Generic programming in modern C++ for Image Processing

Michaël Roynard Director: *Thierry Géraud* Supervisor: *Edwin Carlinet*

Thesis Defense — Friday 4th November, 2022

EPITA Research Laboratory (LRE) — Le Kremlin-Bicêtre, France

Overview

Introduction

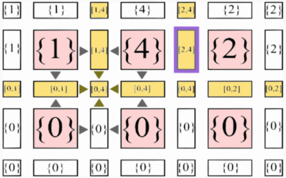
Image processing nowadays

- Every industry and research domain is using it (production line automation, automatic document reading, social network apps, etc.)
- Image processing is everywhere, in most of the devices (computer, phone, embedded devices, etc.)
- Applications are multiples (medical imaging, social network filters, facial recognition, etc.)
- Image Processing is costly in resources. Performance is crucial.

Image processing nowadays

Different data types and algorithms

- image-ND, hexagonal grid, cubical complexes value, etc.
- pixel-wise algorithms, local algorithms (convolution), global algorithms (propagation)



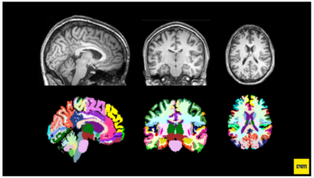


Image processing nowadays

Different user profiles and their use cases

- The end user (non-programmer, wants UI interface).
- The practitioner (end user of an image processing library).
- The contributor (advanced user of a library familiar with its internals).
- The maintainer (founder/creator of the library or took it over to make it grow).

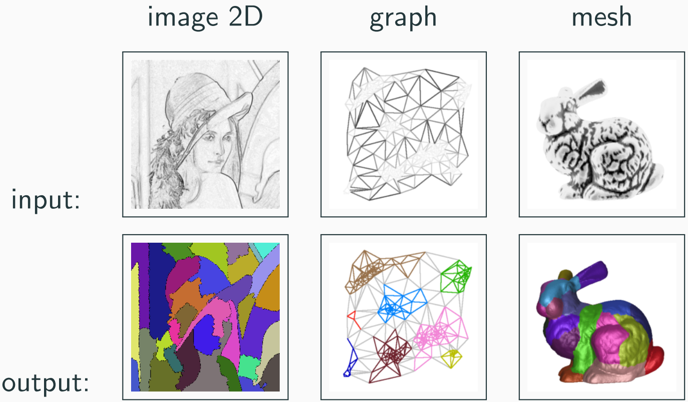
Our work is aimed toward the *practitioner*, the *contributor* and the *maintainer*.

Image processing nowadays

Different tools

- Graphic editors (GIMP, Photoshop).
- Command line utilities (ImageMagick, GraphicsMagick or MegaWave).
- Visual programming environment (Mathcad).
- Integrated environment (Matlab, Scilab, Octave, Mathematica and Jupyter).
- Package for Python (SciPy, NumPy, Scikit-image, Pillow or OpenCV bindings, via PyPi or Conda).
- Programming libraries (IPP, ITK, Boost.GIL, Vigra, Higra, GrAL, DGTal, OpenCV, CImg, Video++, Generic Graphic Library, Milena and Pylena).
- Domain Specific Languages (DSL) (Eigen, Blaze, Blitz++ or Armadillo via C++ Expression template, or Halide and SYCL with their own toolchain).

Need of Genericity for image processing



Algorithms are intrinsically generic.

Figure 1: Watershed algorithm applied to three different image types. [1]

[1] Levillain et al., Practical genericity: Writing image processing algorithms both reusable and efficient. CIARP, 2014

Need of Genericity for image processing

Algorithm must support combination whose cardinality increases with:

- supported underlying image value type (grayscale, rgb, floating-point, . . .)
- supported data structure (ND-buffers, graphs, meshes, . . .)
- additional data type (structuring element, masks, . . .)

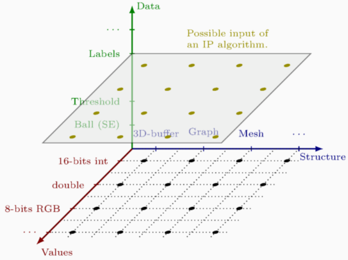


Figure 2: Space of possibilities.

What is a generic algorithm?
An algorithm is generic when it is written once to perform over a wide array of data structures.

Context and history of generic programming

Outline

Non-generic algorithm: gamma correction

Introducing the process of (generic) **Generalization** [2].

Non-generic gamma-correction algorithm:

```
void gamma_correction(Image& ima, double gamma)
{
  const auto gamma_corr = 1.f / gamma;
  for (int x = 0; x < ima.width(); ++x)
    for (int y = 0; y < ima.height(); ++y)
    {
      ima(x, y).r = 256.f * std::pow(ima(x, y).r / 256.f, gamma_corr);
      ima(x, y).g = 256.f * std::pow(ima(x, y).g / 256.f, gamma_corr);
      ima(x, y).b = 256.f * std::pow(ima(x, y).b / 256.f, gamma_corr);
    }
}
```

This algorithm is **over-constrained**. How can we make it **Generic**?

