocks)
  - a fast learning curve (close to python)
- for image processing practitioners:
  - basic stuff + many adv structures
  - mathematical morphology
  - discrete topology
- for dummy users:
  - all the lib contents accessible through python
  - easy prototyping
- for engineers:
  - efficiency
  - documentation + test suite

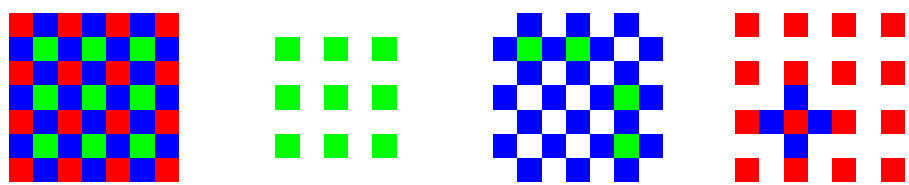
Conclusion: A tool for discrete topology

Left as an exercise to the reader :-)



- How to encode a 2D cubical complex with a simple 2D image?
- How to copy the values of a 2D image to the set of 2-faces?
- How to value the 1-faces and 0-faces to get an usc function?

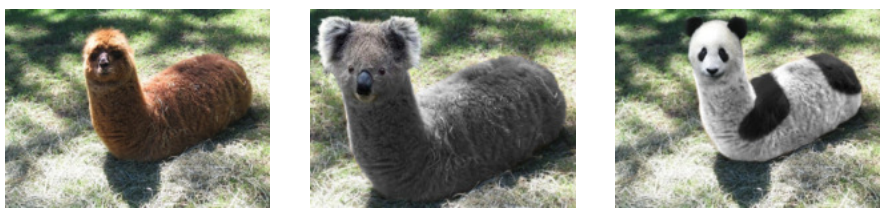
hints:



The end (or a start. . .)

URL: `https://gitlab.lrde.epita.fr/olena/pylene`

Thanks for your attention; any questions?



Genericity for maggots.

Extra materials

Sample code with Pylena (a-la Milena)

```
template <class I, class SE>
    // whatever the image type (I)
    // and the type of structuring element (SE)
mln_concrete(I) dilate(const I& f, const SE& se)
{
  auto g = f.concretize();
  auto supr = accu::supremum<mln_value(I)>();
  for (auto p : f.domain()) // for all p in f domain
  {
    for (auto q : se(p)) // for all q in se(p)
      supr.take(f(q));
    g(p) = supr.result();
  }
  return g;
}
```

Sample code with Milena

```
template <class I, class SE>
    // whatever the image type (I)
    // and the type of structuring element (SE)
mln_concrete(I) dilate(const I& f, const SE& se)
{
  mln_concrete(I) g;
  initialize(g, f);
  mln_piter(I) p(f.domain());
  mln_qiter(SE) q(se, p);
  for_all(p) // for all p in f domain
  {
    mln_value(I) v = f(p);
    for_all(q) // for all q in se(p)
      if (f.has(q) and f(q) > v)
        v = f(q);
    g(p) = v;
  }
  return g;
}
```

The *preferred* version with Pylena

```
template <class I, class SE>
mln_concrete(I) dilate(const I& f, const SE& se)
{
  auto g = f.concretize();
  auto supr = accu::supremum<mln_value(I)>();
  for (auto [f_px, g_px] : zip(f.pixels(), g.pixels()))
  {
    for (auto qx : se(f_px))
      supr.take(qx.val());
    g_px.val() = supr.result();
  }
  return g;
}
```

Most efficient code (vectorization, unrolled inner loop. . .)

main.cpp

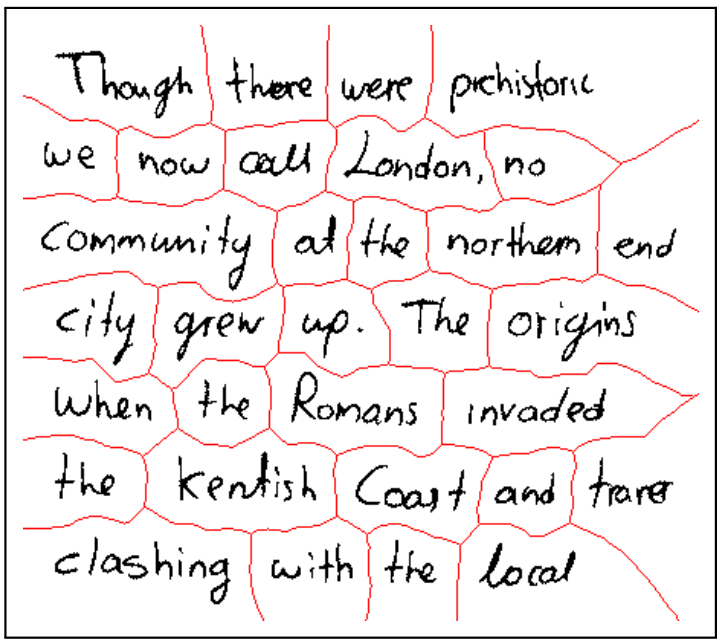
```cpp
#include <mln/core/image/image2d.hpp>
#include <mln/morpho/hit_or_miss.hpp>
#include <mln/io/imread.hpp>
#include <mln/io/imsave.hpp>

int main()
{
  using namespace mln;

  image2d<uint8> lena;
  io::imread("lena.pgm", lena);

  auto se_hit  = se::make({ { 0, -1}, { 0, 0}, { 0, 1} });
  auto se_miss = se::make({ {-1, -1}, {-1, 0}, {-1, 1},
                            {+1, -1}, {+1, 0}, {+1, 1} });

  auto out = morpho::hit_or_miss(lena, se_hit, se_miss);
  io::imsave(out, "out.pgm");
}
```

hit_or_miss.hpp

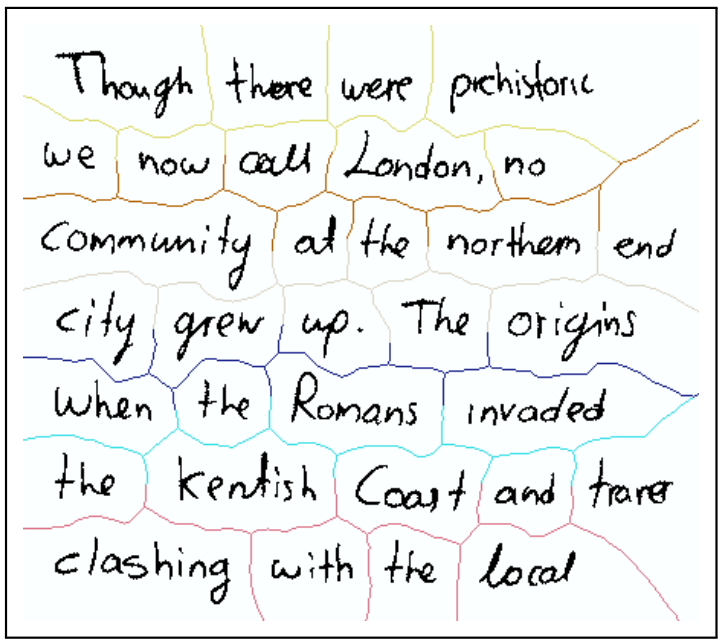
```
template <class I, class SEh, class SEm>
mln_concrete(I) hit_or_miss(const I& f,
                            const SEh& se_hit, const SEm& se_miss)
{
  static_assert(is_a<I,Image>::value, "1st arg shall be an Image");
  // ...
  auto ero = erode(f, se_hit);
  auto dil = dilate(f, se_miss);

  using V = typename I::value_type;
  auto res = where(dil < ero, ero - dil, V(literal::zero));
  // 'res' is a lightweight image (about no data)
  return eval(res);
}
```

A common use case (soluce)

You need to do some image processing stuff, for example:



$\mapsto$



A common use case (soluce)

```
auto mask = (ima == literal::red);
auto label = image2d<unsigned>{ima.domain(), 0};
unsigned nlabels;
copy(labeling::blobs(mask | column(0), // 1
                     c4, nlabels), // 2
     label); // 3
labeling::iz_inplace(label | mask, c8); // 4
```