[2] Roynard et al., An image processing library in modern C++: Getting simplicity and efficiency with generic programming. RRPR, 2019

1rst approach: Code duplication

- Writing and optimizing an algorithm for a particular data type in mind.
- Often results in multiple switch/cases to enumerate all the supported combination of supported data types.

```
void gamma_correction(any_image img, double gamma)
{
  switch((img.structure_kind, img.value_kind))
  {
  case (BUFFER2D, UINT8):
    gamma_correction_img2d_uint8( (image2d<uint8>) img, gamma );
  // ...
  case (LUT, RGB8):
    gamma_correction_lut_rgb8( (image_lut<rgb8>) img, gamma );
  // ...
  }
}
```

2nd approach: Generalization

- Necessity to have a common denominator to all the supported types: the super-type.
- All supported data types must be convertible to and from this super-type.
- Good for maintenance but conversions impact performance.

```
struct image4D { // generalized super-type with generalized underlying value-type
  using value_type = std::array<double, 4>; // every value is converted to this one
}; // specific types w/ conversion routines
struct image2D { image4D to(); void from(image4D); };
struct image3D { image4D to(); void from(image4D); };
void gamma_correction(image4D ima, double gamma) {
  for(auto t : ima.time())
    for(auto z : ima.depth())
      for(auto y : ima.width())
        for(auto x : ima.height())
          auto& v = ima(x, y, z, t); // image4D::value_type
          // correct v with gamma
}
```

Toward Genericity: Step 1

Lifting RGB constraint:

```
void gamma_correction(Image& ima, double gamma)
{
  using value_t = typename Image::value_type;

  const auto gamma_corr = 1.f / gamma;
  const auto max_val = std::numeric_limits<value_t>::max();

  for(int x = 0; x < ima.width(); ++x)
    for(int y = 0; y < ima.height(); ++y)
      ima(x, y) = max_val * std::pow(ima(x, y) / max_val, gamma_corr);
}
```

There is no valid reason to constrain this algorithm to RGB images.

Toward Genericity: Step 2

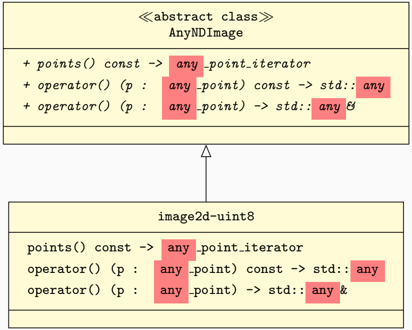
Lifting 2-dimensional constraint:

```
void gamma_correction(Image& ima, double gamma)
{
  using value_t = typename Image::value_type;

  const auto gamma_corr = 1.f / gamma;
  const auto max_val = std::numeric_limits<value_t>::max();

  for (auto&& pix : ima.pixels())
    pix.val() = max_val * std::pow(pix.val() / max_val, gamma_corr);
}
```

There is no valid reason to constrain this algorithm to 2D-images.

3rd approach: Inclusion Polymorphism

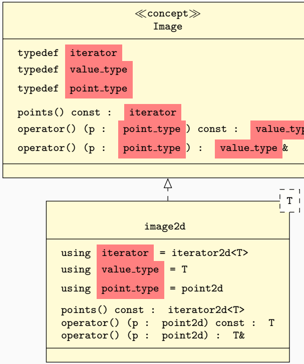
- Extracting behavior pattern from algorithms
- Grouping them into logical bricks called **interfaces**.
- Each algorithm can require a set of behavioral pattern to be satisfied.
- All happens at runtime (dynamic dispatch).




```
void gamma_correction(
  AnyNDImage& ima,
  double gamma)
{
  for (auto pnt : ima.points())
    auto& v = ima(pnt);
    // correct v with gamma
}
```

4th approach: Parametric Polymorphism

- Extracting behavior pattern from algorithms
- Grouping them into logical bricks called **concepts**.
- Each algorithm can require a set of behavioral pattern to be satisfied.
- All happens at compile-time (static dispatch).




```cpp
template <class InImage>
void gamma_correction(
  InImage& ima,
  double gamma)
{
  for (auto pnt : ima.points())
    auto& v = ima(pnt);
    // correct v with gamma
}
```

Genericity within Libraries

All existing library do not fall into one single category and combine those techniques to achieve diverse degree of genericity.

- CImg mixes *Generalization* and *Parametric polymorphism* by considering only 4D-images parametrized by their value type.
- OpenCV's algorithms take polymorphic input types (Inclusion polymorphism) but dispatch on the value type on specialized algorithm (code duplication) that then re-dispatch on generic routines (parametric polymorphism).
- Scikit-image relies on Scipy which uses dynamic abstraction (inclusion polymorphism) for its nd-array and sometimes dispatch on specialized routine (code duplication) for performance.
- Many other libraries have chosen to leverage Parametric polymorphism at diverse degree (Boost.GIL, Higra, Vigra, GrAL, DGTal, Milena and Pylene)

Genericity: Summary

Paradigm	TC[1]	CS[2]	E[3]	One IA[4]	EA[5]
Code Duplication	✓	✗	✓	✗	✗
Code Generalization	✗	≈	≈	✓	✗
Inclusion Polymorphism	≈	✓	✗	✓	✓
Parametric Polymorphism:					
with C++11	✓	≈	✓	✓	≈
with C++17	✓	✓	✓	✓	≈
with C++20	✓	✓	✓	✓	✓

[1] TC: type checking.

[2] CS: code simplicity.

[3] E: efficiency.

[4] One IA: one implementation per algorithm.

[4] EA: explicit abstractions / constrained genericity.

Toward Genericity: Constraints so far

```
void gamma_correction(Image& ima, double gamma)
{
  using value_t = typename Image::value_type;

  const auto gamma_corr = 1.f / gamma;
  const auto max_val = std::numeric_limits<value_t>::max();

  for (auto&& pix : ima.pixels())
    pix.val() = max_val * std::pow(pix.val() / max_val, gamma_corr);
}
```

Prerequisite for the algorithm

- Image type provides subtype `value_type`.
- Image type provides a member function `pixels()` for traversing.
- Underlying image's value type has a maximum bound.
- Underlying image's value type behaves properly with `pow`.

Toward constrained Genericity: Old techniques (pre-C++20)

Before C++20 (2020) we relied on metaprogramming techniques to achieve constrained Genericity:

- Functions on types: metafunctions or type-traits
- SFINAE: Substitution Failure Is Not An Error
- CRTP: Curiously Recurring Template Pattern (used in the SCOOP paradigm [3] in Milena).

We do not want this anymore!

[3] Géraud et al., Semantics-driven genericity: A sequel to the static C++ object-oriented programming paradigm (SCOOP 2). MPOOL, 2008

Final step: Conceptification 1/2

C++ introduce the syntax for Concepts:

```
template <class I>
concept Image = requires {
    typename I::value_type; // type of value
    typename I::point_type; // type of points
    typename I::pixel_type; // type of pixels
  } &&
  Pixel<I::pixel_type> &&
  requires (I ima) {
    { ima.pixels() } -> mdrange<I::pixel_type>;
  };
template <Image I>
concept WritableImage =
  WritablePixel<I::pixel_type> &&
  requires (I ima) {
    { ima.pixels() } -> mdrange<I::pixel_type>;
  };
```

```
template <class P>
concept Pixel = requires {
    typename P::value_type; // type of value
    typename P::point_type; // type of points
  } && requires (P pix) {
    { pix.val() } -> P::value_type;
    { pix.pnt() } -> P::point_type;
  };
template <Pixel P>
concept WritablePixel =
  requires (P pix, P::value_type v) {
    { pix.val() = v };
  };
```

Final step: Conceptification 2/2

```
template <WritableImage Image>
  requires requires(Image::value_type v, double d) {
    { std::pow(v, d) } -> Image::value_type;
  }
void gamma_correction(Image& ima, double gamma)
{
  using value_t = typename Image::value_type;

  const auto gamma_corr = 1.f / gamma;
  const auto max_val = std::numeric_limits<value_t>::max();

  for (auto&& pix : ima.pixels())
    pix.val() = max_val * std::pow(pix.val() / max_val, gamma_corr);
}
```

A Concept is a set of behavioral requirements (constraints) applied to a type.

This is the final (most generalized) version of the algorithm.

Generic programming for image processing in the static world (contribution)

Outline

Taxonomy for Image Processing: Goal

Taxonomy definition

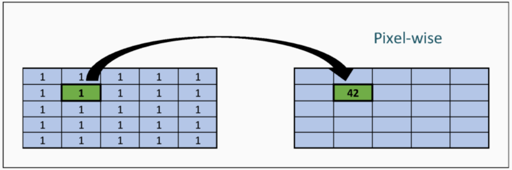
Act of doing a classification of something in a systematic way.

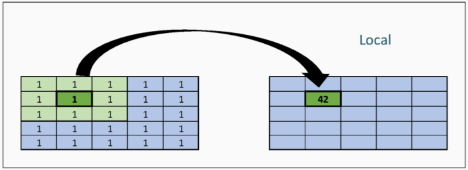
- Provide zero-costs abstractions so that algorithm families can work on image type families.
- Code will be easy and efficient **by default**.

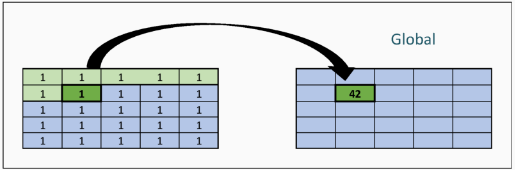
Concepts are not designed after data structures but after algorithms.

Taxonomy of Image Processing algorithms: three families

- Pixel-wise algorithms: thresholding, gamma correction.
- Local algorithms: dilation, erosion, closing, hit-or-miss, gradient, rank filter, etc.
- Global algorithms: Chamfer distance transform, labeling, union-find, max-tree, etc.







From those families emerge several concepts.

Our Concepts for Image Processing: fundamentals

Fundamental concepts are necessary to be able to do basic manipulations over an image.

- Value, Point, Pixels: represent the trivial building blocks of an image.
- Domain: represent the set of points valid for a given image (definition domain).
- Image: represent the algebraic relation $y = f(x)$ where y is a value generated by the image f for the input (point) x.
- An image can also store a value, as in $f(x) = y$.

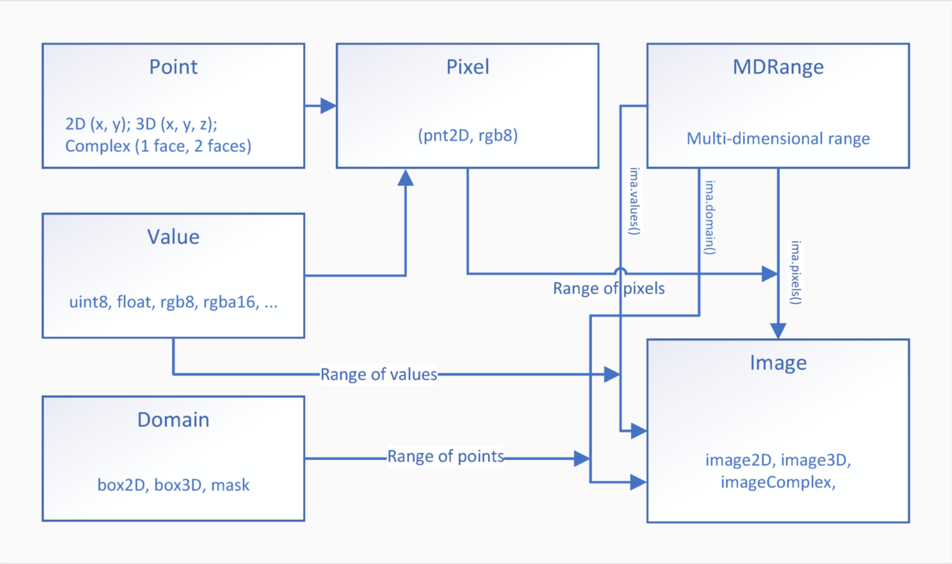
Concepts interact with each other's through composition and refinement (close to inheritance).

Our Concepts for Image Processing: fundamentals

The basic use case of a pixel-wise algorithm on an image is illustrated by the following code:

```
for (auto pnt : ima.domain()) // traverse image
  ima(pnt) = 42;              // Set the image's value at pnt

// OR

for (auto pix : ima.pixels()) // 0 cost abstraction
  pix.val() = 42;             // Set the pixel's value
  pix.val() = 42;             // Set the pixel's value
```

Our Concepts for Image Processing: Image concept



Our Concepts for Image Processing: Advanced Image concept

- IndexableImage: access an image via indexes. Needed for sorting pixels.

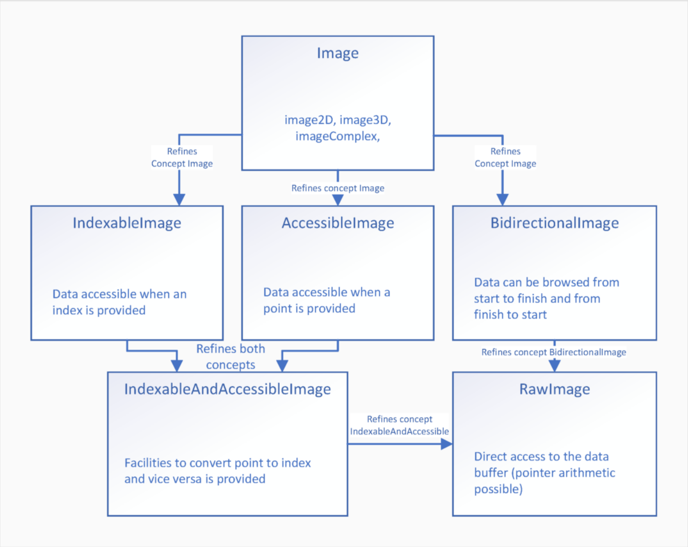
```
auto v = ima[k];                                    // Access value from an index
```

- AccessibleImage: accessing image's value through points (seen above)
- BidirectionalImage: traversing image forward and backward (such as Chamfer distance transform.

```
for(auto pix : ima.pixels())                        // Forward pass
  pix.val() = /* ... */;
for(auto pix : reversed(ima.pixels()))              // Backward pass
  pix.val() = /* ... */;
```

- RawImage: direct interface for accessing the image's data buffer. Interface for experts.

Our Concepts for Image Processing: Advanced Image concept



+ **Writability**

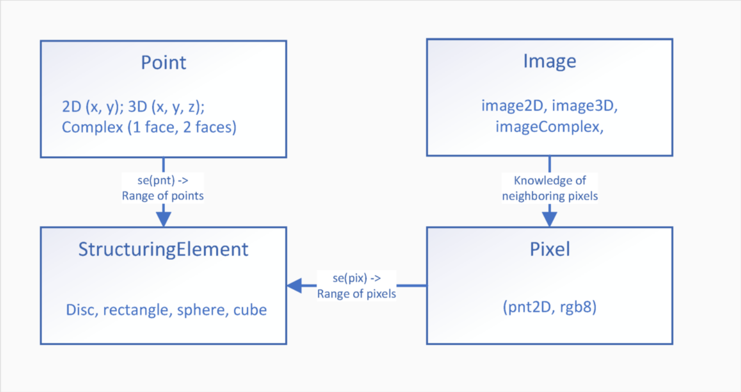
Our Concepts for Image Processing: Local algorithms

- Support for structuring elements (disc, rectangle, sphere, cube, etc.)
- Support retrieving neighboring pixels via the structuring element.

Typical usage in local algorithms:

```
auto se = se::disc(.radius=3); // get a structuring element
for(auto pnt : ima.domain())   // traverse image's points
  for(auto nbp : se(pnt))      // traverse neighboring points
    // nbp is a range of points

// OR

for(auto pix : ima.pixels())   // 0 cost abstraction
  for(auto nbx : se(pix))      // traverse neighboring pixels
    // nbx is a range of pixels
```

Our Concepts for Image Processing: Local algorithms



0-overhead for designed abstraction

- We designed abstraction around those concepts that enables 0-overhead when they are used properly.
- For the majority of cases the default code will be the most efficient.
- It is possible to dispatch (dynamically of statically) to algorithms specialization for the remaining edge cases for maximum efficiency.

```
template <Image Img, StructuringElement SE>
auto dilate(Img img, SE se) {
  if (se.is_decomposable()) {
    lst_small_se = se.decompose();
    for (auto small_se : lst_small_se)
      // Recursive call
      img = dilate(img, small_se);
    return img;
  }
  else if (is_pediodic_line(se))
      // Van Herk's algorithm
    return fast_dilate1d(img, se);
  else
      // Classic algorithm
    return dilate_normal(img, se);
}
```

Taxonomy for Image Processing: Canvas for Dilation & Erosion

Dilation and Erosion have the same shape:

```cpp
template <Image In, StructuringElement SE,
          WritableImage Out>
void dilate(const In& ima, const SE& se,
            Out& out){
  for(auto pnt : ima.points()) {
    out(pnt) = ima(pnt);
    for(auto nbx : se(pnt))
      out(pnt) = max(out(pnt), img(nx));
  }
}
```

```cpp
template <Image In, StructuringElement SE,
          WritableImage Out>
void erode(const In& ima, const SE& se,
            Out& out){
  for(auto pnt : ima.points()) {
    out(pnt) = ima(pnt);
    for(auto nbx : se(pnt))
      out(pnt) = min(out(pnt), img(nx));
  }
}
```

They can be rewritten in a common canvas:

```cpp
template <Image In, StructuringElement SE,
          WritableImage Out, class Op>
void local_op(const In& ima, const SE& se,
            Out& out, Op&& op){
  for(auto pnt : ima.points()) {
    out(pnt) = ima(pnt);
    for(auto nbx : se(pnt))
      out(pnt) = op(out(pnt), img(nx));
  }
}
```

```cpp
template <Image In, StructuringElement SE, WritableImage Out>
void dilate(const In& ima, const SE& se, Out& out){
  local_op(img, se, max, out);
}
template <Image In, StructuringElement SE, WritableImage Out>
void erode(const In& ima, const SE& se, Out& out){
  local_op(img, se, min, out);
}
```

Taxonomy for Image Processing: Algorithms over-generalization

- Algorithms can have the same computational shape.
- When this shape is known, an algorithm canvas can be written (meta-algorithm)
- This canvas shifts the algorithm implementation into a descriptive paradigm.
- This canvas allows reusing code easily.
- This canvas allows the maintainer to factorize optimizations (GPU offloading, etc.)

Views for image processing (contribution)

Outline

Views for Image Processing

View definition

A view ==is an *image*== which embarks an algorithm and features properties. They are inspired from Milena's morphers [4] and STL ranges [5]. We published our take in [6].

Views can replace pixel-wise algorithms.

[4] Levillain et al., Milena: Write generic morphological algorithms once, run on many kinds of images. ISMM 2009.

[5] Niebler et al., Ranges for the standard library: Revision 1. ISO WG21, 2014.

[6] Roynard et al., A modern C++ point of View of programming in image processing. ACM SIGPLAN GPCE, 2022.

View properties

Views feature interesting properties.

They are:

View properties

- Cheap-to-copy (lightweight),
- Non-owning (shared semantic of the data buffer),
- Lazy evaluated,
- Composable.

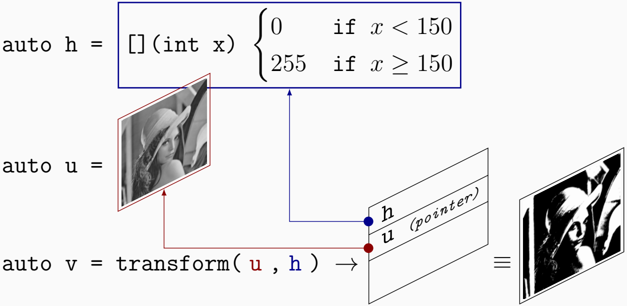


Figure 3: An image view performing a thresholding.

Raising abstraction level

Views enable the Image Processing practitioner to rewrite the following alpha-blending algorithm at image level.

```
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);
  }
}
```

```
// ROI overload
void blend_inplace(
  const uint8_t* ima1,
  uint8_t* ima2,
  float alpha, int width,
  int height, int stride1,
  int stride2,
  int width_b, int width_e
  int height_b, int height_e);
```


ima

$\leftarrow 0.2 \times$


ima1

$+ 0.8 \times$


ima2

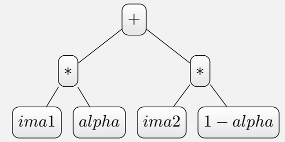
Views for Image Processing: Raising abstraction level

Composable:

```
auto roi = /* ... */;
auto blend_roi = alphablend(view::clip(ima1, roi), view::clip(ima2, roi), 0.2);
auto blend_red = alphablend(view::red(ima1), view::red(ima2), 0.2);
```

Lazy-evaluated, Non-owning

```
auto alphablend =
[](auto ima1, auto ima2, float alpha) {
  return alpha * ima1 +
                (1 - alpha) * ima2;
};
```



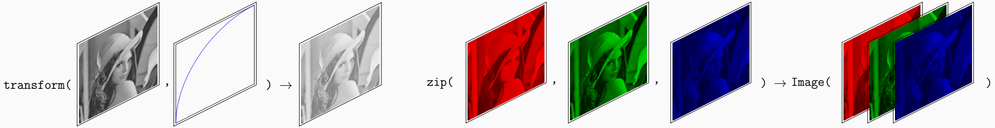
Cheap-to-copy:

```
auto ima1, ima2 = /* ... */;
auto ima_blended = alphablend(ima1, ima2, 0.2);
```

Overview of basic views

Domain-restrictive views: filter, clip, mask

Value-transforming view: transform, zip, cast, $+$, $\times$, etc.



Property preservation (of concepts) through views

View type	Property / Expression	Forward	Bidirectional	Raw	Writable	Accessible	Indexable
Image	ima, ima1, ima2	✓	✓	✓	✓	✓	✓
Cast	cast<T>(ima)	✓	✓	✗	✗	✓	✓
Transform	transform(ima, func)	✓	✓	✗	✓[1]	✓	✓
Filter	filter(ima, pred)	✓	✓	✗	✓	✓	✓
Clip	clip(ima, dom)	✓	✓	✗	✓	✓	✓
Mask	mask(ima, mask)	✓	✓	✗	✓	✓	✓
Zip	zip(ima1, ima2)	✓	✓	✗	✓	✓	✓
Channel	red(ima)	✓	✓	✗	✓	✓	✓
Arithmetic	ima1 + ima2	✓	✓	✗	✗	✓	✓
Logical	ima > 125	✓	✓	✗	✗[2]	✓	✓
Mathematical	abs(ima)	✓	✓	✗	✗	✓	✓

[1]: writability is preserved only if `func` is a projection.

[2]: writability not preserved except for the expression `ifelse(ima, ima1, ima2)`.

Concrete application of Views: Background subtraction

Background | Candidate | Result

Figure 4: Background detection: data set samples.

Background subtraction pipeline with algorithms

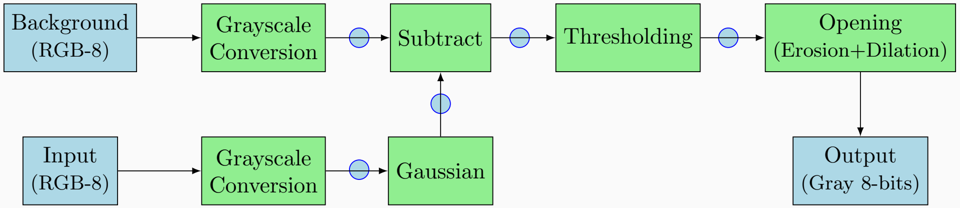


Figure 5: Background subtraction pipeline using algorithms. Superfluous copies are shown with ◯

5 intermediary images in memory.

Background subtraction pipeline with views

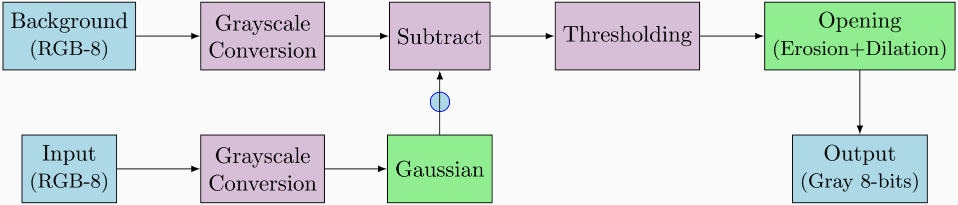


Figure 6: Background subtraction pipeline using algorithms and views.

1 intermediary image in memory.

Background subtraction concrete code

```
float kThreshold = 150; float kVSigma = 10;
float kHSigma = 10;  int kOpeningRadius = 32;
auto img_gray = view::transform(img_color, to_gray);
auto bg_gray  = view::transform(bg_color, to_gray);
auto bg_blurred = gaussian2d(bg_gray,  kHSigma, kVSigma);
auto tmp_gray = img_gray - bg_blurred;
auto thresholdf = [](auto x) { return x < kThreshold; };
auto tmp_bin = view::transform(tmp_gray, thresholdf);
auto ero = erosion(tmp_bin, disc(kOpeningRadius));
dilation(ero, disc(kOpeningRadius), output);
```

Figure 7: Pipeline implementation with *views*. Highlighted code uses *views* by prefixing operators with the namespace `view`.

Background subtraction benchmark result

Framework	Compute Time	Memory usage	ΔMemory usage
Pylene (w/o views)	2.11*s* *(± 144ms)*	106 MB	**+0%**
Pylene (views)	2.13*s* *(± 164ms)*	51 MB	-**52%**
OpenCV	2.41*s* *(± 134ms)*	59 MB	-**44%**

Table 1: Benchmarks of the previous pipeline on a dataset (12 images) of 10MPix images. Average computation time and memory usage of implementations with/without *views* and with OpenCV as a baseline.

Genericity limitations: C++ templates in the dynamic world

- Static templates does not mix well with dynamic code (such as Python).
- Templates belong to the static world (compiled once)
- Python belongs to the dynamic world (interpreted multiple time)
- Specific work is needed to make our library available for Python.

Static-dynamic bridge: a hybrid solution

Hybrid solution

- Provide type-erased interface to the static-world.
- Dispatch to the most efficient algorithm (precompiled) depending on type information.
- Allow foreign type injection.

Limitations

- Code bloat.
- Foreign type injection is costly in performance.
- Still experimental.

Conclusion & perspectives

Outline

Conclusion

- We have shown the progress in generic programming in C++20 and how it serves our purpose.
- We have designed a Concept framework for Image Processing leveraging these progresses to write an *easy to use*, *efficient* and *generic* Image processing library: **Pylene**.
- We have show-case a new abstraction layer: *Image Views*, that are fully-fledged image type embedding Image Processing algorithm.
- We have explored way to provide those tools to the dynamic world via a static-dynamic bridge, which will be researched further in the future.

Limitations & Perspective

Limitations

- Is C++ the best choice (slow to evolve, inertia, tooling, state of package management, difficult language to learn compared to others).
- Evaluate the gain in usage simplicity is not trivial (metrics, studies with people experiencing writing code).

Perspectives

- JIT compilation for the static-dynamic bridge via Cython, AutoWIG/Cppyy, Xeus-cling or Pythran.
- Improve the Concept framework to encompass more use-cases for other algorithm families from other domains than image processing.
- Research on heterogeneous computation and how to integrate it in our Pylene library.

Publications i

- Roynard, M., Carlinet, E., & Géraud, T. (2019). An image processing library in modern C++: Getting simplicity and efficiency with generic programming. In B. Kerautret, M. Colom, D. Lopresti, P. Monasse, & H. Talbot (Eds.), *Reproducible research in pattern recognition* (pp. 121–137). Springer International Publishing
- Roynard, M., Carlinet, E., & Géraud, T. (2022). A modern C++ point of *View* of programming in image processing. *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '22), December 06–07, 2022, Auckland, New Zealand*. https://doi.org/10.1145/3564719.3568692

Thanks for listening!

Any question?

Generic programming in modern C++ for Image Processing

Michaël Roynard Director: *Thierry Géraud* Supervisor: *Edwin Carlinet*

Thesis Defense — Friday 4th November, 2022

EPITA Research Laboratory (LRE) — Le Kremlin-Bicêtre, France

References i

Géraud, T., & Levillain, R. (2008). Semantics-driven genericity: A sequel to the static C++ object-oriented programming paradigm (SCOOP 2). *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL)*.

Levillain, R., Géraud, T., & Najman, L. (2009). Milena: Write generic morphological algorithms once, run on many kinds of images. In M. H. F. Wilkinson & J. B. T. M. Roerdink (Eds.), *Mathematical morphology and its application to signal and image processing — proceedings of the ninth international symposium on mathematical morphology (ismm)* (pp. 295–306, Vol. 5720). Springer Berlin / Heidelberg.

References ii

Levillain, R., Géraud, T., Najman, L., & Carlinet, E. (2014). Practical genericity: Writing image processing algorithms both reusable and efficient. In E. Bayro & E. Hancock (Eds.), *Progress in pattern recognition, image analysis, computer vision, and applications — proceedings of the 19th iberoamerican congress on pattern recognition (ciarp)* (pp. 70–79, Vol. 8827). Springer-Verlag.

Roynard, M., Carlinet, E., & Géraud, T. (2019). An image processing library in modern C++: Getting simplicity and efficiency with generic programming. In B. Kerautret, M. Colom, D. Lopresti, P. Monasse, & H. Talbot (Eds.), *Reproducible research in pattern recognition* (pp. 121–137). Springer International Publishing. (Cit. on p. 58).

References iii

Roynard, M., Carlinet, E., & Géraud, T. (2022). A modern C++ point of *View* of programming in image processing. *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '22), December 06–07, 2022, Auckland, New Zealand*. https://doi.org/10.1145/3564719.3568692 (cit. on p. 58).

Eric Niebler, A. S., Sean Parent. (2014). Ranges for the standard library: Revision 1. https://ericniebler.github.io/std/wg21/D4128.html

Efficient way to traverse an image

Introducing segmented ranges (cf. issues std::mdspan)

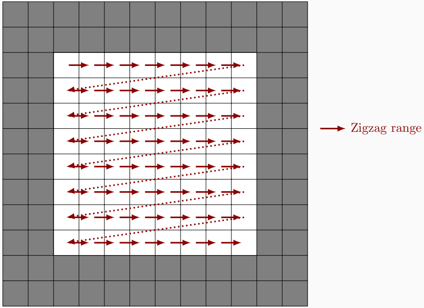


Figure 8: Range-v3's ranges

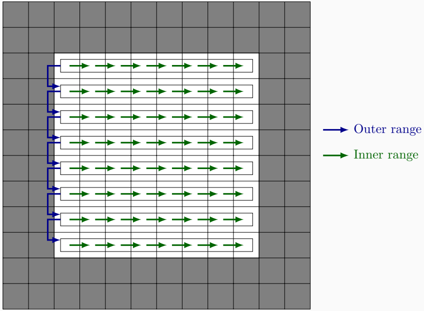


Figure 9: Segmented ranges

Efficient way to traverse an image

Compiler needs explicit contiguous dimension to generate vectorized instructions.

Milena's algorithm (unvectorized):

```
template <class I, class O, class SE>
void dilate(const Image<I>& input_, Image<O>& output_, const SE& se)
{
    const I& input = exact(input_);
    O& output = exact(output_);
    mln_piter(I)  p(input.domain());
    mln_qiter(SE) n(se, p);
    for_all(p)
    {
        mln_value(I) M = f(p);
        for_all(q)
            M = max(f(q), M);
        output(p) = M;
    }
}
```

Efficient way to traverse an image

Compiler needs explicit contiguous dimension to generate vectorized instructions.

Pylene's v1, unoptimized algorithm:

```
template<Image I, OutputImage O, class SE>
  requires (StructuringElement<SE,
                image_point_t<I>>)
void dilate(I input, O output, SE se) {
    mln_foreach(auto p, input.domain()) {
        auto M = input(p);
        for(auto n : se(p))
            M = std::max(M, f(n));
        output(p) = M;
    }
}
```

Pylene's v2, optimized algorithm using pixels (segmented ranges under the hood):

```
template<Image I, OutputImage O, class SE>
  requires (StructuringElement<SE,
                image_point_t<I>>)
void dilate(I input, O output, SE se) {
    auto z = view::zip(input.pixels(),
                  output.pixels());
    mln_foreach((auto [pxin, pxout]), z) {
        auto M = pxin.val();
        for(auto nx : se(pxin))
            M = std::max(M, nx.val());
        pxout.val() = M;
    }
}
```

Concept formal definition

	Definition	Description	Requirement
Image	`Ima::const_pixel_range`	type of the range to iterate over all the constant pixels	models the concept *ForwardRange*
	`Ima::pixel_type`	type of a pixel	models the concept *Pixel*
	`Ima::value_type`	type of a value	models the concept *Regular*
Writable Image	`WIma::pixel_range`	type of the range to iterate over all the non-constant pixels	models the concept *ForwardRange*

Table 2: Concepts formalization: definitions

	Expression	Return Type	Description
Image	`cima.pixels()`	`Ima::const_pixel_range`	returns a range of constant pixels to iterate over it
Writable Image	`wima.pixels()`	`WIma::pixel_range`	returns a range of pixels to iterate over it

Table 3: Concepts formalization: expressions

Pylene's views vs. Milena's morphers vs. STL ranges' views

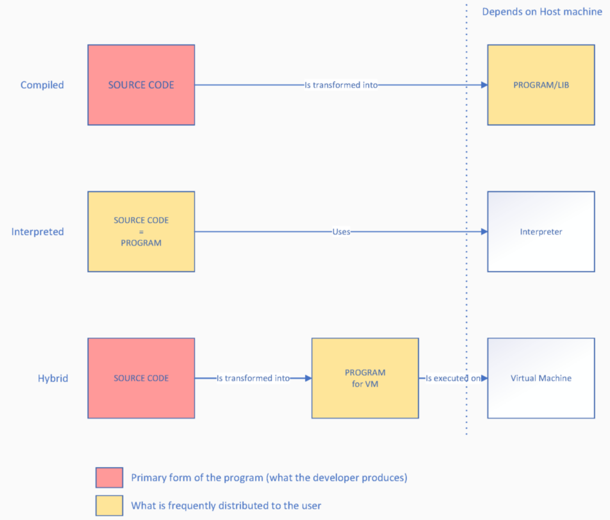
Differences with STL ranges' views

- STL's views are distinct from STL's containers and are constructed from ranges.
- Ranges are constructed from a container (and their iterators)
- Views is 3 levels of abstraction above a container.

Differences with Milena's morphers

- Milena's morphers were not cheap to copy.
- Constructed from an image's reference otherwise need of deep-copy.
- As base image was taken by reference, possibility of dangling pointers.
- Dangle happens when chaining transformations with pipe (temporary copies).
- Const semantic (const values vs. writable values) ill-defined.

Static-dynamic bridge: languages type



Static information vs. dynamic information

Static information (known at compile-time)

- Image's value type (unit8, rgb8, complex, etc.),
- Image's dimension size (1D, 2D, 3D, etc.),
- Architecture of the hardware hosting the program (x86, ARM, PowerPC, GPU, etc.).

Dynamic information (known at runtime)

- Image's actual values,
- Image's actual size,
- Architecture of the hardware hosting the program (x86, ARM, PowerPC, GPU, etc.).